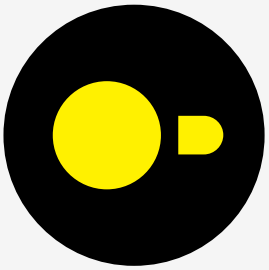
DuckDB Documentation

DuckDB version 1.2.3-dev
Generated on 2025-07-03 at 14:03 UTC



Contents

Contents i

Summary 1

Connect 5

Connect 7
- Connect or Create a Database . . . 7
- Persistence . . . 7
  - Persistent Database . . . 7
  - In-Memory Database . . . 7

Concurrency 9
- Handling Concurrency . . . 9
- Concurrency within a Single Process . . . 9
- Writing to DuckDB from Multiple Processes . . . 9
- Optimistic Concurrency Control . . . 9

Data Import 11

Importing Data 13
- `INSERT` Statements . . . 13
- File Loading: Relative Paths . . . 13
- File Formats . . . 13
  - CSV Loading . . . 13
  - Parquet Loading . . . 14
  - JSON Loading . . . 14
  - Returning the Filename . . . 14
- Appender . . . 14

Data Sources 15

CSV Files 17
- CSV Import . . . 17
- Examples . . . 17
- CSV Loading . . . 17
- Parameters . . . 18
  - `auto_type_candidates` Details . . . 20
- CSV Functions . . . 20
- Writing Using the COPY Statement . . . 21
- Reading Faulty CSV Files . . . 22
- Order Preservation . . . 22
- Writing CSV Files . . . 22
- CSV Auto Detection . . . 22

Sample Size . . . 22
`sniff_csv` Function . . . 22
Prompt . . . 23
Detection Steps . . . 23
Dialect Detection . . . 23
Type Detection . . . 24
Header Detection . . . 25
Dates and Timestamps . . . 25
Reading Faulty CSV Files . . . 26
Structural Errors . . . 26
Anatomy of a CSV Error . . . 27
Using the `ignore_errors` Option . . . 28
Retrieving Faulty CSV Lines . . . 28
Reject Scans . . . 29
Reject Errors . . . 29
Parameters . . . 29
CSV Import Tips . . . 30
Override the Header Flag if the Header Is Not Correctly Detected . . . 31
Provide Names if the File Does Not Contain a Header . . . 31
Override the Types of Specific Columns . . . 31
Use COPY When Loading Data into a Table . . . 31
Use `union_by_name` When Loading Files with Different Schemas . . . 31
Sample Size . . . 31

JSON Files **33**
JSON Overview . . . 33
About JSON . . . 33
JSONPath and JSON Pointer Syntax . . . 33
Indexing . . . 33
Examples . . . 33
Loading JSON . . . 33
Writing JSON . . . 34
JSON Data Type . . . 34
Retrieving JSON Data . . . 34
Using Quotes for Special Characters . . . 35
Creating JSON . . . 35
JSON Creation Functions . . . 35
Loading JSON . . . 36
The `read_json` Function . . . 36
Functions for Reading JSON Objects . . . 37
Parameters . . . 37
Functions for Reading JSON as a Table . . . 39
Parameters . . . 39
Loading with the COPY Statement Using `FORMAT json` . . . 41
Parameters . . . 42
Writing JSON . . . 43
JSON Type . . . 43
JSON Processing Functions . . . 43
JSON Extraction Functions . . . 43
JSON Scalar Functions . . . 45
JSON Aggregate Functions . . . 47
Transforming JSON to Nested Types . . . 48
JSON Table Functions . . . 49

JSON Format Settings . . . 50
Records Settings . . . 52
Installing and Loading the JSON extension . . . 53
SQL to/from JSON . . . 53
Examples . . . 53
Caveats . . . 55
Equality Comparison . . . 55

Multiple Files **57**
Reading Multiple Files . . . 57
CSV . . . 57
Parquet . . . 57
Multi-File Reads and Globs . . . 58
List Parameter . . . 58
Glob Syntax . . . 58
List of Globs . . . 58
Filename . . . 58
Glob Function to Find Filenames . . . 59
Combining Schemas . . . 59
Examples . . . 59
Combining Schemas . . . 59
Union by Position . . . 60
Union by Name . . . 60

Parquet Files **63**
Reading and Writing Parquet Files . . . 63
Examples . . . 63
Parquet Files . . . 64
`read_parquet` Function . . . 64
Parameters . . . 65
Partial Reading . . . 65
Inserts and Views . . . 65
Writing to Parquet Files . . . 66
Encryption . . . 67
Supported Features . . . 67
Installing and Loading the Parquet Extension . . . 67
Querying Parquet Metadata . . . 68
Parquet Metadata . . . 68
Parquet Schema . . . 69
Parquet File Metadata . . . 69
Parquet Key-Value Metadata . . . 70
Bloom Filters . . . 70
Parquet Encryption . . . 70
Reading and Writing Encrypted Files . . . 70
Writing Encrypted Parquet Files . . . 71
Reading Encrypted Parquet Files . . . 71
Limitations . . . 71
Performance Implications . . . 71
Parquet Tips . . . 71
Tips for Reading Parquet Files . . . 71
Use `union_by_name` When Loading Files with Different Schemas . . . 71
Tips for Writing Parquet Files . . . 72
Enabling `PER_THREAD_OUTPUT` . . . 72
Selecting a `ROW_GROUP_SIZE` . . . 72

The ROW_GROUPS_PER_FILE Option . . . 72

Partitioning **73**
Hive Partitioning . . . 73
Examples . . . 73
Hive Partitioning . . . 73
Filter Pushdown . . . 74
Autodetection . . . 74
Hive Types . . . 74
Writing Partitioned Files . . . 74
Partitioned Writes . . . 74
Examples . . . 74
Partitioned Writes . . . 75
Filename Pattern . . . 75
Overwriting . . . 76
Appending . . . 76
Handling Slashes in Columns . . . 76

Appender **77**
Date, Time and Timestamps . . . 77
Commit Frequency . . . 78
Handling Constraint Violations . . . 78
Appender Support in Other Clients . . . 78

INSERT Statements **79**
Syntax . . . 79

Client APIs 81

Client Overview **83**
Support Tiers . . . 83
Compatibility . . . 84

ADBC Client **85**
Implemented Functionality . . . 85
Database . . . 85
Connection . . . 85
Statement . . . 87
Examples . . . 88
C++ . . . 88
Python . . . 89
Go . . . 89

C **95**
Overview . . . 95
Installation . . . 95
Startup & Shutdown . . . 95
Example . . . 95
API Reference Overview . . . 96
Configuration . . . 101
Example . . . 101
API Reference Overview . . . 101
Query . . . 104
Example . . . 104

Value Extraction . . . 104
duckdb_fetch_chunk . . . 104
duckdb_value . . . 105
API Reference Overview . . . 106
Data Chunks . . . 111
API Reference Overview . . . 111
Vectors . . . 114
Vector Format . . . 114
Primitive Types . . . 114
NULL Values . . . 115
Strings . . . 115
Decimals . . . 116
Enums . . . 116
Structs . . . 116
Lists . . . 116
Arrays . . . 117
Examples . . . 117
Example: Reading an int64 Vector with NULL Values . . . 117
Example: Reading a String Vector . . . 118
Example: Reading a Struct Vector . . . 118
Example: Reading a List Vector . . . 119
API Reference Overview . . . 121
Validity Mask Functions . . . 121
Values . . . 128
API Reference Overview . . . 128
Types . . . 153
Functions . . . 154
duckdb_value . . . 154
duckdb_fetch_chunk . . . 154
API Reference Overview . . . 154
Date Time Timestamp Helpers . . . 155
Hugeint Helpers . . . 155
Decimal Helpers . . . 155
Logical Type Interface . . . 155
Prepared Statements . . . 173
Example . . . 173
API Reference Overview . . . 174
Appender . . . 177
Example . . . 177
API Reference Overview . . . 177
Table Functions . . . 188
API Reference Overview . . . 188
Table Function Bind . . . 188
Table Function Init . . . 189
Table Function . . . 189
Replacement Scans . . . 199
API Reference Overview . . . 200
Complete API . . . 201
API Reference Overview . . . 201
Open Connect . . . 201
Configuration . . . 202
Query Execution . . . 202
Result Functions . . . 202

Safe Fetch Functions . . . 202
Helpers . . . 203
Date Time Timestamp Helpers . . . 203
Hugeint Helpers . . . 203
Unsigned Hugeint Helpers . . . 203
Decimal Helpers . . . 203
Prepared Statements . . . 204
Bind Values to Prepared Statements . . . 204
Execute Prepared Statements . . . 205
Extract Statements . . . 205
Pending Result Interface . . . 205
Value Interface . . . 205
Logical Type Interface . . . 207
Data Chunk Interface . . . 207
Vector Interface . . . 207
Validity Mask Functions . . . 208
Scalar Functions . . . 208
Selection Vector Interface . . . 209
Aggregate Functions . . . 209
Table Functions . . . 209
Table Function Bind . . . 210
Table Function Init . . . 210
Table Function . . . 210
Replacement Scans . . . 210
Profiling Info . . . 210
Appender . . . 211
Table Description . . . 211
Arrow Interface . . . 212
Threading Information . . . 212
Streaming Result Interface . . . 212
Cast Functions . . . 212

C++ API **349**
Installation . . . 349
Basic API Usage . . . 349
Startup & Shutdown . . . 349
Querying . . . 349
UDF API . . . 350

CLI **355**
CLI Client . . . 355
Installation . . . 355
Getting Started . . . 355
Usage . . . 355
Options . . . 355
In-Memory vs. Persistent Database . . . 355
Running SQL Statements in the CLI . . . 356
Editor Features . . . 356
Exiting the CLI . . . 356
Dot Commands . . . 356
Configuring the CLI . . . 358
Setting a Custom Prompt . . . 358
Non-Interactive Usage . . . 359
Loading Extensions . . . 359

Reading from stdin and Writing to stdout . . . 359
Reading Environment Variables . . . 359
Examples . . . 360
Restrictions for Reading Environment Variables . . . 360
Prepared Statements . . . 360
Command Line Arguments . . . 360
Passing a Sequence of Arguments . . . 361
Dot Commands . . . 361
List of Dot Commands . . . 362
Using the `.help` Command . . . 364
`.output`: Writing Results to a File . . . 364
Querying the Database Schema . . . 365
Syntax Highlighters . . . 366
Configuring the Query Syntax Highlighter . . . 366
Configuring the Result Syntax Highlighter . . . 366
Shorthands . . . 367
Importing Data from CSV . . . 367
Output Formats . . . 367
List of Output Formats . . . 367
Changing the Output Format . . . 368
`duckbox` Mode . . . 369
Editing . . . 369
Moving . . . 370
History . . . 370
Changing Text . . . 370
Completing . . . 371
Miscellaneous . . . 371
Using Read-Line . . . 371
Safe Mode . . . 372
Autocomplete . . . 372
Syntax Highlighting . . . 373
Error Highlighting . . . 374

Dart Client **375**
Installation . . . 375
Use This Package as a Library . . . 375
Usage Examples . . . 375
Querying an In-Memory Database . . . 375
Queries on Background Isolates . . . 376

Go Client **377**
Installation . . . 377
Importing . . . 377
Appender . . . 377
Examples . . . 378
Simple Example . . . 378
More Examples . . . 379

Java JDBC Client **381**
Installation . . . 381
Basic API Usage . . . 381
Startup & Shutdown . . . 381
Configuring Connections . . . 382
Querying . . . 382

Arrow Methods . . . 383
Streaming Results . . . 384
Appender . . . 384
Batch Writer . . . 384
Troubleshooting . . . 385
Driver Class Not Found . . . 385

Julia Client **387**
Installation . . . 387
Basics . . . 387
Scanning DataFrames . . . 387
Appender API . . . 388
Concurrency . . . 388
Original Julia Connector . . . 389

Node.js (Deprecated) **391**
Node.js API . . . 391
Initializing . . . 391
Running a Query . . . 391
Connections . . . 392
Prepared Statements . . . 392
Inserting Data via Arrow . . . 393
Loading Unsigned Extensions . . . 393
Node.js API . . . 394
Modules . . . 394
Typedefs . . . 394
duckdb . . . 394
duckdb~Connection . . . 395
duckdb~Statement . . . 399
duckdb~QueryResult . . . 401
duckdb~Database . . . 401
duckdb~TokenType . . . 407
duckdb~ERROR : number . . . 407
duckdb~OPEN_READONLY : number . . . 407
duckdb~OPEN_READWRITE : number . . . 407
duckdb~OPEN_CREATE : number . . . 408
duckdb~OPEN_FULLMUTEX : number . . . 408
duckdb~OPEN_SHAREDCACHE : number . . . 408
duckdb~OPEN_PRIVATECACHE : number . . . 408
ColumnInfo : object . . . 408
TypeInfo : object . . . 408
DuckDbError : object . . . 409
HTTPError : object . . . 409

Node.js (Neo) **411**
Node.js Client (Neo) . . . 411
Features . . . 411
Main Differences from duckdb-node . . . 411
Roadmap . . . 411
Supported Platforms . . . 411
Examples . . . 412
Get Basic Information . . . 412
Connect . . . 412
Create Instance . . . 412

Instance Cache . . . 412
Connect to Instance . . . 412
Disconnect . . . 413
Run SQL . . . 413
Parameterize SQL . . . 413
Specifying Values . . . 414
Stream Results . . . 414
Inspect Result Metadata . . . 414
Read Result Data . . . 414
Get Result Data . . . 415
Convert Result Data . . . 415
Fetch Chunks . . . 420
Inspect Data Types . . . 421
Inspect Data Values . . . 421
Displaying Timezones . . . 424
Append To Table . . . 424
Append Data Chunk . . . 424
Extract Statements . . . 425
Control Evaluation of Tasks . . . 425
Ways to Run SQL . . . 425
Ways to Get Result Data . . . 427

ODBC **429**

ODBC API Overview . . . 429
DuckDB ODBC Driver . . . 429
ODBC API on Linux . . . 429
Driver Manager . . . 429
Setting Up the Driver . . . 429
ODBC API on Windows . . . 430
DSN Windows Setup . . . 431
Default DuckDB DSN . . . 431
Changing DuckDB DSN . . . 432
More Detailed Windows Setup . . . 433
Registry Keys . . . 433
Updating the ODBC Driver . . . 434
ODBC API on macOS . . . 434
ODBC Configuration . . . 435
`odbc.ini` and `.odbc.ini` . . . 435
`odbcinst.ini` and `.odbcinst.ini` . . . 435

Python **437**

Python API . . . 437
Installation . . . 437
Basic API Usage . . . 437
Data Input . . . 437
DataFrames . . . 438
Result Conversion . . . 439
Writing Data to Disk . . . 439
Connection Options . . . 439
Using an In-Memory Database . . . 439
Persistent Storage . . . 439
Configuration . . . 440
Connection Object and Module . . . 440
Using Connections in Parallel Python Programs . . . 440

Loading and Installing Extensions . . . 440
Community Extensions . . . 440
Unsigned Extensions . . . 441
Data Ingestion . . . 441
CSV Files . . . 441
Parquet Files . . . 441
JSON Files . . . 442
Directly Accessing DataFrames and Arrow Objects . . . 442
Pandas DataFrames – `object` Columns . . . 443
Registering Objects . . . 443
Conversion between DuckDB and Python . . . 443
Object Conversion: Python Object to DuckDB . . . 443
`int` . . . 444
`float` . . . 444
`datetime.datetime` . . . 444
`datetime.time` . . . 444
`datetime.date` . . . 444
`bytes` . . . 444
`list` . . . 444
`dict` . . . 445
`tuple` . . . 447
`numpy.ndarray` and `numpy.datetime64` . . . 447
Result Conversion: DuckDB Results to Python . . . 447
NumPy . . . 447
Pandas . . . 447
Apache Arrow . . . 447
Polars . . . 447
Examples . . . 448
Python DB API . . . 448
Connection . . . 448
In-Memory Connection . . . 448
Default Connection . . . 449
File-Based Connection . . . 449
Querying . . . 450
Prepared Statements . . . 450
Named Parameters . . . 451
Relational API . . . 451
Lazy Evaluation . . . 451
Relation Creation . . . 452
Relation Definition Details . . . 469
Transformation . . . 478
Functions . . . 494
Output . . . 549
Python Function API . . . 573
Creating Functions . . . 574
Using Partial Functions . . . 574
Type Annotation . . . 576
NULL Handling . . . 576
Exception Handling . . . 577
Side Effects . . . 577
Python Function Types . . . 578
Arrow . . . 578
Native . . . 578

Types API . . . 578
Converting from Other Types . . . 579
Python Built-Ins . . . 579
Numpy DTypes . . . 579
Nested Types . . . 579
Creation Functions . . . 580
Expression API . . . 582
Why Would I Use the Expression API? . . . 582
Column Expression . . . 582
Star Expression . . . 583
Constant Expression . . . 583
Case Expression . . . 584
Function Expression . . . 584
SQL Expression . . . 585
Common Operations . . . 586
Order Operations . . . 586
Spark API . . . 586
Example . . . 586
Contribution Guidelines . . . 587
Python Client API . . . 587
Known Python Issues . . . 587
Troubleshooting . . . 587
Running `EXPLAIN` Renders Newlines . . . 587
Crashes and Errors on Windos . . . 588
Known Issues . . . 588
Numpy Import Multithreading . . . 588
`DESCRIBE` and `SUMMARIZE` Return Empty Tables in Jupyter . . . 588
Protobuf Error for JupySQL in IPython . . . 589

R Client **591**
Installation . . . 591
`duckdb`: R Client . . . 591
`duckplyr`: dplyr Client . . . 591
Reference Manual . . . 591
Basic Client Usage . . . 591
Startup & Shutdown . . . 591
Querying . . . 592
Efficient Transfer . . . 592
dbplyr . . . 593
Memory Limit . . . 593
Troubleshooting . . . 593
Warning When Installing on macOS . . . 593

Rust Client **595**
Installation . . . 595
Basic API Usage . . . 595
Startup & Shutdown . . . 595
Querying . . . 595
Appender . . . 596

Swift Client **597**
Instantiating DuckDB . . . 597
Application Example . . . 597
Creating an Application-Specific Type . . . 597

Loading a CSV File . . . 597
Querying the Database . . . 598
Complete Project . . . 599

Wasm **601**
DuckDB Wasm . . . 601
Getting Started with DuckDB-Wasm . . . 601
Limitations . . . 601
Instantiation . . . 601
`cdn(jsdelivr)` . . . 601
`webpack` . . . 602
`vite` . . . 602
Statically Served . . . 603
Data Ingestion . . . 603
Data Import . . . 603
Open & Close Connection . . . 603
Apache Arrow . . . 603
CSV . . . 604
JSON . . . 604
Parquet . . . 605
httpfs (Wasm-Flavored) . . . 605
Insert Statement . . . 606
Query . . . 606
Query Execution . . . 606
Prepared Statements . . . 606
Arrow Table to JSON . . . 607
Export Parquet . . . 607
Extensions . . . 607
Format . . . 607
`INSTALL` and `LOAD` . . . 608
Autoloading . . . 608
List of Officially Available Extensions . . . 608
HTTPFS . . . 608
Extension Signing . . . 609
Fetching DuckDB-Wasm Extensions . . . 609
Serving Extensions from a Third-Party Repository . . . 609
Tooling . . . 609

SQL **611**

SQL Introduction **613**
Concepts . . . 613
Creating a New Table . . . 613
Populating a Table with Rows . . . 614
Querying a Table . . . 614
Joins between Tables . . . 616
Aggregate Functions . . . 617
Updates . . . 619
Deletions . . . 619

Statements **621**
Statements Overview . . . 621
ANALYZE Statement . . . 621

Usage . . . 621
ALTER TABLE Statement . . . 621
Examples . . . 621
Syntax . . . 622
RENAME TABLE . . . 622
RENAME COLUMN . . . 622
ADD COLUMN . . . 623
DROP COLUMN . . . 623
[SET [DATA]] TYPE . . . 623
Handling Structs . . . 624
SET / DROP DEFAULT . . . 624
ADD PRIMARY KEY . . . 624
ADD / DROP CONSTRAINT . . . 625
Limitations . . . 625
ALTER VIEW Statement . . . 625
Examples . . . 625
ATTACH and DETACH Statements . . . 625
Examples . . . 625
ATTACH . . . 626
Attach Syntax . . . 626
Explicit Storage Versions . . . 627
DETACH . . . 627
Detach Syntax . . . 627
Options . . . 627
Name Qualification . . . 627
Default Database and Schema . . . 628
Changing the Default Database and Schema . . . 628
Resolving Conflicts . . . 628
Changing the Catalog Search Path . . . 629
Transactional Semantics . . . 629
CALL Statement . . . 629
Examples . . . 629
Syntax . . . 630
CHECKPOINT Statement . . . 630
Examples . . . 630
Syntax . . . 630
Behavior . . . 630
Reclaiming Space . . . 630
COMMENT ON Statement . . . 630
Examples . . . 631
Reading Comments . . . 631
Limitations . . . 632
Syntax . . . 632
COPY Statement . . . 632
Examples . . . 632
Overview . . . 633
COPY ... FROM . . . 633
Syntax . . . 633
COPY ... TO . . . 634
COPY ... TO Options . . . 635
Syntax . . . 636
COPY FROM DATABASE ... TO . . . 636
Syntax . . . 636

Format-Specific Options . . . 636
CSV Options . . . 636
Parquet Options . . . 637
JSON Options . . . 638
Limitations . . . 639
CREATE MACRO Statement . . . 639
Examples . . . 639
Scalar Macros . . . 639
Table Macros . . . 640
Overloading . . . 641
Syntax . . . 641
Limitations . . . 642
Using Named Parameters . . . 642
Using Subquery Macros . . . 643
Overloads . . . 643
Recursive Functions . . . 643
CREATE SCHEMA Statement . . . 643
Examples . . . 643
Syntax . . . 644
CREATE SECRET Statement . . . 644
Syntax for `CREATE SECRET` . . . 644
Syntax for `DROP SECRET` . . . 644
CREATE SEQUENCE Statement . . . 644
Examples . . . 644
Creating and Dropping Sequences . . . 644
Using Sequences for Primary Keys . . . 645
Selecting the Next Value . . . 645
Selecting the Current Value . . . 645
Syntax . . . 646
Parameters . . . 646
Limitations . . . 647
CREATE TABLE Statement . . . 647
Examples . . . 647
Temporary Tables . . . 648
`CREATE OR REPLACE` . . . 648
`IF NOT EXISTS` . . . 648
`CREATE TABLE ... AS SELECT` (CTAS) . . . 649
Copying the Schema . . . 649
Check Constraints . . . 649
Foreign Key Constraints . . . 650
Limitations . . . 651
Generated Columns . . . 651
Syntax . . . 651
CREATE VIEW Statement . . . 651
Examples . . . 651
Syntax . . . 652
CREATE TYPE Statement . . . 652
Examples . . . 652
Syntax . . . 652
Limitations . . . 652
DELETE Statement . . . 653
Examples . . . 653
`USING` Clause . . . 653

RETURNING Clause . . . 653
Syntax . . . 653
TheTRUNCATE Statement . . . 653
Limitations on Reclaiming Memory and Disk Space . . . 653
DESCRIBE Statement . . . 653
Usage . . . 654
Alias . . . 654
See Also . . . 654
DROP Statement . . . 654
Examples . . . 654
Syntax . . . 655
Dependencies of Dropped Objects . . . 655
Limitations . . . 655
Dependencies on Views . . . 655
Limitations on Reclaiming Disk Space . . . 656
EXPORT and IMPORT DATABASE Statements . . . 656
Examples . . . 656
EXPORT DATABASE . . . 656
Syntax . . . 657
IMPORT DATABASE . . . 657
Syntax . . . 657
INSERT Statement . . . 657
Examples . . . 657
Syntax . . . 657
Insert Column Order . . . 658
INSERT INTO ... [BY POSITION] . . . 658
INSERT INTO ... BY NAME . . . 658
ON CONFLICT Clause . . . 659
DO NOTHING Clause . . . 659
DO UPDATE Clause (Upsert) . . . 659
Defining a Conflict Target . . . 661
RETURNING Clause . . . 662
LOAD / INSTALL Statements . . . 663
INSTALL . . . 663
Examples . . . 663
Syntax . . . 663
LOAD . . . 663
Examples . . . 663
Syntax . . . 664
PIVOT Statement . . . 664
Simplified PIVOT Syntax . . . 664
Example Data . . . 664
PIVOT ON and USING . . . 665
PIVOT ON, USING, and GROUP BY . . . 665
IN Filter for ON Clause . . . 666
Multiple Expressions per Clause . . . 666
Using PIVOT within a SELECT Statement . . . 667
Multiple PIVOT Statements . . . 668
Simplified PIVOT Full Syntax Diagram . . . 668
SQL Standard PIVOT Syntax . . . 668
Examples . . . 669
SQL Standard PIVOT Full Syntax Diagram . . . 669
Limitations . . . 670

Profiling Queries . . . 670
EXPLAIN . . . 670
EXPLAIN ANALYZE . . . 670
SELECT Statement . . . 670
Examples . . . 671
Syntax . . . 671
SELECT Clause . . . 671
FROM Clause . . . 672
SAMPLE Clause . . . 672
WHERE Clause . . . 672
GROUP BY and HAVING Clauses . . . 672
WINDOW Clause . . . 672
QUALIFY Clause . . . 672
ORDER BY, LIMIT and OFFSET Clauses . . . 672
VALUES List . . . 672
Row IDs . . . 673
Common Table Expressions . . . 673
Full Syntax Diagram . . . 673
SET and RESET Statements . . . 673
Examples . . . 674
Syntax . . . 674
RESET . . . 674
Scopes . . . 674
Configuration . . . 674
SET VARIABLE and RESET VARIABLE Statements . . . 674
SET VARIABLE . . . 675
Syntax . . . 676
RESET VARIABLE . . . 676
Syntax . . . 676
SUMMARIZE Statement . . . 676
Usage . . . 676
See Also . . . 676
Transaction Management . . . 677
Statements . . . 677
Starting a Transaction . . . 677
Committing a Transaction . . . 677
Rolling Back a Transaction . . . 677
Example . . . 677
UNPIVOT Statement . . . 678
Simplified UNPIVOT Syntax . . . 678
Example Data . . . 678
UNPIVOT Manually . . . 678
UNPIVOT Dynamically Using Columns Expression . . . 679
UNPIVOT into Multiple Value Columns . . . 680
Using UNPIVOT within a SELECT Statement . . . 680
Expressions within UNPIVOT Statements . . . 681
Simplified UNPIVOT Full Syntax Diagram . . . 681
SQL Standard UNPIVOT Syntax . . . 681
SQL Standard UNPIVOT Manually . . . 681
SQL Standard UNPIVOT Dynamically Using the COLUMNS Expression . . . 682
SQL Standard UNPIVOT into Multiple Value Columns . . . 682
SQL Standard UNPIVOT Full Syntax Diagram . . . 683
UPDATE Statement . . . 683

Examples . . . 683
Syntax . . . 683
Update from Other Table . . . 683
Update from Same Table . . . 684
Update Using Joins . . . 684
Upsert (Insert or Update) . . . 685
USE Statement . . . 685
Examples . . . 685
Syntax . . . 685
VACUUM Statement . . . 685
Examples . . . 686
Reclaiming Space . . . 686
Syntax . . . 686

Query Syntax **687**
SELECT Clause . . . 687
Examples . . . 687
Syntax . . . 688
SELECT List . . . 688
Star Expressions . . . 688
DISTINCT Clause . . . 688
DISTINCT ON Clause . . . 688
Aggregates . . . 689
Window Functions . . . 689
unnest Function . . . 689
FROM and JOIN Clauses . . . 689
Examples . . . 690
Joins . . . 691
Outer Joins . . . 691
Cross Product Joins (Cartesian Product) . . . 691
Conditional Joins . . . 691
Natural Joins . . . 692
Semi and Anti Joins . . . 693
Lateral Joins . . . 694
Positional Joins . . . 695
As-Of Joins . . . 696
Self-Joins . . . 696
Shorthands in the JOIN Clause . . . 697
FROM-First Syntax . . . 697
FROM-First Syntax with a SELECT Clause . . . 697
FROM-First Syntax without a SELECT Clause . . . 698
Time Travel Using AT . . . 698
Syntax . . . 698
WHERE Clause . . . 698
Examples . . . 698
Syntax . . . 699
GROUP BY Clause . . . 699
GROUP BY ALL . . . 699
Multiple Dimensions . . . 699
Examples . . . 699
GROUP BY ALL Examples . . . 700
Syntax . . . 700
GROUPING SETS . . . 700
Examples . . . 700

Description . . . 700
Identifying Grouping Sets with GROUPING_ID() . . . 702
Syntax . . . 704
HAVING Clause . . . 704
Examples . . . 704
Syntax . . . 704
ORDER BY Clause . . . 704
ORDER BY ALL . . . 705
NULL Order Modifier . . . 705
Collations . . . 705
Examples . . . 705
ORDER BY ALL Examples . . . 706
Syntax . . . 706
LIMIT and OFFSET Clauses . . . 706
Examples . . . 707
Syntax . . . 707
SAMPLE Clause . . . 707
Examples . . . 707
Syntax . . . 708
Unnesting . . . 708
Examples . . . 708
Unnesting Lists . . . 708
Unnesting Structs . . . 709
Recursive Unnest . . . 709
Setting the Maximum Depth of Unnesting . . . 709
Keeping Track of List Entry Positions . . . 710
WITH Clause . . . 710
Basic CTE Examples . . . 710
CTE Materialization . . . 711
Recursive CTEs . . . 712
Example: Fibonacci Sequence . . . 712
Example: Tree Traversal . . . 712
Graph Traversal . . . 713
Recursive CTEs with USING KEY . . . 716
Example: USING KEY . . . 716
Using VALUES . . . 716
Example: USING KEY References Union Table . . . 717
Limitations . . . 717
Syntax . . . 718
WINDOW Clause . . . 718
Syntax . . . 718
QUALIFY Clause . . . 718
Examples . . . 718
Syntax . . . 719
VALUES Clause . . . 719
Examples . . . 719
Syntax . . . 720
FILTER Clause . . . 720
Examples . . . 720
Aggregate Function Syntax (Including FILTER Clause) . . . 722
Set Operations . . . 722
UNION . . . 722
Vanilla UNION (Set Semantics) . . . 723

UNION ALL (Bag Semantics) . . . 723
UNION [ALL] BY NAME . . . 723
INTERSECT . . . 724
Vanilla INTERSECT (Set Semantics) . . . 724
INTERSECT ALL (Bag Semantics) . . . 724
EXCEPT . . . 724
Vanilla EXCEPT (Set Semantics) . . . 725
EXCEPT ALL (Bag Semantics) . . . 725
Syntax . . . 725
Prepared Statements . . . 725
Syntax . . . 725
Example Data Set . . . 725
Auto-Incremented Parameters: ? . . . 726
Positional Parameters: $1 . . . 726
Named Parameters: $parameter . . . 726
Dropping Prepared Statements: DEALLOCATE . . . 726

Data Types **727**
Data Types . . . 727
General-Purpose Data Types . . . 727
Nested / Composite Types . . . 728
Rules for Case Sensitivity . . . 728
Updating Values of Nested Types . . . 729
Nesting . . . 729
Performance Implications . . . 729
Array Type . . . 729
Creating Arrays . . . 729
Defining an Array Field . . . 730
Retrieving Values from Arrays . . . 730
Functions . . . 730
Examples . . . 730
Ordering . . . 731
See Also . . . 731
Bitstring Type . . . 731
Creating a Bitstring . . . 731
Functions . . . 732
Blob Type . . . 732
Functions . . . 732
Boolean Type . . . 732
Conjunctions . . . 733
Expressions . . . 733
Date Types . . . 733
Special Values . . . 734
Functions . . . 734
Enum Data Type . . . 734
Enum Definition . . . 735
Examples . . . 735
Enum Usage . . . 735
Enums vs. Strings . . . 736
Enum Removal . . . 737
Comparison of Enums . . . 737
Functions . . . 738
Interval Type . . . 738
Arithmetic with Timestamps, Dates and Intervals . . . 739

Equality and Comparison . . . 740
Functions . . . 740
List Type . . . 740
Creating Lists . . . 740
Retrieving from Lists . . . 741
Comparison and Ordering . . . 741
Functions . . . 741
Literal Types . . . 742
Null Literals . . . 742
Integer Literals . . . 742
Other Numeric Literals . . . 742
Underscores in Numeric Literals . . . 742
String Literals . . . 743
Implicit String Literal Concatenation . . . 743
Implicit String Conversion . . . 743
Escape String Literals . . . 744
Dollar-Quoted String Literals . . . 744
Map Type . . . 745
Creating Maps . . . 745
Retrieving from Maps . . . 745
Comparison Operators . . . 746
Functions . . . 746
NULL Values . . . 746
NULL and Functions . . . 747
NULL and AND / OR . . . 747
NULL and IN / NOT IN . . . 747
NULL and Aggregate Functions . . . 747
Numeric Types . . . 748
Integer Types . . . 748
Variable Integer . . . 748
Fixed-Point Decimals . . . 748
Floating-Point Types . . . 749
Universally Unique Identifiers (UUIDs) . . . 750
UUIDv4 . . . 750
UUIDv7 . . . 750
Functions . . . 750
Struct Data Type . . . 751
Creating Structs . . . 751
Adding Field(s)/Value(s) to Structs . . . 751
Retrieving from Structs . . . 752
Dot Notation Order of Operations . . . 752
Creating Structs with the row Function . . . 753
Comparison and Ordering . . . 754
Updating the Schema . . . 754
Adding a Field . . . 755
Dropping a Field . . . 755
Renaming a Field . . . 755
Functions . . . 755
Text Types . . . 755
Specifying a Length Limit . . . 756
Specifying a Compression Type . . . 756
Text Type Values . . . 756
Strings with Special Characters . . . 756

Functions . . . 757
Time Types . . . 757
Timestamp Types . . . 757
Timestamp Types . . . 757
Conversion between Strings and Naïve / Time Zone-Aware Timestamps . . . 758
Special Values . . . 759
Functions . . . 759
Time Zones . . . 759
Instants . . . 760
Temporal Binning . . . 760
Time Zone Support . . . 761
Calendar Support . . . 762
Settings . . . 762
Time Zone Reference List . . . 762
Union Type . . . 779
Example . . . 779
Union Casts . . . 780
Casting to Unions . . . 781
Casting between Unions . . . 781
Comparison and Sorting . . . 781
Functions . . . 781
Typecasting . . . 781
Explicit Casting . . . 781
Implicit Casting . . . 782
Combination Casting . . . 782
Casting Operations Matrix . . . 782
Lossy Casts . . . 784
Overflows . . . 784
Varchar . . . 784
Literal Types . . . 784
Lists / Arrays . . . 784
Arrays . . . 784
Structs . . . 785
Unions . . . 785

Expressions **787**
Expressions . . . 787
CASE Expression . . . 787
Casting . . . 788
Explicit Casting . . . 788
Casting Rules . . . 789
TRY_CAST . . . 789
`cast_to_type` Function . . . 789
Collations . . . 790
Using Collations . . . 790
Default Collations . . . 791
ICU Collations . . . 792
Comparisons . . . 792
Comparison Operators . . . 792
Combination Casting . . . 793
`BETWEEN` and `IS [NOT] NULL` . . . 793
IN Operator . . . 793
`IN (val1, val2, ...)` (Tuple) . . . 793
`IN [val1, val2, ...]` (List) . . . 794

IN Map . . . 794
IN Subquery . . . 794
NOT IN . . . 794
Logical Operators . . . 794
Binary Operators: AND and OR . . . 795
Unary Operator: NOT . . . 795
Star Expression . . . 795
Syntax . . . 795
TABLE.* and STRUCT.* . . . 795
EXCLUDE Clause . . . 796
REPLACE Clause . . . 796
RENAME Clause . . . 796
Column Filtering via Pattern Matching Operators . . . 796
COLUMNS Expression . . . 796
COLUMNS Expression in a WHERE Clause . . . 797
Regular Expressions in a COLUMNS Expression . . . 797
Renaming Columns with Regular Expressions in a COLUMNS Expression . . . 798
COLUMNS Lambda Function . . . 798
COLUMNS List . . . 798
*COLUMNS Unpacked Columns . . . 799
STRUCT.* . . . 799
Subqueries . . . 800
Scalar Subquery . . . 800
Grades . . . 800
Subquery Comparisons: ALL, ANY and SOME . . . 801
ALL . . . 801
ANY . . . 801
EXISTS . . . 802
NOT EXISTS . . . 802
IN Operator . . . 802
Correlated Subqueries . . . 803
Returning Each Row of the Subquery as a Struct . . . 803
TRY expression . . . 804
Examples . . . 804
Casting . . . 804
Logarithm on Zero . . . 804
Casting Multiple Rows . . . 805
Limitations . . . 805

Functions **807**
Functions . . . 807
Function Syntax . . . 807
Function Chaining via the Dot Operator . . . 807
Using with Literals and Arrays . . . 807
Limitations . . . 807
Query Functions . . . 807
Aggregate Functions . . . 808
Examples . . . 808
Syntax . . . 809
DISTINCT Clause in Aggregate Functions . . . 809
ORDER BY Clause in Aggregate Functions . . . 809
Handling NULL Values . . . 810
General Aggregate Functions . . . 810
Approximate Aggregates . . . 818

Statistical Aggregates . . . 819
Ordered Set Aggregate Functions . . . 825
Miscellaneous Aggregate Functions . . . 825
Array Functions . . . 825
Array-Native Functions . . . 825
Bitstring Functions . . . 828
Bitstring Operators . . . 828
Bitstring Functions . . . 829
Bitstring Aggregate Functions . . . 831
Blob Functions . . . 832
Date Format Functions . . . 836
`strftime` Examples . . . 836
`strptime` Examples . . . 837
CSV Parsing . . . 837
Format Specifiers . . . 837
Date Functions . . . 838
Date Operators . . . 839
Date Functions . . . 839
Date Part Extraction Functions . . . 844
Date Part Functions . . . 844
Part Specifiers Usable as Date Part Specifiers and in Intervals . . . 844
Part Specifiers Only Usable as Date Part Specifiers . . . 845
Part Functions . . . 846
Enum Functions . . . 851
Interval Functions . . . 852
Interval Operators . . . 852
Interval Functions . . . 852
Lambda Functions . . . 856
Scalar Functions That Accept Lambda Functions . . . 856
`list_transform(list, lambda(x))` . . . 857
`list_filter(list, lambda(x))` . . . 857
`list_reduce(list, lambda(x, y)[, initial_value]` . . . 857
Nesting Lambda Functions . . . 857
Scoping . . . 858
Indexes as Parameters . . . 858
Examples . . . 858
`list_transform` Examples . . . 858
`list_filter` Examples . . . 859
`list_reduce` Examples . . . 859
Limitations . . . 860
List Functions . . . 860
List Operators . . . 869
List Comprehension . . . 870
Range Functions . . . 870
`range` . . . 871
`generate_series` . . . 871
Date Ranges . . . 872
Slicing . . . 872
List Aggregates . . . 873
`list_*` Rewrite Functions . . . 874
Sorting Lists . . . 874
Flattening . . . 875
Lambda Functions . . . 876

Related Functions . . . 876
Map Functions . . . 876
Nested Functions . . . 880
Numeric Functions . . . 880
Numeric Operators . . . 880
Division and Modulo Operators . . . 880
Supported Types . . . 880
Numeric Functions . . . 881
Pattern Matching . . . 892
`LIKE` . . . 892
`SIMILAR TO` . . . 893
Globbing . . . 893
`GLOB` . . . 893
Glob Function to Find Filenames . . . 894
Globbing Semantics . . . 894
Regular Expressions . . . 895
Regular Expressions . . . 895
Regular Expression Syntax . . . 895
Functions . . . 895
Options for Regular Expression Functions . . . 898
Using `regexp_matches` . . . 898
Using `regexp_replace` . . . 898
Using `regexp_extract` . . . 899
Limitations . . . 899
Struct Functions . . . 899
Text Functions . . . 901
Text Functions and Operators . . . 901
Text Similarity Functions . . . 926
Formatters . . . 928
`fmt` Syntax . . . 928
`printf` Syntax . . . 929
Time Functions . . . 931
Time Operators . . . 931
Time Functions . . . 931
Timestamp Functions . . . 933
Timestamp Operators . . . 934
Scalar Timestamp Functions . . . 934
Timestamp Table Functions . . . 942
Timestamp with Time Zone Functions . . . 943
Built-In Timestamp with Time Zone Functions . . . 943
Timestamp with Time Zone Strings . . . 945
ICU Timestamp with Time Zone Operators . . . 945
ICU Timestamp with Time Zone Functions . . . 946
ICU Timestamp Table Functions . . . 951
ICU Timestamp Without Time Zone Functions . . . 952
At Time Zone . . . 954
Infinities . . . 954
Calendars . . . 954
Union Functions . . . 954
Utility Functions . . . 955
Scalar Utility Functions . . . 955
Utility Table Functions . . . 965
Window Functions . . . 966

Examples . . . 966
Syntax . . . 966
General-Purpose Window Functions . . . 966
Aggregate Window Functions . . . 970
DISTINCT Arguments . . . 970
ORDER BY Arguments . . . 970
Nulls . . . 971
Evaluation . . . 971
Partition and Ordering . . . 972
Framing . . . 974
`WINDOW` Clauses . . . 976
Filtering the Results of Window Functions Using `QUALIFY` . . . 977
Box and Whisker Queries . . . 977

Constraints **979**
Syntax . . . 979
Check Constraint . . . 979
Not Null Constraint . . . 979
Primary Key and Unique Constraint . . . 979
Foreign Keys . . . 980

Indexes **981**
Index Types . . . 981
Min-Max Index (Zonemap) . . . 981
Adaptive Radix Tree (ART) . . . 981
Indexes Defined by Extensions . . . 981
Persistence . . . 981
`CREATE INDEX` and `DROP INDEX` Statements . . . 981
Limitations of ART Indexes . . . 981
Constraint Checking in `UPDATE` Statements . . . 982
Over-Eager Constraint Checking in Foreign Keys . . . 982

Meta Queries **983**
Information Schema . . . 983
Tables . . . 983
`character_sets`: Character Sets . . . 983
`columns`: Columns . . . 983
`constraint_column_usage`: Constraint Column Usage . . . 984
`key_column_usage`: Key Column Usage . . . 985
`referential_constraints`: Referential Constraints . . . 985
`schemata`: Database, Catalog and Schema . . . 986
`tables`: Tables and Views . . . 986
`table_constraints`: Table Constraints . . . 987
Catalog Functions . . . 988
DuckDB_% Metadata Functions . . . 988
`duckdb_columns` . . . 989
`duckdb_constraints` . . . 989
`duckdb_databases` . . . 990
`duckdb_dependencies` . . . 991
`duckdb_extensions` . . . 991
`duckdb_functions` . . . 992
`duckdb_indexes` . . . 992
`duckdb_keywords` . . . 993
`duckdb_log_contexts` . . . 993

duckdb_logs . . . 994
duckdb_memory . . . 994
duckdb_optimizers . . . 994
duckdb_prepared_statements . . . 995
duckdb_schemas . . . 995
duckdb_secret_types . . . 995
duckdb_secrets . . . 996
duckdb_sequences . . . 996
duckdb_settings . . . 997
duckdb_tables . . . 997
duckdb_temporary_files . . . 998
duckdb_types . . . 998
duckdb_variables . . . 999
duckdb_views . . . 999

DuckDB's SQL Dialect **1001**
Overview . . . 1001
Indexing . . . 1001
Examples . . . 1001
Friendly SQL . . . 1001
Clauses . . . 1002
Query Features . . . 1002
Literals and Identifiers . . . 1003
Data Types . . . 1003
Data Import . . . 1003
Functions and Expressions . . . 1003
Join Types . . . 1003
Trailing Commas . . . 1003
”Top-N in Group” Queries . . . 1004
Related Blog Posts . . . 1004
Keywords and Identifiers . . . 1005
Identifiers . . . 1005
Deduplicating Identifiers . . . 1005
Database Names . . . 1005
Rules for Case-Sensitivity . . . 1005
Keywords and Function Names . . . 1005
Case-Sensitivity of Identifiers . . . 1006
Case-Sensitivity of Keys in Nested Data Structures . . . 1006
Order Preservation . . . 1007
Example . . . 1007
Clauses . . . 1008
Insertion Order . . . 1008
PostgreSQL Compatibility . . . 1008
Floating-Point Arithmetic . . . 1008
Division on Integers . . . 1009
UNION of Boolean and Integer Values . . . 1009
Implicit Casting on Equality Checks . . . 1010
Case Sensitivity for Quoted Identifiers . . . 1010
Using Double Equality Sign for Comparison . . . 1011
Vacuuming Tables . . . 1011
Strings . . . 1011
Functions . . . 1011
regexp_extract Function . . . 1011
to_date Function . . . 1012

Resolution of Type Names in the Schema . . . 1012
Exploiting Functional Dependencies for GROUP BY . . . 1012
Behavior of Regular Expression Match Operators . . . 1012
SQL Quirks . . . 1013
Aggregating Empty Groups . . . 1013
0 vs. 1-Based Indexing . . . 1013
Types . . . 1013
UINT8 vs. INT8 . . . 1013
Expressions . . . 1013
Results That May Surprise You . . . 1013
NaN Values . . . 1014
age Function . . . 1014
Extract Functions . . . 1014
Clauses . . . 1014
Automatic Column Deduplication in SELECT . . . 1014
Case Insensitivity for SELECTing Columns . . . 1015
USING SAMPLE . . . 1015

Samples **1017**
Examples . . . 1017
Syntax . . . 1018
reservoir . . . 1018
bernoulli . . . 1018
system . . . 1019
Table Samples . . . 1019

Configuration 1021

Configuration **1023**
Examples . . . 1023
Secrets Manager . . . 1024
Configuration Reference . . . 1024
Global Configuration Options . . . 1024
Local Configuration Options . . . 1028

Pragmas **1033**
Metadata . . . 1033
Resource Management . . . 1035
Collations . . . 1035
Default Ordering for NULLs . . . 1035
Ordering by Non-Integer Literals . . . 1036
Implicit Casting to VARCHAR . . . 1036
Python: Scan All Dataframes . . . 1036
Information on DuckDB . . . 1036
Progress Bar . . . 1037
EXPLAIN Output . . . 1037
Profiling . . . 1037
Enable Profiling . . . 1037
Profiling Format . . . 1037
Profiling Output . . . 1038
Profiling Mode . . . 1038
Custom Metrics . . . 1038
Disable Profiling . . . 1039

Query Optimization . . . 1039
Logging . . . 1039
Full-Text Search Indexes . . . 1039
Verification . . . 1040
Object Cache . . . 1040
Checkpointing . . . 1040
Temp Directory for Spilling Data to Disk . . . 1041
Returning Errors as JSON . . . 1041
IEEE Floating-Point Operation Semantics . . . 1041
Query Verification (for Development) . . . 1041
Block Sizes . . . 1042

Secrets Manager **1043**
Types of Secrets . . . 1043
Creating a Secret . . . 1043
Secret Providers . . . 1043
Temporary Secrets . . . 1044
Persistent Secrets . . . 1044
Deleting Secrets . . . 1044
Creating Multiple Secrets for the Same Service Type . . . 1044
Listing Secrets . . . 1045

Extensions 1047

Extensions **1049**
Listing Extensions . . . 1049
Built-In Extensions . . . 1049
Installing More Extensions . . . 1049
Explicit `INSTALL` and `LOAD` . . . 1050
Autoloading Extensions . . . 1050
Community Extensions . . . 1050
Updating Extensions . . . 1050
Developing Extensions . . . 1051
Working with Extensions . . . 1051

Installing Extensions **1053**
Extension Repositories . . . 1053
Installing Extensions from Different Repositories . . . 1053
Working with Multiple Repositories . . . 1054
Force Installing to Upgrade Extensions . . . 1054
Switching between Repositories . . . 1054
Installing Extensions through Client APIs . . . 1055
Installation Location . . . 1055
Uninstalling Extensions . . . 1055
Sharing Extensions between Clients . . . 1055
Limitations . . . 1055

Advanced Installation Methods **1057**
Downloading Extensions Directly from S3 . . . 1057
Installing an Extension from an Explicit Path . . . 1057
Loading an Extension from an Explicit Path . . . 1057
Building and Installing Extensions from Source . . . 1057
Statically Linking Extensions . . . 1057

Extension Distribution **1059**
Platforms . . . 1059
Extensions Signing . . . 1059
Signed Extensions . . . 1059
Unsigned Extensions . . . 1059
Binary Compatibility . . . 1060
Creating a Custom Repository . . . 1060

Versioning of Extensions **1061**
Extension Versioning . . . 1061
Unstable Extensions . . . 1061
Pre-Release Extensions . . . 1061
Stable Extensions . . . 1062
Release Cycle of Pre-Release and Stable Core Extensions . . . 1062
Nightly Builds . . . 1062
From Stable DuckDB . . . 1062
From Nightly DuckDB . . . 1062
Updating Extensions . . . 1063
Target DuckDB Version . . . 1063
In-Tree vs. Out-of-Tree . . . 1063

Troubleshooting of Extensions **1065**

Core Extensions **1067**

Core Extensions **1069**
List of Core Extensions . . . 1069
Default Extensions . . . 1070

AutoComplete Extension **1071**
Behavior . . . 1071
Functions . . . 1071
Example . . . 1071

Avro Extension **1073**
The `read_avro` Function . . . 1073
File IO . . . 1073
Schema Conversion . . . 1073
Implementation . . . 1074
Limitations and Future Plans . . . 1074

AWS Extension **1075**
Installing and Loading . . . 1075
`config` Provider . . . 1075
`credential_chain` Provider . . . 1075
Auto-Refresh . . . 1076
Legacy Features . . . 1076
Load AWS Credentials (Legacy) . . . 1076

Azure Extension **1079**
Installing and Loading . . . 1079
Usage . . . 1079
Azure Blob Storage . . . 1079
Azure Data Lake Storage (ADLS) . . . 1079
Configuration . . . 1080

Authentication . . . 1081
Authentication with Secret . . . 1081
Authentication with Variables (Deprecated) . . . 1083
Additional Information . . . 1083
Logging . . . 1083
Difference between ADLS and Blob Storage . . . 1084

Delta Extension **1087**
Installing and Loading . . . 1087
Usage . . . 1087
Reading from an S3 Bucket . . . 1087
Reading from Azure Blob Storage . . . 1088
Features . . . 1088
Supported DuckDB Versions and Platforms . . . 1088

DuckLake **1089**
Installing and Loading . . . 1089
Usage . . . 1089
Tables . . . 1089
Functions . . . 1089
`ducklake_snapshots` . . . 1089
`ducklake_table_info` . . . 1090
`ducklake_table_insertions` . . . 1090
`ducklake_table_deletions` . . . 1091
`ducklake_table_changes` . . . 1091
Commands . . . 1092
`ducklake_cleanup_old_files` . . . 1092
`ducklake_expire_snapshots` . . . 1092
`ducklake_merge_adjacent_files` . . . 1092

Encodings Extension **1093**
Installing and Loading . . . 1093
Usage . . . 1093

Excel Extension **1095**
Installing and Loading . . . 1095
Excel Scalar Functions . . . 1095
Examples . . . 1095
Reading XLSX Files . . . 1096
Type and Range Inference . . . 1097
Writing XLSX Files . . . 1097
Type Conversions . . . 1098

Full-Text Search Extension **1099**
Installing and Loading . . . 1099
Usage . . . 1099
`PRAGMA create_fts_index` . . . 1099
`PRAGMA drop_fts_index` . . . 1100
`match_bm25` Function . . . 1100
`stem` Function . . . 1100
Example Usage . . . 1101

httpfs (HTTP and S3) **1103**
httpfs Extension for HTTP and S3 Support . . . 1103
Installation and Loading . . . 1103

HTTP(S) . . . 1103
S3 API . . . 1103
HTTP(S) Support . . . 1103
Partial Reading . . . 1103
Scanning Multiple Files . . . 1104
Authenticating . . . 1104
HTTP Proxy . . . 1104
Using a Custom Certificate File . . . 1105
Hugging Face Support . . . 1105
Usage . . . 1105
Creating a Local Table . . . 1106
Multiple Files . . . 1106
Versioning and Revisions . . . 1106
Authentication . . . 1107
CONFIG . . . 1107
credential_chain . . . 1107
S3 API Support . . . 1107
Platforms . . . 1107
Configuration and Authentication . . . 1108
config Provider . . . 1108
credential_chain Provider . . . 1108
Overview of S3 Secret Parameters . . . 1109
Platform-Specific Secret Types . . . 1110
Reading . . . 1111
Partial Reading . . . 1111
Reading Multiple Files . . . 1111
Globbing . . . 1111
Hive Partitioning . . . 1112
Writing . . . 1112
Configuration . . . 1112
Legacy Authentication Scheme for S3 API . . . 1112
Legacy Authentication Scheme . . . 1113
Per-Request Configuration . . . 1113
Configuration . . . 1113

Iceberg **1115**
Iceberg Extension . . . 1115
Installing and Loading . . . 1115
Updating the Extension . . . 1115
Usage . . . 1115
Common Parameters . . . 1115
Querying Individual Tables . . . 1116
Access Iceberg Metadata . . . 1116
Visualizing Snapshots . . . 1117
Selecting Metadata Versions . . . 1117
Working with Alternative Metadata Naming Conventions . . . 1117
“Guessing” Metadata Versions . . . 1118
Limitations . . . 1118
Iceberg REST Catalogs . . . 1118
ATTACH OPTIONS . . . 1119
Specific Catalog Examples . . . 1119
R2 Catalog . . . 1119
Polaris . . . 1120
Lakekeeper . . . 1120

Limitations . . . 1120
Amazon S3 Tables . . . 1120
Requirements . . . 1121
Connecting to Amazon S3 Tables . . . 1121
Amazon SageMaker Lakehouse (AWS Glue) . . . 1121
Connecting to Amazon SageMaker Lakehouse . . . 1122
Troubleshooting . . . 1122
Limitations . . . 1122
Curl Request Fails . . . 1122
Problem . . . 1122
Solution . . . 1122
HTTP Error 403 . . . 1123
Problem . . . 1123
Solution . . . 1123

ICU Extension **1125**
Installing and Loading . . . 1125
Features . . . 1125

inet Extension **1127**
Installing and Loading . . . 1127
Examples . . . 1127
Operations on `INET` Values . . . 1127
`host` Function . . . 1128
`netmask` Function . . . 1128
`network` Function . . . 1129
`broadcast` Function . . . 1129
<<= Predicate . . . 1129
>>= Predicate . . . 1130
HTML Escape and Unescape Functions . . . 1130

jemalloc Extension **1131**
Operating System Support . . . 1131
Linux . . . 1131
macOS . . . 1131
Windows . . . 1131
Configuration . . . 1131
Environment Variables . . . 1131
Background Threads . . . 1131

MySQL Extension **1133**
Installing and Loading . . . 1133
Reading Data from MySQL . . . 1133
Configuration . . . 1133
Configuring via Secrets . . . 1134
SSL Connections . . . 1134
Reading MySQL Tables . . . 1135
Writing Data to MySQL . . . 1135
Supported Operations . . . 1136
`CREATE TABLE` . . . 1136
`INSERT INTO` . . . 1136
`SELECT` . . . 1136
`COPY` . . . 1136
`UPDATE` . . . 1136

DELETE . . . 1136
ALTER TABLE . . . 1137
DROP TABLE . . . 1137
CREATE VIEW . . . 1137
CREATE SCHEMA and DROP SCHEMA . . . 1137
Transactions . . . 1137
Running SQL Queries in MySQL . . . 1137
The mysql_query Table Function . . . 1137
The mysql_execute Function . . . 1138
Settings . . . 1138
Schema Cache . . . 1138

PostgreSQL Extension **1139**
Installing and Loading . . . 1139
Connecting . . . 1139
Configuration . . . 1139
Configuring via Secrets . . . 1140
Configuring via Environment Variables . . . 1140
Usage . . . 1141
Writing Data to PostgreSQL . . . 1141
CREATE TABLE . . . 1142
INSERT INTO . . . 1142
SELECT . . . 1142
COPY . . . 1142
UPDATE . . . 1142
DELETE . . . 1142
ALTER TABLE . . . 1143
DROP TABLE . . . 1143
CREATE VIEW . . . 1143
CREATE SCHEMA / DROP SCHEMA . . . 1143
DETACH . . . 1143
Transactions . . . 1143
Running SQL Queries in PostgreSQL . . . 1144
The postgres_query Table Function . . . 1144
The postgres_execute Function . . . 1144
Settings . . . 1144
Schema Cache . . . 1145

Spatial **1147**
Spatial Extension . . . 1147
Installing and Loading . . . 1147
The GEOMETRY Type . . . 1147
Spatial Functions . . . 1147
Function Index . . . 1147
Scalar Functions . . . 1153
DuckDB_PROJ_Compiled_Version . . . 1153
DuckDB_Proj_Version . . . 1153
ST_Affine . . . 1154
ST_Area . . . 1155
ST_Area_Spheroid . . . 1155
ST_AsGeoJSON . . . 1155
ST_AsHEXWKB . . . 1156
ST_AsSVG . . . 1156
ST_AsText . . . 1157

ST_AsWKB . . . 1157
ST_Azimuth . . . 1158
ST_Boundary . . . 1158
ST_Buffer . . . 1158
ST_BuildArea . . . 1159
ST_Centroid . . . 1159
ST_Collect . . . 1160
ST_CollectionExtract . . . 1160
ST_ConcaveHull . . . 1161
ST_Contains . . . 1161
ST_ContainsProperly . . . 1162
ST_ConvexHull . . . 1162
ST_CoverageInvalidEdges . . . 1162
ST_CoverageSimplify . . . 1163
ST_CoverageUnion . . . 1163
ST_CoveredBy . . . 1163
ST_Covers . . . 1163
ST_Crosses . . . 1164
ST_DWithin . . . 1164
ST_DWithin_GEOS . . . 1164
ST_DWithin_Spheroid . . . 1164
ST_Difference . . . 1165
ST_Dimension . . . 1165
ST_Disjoint . . . 1165
ST_Distance . . . 1166
ST_Distance_GEOS . . . 1166
ST_Distance_Sphere . . . 1166
ST_Distance_Spheroid . . . 1167
ST_Dump . . . 1167
ST_EndPoint . . . 1168
ST_Envelope . . . 1168
ST_Equals . . . 1168
ST_Extent . . . 1169
ST_Extent_Approx . . . 1169
ST_ExteriorRing . . . 1169
ST_FlipCoordinates . . . 1169
ST_Force2D . . . 1170
ST_Force3DM . . . 1170
ST_Force3DZ . . . 1170
ST_Force4D . . . 1171
ST_GeomFromGeoJSON . . . 1171
ST_GeomFromHEXEWKB . . . 1172
ST_GeomFromHEXWKB . . . 1172
ST_GeomFromText . . . 1172
ST_GeomFromWKB . . . 1172
ST_GeometryType . . . 1173
ST_HasM . . . 1173
ST_HasZ . . . 1174
ST_Hilbert . . . 1174
ST_Intersection . . . 1175
ST_Intersects . . . 1175
ST_Intersects_Extent . . . 1175
ST_IsClosed . . . 1175

ST_IsEmpty . . . 1176
ST_IsRing . . . 1176
ST_IsSimple . . . 1176
ST_IsValid . . . 1176
ST_Length . . . 1177
ST_Length_Spheroid . . . 1177
ST_LineInterpolatePoint . . . 1177
ST_LineInterpolatePoints . . . 1178
ST_LineMerge . . . 1178
ST_LineString2DFromWKB . . . 1178
ST_LineSubstring . . . 1178
ST_M . . . 1179
ST_MMax . . . 1179
ST_MMin . . . 1179
ST_MakeEnvelope . . . 1180
ST_MakeLine . . . 1180
ST_MakePolygon . . . 1180
ST_MakeValid . . . 1181
ST_MaximumInscribedCircle . . . 1181
ST_MinimumRotatedRectangle . . . 1181
ST_Multi . . . 1182
ST_NGeometries . . . 1182
ST_NInteriorRings . . . 1182
ST_NPoints . . . 1183
ST_Node . . . 1183
ST_Normalize . . . 1183
ST_NumGeometries . . . 1184
ST_NumInteriorRings . . . 1184
ST_NumPoints . . . 1184
ST_Overlaps . . . 1184
ST_Perimeter . . . 1185
ST_Perimeter_Spheroid . . . 1185
ST_Point . . . 1185
ST_Point2D . . . 1186
ST_Point2DFromWKB . . . 1186
ST_Point3D . . . 1186
ST_Point4D . . . 1186
ST_PointN . . . 1187
ST_PointOnSurface . . . 1187
ST_Points . . . 1187
ST_Polygon2DFromWKB . . . 1187
ST_Polygonize . . . 1188
ST_QuadKey . . . 1188
ST_ReducePrecision . . . 1189
ST_RemoveRepeatedPoints . . . 1189
ST_Reverse . . . 1189
ST_ShortestLine . . . 1189
ST_Simplify . . . 1190
ST_SimplifyPreserveTopology . . . 1190
ST_StartPoint . . . 1190
ST_TileEnvelope . . . 1190
ST_Touches . . . 1191
ST_Transform . . . 1191

ST_Union . . . 1193
ST_VoronoiDiagram . . . 1193
ST_Within . . . 1193
ST_WithinProperly . . . 1194
ST_X . . . 1194
ST_XMax . . . 1194
ST_XMin . . . 1195
ST_Y . . . 1195
ST_YMax . . . 1195
ST_YMin . . . 1196
ST_Z . . . 1196
ST_ZMFlag . . . 1196
ST_ZMax . . . 1197
ST_ZMin . . . 1197
Aggregate Functions . . . 1198
ST_CoverageInvalidEdges_Agg . . . 1198
ST_CoverageSimplify_Agg . . . 1198
ST_CoverageUnion_Agg . . . 1198
ST_Envelope_Agg . . . 1199
ST_Extent_Agg . . . 1199
ST_Intersection_Agg . . . 1199
ST_MemUnion_Agg . . . 1200
ST_Union_Agg . . . 1200
Macro Functions . . . 1200
ST_Rotate . . . 1200
ST_RotateX . . . 1200
ST_RotateY . . . 1201
ST_RotateZ . . . 1201
ST_Scale . . . 1202
ST_TransScale . . . 1202
ST_Translate . . . 1202
Table Functions . . . 1202
ST_Drivers . . . 1202
ST_GeneratePoints . . . 1203
ST_Read . . . 1203
ST_ReadOSM . . . 1205
ST_ReadSHP . . . 1206
ST_Read_Meta . . . 1206
R-Tree Indexes . . . 1206
Why Should I Use an R-Tree Index? . . . 1206
How Do R-Tree Indexes Work? . . . 1207
What Are the Limitations of R-Tree Indexes in DuckDB? . . . 1207
How to Use R-Tree Indexes in DuckDB . . . 1207
Example . . . 1208
Performance Considerations . . . 1208
Bulk Loading & Maintenance . . . 1208
Memory Usage . . . 1209
Tuning . . . 1209
Options . . . 1209
R-Tree Table Functions . . . 1209
GDAL Integration . . . 1210
GDAL Based COPY Function . . . 1211
Limitations . . . 1211

SQLite Extension **1213**
Installing and Loading . . . 1213
Usage . . . 1213
Data Types . . . 1214
Opening SQLite Databases Directly . . . 1215
Writing Data to SQLite . . . 1215
Concurrency . . . 1216
Settings . . . 1216
Supported Operations . . . 1216
`CREATE TABLE` . . . 1216
`INSERT INTO` . . . 1216
`SELECT` . . . 1216
`COPY` . . . 1216
`UPDATE` . . . 1216
`DELETE` . . . 1217
`ALTER TABLE` . . . 1217
`DROP TABLE` . . . 1217
`CREATE VIEW` . . . 1217
Transactions . . . 1217

TPC-DS Extension **1219**
Installing and Loading . . . 1219
Usage . . . 1219
Generating the Schema . . . 1219
Limitations . . . 1219

TPC-H Extension **1221**
Installing and Loading . . . 1221
Usage . . . 1221
Generating Data . . . 1221
Running a Query . . . 1221
Listing Queries . . . 1222
Listing Expected Answers . . . 1222
Generating the Schema . . . 1222
Data Generator Parameters . . . 1222
Pre-Generated Data Sets . . . 1222
Resource Usage of the Data Generator . . . 1223
Limitation . . . 1223

UI Extension **1225**
Requirements . . . 1225
Usage . . . 1225
Local Query Execution . . . 1225
Configuration . . . 1225
Local Port . . . 1225
Remote URL . . . 1226
Polling Interval . . . 1226
Tips . . . 1226
Opening a CSV File with the DuckDB UI . . . 1226
Running the UI in Read-Only Mode . . . 1226
Limitations . . . 1227

Vector Similarity Search Extension **1229**
Usage . . . 1229

Index Options . . . 1230
Persistence . . . 1231
Inserts, Updates, Deletes and Re-Compaction . . . 1231
Bonus: Vector Similarity Search Joins . . . 1231
Limitations . . . 1232

Guides **1235**

Guides **1237**
Data Import and Export . . . 1237
CSV Files . . . 1237
Parquet Files . . . 1237
HTTP(S), S3 and GCP . . . 1237
JSON Files . . . 1237
Excel Files with the Spatial Extension . . . 1237
Querying Other Database Systems . . . 1238
Directly Reading Files . . . 1238
Performance . . . 1238
Meta Queries . . . 1238
ODBC . . . 1238
Python Client . . . 1238
Pandas . . . 1238
Apache Arrow . . . 1239
Relational API . . . 1239
Python Library Integrations . . . 1239
SQL Features . . . 1239
SQL Editors and IDEs . . . 1239
Data Viewers . . . 1239

Data Viewers **1241**
Tableau – A Data Visualization Tool . . . 1241
Database Creation . . . 1241
Installing the JDBC Driver . . . 1241
Driver Links . . . 1241
Using the PostgreSQL Dialect . . . 1242
Installing the Tableau DuckDB Connector . . . 1242
Server (Online) . . . 1243
macOS . . . 1243
Windows Desktop . . . 1244
Output . . . 1244
CLI Charting with YouPlot . . . 1245
Installing YouPlot . . . 1245
Piping DuckDB Queries to stdout . . . 1246
Connecting DuckDB to YouPlot . . . 1246
Bonus Round! stdin + stdout . . . 1247

Database Integration **1249**
Database Integration . . . 1249
MySQL Import . . . 1249
Installation and Loading . . . 1249
Usage . . . 1249
PostgreSQL Import . . . 1250
Installation and Loading . . . 1250

Usage . . . 1250
SQLite Import . . . 1250
Installation and Loading . . . 1251
Usage . . . 1251

File Formats **1253**
File Formats . . . 1253
CSV Import . . . 1253
CSV Export . . . 1253
Directly Reading Files . . . 1253
`read_text` . . . 1254
`read_blob` . . . 1254
Schema . . . 1254
Handling Missing Metadata . . . 1254
Support for Projection Pushdown . . . 1255
Excel Import . . . 1255
Importing Excel Sheets . . . 1255
Importing a Specific Range . . . 1255
Creating a New Table . . . 1256
Loading to an Existing Table . . . 1256
Importing a Sheet with/without a Header . . . 1256
Detecting Types . . . 1256
See Also . . . 1257
Excel Export . . . 1257
Exporting Excel Sheets . . . 1257
Type Conversions . . . 1257
See Also . . . 1257
JSON Import . . . 1258
JSON Export . . . 1258
Parquet Import . . . 1258
Adjusting the Schema on the Fly . . . 1259
Parquet Export . . . 1259
Querying Parquet Files . . . 1260
File Access with the file: Protocol . . . 1260

Network and Cloud Storage **1261**
Network and Cloud Storage . . . 1261
HTTP Parquet Import . . . 1261
S3 Parquet Import . . . 1261
Prerequisites . . . 1261
Credentials and Configuration . . . 1261
Querying . . . 1262
Google Cloud Storage (GCS) and Cloudflare R2 . . . 1262
S3 Parquet Export . . . 1262
S3 Iceberg Import . . . 1263
Prerequisites . . . 1263
Credentials . . . 1263
Loading Iceberg Tables from S3 . . . 1264
S3 Express One . . . 1264
Credentials and Configuration . . . 1264
Instance Location . . . 1264
Querying . . . 1265
Performance . . . 1265
Google Cloud Storage Import . . . 1265

Prerequisites . . . 1265
Credentials and Configuration . . . 1265
Querying . . . 1265
Attaching to a Database . . . 1266
Cloudflare R2 Import . . . 1266
Prerequisites . . . 1266
Credentials and Configuration . . . 1266
Querying . . . 1266
Attach to a DuckDB Database over HTTPS or S3 . . . 1266
Prerequisites . . . 1266
Attaching to a Database over HTTPS . . . 1266
Attaching to a Database over the S3 API . . . 1267
Limitations . . . 1267
Fastly Object Storage Import . . . 1267
Prerequisites . . . 1267
Credentials and Configuration . . . 1268
Querying . . . 1268

Meta Queries **1269**
Describe . . . 1269
Describing a Table . . . 1269
Describing a Query . . . 1269
Using `DESCRIBE` in a Subquery . . . 1269
Describing Remote Tables . . . 1269
EXPLAIN: Inspect Query Plans . . . 1270
Additional Explain Settings . . . 1271
See Also . . . 1271
EXPLAIN ANALYZE: Profile Queries . . . 1272
See Also . . . 1273
List Tables . . . 1273
See Also . . . 1274
Summarize . . . 1274
Usage . . . 1274
Example . . . 1274
Using `SUMMARIZE` in a Subquery . . . 1275
Summarizing Remote Tables . . . 1275
DuckDB Environment . . . 1275
Version . . . 1275
Platform . . . 1276
Extensions . . . 1276
Meta Table Functions . . . 1276

ODBC **1277**
ODBC 101: A Duck Themed Guide to ODBC . . . 1277
What is ODBC? . . . 1277
General Concepts . . . 1277
Handles . . . 1277
Connecting . . . 1278
Error Handling and Diagnostics . . . 1279
Buffers and Binding . . . 1279
Setting up an Application . . . 1279
1. Include the SQL Header Files . . . 1279
2. Define the ODBC Handles and Connect to the Database . . . 1280
3. Adding a Query . . . 1281

4. Fetching Results . . . 1281
5. Go Wild . . . 1281
6. Free the Handles and Disconnecting . . . 1282
Sample Application . . . 1282
Sample `.cpp` File . . . 1282
Sample `CMakelists.txt` File . . . 1283

Performance **1285**
Performance Guide . . . 1285
Environment . . . 1285
Hardware Configuration . . . 1285
CPU . . . 1285
Memory . . . 1285
Local Disk . . . 1286
Network-Attached Disks . . . 1286
Operating System . . . 1286
Linux . . . 1286
Memory Allocator . . . 1286
Data Import . . . 1287
Recommended Import Methods . . . 1287
Methods to Avoid . . . 1287
Schema . . . 1287
Types . . . 1287
Microbenchmark: Using Timestamps . . . 1287
Microbenchmark: Joining on Strings . . . 1288
Constraints . . . 1288
Microbenchmark: The Effect of Primary Keys . . . 1288
Indexing . . . 1289
Zonemaps . . . 1289
The Effect of Ordering on Zonemaps . . . 1289
Microbenchmark: The Effect of Ordering . . . 1289
Ordered Integers . . . 1289
ART Indexes . . . 1290
ART Index Scans . . . 1290
Indexes and Memory . . . 1290
Indexes and Opening Databases . . . 1290
Join Operations . . . 1291
How to Force a Join Order . . . 1291
Turn off the Join Order Optimizer . . . 1291
Create Temporary Tables . . . 1291
File Formats . . . 1291
Handling Parquet Files . . . 1291
Reasons for Querying Parquet Files . . . 1292
Reasons against Querying Parquet Files . . . 1292
The Effect of Row Group Sizes . . . 1292
Parquet File Sizes . . . 1293
Hive Partitioning for Filter Pushdown . . . 1293
More Tips on Reading and Writing Parquet Files . . . 1293
Loading CSV Files . . . 1293
Loading Many Small CSV Files . . . 1293
Tuning Workloads . . . 1294
The `preserve_insertion_order` Option . . . 1294
Parallelism (Multi-Core Processing) . . . 1294
The Effect of Row Groups on Parallelism . . . 1294

Too Many Threads . . . 1294
Larger-than-Memory Workloads (Out-of-Core Processing) . . . 1294
Spilling to Disk . . . 1294
Blocking Operators . . . 1295
Limitations . . . 1295
Profiling . . . 1295
Prepared Statements . . . 1295
Querying Remote Files . . . 1296
Avoid Reading Unnecessary Data . . . 1296
Avoid Reading Data More Than Once . . . 1296
Best Practices for Using Connections . . . 1296
Persistent vs. In-Memory Tables . . . 1297
My Workload Is Slow . . . 1297
Benchmarks . . . 1297
Data Sets . . . 1298
A Note on Benchmarks . . . 1298
Disclaimer on Benchmarks . . . 1298
Working with Huge Databases . . . 1298

Python **1299**
Installing the Python Client . . . 1299
Installing via Pip . . . 1299
Installing from Source . . . 1299
Executing SQL in Python . . . 1299
Jupyter Notebooks . . . 1300
Library Installation . . . 1300
Library Import and Configuration . . . 1300
Connecting to DuckDB Natively . . . 1300
Connecting to DuckDB via SQLAlchemy . . . 1301
Querying DuckDB . . . 1301
Querying Pandas Dataframes . . . 1302
Visualizing DuckDB Data . . . 1302
Boxplot & Histogram . . . 1302
Install and Load DuckDB httpfs Extension . . . 1303
Summary . . . 1303
marimo Notebooks . . . 1304
Installation . . . 1304
SQL in marimo . . . 1304
Connecting a Custom DuckDB Connection . . . 1305
Database, Schema, and Table Auto-Discovery . . . 1305
Reference a Local Dataframe . . . 1305
Reference the Output of a SQL Cell . . . 1305
Reactive SQL Cells . . . 1305
DuckDB-Powered OLAP Analytics in marimo . . . 1306
Next Steps . . . 1306
SQL on Pandas . . . 1307
Import from Pandas . . . 1307
See Also . . . 1307
Export to Pandas . . . 1307
See Also . . . 1308
Import from Numpy . . . 1308
See Also . . . 1308
Export to Numpy . . . 1308
See Also . . . 1308

SQL on Apache Arrow . . . 1309
Apache Arrow Tables . . . 1309
Apache Arrow Datasets . . . 1309
Apache Arrow Scanners . . . 1309
Apache Arrow RecordBatchReaders . . . 1310
Import from Apache Arrow . . . 1310
Export to Apache Arrow . . . 1311
Export to an Arrow Table . . . 1311
Export as a RecordBatchReader . . . 1311
Export from Relational API . . . 1312
Relational API on Pandas . . . 1312
Multiple Python Threads . . . 1313
Setup . . . 1313
Reader and Writer Functions . . . 1313
Create Threads . . . 1314
Run Threads and Show Results . . . 1314
Integration with Ibis . . . 1314
Installation . . . 1314
Create a Database File . . . 1315
Interactive Mode . . . 1316
Common Operations . . . 1316
filter . . . 1317
select . . . 1317
mutate . . . 1318
selectors . . . 1319
`order_by` . . . 1320
aggregate . . . 1321
`group_by` . . . 1322
Chaining It All Together . . . 1323
Learn More . . . 1324
Integration with Polars . . . 1324
Installation . . . 1324
Polars to DuckDB . . . 1324
DuckDB to Polars . . . 1325
Using fsspec Filesystems . . . 1325
Example . . . 1325

SQL Editors **1327**
DBeaver SQL IDE . . . 1327
Installing DBeaver . . . 1327
Alternative Driver Installation . . . 1333

SQL Features **1339**
AsOf Join . . . 1339
What is an AsOf Join? . . . 1339
Portfolio Example Data Set . . . 1339
Inner AsOf Joins . . . 1340
Outer AsOf Joins . . . 1340
AsOf Joins with the `USING` Keyword . . . 1341
Clarification on Column Selection with `USING` in ASOF Joins . . . 1341
See Also . . . 1342
Full-Text Search . . . 1342
Example: Shakespeare Corpus . . . 1342
Creating a Full-Text Search Index . . . 1342

Note on Generating the Corpus Table . . . 1343
query and query_table Functions . . . 1343

Snippets **1345**
Create Synthetic Data . . . 1345
Dutch Railway Datasets . . . 1346
Loading the Datasets . . . 1346
Datasets . . . 1347
Yearly Data Sets . . . 1347
Monthly Data Sets . . . 1347
Sharing Macros . . . 1347
Sharing a Scalar Macro . . . 1347
Sharing a Table Macro . . . 1348
Analyzing a Git Repository . . . 1349
Exporting the Git Log . . . 1349
Loading the Git Log into DuckDB . . . 1349
Analyzing the Log . . . 1350
Common Topics . . . 1350
Visualizing the Number of Commits . . . 1352
Importing Duckbox Tables . . . 1352
Loading Duckbox Tables to DuckDB . . . 1352
Using `shellfs` . . . 1353
Limitations . . . 1353
Copying an In-Memory Database to a File . . . 1354

Troubleshooting **1355**
Crashes . . . 1355
Types of Crashes . . . 1355
Recovering Data . . . 1355
Troubleshooting the Crash . . . 1355
Using the Latest Stable and Preview Builds . . . 1355
Search for Existing Issues . . . 1356
Disabling the Query Optimizer . . . 1356
Try to Isolate the Issue . . . 1356
Submitting an Issue . . . 1356
Out of Memory Errors . . . 1356
Types of Out of Memory Errors . . . 1357
`OutOfMemoryException` . . . 1357
OOM Reaper (Linux) . . . 1357
Troubleshooting Out of Memory Errors . . . 1357
See Also . . . 1357

Glossary of Terms **1359**
Terms . . . 1359
In-Process Database Management System . . . 1359
Replacement Scan . . . 1359
Extension . . . 1359
Platform . . . 1359

Browsing Offline **1361**

Operations Manual **1363**

Overview **1365**

DuckDB's Footprint **1367**
Files Created by DuckDB . . . 1367
Global Files and Directories . . . 1367
Local Files and Directories . . . 1367
Gitignore for DuckDB . . . 1368
Sample Gitignore Files . . . 1368
Ignore Temporary Files but Keep Database . . . 1368
Ignore Database and Temporary Files . . . 1368
Reclaiming Space . . . 1368
`CHECKPOINT` . . . 1368
`VACUUM` . . . 1368
Compacting a Database by Copying . . . 1369

Logging **1371**
Logging . . . 1371
Basics . . . 1371
Log Level . . . 1371
Log Types . . . 1371
Logging-Specific Types . . . 1371
Structured Logging . . . 1372
List of Available Log Types . . . 1372
Log Storage . . . 1372

Securing DuckDB **1373**
Securing DuckDB . . . 1373
Reporting Vulnerabilities . . . 1373
Safe Mode (CLI) . . . 1373
Restricting File Access . . . 1373
Disabling File Access . . . 1373
The `allowed_directories` and `allowed_paths` Options . . . 1374
Secrets . . . 1374
Locking Configurations . . . 1374
Prepared Statements to Prevent SQL Injection . . . 1374
Constrain Resource Usage . . . 1375
Extensions . . . 1375
Privileges . . . 1375
Generic Solutions . . . 1375
Embedding DuckDB . . . 1375
CLI Client . . . 1375
Securing Extensions . . . 1376
DuckDB Signature Checks . . . 1376
Overview of Security Levels for Extensions . . . 1376
Community Extensions . . . 1376
Disabling Autoinstalling and Autoloading Known Extensions . . . 1377
Always Require Signed Extensions . . . 1377

Non-Deterministic Behavior **1379**
Set Semantics . . . 1379
Different Results on Different Platforms: `array_distinct` . . . 1379
Floating-Point Aggregate Operations with Multi-Threading . . . 1379
Working Around Non-Determinism . . . 1380

Limits **1381**
Limit Values . . . 1381

Size of Database Files . . . 1381

Development 1383

DuckDB Repositories 1385

Main Repositories . . . 1385
Clients . . . 1385
Connectors . . . 1385
Extensions . . . 1385
Specifications . . . 1385

Profiling 1387

Statements . . . 1387
The `EXPLAIN` Statement . . . 1387
The `EXPLAIN ANALYZE` Statement . . . 1387
The `FORMAT` Option . . . 1387
Pragmas . . . 1387
Metrics . . . 1388
Cumulative Metrics . . . 1389
Detailed Profiling . . . 1389
Optimizer Metrics . . . 1389
Planner Metrics . . . 1389
Physical Planner Metrics . . . 1390
Custom Metrics Examples . . . 1390
Query Graphs . . . 1391
Notation in Query Plans . . . 1392

Building DuckDB 1393

Building DuckDB from Source . . . 1393
When Should You Build DuckDB? . . . 1393
Prerequisites . . . 1393
Getting Started . . . 1393
Platforms . . . 1393
Platforms with Full Support . . . 1393
Platforms with Partial Support . . . 1394
Platforms with Best Effort Support . . . 1394
Outdated Platforms . . . 1394
Amalgamation Build . . . 1395
Limitations . . . 1395
Troubleshooting Guides . . . 1395
Building Configuration . . . 1395
Build Types . . . 1395
`release` . . . 1395
`debug` . . . 1395
`relassert` . . . 1396
`reldebug` . . . 1396
`benchmark` . . . 1396
`tidy-check` . . . 1396
`format-fix` | `format-changes` | `format-main` . . . 1396
Extension Selection . . . 1396
Package Flags . . . 1396
`BUILD_PYTHON` . . . 1396
`BUILD_SHELL` . . . 1397

BUILD_BENCHMARK . . . 1397
BUILD_JDBC . . . 1397
BUILD_ODBC . . . 1397
Miscellaneous Flags . . . 1397
DISABLE_UNITY . . . 1397
DISABLE_SANITIZER . . . 1397
Overriding Git Hash and Version . . . 1397
Building Extensions . . . 1397
Building Extensions . . . 1398
Special Extension Flags . . . 1398
BUILD_JEMALLOC . . . 1398
BUILD_TPCE . . . 1398
Debug Flags . . . 1398
CRASH_ON_ASSERT . . . 1398
DISABLE_STRING_INLINE . . . 1398
DISABLE_MEMORY_SAFETY . . . 1398
DESTROY_UNPINNED_BLOCKS . . . 1398
DEBUG_STACKTRACE . . . 1399
Using a CMake Configuration File . . . 1399
Android . . . 1399
Building the DuckDB Library Using the Android NDK . . . 1399
Building the CLI in Termux . . . 1400
Troubleshooting . . . 1401
Log Library Is Missing . . . 1401
Linux . . . 1401
Prerequisites . . . 1401
Ubuntu and Debian . . . 1401
Fedora, CentOS and Red Hat . . . 1401
Alpine Linux . . . 1401
Using the DuckDB CLI Client on Linux . . . 1402
Building Using Extension Flags . . . 1402
Troubleshooting . . . 1402
R Package on Linux AArch64: `too many GOT entries` Build Error . . . 1402
Building the httpfs Extension Fails . . . 1403
macOS . . . 1403
Prerequisites . . . 1403
Building DuckDB . . . 1403
Troubleshooting . . . 1403
Build Failure: `'string' file not found` . . . 1403
Debug Build Prints malloc Warning . . . 1404
Raspberry Pi . . . 1404
Raspberry Pi (64-bit) . . . 1404
Raspberry Pi (32-bit) . . . 1405
Windows . . . 1405
Visual Studio . . . 1405
MSYS2 and MinGW64 . . . 1405
Building the Go Client . . . 1406
Python . . . 1406
Prerequisites . . . 1406
1. DuckDB Repository . . . 1406
2. Python Virtual Environment . . . 1407
Building from Source . . . 1407
Default Release, Debug Build or Cloud Storage . . . 1407

Adding Extensions . . . 1408
Show All Installed Extensions . . . 1408
Development Environment . . . 1409
Debugging from the CLI . . . 1409
Debugging . . . 1409
Debugging in an IDE / CLion . . . 1410
Development and Stubs . . . 1410
What are py::objects and py::handles? . . . 1411
Troubleshooting . . . 1411
Pip Fails with `No names found, cannot describe anything` . . . 1411
Building with the httpfs Extension Fails . . . 1411
Importing DuckDB Fails with `symbol not found in flat namespace` . . . 1411
Python Fails with `No module named 'duckdb.duckdb'` . . . 1412
R . . . 1412
The Build Only Uses a Single Thread . . . 1412
Troubleshooting . . . 1412
The Build Runs Out of Memory . . . 1412
Unofficial and Unsupported Platforms . . . 1412
32-bit Architectures . . . 1413
Big-Endian Architectures . . . 1413
RISC-V Architectures . . . 1413

Benchmark Suite **1415**
Getting Started . . . 1415
Listing Benchmarks . . . 1415
Running Benchmarks . . . 1415
Running a Single Benchmark . . . 1415
Running Multiple Benchmark Using a Regular Expression . . . 1416

Testing **1417**
Overview . . . 1417
How is DuckDB Tested? . . . 1417
sqllogictest Introduction . . . 1417
Query Verification . . . 1418
Editors & Syntax Highlighting . . . 1418
Temporary Files . . . 1418
Require & Extensions . . . 1418
Writing Tests . . . 1419
Development and Testing . . . 1419
Philosophy . . . 1419
Frameworks . . . 1419
Client Connector Tests . . . 1419
Functions for Generating Test Data . . . 1419
`test_all_types` Function . . . 1419
`test_vector_types` Function . . . 1420
Debugging . . . 1420
Triggering Which Tests to Run . . . 1421
Result Verification . . . 1421
NULL Values and Empty Strings . . . 1422
Error Verification . . . 1422
Regex . . . 1422
File . . . 1422
Row-Wise vs. Value-Wise Result Ordering . . . 1423
Hashes and Outputting Values . . . 1423

Result Sorting . . . 1424
Query Labels . . . 1424
Persistent Testing . . . 1424
Loops . . . 1425
Data Generation without Loops . . . 1426
Multiple Connections . . . 1426
Concurrent Connections . . . 1427
Catch C/C++ Tests . . . 1427

Internals **1429**

Overview of DuckDB Internals **1431**
Parser . . . 1431
ParsedExpression . . . 1431
TableRef . . . 1431
QueryNode . . . 1431
SQL Statement . . . 1431
Binder . . . 1432
Logical Planner . . . 1432
Optimizer . . . 1432
Column Binding Resolver . . . 1432
Physical Plan Generator . . . 1432
Execution . . . 1432

Storage Versions and Format **1433**
Compatibility . . . 1433
Backward Compatibility . . . 1433
Forward Compatibility . . . 1433
How to Move between Storage Formats . . . 1433
Explicit Storage Versions . . . 1433
Converting between Storage Versions . . . 1434
Storage Header . . . 1434
Storage Version Table . . . 1434
Compression . . . 1435
Compression Algorithms . . . 1435
Disk Usage . . . 1436
Row Groups . . . 1436
Troubleshooting . . . 1436
Error Message When Opening an Incompatible Database File . . . 1436

Execution Format **1437**
Data Flow . . . 1437
Vector Format . . . 1437
Flat Vectors . . . 1437
Constant Vectors . . . 1438
Dictionary Vectors . . . 1440
Sequence Vectors . . . 1440
Unified Vector Format . . . 1441
Complex Types . . . 1442
String Vectors . . . 1442
List Vectors . . . 1442
Struct Vectors . . . 1443
Map Vectors . . . 1443

Union Vectors . . . 1443

Pivot Internals **1445**
PIVOT . . . 1445
UNPIVOT . . . 1446
Internals . . . 1446

DuckDB Blog 1449

Testing Out DuckDB's Full Text Search Extension **1451**
Preparing the Data . . . 1451
Building the Search Engine . . . 1452
Running the Benchmark . . . 1452
Results . . . 1453

Efficient SQL on Pandas with DuckDB **1455**
SQL on Pandas . . . 1455
Data Integration & SQL on Pandas . . . 1455
SQL on Pandas Performance . . . 1456
Benchmark Setup and Data Set . . . 1456
Setup . . . 1456
Ungrouped Aggregates . . . 1456
Grouped Aggregate . . . 1457
Grouped Aggregate with a Filter . . . 1458
Joins . . . 1459
Takeaway . . . 1461
Appendix A: There and Back Again: Transferring Data from Pandas to a SQL Engine and Back . . . 1461
Appendix B: Comparison to PandaSQL . . . 1462
Appendix C: Query on Parquet Directly . . . 1462
Setup . . . 1462
Ungrouped Aggregate . . . 1462
Joins . . . 1463

Querying Parquet with Precision Using DuckDB **1465**
DuckDB and Parquet . . . 1465
Automatic Filter & Projection Pushdown . . . 1466
DuckDB versus Pandas . . . 1467
Reading Multiple Parquet Files . . . 1467
Concatenate into a Single File . . . 1468
Querying the Large File . . . 1468
Counting Rows . . . 1469
Filtering Rows . . . 1470
Aggregates . . . 1471
Conclusion . . . 1472

Fastest Table Sort in the West – Redesigning DuckDB's Sort **1473**
Sorting Relational Data . . . 1473
Binary String Comparison . . . 1474
Radix Sort . . . 1475
Two-Phase Parallel Sorting . . . 1475
Columns or Rows? . . . 1476
External Sorting . . . 1477
Comparison with Other Systems . . . 1478
Random Integers . . . 1479

TPC-DS . . . 1482
Catalog Sales (Numeric Types) . . . 1483
Customer (Strings & Integers) . . . 1484
Conclusion . . . 1485
Appendix A: Predication . . . 1485
Appendix B: Zig-Zagging . . . 1486
Appendix C: x86 Experiment . . . 1487

Windowing in DuckDB **1489**
Beyond Sets . . . 1489
Window Functions . . . 1489
Power Generation Example . . . 1490
Under the Feathers . . . 1492
Pipeline Breaking . . . 1493
Partitioning and Sorting . . . 1494
Aggregation . . . 1494
Conclusion . . . 1496

DuckDB-Wasm: Efficient Analytical SQL in the Browser **1497**
Efficient Analytics in the Browser . . . 1497
How to Get Data In? . . . 1498
How to Get Data Out? . . . 1499
Looks like Arrow to Me . . . 1499
Web Filesystem . . . 1500
Advanced Features . . . 1500
Multithreading . . . 1501
Web Shell . . . 1502
Evaluation . . . 1503
Future Research . . . 1503

Fast Moving Holistic Aggregates **1505**
What Is an Aggregate Function? . . . 1505
Holistic Aggregates . . . 1505
Statistical Holistic Aggregates . . . 1505
Python Example . . . 1506
Moving Holistic Aggregation . . . 1507
Quantile . . . 1507
InterQuartile Ranges (IQR) . . . 1508
Median Absolute Deviation (MAD) . . . 1508
Mode . . . 1508
Microbenchmarks . . . 1509
Conclusion . . . 1509

DuckDB – Lord of the Enums: The Fellowship of the Categorical and Factors **1511**
SQL . . . 1512
Python . . . 1513
Setup . . . 1513
Usage . . . 1513
R . . . 1513
Setup . . . 1513
Usage . . . 1514
Benchmark Comparison . . . 1514
Dataset . . . 1514
Grouped Aggregation . . . 1514

Filter . . . 1515
Enum – Enum Comparison . . . 1515
Storage . . . 1516
What about the Sequels? . . . 1516
Feedback . . . 1517

DuckDB Quacks Arrow: A Zero-Copy Data Integration between Apache Arrow and DuckDB **1519**
Quick Tour . . . 1519
R . . . 1519
Python . . . 1520
DuckDB and Arrow: The Basics . . . 1520
Setup . . . 1520
Streaming Data from/to Arrow . . . 1522
Benchmark Comparison . . . 1523
Projection Pushdown . . . 1523
Filter Pushdown . . . 1523
Streaming . . . 1524
Conclusion and Feedback . . . 1526

DuckDB Time Zones: Supporting Calendar Extensions **1527**
What is Time? . . . 1527
Instants . . . 1527
Temporal Binning . . . 1527
Naïve Timestamps . . . 1529
Time Zone Data Types . . . 1529
ICU Temporal Binning . . . 1530
ICU Time Zones . . . 1530
ICU Temporal Binning Functions . . . 1530
ICU Calendar Support . . . 1531
Caveats . . . 1531
Future Work . . . 1532
DuckDB Features . . . 1532
ICU Functionality . . . 1532
Separation of Concerns . . . 1532
Conclusion and Feedback . . . 1532

Parallel Grouped Aggregation in DuckDB **1533**
Introduction . . . 1533
Hash Tables for Aggregation . . . 1534
Collision Handling . . . 1534
Parallel Aggregation . . . 1538
Experiments . . . 1539
Conclusion . . . 1542

Friendlier SQL with DuckDB **1543**
`SELECT * EXCLUDE` . . . 1544
`SELECT * REPLACE` . . . 1544
`GROUP BY ALL` . . . 1544
`ORDER BY ALL` . . . 1545
Column Aliases in `WHERE` / `GROUP BY` / `HAVING` . . . 1545
Case Insensitivity While Maintaining Case . . . 1546
Friendly Error Messages . . . 1546
String Slicing . . . 1546
Simple List and Struct Creation . . . 1547

List Slicing . . . 1547
Struct Dot Notation . . . 1547
Trailing Commas . . . 1547
Function Aliases from Other Databases . . . 1548
Auto-Increment Duplicate Column Names . . . 1548
Implicit Type Casts . . . 1548
Other Friendly Features . . . 1549
Ideas for the Future . . . 1549

Range Joins in DuckDB **1551**
Introduction . . . 1551
Flight Data . . . 1551
Equality Predicates . . . 1552
Range Predicates . . . 1553
Infinite Time . . . 1553
Common Join Algorithms . . . 1554
Hash Joins . . . 1554
Nested Loop Joins . . . 1554
Range Joins . . . 1554
Piecewise Merge Join . . . 1555
Inequality Join (IEJoin) . . . 1555
Special Joins . . . 1557
Performance . . . 1558
Simple Measurements . . . 1558
Optimization Measurements . . . 1558
Conclusion and Feedback . . . 1559

Persistent Storage of Adaptive Radix Trees in DuckDB **1561**
ART Index . . . 1562
Trie . . . 1562
Vertical Compression (i.e., Radix Trees) . . . 1564
Horizontal Compression (i.e., ART) . . . 1565
ART in DuckDB . . . 1568
What Is It Used For? . . . 1568
ART Storage . . . 1569
Post-Order Traversal . . . 1569
Pointer Swizzling . . . 1571
Benchmarks . . . 1572
Storing Time . . . 1572
Load Time . . . 1573
Query Time (Cold) . . . 1573
Query Time (Hot) . . . 1574
Future Work . . . 1575
Roadmap . . . 1575

Querying Postgres Tables Directly from DuckDB **1577**
Introduction . . . 1578
Usage . . . 1579
Implementation . . . 1580
Parallelization . . . 1580
Transactional Synchronization . . . 1581
Projection and Selection Push-Down . . . 1581
Performance . . . 1581
Other Use Cases . . . 1582

Conclusion . . . 1583

Modern Data Stack in a Box with DuckDB **1585**
Summary . . . 1586
Motivation . . . 1586
Trade-offs . . . 1586
Choosing a Problem . . . 1587
Building the Environment . . . 1587
Meltano as a Wrapper for Pipeline Plugins . . . 1587
Wrangling the Data . . . 1588
Loading Sources . . . 1588
Building dbt Models . . . 1588
Connecting Superset . . . 1588
Conclusion . . . 1589
Next Steps . . . 1590

Lightweight Compression in DuckDB **1591**
Compression Intro . . . 1592
General Purpose Compression Algorithms . . . 1593
Lightweight Compression Algorithms . . . 1593
Compression Framework . . . 1594
Compression Algorithms . . . 1594
Constant Encoding . . . 1594
Run-Length Encoding (RLE) . . . 1595
Bit Packing . . . 1596
Frame of Reference . . . 1597
Dictionary Encoding . . . 1598
FSST . . . 1599
Chimp & Patas . . . 1600
Inspecting Compression . . . 1600
Conclusion & Future Goals . . . 1601

Announcing DuckDB 0.6.0 **1603**
What's in 0.6.0 . . . 1604
Storage Improvements . . . 1604
Compression Improvements . . . 1604
Performance Improvements . . . 1605
SQL Syntax Improvements . . . 1606
Memory Management Improvements . . . 1607
Shell Improvements . . . 1607

Announcing DuckDB 0.7.0 **1611**
What's in 0.7.0 . . . 1611
Data Ingestion/Export Improvements . . . 1611
Multi-Database Support . . . 1612
New SQL Features . . . 1613
Python API Improvements . . . 1613
Storage Improvements . . . 1615
Final Thoughts . . . 1615

JupySQL Plotting with DuckDB **1617**
Introduction . . . 1617
The Problem . . . 1617
DuckDB: A Highly Scalable Backend for Statistical Visualizations . . . 1620
Using DuckDB to Compute Histogram Statistics . . . 1622

Final Thoughts . . . 1623
Try It Out . . . 1624

Shredding Deeply Nested JSON, One Vector at a Time **1625**
Reading JSON Automatically with DuckDB . . . 1627
Newline Delimited JSON . . . 1628
Other JSON Formats . . . 1628
Manual Schemas . . . 1628
GitHub Archive Examples . . . 1629
Handling Inconsistent JSON Schemas . . . 1632
Storing as JSON to Parse at Query Time . . . 1636
Conclusion . . . 1637

The Return of the H2O.ai Database-like Ops Benchmark **1639**
The H2O.ai Database-like Ops Benchmark . . . 1639
The Data and Queries . . . 1639
Modifications to the Benchmark & Hardware . . . 1641
Changes Made to Install Scripts of Other Systems . . . 1641
Results . . . 1641
Questions about Certain Results? . . . 1642
Maintenance Plan . . . 1642

Introducing DuckDB for Swift **1643**
What's Included . . . 1643
Usage . . . 1643
Instantiating DuckDB . . . 1643
Populating DuckDB with a Remote CSV File . . . 1644
Querying the Database . . . 1645
Visualizing the Results . . . 1646
Conclusion . . . 1648

PostGEESE? Introducing The DuckDB Spatial Extension **1649**
What's in It? . . . 1649
Example Usage . . . 1650
What's Next? . . . 1651
Conclusion . . . 1652

10 000 Stars on GitHub **1653**

Announcing DuckDB 0.8.0 **1655**
What's New in 0.8.0 . . . 1655
Breaking SQL Changes . . . 1655
New SQL Features . . . 1656
Data Integration Improvements . . . 1657
Storage Improvements . . . 1658
Clients . . . 1658
Final Thoughts . . . 1658

Correlated Subqueries in SQL **1659**
Types of Subqueries . . . 1659
Uncorrelated Scalar Subqueries . . . 1659
Correlated Scalar Subqueries . . . 1660
EXISTS . . . 1661
IN / ANY / ALL . . . 1662
Performance . . . 1663

Conclusion . . . 1664

From Waddle to Flying: Quickly Expanding DuckDB's Functionality with Scalar Python UDFs 1665

Python UDFs . . . 1666
Quick-Tour . . . 1666
Generating Fake Data with Faker (Built-In Type UDF) . . . 1667
Swap String Case (PyArrow Type UDF) . . . 1667
Predicting Taxi Fare Costs (Ibis + PyArrow UDF) . . . 1668
Benchmarks . . . 1669
Built-In Python vs. PyArrow . . . 1669
Python UDFs vs. External Functions . . . 1670
Conclusions and Further Development . . . 1671

DuckDB ADBC – Zero-Copy Data Transfer via Arrow Database Connectivity 1673

Quick Tour . . . 1674
Setup . . . 1675
Insert Data . . . 1675
Read Data . . . 1675
Benchmark ADBC vs ODBC . . . 1676
Conclusions . . . 1676

Even Friendlier SQL with DuckDB 1677

The Future Is Now . . . 1679
Reusable Column Aliases . . . 1679
Dynamic Column Selection . . . 1680
`COLUMNS()` with `EXCLUDE` and `REPLACE` . . . 1682
`COLUMNS()` with Lambda Functions . . . 1682
Automatic JSON to Nested Types Conversion . . . 1683
`FROM` First in `SELECT` Statements . . . 1683
Function Chaining . . . 1684
Union by Name . . . 1684
Insert by Name . . . 1685
Dynamic `PIVOT` and `UNPIVOT` . . . 1685
List Lambda Functions . . . 1687
List Comprehensions . . . 1687
Exploding Struct.* . . . 1688
Automatic Struct Creation . . . 1688
Union Data Type . . . 1688
Additional Friendly Features . . . 1689
Summary and Future Work . . . 1690

DuckDB's AsOf Joins: Fuzzy Temporal Lookups 1691

What Is an AsOf Join? . . . 1691
Portfolio Example . . . 1691
Outer AsOf Joins . . . 1692
Windowing Alternative . . . 1693
Why AsOf? . . . 1694
State Tables . . . 1694
Sentinel Values . . . 1694
Event Table Variants . . . 1694
Usage . . . 1696
Under the Hood . . . 1697
Benchmarks . . . 1697
Window as State Table . . . 1698

Window with Ranking . . . 1700
Future Work . . . 1702
Happy Joining! . . . 1702

Announcing DuckDB 0.9.0 **1703**
What's New in 0.9.0 . . . 1703
Breaking SQL Changes . . . 1704
Core System Improvements . . . 1704
Storage Improvements . . . 1706
Extensions . . . 1706
Clients . . . 1707
Final Thoughts . . . 1708

DuckDB's CSV Sniffer: Automatic Detection of Types and Dialects **1709**
DuckDB's Automatic Detection . . . 1711
Dialect Detection . . . 1712
Type Detection . . . 1713
Header Detection . . . 1714
Type Replacement . . . 1714
Type Refinement . . . 1714
How Fast Is the Sniffing? . . . 1714
Varying Sampling Size . . . 1714
Varying Number of Columns . . . 1715
Conclusion & Future Work . . . 1716

Updates to the H2O.ai db-benchmark! **1719**
The Benchmark Has Been Updated! . . . 1719
New Benchmark Environment: c6id.metal Instance . . . 1719
Updating the Benchmark . . . 1720
Updated Settings . . . 1720
Results . . . 1720

Extensions for DuckDB-Wasm **1721**
DuckDB Extensions . . . 1721
Running DuckDB Extensions Locally . . . 1721
DuckDB-Wasm . . . 1721
DuckDB Extensions, in DuckDB-Wasm! . . . 1722
Using the TPC-H Extension in DuckDB-Wasm . . . 1722
Using the Spatial Extension in DuckDB-Wasm . . . 1723
Under the Hood . . . 1724
Developer Guide . . . 1726
Limitations . . . 1726
Conclusions . . . 1726

Multi-Database Support in DuckDB **1729**
Attaching Databases . . . 1730
Mix and Match . . . 1731
Transactions . . . 1732
Multi-Database Transactions . . . 1732
Copying Data between Databases . . . 1733
Directly Opening a Database . . . 1733
Conclusion . . . 1733
Future Work . . . 1733

Announcing DuckDB 0.10.0 **1735**
What's New in 0.10.0 . . . 1735
Breaking SQL Changes . . . 1736
Backward Compatibility . . . 1737
Forward Compatibility . . . 1737
CSV Reader Rework . . . 1738
Fixed-Length Arrays . . . 1738
Multi-Database Support . . . 1739
Secret Manager . . . 1739
Temporary Memory Manager . . . 1740
Adaptive Lossless Floating-Point Compression (ALP) . . . 1741
CLI Improvements . . . 1741
Final Thoughts . . . 1742
New Features . . . 1742
New Functions . . . 1742
Storage Improvements . . . 1743
Optimizations . . . 1743
Acknowledgments . . . 1743

SQL Gymnastics: Bending SQL into Flexible New Shapes **1745**
Traditional SQL Is Too Rigid to Reuse . . . 1746
Dynamic Aggregates Macro . . . 1746
Creating the Macro . . . 1746
Creating Version 2 of the Macro . . . 1748
Custom Summaries for Any Dataset . . . 1750
Creation . . . 1750
Execution . . . 1751
Step by Step Breakdown . . . 1751
Conclusion . . . 1754

Dependency Management in DuckDB Extensions **1755**
Introduction . . . 1755
The Difficulties of Complete Abstinence . . . 1755
DuckDB Extensions . . . 1756
Dependency Management . . . 1756
Using vcpkg with DuckDB . . . 1756
Example: Azure extension . . . 1757
Building Your Own DuckDB Extension . . . 1757
Setting up the Extension Template . . . 1757
Adding Functionality . . . 1758
Conclusion . . . 1759

42.parquet – A Zip Bomb for the Big Data Age **1761**

No Memory? No Problem. External Aggregation in DuckDB **1763**
Introduction . . . 1763
Memory Management . . . 1763
Memory Fragmentation . . . 1764
Invalid References . . . 1764
External Aggregation . . . 1765
Experiments . . . 1765
Conclusion . . . 1768

duckplyr: dplyr Powered by DuckDB **1769**
Background . . . 1769

The duckplyr R Package . . . 1772
Verbs . . . 1772
Expressions . . . 1773
Eager vs. Lazy Materialization . . . 1774
Benchmark: TPC-H Q1 . . . 1775
Conclusion . . . 1776

Vector Similarity Search in DuckDB **1777**
The Vector Similarity Search (VSS) Extension . . . 1777
Implementation . . . 1778
Limitations . . . 1779
Conclusion . . . 1779

Access 150k+ Datasets from Hugging Face with DuckDB **1781**
Dataset Repositories . . . 1781
Read Using `hf://` Paths . . . 1782
Creating a Local Table . . . 1782
Multiple Files . . . 1783
Versioning and Revisions . . . 1784
Authentication . . . 1784
Conclusion . . . 1785

Analyzing Railway Traffic in the Netherlands **1787**
Introduction . . . 1787
Loading the Data . . . 1787
Finding the Busiest Station per Month . . . 1788
Finding the Top-3 Busiest Stations for Each Summer Month . . . 1789
Using a Window Function (`OVER`) . . . 1789
Using the `max_by(arg, val, n)` Function . . . 1789
Directly Querying Parquet Files through HTTPS or S3 . . . 1790
Largest Distance between Train Stations in the Netherlands . . . 1791
Conclusion . . . 1793

Announcing DuckDB 1.0.0 **1795**
Why Now? . . . 1795
Stability . . . 1796
Looking ahead . . . 1796
Acknowledgments . . . 1797

Native Delta Lake Support in DuckDB **1799**
Intro . . . 1799
Delta Lake . . . 1800
Implementation . . . 1802
The Delta Kernel . . . 1802
DuckDB Delta Extension `delta_scan` . . . 1803
How to Use Delta in DuckDB . . . 1805
Current State of the Delta Extension . . . 1806
Conclusion . . . 1806

Command Line Data Processing: Using DuckDB as a Unix Tool **1807**
The Unix Philosophy . . . 1807
Portability and Usability . . . 1807
Data Processing with Unix Tools and DuckDB . . . 1808
Datasets . . . 1808
Projecting Columns . . . 1808

Sorting Files . . . 1809
Intersecting Columns . . . 1810
Pasting Rows Together . . . 1810
Filtering . . . 1811
Joining Files . . . 1812
Replacing Strings . . . 1813
Reading JSON . . . 1814
Performance . . . 1814
Summary . . . 1815

20 000 Stars on GitHub **1817**
What Else Happened in June? . . . 1817
Looking Ahead . . . 1819

Benchmarking Ourselves over Time at DuckDB **1821**
Benchmark Design Summary . . . 1821
Overall Benchmark Results . . . 1822
Performance over Time . . . 1822
Performance by Version . . . 1822
Results by Workload . . . 1822
CSV Reader . . . 1822
Group By . . . 1823
Join . . . 1823
Window Functions . . . 1823
Export . . . 1823
Scan Other Formats . . . 1824
Scale tests . . . 1824
Hardware Capabilities over Time . . . 1824
Analyzing the Results Yourself . . . 1825
Conclusion . . . 1826
Appendix . . . 1826
Benchmark Design . . . 1826
Window Functions Benchmark . . . 1828

DuckDB Community Extensions **1831**
DuckDB Extensions . . . 1831
Design Philosophy . . . 1831
Using Extensions . . . 1831
DuckDB Community Extensions . . . 1832
User Experience . . . 1832
Developer Experience . . . 1832
Published Extensions . . . 1833
Summary and Looking Ahead . . . 1833

Memory Management in DuckDB **1835**
Streaming Execution . . . 1835
Intermediate Spilling . . . 1836
Buffer Manager . . . 1837
Profiling Memory Usage . . . 1837
Conclusion . . . 1838

Friendly Lists and Their Buddies, the Lambdas **1839**
Introduction . . . 1839
Lists . . . 1839
Lambdas . . . 1841

Zooming In: List Transformations . . . 1841
Pure Relational Solution . . . 1841
Native List Functions . . . 1842
Lists and Lambdas in the Community . . . 1843
`list_transform` . . . 1843
`list_filter` . . . 1844
`list_reduce` . . . 1844
Conclusion . . . 1845

DuckDB Tricks – Part 1 **1847**
Creating the Example Data Set . . . 1847
Pretty-Printing Floating-Point Numbers . . . 1848
Copying the Schema of a Table . . . 1848
Shuffling Data . . . 1849
Specifying Types in the CSV Loader . . . 1849
Updating CSV Files In-Place . . . 1850
Closing Thoughts . . . 1850

Announcing DuckDB 1.1.0 **1851**
What's New in 1.1.0 . . . 1851
Breaking SQL Changes . . . 1851
Community Extensions . . . 1852
Friendly SQL . . . 1852
Unpacked Columns . . . 1853
`query` and `query_table` Functions . . . 1853
Performance . . . 1854
Dynamic Filter Pushdown from Joins . . . 1854
Automatic CTE Materialization . . . 1855
Parallel Streaming Queries . . . 1855
Parallel `union_by_name` . . . 1855
Nested ART Rework (Foreign Key Load Speed-Up) . . . 1856
Window Function Improvements . . . 1856
Spatial Features . . . 1857
GeoParquet . . . 1857
R-Tree . . . 1857
Final Thoughts . . . 1858

Changing Data with Confidence and ACID **1859**
Atomicity . . . 1859
Consistency . . . 1860
Isolation . . . 1860
Durability . . . 1861
Why ACID in OLAP? . . . 1862
Full TPC-H Benchmark Implementation . . . 1863
Metrics . . . 1864
ACID Tests . . . 1864
Conclusion . . . 1864

Creating a SQL-Only Extension for Excel-Style Pivoting in DuckDB **1865**
The Power of SQL-Only Extensions . . . 1865
Reusability . . . 1865
Community Extensions as a Central Repository . . . 1865
Powerful SQL . . . 1866

Create Your Own SQL Extension . . . 1866
Writing the Extension . . . 1866
Testing the Extension . . . 1868
Uploading to the Community Extensions Repository . . . 1868
Capabilities of the `pivot_table` Extension . . . 1869
Example Using `pivot_table` . . . 1869
How the `pivot_table` Extension Works . . . 1871
Building Block Scalar Functions . . . 1871
Functions Creating During Refactoring for Modularity . . . 1871
Core Pivoting Logic Functions . . . 1871
The `build_my_enum` Function . . . 1872
The `pivot_table` Function . . . 1872
Conclusion . . . 1875

DuckDB in Python in the Browser with Pyodide, PyScript, and JupyterLite **1877**
Time to "Hello World" . . . 1877
Difficulties of Server-Side Python . . . 1877
Enter Pyodide . . . 1877
Use Cases for Pyodide DuckDB . . . 1878
Pyodide Examples . . . 1878
PyScript Editor . . . 1878
JupyterLite Notebook . . . 1879
JupyterLite Lab IDE . . . 1879
Architecture of DuckDB in Pyodide . . . 1879
Limitations . . . 1879
Conclusion . . . 1880

DuckDB User Survey Analysis **1881**
Summary . . . 1881
Using DuckDB . . . 1881
Environments . . . 1881
Clients . . . 1882
Operating Systems . . . 1882
Server Types . . . 1882
Data . . . 1883
Data Formats . . . 1883
Dataset Sizes . . . 1884
Features . . . 1884
Most Liked Features . . . 1884
Feature Requests . . . 1885
User Roles . . . 1885
Conclusion . . . 1885

Analyzing Open Government Data with duckplyr **1887**

DuckDB Tricks – Part 2 **1893**
Overview . . . 1893
Dataset . . . 1893
Fixing Timestamps in CSV Files . . . 1893
Filling in Missing Values . . . 1894
Repeated Data Transformation Steps . . . 1895
Computing Checksums for Columns . . . 1896
Creating a Macro for the Checksum Query . . . 1897
Closing Thoughts . . . 1897

Driving CSV Performance: Benchmarking DuckDB with the NYC Taxi Dataset **1899**
The Taxi Data Set as CSV Files . . . 1899
Reproducing the Benchmark . . . 1899
Preparing the Dataset . . . 1900
Loading . . . 1900
Querying . . . 1900
Results . . . 1900
Loading Time . . . 1900
Under the Hood . . . 1901
Query Times . . . 1902
How This Dataset Was Generated . . . 1902
How Does This Dataset Differ from the Original One? . . . 1903
Conclusion . . . 1903

What's New in the Vector Similarity Search Extension? **1905**
Indexing Speed Improvements . . . 1905
New Distance Functions . . . 1905
Index Accelerated ”Top-K” Aggregates . . . 1906
Index Accelerated LATERAL Joins . . . 1906
Conclusion . . . 1907

Fast Top N Aggregation and Filtering with DuckDB **1909**
Introduction to Top N . . . 1909
Traditional Top N by Group . . . 1910
Top N in DuckDB . . . 1911
Top N by Column in DuckDB . . . 1911
Top N by Group in DuckDB . . . 1911
The Final Top N by Group Query . . . 1913
Performance Comparisons . . . 1914
Conclusion . . . 1914

Analytics-Optimized Concurrent Transactions **1917**
Concurrency Control . . . 1917
Multi-Version Concurrency Control . . . 1917
Efficient MVCC for Analytics . . . 1918
Benchmarks . . . 1919
Write-Ahead Logging and Checkpointing . . . 1920
More Experiments . . . 1921
Conclusion . . . 1922

Optimizers: The Low-Key MVP **1923**
Normal Queries vs. Optimized Queries . . . 1923
Hand-Optimized Queries . . . 1926
Optimizations That Are Impossible by Hand . . . 1928
Summary of All Optimizers . . . 1928
Expression Rewriter . . . 1928
Filter Pull-Up & Filter Pushdown . . . 1929
IN Clause Rewriter . . . 1929
Join Order Optimizer . . . 1929
Statistics Propagation . . . 1929
Reorder Filters . . . 1930
Conclusion . . . 1930

Runtime-Extensible SQL Parsers Using PEG **1931**
Parsing Expression Grammar . . . 1932

Proof-of-Concept Experiments . . . 1932
Adding the UNPIVOT Statement . . . 1934
Extending SELECT with GRAPH_TABLE . . . 1934
Better Error Messages . . . 1935
Conclusion and Future Work . . . 1936
Acknowledgments . . . 1936

DuckDB Tricks – Part 3 **1937**
Overview . . . 1937
Dataset . . . 1937
Excluding Columns from a Table . . . 1937
Renaming Columns with Pattern Matching . . . 1938
Loading with Globbing . . . 1939
Reordering Parquet Files . . . 1940
Hive Partitioning . . . 1941
Closing Thoughts . . . 1942

CSV Files: Dethroning Parquet as the Ultimate Storage File Format — or Not? **1943**
File Formats . . . 1943
CSV Files . . . 1943
Parquet Files . . . 1943
Reading CSV Files in DuckDB . . . 1944
Comparing CSV and Parquet . . . 1944
Usability . . . 1944
Performance . . . 1945
Conclusion . . . 1948

DuckDB: Running TPC-H SF100 on Mobile Phones **1949**
A Song of Dry Ice and Fire . . . 1949
Do Androids Dream of Electric Ducks? . . . 1952
Never Was a Cloudy Day . . . 1953
Summary of DuckDB Results . . . 1954
Historical Context . . . 1954
Conclusion . . . 1957

The DuckDB Avro Extension **1959**
The Apache™ Avro™ Format . . . 1959
Header Block . . . 1959
Data Blocks . . . 1960
The DuckDB avro Community Extension . . . 1960
Installation & Loading . . . 1960
The read_avro Function . . . 1960
File IO . . . 1960
Schema Conversion . . . 1961
Implementation . . . 1961
Limitations & Next Steps . . . 1961
Conclusion . . . 1961

25 000 Stars on GitHub **1963**

DuckDB Node Neo Client **1965**
What Does It Offer? . . . 1965
Friendly, Modern API . . . 1965
Full Data Type Support . . . 1965
Advanced Features . . . 1966

How Is It Built? . . . 1967
Dependencies . . . 1967
Packages . . . 1968
Layers . . . 1968
Where Is It Headed? . . . 1968
What Now? . . . 1969

Vertical Stacking as the Relational Model Intended: UNION ALL BY NAME **1971**
Overview . . . 1971
Making Vertical Stacking Groovy Again . . . 1971
`UNION` vs. `UNION ALL` . . . 1972
Reading Multiple Files . . . 1972
Data Lakes . . . 1973
Inserting Data by Name . . . 1973
The Inspirations for `UNION ALL BY NAME` . . . 1974
Improved Performance in DuckDB 1.1 . . . 1974
Micro-Benchmark . . . 1974
Closing Thoughts . . . 1976

TPC-H SF300 on a Raspberry Pi **1977**
Introduction . . . 1977
Setup . . . 1977
Experiments . . . 1978
Aggregated Runtimes . . . 1979
Individual Query Runtimes . . . 1979
Perspective . . . 1979
Summary . . . 1979

Query Engines: Gatekeepers of the Parquet File Format **1981**
The Apache® Parquet™ Format . . . 1981
Updates . . . 1981
Encodings . . . 1982
Wasted Bits . . . 1983
Conclusion . . . 1983

Announcing DuckDB 1.2.0 **1985**
What's New in 1.2.0 . . . 1985
Breaking Changes . . . 1985
Explicit Storage Versions . . . 1985
Indexing . . . 1986
CSV Features . . . 1986
Parquet Features . . . 1987
CLI Improvements . . . 1987
Friendly SQL . . . 1987
Optimizations . . . 1988
C API for Extensions . . . 1988
musl Extensions . . . 1988
Final Thoughts . . . 1988

Catching up with Windowing **1991**
Background . . . 1991
GROUPS Framing . . . 1991
Frame Exclusion . . . 1992
`QUALIFY` Clause . . . 1992
Aggregate Modifiers . . . 1993

Function Modifiers . . . 1994
Conclusion . . . 1994

Flying Through Windows **1995**
Introduction . . . 1995
Segment Tree Vectorization . . . 1995
Constant Aggregation . . . 1996
Streaming Windows . . . 1996
Partition Major Evaluation . . . 1998
Out of Memory Operation . . . 2002
Shared Expressions . . . 2002
Future Work . . . 2002
Conclusion . . . 2003

Planning AsOf Joins **2005**
Introduction . . . 2005
What's the Plan? . . . 2005
Let's Get Small . . . 2005
Simple Joins . . . 2006
Grouping . . . 2006
Streaming Window . . . 2006
Coming Together . . . 2006
Memory . . . 2008
Backup Plans . . . 2008
Setting . . . 2009
Roll Your Own . . . 2009
Future Work . . . 2010
Conclusion . . . 2010
Notes . . . 2010

Prefix Aliases in SQL **2011**
Syntax . . . 2011
Aliases . . . 2011
Prefix Aliases in DuckDB . . . 2012
Credits . . . 2012

Reading and Writing Google Sheets in DuckDB **2013**
Spreadsheets Are Everywhere . . . 2013
Getting Started with the GSheets Extension . . . 2013
Examples of Reading from Sheets . . . 2014
Reading a Specific Sheet and Range . . . 2014
Data Types . . . 2015
Examples of Writing to a GSheet . . . 2015
Writing to a Specific Sheet and Range . . . 2015
Overwriting or Appending . . . 2016
Automated Workflows . . . 2016
Developing the Extension . . . 2016
Roadmap . . . 2017
Closing Thoughts . . . 2017

Gems of DuckDB 1.2 **2019**
New Clients Page . . . 2019
Simpler Installation . . . 2019
Installation Script on Linux and macOS . . . 2019
Signed Binaries on Windows . . . 2019

Unsung DuckDB 1.2 Features . . . 2019
OR / IN Filter Pushdown . . . 2020
-f Command Line Flag . . . 2020
allowed_directories / allowed_paths Options . . . 2020
sum(BOOLEAN) . . . 2020
Excel Extension . . . 2020
On the Interweb . . . 2020
Awesome DuckDB . . . 2021
DuckDB File Signature . . . 2021
Parquet Information Sheet . . . 2021

Parquet Bloom Filters in DuckDB **2023**
Parquet Bloom Filters . . . 2023
DuckDB Bloom Filters . . . 2024
Writing . . . 2024
Reading . . . 2024
Example Use Case . . . 2024
Conclusion . . . 2026

The DuckDB Local UI **2027**
Introducing the DuckDB UI . . . 2027
Features . . . 2027
Databases . . . 2027
Table Summary . . . 2028
Notebooks . . . 2028
Column Explorer . . . 2028
MotherDuck Integration . . . 2028
…And More! . . . 2028
Footprint . . . 2028
Internals . . . 2028
Summary . . . 2029
Update (2025-03-20) . . . 2029

Preview: Amazon S3 Tables in DuckDB **2031**
Iceberg Ahead! . . . 2031
Using Apache Iceberg REST Catalogs in DuckDB . . . 2031
Steps for Installing . . . 2031
Setting up an Amazon S3 Table Bucket . . . 2032
Populating an Amazon S3 Table Bucket . . . 2032
Reading Amazon S3 Tables with DuckDB . . . 2032
Schema Evolution . . . 2033
Conclusion . . . 2034
Footnotes . . . 2034
Cleaning Up . . . 2034
Using the core_nightly Repository . . . 2034

Maximizing Your Delta Scan Performance in DuckDB **2035**
Overview . . . 2035
The Delta Open Table Format . . . 2035
The delta Extension . . . 2035
Performance Improvements between delta v0.1.0 and 0.3.0 . . . 2036
Benchmark Setup . . . 2036
Analysis . . . 2036

Further Optimizations . . . 2036
Attaching Delta Tables . . . 2036
File Skipping . . . 2038
Partition Information Pushdown . . . 2040
A Note on Pushdown / ATTACH Performance Interplay . . . 2042
Conclusion . . . 2042

Using DuckDB in Streamlit **2043**
Introduction . . . 2043
Connecting to DuckDB in Streamlit . . . 2043
In-Memory Connection . . . 2044
Persisted Local File Connection . . . 2044
Attaching an External Database . . . 2044
Considerations . . . 2044
Analyzing Dutch Railway Data . . . 2045
Top 5 Busiest Train Stations per Month . . . 2046
Railway Network Utilization Analysis . . . 2046
Finding the Closest 5 Stations with Folium . . . 2048
Conclusion . . . 2049

Fully Local Data Transformation with dbt and DuckDB **2051**
Introduction . . . 2051
Data Model . . . 2051
Processing Data with DuckDB and dbt . . . 2051
Loading Data to DuckDB . . . 2052
Exporting Data from DuckDB . . . 2053
External Files . . . 2053
PostgreSQL . . . 2054
Execution Details . . . 2056
Conclusion . . . 2057

DuckDB's CSV Reader and the Pollock Robustness Benchmark: Into the CSV Abyss **2059**
The Sorry State of the CSV Landscape . . . 2060
DuckDB's CSV Parser . . . 2061
Running Example . . . 2061
Loading the Example Naïvely . . . 2062
Loading the Example with a User-Defined CSV Dialect . . . 2062
Disabling Strict Mode . . . 2063
Ignoring Errors . . . 2064
Null Padding for Missing Values . . . 2064
Minimal Configuration to Read the Example File . . . 2065
The Pollock Benchmark . . . 2065
Common Errors . . . 2066
Methodology . . . 2066
Pollock Scores . . . 2066
Contributing to the Benchmark . . . 2067

Temporal Analysis with Stream Windowing Functions in DuckDB **2069**
Introduction . . . 2069
Tumbling Windows . . . 2069
Hopping Windows . . . 2071
Sliding Windows . . . 2072
Session Windows . . . 2072
Conclusion . . . 2075

Sorting on Insert for Fast Selective Queries **2077**
Use Cases . . . 2077
How Does the DuckDB Format Help? . . . 2077
Using Zone Maps to Skip Reading Data . . . 2079
Strategically Skipping Data . . . 2079
Sorting Best Practices . . . 2079
Additional Techniques . . . 2080
Avoid Small Inserts . . . 2080
Sort in Chunks . . . 2080
Sort the First Few Characters of Strings . . . 2080
Filter by More Columns . . . 2081
Adjust the Row Group Size . . . 2081
Sneak Peak of the Next Post in the Series . . . 2081
Conclusion . . . 2081

Machine Learning Prototyping with DuckDB and scikit-learn **2083**
Introduction . . . 2083
Data Preparation . . . 2083
Model Training . . . 2084
Inference with DuckDB . . . 2085
With Pandas . . . 2085
DuckDB Python UDF, Row by Row . . . 2086
DuckDB Python UDF, Batch Style . . . 2087
Performance Considerations . . . 2088
Conclusion . . . 2089

The Lost Decade of Small Data? **2091**
Software . . . 2092
Benchmarks . . . 2092
2023 Improvements . . . 2094
Reproducibility . . . 2095
Discussion . . . 2095

Announcing DuckDB 1.3.0 **2097**
Breaking Changes and Deprecations . . . 2097
Old Linux glibc Deprecation . . . 2097
Lambda Function Syntax . . . 2097
Serializing Strings in List with Escapes . . . 2098
Minor SQL Parser Changes . . . 2098
New Features . . . 2098
External File Cache . . . 2098
Directly Query Data Files with the CLI . . . 2099
TRY Expression . . . 2099
Updatings Structs . . . 2100
Swapping in New Databases . . . 2100
UUID v7 Support . . . 2100
Expression Support in `CREATE SECRET` . . . 2101
Unpacking Columns . . . 2101
Spatial `JOIN` Operator . . . 2102
Internal Changes . . . 2103
Final Thoughts . . . 2103

Arrow IPC Support in DuckDB **2105**
DuckDB and Arrow . . . 2105

Arrow Interprocess Communication (Arrow IPC) . . . 2105
The Arrow Community Extension . . . 2106
Demo . . . 2106
Bonus: Cool Use Cases for Arrow IPC . . . 2108
Conclusion & What's Next . . . 2109

USING KEY in Recursive CTEs **2111**
Assembling SQL Queries from Pieces: CTEs . . . 2111
Iterate Like It's 1999: Recursive CTEs . . . 2111
Recursive CTEs Suffer from Amnesia . . . 2112
USING a KEY to Cure Amnesia . . . 2113
Overwriting Old Intermediate Results . . . 2113
A Change of Key . . . 2114
Accessing Relevant History . . . 2115
USING KEY Can Unlock Performance Advantages . . . 2115

DuckLake: SQL as a Lakehouse Format **2119**
Background . . . 2119
Iceberg and Delta . . . 2119
Catalogs . . . 2119
A Database You Say? . . . 2120
DuckLake . . . 2120
Simplicity . . . 2121
Scalability . . . 2121
Speed . . . 2121
Features . . . 2122
The ducklake DuckDB Extension . . . 2123
Installation . . . 2123
Usage . . . 2123
Press Inquiries . . . 2125

Faster Dashboards with Multi-Column Approximate Sorting **2127**
The Strategy . . . 2128
Space Filling Curves . . . 2128
Truncated Timestamps . . . 2128
The “On-Time Flights” Dataset . . . 2128
Experimental Design . . . 2129
Experimental Results . . . 2129
Approximate Time Sorting . . . 2130
Table Creation Time . . . 2130
Measuring Sortedness . . . 2131
Conclusion . . . 2131

30 000 Stars on GitHub **2133**
Recent Updates . . . 2133
Metrics . . . 2134
Events . . . 2134
Closing Thoughts . . . 2134

Lightweight Text Analytics Workflows with DuckDB **2135**
Introduction . . . 2135
Data Preparation . . . 2135
Keyword Search . . . 2136
Full-Text Search . . . 2137

Semantic Search . . . 2138
Similarity Joins . . . 2140
Hybrid Search . . . 2140
Conclusion . . . 2141

Discovering DuckDB Use Cases via GitHub **2143**
Introduction . . . 2143
Data Retrieval from GitHub . . . 2143
Saving Data to a Markdown File . . . 2145
Automating with GitHub Workflow . . . 2148
Visualizing Historical Data . . . 2149
Conclusion . . . 2150

Acknowledgments **2151**

Summary

This document contains DuckDB's official documentation and guides in a single-file easy-to-search form. If you find any issues, please report them as a GitHub issue. Contributions are very welcome in the form of pull requests. If you are considering submitting a contribution to the documentation, please consult our contributor guide.

Code repositories:

- DuckDB source code: github.com/duckdb/duckdb
- DuckDB documentation source code: github.com/duckdb/duckdb-web

Connect

Connect

Connect or Create a Database

To use DuckDB, you must first create a connection to a database. The exact syntax varies between the client APIs but it typically involves passing an argument to configure persistence.

Persistence

DuckDB can operate in both persistent mode, where the data is saved to disk, and in in-memory mode, where the entire data set is stored in the main memory.

> **Tip.** Both persistent and in-memory databases use spilling to disk to facilitate larger-than-memory workloads (i.e., out-of-core-processing).

Persistent Database

To create or open a persistent database, set the path of the database file, e.g., `my_database.duckdb`, when creating the connection. This path can point to an existing database or to a file that does not yet exist and DuckDB will open or create a database at that location as needed. The file may have an arbitrary extension, but `.db` or `.duckdb` are two common choices with `.ddb` also used sometimes.

Starting with v0.10, DuckDB's storage format is backwards-compatible, i.e., DuckDB is able to read database files produced by an older versions of DuckDB. For example, DuckDB v0.10 can read and operate on files created by the previous DuckDB version, v0.9. For more details on DuckDB's storage format, see the storage page.

In-Memory Database

DuckDB can operate in in-memory mode. In most clients, this can be activated by passing the special value `:memory:` as the database file or omitting the database file argument. In in-memory mode, no data is persisted to disk, therefore, all data is lost when the process finishes.

Concurrency

Handling Concurrency

DuckDB has two configurable options for concurrency:

1. One process can both read and write to the database.
2. Multiple processes can read from the database, but no processes can write (`access_mode = 'READ_ONLY'`).

When using option 1, DuckDB supports multiple writer threads using a combination of MVCC (Multi-Version Concurrency Control) and optimistic concurrency control (see Concurrency within a Single Process), but all within that single writer process. The reason for this concurrency model is to allow for the caching of data in RAM for faster analytical queries, rather than going back and forth to disk during each query. It also allows the caching of functions pointers, the database catalog, and other items so that subsequent queries on the same connection are faster.

> DuckDB is optimized for bulk operations, so executing many small transactions is not a primary design goal.

Concurrency within a Single Process

DuckDB supports concurrency within a single process according to the following rules. As long as there are no write conflicts, multiple concurrent writes will succeed. Appends will never conflict, even on the same table. Multiple threads can also simultaneously update separate tables or separate subsets of the same table. Optimistic concurrency control comes into play when two threads attempt to edit (update or delete) the same row at the same time. In that situation, the second thread to attempt the edit will fail with a conflict error.

Writing to DuckDB from Multiple Processes

Writing to DuckDB from multiple processes is not supported automatically and is not a primary design goal (see Handling Concurrency).

If multiple processes must write to the same file, several design patterns are possible, but would need to be implemented in application logic. For example, each process could acquire a cross-process mutex lock, then open the database in read/write mode and close it when the query is complete. Instead of using a mutex lock, each process could instead retry the connection if another process is already connected to the database (being sure to close the connection upon query completion). Another alternative would be to do multi-process transactions on a MySQL, PostgreSQL, or SQLite database, and use DuckDB's MySQL, PostgreSQL, or SQLite extensions to execute analytical queries on that data periodically.

Additional options include writing data to Parquet files and using DuckDB's ability to read multiple Parquet files, taking a similar approach with CSV files, or creating a web server to receive requests and manage reads and writes to DuckDB.

Optimistic Concurrency Control

DuckDB uses optimistic concurrency control, an approach generally considered to be the best fit for read-intensive analytical database systems as it speeds up read query processing. As a result any transactions that modify the same rows at the same time will cause a transaction conflict error:

```
Transaction conflict: cannot update a table that has been altered!
```

> **Tip.** A common workaround when a transaction conflict is encountered is to rerun the transaction.

Data Import

Importing Data

The first step to using a database system is to insert data into that system. DuckDB can directly connect to many popular data sources and offers several data ingestion methods that allow you to easily and efficiently fill up the database. On this page, we provide an overview of these methods so you can select which one is best suited for your use case.

`INSERT` Statements

`INSERT` statements are the standard way of loading data into a database system. They are suitable for quick prototyping, but should be avoided for bulk loading as they have significant per-row overhead.

```
INSERT INTO people VALUES (1, 'Mark');
```

For a more detailed description, see the page on the `INSERT` statement.

File Loading: Relative Paths

Use the configuration option `file_search_path` to configure to which "root directories" relative paths are expanded on. If `file_search_path` is not set, the working directory is used as the basis for relative paths.

File Formats

CSV Loading

Data can be efficiently loaded from CSV files using several methods. The simplest is to use the CSV file's name:

```
SELECT * FROM 'test.csv';
```

Alternatively, use the `read_csv` function to pass along options:

```
SELECT * FROM read_csv('test.csv', header = false);
```

Or use the COPY statement:

```
COPY tbl FROM 'test.csv' (HEADER false);
```

It is also possible to read data directly from **compressed CSV files** (e.g., compressed with gzip):

```
SELECT * FROM 'test.csv.gz';
```

DuckDB can create a table from the loaded data using the `CREATE TABLE ... AS SELECT` statement:

```
CREATE TABLE test AS
    SELECT * FROM 'test.csv';
```

For more details, see the page on CSV loading.

Parquet Loading

Parquet files can be efficiently loaded and queried using their filename:

```
SELECT * FROM 'test.parquet';
```

Alternatively, use the `read_parquet` function:

```
SELECT * FROM read_parquet('test.parquet');
```

Or use the COPY statement:

```
COPY tbl FROM 'test.parquet';
```

For more details, see the page on Parquet loading.

JSON Loading

JSON files can be efficiently loaded and queried using their filename:

```
SELECT * FROM 'test.json';
```

Alternatively, use the `read_json_auto` function:

```
SELECT * FROM read_json_auto('test.json');
```

Or use the COPY statement:

```
COPY tbl FROM 'test.json';
```

For more details, see the page on JSON loading.

Returning the Filename

Since DuckDB v1.3.0, the CSV, JSON and Parquet readers support the `filename` virtual column:

```
COPY (FROM (VALUES (42), (43)) t(x)) TO 'test.parquet';
SELECT *, filename FROM 'test.parquet';
```

Appender

In several APIs (C, C++, Go, Java, and Rust), the Appender can be used as an alternative for bulk data loading. This class can be used to efficiently add rows to the database system without using SQL statements.

Data Sources

DuckDB sources several data sources, including file formats, network protocols, and database systems:

- AWS S3 buckets and storage with S3-compatible API
- Azure Blob Storage
- Blob files
- Cloudflare R2
- CSV
- Delta Lake
- Excel
- httpfs
- Iceberg
- JSON
- MySQL
- Parquet
- PostgreSQL
- SQLite
- Text files

CSV Files

CSV Import

Examples

The following examples use the `flights.csv` file.

Read a CSV file from disk, auto-infer options:

```
SELECT * FROM 'flights.csv';
```

Use the `read_csv` function with custom options:

```
SELECT *
FROM read_csv('flights.csv',
    delim = '|',
    header = true,
    columns = {
        'FlightDate': 'DATE',
        'UniqueCarrier': 'VARCHAR',
        'OriginCityName': 'VARCHAR',
        'DestCityName': 'VARCHAR'
    });
```

Read a CSV from stdin, auto-infer options:

```
cat flights.csv | duckdb -c "SELECT * FROM read_csv('/dev/stdin')"
```

Read a CSV file into a table:

```
CREATE TABLE ontime (
    FlightDate DATE,
    UniqueCarrier VARCHAR,
    OriginCityName VARCHAR,
    DestCityName VARCHAR
);
COPY ontime FROM 'flights.csv';
```

Alternatively, create a table without specifying the schema manually using a `CREATE TABLE .. AS SELECT` statement:

```
CREATE TABLE ontime AS
    SELECT * FROM 'flights.csv';
```

We can use the FROM-first syntax to omit `SELECT *`.

```
CREATE TABLE ontime AS
    FROM 'flights.csv';
```

CSV Loading

CSV loading, i.e., importing CSV files to the database, is a very common, and yet surprisingly tricky, task. While CSVs seem simple on the surface, there are a lot of inconsistencies found within CSV files that can make loading them a challenge. CSV files come in many different varieties, are often corrupt, and do not have a schema. The CSV reader needs to cope with all of these different situations.

The DuckDB CSV reader can automatically infer which configuration flags to use by analyzing the CSV file using the CSV sniffer. This will work correctly in most situations, and should be the first option attempted. In rare situations where the CSV reader cannot figure out the correct configuration it is possible to manually configure the CSV reader to correctly parse the CSV file. See the auto detection page for more information.

Parameters

Below are parameters that can be passed to the `read_csv` function. Where meaningfully applicable, these parameters can also be passed to the COPY statement.

Name	Description	Type	Default
`all_varchar`	Skip type detection and assume all columns are of type VARCHAR. This option is only supported by the `read_csv` function.	`BOOL`	`false`
`allow_quoted_nulls`	Allow the conversion of quoted values to `NULL` values	`BOOL`	`true`
`auto_detect`	Auto detect CSV parameters.	`BOOL`	`true`
`auto_type_candidates`	Types that the sniffer uses when detecting column types. The VARCHAR type is always included as a fallback option. See example.	`TYPE[]`	default types
`buffer_size`	Size of the buffers used to read files, in bytes. Must be large enough to hold four lines and can significantly impact performance.	`BIGINT`	`16 * max_line_size`
`columns`	Column names and types, as a struct (e.g., `{'col1': 'INTEGER', 'col2': 'VARCHAR'}`). Using this option disables auto detection.	`STRUCT`	(empty)
`comment`	Character used to initiate comments. Lines starting with a comment character (optionally preceded by space characters) are completely ignored; other lines containing a comment character are parsed only up to that point.	VARCHAR	(empty)
`compression`	Method used to compress CSV files. By default this is detected automatically from the file extension (e.g., `t.csv.gz` will use gzip, `t.csv` will use `none`). Options are `none`, `gzip`, `zstd`.	VARCHAR	`auto`
`dateformat`	Date format used when parsing and writing dates.	VARCHAR	(empty)
`date_format`	Alias for `dateformat`; only available in the COPY statement.	VARCHAR	(empty)
`decimal_separator`	Decimal separator for numbers.	VARCHAR	`.`
`delim`	Delimiter character used to separate columns within each line, e.g., `,` `;` `\t`. The delimiter character can be up to 4 bytes, e.g., 🦆. Alias for `sep`.	VARCHAR	`,`
`delimiter`	Alias for `delim`; only available in the COPY statement.	VARCHAR	`,`
`escape`	String used to escape the `quote` character within quoted values.	VARCHAR	`"`
`encoding`	Encoding used by the CSV file. Options are `utf-8`, `utf-16`, `latin-1`. Not available in the COPY statement (which always uses `utf-8`).	VARCHAR	`utf-8`

Name	Description	Type	Default
`filename`	Add path of the containing file to each row, as a string column named `filename`. Relative or absolute paths are returned depending on the path or glob pattern provided to `read_csv`, not just filenames. Since DuckDB v1.3.0, the `filename` column is added automatically as a virtual column and this option is only kept for compatibility reasons.	`BOOL`	`false`
`force_not_null`	Do not match values in the specified columns against the `NULL` string. In the default case where the `NULL` string is empty, this means that empty values are read as zero-length strings instead of `NULL`s.	`VARCHAR[]`	`[]`
`header`	First line of each file contains the column names.	`BOOL`	`false`
`hive_partitioning`	Interpret the path as a Hive partitioned path.	`BOOL`	(auto-detected)
`ignore_errors`	Ignore any parsing errors encountered.	`BOOL`	`false`
`max_line_size` or `maximum_line_size`. Not available in the COPY statement.	Maximum line size, in bytes.	`BIGINT`	2000000
`names` or `column_names`	Column names, as a list. See example.	`VARCHAR[]`	(empty)
`new_line`	New line character(s). Options are `'\r'`,`'\n'`, or `'\r\n'`. The CSV parser only distinguishes between single-character and double-character line delimiters. Therefore, it does not differentiate between `'\r'` and `'\n'`.	`VARCHAR`	(empty)
`normalize_names`	Normalize column names. This removes any non-alphanumeric characters from them. Column names that are reserved SQL keywords are prefixed with an underscore character (_).	`BOOL`	`false`
`null_padding`	Pad the remaining columns on the right with `NULL` values when a line lacks columns.	`BOOL`	`false`
`nullstr` or `null`	Strings that represent a `NULL` value.	`VARCHAR` or `VARCHAR[]`	(empty)
`parallel`	Use the parallel CSV reader.	`BOOL`	`true`
`quote`	String used to quote values.	`VARCHAR`	`"`
`rejects_scan`	Name of the temporary table where information on faulty scans is stored.	`VARCHAR`	`reject_scans`
`rejects_table`	Name of the temporary table where information on faulty lines is stored.	`VARCHAR`	`reject_errors`
`rejects_limit`	Upper limit on the number of faulty lines per file that are recorded in the rejects table. Setting this to `0` means that no limit is applied.	`BIGINT`	`0`
`sample_size`	Number of sample lines for auto detection of parameters.	`BIGINT`	20480
`sep`	Delimiter character used to separate columns within each line, e.g., `,` `;` `\t`. The delimiter character can be up to 4 bytes, e.g., 🪿. Alias for `delim`.	`VARCHAR`	`,`
`skip`	Number of lines to skip at the start of each file.	`BIGINT`	0

Name	Description	Type	Default
`store_rejects`	Skip any lines with errors and store them in the rejects table.	BOOL	`false`
`strict_mode`	Enforces the strictness level of the CSV Reader. When set to `true`, the parser will throw an error upon encountering any issues. When set to `false`, the parser will attempt to read structurally incorrect files. It is important to note that reading structurally incorrect files can cause ambiguity; therefore, this option should be used with caution.	BOOL	`true`
`thousands`	Character used to identify thousands separators in numeric values. It must be a single character and different from the `decimal_separator` option.	VARCHAR	(empty)
`timestampformat`	Timestamp format used when parsing and writing timestamps.	VARCHAR	(empty)
`timestamp_format`	Alias for `timestampformat`; only available in the COPY statement.	VARCHAR	(empty)
`types` or `dtypes` or `column_types`	Column types, as either a list (by position) or a struct (by name). See example.	VARCHAR`[]` or STRUCT	(empty)
`union_by_name`	Align columns from different files by column name instead of position. Using this option increases memory consumption.	BOOL	`false`

> **Tip.** DuckDB's CSV reader supports UTF-8 (default), UTF-16 and Latin-1 encodings (see the `encoding` option). To convert files with different encodings, we recommend using the `iconv` command-line tool.
>
> ```
> iconv -f ISO-8859-2 -t UTF-8 input.csv > input-utf-8.csv
> ```

`auto_type_candidates` Details

The `auto_type_candidates` option lets you specify the data types that should be considered by the CSV reader for column data type detection. Usage example:

```
SELECT * FROM read_csv('csv_file.csv', auto_type_candidates = ['BIGINT', 'DATE']);
```

The default value for the `auto_type_candidates` option is `['SQLNULL', 'BOOLEAN', 'BIGINT', 'DOUBLE', 'TIME', 'DATE', 'TIMESTAMP', 'VARCHAR']`.

CSV Functions

The `read_csv` automatically attempts to figure out the correct configuration of the CSV reader using the CSV sniffer. It also automatically deduces types of columns. If the CSV file has a header, it will use the names found in that header to name the columns. Otherwise, the columns will be named `column0, column1, column2, ...`. An example with the `flights.csv` file:

```
SELECT * FROM read_csv('flights.csv');
```

FlightDate	UniqueCarrier	OriginCityName	DestCityName
1988-01-01	AA	New York, NY	Los Angeles, CA

FlightDate	UniqueCarrier	OriginCityName	DestCityName
1988-01-02	AA	New York, NY	Los Angeles, CA
1988-01-03	AA	New York, NY	Los Angeles, CA

The path can either be a relative path (relative to the current working directory) or an absolute path.

We can use `read_csv` to create a persistent table as well:

```
CREATE TABLE ontime AS
    SELECT * FROM read_csv('flights.csv');
DESCRIBE ontime;
```

column_name	column_type	null	key	default	extra
FlightDate	DATE	YES	NULL	NULL	NULL
UniqueCarrier	VARCHAR	YES	NULL	NULL	NULL
OriginCityName	VARCHAR	YES	NULL	NULL	NULL
DestCityName	VARCHAR	YES	NULL	NULL	NULL

```
SELECT * FROM read_csv('flights.csv', sample_size = 20_000);
```

If we set `delim` / `sep`, `quote`, `escape`, or `header` explicitly, we can bypass the automatic detection of this particular parameter:

```
SELECT * FROM read_csv('flights.csv', header = true);
```

Multiple files can be read at once by providing a glob or a list of files. Refer to the multiple files section for more information.

Writing Using the COPY Statement

The COPY statement can be used to load data from a CSV file into a table. This statement has the same syntax as the one used in PostgreSQL. To load the data using the COPY statement, we must first create a table with the correct schema (which matches the order of the columns in the CSV file and uses types that fit the values in the CSV file). COPY detects the CSV's configuration options automatically.

```
CREATE TABLE ontime (
    flightdate DATE,
    uniquecarrier VARCHAR,
    origincityname VARCHAR,
    destcityname VARCHAR
);
COPY ontime FROM 'flights.csv';
SELECT * FROM ontime;
```

flightdate	uniquecarrier	origincityname	destcityname
1988-01-01	AA	New York, NY	Los Angeles, CA
1988-01-02	AA	New York, NY	Los Angeles, CA
1988-01-03	AA	New York, NY	Los Angeles, CA

If we want to manually specify the CSV format, we can do so using the configuration options of COPY.

```
CREATE TABLE ontime (flightdate DATE, uniquecarrier VARCHAR, origincityname VARCHAR, destcityname
VARCHAR);
COPY ontime FROM 'flights.csv' (DELIMITER '|', HEADER);
SELECT * FROM ontime;
```

Reading Faulty CSV Files

DuckDB supports reading erroneous CSV files. For details, see the Reading Faulty CSV Files page.

Order Preservation

The CSV reader respects the `preserve_insertion_order` configuration option to preserve insertion order. When `true` (the default), the order of the rows in the result set returned by the CSV reader is the same as the order of the corresponding lines read from the file(s). When `false`, there is no guarantee that the order is preserved.

Writing CSV Files

DuckDB can write CSV files using the `COPY ... TO` statement.

CSV Auto Detection

When using `read_csv`, the system tries to automatically infer how to read the CSV file using the CSV sniffer. This step is necessary because CSV files are not self-describing and come in many different dialects. The auto-detection works roughly as follows:

- Detect the dialect of the CSV file (delimiter, quoting rule, escape)
- Detect the types of each of the columns
- Detect whether or not the file has a header row

By default the system will try to auto-detect all options. However, options can be individually overridden by the user. This can be useful in case the system makes a mistake. For example, if the delimiter is chosen incorrectly, we can override it by calling the `read_csv` with an explicit delimiter (e.g., `read_csv('file.csv', delim = '|')`).

Sample Size

The type detection works by operating on a sample of the file. The size of the sample can be modified by setting the `sample_size` parameter. The default sample size is 20,480 rows. Setting the `sample_size` parameter to `-1` means the entire file is read for sampling:

```
SELECT * FROM read_csv('my_csv_file.csv', sample_size = -1);
```

The way sampling is performed depends on the type of file. If we are reading from a regular file on disk, we will jump into the file and try to sample from different locations in the file. If we are reading from a file in which we cannot jump – such as a `.gz` compressed CSV file or `stdin` – samples are taken only from the beginning of the file.

`sniff_csv` Function

It is possible to run the CSV sniffer as a separate step using the `sniff_csv(filename)` function, which returns the detected CSV properties as a table with a single row. The `sniff_csv` function accepts an optional `sample_size` parameter to configure the number of rows sampled.

```
FROM sniff_csv('my_file.csv');
FROM sniff_csv('my_file.csv', sample_size = 1000);
```

Column name	Description	Example
`Delimiter`	Delimiter	`,`
`Quote`	Quote character	`"`
`Escape`	Escape	`\`
`NewLineDelimiter`	New-line delimiter	`\r\n`
`Comment`	Comment character	`#`
`SkipRows`	Number of rows skipped	`1`
`HasHeader`	Whether the CSV has a header	`true`
`Columns`	Column types encoded as a `LIST` of `STRUCT`s	`({'name': 'VARCHAR', 'age': 'BIGINT'})`
`DateFormat`	Date format	`%d/%m/%Y`
`TimestampFormat`	Timestamp Format	`%Y-%m-%dT%H:%M:%S.%f`
`UserArguments`	Arguments used to invoke `sniff_csv`	`sample_size = 1000`
`Prompt`	Prompt ready to be used to read the CSV	`FROM read_csv('my_file.csv', auto_detect=false, delim=',', ...)`

Prompt

The `Prompt` column contains a SQL command with the configurations detected by the sniffer.

```
-- use line mode in CLI to get the full command
.mode line
SELECT Prompt FROM sniff_csv('my_file.csv');
```

```
Prompt = FROM read_csv('my_file.csv', auto_detect=false, delim=',', quote='"', escape='"', new_
line='\n', skip=0, header=true, columns={...});
```

Detection Steps

Dialect Detection

Dialect detection works by attempting to parse the samples using the set of considered values. The detected dialect is the dialect that has (1) a consistent number of columns for each row, and (2) the highest number of columns for each row.

The following dialects are considered for automatic dialect detection.

Parameters	Considered values
`delim`	`,` `|` `;` `\t`
`quote`	`"` `'` (empty)
`escape`	`"` `'` `\` (empty)

Consider the example file `flights.csv`:

```
FlightDate|UniqueCarrier|OriginCityName|DestCityName
1988-01-01|AA|New York, NY|Los Angeles, CA
1988-01-02|AA|New York, NY|Los Angeles, CA
1988-01-03|AA|New York, NY|Los Angeles, CA
```

In this file, the dialect detection works as follows:

- If we split by a `|` every row is split into 4 columns
- If we split by a `,` rows 2-4 are split into 3 columns, while the first row is split into 1 column
- If we split by `;`, every row is split into 1 column
- If we split by `\t`, every row is split into 1 column

In this example – the system selects the `|` as the delimiter. All rows are split into the same amount of columns, and there is more than one column per row meaning the delimiter was actually found in the CSV file.

Type Detection

After detecting the dialect, the system will attempt to figure out the types of each of the columns. Note that this step is only performed if we are calling `read_csv`. In case of the COPY statement the types of the table that we are copying into will be used instead.

The type detection works by attempting to convert the values in each column to the candidate types. If the conversion is unsuccessful, the candidate type is removed from the set of candidate types for that column. After all samples have been handled – the remaining candidate type with the highest priority is chosen. The default set of candidate types is given below, in order of priority:

Types
BOOLEAN
BIGINT
DOUBLE
TIME
DATE
TIMESTAMP
VARCHAR

Everything can be cast to VARCHAR, therefore, this type has the lowest priority meaning that all columns are converted to VARCHAR if they cannot be cast to anything else. In `flights.csv` the `FlightDate` column will be cast to a DATE, while the other columns will be cast to VARCHAR.

The set of candidate types that should be considered by the CSV reader can be explicitly specified using the `auto_type_candidates` option.

In addition to the default set of candidate types, other types that may be specified using the `auto_type_candidates` options are:

Types
DECIMAL
FLOAT
INTEGER
SMALLINT
TINYINT

Even though the set of data types that can be automatically detected may appear quite limited, the CSV reader can configured to read arbitrarily complex types by using the `types`-option described in the next section.

Type detection can be entirely disabled by using the `all_varchar` option. If this is set all columns will remain as VARCHAR (as they originally occur in the CSV file).

Note that using quote characters vs. no quote characters (e.g., `"42"` and `42`) does not make a difference for type detection. Quoted fields will not be converted to VARCHAR, instead, the sniffer will try to find the type candidate with the highest priority.

Overriding Type Detection

The detected types can be individually overridden using the `types` option. This option takes either of two options:

- A list of type definitions (e.g., `types = ['INTEGER', 'VARCHAR', 'DATE']`). This overrides the types of the columns in-order of occurrence in the CSV file.
- Alternatively, `types` takes a `name` → `type` map which overrides options of individual columns (e.g., `types = {'quarter': 'INTEGER'}`).

The set of column types that may be specified using the `types` option is not as limited as the types available for the `auto_type_candidates` option: any valid type definition is acceptable to the `types`-option. (To get a valid type definition, use the `typeof()` function, or use the `column_type` column of the `DESCRIBE` result.)

The `sniff_csv()` function's `Column` field returns a struct with column names and types that can be used as a basis for overriding types.

Header Detection

Header detection works by checking if the candidate header row deviates from the other rows in the file in terms of types. For example, in `flights.csv`, we can see that the header row consists of only VARCHAR columns – whereas the values contain a `DATE` value for the `FlightDate` column. As such – the system defines the first row as the header row and extracts the column names from the header row.

In files that do not have a header row, the column names are generated as `column0`, `column1`, etc.

Note that headers cannot be detected correctly if all columns are of type VARCHAR – as in this case the system cannot distinguish the header row from the other rows in the file. In this case, the system assumes the file has a header. This can be overridden by setting the `header` option to `false`.

Dates and Timestamps

DuckDB supports the ISO 8601 format format by default for timestamps, dates and times. Unfortunately, not all dates and times are formatted using this standard. For that reason, the CSV reader also supports the `dateformat` and `timestampformat` options. Using this format the user can specify a format string that specifies how the date or timestamp should be read.

As part of the auto-detection, the system tries to figure out if dates and times are stored in a different representation. This is not always possible – as there are ambiguities in the representation. For example, the date `01-02-2000` can be parsed as either January 2nd or February 1st. Often these ambiguities can be resolved. For example, if we later encounter the date `21-02-2000` then we know that the format must have been `DD-MM-YYYY`. `MM-DD-YYYY` is no longer possible as there is no 21nd month.

If the ambiguities cannot be resolved by looking at the data the system has a list of preferences for which date format to use. If the system choses incorrectly, the user can specify the `dateformat` and `timestampformat` options manually.

The system considers the following formats for dates (`dateformat`). Higher entries are chosen over lower entries in case of ambiguities (i.e., ISO 8601 is preferred over `MM-DD-YYYY`).

dateformat
ISO 8601
%y-%m-%d
%Y-%m-%d
%d-%m-%y

dateformat
%d-%m-%Y
%m-%d-%y
%m-%d-%Y

The system considers the following formats for timestamps (`timestampformat`). Higher entries are chosen over lower entries in case of ambiguities.

timestampformat
ISO 8601
%y-%m-%d %H:%M:%S
%Y-%m-%d %H:%M:%S
%d-%m-%y %H:%M:%S
%d-%m-%Y %H:%M:%S
%m-%d-%y %I:%M:%S %p
%m-%d-%Y %I:%M:%S %p
%Y-%m-%d %H:%M:%S.%f

Reading Faulty CSV Files

CSV files can come in all shapes and forms, with some presenting many errors that make the process of cleanly reading them inherently difficult. To help users read these files, DuckDB supports detailed error messages, the ability to skip faulty lines, and the possibility of storing faulty lines in a temporary table to assist users with a data cleaning step.

Structural Errors

DuckDB supports the detection and skipping of several different structural errors. In this section, we will go over each error with an example. For the examples, consider the following table:

```
CREATE TABLE people (name VARCHAR, birth_date DATE);
```

DuckDB detects the following error types:

- `CAST`: Casting errors occur when a column in the CSV file cannot be cast to the expected schema value. For example, the line `Pedro,The 90s` would cause an error since the string `The 90s` cannot be cast to a date.
- `MISSING COLUMNS`: This error occurs if a line in the CSV file has fewer columns than expected. In our example, we expect two columns; therefore, a row with just one value, e.g., `Pedro`, would cause this error.
- `TOO MANY COLUMNS`: This error occurs if a line in the CSV has more columns than expected. In our example, any line with more than two columns would cause this error, e.g., `Pedro,01-01-1992,pdet`.
- `UNQUOTED VALUE`: Quoted values in CSV lines must always be unquoted at the end; if a quoted value remains quoted throughout, it will cause an error. For example, assuming our scanner uses `quote='"'`, the line `"pedro"holanda, 01-01-1992` would present an unquoted value error.
- `LINE SIZE OVER MAXIMUM`: DuckDB has a parameter that sets the maximum line size a CSV file can have, which by default is set to 2,097,152 bytes. Assuming our scanner is set to `max_line_size = 25`, the line `Pedro Holanda, 01-01-1992` would produce an error, as it exceeds 25 bytes.
- `INVALID ENCODING`: DuckDB supports UTF-8 strings, UTF-16 and Latin-1 encodings. Lines containing other characters will produce an error. For example, the line `pedro\xff\xff, 01-01-1992` would be problematic.

Anatomy of a CSV Error

By default, when performing a CSV read, if any structural errors are encountered, the scanner will immediately stop the scanning process and throw the error to the user. These errors are designed to provide as much information as possible to allow users to evaluate them directly in their CSV file.

This is an example for a full error message:

```
Conversion Error:
CSV Error on Line: 5648
Original Line: Pedro,The 90s
Error when converting column "birth_date". date field value out of range: "The 90s", expected format is
(DD-MM-YYYY)

Column date is being converted as type DATE
This type was auto-detected from the CSV file.
Possible solutions:
* Override the type for this column manually by setting the type explicitly, e.g., types={'birth_date':
'VARCHAR'}
* Set the sample size to a larger value to enable the auto-detection to scan more values, e.g., sample_
size=-1
* Use a COPY statement to automatically derive types from an existing table.

  file= people.csv
  delimiter = , (Auto-Detected)
  quote = " (Auto-Detected)
  escape = " (Auto-Detected)
  new_line = \r\n (Auto-Detected)
  header = true (Auto-Detected)
  skip_rows = 0 (Auto-Detected)
  date_format = (DD-MM-YYYY) (Auto-Detected)
  timestamp_format =  (Auto-Detected)
  null_padding=0
  sample_size=20480
  ignore_errors=false
  all_varchar=0
```

The first block provides us with information regarding where the error occurred, including the line number, the original CSV line, and which field was problematic:

```
Conversion Error:
CSV Error on Line: 5648
Original Line: Pedro,The 90s
Error when converting column "birth_date". date field value out of range: "The 90s", expected format is
(DD-MM-YYYY)
```

The second block provides us with potential solutions:

```
Column date is being converted as type DATE
This type was auto-detected from the CSV file.
Possible solutions:
* Override the type for this column manually by setting the type explicitly, e.g., types={'birth_date':
'VARCHAR'}
* Set the sample size to a larger value to enable the auto-detection to scan more values, e.g., sample_
size=-1
* Use a COPY statement to automatically derive types from an existing table.
```

Since the type of this field was auto-detected, it suggests defining the field as a VARCHAR or fully utilizing the dataset for type detection.

Finally, the last block presents some of the options used in the scanner that can cause errors, indicating whether they were auto-detected or manually set by the user.

Using the `ignore_errors` Option

There are cases where CSV files may have multiple structural errors, and users simply wish to skip these and read the correct data. Reading erroneous CSV files is possible by utilizing the `ignore_errors` option. With this option set, rows containing data that would otherwise cause the CSV parser to generate an error will be ignored. In our example, we will demonstrate a CAST error, but note that any of the errors described in our Structural Error section would cause the faulty line to be skipped.

For example, consider the following CSV file, `faulty.csv`:

```
Pedro,31
Oogie Boogie, three
```

If you read the CSV file, specifying that the first column is a VARCHAR and the second column is an `INTEGER`, loading the file would fail, as the string `three` cannot be converted to an `INTEGER`.

For example, the following query will throw a casting error.

```
FROM read_csv('faulty.csv', columns = {'name': 'VARCHAR', 'age': 'INTEGER'});
```

However, with `ignore_errors` set, the second row of the file is skipped, outputting only the complete first row. For example:

```
FROM read_csv(
    'faulty.csv',
    columns = {'name': 'VARCHAR', 'age': 'INTEGER'},
    ignore_errors = true
);
```

Outputs:

name	age
Pedro	31

One should note that the CSV Parser is affected by the projection pushdown optimization. Hence, if we were to select only the name column, both rows would be considered valid, as the casting error on the age would never occur. For example:

```
SELECT name
FROM read_csv('faulty.csv', columns = {'name': 'VARCHAR', 'age': 'INTEGER'});
```

Outputs:

name
Pedro
Oogie Boogie

Retrieving Faulty CSV Lines

Being able to read faulty CSV files is important, but for many data cleaning operations, it is also necessary to know exactly which lines are corrupted and what errors the parser discovered on them. For scenarios like these, it is possible to use DuckDB's CSV Rejects Table feature. By default, this feature creates two temporary tables.

1. `reject_scans`: Stores information regarding the parameters of the CSV Scanner
2. `reject_errors`: Stores information regarding each CSV faulty line and in which CSV Scanner they happened.

Note that any of the errors described in our Structural Error section will be stored in the rejects tables. Also, if a line has multiple errors, multiple entries will be stored for the same line, one for each error.

Reject Scans

The CSV Reject Scans Table returns the following information:

Column name	Description	Type
`scan_id`	The internal ID used in DuckDB to represent that scanner	`UBIGINT`
`file_id`	A scanner might happen over multiple files, so the file_id represents a unique file in a scanner	`UBIGINT`
`file_path`	The file path	`VARCHAR`
`delimiter`	The delimiter used e.g., ;	`VARCHAR`
`quote`	The quote used e.g., ”	`VARCHAR`
`escape`	The quote used e.g., ”	`VARCHAR`
`newline_delimiter`	The newline delimiter used e.g., \r\n	`VARCHAR`
`skip_rows`	If any rows were skipped from the top of the file	`UINTEGER`
`has_header`	If the file has a header	`BOOLEAN`
`columns`	The schema of the file (i.e., all column names and types)	`VARCHAR`
`date_format`	The format used for date types	`VARCHAR`
`timestamp_format`	The format used for timestamp types	`VARCHAR`
`user_arguments`	Any extra scanner parameters manually set by the user	`VARCHAR`

Reject Errors

The CSV Reject Errors Table returns the following information:

Column name	Description	Type
`scan_id`	The internal ID used in DuckDB to represent that scanner, used to join with reject scans tables	`UBIGINT`
`file_id`	The file_id represents a unique file in a scanner, used to join with reject scans tables	`UBIGINT`
`line`	Line number, from the CSV File, where the error occurred.	`UBIGINT`
`line_byte_position`	Byte Position of the start of the line, where the error occurred.	`UBIGINT`
`byte_position`	Byte Position where the error occurred.	`UBIGINT`
`column_idx`	If the error happens in a specific column, the index of the column.	`UBIGINT`
`column_name`	If the error happens in a specific column, the name of the column.	`VARCHAR`
`error_type`	The type of the error that happened.	`ENUM`
`csv_line`	The original CSV line.	`VARCHAR`
`error_message`	The error message produced by DuckDB.	`VARCHAR`

Parameters

The parameters listed below are used in the `read_csv` function to configure the CSV Rejects Table.

Name	Description	Type	Default
`store_rejects`	If set to true, any errors in the file will be skipped and stored in the default rejects temporary tables.	BOOLEAN	False
`rejects_scan`	Name of a temporary table where the information of the scan information of faulty CSV file are stored.	VARCHAR	reject_scans
`rejects_table`	Name of a temporary table where the information of the faulty lines of a CSV file are stored.	VARCHAR	reject_errors
`rejects_limit`	Upper limit on the number of faulty records from a CSV file that will be recorded in the rejects table. 0 is used when no limit should be applied.	BIGINT	0

To store the information of the faulty CSV lines in a rejects table, the user must simply set the `store_rejects` option to true. For example:

```
FROM read_csv(
    'faulty.csv',
    columns = {'name': 'VARCHAR', 'age': 'INTEGER'},
    store_rejects = true
);
```

You can then query both the `reject_scans` and `reject_errors` tables, to retrieve information about the rejected tuples. For example:

```
FROM reject_scans;
```

Outputs:

scan_id	file_id	file_path	delimiter	quote	escape	newline_delimiter	skip_rows	has_header	columns	date_format	timestamp_format	user_arguments
5	0	faulty.csv	,	”	”	\n	0	false	{'name': 'VARCHAR','age': 'INTEGER'}			store_rejects=true

```
FROM reject_errors;
```

Outputs:

scan_id	file_id	line	line_byte_position	byte_position	column_idx	column_name	error_type	csv_line	error_message
5	0	2	10	23	2	age	CAST	Oogie Boogie, three	Error when converting column ”age”. Could not convert string ” three” to 'INTEGER'

CSV Import Tips

Below is a collection of tips to help when attempting to import complex CSV files. In the examples, we use the `flights.csv` file.

Override the Header Flag if the Header Is Not Correctly Detected

If a file contains only string columns the `header` auto-detection might fail. Provide the `header` option to override this behavior.

```
SELECT * FROM read_csv('flights.csv', header = true);
```

Provide Names if the File Does Not Contain a Header

If the file does not contain a header, names will be auto-generated by default. You can provide your own names with the `names` option.

```
SELECT * FROM read_csv('flights.csv', names = ['DateOfFlight', 'CarrierName']);
```

Override the Types of Specific Columns

The `types` flag can be used to override types of only certain columns by providing a struct of `name` → `type` mappings.

```
SELECT * FROM read_csv('flights.csv', types = {'FlightDate': 'DATE'});
```

Use COPY When Loading Data into a Table

The COPY statement copies data directly into a table. The CSV reader uses the schema of the table instead of auto-detecting types from the file. This speeds up the auto-detection, and prevents mistakes from being made during auto-detection.

```
COPY tbl FROM 'test.csv';
```

Use `union_by_name` When Loading Files with Different Schemas

The `union_by_name` option can be used to unify the schema of files that have different or missing columns. For files that do not have certain columns, `NULL` values are filled in.

```
SELECT * FROM read_csv('flights*.csv', union_by_name = true);
```

To load data into *an existing table* where the table has more columns than the CSV file, you can use the `INSERT INTO ... BY NAME` clause:

```
INSERT INTO tbl BY NAME
    SELECT * FROM read_csv('input.csv');
```

Sample Size

If the CSV sniffer is not detecting the correct type, try increasing the sample size. The option `sample_size = -1` forces the sniffer to read the entire file:

```
SELECT * FROM read_csv('my_csv_file.csv', sample_size = -1);
```

JSON Files

JSON Overview

DuckDB supports SQL functions that are useful for reading values from existing JSON and creating new JSON data. JSON is supported with the `json` extension which is shipped with most DuckDB distributions and is auto-loaded on first use. If you would like to install or load it manually, please consult the "Installing and Loading" page.

About JSON

JSON is an open standard file format and data interchange format that uses human-readable text to store and transmit data objects consisting of attribute–value pairs and arrays (or other serializable values). While it is not a very efficient format for tabular data, it is very commonly used, especially as a data interchange format.

JSONPath and JSON Pointer Syntax

DuckDB implements multiple interfaces for JSON extraction: JSONPath and JSON Pointer. Both of them work with the arrow operator (`->`) and the `json_extract` function call.

Note that DuckDB only supports lookups in JSONPath, i.e., extracting fields with `.<key>` or array elements with `[<index>]`. Arrays can be indexed from the back and both approaches support the wildcard `*`. DuckDB *not* support the full JSONPath syntax because SQL is readily available for any further transformations.

> It's best to pick either the JSONPath or the JSON Pointer syntax and use it in your entire application.

Indexing

> **Warning.** Following PostgreSQL's conventions, DuckDB uses 1-based indexing for its `ARRAY` and `LIST` data types but 0-based indexing for the JSON data type.

Examples

Loading JSON

Read a JSON file from disk, auto-infer options:

```
SELECT * FROM 'todos.json';
```

Use the `read_json` function with custom options:

```
SELECT *
FROM read_json('todos.json',
               format = 'array',
               columns = {userId: 'UBIGINT',
```

```
                        id: 'UBIGINT',
                        title: 'VARCHAR',
                        completed: 'BOOLEAN'});
```

Read a JSON file from stdin, auto-infer options:

```
cat data/json/todos.json | duckdb -c "SELECT * FROM read_json('/dev/stdin')"
```

Read a JSON file into a table:

```
CREATE TABLE todos (userId UBIGINT, id UBIGINT, title VARCHAR, completed BOOLEAN);
COPY todos FROM 'todos.json' (AUTO_DETECT true);
```

Alternatively, create a table without specifying the schema manually with a `CREATE TABLE ... AS SELECT` clause:

```
CREATE TABLE todos AS
    SELECT * FROM 'todos.json';
```

Since DuckDB v1.3.0, the JSON reader returns the `filename` virtual column:

```
SELECT filename, *
FROM 'todos-*.json';
```

Writing JSON

Write the result of a query to a JSON file:

```
COPY (SELECT * FROM todos) TO 'todos.json';
```

JSON Data Type

Create a table with a column for storing JSON data and insert data into it:

```
CREATE TABLE example (j JSON);
INSERT INTO example VALUES
    ('{ "family": "anatidae", "species": [ "duck", "goose", "swan", null ] }');
```

Retrieving JSON Data

Retrieve the family key's value:

```
SELECT j.family FROM example;
```

```
"anatidae"
```

Extract the family key's value with a JSONPath expression as `JSON`:

```
SELECT j->'$.family' FROM example;
```

```
"anatidae"
```

Extract the family key's value with a JSONPath expression as a `VARCHAR`:

```
SELECT j->>'$.family' FROM example;
```

```
anatidae
```

Using Quotes for Special Characters

JSON object keys that contain the special `[` and `.` characters can be used by surrounding them with double quotes (`"`):

```
SELECT '{"d[u]._\"ck":42}'->'$."d[u]._\"ck"' AS v;
```

```
42
```

Creating JSON

JSON Creation Functions

The following functions are used to create JSON.

Function	Description
`to_json(any)`	Create JSON from a value of `any` type. Our `LIST` is converted to a JSON array, and our `STRUCT` and `MAP` are converted to a JSON object.
`json_quote(any)`	Alias for `to_json`.
`array_to_json(list)`	Alias for `to_json` that only accepts `LIST`.
`row_to_json(list)`	Alias for `to_json` that only accepts `STRUCT`.
`json_array(any, ...)`	Create a JSON array from the values in the argument lists.
`json_object(key, value, ...)`	Create a JSON object from `key`, `value` pairs in the argument list. Requires an even number of arguments.
`json_merge_patch(json, json)`	Merge two JSON documents together.

Examples:

```
SELECT to_json('duck');
```

```
"duck"
```

```
SELECT to_json([1, 2, 3]);
```

```
[1,2,3]
```

```
SELECT to_json({duck : 42});
```

```
{"duck":42}
```

```
SELECT to_json(MAP(['duck'], [42]));
```

```
{"duck":42}
```

```
SELECT json_array('duck', 42, 'goose', 123);
```

```
["duck",42,"goose",123]
```

```
SELECT json_object('duck', 42, 'goose', 123);
```

```
{"duck":42,"goose":123}
```

```
SELECT json_merge_patch('{"duck": 42}', '{"goose": 123}');
```

```
{"goose":123,"duck":42}
```

Loading JSON

The DuckDB JSON reader can automatically infer which configuration flags to use by analyzing the JSON file. This will work correctly in most situations, and should be the first option attempted. In rare situations where the JSON reader cannot figure out the correct configuration, it is possible to manually configure the JSON reader to correctly parse the JSON file.

The `read_json` Function

The `read_json` is the simplest method of loading JSON files: it automatically attempts to figure out the correct configuration of the JSON reader. It also automatically deduces types of columns. In the following example, we use the `todos.json` file,

```
SELECT *
FROM read_json('todos.json')
LIMIT 5;
```

userId	id	title	completed
1	1	delectus aut autem	false
1	2	quis ut nam facilis et officia qui	false
1	3	fugiat veniam minus	false
1	4	et porro tempora	true
1	5	laboriosam mollitia et enim quasi adipisci quia provident illum	false

We can use `read_json` to create a persistent table as well:

```
CREATE TABLE todos AS
    SELECT *
    FROM read_json('todos.json');
DESCRIBE todos;
```

column_name	column_type	null	key	default	extra
userId	UBIGINT	YES	NULL	NULL	NULL
id	UBIGINT	YES	NULL	NULL	NULL
title	VARCHAR	YES	NULL	NULL	NULL
completed	BOOLEAN	YES	NULL	NULL	NULL

If we specify types for subset of columns, `read_json` excludes columns that we don't specify:

```
SELECT *
FROM read_json(
        'todos.json',
        columns = {userId: 'UBIGINT', completed: 'BOOLEAN'}
    )
LIMIT 5;
```

Note that only the `userId` and `completed` columns are shown:

userId	completed
1	false
1	false
1	false
1	true
1	false

Multiple files can be read at once by providing a glob or a list of files. Refer to the multiple files section for more information.

Functions for Reading JSON Objects

The following table functions are used to read JSON:

Function	Description
`read_json_objects(filename)`	Read a JSON object from `filename`, where `filename` can also be a list of files or a glob pattern.
`read_ndjson_objects(filename)`	Alias for `read_json_objects` with the parameter `format` set to `newline_delimited`.
`read_json_objects_auto(filename)`	Alias for `read_json_objects` with the parameter `format` set to `auto`.

Parameters

These functions have the following parameters:

Name	Description	Type	Default
`compression`	The compression type for the file. By default this will be detected automatically from the file extension (e.g., `t.json.gz` will use gzip, `t.json` will use none). Options are `none`, `gzip`, `zstd` and `auto_detect`.	`VARCHAR`	`auto_detect`
`filename`	Whether or not an extra `filename` column should be included in the result. Since DuckDB v1.3.0, the `filename` column is added automatically as a virtual column and this option is only kept for compatibility reasons.	`BOOL`	`false`
`format`	Can be one of `auto`, `unstructured`, `newline_delimited` and `array`.	`VARCHAR`	`array`
`hive_partitioning`	Whether or not to interpret the path as a Hive partitioned path.	`BOOL`	(auto-detected)
`ignore_errors`	Whether to ignore parse errors (only possible when `format` is `newline_delimited`).	`BOOL`	`false`
`maximum_sample_files`	The maximum number of JSON files sampled for auto-detection.	`BIGINT`	`32`

Name	Description	Type	Default
`maximum_object_size`	The maximum size of a JSON object (in bytes).	`UINTEGER`	`16777216`

The `format` parameter specifies how to read the JSON from a file. With `unstructured`, the top-level JSON is read, e.g., for `birds.json`:

```
{
  "duck": 42
}
{
  "goose": [1, 2, 3]
}
```

```
FROM read_json_objects('birds.json', format = 'unstructured');
```

will result in two objects being read:

```
┌──────────────────────────────────┐
│               json               │
│               json               │
├──────────────────────────────────┤
│ {\n    "duck": 42\n}             │
│ {\n    "goose": [1, 2, 3]\n}     │
└──────────────────────────────────┘
```

With `newline_delimited`, NDJSON is read, where each JSON is separated by a newline (`\n`), e.g., for `birds-nd.json`:

```
{"duck": 42}
{"goose": [1, 2, 3]}
```

```
FROM read_json_objects('birds-nd.json', format = 'newline_delimited');
```

will also result in two objects being read:

```
┌──────────────────────┐
│         json         │
│         json         │
├──────────────────────┤
│ {"duck": 42}         │
│ {"goose": [1, 2, 3]} │
└──────────────────────┘
```

With `array`, each array element is read, e.g., for `birds-array.json`:

```
[
    {
        "duck": 42
    },
    {
        "goose": [1, 2, 3]
    }
]
```

```
FROM read_json_objects('birds-array.json', format = 'array');
```

will again result in two objects being read:

```
┌──────────────────────────────────┐
│               json               │
│               json               │
├──────────────────────────────────┤
```

```
{\n        "duck": 42\n    }
{\n        "goose": [1, 2, 3]\n    }
```

Functions for Reading JSON as a Table

DuckDB also supports reading JSON as a table, using the following functions:

Function	Description
`read_json(filename)`	Read JSON from `filename`, where `filename` can also be a list of files, or a glob pattern.
`read_json_auto(filename)`	Alias for `read_json`.
`read_ndjson(filename)`	Alias for `read_json` with parameter `format` set to `newline_delimited`.
`read_ndjson_auto(filename)`	Alias for `read_json` with parameter `format` set to `newline_delimited`.

Parameters

Besides the `maximum_object_size`, `format`, `ignore_errors` and `compression`, these functions have additional parameters:

Name	Description	Type	Default
`auto_detect`	Whether to auto-detect the names of the keys and data types of the values automatically	`BOOL`	`true`
`columns`	A struct that specifies the key names and value types contained within the JSON file (e.g., `{key1: 'INTEGER', key2: 'VARCHAR'}`). If `auto_detect` is enabled these will be inferred	`STRUCT`	`(empty)`
`dateformat`	Specifies the date format to use when parsing dates. See Date Format	`VARCHAR`	`iso`
`maximum_depth`	Maximum nesting depth to which the automatic schema detection detects types. Set to -1 to fully detect nested JSON types	`BIGINT`	`-1`
`records`	Can be one of `auto`, `true`, `false`	`VARCHAR`	`auto`
`sample_size`	Option to define number of sample objects for automatic JSON type detection. Set to -1 to scan the entire input file	`UBIGINT`	`20480`
`timestampformat`	Specifies the date format to use when parsing timestamps. See Date Format	`VARCHAR`	`iso`
`union_by_name`	Whether the schema's of multiple JSON files should be unified	`BOOL`	`false`
`map_inference_threshold`	Controls the threshold for number of columns whose schema will be auto-detected; if JSON schema auto-detection would infer a `STRUCT` type for a field that has *more* than this threshold number of subfields, it infers a `MAP` type instead. Set to `-1` to disable `MAP` inference.	`BIGINT`	`200`

Name	Description	Type	Default
`field_appearance_threshold`	The JSON reader divides the number of appearances of each JSON field by the auto-detection sample size. If the average over the fields of an object is less than this threshold, it will default to using a `MAP` type with value type of merged field types.	`DOUBLE`	`0.1`

Note that DuckDB can convert JSON arrays directly to its internal `LIST` type, and missing keys become `NULL`:

```
SELECT *
FROM read_json(
    ['birds1.json', 'birds2.json'],
    columns = {duck: 'INTEGER', goose: 'INTEGER[]', swan: 'DOUBLE'}
);
```

duck	goose	swan
42	[1, 2, 3]	NULL
43	[4, 5, 6]	3.3

DuckDB can automatically detect the types like so:

```
SELECT goose, duck FROM read_json('*.json.gz');
SELECT goose, duck FROM '*.json.gz'; -- equivalent
```

DuckDB can read (and auto-detect) a variety of formats, specified with the `format` parameter. Querying a JSON file that contains an `array`, e.g.:

```
[
  {
    "duck": 42,
    "goose": 4.2
  },
  {
    "duck": 43,
    "goose": 4.3
  }
]
```

Can be queried exactly the same as a JSON file that contains `unstructured` JSON, e.g.:

```
{
    "duck": 42,
    "goose": 4.2
}
{
    "duck": 43,
    "goose": 4.3
}
```

Both can be read as the table:

```
SELECT
FROM read_json('birds.json');
```

duck	goose
42	4.2
43	4.3

If your JSON file does not contain "records", i.e., any other type of JSON than objects, DuckDB can still read it. This is specified with the `records` parameter. The `records` parameter specifies whether the JSON contains records that should be unpacked into individual columns. DuckDB also attempts to auto-detect this. For example, take the following file, `birds-records.json`:

```
{"duck": 42, "goose": [1, 2, 3]}
{"duck": 43, "goose": [4, 5, 6]}
```

```
SELECT *
FROM read_json('birds-records.json');
```

The query results in two columns:

duck	goose
42	[1,2,3]
43	[4,5,6]

You can read the same file with `records` set to `false`, to get a single column, which is a STRUCT containing the data:

json
{'duck': 42, 'goose': [1,2,3]}
{'duck': 43, 'goose': [4,5,6]}

For additional examples reading more complex data, please see the "Shredding Deeply Nested JSON, One Vector at a Time" blog post.

Loading with the COPY Statement Using FORMAT `json`

When the `json` extension is installed, `FORMAT json` is supported for `COPY FROM`, `IMPORT DATABASE`, as well as `COPY TO` and `EXPORT DATABASE`. See the COPY statement and the `IMPORT` / `EXPORT` clauses.

By default, COPY expects newline-delimited JSON. If you prefer copying data to/from a JSON array, you can specify `ARRAY true`, e.g.,

```
COPY (SELECT * FROM range(5) r(i))
TO 'numbers.json' (ARRAY true);
```

will create the following file:

```
[
    {"i":0},
    {"i":1},
    {"i":2},
    {"i":3},
    {"i":4}
]
```

This can be read back to DuckDB as follows:

```
CREATE TABLE numbers (i BIGINT);
COPY numbers FROM 'numbers.json' (ARRAY true);
```

The format can be detected automatically the format like so:

```
CREATE TABLE numbers (i BIGINT);
COPY numbers FROM 'numbers.json' (AUTO_DETECT true);
```

We can also create a table from the auto-detected schema:

```
CREATE TABLE numbers AS
    FROM 'numbers.json';
```

Parameters

Name	Description	Type	Default
`auto_detect`	Whether to auto-detect the names of the keys and data types of the values automatically	`BOOL`	`false`
`columns`	A struct that specifies the key names and value types contained within the JSON file (e.g., `{key1: 'INTEGER', key2: 'VARCHAR'}`). If `auto_detect` is enabled these will be inferred	`STRUCT`	`(empty)`
`compression`	The compression type for the file. By default this will be detected automatically from the file extension (e.g., `t.json.gz` will use gzip, `t.json` will use none). Options are `uncompressed`, `gzip`, `zstd` and `auto_detect`.	`VARCHAR`	`auto_detect`
`convert_strings_to_integers`	Whether strings representing integer values should be converted to a numerical type.	`BOOL`	`false`
`dateformat`	Specifies the date format to use when parsing dates. See Date Format	`VARCHAR`	`iso`
`filename`	Whether or not an extra `filename` column should be included in the result.	`BOOL`	`false`
`format`	Can be one of `auto`, `unstructured`, `newline_delimited`, `array`	`VARCHAR`	`array`
`hive_partitioning`	Whether or not to interpret the path as a Hive partitioned path.	`BOOL`	`false`
`ignore_errors`	Whether to ignore parse errors (only possible when `format` is `newline_delimited`)	`BOOL`	`false`
`maximum_depth`	Maximum nesting depth to which the automatic schema detection detects types. Set to -1 to fully detect nested JSON types	`BIGINT`	`-1`
`maximum_object_size`	The maximum size of a JSON object (in bytes)	`UINTEGER`	`16777216`
`records`	Can be one of `auto`, `true`, `false`	`VARCHAR`	`records`
`sample_size`	Option to define number of sample objects for automatic JSON type detection. Set to -1 to scan the entire input file	`UBIGINT`	`20480`
`timestampformat`	Specifies the date format to use when parsing timestamps. See Date Format	`VARCHAR`	`iso`
`union_by_name`	Whether the schema's of multiple JSON files should be unified.	`BOOL`	`false`

Writing JSON

The contents of tables or the result of queries can be written directly to a JSON file using the COPY statement. For example:

```
CREATE TABLE cities AS
    FROM (VALUES ('Amsterdam', 1), ('London', 2)) cities(name, id);
COPY cities TO 'cities.json';
```

This will result in `cities.json` with the following content:

```
{"name":"Amsterdam","id":1}
{"name":"London","id":2}
```

See the COPY statement for more information.

JSON Type

DuckDB supports `json` via the JSON logical type. The JSON logical type is interpreted as JSON, i.e., parsed, in JSON functions rather than interpreted as VARCHAR, i.e., a regular string (modulo the equality-comparison caveat at the bottom of this page). All JSON creation functions return values of this type.

We also allow any of DuckDB's types to be cast to JSON, and JSON to be cast back to any of DuckDB's types, for example, to cast JSON to DuckDB's STRUCT type, run:

```
SELECT '{"duck": 42}'::JSON::STRUCT(duck INTEGER);
```

```
{'duck': 42}
```

And back:

```
SELECT {duck: 42}::JSON;
```

```
{"duck":42}
```

This works for our nested types as shown in the example, but also for non-nested types:

```
SELECT '2023-05-12'::DATE::JSON;
```

```
"2023-05-12"
```

The only exception to this behavior is the cast from VARCHAR to JSON, which does not alter the data, but instead parses and validates the contents of the VARCHAR as JSON.

JSON Processing Functions

JSON Extraction Functions

There are two extraction functions, which have their respective operators. The operators can only be used if the string is stored as the JSON logical type. These functions supports the same two location notations as JSON Scalar functions.

Function	Alias	Operator	Description
`json_exists(json, path)`			Returns `true` if the supplied path exists in the `json`, and `false` otherwise.

Function	Alias	Operator	Description
`json_extract(json, path)`	`json_extract_path`	`->`	Extracts JSON from `json` at the given `path`. If `path` is a `LIST`, the result will be a `LIST` of `JSON`.
`json_extract_string(json, path)`	`json_extract_path_text`	`->>`	Extracts `VARCHAR` from `json` at the given `path`. If `path` is a `LIST`, the result will be a `LIST` of `VARCHAR`.
`json_value(json, path)`			Extracts JSON from `json` at the given `path`. If the `json` at the supplied path is not a scalar value, it will return `NULL`.

Note that the arrow operator `->`, which is used for JSON extracts, has a low precedence as it is also used in lambda functions. Therefore, you need to surround the `->` operator with parentheses when expressing operations such as equality comparisons (`=`). For example:

```
SELECT ((JSON '{"field": 42}')->'field') = 42;
```

> **Warning.** DuckDB's JSON data type uses 0-based indexing.

Examples:

```
CREATE TABLE example (j JSON);
INSERT INTO example VALUES
    ('{ "family": "anatidae", "species": [ "duck", "goose", "swan", null ] }');
```

```
SELECT json_extract(j, '$.family') FROM example;
```

```
"anatidae"
```

```
SELECT j->'$.family' FROM example;
```

```
"anatidae"
```

```
SELECT j->'$.species[0]' FROM example;
```

```
"duck"
```

```
SELECT j->'$.species[*]' FROM example;
```

```
["duck", "goose", "swan", null]
```

```
SELECT j->>'$.species[*]' FROM example;
```

```
[duck, goose, swan, null]
```

```
SELECT j->'$.species'->0 FROM example;
```

```
"duck"
```

```
SELECT j->'species'->['/0', '/1'] FROM example;
```

```
['"duck"', '"goose"']
```

```
SELECT json_extract_string(j, '$.family') FROM example;
```

```
anatidae
```

```
SELECT j->>'$.family' FROM example;
```

```
anatidae
```

```
SELECT j->>'$.species[0]' FROM example;
```

```
duck
```

```
SELECT j->'species'->>0 FROM example;
```

```
duck
```

```
SELECT j->'species'->>['/0', '/1'] FROM example;
```

```
[duck, goose]
```

Note that DuckDB's JSON data type uses 0-based indexing.

If multiple values need to be extracted from the same JSON, it is more efficient to extract a list of paths:

The following will cause the JSON to be parsed twice,:

Resulting in a slower query that uses more memory:

```
SELECT
    json_extract(j, 'family') AS family,
    json_extract(j, 'species') AS species
FROM example;
```

family	species
”anatidae”	[”duck”,”goose”,”swan”,null]

The following produces the same result but is faster and more memory-efficient:

```
WITH extracted AS (
    SELECT json_extract(j, ['family', 'species']) AS extracted_list
    FROM example
)
SELECT
    extracted_list[1] AS family,
    extracted_list[2] AS species
FROM extracted;
```

JSON Scalar Functions

The following scalar JSON functions can be used to gain information about the stored JSON values. With the exception of `json_valid(json)`, all JSON functions produce an error when invalid JSON is supplied.

We support two kinds of notations to describe locations within JSON: JSON Pointer and JSONPath.

Function	Description
`json_array_length(json[, path])`	Return the number of elements in the JSON array `json`, or `0` if it is not a JSON array. If `path` is specified, return the number of elements in the JSON array at the given `path`. If `path` is a `LIST`, the result will be `LIST` of array lengths.
`json_contains(json_haystack, json_needle)`	Returns `true` if `json_needle` is contained in `json_haystack`. Both parameters are of JSON type, but `json_needle` can also be a numeric value or a string, however the string must be wrapped in double quotes.
`json_keys(json[, path])`	Returns the keys of `json` as a `LIST` of VARCHAR, if `json` is a JSON object. If `path` is specified, return the keys of the JSON object at the given `path`. If `path` is a `LIST`, the result will be `LIST` of `LIST` of VARCHAR.

Function	Description
`json_structure(json)`	Return the structure of `json`. Defaults to `JSON` if the structure is inconsistent (e.g., incompatible types in an array).
`json_type(json[, path])`	Return the type of the supplied `json`, which is one of `ARRAY`, `BIGINT`, `BOOLEAN`, `DOUBLE`, `OBJECT`, `UBIGINT`, `VARCHAR`, and `NULL`. If `path` is specified, return the type of the element at the given `path`. If `path` is a `LIST`, the result will be `LIST` of types.
`json_valid(json)`	Return whether `json` is valid JSON.
`json(json)`	Parse and minify `json`.

The JSONPointer syntax separates each field with a `/`. For example, to extract the first element of the array with key `duck`, you can do:

```
SELECT json_extract('{"duck": [1, 2, 3]}', '/duck/0');
```

```
1
```

The JSONPath syntax separates fields with a `.`, and accesses array elements with `[i]`, and always starts with `$`. Using the same example, we can do the following:

```
SELECT json_extract('{"duck": [1, 2, 3]}', '$.duck[0]');
```

```
1
```

Note that DuckDB's JSON data type uses 0-based indexing.

JSONPath is more expressive, and can also access from the back of lists:

```
SELECT json_extract('{"duck": [1, 2, 3]}', '$.duck[#-1]');
```

```
3
```

JSONPath also allows escaping syntax tokens, using double quotes:

```
SELECT json_extract('{"duck.goose": [1, 2, 3]}', '$."duck.goose"[1]');
```

```
2
```

Examples using the anatidae biological family:

```
CREATE TABLE example (j JSON);
INSERT INTO example VALUES
    ('{ "family": "anatidae", "species": [ "duck", "goose", "swan", null ] }');
```

```
SELECT json(j) FROM example;
```

```
{"family":"anatidae","species":["duck","goose","swan",null]}
```

```
SELECT j.family FROM example;
```

```
"anatidae"
```

```
SELECT j.species[0] FROM example;
```

```
"duck"
```

```
SELECT json_valid(j) FROM example;
```

```
true
```

```
SELECT json_valid('{');
```

```
false
```

```
SELECT json_array_length('["duck", "goose", "swan", null]');
```

```
4
```

```
SELECT json_array_length(j, 'species') FROM example;
```

```
4
```

```
SELECT json_array_length(j, '/species') FROM example;
```

```
4
```

```
SELECT json_array_length(j, '$.species') FROM example;
```

```
4
```

```
SELECT json_array_length(j, ['$.species']) FROM example;
```

```
[4]
```

```
SELECT json_type(j) FROM example;
```

```
OBJECT
```

```
SELECT json_keys(j) FROM example;
```

```
[family, species]
```

```
SELECT json_structure(j) FROM example;
```

```
{"family":"VARCHAR","species":["VARCHAR"]}
```

```
SELECT json_structure('["duck", {"family": "anatidae"}]');
```

```
["JSON"]
```

```
SELECT json_contains('{"key": "value"}', '"value"');
```

```
true
```

```
SELECT json_contains('{"key": 1}', '1');
```

```
true
```

```
SELECT json_contains('{"top_key": {"key": "value"}}', '{"key": "value"}');
```

```
true
```

JSON Aggregate Functions

There are three JSON aggregate functions.

Function	Description
`json_group_array(any)`	Return a JSON array with all values of `any` in the aggregation.
`json_group_object(key, value)`	Return a JSON object with all `key`, `value` pairs in the aggregation.
`json_group_structure(json)`	Return the combined `json_structure` of all `json` in the aggregation.

Examples:

```
CREATE TABLE example1 (k VARCHAR, v INTEGER);
INSERT INTO example1 VALUES ('duck', 42), ('goose', 7);
```

```
SELECT json_group_array(v) FROM example1;
```

```
[42, 7]
```

```
SELECT json_group_object(k, v) FROM example1;
```

```
{"duck":42,"goose":7}
```

```
CREATE TABLE example2 (j JSON);
INSERT INTO example2 VALUES
    ('{"family": "anatidae", "species": ["duck", "goose"], "coolness": 42.42}'),
    ('{"family": "canidae", "species": ["labrador", "bulldog"], "hair": true}');

SELECT json_group_structure(j) FROM example2;
```

```
{"family":"VARCHAR","species":["VARCHAR"],"coolness":"DOUBLE","hair":"BOOLEAN"}
```

Transforming JSON to Nested Types

In many cases, it is inefficient to extract values from JSON one-by-one. Instead, we can "extract" all values at once, transforming JSON to the nested types `LIST` and `STRUCT`.

Function	Description
`json_transform(json, structure)`	Transform `json` according to the specified `structure`.
`from_json(json, structure)`	Alias for `json_transform`.
`json_transform_strict(json, structure)`	Same as `json_transform`, but throws an error when type casting fails.
`from_json_strict(json, structure)`	Alias for `json_transform_strict`.

The `structure` argument is JSON of the same form as returned by `json_structure`. The `structure` argument can be modified to transform the JSON into the desired structure and types. It is possible to extract fewer key/value pairs than are present in the JSON, and it is also possible to extract more: missing keys become `NULL`.

Examples:

```
CREATE TABLE example (j JSON);
INSERT INTO example VALUES
    ('{"family": "anatidae", "species": ["duck", "goose"], "coolness": 42.42}'),
    ('{"family": "canidae", "species": ["labrador", "bulldog"], "hair": true}');

SELECT json_transform(j, '{"family": "VARCHAR", "coolness": "DOUBLE"}') FROM example;
```

```
{'family': anatidae, 'coolness': 42.420000}
{'family': canidae, 'coolness': NULL}
```

```
SELECT json_transform(j, '{"family": "TINYINT", "coolness": "DECIMAL(4, 2)"}') FROM example;
```

```
{'family': NULL, 'coolness': 42.42}
{'family': NULL, 'coolness': NULL}
```

```
SELECT json_transform_strict(j, '{"family": "TINYINT", "coolness": "DOUBLE"}') FROM example;
```

```
Invalid Input Error: Failed to cast value: "anatidae"
```

JSON Table Functions

DuckDB implements two JSON table functions that take a JSON value and produce a table from it.

Function	Description
`json_each(json[,path]`	Traverse `json` and return one row for each element in the top-level array or object.
`json_tree(json[,path]`	Traverse `json` in depth-first fashion and return one row for each element in the structure.

If the element is not an array or object, the element itself is returned. If the optional `path` argument is supplied, traversal starts from the element at the given path instead of the root element.

The resulting table has the following columns:

Field	Type	Description
`key`	`VARCHAR`	Key of element relative to its parent
`value`	`JSON`	Value of element
`type`	`VARCHAR`	`json_type` (function) of this element
`atom`	`JSON`	`json_value` (function) of this element
`id`	`UBIGINT`	Element identifier, numbered by parse order
`parent`	`UBIGINT`	`id` of parent element
`fullkey`	`VARCHAR`	JSON path to element
`path`	`VARCHAR`	JSON path to parent element
`json`	`JSON` (Virtual)	The `json` parameter
`root`	`TEXT` (Virtual)	The `path` parameter
`rowid`	`BIGINT` (Virtual)	The row identifier

These functions are analogous to SQLite's functions with the same name. Note that, because the `json_each` and `json_tree` functions refer to previous subqueries in the same FROM clause, they are *lateral joins*.

Examples:

```
CREATE TABLE example (j JSON);
INSERT INTO example VALUES
    ('{"family": "anatidae", "species": ["duck", "goose"], "coolness": 42.42}'),
    ('{"family": "canidae", "species": ["labrador", "bulldog"], "hair": true}');

SELECT je.*, je.rowid
FROM example AS e, json_each(e.j) AS je;
```

key	value	type	atom	id	parent	fullkey	path	rowid
family	”anatidae”	VARCHAR	”anatidae”	2	NULL	$.family	$	0
species	[”duck”,”goose”]	ARRAY	NULL	4	NULL	$.species	$	1
coolness	42.42	DOUBLE	42.42	8	NULL	$.coolness	$	2
family	”canidae”	VARCHAR	”canidae”	2	NULL	$.family	$	0

key	value	type	atom	id	parent	fullkey	path	rowid
species	[”labrador”,”bulldog”]	ARRAY	NULL	4	NULL	$.species	$	1
hair	true	BOOLEAN	true	8	NULL	$.hair	$	2

```
SELECT je.*, je.rowid
FROM example AS e, json_each(e.j, '$.species') AS je;
```

key	value	type	atom	id	parent	fullkey	path	rowid
0	”duck”	VARCHAR	”duck”	5	NULL	$.species[0]	$.species	0
1	”goose”	VARCHAR	”goose”	6	NULL	$.species[1]	$.species	1
0	”labrador”	VARCHAR	”labrador”	5	NULL	$.species[0]	$.species	0
1	”bulldog”	VARCHAR	”bulldog”	6	NULL	$.species[1]	$.species	1

```
SELECT je.key, je.value, je.type, je.id, je.parent, je.fullkey, je.rowid
FROM example AS e, json_tree(e.j) AS je;
```

key	value	type	id	parent	fullkey	rowid
NULL	{”family”:”anatidae”,”species”:[”duck”,”goose”],”coolness”:42.42}	OBJECT	0	NULL	$	0
family	”anatidae”	VARCHAR	2	0	$.family	1
species	[”duck”,”goose”]	ARRAY	4	0	$.species	2
0	”duck”	VARCHAR	5	4	$.species[0]	3
1	”goose”	VARCHAR	6	4	$.species[1]	4
coolness	42.42	DOUBLE	8	0	$.coolness	5
NULL	{”family”:”canidae”,”species”:[”labrador”,”bulldog”],”hair”:true}	OBJECT	0	NULL	$	0
family	”canidae”	VARCHAR	2	0	$.family	1
species	[”labrador”,”bulldog”]	ARRAY	4	0	$.species	2
0	”labrador”	VARCHAR	5	4	$.species[0]	3
1	”bulldog”	VARCHAR	6	4	$.species[1]	4
hair	true	BOOLEAN	8	0	$.hair	5

JSON Format Settings

The JSON extension can attempt to determine the format of a JSON file when setting `format` to `auto`. Here are some example JSON files and the corresponding `format` settings that should be used.

In each of the below cases, the `format` setting was not needed, as DuckDB was able to infer it correctly, but it is included for illustrative purposes. A query of this shape would work in each case:

```
SELECT *
FROM filename.json;
```

Format: `newline_delimited`

With `format = 'newline_delimited'` newline-delimited JSON can be parsed. Each line is a JSON.

We use the example file `records.json` with the following content:

```
{"key1":"value1", "key2": "value1"}
{"key1":"value2", "key2": "value2"}
{"key1":"value3", "key2": "value3"}
```

```
SELECT *
FROM read_json('records.json', format = 'newline_delimited');
```

key1	key2
value1	value1
value2	value2
value3	value3

Format: `array`

If the JSON file contains a JSON array of objects (pretty-printed or not), `array_of_objects` may be used. To demonstrate its use, we use the example file `records-in-array.json`:

```
[
    {"key1":"value1", "key2": "value1"},
    {"key1":"value2", "key2": "value2"},
    {"key1":"value3", "key2": "value3"}
]
```

```
SELECT *
FROM read_json('records-in-array.json', format = 'array');
```

key1	key2
value1	value1
value2	value2
value3	value3

Format: `unstructured`

If the JSON file contains JSON that is not newline-delimited or an array, `unstructured` may be used. To demonstrate its use, we use the example file `unstructured.json`:

```
{
    "key1":"value1",
    "key2":"value1"
}
{
    "key1":"value2",
    "key2":"value2"
}
{
    "key1":"value3",
    "key2":"value3"
}
```

```
SELECT *
FROM read_json('unstructured.json', format = 'unstructured');
```

key1	key2
value1	value1
value2	value2
value3	value3

Records Settings

The JSON extension can attempt to determine whether a JSON file contains records when setting `records = auto`. When `records = true`, the JSON extension expects JSON objects, and will unpack the fields of JSON objects into individual columns.

Continuing with the same example file, `records.json`:

```
{"key1":"value1", "key2": "value1"}
{"key1":"value2", "key2": "value2"}
{"key1":"value3", "key2": "value3"}
```

```
SELECT *
FROM read_json('records.json', records = true);
```

key1	key2
value1	value1
value2	value2
value3	value3

When `records = false`, the JSON extension will not unpack the top-level objects, and create STRUCTs instead:

```
SELECT *
FROM read_json('records.json', records = false);
```

json
{'key1': value1, 'key2': value1}
{'key1': value2, 'key2': value2}
{'key1': value3, 'key2': value3}

This is especially useful if we have non-object JSON, for example, `arrays.json`:

```
[1, 2, 3]
[4, 5, 6]
[7, 8, 9]
```

```
SELECT *
FROM read_json('arrays.json', records = false);
```

json
[1, 2, 3]
[4, 5, 6]
[7, 8, 9]

json

Installing and Loading the JSON extension

The `json` extension is shipped by default in DuckDB builds, otherwise, it will be transparently autoloaded on first use. If you would like to install and load it manually, run:

```
INSTALL json;
LOAD json;
```

SQL to/from JSON

DuckDB provides functions to serialize and deserialize `SELECT` statements between SQL and JSON, as well as executing JSON serialized statements.

Function	Type	Description
`json_deserialize_sql(json)`	Scalar	Deserialize one or many `json` serialized statements back to an equivalent SQL string.
`json_execute_serialized_sql(varchar)`	Table	Execute `json` serialized statements and return the resulting rows. Only one statement at a time is supported for now.
`json_serialize_sql(varchar, skip_default := boolean, skip_empty := boolean, skip_null := boolean, format := boolean)`	Scalar	Serialize a set of semicolon-separated (`;`) select statements to an equivalent list of `json` serialized statements.
`PRAGMA json_execute_serialized_sql(varchar)`	Pragma	Pragma version of the `json_execute_serialized_sql` function.

The `json_serialize_sql(varchar)` function takes three optional parameters, `skip_empty`, `skip_null`, and `format` that can be used to control the output of the serialized statements.

If you run the `json_execute_serialized_sql(varchar)` table function inside of a transaction the serialized statements will not be able to see any transaction local changes. This is because the statements are executed in a separate query context. You can use the `PRAGMA json_execute_serialized_sql(varchar)` pragma version to execute the statements in the same query context as the pragma, although with the limitation that the serialized JSON must be provided as a constant string, i.e., you cannot do `PRAGMA json_execute_serialized_sql(json_serialize_sql(...))`.

Note that these functions do not preserve syntactic sugar such as `FROM * SELECT ...`, so a statement round-tripped through `json_deserialize_sql(json_serialize_sql(...))` may not be identical to the original statement, but should always be semantically equivalent and produce the same output.

Examples

Simple example:

```
SELECT json_serialize_sql('SELECT 2');
```

{"error":false,"statements":[{"node":{"type":"SELECT_NODE","modifiers":[],"cte_map":{"map":[]},"select_list":[{"class":"CONSTANT","type":"VALUE_CONSTANT","alias":"","query_location":7,"value":{"type":{"id":"INTEGER","type_info":null},"is_null":false,"value":2}}],"from_table":{"type":"EMPTY","alias":"","sample":null,"query_location":18446744073709551615},"where_clause":null,"group_expressions":[],"group_sets":[],"aggregate_handling":"STANDARD_HANDLING","having":null,"sample":null,"qualify":null},"named_param_map":[]}]}

Example with multiple statements and skip options:

```
SELECT json_serialize_sql('SELECT 1 + 2; SELECT a + b FROM tbl1', skip_empty := true, skip_null := true);
```

{"error":false,"statements":[{"node":{"type":"SELECT_NODE","select_list":[{"class":"FUNCTION","type":"FUNCTION","query_location":9,"function_name":"+","children":[{"class":"CONSTANT","type":"VALUE_CONSTANT","query_location":7,"value":{"type":{"id":"INTEGER"},"is_null":false,"value":1}},{"class":"CONSTANT","type":"VALUE_CONSTANT","query_location":11,"value":{"type":{"id":"INTEGER"},"is_null":false,"value":2}}],"order_bys":{"type":"ORDER_MODIFIER"},"distinct":false,"is_operator":true,"export_state":false}],"from_table":{"type":"EMPTY","query_location":18446744073709551615},"aggregate_handling":"STANDARD_HANDLING"}},{"node":{"type":"SELECT_NODE","select_list":[{"class":"FUNCTION","type":"FUNCTION","query_location":23,"function_name":"+","children":[{"class":"COLUMN_REF","type":"COLUMN_REF","query_location":21,"column_names":["a"]},{"class":"COLUMN_REF","type":"COLUMN_REF","query_location":25,"column_names":["b"]}],"order_bys":{"type":"ORDER_MODIFIER"},"distinct":false,"is_operator":true,"export_state":false}],"from_table":{"type":"BASE_TABLE","query_location":32,"table_name":"tbl1"},"aggregate_handling":"STANDARD_HANDLING"}}]}

Skip the default values in the AST (e.g., `"distinct":false`):

```
SELECT json_serialize_sql('SELECT 1 + 2; SELECT a + b FROM tbl1', skip_default := true, skip_empty := true, skip_null := true);
```

{"error":false,"statements":[{"node":{"type":"SELECT_NODE","select_list":[{"class":"FUNCTION","type":"FUNCTION","query_location":9,"function_name":"+","children":[{"class":"CONSTANT","type":"VALUE_CONSTANT","query_location":7,"value":{"type":{"id":"INTEGER"},"is_null":false,"value":1}},{"class":"CONSTANT","type":"VALUE_CONSTANT","query_location":11,"value":{"type":{"id":"INTEGER"},"is_null":false,"value":2}}],"order_bys":{"type":"ORDER_MODIFIER"},"is_operator":true}],"from_table":{"type":"EMPTY"},"aggregate_handling":"STANDARD_HANDLING"}},{"node":{"type":"SELECT_NODE","select_list":[{"class":"FUNCTION","type":"FUNCTION","query_location":23,"function_name":"+","children":[{"class":"COLUMN_REF","type":"COLUMN_REF","query_location":21,"column_names":["a"]},{"class":"COLUMN_REF","type":"COLUMN_REF","query_location":25,"column_names":["b"]}],"order_bys":{"type":"ORDER_MODIFIER"},"is_operator":true}],"from_table":{"type":"BASE_TABLE","query_location":32,"table_name":"tbl1"},"aggregate_handling":"STANDARD_HANDLING"}}]}

Example with a syntax error:

```
SELECT json_serialize_sql('TOTALLY NOT VALID SQL');
```

{"error":true,"error_type":"parser","error_message":"syntax error at or near \"TOTALLY\"","error_subtype":"SYNTAX_ERROR","position":"0"}

Example with deserialize:

```
SELECT json_deserialize_sql(json_serialize_sql('SELECT 1 + 2'));
```

```
SELECT (1 + 2)
```

Example with deserialize and syntax sugar, which is lost during the transformation:

```
SELECT json_deserialize_sql(json_serialize_sql('FROM x SELECT 1 + 2'));
```

```
SELECT (1 + 2) FROM x
```

Example with execute:

```sql
SELECT * FROM json_execute_serialized_sql(json_serialize_sql('SELECT 1 + 2'));
```

```
3
```

Example with error:

```sql
SELECT * FROM json_execute_serialized_sql(json_serialize_sql('TOTALLY NOT VALID SQL'));
```

```
Parser Error:
Error parsing json: parser: syntax error at or near "TOTALLY"
```

Caveats

Equality Comparison

> **Warning.** Currently, equality comparison of JSON files can differ based on the context. In some cases, it is based on raw text comparison, while in other cases, it uses logical content comparison.

The following query returns true for all fields:

```sql
SELECT
    a != b, -- Space is part of physical JSON content. Despite equal logical content, values are treated as not equal.
    c != d, -- Same.
    c[0] = d[0], -- Equality because space was removed from physical content of fields:
    a = c[0], -- Indeed, field is equal to empty list without space...
    b != c[0], -- ... but different from empty list with space.
FROM (
    SELECT
        '[]'::JSON AS a,
        '[ ]'::JSON AS b,
        '[[]]'::JSON AS c,
        '[[ ]]'::JSON AS d
    );
```

(a != b)	(c != d)	(c[0] = d[0])	(a = c[0])	(b != c[0])
true	true	true	true	true

Multiple Files

Reading Multiple Files

DuckDB can read multiple files of different types (CSV, Parquet, JSON files) at the same time using either the glob syntax, or by providing a list of files to read. See the combining schemas page for tips on reading files with different schemas.

CSV

Read all files with a name ending in `.csv` in the folder `dir`:

```
SELECT *
FROM 'dir/*.csv';
```

Read all files with a name ending in `.csv`, two directories deep:

```
SELECT *
FROM '*/*/*.csv';
```

Read all files with a name ending in `.csv`, at any depth in the folder `dir`:

```
SELECT *
FROM 'dir/**/*.csv';
```

Read the CSV files `flights1.csv` and `flights2.csv`:

```
SELECT *
FROM read_csv(['flights1.csv', 'flights2.csv']);
```

Read the CSV files `flights1.csv` and `flights2.csv`, unifying schemas by name and outputting a `filename` column:

```
SELECT *
FROM read_csv(['flights1.csv', 'flights2.csv'], union_by_name = true, filename = true);
```

Parquet

Read all files that match the glob pattern:

```
SELECT *
FROM 'test/*.parquet';
```

Read three Parquet files and treat them as a single table:

```
SELECT *
FROM read_parquet(['file1.parquet', 'file2.parquet', 'file3.parquet']);
```

Read all Parquet files from two specific folders:

```
SELECT *
FROM read_parquet(['folder1/*.parquet', 'folder2/*.parquet']);
```

Read all Parquet files that match the glob pattern at any depth:

```
SELECT *
FROM read_parquet('dir/**/*.parquet');
```

Multi-File Reads and Globs

DuckDB can also read a series of Parquet files and treat them as if they were a single table. Note that this only works if the Parquet files have the same schema. You can specify which Parquet files you want to read using a list parameter, glob pattern matching syntax, or a combination of both.

List Parameter

The `read_parquet` function can accept a list of filenames as the input parameter.

Read three Parquet files and treat them as a single table:

```
SELECT *
FROM read_parquet(['file1.parquet', 'file2.parquet', 'file3.parquet']);
```

Glob Syntax

Any file name input to the `read_parquet` function can either be an exact filename, or use a glob syntax to read multiple files that match a pattern.

Wildcard	Description
`*`	Matches any number of any characters (including none)
`**`	Matches any number of subdirectories (including none)
`?`	Matches any single character
`[abc]`	Matches one character given in the bracket
`[a-z]`	Matches one character from the range given in the bracket

Note that the `?` wildcard in globs is not supported for reads over S3 due to HTTP encoding issues.

Here is an example that reads all the files that end with `.parquet` located in the `test` folder:

Read all files that match the glob pattern:

```
SELECT *
FROM read_parquet('test/*.parquet');
```

List of Globs

The glob syntax and the list input parameter can be combined to scan files that meet one of multiple patterns.

Read all Parquet files from 2 specific folders.

```
SELECT *
FROM read_parquet(['folder1/*.parquet', 'folder2/*.parquet']);
```

DuckDB can read multiple CSV files at the same time using either the glob syntax, or by providing a list of files to read.

Filename

The `filename` argument can be used to add an extra `filename` column to the result that indicates which row came from which file. For example:

```
SELECT *
FROM read_csv(['flights1.csv', 'flights2.csv'], union_by_name = true, filename = true);
```

FlightDate	OriginCityName	DestCityName	UniqueCarrier	filename
1988-01-01	New York, NY	Los Angeles, CA	NULL	flights1.csv
1988-01-02	New York, NY	Los Angeles, CA	NULL	flights1.csv
1988-01-03	New York, NY	Los Angeles, CA	AA	flights2.csv

Glob Function to Find Filenames

The glob pattern matching syntax can also be used to search for filenames using the `glob` table function. It accepts one parameter: the path to search (which may include glob patterns).

Search the current directory for all files.

```
SELECT *
FROM glob('*');
```

file
test.csv
test.json
test.parquet
test2.csv
test2.parquet
todos.json

Combining Schemas

Examples

Read a set of CSV files combining columns by position:

```
SELECT * FROM read_csv('flights*.csv');
```

Read a set of CSV files combining columns by name:

```
SELECT * FROM read_csv('flights*.csv', union_by_name = true);
```

Combining Schemas

When reading from multiple files, we have to **combine schemas** from those files. That is because each file has its own schema that can differ from the other files. DuckDB offers two ways of unifying schemas of multiple files: **by column position** and **by column name**.

By default, DuckDB reads the schema of the first file provided, and then unifies columns in subsequent files by column position. This works correctly as long as all files have the same schema. If the schema of the files differs, you might want to use the `union_by_name` option to allow DuckDB to construct the schema by reading all of the names instead.

Below is an example of how both methods work.

Union by Position

By default, DuckDB unifies the columns of these different files **by position**. This means that the first column in each file is combined together, as well as the second column in each file, etc. For example, consider the following two files.

`flights1.csv`:

```
FlightDate|UniqueCarrier|OriginCityName|DestCityName
1988-01-01|AA|New York, NY|Los Angeles, CA
1988-01-02|AA|New York, NY|Los Angeles, CA
```

`flights2.csv`:

```
FlightDate|UniqueCarrier|OriginCityName|DestCityName
1988-01-03|AA|New York, NY|Los Angeles, CA
```

Reading the two files at the same time will produce the following result set:

FlightDate	UniqueCarrier	OriginCityName	DestCityName
1988-01-01	AA	New York, NY	Los Angeles, CA
1988-01-02	AA	New York, NY	Los Angeles, CA
1988-01-03	AA	New York, NY	Los Angeles, CA

This is equivalent to the SQL construct `UNION ALL`.

Union by Name

If you are processing multiple files that have different schemas, perhaps because columns have been added or renamed, it might be desirable to unify the columns of different files **by name** instead. This can be done by providing the `union_by_name` option. For example, consider the following two files, where `flights4.csv` has an extra column (`UniqueCarrier`).

`flights3.csv`:

```
FlightDate|OriginCityName|DestCityName
1988-01-01|New York, NY|Los Angeles, CA
1988-01-02|New York, NY|Los Angeles, CA
```

`flights4.csv`:

```
FlightDate|UniqueCarrier|OriginCityName|DestCityName
1988-01-03|AA|New York, NY|Los Angeles, CA
```

Reading these when unifying column names **by position** results in an error – as the two files have a different number of columns. When specifying the `union_by_name` option, the columns are correctly unified, and any missing values are set to `NULL`.

```
SELECT * FROM read_csv(['flights3.csv', 'flights4.csv'], union_by_name = true);
```

FlightDate	OriginCityName	DestCityName	UniqueCarrier
1988-01-01	New York, NY	Los Angeles, CA	NULL
1988-01-02	New York, NY	Los Angeles, CA	NULL
1988-01-03	New York, NY	Los Angeles, CA	AA

This is equivalent to the SQL construct `UNION ALL BY NAME`.

> Using the `union_by_name` option increases memory consumption.

Parquet Files

Reading and Writing Parquet Files

Examples

Read a single Parquet file:

```
SELECT * FROM 'test.parquet';
```

Figure out which columns/types are in a Parquet file:

```
DESCRIBE SELECT * FROM 'test.parquet';
```

Create a table from a Parquet file:

```
CREATE TABLE test AS
    SELECT * FROM 'test.parquet';
```

If the file does not end in `.parquet`, use the `read_parquet` function:

```
SELECT *
FROM read_parquet('test.parq');
```

Use list parameter to read three Parquet files and treat them as a single table:

```
SELECT *
FROM read_parquet(['file1.parquet', 'file2.parquet', 'file3.parquet']);
```

Read all files that match the glob pattern:

```
SELECT *
FROM 'test/*.parquet';
```

Read all files that match the glob pattern, and include the `filename` virtual column that specifies which file each row came from (this column is available by default without a configuration options since DuckDB v1.3.0):

```
SELECT *, filename
FROM read_parquet('test/*.parquet');
```

Use a list of globs to read all Parquet files from two specific folders:

```
SELECT *
FROM read_parquet(['folder1/*.parquet', 'folder2/*.parquet']);
```

Read over HTTPS:

```
SELECT *
FROM read_parquet('https://some.url/some_file.parquet');
```

Query the metadata of a Parquet file:

```
SELECT *
FROM parquet_metadata('test.parquet');
```

Query the file metadata of a Parquet file:

```sql
SELECT *
FROM parquet_file_metadata('test.parquet');
```

Query the key-value metadata of a Parquet file:

```sql
SELECT *
FROM parquet_kv_metadata('test.parquet');
```

Query the schema of a Parquet file:

```sql
SELECT *
FROM parquet_schema('test.parquet');
```

Write the results of a query to a Parquet file using the default compression (Snappy):

```sql
COPY
    (SELECT * FROM tbl)
    TO 'result-snappy.parquet'
    (FORMAT parquet);
```

Write the results from a query to a Parquet file with specific compression and row group size:

```sql
COPY
    (FROM generate_series(100_000))
    TO 'test.parquet'
    (FORMAT parquet, COMPRESSION zstd, ROW_GROUP_SIZE 100_000);
```

Export the table contents of the entire database as parquet:

```sql
EXPORT DATABASE 'target_directory' (FORMAT parquet);
```

Parquet Files

Parquet files are compressed columnar files that are efficient to load and process. DuckDB provides support for both reading and writing Parquet files in an efficient manner, as well as support for pushing filters and projections into the Parquet file scans.

> Parquet data sets differ based on the number of files, the size of individual files, the compression algorithm used, row group size, etc. These have a significant effect on performance. Please consult the Performance Guide for details.

`read_parquet` Function

Function	Description	Example
`read_parquet(path_or_list_of_paths)`	Read Parquet file(s)	`SELECT * FROM read_parquet('test.parquet');`
`parquet_scan(path_or_list_of_paths)`	Alias for `read_parquet`	`SELECT * FROM parquet_scan('test.parquet');`

If your file ends in `.parquet`, the function syntax is optional. The system will automatically infer that you are reading a Parquet file:

```sql
SELECT * FROM 'test.parquet';
```

Multiple files can be read at once by providing a glob or a list of files. Refer to the multiple files section for more information.

Parameters

There are a number of options exposed that can be passed to the `read_parquet` function or the COPY statement.

Name	Description	Type	Default
`binary_as_string`	Parquet files generated by legacy writers do not correctly set the UTF8 flag for strings, causing string columns to be loaded as BLOB instead. Set this to true to load binary columns as strings.	BOOL	`false`
`encryption_config`	Configuration for Parquet encryption.	STRUCT	-
`filename`	Whether or not an extra `filename` column should be included in the result. Since DuckDB v1.3.0, the `filename` column is added automatically as a virtual column and this option is only kept for compatibility reasons.	BOOL	`false`
`file_row_number`	Whether or not to include the `file_row_number` column.	BOOL	`false`
`hive_partitioning`	Whether or not to interpret the path as a Hive partitioned path.	BOOL	(auto-detected)
`union_by_name`	Whether the columns of multiple schemas should be unified by name, rather than by position.	BOOL	`false`

Partial Reading

DuckDB supports projection pushdown into the Parquet file itself. That is to say, when querying a Parquet file, only the columns required for the query are read. This allows you to read only the part of the Parquet file that you are interested in. This will be done automatically by DuckDB.

DuckDB also supports filter pushdown into the Parquet reader. When you apply a filter to a column that is scanned from a Parquet file, the filter will be pushed down into the scan, and can even be used to skip parts of the file using the built-in zonemaps. Note that this will depend on whether or not your Parquet file contains zonemaps.

Filter and projection pushdown provide significant performance benefits. See our blog post "Querying Parquet with Precision Using DuckDB" for more information.

Inserts and Views

You can also insert the data into a table or create a table from the Parquet file directly. This will load the data from the Parquet file and insert it into the database:

Insert the data from the Parquet file in the table:

```
INSERT INTO people
    SELECT * FROM read_parquet('test.parquet');
```

Create a table directly from a Parquet file:

```
CREATE TABLE people AS
    SELECT * FROM read_parquet('test.parquet');
```

If you wish to keep the data stored inside the Parquet file, but want to query the Parquet file directly, you can create a view over the `read_parquet` function. You can then query the Parquet file as if it were a built-in table:

Create a view over the Parquet file:

```
CREATE VIEW people AS
    SELECT * FROM read_parquet('test.parquet');
```

Query the Parquet file:

```
SELECT * FROM people;
```

Writing to Parquet Files

DuckDB also has support for writing to Parquet files using the COPY statement syntax. See the COPY Statement page for details, including all possible parameters for the COPY statement.

Write a query to a Snappy-compressed Parquet file:

```
COPY
    (SELECT * FROM tbl)
    TO 'result-snappy.parquet'
    (FORMAT parquet);
```

Write `tbl` to a zstd-compressed Parquet file:

```
COPY tbl
    TO 'result-zstd.parquet'
    (FORMAT parquet, COMPRESSION zstd);
```

Write `tbl` to a zstd-compressed Parquet file with the lowest compression level yielding the fastest compression:

```
COPY tbl
    TO 'result-zstd.parquet'
    (FORMAT parquet, COMPRESSION zstd, COMPRESSION_LEVEL 1);
```

Write to Parquet file with key-value metadata:

```
COPY (
    SELECT
        42 AS number,
        true AS is_even
) TO 'kv_metadata.parquet' (
    FORMAT parquet,
    KV_METADATA {
        number: 'Answer to life, universe, and everything',
        is_even: 'not ''odd''' -- single quotes in values must be escaped
    }
);
```

Write a CSV file to an uncompressed Parquet file:

```
COPY
    'test.csv'
    TO 'result-uncompressed.parquet'
    (FORMAT parquet, COMPRESSION uncompressed);
```

Write a query to a Parquet file with zstd-compression and row group size:

```
COPY
    (FROM generate_series(100_000))
    TO 'row-groups-zstd.parquet'
    (FORMAT parquet, COMPRESSION zstd, ROW_GROUP_SIZE 100_000);
```

Write data to an LZ4-compressed Parquet file:

```
COPY
    (FROM generate_series(100_000))
    TO 'result-lz4.parquet'
    (FORMAT parquet, COMPRESSION lz4);
```

Or, equivalently:

```
COPY
    (FROM generate_series(100_000))
    TO 'result-lz4.parquet'
    (FORMAT parquet, COMPRESSION lz4_raw);
```

Write data to a Brotli-compressed Parquet file:

```
COPY
    (FROM generate_series(100_000))
    TO 'result-brotli.parquet'
    (FORMAT parquet, COMPRESSION brotli);
```

To configure the page size of Parquet file's dictionary pages, use the `STRING_DICTIONARY_PAGE_SIZE_LIMIT` option (default: 1 MB):

```
COPY
    lineitem
    TO 'lineitem-with-custom-dictionary-size.parquet'
    (FORMAT parquet, STRING_DICTIONARY_PAGE_SIZE_LIMIT 100_000);
```

DuckDB's `EXPORT` command can be used to export an entire database to a series of Parquet files. See the "EXPORT statement" page for more details:

Export the table contents of the entire database as Parquet:

```
EXPORT DATABASE 'target_directory' (FORMAT parquet);
```

Encryption

DuckDB supports reading and writing encrypted Parquet files.

Supported Features

The list of supported Parquet features is available in the Parquet documentation's "Implementation status" page.

Installing and Loading the Parquet Extension

The support for Parquet files is enabled via extension. The `parquet` extension is bundled with almost all clients. However, if your client does not bundle the `parquet` extension, the extension must be installed separately:

```
INSTALL parquet;
```

Querying Parquet Metadata

Parquet Metadata

The `parquet_metadata` function can be used to query the metadata contained within a Parquet file, which reveals various internal details of the Parquet file such as the statistics of the different columns. This can be useful for figuring out what kind of skipping is possible in Parquet files, or even to obtain a quick overview of what the different columns contain:

```
SELECT *
FROM parquet_metadata('test.parquet');
```

Below is a table of the columns returned by `parquet_metadata`.

Field	Type
file_name	VARCHAR
row_group_id	BIGINT
row_group_num_rows	BIGINT
row_group_num_columns	BIGINT
row_group_bytes	BIGINT
column_id	BIGINT
file_offset	BIGINT
num_values	BIGINT
path_in_schema	VARCHAR
type	VARCHAR
stats_min	VARCHAR
stats_max	VARCHAR
stats_null_count	BIGINT
stats_distinct_count	BIGINT
stats_min_value	VARCHAR
stats_max_value	VARCHAR
compression	VARCHAR
encodings	VARCHAR
index_page_offset	BIGINT
dictionary_page_offset	BIGINT
data_page_offset	BIGINT
total_compressed_size	BIGINT
total_uncompressed_size	BIGINT
key_value_metadata	MAP(BLOB, BLOB)
bloom_filter_offset	BIGINT
bloom_filter_length	BIGINT

Parquet Schema

The `parquet_schema` function can be used to query the internal schema contained within a Parquet file. Note that this is the schema as it is contained within the metadata of the Parquet file. If you want to figure out the column names and types contained within a Parquet file it is easier to use `DESCRIBE`.

Fetch the column names and column types:

```
DESCRIBE SELECT * FROM 'test.parquet';
```

Fetch the internal schema of a Parquet file:

```
SELECT *
FROM parquet_schema('test.parquet');
```

Below is a table of the columns returned by `parquet_schema`.

Field	Type
file_name	VARCHAR
name	VARCHAR
type	VARCHAR
type_length	VARCHAR
repetition_type	VARCHAR
num_children	BIGINT
converted_type	VARCHAR
scale	BIGINT
precision	BIGINT
field_id	BIGINT
logical_type	VARCHAR

Parquet File Metadata

The `parquet_file_metadata` function can be used to query file-level metadata such as the format version and the encryption algorithm used:

```
SELECT *
FROM parquet_file_metadata('test.parquet');
```

Below is a table of the columns returned by `parquet_file_metadata`.

Field	Type
file_name	VARCHAR
created_by	VARCHAR
num_rows	BIGINT
num_row_groups	BIGINT
format_version	BIGINT
encryption_algorithm	VARCHAR
footer_signing_key_metadata	VARCHAR

Parquet Key-Value Metadata

The `parquet_kv_metadata` function can be used to query custom metadata defined as key-value pairs:

```
SELECT *
FROM parquet_kv_metadata('test.parquet');
```

Below is a table of the columns returned by `parquet_kv_metadata`.

Field	Type
file_name	VARCHAR
key	BLOB
value	BLOB

Bloom Filters

DuckDB supports Bloom filters for pruning the row groups that need to be read to answer highly selective queries. Currently, Bloom filters are supported for the following types:

- Integer types: `TINYINT`, `UTINYINT`, `SMALLINT`, `USMALLINT`, `INTEGER`, `UINTEGER`, `BIGINT`, `UBIGINT`
- Floating point types: `FLOAT`, `DOUBLE`
- `VARCHAR`
- `BLOB`

The `parquet_bloom_probe(filename, column_name, value)` function shows which row groups can excluded when filtering for a given value of a given column using the Bloom filter. For example:

```
FROM parquet_bloom_probe('my_file.parquet', 'my_col', 500);
```

file_name	row_group_id	bloom_filter_excludes
my_file.parquet	0	true
…	…	…
my_file.parquet	9	false

Parquet Encryption

Starting with version 0.10.0, DuckDB supports reading and writing encrypted Parquet files. DuckDB broadly follows the Parquet Modular Encryption specification with some limitations.

Reading and Writing Encrypted Files

Using the PRAGMA `add_parquet_key` function, named encryption keys of 128, 192, or 256 bits can be added to a session. These keys are stored in-memory:

```
PRAGMA add_parquet_key('key128', '0123456789112345');
PRAGMA add_parquet_key('key192', '012345678911234501234567');
PRAGMA add_parquet_key('key256', '0123456789112345012345678911234 5');
PRAGMA add_parquet_key('key256base64', 'MDEyMzQ1Njc4OTExMjM0NTAxMjM0NTY3ODkxMTIzNDU=');
```

Writing Encrypted Parquet Files

After specifying the key (e.g., `key256`), files can be encrypted as follows:

```
COPY tbl TO 'tbl.parquet' (ENCRYPTION_CONFIG {footer_key: 'key256'});
```

Reading Encrypted Parquet Files

An encrypted Parquet file using a specific key (e.g., `key256`), can then be read as follows:

```
COPY tbl FROM 'tbl.parquet' (ENCRYPTION_CONFIG {footer_key: 'key256'});
```

Or:

```
SELECT *
FROM read_parquet('tbl.parquet', encryption_config = {footer_key: 'key256'});
```

Limitations

DuckDB's Parquet encryption currently has the following limitations.

1. It is not compatible with the encryption of, e.g., PyArrow, until the missing details are implemented.
2. DuckDB encrypts the footer and all columns using the `footer_key`. The Parquet specification allows encryption of individual columns with different keys, e.g.:

   ```
   COPY tbl TO 'tbl.parquet'
       (ENCRYPTION_CONFIG {
           footer_key: 'key256',
           column_keys: {key256: ['col0', 'col1']}
       });
   ```

   However, this is unsupported at the moment and will cause an error to be thrown (for now):

   ```
   Not implemented Error: Parquet encryption_config column_keys not yet implemented
   ```

Performance Implications

Note that encryption has some performance implications. Without encryption, reading/writing the `lineitem` table from TPC-H at SF1, which is 6M rows and 15 columns, from/to a Parquet file takes 0.26 and 0.99 seconds, respectively. With encryption, this takes 0.64 and 2.21 seconds, both approximately 2.5× slower than the unencrypted version.

Parquet Tips

Below is a collection of tips to help when dealing with Parquet files.

Tips for Reading Parquet Files

Use `union_by_name` When Loading Files with Different Schemas

The `union_by_name` option can be used to unify the schema of files that have different or missing columns. For files that do not have certain columns, `NULL` values are filled in:

```
SELECT *
FROM read_parquet('flights*.parquet', union_by_name = true);
```

Tips for Writing Parquet Files

Using a glob pattern upon read or a Hive partitioning structure are good ways to transparently handle multiple files.

Enabling PER_THREAD_OUTPUT

If the final number of Parquet files is not important, writing one file per thread can significantly improve performance:

```
COPY
    (FROM generate_series(10_000_000))
    TO 'test.parquet'
    (FORMAT parquet, PER_THREAD_OUTPUT);
```

Selecting a ROW_GROUP_SIZE

The ROW_GROUP_SIZE parameter specifies the minimum number of rows in a Parquet row group, with a minimum value equal to DuckDB's vector size, 2,048, and a default of 122,880. A Parquet row group is a partition of rows, consisting of a column chunk for each column in the dataset.

Compression algorithms are only applied per row group, so the larger the row group size, the more opportunities to compress the data. DuckDB can read Parquet row groups in parallel even within the same file and uses predicate pushdown to only scan the row groups whose metadata ranges match the WHERE clause of the query. However there is some overhead associated with reading the metadata in each group. A good approach would be to ensure that within each file, the total number of row groups is at least as large as the number of CPU threads used to query that file. More row groups beyond the thread count would improve the speed of highly selective queries, but slow down queries that must scan the whole file like aggregations.

To write a query to a Parquet file with a different row group size, run:

```
COPY
    (FROM generate_series(100_000))
    TO 'row-groups.parquet'
    (FORMAT parquet, ROW_GROUP_SIZE 100_000);
```

The ROW_GROUPS_PER_FILE Option

The ROW_GROUPS_PER_FILE parameter creates a new Parquet file if the current one has a specified number of row groups.

```
COPY
    (FROM generate_series(100_000))
    TO 'output-directory'
    (FORMAT parquet, ROW_GROUP_SIZE 20_000, ROW_GROUPS_PER_FILE 2);
```

> If multiple threads are active, the number of row groups in a file may slightly exceed the specified number of row groups to limit the amount of locking – similarly to the behaviour of FILE_SIZE_BYTES. However, if PER_THREAD_OUTPUT is set, only one thread writes to each file, and it becomes accurate again.

See the Performance Guide on "File Formats" for more tips.

Partitioning

Hive Partitioning

Examples

Read data from a Hive partitioned data set:

```sql
SELECT *
FROM read_parquet('orders/*/*/*.parquet', hive_partitioning = true);
```

Write a table to a Hive partitioned data set:

```sql
COPY orders
TO 'orders' (FORMAT parquet, PARTITION_BY (year, month));
```

Note that the PARTITION_BY options cannot use expressions. You can produce columns on the fly using the following syntax:

```sql
COPY (SELECT *, year(timestamp) AS year, month(timestamp) AS month FROM services)
TO 'test' (PARTITION_BY (year, month));
```

When reading, the partition columns are read from the directory structure and can be included or excluded depending on the `hive_partitioning` parameter.

```sql
FROM read_parquet('test/*/*/*.parquet', hive_partitioning = false); -- will not include year, month columns
FROM read_parquet('test/*/*/*.parquet', hive_partitioning = true);  -- will include year, month partition columns
```

Hive Partitioning

Hive partitioning is a partitioning strategy that is used to split a table into multiple files based on **partition keys**. The files are organized into folders. Within each folder, the **partition key** has a value that is determined by the name of the folder.

Below is an example of a Hive partitioned file hierarchy. The files are partitioned on two keys (`year` and `month`).

```
orders
├── year=2021
│   ├── month=1
│   │   ├── file1.parquet
│   │   └── file2.parquet
│   └── month=2
│       └── file3.parquet
└── year=2022
    ├── month=11
    │   ├── file4.parquet
    │   └── file5.parquet
    └── month=12
        └── file6.parquet
```

Files stored in this hierarchy can be read using the `hive_partitioning` flag.

```sql
SELECT *
FROM read_parquet('orders/*/*/*.parquet', hive_partitioning = true);
```

When we specify the `hive_partitioning` flag, the values of the columns will be read from the directories.

Filter Pushdown

Filters on the partition keys are automatically pushed down into the files. This way the system skips reading files that are not necessary to answer a query. For example, consider the following query on the above dataset:

```sql
SELECT *
FROM read_parquet('orders/*/*/*.parquet', hive_partitioning = true)
WHERE year = 2022
  AND month = 11;
```

When executing this query, only the following files will be read:

```
orders
└── year=2022
    └── month=11
        ├── file4.parquet
        └── file5.parquet
```

Autodetection

By default the system tries to infer if the provided files are in a hive partitioned hierarchy. And if so, the `hive_partitioning` flag is enabled automatically. The autodetection will look at the names of the folders and search for a `'key' = 'value'` pattern. This behavior can be overridden by using the `hive_partitioning` configuration option:

```sql
SET hive_partitioning = false;
```

Hive Types

`hive_types` is a way to specify the logical types of the hive partitions in a struct:

```sql
SELECT *
FROM read_parquet(
    'dir/**/*.parquet',
    hive_partitioning = true,
    hive_types = {'release': DATE, 'orders': BIGINT}
);
```

`hive_types` will be autodetected for the following types: `DATE`, `TIMESTAMP` and `BIGINT`. To switch off the autodetection, the flag `hive_types_autocast = 0` can be set.

Writing Partitioned Files

See the Partitioned Writes section.

Partitioned Writes

Examples

Write a table to a Hive partitioned data set of Parquet files:

```
COPY orders TO 'orders'
(FORMAT parquet, PARTITION_BY (year, month));
```

Write a table to a Hive partitioned data set of CSV files, allowing overwrites:

```
COPY orders TO 'orders'
(FORMAT csv, PARTITION_BY (year, month), OVERWRITE_OR_IGNORE);
```

Write a table to a Hive partitioned data set of GZIP-compressed CSV files, setting explicit data files' extension:

```
COPY orders TO 'orders'
(FORMAT csv, PARTITION_BY (year, month), COMPRESSION gzip, FILE_EXTENSION 'csv.gz');
```

Partitioned Writes

When the PARTITION_BY clause is specified for the COPY statement, the files are written in a Hive partitioned folder hierarchy. The target is the name of the root directory (in the example above: `orders`). The files are written in-order in the file hierarchy. Currently, one file is written per thread to each directory.

```
orders
├── year=2021
│   ├── month=1
│   │   ├── data_1.parquet
│   │   └── data_2.parquet
│   └── month=2
│       └── data_1.parquet
└── year=2022
    ├── month=11
    │   ├── data_1.parquet
    │   └── data_2.parquet
    └── month=12
        └── data_1.parquet
```

The values of the partitions are automatically extracted from the data. Note that it can be very expensive to write a larger number of partitions as many files will be created. The ideal partition count depends on how large your data set is.

To limit the maximum number of files the system can keep open before flushing to disk when writing using PARTITION_BY, use the `partitioned_write_max_open_files` configuration option (default: 100):

```
SET partitioned_write_max_open_files = 10;
```

> **Best practice.** Writing data into many small partitions is expensive. It is generally recommended to have at least `100 MB` of data per partition.

Filename Pattern

By default, files will be named `data_0.parquet` or `data_0.csv`. With the flag FILENAME_PATTERN a pattern with `{i}` or `{uuid}` can be defined to create specific filenames:

- `{i}` will be replaced by an index
- `{uuid}` will be replaced by a 128 bits long UUID

Write a table to a Hive partitioned data set of .parquet files, with an index in the filename:

```
COPY orders TO 'orders'
(FORMAT parquet, PARTITION_BY (year, month), OVERWRITE_OR_IGNORE, FILENAME_PATTERN 'orders_{i}');
```

Write a table to a Hive partitioned data set of .parquet files, with unique filenames:

```
COPY orders TO 'orders'
(FORMAT parquet, PARTITION_BY (year, month), OVERWRITE_OR_IGNORE, FILENAME_PATTERN 'file_{uuid}');
```

Overwriting

By default the partitioned write will not allow overwriting existing directories. On a local file system, the `OVERWRITE` and `OVERWRITE_OR_IGNORE` options remove the existing directories. On remote file systems, overwriting is not supported.

Appending

To append to an existing Hive partitioned directory structure, use the `APPEND` option:

```
COPY orders TO 'orders'
(FORMAT parquet, PARTITION_BY (year, month), APPEND);
```

Using the `APPEND` option result in a behavior similar the `OVERWRITE_OR_IGNORE, FILENAME_PATTERN '{uuid}'` options, but DuckDB performs an extra check for whether the file already exists and then regenerates the UUID in the rare event that it does (to avoid clashes).

Handling Slashes in Columns

To handle slashes in column names, use Percent-Encoding implemented by the `url_encode` function.

Appender

The Appender can be used to load bulk data into a DuckDB database. It is currently available in the C, C++, Go, Java, and Rust APIs. The Appender is tied to a connection, and will use the transaction context of that connection when appending. An Appender always appends to a single table in the database file.

In the C++ API, the Appender works as follows:

```
DuckDB db;
Connection con(db);
// create the table
con.Query("CREATE TABLE people (id INTEGER, name VARCHAR)");
// initialize the appender
Appender appender(con, "people");
```

The AppendRow function is the easiest way of appending data. It uses recursive templates to allow you to put all the values of a single row within one function call, as follows:

```
appender.AppendRow(1, "Mark");
```

Rows can also be individually constructed using the BeginRow, EndRow and Append methods. This is done internally by AppendRow, and hence has the same performance characteristics.

```
appender.BeginRow();
appender.Append<int32_t>(2);
appender.Append<string>("Hannes");
appender.EndRow();
```

Any values added to the Appender are cached prior to being inserted into the database system for performance reasons. That means that, while appending, the rows might not be immediately visible in the system. The cache is automatically flushed when the Appender goes out of scope or when `appender.Close()` is called. The cache can also be manually flushed using the `appender.Flush()` method. After either `Flush` or `Close` is called, all the data has been written to the database system.

Date, Time and Timestamps

While numbers and strings are rather self-explanatory, dates, times and timestamps require some explanation. They can be directly appended using the methods provided by `duckdb::Date`, `duckdb::Time` or `duckdb::Timestamp`. They can also be appended using the internal `duckdb::Value` type, however, this adds some additional overheads and should be avoided if possible.

Below is a short example:

```
con.Query("CREATE TABLE dates (d DATE, t TIME, ts TIMESTAMP)");
Appender appender(con, "dates");

// construct the values using the Date/Time/Timestamp types
// (this is the most efficient approach)
appender.AppendRow(
    Date::FromDate(1992, 1, 1),
    Time::FromTime(1, 1, 1, 0),
    Timestamp::FromDatetime(Date::FromDate(1992, 1, 1), Time::FromTime(1, 1, 1, 0))
);
// construct duckdb::Value objects
```

```
appender.AppendRow(
    Value::DATE(1992, 1, 1),
    Value::TIME(1, 1, 1, 0),
    Value::TIMESTAMP(1992, 1, 1, 1, 1, 1, 0)
);
```

Commit Frequency

By default, the appender performs a commits every 204,800 rows. You can change this by explicitly using transactions and surrounding your batches of AppendRow calls by `BEGIN TRANSACTION` and `COMMIT` statements.

Handling Constraint Violations

If the Appender encounters a `PRIMARY KEY` conflict or a `UNIQUE` constraint violation, it fails and returns the following error:

```
Constraint Error:
PRIMARY KEY or UNIQUE constraint violated: duplicate key "..."
```

In this case, the entire append operation fails and no rows are inserted.

Appender Support in Other Clients

The Appender is also available in the following client APIs:

- C
- Go
- Java (JDBC)
- Julia
- Rust

INSERT Statements

`INSERT` statements are the standard way of loading data into a relational database. When using `INSERT` statements, the values are supplied row-by-row. While simple, there is significant overhead involved in parsing and processing individual `INSERT` statements. This makes lots of individual row-by-row insertions very inefficient for bulk insertion.

> **Best practice.** As a rule-of-thumb, avoid using lots of individual row-by-row `INSERT` statements when inserting more than a few rows (i.e., avoid using `INSERT` statements as part of a loop). When bulk inserting data, try to maximize the amount of data that is inserted per statement.

If you must use `INSERT` statements to load data in a loop, avoid executing the statements in auto-commit mode. After every commit, the database is required to sync the changes made to disk to ensure no data is lost. In auto-commit mode every single statement will be wrapped in a separate transaction, meaning `fsync` will be called for every statement. This is typically unnecessary when bulk loading and will significantly slow down your program.

> **Tip.** If you absolutely must use `INSERT` statements in a loop to load data, wrap them in calls to `BEGIN TRANSACTION` and `COMMIT`.

Syntax

An example of using `INSERT INTO` to load data in a table is as follows:

```
CREATE TABLE people (id INTEGER, name VARCHAR);
INSERT INTO people VALUES (1, 'Mark'), (2, 'Hannes');
```

For a more detailed description together with syntax diagram can be found, see the page on the `INSERT` statement.

Client APIs

Client Overview

DuckDB is an in-process database system and offers client APIs (also known as “drivers”) for several languages.

Client API	Maintainer	Support tier	Version
C	The DuckDB team	Primary	1.3.1
Command Line Interface (CLI)	The DuckDB team	Primary	1.3.1
Java (JDBC)	The DuckDB team	Primary	{{ site.current_duckdb_java_short_version }}
Go	Marc Boeker and the DuckDB team	Primary	{{ site.current_duckdb_go_version }}
Node.js (node-neo)	Jeff Raymakers (MotherDuck)	Primary	{{ site.current_duckdb_node_neo_version }}
ODBC	The DuckDB team	Primary	{{ site.current_duckdb_odbc_short_version }}
Python	The DuckDB team	Primary	1.3.1
R	Kirill Müller and the DuckDB team	Primary	{{ site.current_duckdb_r_version }}
Rust	The DuckDB team	Primary	{{ site.current_duckdb_rust_version }}
WebAssembly (Wasm)	The DuckDB team	Primary	{{ site.current_duckdb_wasm_version }}
ADBC (Arrow)	The DuckDB team	Secondary	1.3.1
C# (.NET)	Giorgi	Secondary	{{ site.current_duckdb_csharp_version}}
C++	The DuckDB team	Secondary	1.3.1
Dart	TigerEye	Secondary	{{ site.current_duckdb_dart_version }}
Node.js (deprecated)	The DuckDB team	Secondary	{{ site.current_duckdb_nodejs_version }}
Common Lisp	ak-coram	Tertiary	
Crystal	amauryt	Tertiary	
Elixir	AlexR2D2	Tertiary	
Erlang	MM Zeeman	Tertiary	
Julia	The DuckDB team	Tertiary	
PHP	satur-io	Tertiary	
Pyodide	The DuckDB team	Tertiary	
Ruby	suketa	Tertiary	
Scala	Salar Rahmanian	Tertiary	
Swift	The DuckDB team	Tertiary	
Zig	karlseguin	Tertiary	

Support Tiers

Since there is such a wide variety of clients, the DuckDB team focuses their development effort on the most popular clients. To reflect this, we distinguish three tiers of support for clients. Primary clients are the first to receive new features and are covered by community support.

Secondary clients receive new features but are not covered by community support. Finally, there are no feature or support guarantees for tertiary clients.

> The DuckDB clients listed above are open-source and we welcome community contributions to these libraries. All primary and secondary clients are available for the MIT license. For tertiary clients, please consult the repository for the license.

We report the latest stable version for the clients in the primary and secondary support tiers.

Compatibility

All DuckDB clients support the same DuckDB SQL syntax and use the same on-disk database format. DuckDB extensions are also portable between clients with some exceptions (see Wasm extensions).

ADBC Client

> The latest version of the DuckDB ADBC client is 1.3.1.

Arrow Database Connectivity (ADBC), similarly to ODBC and JDBC, is a C-style API that enables code portability between different database systems. This allows developers to effortlessly build applications that communicate with database systems without using code specific to that system. The main difference between ADBC and ODBC/JDBC is that ADBC uses Arrow to transfer data between the database system and the application. DuckDB has an ADBC driver, which takes advantage of the zero-copy integration between DuckDB and Arrow to efficiently transfer data.

Please refer to the ADBC documentation page for a more extensive discussion on ADBC and a detailed API explanation.

Implemented Functionality

The DuckDB-ADBC driver implements the full ADBC specification, with the exception of the `ConnectionReadPartition` and `StatementExecutePartitions` functions. Both of these functions exist to support systems that internally partition the query results, which does not apply to DuckDB. In this section, we will describe the main functions that exist in ADBC, along with the arguments they take and provide examples for each function.

Database

Set of functions that operate on a database.

Function name	Description	Arguments	Example
`DatabaseNew`	Allocate a new (but uninitialized) database.	`(AdbcDatabase *database, AdbcError *error)`	`AdbcDatabaseNew(&adbc_database, &adbc_error)`
`DatabaseSetOption`	Set a char* option.	`(AdbcDatabase *database, const char *key, const char *value, AdbcError *error)`	`AdbcDatabaseSetOption(&adbc_database, "path", "test.db", &adbc_error)`
`DatabaseInit`	Finish setting options and initialize the database.	`(AdbcDatabase *database, AdbcError *error)`	`AdbcDatabaseInit(&adbc_database, &adbc_error)`
`DatabaseRelease`	Destroy the database.	`(AdbcDatabase *database, AdbcError *error)`	`AdbcDatabaseRelease(&adbc_database, &adbc_error)`

Connection

A set of functions that create and destroy a connection to interact with a database.

Function name	Description	Arguments	Example
`ConnectionNew`	Allocate a new (but uninitialized) connection.	`(AdbcConnection*, AdbcError*)`	`AdbcConnectionNew(&adbc_connection, &adbc_error)`
`ConnectionSetOption`	Options may be set before ConnectionInit.	`(AdbcConnection*, const char*, const char*, AdbcError*)`	`AdbcConnectionSetOption(&adbc_connection, ADBC_CONNECTION_OPTION_AUTOCOMMIT, ADBC_OPTION_VALUE_DISABLED, &adbc_error)`
`ConnectionInit`	Finish setting options and initialize the connection.	`(AdbcConnection*, AdbcDatabase*, AdbcError*)`	`AdbcConnectionInit(&adbc_connection, &adbc_database, &adbc_error)`
`ConnectionRelease`	Destroy this connection.	`(AdbcConnection*, AdbcError*)`	`AdbcConnectionRelease(&adbc_connection, &adbc_error)`

A set of functions that retrieve metadata about the database. In general, these functions will return Arrow objects, specifically an ArrowArrayStream.

Function name	Description	Arguments	Example
`ConnectionGetObjects`	Get a hierarchical view of all catalogs, database schemas, tables, and columns.	`(AdbcConnection*, int, const char*, const char*, const char*, const char**, const char*, ArrowArrayStream*, AdbcError*)`	`AdbcDatabaseInit(&adbc_database, &adbc_error)`
`ConnectionGetTableSchema`	Get the Arrow schema of a table.	`(AdbcConnection*, const char*, const char*, const char*, ArrowSchema*, AdbcError*)`	`AdbcDatabaseRelease(&adbc_database, &adbc_error)`
`ConnectionGetTableTypes`	Get a list of table types in the database.	`(AdbcConnection*, ArrowArrayStream*, AdbcError*)`	`AdbcDatabaseNew(&adbc_database, &adbc_error)`

A set of functions with transaction semantics for the connection. By default, all connections start with auto-commit mode on, but this can be turned off via the ConnectionSetOption function.

Function name	Description	Arguments	Example
`ConnectionCommit`	Commit any pending transactions.	`(AdbcConnection*, AdbcError*)`	`AdbcConnectionCommit(&adbc_connection, &adbc_error)`

Function name	Description	Arguments	Example
`ConnectionRollback`	Rollback any pending transactions.	`(AdbcConnection*, AdbcError*)`	`AdbcConnectionRollback(&adbc_connection, &adbc_error)`

Statement

Statements hold state related to query execution. They represent both one-off queries and prepared statements. They can be reused; however, doing so will invalidate prior result sets from that statement.

The functions used to create, destroy, and set options for a statement:

Function name	Description	Arguments	Example
`StatementNew`	Create a new statement for a given connection.	`(AdbcConnection*, AdbcStatement*, AdbcError*)`	`AdbcStatementNew(&adbc_connection, &adbc_statement, &adbc_error)`
`StatementRelease`	Destroy a statement.	`(AdbcStatement*, AdbcError*)`	`AdbcStatementRelease(&adbc_statement, &adbc_error)`
`StatementSetOption`	Set a string option on a statement.	`(AdbcStatement*, const char*, const char*, AdbcError*)`	`StatementSetOption(&adbc_statement, ADBC_INGEST_OPTION_TARGET_TABLE, "TABLE_NAME", &adbc_error)`

Functions related to query execution:

Function name	Description	Arguments	Example
`StatementSetSqlQuery`	Set the SQL query to execute. The query can then be executed with StatementExecuteQuery.	`(AdbcStatement*, const char*, AdbcError*)`	`AdbcStatementSetSqlQuery(&adbc_statement, "SELECT * FROM TABLE", &adbc_error)`
`StatementSetSubstraitPlan`	Set a substrait plan to execute. The query can then be executed with StatementExecuteQuery.	`(AdbcStatement*, const uint8_t*, size_t, AdbcError*)`	`AdbcStatementSetSubstraitPlan(&adbc_statement, substrait_plan, length, &adbc_error)`
`StatementExecuteQuery`	Execute a statement and get the results.	`(AdbcStatement*, ArrowArrayStream*, int64_t*, AdbcError*)`	`AdbcStatementExecuteQuery(&adbc_statement, &arrow_stream, &rows_affected, &adbc_error)`

Function name	Description	Arguments	Example
`StatementPrepare`	Turn this statement into a prepared statement to be executed multiple times.	`(AdbcStatement*, AdbcError*)`	`AdbcStatementPrepare(&adbc_statement, &adbc_error)`

Functions related to binding, used for bulk insertion or in prepared statements.

Function name	Description	Arguments	Example
`StatementBindStream`	Bind Arrow Stream. This can be used for bulk inserts or prepared statements.	`(AdbcStatement*, ArrowArrayStream*, AdbcError*)`	`StatementBindStream(&adbc_statement, &input_data, &adbc_error)`

Examples

Regardless of the programming language being used, there are two database options which will be required to utilize ADBC with DuckDB. The first one is the `driver`, which takes a path to the DuckDB library. The second option is the `entrypoint`, which is an exported function from the DuckDB-ADBC driver that initializes all the ADBC functions. Once we have configured these two options, we can optionally set the `path` option, providing a path on disk to store our DuckDB database. If not set, an in-memory database is created. After configuring all the necessary options, we can proceed to initialize our database. Below is how you can do so with various different language environments.

C++

We begin our C++ example by declaring the essential variables for querying data through ADBC. These variables include Error, Database, Connection, Statement handling, and an Arrow Stream to transfer data between DuckDB and the application.

```
AdbcError adbc_error;
AdbcDatabase adbc_database;
AdbcConnection adbc_connection;
AdbcStatement adbc_statement;
ArrowArrayStream arrow_stream;
```

We can then initialize our database variable. Before initializing the database, we need to set the `driver` and `entrypoint` options as mentioned above. Then we set the `path` option and initialize the database. With the example below, the string `"path/to/libduckdb.dylib"` should be the path to the dynamic library for DuckDB. This will be `.dylib` on macOS, and `.so` on Linux.

```
AdbcDatabaseNew(&adbc_database, &adbc_error);
AdbcDatabaseSetOption(&adbc_database, "driver", "path/to/libduckdb.dylib", &adbc_error);
AdbcDatabaseSetOption(&adbc_database, "entrypoint", "duckdb_adbc_init", &adbc_error);
// By default, we start an in-memory database, but you can optionally define a path to store it on disk.
AdbcDatabaseSetOption(&adbc_database, "path", "test.db", &adbc_error);
AdbcDatabaseInit(&adbc_database, &adbc_error);
```

After initializing the database, we must create and initialize a connection to it.

```
AdbcConnectionNew(&adbc_connection, &adbc_error);
AdbcConnectionInit(&adbc_connection, &adbc_database, &adbc_error);
```

We can now initialize our statement and run queries through our connection. After the `AdbcStatementExecuteQuery` the `arrow_stream` is populated with the result.

```
AdbcStatementNew(&adbc_connection, &adbc_statement, &adbc_error);
AdbcStatementSetSqlQuery(&adbc_statement, "SELECT 42", &adbc_error);
int64_t rows_affected;
AdbcStatementExecuteQuery(&adbc_statement, &arrow_stream, &rows_affected, &adbc_error);
arrow_stream.release(arrow_stream)
```

Besides running queries, we can also ingest data via `arrow_streams`. For this we need to set an option with the table name we want to insert to, bind the stream and then execute the query.

```
StatementSetOption(&adbc_statement, ADBC_INGEST_OPTION_TARGET_TABLE, "AnswerToEverything", &adbc_error);
StatementBindStream(&adbc_statement, &arrow_stream, &adbc_error);
StatementExecuteQuery(&adbc_statement, nullptr, nullptr, &adbc_error);
```

Python

The first thing to do is to use `pip` and install the ADBC Driver manager. You will also need to install the `pyarrow` to directly access Apache Arrow formatted result sets (such as using `fetch_arrow_table`).

```
pip install adbc_driver_manager pyarrow
```

> For details on the `adbc_driver_manager` package, see the `adbc_driver_manager` package documentation.

As with C++, we need to provide initialization options consisting of the location of the libduckdb shared object and entrypoint function. Notice that the `path` argument for DuckDB is passed in through the `db_kwargs` dictionary.

```
import adbc_driver_duckdb.dbapi

with adbc_driver_duckdb.dbapi.connect("test.db") as conn, conn.cursor() as cur:
    cur.execute("SELECT 42")
    # fetch a pyarrow table
    tbl = cur.fetch_arrow_table()
    print(tbl)
```

Alongside `fetch_arrow_table`, other methods from DBApi are also implemented on the cursor, such as `fetchone` and `fetchall`. Data can also be ingested via `arrow_streams`. We just need to set options on the statement to bind the stream of data and execute the query.

```
import adbc_driver_duckdb.dbapi
import pyarrow

data = pyarrow.record_batch(
    [[1, 2, 3, 4], ["a", "b", "c", "d"]],
    names = ["ints", "strs"],
)

with adbc_driver_duckdb.dbapi.connect("test.db") as conn, conn.cursor() as cur:
    cur.adbc_ingest("AnswerToEverything", data)
```

Go

Make sure to download the `libduckdb` library first (i.e., the `.so` on Linux, `.dylib` on Mac or `.dll` on Windows) from the releases page, and put it on your `LD_LIBRARY_PATH` before you run the code (but if you don't, the error will explain your options regarding the location of this file.)

The following example uses an in-memory DuckDB database to modify in-memory Arrow RecordBatches via SQL queries:

{% raw %}

```go
package main

import (
    "bytes"
    "context"
    "fmt"
    "io"

    "github.com/apache/arrow-adbc/go/adbc"
    "github.com/apache/arrow-adbc/go/adbc/drivermgr"
    "github.com/apache/arrow-go/v18/arrow"
    "github.com/apache/arrow-go/v18/arrow/array"
    "github.com/apache/arrow-go/v18/arrow/ipc"
    "github.com/apache/arrow-go/v18/arrow/memory"
)

func _makeSampleArrowRecord() arrow.Record {
    b := array.NewFloat64Builder(memory.DefaultAllocator)
    b.AppendValues([]float64{1, 2, 3}, nil)
    col := b.NewArray()

    defer col.Release()
    defer b.Release()

    schema := arrow.NewSchema([]arrow.Field{{Name: "column1", Type: arrow.PrimitiveTypes.Float64}}, nil)
    return array.NewRecord(schema, []arrow.Array{col}, int64(col.Len()))
}

type DuckDBSQLRunner struct {
    ctx  context.Context
    conn adbc.Connection
    db   adbc.Database
}

func NewDuckDBSQLRunner(ctx context.Context) (*DuckDBSQLRunner, error) {
    var drv drivermgr.Driver
    db, err := drv.NewDatabase(map[string]string{
        "driver":     "duckdb",
        "entrypoint": "duckdb_adbc_init",
        "path":       ":memory:",
    })
    if err != nil {
        return nil, fmt.Errorf("failed to create new in-memory DuckDB database: %w", err)
    }
    conn, err := db.Open(ctx)
    if err != nil {
        return nil, fmt.Errorf("failed to open connection to new in-memory DuckDB database: %w", err)
    }
    return &DuckDBSQLRunner{ctx: ctx, conn: conn, db: db}, nil
}

func serializeRecord(record arrow.Record) (io.Reader, error) {
    buf := new(bytes.Buffer)
    wr := ipc.NewWriter(buf, ipc.WithSchema(record.Schema()))
    if err := wr.Write(record); err != nil {
        return nil, fmt.Errorf("failed to write record: %w", err)
```

```
    }
    if err := wr.Close(); err != nil {
        return nil, fmt.Errorf("failed to close writer: %w", err)
    }
    return buf, nil
}

func (r *DuckDBSQLRunner) importRecord(sr io.Reader) error {
    rdr, err := ipc.NewReader(sr)
    if err != nil {
        return fmt.Errorf("failed to create IPC reader: %w", err)
    }
    defer rdr.Release()
    stmt, err := r.conn.NewStatement()
    if err != nil {
        return fmt.Errorf("failed to create new statement: %w", err)
    }
    if err := stmt.SetOption(adbc.OptionKeyIngestMode, adbc.OptionValueIngestModeCreate); err != nil {
        return fmt.Errorf("failed to set ingest mode: %w", err)
    }
    if err := stmt.SetOption(adbc.OptionKeyIngestTargetTable, "temp_table"); err != nil {
        return fmt.Errorf("failed to set ingest target table: %w", err)
    }
    if err := stmt.BindStream(r.ctx, rdr); err != nil {
        return fmt.Errorf("failed to bind stream: %w", err)
    }
    if _, err := stmt.ExecuteUpdate(r.ctx); err != nil {
        return fmt.Errorf("failed to execute update: %w", err)
    }
    return stmt.Close()
}

func (r *DuckDBSQLRunner) runSQL(sql string) ([]arrow.Record, error) {
    stmt, err := r.conn.NewStatement()
    if err != nil {
        return nil, fmt.Errorf("failed to create new statement: %w", err)
    }
    defer stmt.Close()

    if err := stmt.SetSqlQuery(sql); err != nil {
        return nil, fmt.Errorf("failed to set SQL query: %w", err)
    }
    out, n, err := stmt.ExecuteQuery(r.ctx)
    if err != nil {
        return nil, fmt.Errorf("failed to execute query: %w", err)
    }
    defer out.Release()

    result := make([]arrow.Record, 0, n)
    for out.Next() {
        rec := out.Record()
        rec.Retain() // .Next() will release the record, so we need to retain it
        result = append(result, rec)
    }
    if out.Err() != nil {
        return nil, out.Err()
    }
    return result, nil
}
```

```go
func (r *DuckDBSQLRunner) RunSQLOnRecord(record arrow.Record, sql string) ([]arrow.Record, error) {
    serializedRecord, err := serializeRecord(record)
    if err != nil {
        return nil, fmt.Errorf("failed to serialize record: %w", err)
    }
    if err := r.importRecord(serializedRecord); err != nil {
        return nil, fmt.Errorf("failed to import record: %w", err)
    }
    result, err := r.runSQL(sql)
    if err != nil {
        return nil, fmt.Errorf("failed to run SQL: %w", err)
    }

    if _, err := r.runSQL("DROP TABLE temp_table"); err != nil {
        return nil, fmt.Errorf("failed to drop temp table after running query: %w", err)
    }
    return result, nil
}

func (r *DuckDBSQLRunner) Close() {
    r.conn.Close()
    r.db.Close()
}

func main() {
    rec := _makeSampleArrowRecord()
    fmt.Println(rec)

    runner, err := NewDuckDBSQLRunner(context.Background())
    if err != nil {
        panic(err)
    }
    defer runner.Close()

    resultRecords, err := runner.RunSQLOnRecord(rec, "SELECT column1+1 FROM temp_table")
    if err != nil {
        panic(err)
    }

    for _, resultRecord := range resultRecords {
        fmt.Println(resultRecord)
        resultRecord.Release()
    }
}
```

{% endraw %}

Running it produces the following output:

```
record:
  schema:
  fields: 1
    - column1: type=float64
  rows: 3
  col[0][column1]: [1 2 3]

record:
  schema:
  fields: 1
    - (column1 + 1): type=float64, nullable
  rows: 3
```

```
col[0][(column1 + 1)]: [2 3 4]
```

C

Overview

> The latest version of the DuckDB C API is 1.3.1.

DuckDB implements a custom C API modeled somewhat following the SQLite C API. The API is contained in the `duckdb.h` header. Continue to Startup & Shutdown to get started, or check out the Full API overview.

We also provide a SQLite API wrapper which means that if your applications is programmed against the SQLite C API, you can re-link to DuckDB and it should continue working. See the `sqlite_api_wrapper` folder in our source repository for more information.

Installation

The DuckDB C API can be installed as part of the `libduckdb` packages. Please see the installation page for details.

Startup & Shutdown

To use DuckDB, you must first initialize a `duckdb_database` handle using `duckdb_open()`. `duckdb_open()` takes as parameter the database file to read and write from. The special value `NULL` (`nullptr`) can be used to create an **in-memory database**. Note that for an in-memory database no data is persisted to disk (i.e., all data is lost when you exit the process).

With the `duckdb_database` handle, you can create one or many `duckdb_connection` using `duckdb_connect()`. While individual connections are thread-safe, they will be locked during querying. It is therefore recommended that each thread uses its own connection to allow for the best parallel performance.

All `duckdb_connections` have to explicitly be disconnected with `duckdb_disconnect()` and the `duckdb_database` has to be explicitly closed with `duckdb_close()` to avoid memory and file handle leaking.

Example

```
duckdb_database db;
duckdb_connection con;

if (duckdb_open(NULL, &db) == DuckDBError) {
    // handle error
}
if (duckdb_connect(db, &con) == DuckDBError) {
    // handle error
}

// run queries...

// cleanup
duckdb_disconnect(&con);
duckdb_close(&db);
```

API Reference Overview

```
            duckdb_instance_cache duckdb_create_instance_cache();
duckdb_state duckdb_get_or_create_from_cache(duckdb_instance_cache instance_cache, const char *path,
duckdb_database *out_database, duckdb_config config, char **out_error);
void duckdb_destroy_instance_cache(duckdb_instance_cache *instance_cache);
duckdb_state duckdb_open(const char *path, duckdb_database *out_database);
duckdb_state duckdb_open_ext(const char *path, duckdb_database *out_database, duckdb_config config, char
**out_error);
void duckdb_close(duckdb_database *database);
duckdb_state duckdb_connect(duckdb_database database, duckdb_connection *out_connection);
void duckdb_interrupt(duckdb_connection connection);
duckdb_query_progress_type duckdb_query_progress(duckdb_connection connection);
void duckdb_disconnect(duckdb_connection *connection);
void duckdb_connection_get_client_context(duckdb_connection connection, duckdb_client_context *out_
context);
idx_t duckdb_client_context_get_connection_id(duckdb_client_context context);
void duckdb_destroy_client_context(duckdb_client_context *context);
const char *duckdb_library_version();
duckdb_value duckdb_get_table_names(duckdb_connection connection, const char *query, bool qualified);
```

`duckdb_create_instance_cache`

Creates a new database instance cache. The instance cache is necessary if a client/program (re)opens multiple databases to the same file within the same process. Must be destroyed with 'duckdb_destroy_instance_cache'.

Return Value The database instance cache.

Syntax

```
            duckdb_instance_cache duckdb_create_instance_cache(

);
```

`duckdb_get_or_create_from_cache`

Creates a new database instance in the instance cache, or retrieves an existing database instance. Must be closed with 'duckdb_close'.

Syntax

```
            duckdb_state duckdb_get_or_create_from_cache(
  duckdb_instance_cache instance_cache,
  const char *path,
  duckdb_database *out_database,
  duckdb_config config,
  char **out_error
);
```

Parameters

- `instance_cache`: The instance cache in which to create the database, or from which to take the database.
- `path`: Path to the database file on disk. Both `nullptr` and `:memory:` open or retrieve an in-memory database.
- `out_database`: The resulting cached database.
- `config`: (Optional) configuration used to create the database.

- `out_error`: If set and the function returns `DuckDBError`, this contains the error message. Note that the error message must be freed using `duckdb_free`.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_destroy_instance_cache`

Destroys an existing database instance cache and de-allocates its memory.

Syntax

```
            void duckdb_destroy_instance_cache(
  duckdb_instance_cache *instance_cache
);
```

Parameters

- `instance_cache`: The instance cache to destroy.

`duckdb_open`

Creates a new database or opens an existing database file stored at the given path. If no path is given a new in-memory database is created instead. The database must be closed with 'duckdb_close'.

Syntax

```
            duckdb_state duckdb_open(
  const char *path,
  duckdb_database *out_database
);
```

Parameters

- `path`: Path to the database file on disk. Both `nullptr` and `:memory:` open an in-memory database.
- `out_database`: The result database object.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_open_ext`

Extended version of duckdb_open. Creates a new database or opens an existing database file stored at the given path. The database must be closed with 'duckdb_close'.

Syntax

```
            duckdb_state duckdb_open_ext(
  const char *path,
  duckdb_database *out_database,
  duckdb_config config,
  char **out_error
);
```

Parameters

- `path`: Path to the database file on disk. Both `nullptr` and `:memory:` open an in-memory database.
- `out_database`: The result database object.
- `config`: (Optional) configuration used to start up the database.
- `out_error`: If set and the function returns `DuckDBError`, this contains the error message. Note that the error message must be freed using `duckdb_free`.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_close`

Closes the specified database and de-allocates all memory allocated for that database. This should be called after you are done with any database allocated through `duckdb_open` or `duckdb_open_ext`. Note that failing to call `duckdb_close` (in case of e.g., a program crash) will not cause data corruption. Still, it is recommended to always correctly close a database object after you are done with it.

Syntax

```
            void duckdb_close(
  duckdb_database *database
);
```

Parameters

- `database`: The database object to shut down.

`duckdb_connect`

Opens a connection to a database. Connections are required to query the database, and store transactional state associated with the connection. The instantiated connection should be closed using 'duckdb_disconnect'.

Syntax

```
            duckdb_state duckdb_connect(
  duckdb_database database,
  duckdb_connection *out_connection
);
```

Parameters

- `database`: The database file to connect to.
- `out_connection`: The result connection object.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_interrupt`

Interrupt running query

Syntax

```
            void duckdb_interrupt(
  duckdb_connection connection
);
```

Parameters

- `connection`: The connection to interrupt

`duckdb_query_progress`

Get progress of the running query

Syntax

```
            duckdb_query_progress_type duckdb_query_progress(
  duckdb_connection connection
);
```

Parameters

- `connection`: The working connection

Return Value -1 if no progress or a percentage of the progress

`duckdb_disconnect`

Closes the specified connection and de-allocates all memory allocated for that connection.

Syntax

```
            void duckdb_disconnect(
  duckdb_connection *connection
);
```

Parameters

- `connection`: The connection to close.

`duckdb_connection_get_client_context`

Retrieves the client context of the connection.

Syntax

```
            void duckdb_connection_get_client_context(
  duckdb_connection connection,
  duckdb_client_context *out_context
);
```

Parameters

- `connection`: The connection.
- `out_context`: The client context of the connection. Must be destroyed with `duckdb_destroy_client_context`.

`duckdb_client_context_get_connection_id`

Returns the connection id of the client context.

Syntax

```
            idx_t duckdb_client_context_get_connection_id(
  duckdb_client_context context
);
```

Parameters

- `context`: The client context.

Return Value The connection id of the client context.

`duckdb_destroy_client_context`

Destroys the client context and deallocates its memory.

Syntax

```
            void duckdb_destroy_client_context(
  duckdb_client_context *context
);
```

Parameters

- `context`: The client context to destroy.

`duckdb_library_version`

Returns the version of the linked DuckDB, with a version postfix for dev versions

Usually used for developing C extensions that must return this for a compatibility check.

Syntax

```
            const char *duckdb_library_version(

);
```

`duckdb_get_table_names`

Get the list of (fully qualified) table names of the query.

Syntax

```
            duckdb_value duckdb_get_table_names(
  duckdb_connection connection,
  const char *query,
  bool qualified
);
```

Parameters

- `connection`: The connection for which to get the table names.
- `query`: The query for which to get the table names.
- `qualified`: Returns fully qualified table names (catalog.schema.table), if set to true, else only the (not escaped) table names.

Return Value A duckdb_value of type VARCHAR[] containing the (fully qualified) table names of the query. Must be destroyed with duckdb_destroy_value.

Configuration

Configuration options can be provided to change different settings of the database system. Note that many of these settings can be changed later on using PRAGMA statements as well. The configuration object should be created, filled with values and passed to `duckdb_open_ext`.

Example

```
duckdb_database db;
duckdb_config config;

// create the configuration object
if (duckdb_create_config(&config) == DuckDBError) {
    // handle error
}
// set some configuration options
duckdb_set_config(config, "access_mode", "READ_WRITE"); // or READ_ONLY
duckdb_set_config(config, "threads", "8");
duckdb_set_config(config, "max_memory", "8GB");
duckdb_set_config(config, "default_order", "DESC");

// open the database using the configuration
if (duckdb_open_ext(NULL, &db, config, NULL) == DuckDBError) {
    // handle error
}
// cleanup the configuration object
duckdb_destroy_config(&config);

// run queries...

// cleanup
duckdb_close(&db);
```

API Reference Overview

```
            duckdb_state duckdb_create_config(duckdb_config *out_config);
```

```
size_t duckdb_config_count();
duckdb_state duckdb_get_config_flag(size_t index, const char **out_name, const char **out_description);
duckdb_state duckdb_set_config(duckdb_config config, const char *name, const char *option);
void duckdb_destroy_config(duckdb_config *config);
```

duckdb_create_config

Initializes an empty configuration object that can be used to provide start-up options for the DuckDB instance through duckdb_open_ext. The duckdb_config must be destroyed using 'duckdb_destroy_config'

This will always succeed unless there is a malloc failure.

Note that duckdb_destroy_config should always be called on the resulting config, even if the function returns DuckDBError.

Syntax

```
            duckdb_state duckdb_create_config(
  duckdb_config *out_config
);
```

Parameters

- out_config: The result configuration object.

Return Value DuckDBSuccess on success or DuckDBError on failure.

duckdb_config_count

This returns the total amount of configuration options available for usage with duckdb_get_config_flag.

This should not be called in a loop as it internally loops over all the options.

Return Value The amount of config options available.

Syntax

```
            size_t duckdb_config_count(

);
```

duckdb_get_config_flag

Obtains a human-readable name and description of a specific configuration option. This can be used to e.g. display configuration options. This will succeed unless index is out of range (i.e., >= duckdb_config_count).

The result name or description MUST NOT be freed.

Syntax

```
            duckdb_state duckdb_get_config_flag(
  size_t index,
  const char **out_name,
  const char **out_description
);
```

Parameters

- `index`: The index of the configuration option (between 0 and `duckdb_config_count`)
- `out_name`: A name of the configuration flag.
- `out_description`: A description of the configuration flag.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_set_config`

Sets the specified option for the specified configuration. The configuration option is indicated by name. To obtain a list of config options, see `duckdb_get_config_flag`.

In the source code, configuration options are defined in `config.cpp`.

This can fail if either the name is invalid, or if the value provided for the option is invalid.

Syntax

```
           duckdb_state duckdb_set_config(
  duckdb_config config,
  const char *name,
  const char *option
);
```

Parameters

- `config`: The configuration object to set the option on.
- `name`: The name of the configuration flag to set.
- `option`: The value to set the configuration flag to.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_destroy_config`

Destroys the specified configuration object and de-allocates all memory allocated for the object.

Syntax

```
           void duckdb_destroy_config(
  duckdb_config *config
);
```

Parameters

- `config`: The configuration object to destroy.

Query

The `duckdb_query` method allows SQL queries to be run in DuckDB from C. This method takes two parameters, a (null-terminated) SQL query string and a `duckdb_result` result pointer. The result pointer may be `NULL` if the application is not interested in the result set or if the query produces no result. After the result is consumed, the `duckdb_destroy_result` method should be used to clean up the result.

Elements can be extracted from the `duckdb_result` object using a variety of methods. The `duckdb_column_count` can be used to extract the number of columns. `duckdb_column_name` and `duckdb_column_type` can be used to extract the names and types of individual columns.

Example

```
duckdb_state state;
duckdb_result result;

// create a table
state = duckdb_query(con, "CREATE TABLE integers (i INTEGER, j INTEGER);", NULL);
if (state == DuckDBError) {
    // handle error
}
// insert three rows into the table
state = duckdb_query(con, "INSERT INTO integers VALUES (3, 4), (5, 6), (7, NULL);", NULL);
if (state == DuckDBError) {
    // handle error
}
// query rows again
state = duckdb_query(con, "SELECT * FROM integers", &result);
if (state == DuckDBError) {
    // handle error
}
// handle the result
// ...

// destroy the result after we are done with it
duckdb_destroy_result(&result);
```

Value Extraction

Values can be extracted using either the `duckdb_fetch_chunk` function, or using the `duckdb_value` convenience functions. The `duckdb_fetch_chunk` function directly hands you data chunks in DuckDB's native array format and can therefore be very fast. The `duckdb_value` functions perform bounds- and type-checking, and will automatically cast values to the desired type. This makes them more convenient and easier to use, at the expense of being slower.

See the Types page for more information.

> For optimal performance, use `duckdb_fetch_chunk` to extract data from the query result. The `duckdb_value` functions perform internal type-checking, bounds-checking and casting which makes them slower.

`duckdb_fetch_chunk`

Below is an end-to-end example that prints the above result to CSV format using the `duckdb_fetch_chunk` function. Note that the function is NOT generic: we do need to know exactly what the types of the result columns are.

```
duckdb_database db;
duckdb_connection con;
duckdb_open(nullptr, &db);
duckdb_connect(db, &con);

duckdb_result res;
duckdb_query(con, "CREATE TABLE integers (i INTEGER, j INTEGER);", NULL);
duckdb_query(con, "INSERT INTO integers VALUES (3, 4), (5, 6), (7, NULL);", NULL);
duckdb_query(con, "SELECT * FROM integers;", &res);

// iterate until result is exhausted
while (true) {
    duckdb_data_chunk result = duckdb_fetch_chunk(res);
    if (!result) {
        // result is exhausted
        break;
    }
    // get the number of rows from the data chunk
    idx_t row_count = duckdb_data_chunk_get_size(result);
    // get the first column
    duckdb_vector col1 = duckdb_data_chunk_get_vector(result, 0);
    int32_t *col1_data = (int32_t *) duckdb_vector_get_data(col1);
    uint64_t *col1_validity = duckdb_vector_get_validity(col1);

    // get the second column
    duckdb_vector col2 = duckdb_data_chunk_get_vector(result, 1);
    int32_t *col2_data = (int32_t *) duckdb_vector_get_data(col2);
    uint64_t *col2_validity = duckdb_vector_get_validity(col2);

    // iterate over the rows
    for (idx_t row = 0; row < row_count; row++) {
        if (duckdb_validity_row_is_valid(col1_validity, row)) {
            printf("%d", col1_data[row]);
        } else {
            printf("NULL");
        }
        printf(",");
        if (duckdb_validity_row_is_valid(col2_validity, row)) {
            printf("%d", col2_data[row]);
        } else {
            printf("NULL");
        }
        printf("\n");
    }
    duckdb_destroy_data_chunk(&result);
}
// clean-up
duckdb_destroy_result(&res);
duckdb_disconnect(&con);
duckdb_close(&db);
```

This prints the following result:

```
3,4
5,6
7,NULL
```

duckdb_value

> **Deprecated.** The `duckdb_value` functions are deprecated and are scheduled for removal in a future release.

Below is an example that prints the above result to CSV format using the `duckdb_value_varchar` function. Note that the function is generic: we do not need to know about the types of the individual result columns.

```
// print the above result to CSV format using `duckdb_value_varchar`
idx_t row_count = duckdb_row_count(&result);
idx_t column_count = duckdb_column_count(&result);
for (idx_t row = 0; row < row_count; row++) {
    for (idx_t col = 0; col < column_count; col++) {
        if (col > 0) printf(",");
        auto str_val = duckdb_value_varchar(&result, col, row);
        printf("%s", str_val);
        duckdb_free(str_val);
   }
   printf("\n");
}
```

API Reference Overview

```
            duckdb_state duckdb_query(duckdb_connection connection, const char *query, duckdb_result
*out_result);
void duckdb_destroy_result(duckdb_result *result);
const char *duckdb_column_name(duckdb_result *result, idx_t col);
duckdb_type duckdb_column_type(duckdb_result *result, idx_t col);
duckdb_statement_type duckdb_result_statement_type(duckdb_result result);
duckdb_logical_type duckdb_column_logical_type(duckdb_result *result, idx_t col);
idx_t duckdb_column_count(duckdb_result *result);
idx_t duckdb_row_count(duckdb_result *result);
idx_t duckdb_rows_changed(duckdb_result *result);
void *duckdb_column_data(duckdb_result *result, idx_t col);
bool *duckdb_nullmask_data(duckdb_result *result, idx_t col);
const char *duckdb_result_error(duckdb_result *result);
duckdb_error_type duckdb_result_error_type(duckdb_result *result);
```

`duckdb_query`

Executes a SQL query within a connection and stores the full (materialized) result in the out_result pointer. If the query fails to execute, DuckDBError is returned and the error message can be retrieved by calling `duckdb_result_error`.

Note that after running `duckdb_query`, `duckdb_destroy_result` must be called on the result object even if the query fails, otherwise the error stored within the result will not be freed correctly.

Syntax

```
            duckdb_state duckdb_query(
  duckdb_connection connection,
  const char *query,
  duckdb_result *out_result
);
```

Parameters

- `connection`: The connection to perform the query in.
- `query`: The SQL query to run.
- `out_result`: The query result.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_destroy_result`

Closes the result and de-allocates all memory allocated for that result.

Syntax

```
            void duckdb_destroy_result(
  duckdb_result *result
);
```

Parameters

- `result`: The result to destroy.

`duckdb_column_name`

Returns the column name of the specified column. The result should not need to be freed; the column names will automatically be destroyed when the result is destroyed.

Returns `NULL` if the column is out of range.

Syntax

```
            const char *duckdb_column_name(
  duckdb_result *result,
  idx_t col
);
```

Parameters

- `result`: The result object to fetch the column name from.
- `col`: The column index.

Return Value The column name of the specified column.

`duckdb_column_type`

Returns the column type of the specified column.

Returns `DUCKDB_TYPE_INVALID` if the column is out of range.

Syntax

```
            duckdb_type duckdb_column_type(
  duckdb_result *result,
  idx_t col
);
```

Parameters

- `result`: The result object to fetch the column type from.
- `col`: The column index.

Return Value The column type of the specified column.

`duckdb_result_statement_type`

Returns the statement type of the statement that was executed

Syntax

```
            duckdb_statement_type duckdb_result_statement_type(
  duckdb_result result
);
```

Parameters

- `result`: The result object to fetch the statement type from.

Return Value duckdb_statement_type value or DUCKDB_STATEMENT_TYPE_INVALID

`duckdb_column_logical_type`

Returns the logical column type of the specified column.

The return type of this call should be destroyed with `duckdb_destroy_logical_type`.

Returns `NULL` if the column is out of range.

Syntax

```
            duckdb_logical_type duckdb_column_logical_type(
  duckdb_result *result,
  idx_t col
);
```

Parameters

- `result`: The result object to fetch the column type from.
- `col`: The column index.

Return Value The logical column type of the specified column.

`duckdb_column_count`

Returns the number of columns present in a the result object.

Syntax

```
            idx_t duckdb_column_count(
  duckdb_result *result
);
```

Parameters

- `result`: The result object.

Return Value The number of columns present in the result object.

`duckdb_row_count`

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Returns the number of rows present in the result object.

Syntax

```
            idx_t duckdb_row_count(
  duckdb_result *result
);
```

Parameters

- `result`: The result object.

Return Value The number of rows present in the result object.

`duckdb_rows_changed`

Returns the number of rows changed by the query stored in the result. This is relevant only for INSERT/UPDATE/DELETE queries. For other queries the rows_changed will be 0.

Syntax

```
            idx_t duckdb_rows_changed(
  duckdb_result *result
);
```

Parameters

- `result`: The result object.

Return Value The number of rows changed.

`duckdb_column_data`

> **Deprecated.** This method has been deprecated. Prefer using `duckdb_result_get_chunk` instead.

Returns the data of a specific column of a result in columnar format.

The function returns a dense array which contains the result data. The exact type stored in the array depends on the corresponding duckdb_type (as provided by `duckdb_column_type`). For the exact type by which the data should be accessed, see the comments in the types section or the DUCKDB_TYPE enum.

For example, for a column of type DUCKDB_TYPE_INTEGER, rows can be accessed in the following manner:

```
int32_t *data = (int32_t *) duckdb_column_data(&result, 0);
printf("Data for row %d: %d\n", row, data[row]);
```

Syntax

```
            void *duckdb_column_data(
  duckdb_result *result,
  idx_t col
);
```

Parameters

- `result`: The result object to fetch the column data from.
- `col`: The column index.

Return Value The column data of the specified column.

`duckdb_nullmask_data`

> **Deprecated.** This method has been deprecated. Prefer using `duckdb_result_get_chunk` instead.

Returns the nullmask of a specific column of a result in columnar format. The nullmask indicates for every row whether or not the corresponding row is `NULL`. If a row is `NULL`, the values present in the array provided by `duckdb_column_data` are undefined.

```
int32_t *data = (int32_t *) duckdb_column_data(&result, 0);
bool *nullmask = duckdb_nullmask_data(&result, 0);
if (nullmask[row]) {
printf("Data for row %d: NULL\n", row);
} else {
printf("Data for row %d: %d\n", row, data[row]);
}
```

Syntax

```
            bool *duckdb_nullmask_data(
  duckdb_result *result,
  idx_t col
);
```

Parameters

- `result`: The result object to fetch the nullmask from.
- `col`: The column index.

Return Value The nullmask of the specified column.

`duckdb_result_error`

Returns the error message contained within the result. The error is only set if `duckdb_query` returns `DuckDBError`.

The result of this function must not be freed. It will be cleaned up when `duckdb_destroy_result` is called.

Syntax

```
            const char *duckdb_result_error(
  duckdb_result *result
);
```

Parameters

- `result`: The result object to fetch the error from.

Return Value The error of the result.

`duckdb_result_error_type`

Returns the result error type contained within the result. The error is only set if `duckdb_query` returns `DuckDBError`.

Syntax

```
            duckdb_error_type duckdb_result_error_type(
  duckdb_result *result
);
```

Parameters

- `result`: The result object to fetch the error from.

Return Value The error type of the result.

Data Chunks

Data chunks represent a horizontal slice of a table. They hold a number of vectors, that can each hold up to the `VECTOR_SIZE` rows. The vector size can be obtained through the `duckdb_vector_size` function and is configurable, but is usually set to `2048`.

Data chunks and vectors are what DuckDB uses natively to store and represent data. For this reason, the data chunk interface is the most efficient way of interfacing with DuckDB. Be aware, however, that correctly interfacing with DuckDB using the data chunk API does require knowledge of DuckDB's internal vector format.

Data chunks can be used in two manners:

- **Reading Data**: Data chunks can be obtained from query results using the `duckdb_fetch_chunk` method, or as input to a user-defined function. In this case, the vector methods can be used to read individual values.
- **Writing Data**: Data chunks can be created using `duckdb_create_data_chunk`. The data chunk can then be filled with values and used in `duckdb_append_data_chunk` to write data to the database.

The primary manner of interfacing with data chunks is by obtaining the internal vectors of the data chunk using the `duckdb_data_chunk_get_vector` method. Afterwards, the vector methods can be used to read from or write to the individual vectors.

API Reference Overview

```
            duckdb_data_chunk duckdb_create_data_chunk(duckdb_logical_type *types, idx_t column_count);
void duckdb_destroy_data_chunk(duckdb_data_chunk *chunk);
void duckdb_data_chunk_reset(duckdb_data_chunk chunk);
idx_t duckdb_data_chunk_get_column_count(duckdb_data_chunk chunk);
duckdb_vector duckdb_data_chunk_get_vector(duckdb_data_chunk chunk, idx_t col_idx);
idx_t duckdb_data_chunk_get_size(duckdb_data_chunk chunk);
void duckdb_data_chunk_set_size(duckdb_data_chunk chunk, idx_t size);
```

duckdb_create_data_chunk

Creates an empty data chunk with the specified column types. The result must be destroyed with `duckdb_destroy_data_chunk`.

Syntax

```
            duckdb_data_chunk duckdb_create_data_chunk(
  duckdb_logical_type *types,
  idx_t column_count
);
```

Parameters

- `types`: An array of column types. Column types can not contain ANY and INVALID types.
- `column_count`: The number of columns.

Return Value The data chunk.

duckdb_destroy_data_chunk

Destroys the data chunk and de-allocates all memory allocated for that chunk.

Syntax

```
            void duckdb_destroy_data_chunk(
  duckdb_data_chunk *chunk
);
```

Parameters

- `chunk`: The data chunk to destroy.

duckdb_data_chunk_reset

Resets a data chunk, clearing the validity masks and setting the cardinality of the data chunk to 0. After calling this method, you must call `duckdb_vector_get_validity` and `duckdb_vector_get_data` to obtain current data and validity pointers

Syntax

```
            void duckdb_data_chunk_reset(
  duckdb_data_chunk chunk
);
```

Parameters

- `chunk`: The data chunk to reset.

duckdb_data_chunk_get_column_count

Retrieves the number of columns in a data chunk.

Syntax

```
            idx_t duckdb_data_chunk_get_column_count(
  duckdb_data_chunk chunk
);
```

Parameters

- chunk: The data chunk to get the data from

Return Value The number of columns in the data chunk

duckdb_data_chunk_get_vector

Retrieves the vector at the specified column index in the data chunk.

The pointer to the vector is valid for as long as the chunk is alive. It does NOT need to be destroyed.

Syntax

```
            duckdb_vector duckdb_data_chunk_get_vector(
  duckdb_data_chunk chunk,
  idx_t col_idx
);
```

Parameters

- chunk: The data chunk to get the data from

Return Value The vector

duckdb_data_chunk_get_size

Retrieves the current number of tuples in a data chunk.

Syntax

```
            idx_t duckdb_data_chunk_get_size(
  duckdb_data_chunk chunk
);
```

Parameters

- chunk: The data chunk to get the data from

Return Value The number of tuples in the data chunk

duckdb_data_chunk_set_size

Sets the current number of tuples in a data chunk.

Syntax

```
           void duckdb_data_chunk_set_size(
  duckdb_data_chunk chunk,
  idx_t size
);
```

Parameters

- `chunk`: The data chunk to set the size in
- `size`: The number of tuples in the data chunk

Vectors

Vectors represent a horizontal slice of a column. They hold a number of values of a specific type, similar to an array. Vectors are the core data representation used in DuckDB. Vectors are typically stored within data chunks.

The vector and data chunk interfaces are the most efficient way of interacting with DuckDB, allowing for the highest performance. However, the interfaces are also difficult to use and care must be taken when using them.

Vector Format

Vectors are arrays of a specific data type. The logical type of a vector can be obtained using `duckdb_vector_get_column_type`. The type id of the logical type can then be obtained using `duckdb_get_type_id`.

Vectors themselves do not have sizes. Instead, the parent data chunk has a size (that can be obtained through `duckdb_data_chunk_get_size`). All vectors that belong to a data chunk have the same size.

Primitive Types

For primitive types, the underlying array can be obtained using the `duckdb_vector_get_data` method. The array can then be accessed using the correct native type. Below is a table that contains a mapping of the `duckdb_type` to the native type of the array.

duckdb_type	NativeType
DUCKDB_TYPE_BOOLEAN	bool
DUCKDB_TYPE_TINYINT	int8_t
DUCKDB_TYPE_SMALLINT	int16_t
DUCKDB_TYPE_INTEGER	int32_t
DUCKDB_TYPE_BIGINT	int64_t
DUCKDB_TYPE_UTINYINT	uint8_t
DUCKDB_TYPE_USMALLINT	uint16_t
DUCKDB_TYPE_UINTEGER	uint32_t
DUCKDB_TYPE_UBIGINT	uint64_t
DUCKDB_TYPE_FLOAT	float
DUCKDB_TYPE_DOUBLE	double
DUCKDB_TYPE_TIMESTAMP	duckdb_timestamp
DUCKDB_TYPE_DATE	duckdb_date

duckdb_type	NativeType
DUCKDB_TYPE_TIME	duckdb_time
DUCKDB_TYPE_INTERVAL	duckdb_interval
DUCKDB_TYPE_HUGEINT	duckdb_hugeint
DUCKDB_TYPE_UHUGEINT	duckdb_uhugeint
DUCKDB_TYPE_VARCHAR	duckdb_string_t
DUCKDB_TYPE_BLOB	duckdb_string_t
DUCKDB_TYPE_TIMESTAMP_S	duckdb_timestamp
DUCKDB_TYPE_TIMESTAMP_MS	duckdb_timestamp
DUCKDB_TYPE_TIMESTAMP_NS	duckdb_timestamp
DUCKDB_TYPE_UUID	duckdb_hugeint
DUCKDB_TYPE_TIME_TZ	duckdb_time_tz
DUCKDB_TYPE_TIMESTAMP_TZ	duckdb_timestamp

NULL Values

Any value in a vector can be `NULL`. When a value is `NULL`, the values contained within the primary array at that index is undefined (and can be uninitialized). The validity mask is a bitmask consisting of `uint64_t` elements. For every 64 values in the vector, one `uint64_t` element exists (rounded up). The validity mask has its bit set to 1 if the value is valid, or set to 0 if the value is invalid (i.e .`NULL`).

The bits of the bitmask can be read directly, or the slower helper method `duckdb_validity_row_is_valid` can be used to check whether or not a value is `NULL`.

The `duckdb_vector_get_validity` returns a pointer to the validity mask. Note that if all values in a vector are valid, this function **might** return `nullptr` in which case the validity mask does not need to be checked.

Strings

String values are stored as a `duckdb_string_t`. This is a special struct that stores the string inline (if it is short, i.e., `<= 12 bytes`) or a pointer to the string data if it is longer than `12` bytes.

```
typedef struct {
    union {
        struct {
            uint32_t length;
            char prefix[4];
            char *ptr;
        } pointer;
        struct {
            uint32_t length;
            char inlined[12];
        } inlined;
    } value;
} duckdb_string_t;
```

The length can either be accessed directly, or the `duckdb_string_is_inlined` can be used to check if a string is inlined.

Decimals

Decimals are stored as integer values internally. The exact native type depends on the `width` of the decimal type, as shown in the following table:

Width	NativeType
<= 4	int16_t
<= 9	int32_t
<= 18	int64_t
<= 38	duckdb_hugeint

The `duckdb_decimal_internal_type` can be used to obtain the internal type of the decimal.

Decimals are stored as integer values multiplied by `10^scale`. The scale of a decimal can be obtained using `duckdb_decimal_scale`. For example, a decimal value of `10.5` with type `DECIMAL(8, 3)` is stored internally as an `int32_t` value of `10500`. In order to obtain the correct decimal value, the value should be divided by the appropriate power-of-ten.

Enums

Enums are stored as unsigned integer values internally. The exact native type depends on the size of the enum dictionary, as shown in the following table:

Dictionary size	NativeType
<= 255	uint8_t
<= 65535	uint16_t
<= 4294967295	uint32_t

The `duckdb_enum_internal_type` can be used to obtain the internal type of the enum.

In order to obtain the actual string value of the enum, the `duckdb_enum_dictionary_value` function must be used to obtain the enum value that corresponds to the given dictionary entry. Note that the enum dictionary is the same for the entire column – and so only needs to be constructed once.

Structs

Structs are nested types that contain any number of child types. Think of them like a `struct` in C. The way to access struct data using vectors is to access the child vectors recursively using the `duckdb_struct_vector_get_child` method.

The struct vector itself does not have any data (i.e., you should not use `duckdb_vector_get_data` method on the struct). **However**, the struct vector itself **does** have a validity mask. The reason for this is that the child elements of a struct can be `NULL`, but the struct **itself** can also be `NULL`.

Lists

Lists are nested types that contain a single child type, repeated `x` times per row. Think of them like a variable-length array in C. The way to access list data using vectors is to access the child vector using the `duckdb_list_vector_get_child` method.

The `duckdb_vector_get_data` must be used to get the offsets and lengths of the lists stored as `duckdb_list_entry`, that can then be applied to the child vector.

```
typedef struct {
    uint64_t offset;
    uint64_t length;
} duckdb_list_entry;
```

Note that both list entries itself **and** any children stored in the lists can also be `NULL`. This must be checked using the validity mask again.

Arrays

Arrays are nested types that contain a single child type, repeated exactly `array_size` times per row. Think of them like a fixed-size array in C. Arrays work exactly the same as lists, **except** the length and offset of each entry is fixed. The fixed array size can be obtained by using `duckdb_array_type_array_size`. The data for entry n then resides at `offset = n * array_size`, and always has `length = array_size`.

Note that much like lists, arrays can still be `NULL`, which must be checked using the validity mask.

Examples

Below are several full end-to-end examples of how to interact with vectors.

Example: Reading an int64 Vector with `NULL` Values

```
duckdb_database db;
duckdb_connection con;
duckdb_open(nullptr, &db);
duckdb_connect(db, &con);

duckdb_result res;
duckdb_query(con, "SELECT CASE WHEN i%2=0 THEN NULL ELSE i END res_col FROM range(10) t(i)", &res);

// iterate until result is exhausted
while (true) {
    duckdb_data_chunk result = duckdb_fetch_chunk(res);
    if (!result) {
        // result is exhausted
        break;
    }
    // get the number of rows from the data chunk
    idx_t row_count = duckdb_data_chunk_get_size(result);
    // get the first column
    duckdb_vector res_col = duckdb_data_chunk_get_vector(result, 0);
    // get the native array and the validity mask of the vector
    int64_t *vector_data = (int64_t *) duckdb_vector_get_data(res_col);
    uint64_t *vector_validity = duckdb_vector_get_validity(res_col);
    // iterate over the rows
    for (idx_t row = 0; row < row_count; row++) {
        if (duckdb_validity_row_is_valid(vector_validity, row)) {
            printf("%lld\n", vector_data[row]);
        } else {
            printf("NULL\n");
        }
    }
    duckdb_destroy_data_chunk(&result);
}
// clean-up
duckdb_destroy_result(&res);
```

```
duckdb_disconnect(&con);
duckdb_close(&db);
```

Example: Reading a String Vector

```
duckdb_database db;
duckdb_connection con;
duckdb_open(nullptr, &db);
duckdb_connect(db, &con);

duckdb_result res;
duckdb_query(con, "SELECT CASE WHEN i%2=0 THEN CONCAT('short_', i) ELSE CONCAT('longstringprefix', i)
END FROM range(10) t(i)", &res);

// iterate until result is exhausted
while (true) {
    duckdb_data_chunk result = duckdb_fetch_chunk(res);
    if (!result) {
        // result is exhausted
        break;
    }
    // get the number of rows from the data chunk
    idx_t row_count = duckdb_data_chunk_get_size(result);
    // get the first column
    duckdb_vector res_col = duckdb_data_chunk_get_vector(result, 0);
    // get the native array and the validity mask of the vector
    duckdb_string_t *vector_data = (duckdb_string_t *) duckdb_vector_get_data(res_col);
    uint64_t *vector_validity = duckdb_vector_get_validity(res_col);
    // iterate over the rows
    for (idx_t row = 0; row < row_count; row++) {
        if (duckdb_validity_row_is_valid(vector_validity, row)) {
            duckdb_string_t str = vector_data[row];
            if (duckdb_string_is_inlined(str)) {
                // use inlined string
                printf("%.*s\n", str.value.inlined.length, str.value.inlined.inlined);
            } else {
                // follow string pointer
                printf("%.*s\n", str.value.pointer.length, str.value.pointer.ptr);
            }
        } else {
            printf("NULL\n");
        }
    }
    duckdb_destroy_data_chunk(&result);
}
// clean-up
duckdb_destroy_result(&res);
duckdb_disconnect(&con);
duckdb_close(&db);
```

Example: Reading a Struct Vector

```
duckdb_database db;
duckdb_connection con;
duckdb_open(nullptr, &db);
duckdb_connect(db, &con);
```

```
duckdb_result res;
duckdb_query(con, "SELECT CASE WHEN i%5=0 THEN NULL ELSE {'col1': i, 'col2': CASE WHEN i%2=0 THEN NULL
ELSE 100 + i * 42 END} END FROM range(10) t(i)", &res);

// iterate until result is exhausted
while (true) {
    duckdb_data_chunk result = duckdb_fetch_chunk(res);
    if (!result) {
        // result is exhausted
        break;
    }
    // get the number of rows from the data chunk
    idx_t row_count = duckdb_data_chunk_get_size(result);
    // get the struct column
    duckdb_vector struct_col = duckdb_data_chunk_get_vector(result, 0);
    uint64_t *struct_validity = duckdb_vector_get_validity(struct_col);
    // get the child columns of the struct
    duckdb_vector col1_vector = duckdb_struct_vector_get_child(struct_col, 0);
    int64_t *col1_data = (int64_t *) duckdb_vector_get_data(col1_vector);
    uint64_t *col1_validity = duckdb_vector_get_validity(col1_vector);

    duckdb_vector col2_vector = duckdb_struct_vector_get_child(struct_col, 1);
    int64_t *col2_data = (int64_t *) duckdb_vector_get_data(col2_vector);
    uint64_t *col2_validity = duckdb_vector_get_validity(col2_vector);

    // iterate over the rows
    for (idx_t row = 0; row < row_count; row++) {
        if (!duckdb_validity_row_is_valid(struct_validity, row)) {
            // entire struct is NULL
            printf("NULL\n");
            continue;
        }
        // read col1
        printf("{'col1': ");
        if (!duckdb_validity_row_is_valid(col1_validity, row)) {
            // col1 is NULL
            printf("NULL");
        } else {
            printf("%lld", col1_data[row]);
        }
        printf(", 'col2': ");
        if (!duckdb_validity_row_is_valid(col2_validity, row)) {
            // col2 is NULL
            printf("NULL");
        } else {
            printf("%lld", col2_data[row]);
        }
        printf("}\n");
    }
    duckdb_destroy_data_chunk(&result);
}
// clean-up
duckdb_destroy_result(&res);
duckdb_disconnect(&con);
duckdb_close(&db);
```

Example: Reading a List Vector

```
duckdb_database db;
```

```
duckdb_connection con;
duckdb_open(nullptr, &db);
duckdb_connect(db, &con);

duckdb_result res;
duckdb_query(con, "SELECT CASE WHEN i % 5 = 0 THEN NULL WHEN i % 2 = 0 THEN [i, i + 1] ELSE [i * 42,
NULL, i * 84] END FROM range(10) t(i)", &res);

// iterate until result is exhausted
while (true) {
    duckdb_data_chunk result = duckdb_fetch_chunk(res);
    if (!result) {
        // result is exhausted
        break;
    }
    // get the number of rows from the data chunk
    idx_t row_count = duckdb_data_chunk_get_size(result);
    // get the list column
    duckdb_vector list_col = duckdb_data_chunk_get_vector(result, 0);
    duckdb_list_entry *list_data = (duckdb_list_entry *) duckdb_vector_get_data(list_col);
    uint64_t *list_validity = duckdb_vector_get_validity(list_col);
    // get the child column of the list
    duckdb_vector list_child = duckdb_list_vector_get_child(list_col);
    int64_t *child_data = (int64_t *) duckdb_vector_get_data(list_child);
    uint64_t *child_validity = duckdb_vector_get_validity(list_child);

    // iterate over the rows
    for (idx_t row = 0; row < row_count; row++) {
        if (!duckdb_validity_row_is_valid(list_validity, row)) {
            // entire list is NULL
            printf("NULL\n");
            continue;
        }
        // read the list offsets for this row
        duckdb_list_entry list = list_data[row];
        printf("[");
        for (idx_t child_idx = list.offset; child_idx < list.offset + list.length; child_idx++) {
            if (child_idx > list.offset) {
                printf(", ");
            }
            if (!duckdb_validity_row_is_valid(child_validity, child_idx)) {
                // col1 is NULL
                printf("NULL");
            } else {
                printf("%lld", child_data[child_idx]);
            }
        }
        printf("]\n");
    }
    duckdb_destroy_data_chunk(&result);
}
// clean-up
duckdb_destroy_result(&res);
duckdb_disconnect(&con);
duckdb_close(&db);
```

API Reference Overview

```
            duckdb_vector duckdb_create_vector(duckdb_logical_type type, idx_t capacity);
void duckdb_destroy_vector(duckdb_vector *vector);
duckdb_logical_type duckdb_vector_get_column_type(duckdb_vector vector);
void *duckdb_vector_get_data(duckdb_vector vector);
uint64_t *duckdb_vector_get_validity(duckdb_vector vector);
void duckdb_vector_ensure_validity_writable(duckdb_vector vector);
void duckdb_vector_assign_string_element(duckdb_vector vector, idx_t index, const char *str);
void duckdb_vector_assign_string_element_len(duckdb_vector vector, idx_t index, const char *str, idx_t
str_len);
duckdb_vector duckdb_list_vector_get_child(duckdb_vector vector);
idx_t duckdb_list_vector_get_size(duckdb_vector vector);
duckdb_state duckdb_list_vector_set_size(duckdb_vector vector, idx_t size);
duckdb_state duckdb_list_vector_reserve(duckdb_vector vector, idx_t required_capacity);
duckdb_vector duckdb_struct_vector_get_child(duckdb_vector vector, idx_t index);
duckdb_vector duckdb_array_vector_get_child(duckdb_vector vector);
void duckdb_slice_vector(duckdb_vector vector, duckdb_selection_vector selection, idx_t len);
void duckdb_vector_reference_value(duckdb_vector vector, duckdb_value value);
void duckdb_vector_reference_vector(duckdb_vector to_vector, duckdb_vector from_vector);
```

Validity Mask Functions

```
            bool duckdb_validity_row_is_valid(uint64_t *validity, idx_t row);
void duckdb_validity_set_row_validity(uint64_t *validity, idx_t row, bool valid);
void duckdb_validity_set_row_invalid(uint64_t *validity, idx_t row);
void duckdb_validity_set_row_valid(uint64_t *validity, idx_t row);
```

`duckdb_create_vector`

Creates a flat vector.

Syntax

```
            duckdb_vector duckdb_create_vector(
  duckdb_logical_type type,
  idx_t capacity
);
```

`duckdb_destroy_vector`

Destroys the vector and de-allocates all memory allocated for that vector, if unused else where.

Syntax

```
            void duckdb_destroy_vector(
  duckdb_vector *vector
);
```

`duckdb_vector_get_column_type`

Retrieves the column type of the specified vector.

The result must be destroyed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_vector_get_column_type(
  duckdb_vector vector
);
```

Parameters

- `vector`: The vector get the data from

Return Value The type of the vector

`duckdb_vector_get_data`

Retrieves the data pointer of the vector.

The data pointer can be used to read or write values from the vector. How to read or write values depends on the type of the vector.

Syntax

```
            void *duckdb_vector_get_data(
  duckdb_vector vector
);
```

Parameters

- `vector`: The vector to get the data from

Return Value The data pointer

`duckdb_vector_get_validity`

Retrieves the validity mask pointer of the specified vector.

If all values are valid, this function MIGHT return NULL!

The validity mask is a bitset that signifies null-ness within the data chunk. It is a series of uint64_t values, where each uint64_t value contains validity for 64 tuples. The bit is set to 1 if the value is valid (i.e., not NULL) or 0 if the value is invalid (i.e., NULL).

Validity of a specific value can be obtained like this:

idx_t entry_idx = row_idx / 64; idx_t idx_in_entry = row_idx % 64; bool is_valid = validity_mask[entry_idx] & (1 « idx_in_entry);

Alternatively, the (slower) duckdb_validity_row_is_valid function can be used.

Syntax

```
            uint64_t *duckdb_vector_get_validity(
  duckdb_vector vector
);
```

Parameters

- `vector`: The vector to get the data from

Return Value The pointer to the validity mask, or NULL if no validity mask is present

`duckdb_vector_ensure_validity_writable`

Ensures the validity mask is writable by allocating it.

After this function is called, `duckdb_vector_get_validity` will ALWAYS return non-NULL. This allows NULL values to be written to the vector, regardless of whether a validity mask was present before.

Syntax

```
            void duckdb_vector_ensure_validity_writable(
  duckdb_vector vector
);
```

Parameters

- `vector`: The vector to alter

`duckdb_vector_assign_string_element`

Assigns a string element in the vector at the specified location.

Syntax

```
            void duckdb_vector_assign_string_element(
  duckdb_vector vector,
  idx_t index,
  const char *str
);
```

Parameters

- `vector`: The vector to alter
- `index`: The row position in the vector to assign the string to
- `str`: The null-terminated string

`duckdb_vector_assign_string_element_len`

Assigns a string element in the vector at the specified location. You may also use this function to assign BLOBs.

Syntax

```
            void duckdb_vector_assign_string_element_len(
  duckdb_vector vector,
  idx_t index,
  const char *str,
  idx_t str_len
);
```

Parameters

- `vector`: The vector to alter
- `index`: The row position in the vector to assign the string to
- `str`: The string
- `str_len`: The length of the string (in bytes)

duckdb_list_vector_get_child

Retrieves the child vector of a list vector.

The resulting vector is valid as long as the parent vector is valid.

Syntax

```
            duckdb_vector duckdb_list_vector_get_child(
  duckdb_vector vector
);
```

Parameters

- `vector`: The vector

Return Value The child vector

duckdb_list_vector_get_size

Returns the size of the child vector of the list.

Syntax

```
            idx_t duckdb_list_vector_get_size(
  duckdb_vector vector
);
```

Parameters

- `vector`: The vector

Return Value The size of the child list

duckdb_list_vector_set_size

Sets the total size of the underlying child-vector of a list vector.

Syntax

```
            duckdb_state duckdb_list_vector_set_size(
  duckdb_vector vector,
  idx_t size
);
```

Parameters

- `vector`: The list vector.
- `size`: The size of the child list.

Return Value The duckdb state. Returns DuckDBError if the vector is nullptr.

`duckdb_list_vector_reserve`

Sets the total capacity of the underlying child-vector of a list.

After calling this method, you must call `duckdb_vector_get_validity` and `duckdb_vector_get_data` to obtain current data and validity pointers

Syntax

```
            duckdb_state duckdb_list_vector_reserve(
  duckdb_vector vector,
  idx_t required_capacity
);
```

Parameters

- `vector`: The list vector.
- `required_capacity`: the total capacity to reserve.

Return Value The duckdb state. Returns DuckDBError if the vector is nullptr.

`duckdb_struct_vector_get_child`

Retrieves the child vector of a struct vector.

The resulting vector is valid as long as the parent vector is valid.

Syntax

```
            duckdb_vector duckdb_struct_vector_get_child(
  duckdb_vector vector,
  idx_t index
);
```

Parameters

- `vector`: The vector
- `index`: The child index

Return Value The child vector

`duckdb_array_vector_get_child`

Retrieves the child vector of an array vector.

The resulting vector is valid as long as the parent vector is valid. The resulting vector has the size of the parent vector multiplied by the array size.

Syntax

```
            duckdb_vector duckdb_array_vector_get_child(
  duckdb_vector vector
);
```

Parameters

- `vector`: The vector

Return Value The child vector

`duckdb_slice_vector`

Slice a vector with a selection vector.

The max value in the selection vector must be less than the length of the vector

The resulting vector happens to be a dictionary vector.

Syntax

```
            void duckdb_slice_vector(
  duckdb_vector vector,
  duckdb_selection_vector selection,
  idx_t len
);
```

Parameters

- `vector`: The vector which is to become a dictionary
- `selection`: The selection vector
- `len`: The length of the selection vector

`duckdb_vector_reference_value`

Copies the value from `value` to `vector`.

Syntax

```
            void duckdb_vector_reference_value(
  duckdb_vector vector,
  duckdb_value value
);
```

`duckdb_vector_reference_vector`

References the `from` vector in the `to` vector, this makes take shared ownership of the values buffer

Syntax

```
            void duckdb_vector_reference_vector(
  duckdb_vector to_vector,
  duckdb_vector from_vector
);
```

`duckdb_validity_row_is_valid`

Returns whether or not a row is valid (i.e., not NULL) in the given validity mask.

Syntax

```
            bool duckdb_validity_row_is_valid(
  uint64_t *validity,
  idx_t row
);
```

Parameters

- `validity`: The validity mask, as obtained through `duckdb_vector_get_validity`
- `row`: The row index

Return Value true if the row is valid, false otherwise

`duckdb_validity_set_row_validity`

In a validity mask, sets a specific row to either valid or invalid.

Note that `duckdb_vector_ensure_validity_writable` should be called before calling `duckdb_vector_get_validity`, to ensure that there is a validity mask to write to.

Syntax

```
            void duckdb_validity_set_row_validity(
  uint64_t *validity,
  idx_t row,
  bool valid
);
```

Parameters

- `validity`: The validity mask, as obtained through `duckdb_vector_get_validity`.
- `row`: The row index
- `valid`: Whether or not to set the row to valid, or invalid

`duckdb_validity_set_row_invalid`

In a validity mask, sets a specific row to invalid.

Equivalent to `duckdb_validity_set_row_validity` with valid set to false.

Syntax

```
            void duckdb_validity_set_row_invalid(
  uint64_t *validity,
  idx_t row
);
```

Parameters

- `validity`: The validity mask
- `row`: The row index

`duckdb_validity_set_row_valid`

In a validity mask, sets a specific row to valid.

Equivalent to `duckdb_validity_set_row_validity` with valid set to true.

Syntax

```
            void duckdb_validity_set_row_valid(
  uint64_t *validity,
  idx_t row
);
```

Parameters

- `validity`: The validity mask
- `row`: The row index

Values

The value class represents a single value of any type.

API Reference Overview

```
            void duckdb_destroy_value(duckdb_value *value);
duckdb_value duckdb_create_varchar(const char *text);
duckdb_value duckdb_create_varchar_length(const char *text, idx_t length);
duckdb_value duckdb_create_bool(bool input);
duckdb_value duckdb_create_int8(int8_t input);
duckdb_value duckdb_create_uint8(uint8_t input);
duckdb_value duckdb_create_int16(int16_t input);
duckdb_value duckdb_create_uint16(uint16_t input);
duckdb_value duckdb_create_int32(int32_t input);
duckdb_value duckdb_create_uint32(uint32_t input);
duckdb_value duckdb_create_uint64(uint64_t input);
duckdb_value duckdb_create_int64(int64_t val);
duckdb_value duckdb_create_hugeint(duckdb_hugeint input);
duckdb_value duckdb_create_uhugeint(duckdb_uhugeint input);
duckdb_value duckdb_create_varint(duckdb_varint input);
duckdb_value duckdb_create_decimal(duckdb_decimal input);
duckdb_value duckdb_create_float(float input);
duckdb_value duckdb_create_double(double input);
duckdb_value duckdb_create_date(duckdb_date input);
duckdb_value duckdb_create_time(duckdb_time input);
duckdb_value duckdb_create_time_tz_value(duckdb_time_tz value);
duckdb_value duckdb_create_timestamp(duckdb_timestamp input);
duckdb_value duckdb_create_timestamp_tz(duckdb_timestamp input);
duckdb_value duckdb_create_timestamp_s(duckdb_timestamp_s input);
duckdb_value duckdb_create_timestamp_ms(duckdb_timestamp_ms input);
duckdb_value duckdb_create_timestamp_ns(duckdb_timestamp_ns input);
duckdb_value duckdb_create_interval(duckdb_interval input);
duckdb_value duckdb_create_blob(const uint8_t *data, idx_t length);
duckdb_value duckdb_create_bit(duckdb_bit input);
duckdb_value duckdb_create_uuid(duckdb_uhugeint input);
bool duckdb_get_bool(duckdb_value val);
```

```
int8_t duckdb_get_int8(duckdb_value val);
uint8_t duckdb_get_uint8(duckdb_value val);
int16_t duckdb_get_int16(duckdb_value val);
uint16_t duckdb_get_uint16(duckdb_value val);
int32_t duckdb_get_int32(duckdb_value val);
uint32_t duckdb_get_uint32(duckdb_value val);
int64_t duckdb_get_int64(duckdb_value val);
uint64_t duckdb_get_uint64(duckdb_value val);
duckdb_hugeint duckdb_get_hugeint(duckdb_value val);
duckdb_uhugeint duckdb_get_uhugeint(duckdb_value val);
duckdb_varint duckdb_get_varint(duckdb_value val);
duckdb_decimal duckdb_get_decimal(duckdb_value val);
float duckdb_get_float(duckdb_value val);
double duckdb_get_double(duckdb_value val);
duckdb_date duckdb_get_date(duckdb_value val);
duckdb_time duckdb_get_time(duckdb_value val);
duckdb_time_tz duckdb_get_time_tz(duckdb_value val);
duckdb_timestamp duckdb_get_timestamp(duckdb_value val);
duckdb_timestamp duckdb_get_timestamp_tz(duckdb_value val);
duckdb_timestamp_s duckdb_get_timestamp_s(duckdb_value val);
duckdb_timestamp_ms duckdb_get_timestamp_ms(duckdb_value val);
duckdb_timestamp_ns duckdb_get_timestamp_ns(duckdb_value val);
duckdb_interval duckdb_get_interval(duckdb_value val);
duckdb_logical_type duckdb_get_value_type(duckdb_value val);
duckdb_blob duckdb_get_blob(duckdb_value val);
duckdb_bit duckdb_get_bit(duckdb_value val);
duckdb_uhugeint duckdb_get_uuid(duckdb_value val);
char *duckdb_get_varchar(duckdb_value value);
duckdb_value duckdb_create_struct_value(duckdb_logical_type type, duckdb_value *values);
duckdb_value duckdb_create_list_value(duckdb_logical_type type, duckdb_value *values, idx_t value_
count);
duckdb_value duckdb_create_array_value(duckdb_logical_type type, duckdb_value *values, idx_t value_
count);
duckdb_value duckdb_create_map_value(duckdb_logical_type map_type, duckdb_value *keys, duckdb_value
*values, idx_t entry_count);
duckdb_value duckdb_create_union_value(duckdb_logical_type union_type, idx_t tag_index, duckdb_value
value);
idx_t duckdb_get_map_size(duckdb_value value);
duckdb_value duckdb_get_map_key(duckdb_value value, idx_t index);
duckdb_value duckdb_get_map_value(duckdb_value value, idx_t index);
bool duckdb_is_null_value(duckdb_value value);
duckdb_value duckdb_create_null_value();
idx_t duckdb_get_list_size(duckdb_value value);
duckdb_value duckdb_get_list_child(duckdb_value value, idx_t index);
duckdb_value duckdb_create_enum_value(duckdb_logical_type type, uint64_t value);
uint64_t duckdb_get_enum_value(duckdb_value value);
duckdb_value duckdb_get_struct_child(duckdb_value value, idx_t index);
char *duckdb_value_to_string(duckdb_value value);
```

duckdb_destroy_value

Destroys the value and de-allocates all memory allocated for that type.

Syntax

```
            void duckdb_destroy_value(
  duckdb_value *value
);
```

Parameters

- `value`: The value to destroy.

`duckdb_create_varchar`

Creates a value from a null-terminated string

Syntax

```
            duckdb_value duckdb_create_varchar(
  const char *text
);
```

Parameters

- `text`: The null-terminated string

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_varchar_length`

Creates a value from a string

Syntax

```
            duckdb_value duckdb_create_varchar_length(
  const char *text,
  idx_t length
);
```

Parameters

- `text`: The text
- `length`: The length of the text

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_bool`

Creates a value from a boolean

Syntax

```
            duckdb_value duckdb_create_bool(
  bool input
);
```

Parameters

- `input`: The boolean value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_int8`

Creates a value from an int8_t (a tinyint)

Syntax

```
            duckdb_value duckdb_create_int8(
  int8_t input
);
```

Parameters

- `input`: The tinyint value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_uint8`

Creates a value from a uint8_t (a utinyint)

Syntax

```
            duckdb_value duckdb_create_uint8(
  uint8_t input
);
```

Parameters

- `input`: The utinyint value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_int16`

Creates a value from an int16_t (a smallint)

Syntax

```
            duckdb_value duckdb_create_int16(
  int16_t input
);
```

Parameters

- `input`: The smallint value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

duckdb_create_uint16

Creates a value from a uint16_t (a usmallint)

Syntax

```
            duckdb_value duckdb_create_uint16(
  uint16_t input
);
```

Parameters

- `input`: The usmallint value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

duckdb_create_int32

Creates a value from an int32_t (an integer)

Syntax

```
            duckdb_value duckdb_create_int32(
  int32_t input
);
```

Parameters

- `input`: The integer value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

duckdb_create_uint32

Creates a value from a uint32_t (a uinteger)

Syntax

```
            duckdb_value duckdb_create_uint32(
  uint32_t input
);
```

Parameters

- `input`: The uinteger value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

duckdb_create_uint64

Creates a value from a uint64_t (a ubigint)

Syntax

```
            duckdb_value duckdb_create_uint64(
  uint64_t input
);
```

Parameters

- `input`: The ubigint value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_int64`

Creates a value from an int64

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

Syntax

```
            duckdb_value duckdb_create_int64(
  int64_t val
);
```

`duckdb_create_hugeint`

Creates a value from a hugeint

Syntax

```
            duckdb_value duckdb_create_hugeint(
  duckdb_hugeint input
);
```

Parameters

- `input`: The hugeint value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_uhugeint`

Creates a value from a uhugeint

Syntax

```
            duckdb_value duckdb_create_uhugeint(
  duckdb_uhugeint input
);
```

Parameters

- `input`: The uhugeint value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_varint`

Creates a VARINT value from a duckdb_varint

Syntax

```
duckdb_value duckdb_create_varint(
  duckdb_varint input
);
```

Parameters

- `input`: The duckdb_varint value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_decimal`

Creates a DECIMAL value from a duckdb_decimal

Syntax

```
duckdb_value duckdb_create_decimal(
  duckdb_decimal input
);
```

Parameters

- `input`: The duckdb_decimal value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_float`

Creates a value from a float

Syntax

```
duckdb_value duckdb_create_float(
  float input
);
```

Parameters

- `input`: The float value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_double`

Creates a value from a double

Syntax

```
            duckdb_value duckdb_create_double(
  double input
);
```

Parameters

- `input`: The double value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_date`

Creates a value from a date

Syntax

```
            duckdb_value duckdb_create_date(
  duckdb_date input
);
```

Parameters

- `input`: The date value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_time`

Creates a value from a time

Syntax

```
            duckdb_value duckdb_create_time(
  duckdb_time input
);
```

Parameters

- `input`: The time value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_time_tz_value`

Creates a value from a time_tz. Not to be confused with `duckdb_create_time_tz`, which creates a duckdb_time_tz_t.

Syntax

```
            duckdb_value duckdb_create_time_tz_value(
  duckdb_time_tz value
);
```

Parameters

- `value`: The time_tz value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_timestamp`

Creates a TIMESTAMP value from a duckdb_timestamp

Syntax

```
            duckdb_value duckdb_create_timestamp(
  duckdb_timestamp input
);
```

Parameters

- `input`: The duckdb_timestamp value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_timestamp_tz`

Creates a TIMESTAMP_TZ value from a duckdb_timestamp

Syntax

```
            duckdb_value duckdb_create_timestamp_tz(
  duckdb_timestamp input
);
```

Parameters

- `input`: The duckdb_timestamp value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_timestamp_s`

Creates a TIMESTAMP_S value from a duckdb_timestamp_s

Syntax

```
            duckdb_value duckdb_create_timestamp_s(
  duckdb_timestamp_s input
);
```

Parameters

- `input`: The duckdb_timestamp_s value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_timestamp_ms`

Creates a TIMESTAMP_MS value from a duckdb_timestamp_ms

Syntax

```
            duckdb_value duckdb_create_timestamp_ms(
  duckdb_timestamp_ms input
);
```

Parameters

- `input`: The duckdb_timestamp_ms value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_timestamp_ns`

Creates a TIMESTAMP_NS value from a duckdb_timestamp_ns

Syntax

```
            duckdb_value duckdb_create_timestamp_ns(
  duckdb_timestamp_ns input
);
```

Parameters

- `input`: The duckdb_timestamp_ns value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_interval`

Creates a value from an interval

Syntax

```
            duckdb_value duckdb_create_interval(
  duckdb_interval input
);
```

Parameters

- `input`: The interval value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_blob`

Creates a value from a blob

Syntax

```
duckdb_value duckdb_create_blob(
  const uint8_t *data,
  idx_t length
);
```

Parameters

- `data`: The blob data
- `length`: The length of the blob data

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_bit`

Creates a BIT value from a duckdb_bit

Syntax

```
duckdb_value duckdb_create_bit(
  duckdb_bit input
);
```

Parameters

- `input`: The duckdb_bit value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_uuid`

Creates a UUID value from a uhugeint

Syntax

```
duckdb_value duckdb_create_uuid(
  duckdb_uhugeint input
);
```

Parameters

- `input`: The duckdb_uhugeint containing the UUID

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

duckdb_get_bool

Returns the boolean value of the given value.

Syntax

```
            bool duckdb_get_bool(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a boolean

Return Value A boolean, or false if the value cannot be converted

duckdb_get_int8

Returns the int8_t value of the given value.

Syntax

```
            int8_t duckdb_get_int8(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a tinyint

Return Value A int8_t, or MinValue if the value cannot be converted

duckdb_get_uint8

Returns the uint8_t value of the given value.

Syntax

```
            uint8_t duckdb_get_uint8(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a utinyint

Return Value A uint8_t, or MinValue if the value cannot be converted

duckdb_get_int16

Returns the int16_t value of the given value.

Syntax

```
            int16_t duckdb_get_int16(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a smallint

Return Value A int16_t, or MinValue if the value cannot be converted

`duckdb_get_uint16`

Returns the uint16_t value of the given value.

Syntax

```
            uint16_t duckdb_get_uint16(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a usmallint

Return Value A uint16_t, or MinValue if the value cannot be converted

`duckdb_get_int32`

Returns the int32_t value of the given value.

Syntax

```
            int32_t duckdb_get_int32(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing an integer

Return Value A int32_t, or MinValue if the value cannot be converted

`duckdb_get_uint32`

Returns the uint32_t value of the given value.

Syntax

```
            uint32_t duckdb_get_uint32(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a uinteger

Return Value A uint32_t, or MinValue if the value cannot be converted

`duckdb_get_int64`

Returns the int64_t value of the given value.

Syntax

```
            int64_t duckdb_get_int64(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a bigint

Return Value A int64_t, or MinValue if the value cannot be converted

`duckdb_get_uint64`

Returns the uint64_t value of the given value.

Syntax

```
            uint64_t duckdb_get_uint64(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a ubigint

Return Value A uint64_t, or MinValue if the value cannot be converted

`duckdb_get_hugeint`

Returns the hugeint value of the given value.

Syntax

```
            duckdb_hugeint duckdb_get_hugeint(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a hugeint

Return Value A duckdb_hugeint, or MinValue if the value cannot be converted

`duckdb_get_uhugeint`

Returns the uhugeint value of the given value.

Syntax

```
            duckdb_uhugeint duckdb_get_uhugeint(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a uhugeint

Return Value A duckdb_uhugeint, or MinValue if the value cannot be converted

`duckdb_get_varint`

Returns the duckdb_varint value of the given value. The `data` field must be destroyed with `duckdb_free`.

Syntax

```
            duckdb_varint duckdb_get_varint(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a VARINT

Return Value A duckdb_varint. The `data` field must be destroyed with `duckdb_free`.

`duckdb_get_decimal`

Returns the duckdb_decimal value of the given value.

Syntax

```
            duckdb_decimal duckdb_get_decimal(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a DECIMAL

Return Value A duckdb_decimal, or MinValue if the value cannot be converted

`duckdb_get_float`

Returns the float value of the given value.

Syntax

```
            float duckdb_get_float(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a float

Return Value A float, or NAN if the value cannot be converted

`duckdb_get_double`

Returns the double value of the given value.

Syntax

```
            double duckdb_get_double(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a double

Return Value A double, or NAN if the value cannot be converted

`duckdb_get_date`

Returns the date value of the given value.

Syntax

```
            duckdb_date duckdb_get_date(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a date

Return Value A duckdb_date, or MinValue if the value cannot be converted

`duckdb_get_time`

Returns the time value of the given value.

Syntax

```
            duckdb_time duckdb_get_time(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a time

Return Value A duckdb_time, or MinValue if the value cannot be converted

`duckdb_get_time_tz`

Returns the time_tz value of the given value.

Syntax

```
            duckdb_time_tz duckdb_get_time_tz(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a time_tz

Return Value A duckdb_time_tz, or MinValue if the value cannot be converted

`duckdb_get_timestamp`

Returns the TIMESTAMP value of the given value.

Syntax

```
            duckdb_timestamp duckdb_get_timestamp(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a TIMESTAMP

Return Value A duckdb_timestamp, or MinValue if the value cannot be converted

`duckdb_get_timestamp_tz`

Returns the TIMESTAMP_TZ value of the given value.

Syntax

```
            duckdb_timestamp duckdb_get_timestamp_tz(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a TIMESTAMP_TZ

Return Value A duckdb_timestamp, or MinValue if the value cannot be converted

`duckdb_get_timestamp_s`

Returns the duckdb_timestamp_s value of the given value.

Syntax

```
            duckdb_timestamp_s duckdb_get_timestamp_s(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a TIMESTAMP_S

Return Value A duckdb_timestamp_s, or MinValue if the value cannot be converted

`duckdb_get_timestamp_ms`

Returns the duckdb_timestamp_ms value of the given value.

Syntax

```
            duckdb_timestamp_ms duckdb_get_timestamp_ms(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a TIMESTAMP_MS

Return Value A duckdb_timestamp_ms, or MinValue if the value cannot be converted

`duckdb_get_timestamp_ns`

Returns the duckdb_timestamp_ns value of the given value.

Syntax

```
            duckdb_timestamp_ns duckdb_get_timestamp_ns(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a TIMESTAMP_NS

Return Value A duckdb_timestamp_ns, or MinValue if the value cannot be converted

`duckdb_get_interval`

Returns the interval value of the given value.

Syntax

```
            duckdb_interval duckdb_get_interval(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a interval

Return Value A duckdb_interval, or MinValue if the value cannot be converted

`duckdb_get_value_type`

Returns the type of the given value. The type is valid as long as the value is not destroyed. The type itself must not be destroyed.

Syntax

```
            duckdb_logical_type duckdb_get_value_type(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value

Return Value A duckdb_logical_type.

`duckdb_get_blob`

Returns the blob value of the given value.

Syntax

```
            duckdb_blob duckdb_get_blob(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a blob

Return Value A duckdb_blob

`duckdb_get_bit`

Returns the duckdb_bit value of the given value. The `data` field must be destroyed with `duckdb_free`.

Syntax

```
            duckdb_bit duckdb_get_bit(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a BIT

Return Value A duckdb_bit

`duckdb_get_uuid`

Returns a duckdb_uhugeint representing the UUID value of the given value.

Syntax

```
            duckdb_uhugeint duckdb_get_uuid(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a UUID

Return Value A duckdb_uhugeint representing the UUID value

`duckdb_get_varchar`

Obtains a string representation of the given value. The result must be destroyed with `duckdb_free`.

Syntax

```
            char *duckdb_get_varchar(
  duckdb_value value
);
```

Parameters

- `value`: The value

Return Value The string value. This must be destroyed with `duckdb_free`.

`duckdb_create_struct_value`

Creates a struct value from a type and an array of values. Must be destroyed with `duckdb_destroy_value`.

Syntax

```
            duckdb_value duckdb_create_struct_value(
  duckdb_logical_type type,
  duckdb_value *values
);
```

Parameters

- `type`: The type of the struct
- `values`: The values for the struct fields

Return Value The struct value, or nullptr, if any child type is `DUCKDB_TYPE_ANY` or `DUCKDB_TYPE_INVALID`.

`duckdb_create_list_value`

Creates a list value from a child (element) type and an array of values of length `value_count`. Must be destroyed with `duckdb_destroy_value`.

Syntax

```
            duckdb_value duckdb_create_list_value(
  duckdb_logical_type type,
  duckdb_value *values,
  idx_t value_count
);
```

Parameters

- `type`: The type of the list
- `values`: The values for the list
- `value_count`: The number of values in the list

Return Value The list value, or nullptr, if the child type is `DUCKDB_TYPE_ANY` or `DUCKDB_TYPE_INVALID`.

`duckdb_create_array_value`

Creates an array value from a child (element) type and an array of values of length `value_count`. Must be destroyed with `duckdb_destroy_value`.

Syntax

```
            duckdb_value duckdb_create_array_value(
  duckdb_logical_type type,
  duckdb_value *values,
  idx_t value_count
);
```

Parameters

- `type`: The type of the array
- `values`: The values for the array
- `value_count`: The number of values in the array

Return Value The array value, or nullptr, if the child type is `DUCKDB_TYPE_ANY` or `DUCKDB_TYPE_INVALID`.

`duckdb_create_map_value`

Creates a map value from a map type and two arrays, one for the keys and one for the values, each of length `entry_count`. Must be destroyed with `duckdb_destroy_value`.

Syntax

```
            duckdb_value duckdb_create_map_value(
  duckdb_logical_type map_type,
  duckdb_value *keys,
  duckdb_value *values,
  idx_t entry_count
);
```

Parameters

- `map_type`: The map type
- `keys`: The keys of the map
- `values`: The values of the map
- `entry_count`: The number of entrys (key-value pairs) in the map

Return Value The map value, or nullptr, if the parameters are invalid.

`duckdb_create_union_value`

Creates a union value from a union type, a tag index, and a value. Must be destroyed with `duckdb_destroy_value`.

Syntax

```
            duckdb_value duckdb_create_union_value(
  duckdb_logical_type union_type,
  idx_t tag_index,
  duckdb_value value
);
```

Parameters

- `union_type`: The union type
- `tag_index`: The index of the tag of the union
- `value`: The value of the union

Return Value The union value, or nullptr, if the parameters are invalid.

`duckdb_get_map_size`

Returns the number of elements in a MAP value.

Syntax

```
            idx_t duckdb_get_map_size(
  duckdb_value value
);
```

Parameters

- `value`: The MAP value.

Return Value The number of elements in the map.

`duckdb_get_map_key`

Returns the MAP key at index as a duckdb_value.

Syntax

```
            duckdb_value duckdb_get_map_key(
  duckdb_value value,
  idx_t index
);
```

Parameters

- `value`: The MAP value.
- `index`: The index of the key.

Return Value The key as a duckdb_value.

`duckdb_get_map_value`

Returns the MAP value at index as a duckdb_value.

Syntax

```
            duckdb_value duckdb_get_map_value(
  duckdb_value value,
  idx_t index
);
```

Parameters

- `value`: The MAP value.
- `index`: The index of the value.

Return Value The value as a duckdb_value.

`duckdb_is_null_value`

Returns whether the value's type is SQLNULL or not.

Syntax

```
            bool duckdb_is_null_value(
  duckdb_value value
);
```

Parameters

- `value`: The value to check.

Return Value True, if the value's type is SQLNULL, otherwise false.

`duckdb_create_null_value`

Creates a value of type SQLNULL.

Return Value The duckdb_value representing SQLNULL. This must be destroyed with `duckdb_destroy_value`.

Syntax

```
            duckdb_value duckdb_create_null_value(

);
```

`duckdb_get_list_size`

Returns the number of elements in a LIST value.

Syntax

```
            idx_t duckdb_get_list_size(
  duckdb_value value
);
```

Parameters

- `value`: The LIST value.

Return Value The number of elements in the list.

`duckdb_get_list_child`

Returns the LIST child at index as a duckdb_value.

Syntax

```
            duckdb_value duckdb_get_list_child(
  duckdb_value value,
  idx_t index
);
```

Parameters

- `value`: The LIST value.
- `index`: The index of the child.

Return Value The child as a duckdb_value.

`duckdb_create_enum_value`

Creates an enum value from a type and a value. Must be destroyed with `duckdb_destroy_value`.

Syntax

```
            duckdb_value duckdb_create_enum_value(
  duckdb_logical_type type,
  uint64_t value
);
```

Parameters

- `type`: The type of the enum
- `value`: The value for the enum

Return Value The enum value, or nullptr.

`duckdb_get_enum_value`

Returns the enum value of the given value.

Syntax

```
            uint64_t duckdb_get_enum_value(
  duckdb_value value
);
```

Parameters

- `value`: A duckdb_value containing an enum

Return Value A uint64_t, or MinValue if the value cannot be converted

`duckdb_get_struct_child`

Returns the STRUCT child at index as a duckdb_value.

Syntax

```
            duckdb_value duckdb_get_struct_child(
  duckdb_value value,
  idx_t index
);
```

Parameters

- `value`: The STRUCT value.
- `index`: The index of the child.

Return Value The child as a duckdb_value.

`duckdb_value_to_string`

Returns the SQL string representation of the given value.

Syntax

```
            char *duckdb_value_to_string(
  duckdb_value value
);
```

Parameters

- `value`: A duckdb_value.

Return Value The SQL string representation as a null-terminated string. The result must be freed with `duckdb_free`.

Types

DuckDB is a strongly typed database system. As such, every column has a single type specified. This type is constant over the entire column. That is to say, a column that is labeled as an `INTEGER` column will only contain `INTEGER` values.

DuckDB also supports columns of composite types. For example, it is possible to define an array of integers (`INTEGER[]`). It is also possible to define types as arbitrary structs (`ROW(i INTEGER, j VARCHAR)`). For that reason, native DuckDB type objects are not mere enums, but a class that can potentially be nested.

Types in the C API are modeled using an enum (`duckdb_type`) and a complex class (`duckdb_logical_type`). For most primitive types, e.g., integers or varchars, the enum is sufficient. For more complex types, such as lists, structs or decimals, the logical type must be used.

```
typedef enum DUCKDB_TYPE {
  DUCKDB_TYPE_INVALID = 0,
  DUCKDB_TYPE_BOOLEAN = 1,
  DUCKDB_TYPE_TINYINT = 2,
  DUCKDB_TYPE_SMALLINT = 3,
  DUCKDB_TYPE_INTEGER = 4,
  DUCKDB_TYPE_BIGINT = 5,
  DUCKDB_TYPE_UTINYINT = 6,
  DUCKDB_TYPE_USMALLINT = 7,
  DUCKDB_TYPE_UINTEGER = 8,
  DUCKDB_TYPE_UBIGINT = 9,
  DUCKDB_TYPE_FLOAT = 10,
  DUCKDB_TYPE_DOUBLE = 11,
  DUCKDB_TYPE_TIMESTAMP = 12,
  DUCKDB_TYPE_DATE = 13,
  DUCKDB_TYPE_TIME = 14,
  DUCKDB_TYPE_INTERVAL = 15,
  DUCKDB_TYPE_HUGEINT = 16,
  DUCKDB_TYPE_UHUGEINT = 32,
```

```
  DUCKDB_TYPE_VARCHAR = 17,
  DUCKDB_TYPE_BLOB = 18,
  DUCKDB_TYPE_DECIMAL = 19,
  DUCKDB_TYPE_TIMESTAMP_S = 20,
  DUCKDB_TYPE_TIMESTAMP_MS = 21,
  DUCKDB_TYPE_TIMESTAMP_NS = 22,
  DUCKDB_TYPE_ENUM = 23,
  DUCKDB_TYPE_LIST = 24,
  DUCKDB_TYPE_STRUCT = 25,
  DUCKDB_TYPE_MAP = 26,
  DUCKDB_TYPE_ARRAY = 33,
  DUCKDB_TYPE_UUID = 27,
  DUCKDB_TYPE_UNION = 28,
  DUCKDB_TYPE_BIT = 29,
  DUCKDB_TYPE_TIME_TZ = 30,
  DUCKDB_TYPE_TIMESTAMP_TZ = 31,
} duckdb_type;
```

Functions

The enum type of a column in the result can be obtained using the `duckdb_column_type` function. The logical type of a column can be obtained using the `duckdb_column_logical_type` function.

`duckdb_value`

The `duckdb_value` functions will auto-cast values as required. For example, it is no problem to use `duckdb_value_double` on a column of type `duckdb_value_int32`. The value will be auto-cast and returned as a double. Note that in certain cases the cast may fail. For example, this can happen if we request a `duckdb_value_int8` and the value does not fit within an `int8` value. In this case, a default value will be returned (usually `0` or `nullptr`). The same default value will also be returned if the corresponding value is `NULL`.

The `duckdb_value_is_null` function can be used to check if a specific value is `NULL` or not.

The exception to the auto-cast rule is the `duckdb_value_varchar_internal` function. This function does not auto-cast and only works for VARCHAR columns. The reason this function exists is that the result does not need to be freed.

> `duckdb_value_varchar` and `duckdb_value_blob` require the result to be de-allocated using `duckdb_free`.

`duckdb_fetch_chunk`

The `duckdb_fetch_chunk` function can be used to read data chunks from a DuckDB result set, and is the most efficient way of reading data from a DuckDB result using the C API. It is also the only way of reading data of certain types from a DuckDB result. For example, the `duckdb_value` functions do not support structural reading of composite types (lists or structs) or more complex types like enums and decimals.

For more information about data chunks, see the documentation on data chunks.

API Reference Overview

```
           duckdb_data_chunk duckdb_result_get_chunk(duckdb_result result, idx_t chunk_index);
bool duckdb_result_is_streaming(duckdb_result result);
idx_t duckdb_result_chunk_count(duckdb_result result);
duckdb_result_type duckdb_result_return_type(duckdb_result result);
```

Date Time Timestamp Helpers

```
            duckdb_date_struct duckdb_from_date(duckdb_date date);
duckdb_date duckdb_to_date(duckdb_date_struct date);
bool duckdb_is_finite_date(duckdb_date date);
duckdb_time_struct duckdb_from_time(duckdb_time time);
duckdb_time_tz duckdb_create_time_tz(int64_t micros, int32_t offset);
duckdb_time_tz_struct duckdb_from_time_tz(duckdb_time_tz micros);
duckdb_time duckdb_to_time(duckdb_time_struct time);
duckdb_timestamp_struct duckdb_from_timestamp(duckdb_timestamp ts);
duckdb_timestamp duckdb_to_timestamp(duckdb_timestamp_struct ts);
bool duckdb_is_finite_timestamp(duckdb_timestamp ts);
bool duckdb_is_finite_timestamp_s(duckdb_timestamp_s ts);
bool duckdb_is_finite_timestamp_ms(duckdb_timestamp_ms ts);
bool duckdb_is_finite_timestamp_ns(duckdb_timestamp_ns ts);
```

Hugeint Helpers

```
            double duckdb_hugeint_to_double(duckdb_hugeint val);
duckdb_hugeint duckdb_double_to_hugeint(double val);
```

Decimal Helpers

```
            duckdb_decimal duckdb_double_to_decimal(double val, uint8_t width, uint8_t scale);
double duckdb_decimal_to_double(duckdb_decimal val);
```

Logical Type Interface

```
            duckdb_logical_type duckdb_create_logical_type(duckdb_type type);
char *duckdb_logical_type_get_alias(duckdb_logical_type type);
void duckdb_logical_type_set_alias(duckdb_logical_type type, const char *alias);
duckdb_logical_type duckdb_create_list_type(duckdb_logical_type type);
duckdb_logical_type duckdb_create_array_type(duckdb_logical_type type, idx_t array_size);
duckdb_logical_type duckdb_create_map_type(duckdb_logical_type key_type, duckdb_logical_type value_
type);
duckdb_logical_type duckdb_create_union_type(duckdb_logical_type *member_types, const char **member_
names, idx_t member_count);
duckdb_logical_type duckdb_create_struct_type(duckdb_logical_type *member_types, const char **member_
names, idx_t member_count);
duckdb_logical_type duckdb_create_enum_type(const char **member_names, idx_t member_count);
duckdb_logical_type duckdb_create_decimal_type(uint8_t width, uint8_t scale);
duckdb_type duckdb_get_type_id(duckdb_logical_type type);
uint8_t duckdb_decimal_width(duckdb_logical_type type);
uint8_t duckdb_decimal_scale(duckdb_logical_type type);
duckdb_type duckdb_decimal_internal_type(duckdb_logical_type type);
duckdb_type duckdb_enum_internal_type(duckdb_logical_type type);
uint32_t duckdb_enum_dictionary_size(duckdb_logical_type type);
char *duckdb_enum_dictionary_value(duckdb_logical_type type, idx_t index);
duckdb_logical_type duckdb_list_type_child_type(duckdb_logical_type type);
duckdb_logical_type duckdb_array_type_child_type(duckdb_logical_type type);
idx_t duckdb_array_type_array_size(duckdb_logical_type type);
duckdb_logical_type duckdb_map_type_key_type(duckdb_logical_type type);
duckdb_logical_type duckdb_map_type_value_type(duckdb_logical_type type);
idx_t duckdb_struct_type_child_count(duckdb_logical_type type);
char *duckdb_struct_type_child_name(duckdb_logical_type type, idx_t index);
duckdb_logical_type duckdb_struct_type_child_type(duckdb_logical_type type, idx_t index);
```

```
idx_t duckdb_union_type_member_count(duckdb_logical_type type);
char *duckdb_union_type_member_name(duckdb_logical_type type, idx_t index);
duckdb_logical_type duckdb_union_type_member_type(duckdb_logical_type type, idx_t index);
void duckdb_destroy_logical_type(duckdb_logical_type *type);
duckdb_state duckdb_register_logical_type(duckdb_connection con, duckdb_logical_type type, duckdb_
create_type_info info);
```

duckdb_result_get_chunk

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Fetches a data chunk from the duckdb_result. This function should be called repeatedly until the result is exhausted.

The result must be destroyed with `duckdb_destroy_data_chunk`.

This function supersedes all `duckdb_value` functions, as well as the `duckdb_column_data` and `duckdb_nullmask_data` functions. It results in significantly better performance, and should be preferred in newer code-bases.

If this function is used, none of the other result functions can be used and vice versa (i.e., this function cannot be mixed with the legacy result functions).

Use `duckdb_result_chunk_count` to figure out how many chunks there are in the result.

Syntax

```
            duckdb_data_chunk duckdb_result_get_chunk(
  duckdb_result result,
  idx_t chunk_index
);
```

Parameters

- `result`: The result object to fetch the data chunk from.
- `chunk_index`: The chunk index to fetch from.

Return Value The resulting data chunk. Returns `NULL` if the chunk index is out of bounds.

duckdb_result_is_streaming

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Checks if the type of the internal result is StreamQueryResult.

Syntax

```
            bool duckdb_result_is_streaming(
  duckdb_result result
);
```

Parameters

- `result`: The result object to check.

Return Value Whether or not the result object is of the type StreamQueryResult

`duckdb_result_chunk_count`

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Returns the number of data chunks present in the result.

Syntax

```
            idx_t duckdb_result_chunk_count(
  duckdb_result result
);
```

Parameters

- `result`: The result object

Return Value Number of data chunks present in the result.

`duckdb_result_return_type`

Returns the return_type of the given result, or DUCKDB_RETURN_TYPE_INVALID on error

Syntax

```
            duckdb_result_type duckdb_result_return_type(
  duckdb_result result
);
```

Parameters

- `result`: The result object

Return Value The return_type

`duckdb_from_date`

Decompose a `duckdb_date` object into year, month and date (stored as `duckdb_date_struct`).

Syntax

```
            duckdb_date_struct duckdb_from_date(
  duckdb_date date
);
```

Parameters

- `date`: The date object, as obtained from a DUCKDB_TYPE_DATE column.

Return Value The `duckdb_date_struct` with the decomposed elements.

`duckdb_to_date`

Re-compose a `duckdb_date` from year, month and date (`duckdb_date_struct`).

Syntax

```
            duckdb_date duckdb_to_date(
  duckdb_date_struct date
);
```

Parameters

- `date`: The year, month and date stored in a `duckdb_date_struct`.

Return Value The `duckdb_date` element.

`duckdb_is_finite_date`

Test a `duckdb_date` to see if it is a finite value.

Syntax

```
            bool duckdb_is_finite_date(
  duckdb_date date
);
```

Parameters

- `date`: The date object, as obtained from a `DUCKDB_TYPE_DATE` column.

Return Value True if the date is finite, false if it is ±infinity.

`duckdb_from_time`

Decompose a `duckdb_time` object into hour, minute, second and microsecond (stored as `duckdb_time_struct`).

Syntax

```
            duckdb_time_struct duckdb_from_time(
  duckdb_time time
);
```

Parameters

- `time`: The time object, as obtained from a `DUCKDB_TYPE_TIME` column.

Return Value The `duckdb_time_struct` with the decomposed elements.

`duckdb_create_time_tz`

Create a `duckdb_time_tz` object from micros and a timezone offset.

Syntax

```
            duckdb_time_tz duckdb_create_time_tz(
  int64_t micros,
  int32_t offset
);
```

Parameters

- `micros`: The microsecond component of the time.
- `offset`: The timezone offset component of the time.

Return Value The `duckdb_time_tz` element.

`duckdb_from_time_tz`

Decompose a TIME_TZ objects into micros and a timezone offset.

Use `duckdb_from_time` to further decompose the micros into hour, minute, second and microsecond.

Syntax

```
            duckdb_time_tz_struct duckdb_from_time_tz(
  duckdb_time_tz micros
);
```

Parameters

- `micros`: The time object, as obtained from a `DUCKDB_TYPE_TIME_TZ` column.

`duckdb_to_time`

Re-compose a `duckdb_time` from hour, minute, second and microsecond (`duckdb_time_struct`).

Syntax

```
            duckdb_time duckdb_to_time(
  duckdb_time_struct time
);
```

Parameters

- `time`: The hour, minute, second and microsecond in a `duckdb_time_struct`.

Return Value The `duckdb_time` element.

`duckdb_from_timestamp`

Decompose a `duckdb_timestamp` object into a `duckdb_timestamp_struct`.

Syntax

```
            duckdb_timestamp_struct duckdb_from_timestamp(
  duckdb_timestamp ts
);
```

Parameters

- `ts`: The ts object, as obtained from a `DUCKDB_TYPE_TIMESTAMP` column.

Return Value The `duckdb_timestamp_struct` with the decomposed elements.

`duckdb_to_timestamp`

Re-compose a `duckdb_timestamp` from a duckdb_timestamp_struct.

Syntax

```
            duckdb_timestamp duckdb_to_timestamp(
  duckdb_timestamp_struct ts
);
```

Parameters

- `ts`: The de-composed elements in a `duckdb_timestamp_struct`.

Return Value The `duckdb_timestamp` element.

`duckdb_is_finite_timestamp`

Test a `duckdb_timestamp` to see if it is a finite value.

Syntax

```
            bool duckdb_is_finite_timestamp(
  duckdb_timestamp ts
);
```

Parameters

- `ts`: The duckdb_timestamp object, as obtained from a `DUCKDB_TYPE_TIMESTAMP` column.

Return Value True if the timestamp is finite, false if it is ±infinity.

`duckdb_is_finite_timestamp_s`

Test a `duckdb_timestamp_s` to see if it is a finite value.

Syntax

```
            bool duckdb_is_finite_timestamp_s(
  duckdb_timestamp_s ts
);
```

Parameters

- `ts`: The duckdb_timestamp_s object, as obtained from a `DUCKDB_TYPE_TIMESTAMP_S` column.

Return Value True if the timestamp is finite, false if it is ±infinity.

`duckdb_is_finite_timestamp_ms`

Test a `duckdb_timestamp_ms` to see if it is a finite value.

Syntax

```
            bool duckdb_is_finite_timestamp_ms(
  duckdb_timestamp_ms ts
);
```

Parameters

- `ts`: The duckdb_timestamp_ms object, as obtained from a `DUCKDB_TYPE_TIMESTAMP_MS` column.

Return Value True if the timestamp is finite, false if it is ±infinity.

`duckdb_is_finite_timestamp_ns`

Test a `duckdb_timestamp_ns` to see if it is a finite value.

Syntax

```
            bool duckdb_is_finite_timestamp_ns(
  duckdb_timestamp_ns ts
);
```

Parameters

- `ts`: The duckdb_timestamp_ns object, as obtained from a `DUCKDB_TYPE_TIMESTAMP_NS` column.

Return Value True if the timestamp is finite, false if it is ±infinity.

`duckdb_hugeint_to_double`

Converts a duckdb_hugeint object (as obtained from a `DUCKDB_TYPE_HUGEINT` column) into a double.

Syntax

```
            double duckdb_hugeint_to_double(
  duckdb_hugeint val
);
```

Parameters

- `val`: The hugeint value.

Return Value The converted `double` element.

`duckdb_double_to_hugeint`

Converts a double value to a duckdb_hugeint object.

If the conversion fails because the double value is too big the result will be 0.

Syntax

```
            duckdb_hugeint duckdb_double_to_hugeint(
  double val
);
```

Parameters

- `val`: The double value.

Return Value The converted `duckdb_hugeint` element.

`duckdb_double_to_decimal`

Converts a double value to a duckdb_decimal object.

If the conversion fails because the double value is too big, or the width/scale are invalid the result will be 0.

Syntax

```
            duckdb_decimal duckdb_double_to_decimal(
  double val,
  uint8_t width,
  uint8_t scale
);
```

Parameters

- `val`: The double value.

Return Value The converted `duckdb_decimal` element.

`duckdb_decimal_to_double`

Converts a duckdb_decimal object (as obtained from a `DUCKDB_TYPE_DECIMAL` column) into a double.

Syntax

```
            double duckdb_decimal_to_double(
  duckdb_decimal val
);
```

Parameters

- `val`: The decimal value.

Return Value The converted `double` element.

`duckdb_create_logical_type`

Creates a `duckdb_logical_type` from a primitive type. The resulting logical type must be destroyed with `duckdb_destroy_logical_type`.

Returns an invalid logical type, if type is: `DUCKDB_TYPE_INVALID`, `DUCKDB_TYPE_DECIMAL`, `DUCKDB_TYPE_ENUM`, `DUCKDB_TYPE_LIST`, `DUCKDB_TYPE_STRUCT`, `DUCKDB_TYPE_MAP`, `DUCKDB_TYPE_ARRAY`, or `DUCKDB_TYPE_UNION`.

Syntax

```
            duckdb_logical_type duckdb_create_logical_type(
  duckdb_type type
);
```

Parameters

- `type`: The primitive type to create.

Return Value The logical type.

`duckdb_logical_type_get_alias`

Returns the alias of a duckdb_logical_type, if set, else `nullptr`. The result must be destroyed with `duckdb_free`.

Syntax

```
            char *duckdb_logical_type_get_alias(
  duckdb_logical_type type
);
```

Parameters

- `type`: The logical type

Return Value The alias or `nullptr`

`duckdb_logical_type_set_alias`

Sets the alias of a duckdb_logical_type.

Syntax

```
            void duckdb_logical_type_set_alias(
  duckdb_logical_type type,
  const char *alias
);
```

Parameters

- `type`: The logical type
- `alias`: The alias to set

`duckdb_create_list_type`

Creates a LIST type from its child type. The return type must be destroyed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_create_list_type(
  duckdb_logical_type type
);
```

Parameters

- `type`: The child type of the list

Return Value The logical type.

`duckdb_create_array_type`

Creates an ARRAY type from its child type. The return type must be destroyed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_create_array_type(
  duckdb_logical_type type,
  idx_t array_size
);
```

Parameters

- `type`: The child type of the array.
- `array_size`: The number of elements in the array.

Return Value The logical type.

`duckdb_create_map_type`

Creates a MAP type from its key type and value type. The return type must be destroyed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_create_map_type(
  duckdb_logical_type key_type,
  duckdb_logical_type value_type
);
```

Parameters

- `key_type`: The map's key type.
- `value_type`: The map's value type.

Return Value The logical type.

duckdb_create_union_type

Creates a UNION type from the passed arrays. The return type must be destroyed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_create_union_type(
  duckdb_logical_type *member_types,
  const char **member_names,
  idx_t member_count
);
```

Parameters

- `member_types`: The array of union member types.
- `member_names`: The union member names.
- `member_count`: The number of union members.

Return Value The logical type.

duckdb_create_struct_type

Creates a STRUCT type based on the member types and names. The resulting type must be destroyed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_create_struct_type(
  duckdb_logical_type *member_types,
  const char **member_names,
  idx_t member_count
);
```

Parameters

- `member_types`: The array of types of the struct members.
- `member_names`: The array of names of the struct members.
- `member_count`: The number of members of the struct.

Return Value The logical type.

duckdb_create_enum_type

Creates an ENUM type from the passed member name array. The resulting type should be destroyed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_create_enum_type(
  const char **member_names,
  idx_t member_count
);
```

Parameters

- `member_names`: The array of names that the enum should consist of.
- `member_count`: The number of elements that were specified in the array.

Return Value The logical type.

`duckdb_create_decimal_type`

Creates a DECIMAL type with the specified width and scale. The resulting type should be destroyed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_create_decimal_type(
  uint8_t width,
  uint8_t scale
);
```

Parameters

- `width`: The width of the decimal type
- `scale`: The scale of the decimal type

Return Value The logical type.

`duckdb_get_type_id`

Retrieves the enum `duckdb_type` of a `duckdb_logical_type`.

Syntax

```
            duckdb_type duckdb_get_type_id(
  duckdb_logical_type type
);
```

Parameters

- `type`: The logical type.

Return Value The `duckdb_type` id.

`duckdb_decimal_width`

Retrieves the width of a decimal type.

Syntax

```
            uint8_t duckdb_decimal_width(
  duckdb_logical_type type
);
```

Parameters

- `type`: The logical type object

Return Value The width of the decimal type

`duckdb_decimal_scale`

Retrieves the scale of a decimal type.

Syntax

```
            uint8_t duckdb_decimal_scale(
  duckdb_logical_type type
);
```

Parameters

- `type`: The logical type object

Return Value The scale of the decimal type

`duckdb_decimal_internal_type`

Retrieves the internal storage type of a decimal type.

Syntax

```
            duckdb_type duckdb_decimal_internal_type(
  duckdb_logical_type type
);
```

Parameters

- `type`: The logical type object

Return Value The internal type of the decimal type

`duckdb_enum_internal_type`

Retrieves the internal storage type of an enum type.

Syntax

```
            duckdb_type duckdb_enum_internal_type(
  duckdb_logical_type type
);
```

Parameters

- `type`: The logical type object

Return Value The internal type of the enum type

`duckdb_enum_dictionary_size`

Retrieves the dictionary size of the enum type.

Syntax

```
            uint32_t duckdb_enum_dictionary_size(
  duckdb_logical_type type
);
```

Parameters

- `type`: The logical type object

Return Value The dictionary size of the enum type

`duckdb_enum_dictionary_value`

Retrieves the dictionary value at the specified position from the enum.

The result must be freed with `duckdb_free`.

Syntax

```
            char *duckdb_enum_dictionary_value(
  duckdb_logical_type type,
  idx_t index
);
```

Parameters

- `type`: The logical type object
- `index`: The index in the dictionary

Return Value The string value of the enum type. Must be freed with `duckdb_free`.

`duckdb_list_type_child_type`

Retrieves the child type of the given LIST type. Also accepts MAP types. The result must be freed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_list_type_child_type(
  duckdb_logical_type type
);
```

Parameters

- `type`: The logical type, either LIST or MAP.

Return Value The child type of the LIST or MAP type.

`duckdb_array_type_child_type`

Retrieves the child type of the given ARRAY type.

The result must be freed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_array_type_child_type(
  duckdb_logical_type type
);
```

Parameters

- `type`: The logical type. Must be ARRAY.

Return Value The child type of the ARRAY type.

`duckdb_array_type_array_size`

Retrieves the array size of the given array type.

Syntax

```
            idx_t duckdb_array_type_array_size(
  duckdb_logical_type type
);
```

Parameters

- `type`: The logical type object

Return Value The fixed number of elements the values of this array type can store.

`duckdb_map_type_key_type`

Retrieves the key type of the given map type.

The result must be freed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_map_type_key_type(
  duckdb_logical_type type
);
```

Parameters

- `type`: The logical type object

Return Value The key type of the map type. Must be destroyed with `duckdb_destroy_logical_type`.

`duckdb_map_type_value_type`

Retrieves the value type of the given map type.

The result must be freed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_map_type_value_type(
  duckdb_logical_type type
);
```

Parameters

- `type`: The logical type object

Return Value The value type of the map type. Must be destroyed with `duckdb_destroy_logical_type`.

`duckdb_struct_type_child_count`

Returns the number of children of a struct type.

Syntax

```
            idx_t duckdb_struct_type_child_count(
  duckdb_logical_type type
);
```

Parameters

- `type`: The logical type object

Return Value The number of children of a struct type.

`duckdb_struct_type_child_name`

Retrieves the name of the struct child.

The result must be freed with `duckdb_free`.

Syntax

```
            char *duckdb_struct_type_child_name(
  duckdb_logical_type type,
  idx_t index
);
```

Parameters

- `type`: The logical type object
- `index`: The child index

Return Value The name of the struct type. Must be freed with `duckdb_free`.

`duckdb_struct_type_child_type`

Retrieves the child type of the given struct type at the specified index.

The result must be freed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_struct_type_child_type(
  duckdb_logical_type type,
  idx_t index
);
```

Parameters

- `type`: The logical type object
- `index`: The child index

Return Value The child type of the struct type. Must be destroyed with `duckdb_destroy_logical_type`.

`duckdb_union_type_member_count`

Returns the number of members that the union type has.

Syntax

```
            idx_t duckdb_union_type_member_count(
  duckdb_logical_type type
);
```

Parameters

- `type`: The logical type (union) object

Return Value The number of members of a union type.

`duckdb_union_type_member_name`

Retrieves the name of the union member.

The result must be freed with `duckdb_free`.

Syntax

```
            char *duckdb_union_type_member_name(
  duckdb_logical_type type,
  idx_t index
);
```

Parameters

- `type`: The logical type object
- `index`: The child index

Return Value The name of the union member. Must be freed with `duckdb_free`.

`duckdb_union_type_member_type`

Retrieves the child type of the given union member at the specified index.

The result must be freed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_union_type_member_type(
  duckdb_logical_type type,
  idx_t index
);
```

Parameters

- `type`: The logical type object
- `index`: The child index

Return Value The child type of the union member. Must be destroyed with `duckdb_destroy_logical_type`.

`duckdb_destroy_logical_type`

Destroys the logical type and de-allocates all memory allocated for that type.

Syntax

```
            void duckdb_destroy_logical_type(
  duckdb_logical_type *type
);
```

Parameters

- `type`: The logical type to destroy.

`duckdb_register_logical_type`

Registers a custom type within the given connection. The type must have an alias

Syntax

```
            duckdb_state duckdb_register_logical_type(
  duckdb_connection con,
  duckdb_logical_type type,
  duckdb_create_type_info info
);
```

Parameters

- `con`: The connection to use
- `type`: The custom type to register

Return Value Whether or not the registration was successful.

Prepared Statements

A prepared statement is a parameterized query. The query is prepared with question marks (`?`) or dollar symbols (`$1`) indicating the parameters of the query. Values can then be bound to these parameters, after which the prepared statement can be executed using those parameters. A single query can be prepared once and executed many times.

Prepared statements are useful to:

- Easily supply parameters to functions while avoiding string concatenation/SQL injection attacks.
- Speeding up queries that will be executed many times with different parameters.

DuckDB supports prepared statements in the C API with the `duckdb_prepare` method. The `duckdb_bind` family of functions is used to supply values for subsequent execution of the prepared statement using `duckdb_execute_prepared`. After we are done with the prepared statement it can be cleaned up using the `duckdb_destroy_prepare` method.

Example

```
duckdb_prepared_statement stmt;
duckdb_result result;
if (duckdb_prepare(con, "INSERT INTO integers VALUES ($1, $2)", &stmt) == DuckDBError) {
    // handle error
}

duckdb_bind_int32(stmt, 1, 42); // the parameter index starts counting at 1!
duckdb_bind_int32(stmt, 2, 43);
// NULL as second parameter means no result set is requested
duckdb_execute_prepared(stmt, NULL);
duckdb_destroy_prepare(&stmt);

// we can also query result sets using prepared statements
if (duckdb_prepare(con, "SELECT * FROM integers WHERE i = ?", &stmt) == DuckDBError) {
    // handle error
}
duckdb_bind_int32(stmt, 1, 42);
duckdb_execute_prepared(stmt, &result);

// do something with result

// clean up
duckdb_destroy_result(&result);
duckdb_destroy_prepare(&stmt);
```

After calling `duckdb_prepare`, the prepared statement parameters can be inspected using `duckdb_nparams` and `duckdb_param_type`. In case the prepare fails, the error can be obtained through `duckdb_prepare_error`.

It is not required that the `duckdb_bind` family of functions matches the prepared statement parameter type exactly. The values will be auto-cast to the required value as required. For example, calling `duckdb_bind_int8` on a parameter type of `DUCKDB_TYPE_INTEGER` will work as expected.

> **Warning.** Do **not** use prepared statements to insert large amounts of data into DuckDB. Instead it is recommended to use the Appender.

API Reference Overview

```
duckdb_state duckdb_prepare(duckdb_connection connection, const char *query, duckdb_
prepared_statement *out_prepared_statement);
void duckdb_destroy_prepare(duckdb_prepared_statement *prepared_statement);
const char *duckdb_prepare_error(duckdb_prepared_statement prepared_statement);
idx_t duckdb_nparams(duckdb_prepared_statement prepared_statement);
const char *duckdb_parameter_name(duckdb_prepared_statement prepared_statement, idx_t index);
duckdb_type duckdb_param_type(duckdb_prepared_statement prepared_statement, idx_t param_idx);
duckdb_logical_type duckdb_param_logical_type(duckdb_prepared_statement prepared_statement, idx_t param_
idx);
duckdb_state duckdb_clear_bindings(duckdb_prepared_statement prepared_statement);
duckdb_statement_type duckdb_prepared_statement_type(duckdb_prepared_statement statement);
```

duckdb_prepare

Create a prepared statement object from a query.

Note that after calling `duckdb_prepare`, the prepared statement should always be destroyed using `duckdb_destroy_prepare`, even if the prepare fails.

If the prepare fails, `duckdb_prepare_error` can be called to obtain the reason why the prepare failed.

Syntax

```
duckdb_state duckdb_prepare(
  duckdb_connection connection,
  const char *query,
  duckdb_prepared_statement *out_prepared_statement
);
```

Parameters

- `connection`: The connection object
- `query`: The SQL query to prepare
- `out_prepared_statement`: The resulting prepared statement object

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

duckdb_destroy_prepare

Closes the prepared statement and de-allocates all memory allocated for the statement.

Syntax

```
void duckdb_destroy_prepare(
  duckdb_prepared_statement *prepared_statement
);
```

Parameters

- `prepared_statement`: The prepared statement to destroy.

`duckdb_prepare_error`

Returns the error message associated with the given prepared statement. If the prepared statement has no error message, this returns `nullptr` instead.

The error message should not be freed. It will be de-allocated when `duckdb_destroy_prepare` is called.

Syntax

```
            const char *duckdb_prepare_error(
  duckdb_prepared_statement prepared_statement
);
```

Parameters

- `prepared_statement`: The prepared statement to obtain the error from.

Return Value The error message, or `nullptr` if there is none.

`duckdb_nparams`

Returns the number of parameters that can be provided to the given prepared statement.

Returns 0 if the query was not successfully prepared.

Syntax

```
            idx_t duckdb_nparams(
  duckdb_prepared_statement prepared_statement
);
```

Parameters

- `prepared_statement`: The prepared statement to obtain the number of parameters for.

`duckdb_parameter_name`

Returns the name used to identify the parameter The returned string should be freed using `duckdb_free`.

Returns NULL if the index is out of range for the provided prepared statement.

Syntax

```
            const char *duckdb_parameter_name(
  duckdb_prepared_statement prepared_statement,
  idx_t index
);
```

Parameters

- `prepared_statement`: The prepared statement for which to get the parameter name from.

`duckdb_param_type`

Returns the parameter type for the parameter at the given index.

Returns `DUCKDB_TYPE_INVALID` if the parameter index is out of range or the statement was not successfully prepared.

Syntax

```
           duckdb_type duckdb_param_type(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx
);
```

Parameters

- `prepared_statement`: The prepared statement.
- `param_idx`: The parameter index.

Return Value The parameter type

`duckdb_param_logical_type`

Returns the logical type for the parameter at the given index.

Returns `nullptr` if the parameter index is out of range or the statement was not successfully prepared.

The return type of this call should be destroyed with `duckdb_destroy_logical_type`.

Syntax

```
           duckdb_logical_type duckdb_param_logical_type(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx
);
```

Parameters

- `prepared_statement`: The prepared statement.
- `param_idx`: The parameter index.

Return Value The logical type of the parameter

`duckdb_clear_bindings`

Clear the params bind to the prepared statement.

Syntax

```
           duckdb_state duckdb_clear_bindings(
  duckdb_prepared_statement prepared_statement
);
```

duckdb_prepared_statement_type

Returns the statement type of the statement to be executed

Syntax

```
            duckdb_statement_type duckdb_prepared_statement_type(
  duckdb_prepared_statement statement
);
```

Parameters

- `statement`: The prepared statement.

Return Value duckdb_statement_type value or DUCKDB_STATEMENT_TYPE_INVALID

Appender

Appenders are the most efficient way of loading data into DuckDB from within the C interface, and are recommended for fast data loading. The appender is much faster than using prepared statements or individual `INSERT INTO` statements.

Appends are made in row-wise format. For every column, a `duckdb_append_[type]` call should be made, after which the row should be finished by calling `duckdb_appender_end_row`. After all rows have been appended, `duckdb_appender_destroy` should be used to finalize the appender and clean up the resulting memory.

Note that `duckdb_appender_destroy` should always be called on the resulting appender, even if the function returns `DuckDBError`.

Example

```
duckdb_query(con, "CREATE TABLE people (id INTEGER, name VARCHAR)", NULL);

duckdb_appender appender;
if (duckdb_appender_create(con, NULL, "people", &appender) == DuckDBError) {
  // handle error
}
// append the first row (1, Mark)
duckdb_append_int32(appender, 1);
duckdb_append_varchar(appender, "Mark");
duckdb_appender_end_row(appender);

// append the second row (2, Hannes)
duckdb_append_int32(appender, 2);
duckdb_append_varchar(appender, "Hannes");
duckdb_appender_end_row(appender);

// finish appending and flush all the rows to the table
duckdb_appender_destroy(&appender);
```

API Reference Overview

```
            duckdb_state duckdb_appender_create(duckdb_connection connection, const char *schema, const
char *table, duckdb_appender *out_appender);
```

```
duckdb_state duckdb_appender_create_ext(duckdb_connection connection, const char *catalog, const char
*schema, const char *table, duckdb_appender *out_appender);
idx_t duckdb_appender_column_count(duckdb_appender appender);
duckdb_logical_type duckdb_appender_column_type(duckdb_appender appender, idx_t col_idx);
const char *duckdb_appender_error(duckdb_appender appender);
duckdb_state duckdb_appender_flush(duckdb_appender appender);
duckdb_state duckdb_appender_close(duckdb_appender appender);
duckdb_state duckdb_appender_destroy(duckdb_appender *appender);
duckdb_state duckdb_appender_add_column(duckdb_appender appender, const char *name);
duckdb_state duckdb_appender_clear_columns(duckdb_appender appender);
duckdb_state duckdb_appender_begin_row(duckdb_appender appender);
duckdb_state duckdb_appender_end_row(duckdb_appender appender);
duckdb_state duckdb_append_default(duckdb_appender appender);
duckdb_state duckdb_append_default_to_chunk(duckdb_appender appender, duckdb_data_chunk chunk, idx_t
col, idx_t row);
duckdb_state duckdb_append_bool(duckdb_appender appender, bool value);
duckdb_state duckdb_append_int8(duckdb_appender appender, int8_t value);
duckdb_state duckdb_append_int16(duckdb_appender appender, int16_t value);
duckdb_state duckdb_append_int32(duckdb_appender appender, int32_t value);
duckdb_state duckdb_append_int64(duckdb_appender appender, int64_t value);
duckdb_state duckdb_append_hugeint(duckdb_appender appender, duckdb_hugeint value);
duckdb_state duckdb_append_uint8(duckdb_appender appender, uint8_t value);
duckdb_state duckdb_append_uint16(duckdb_appender appender, uint16_t value);
duckdb_state duckdb_append_uint32(duckdb_appender appender, uint32_t value);
duckdb_state duckdb_append_uint64(duckdb_appender appender, uint64_t value);
duckdb_state duckdb_append_uhugeint(duckdb_appender appender, duckdb_uhugeint value);
duckdb_state duckdb_append_float(duckdb_appender appender, float value);
duckdb_state duckdb_append_double(duckdb_appender appender, double value);
duckdb_state duckdb_append_date(duckdb_appender appender, duckdb_date value);
duckdb_state duckdb_append_time(duckdb_appender appender, duckdb_time value);
duckdb_state duckdb_append_timestamp(duckdb_appender appender, duckdb_timestamp value);
duckdb_state duckdb_append_interval(duckdb_appender appender, duckdb_interval value);
duckdb_state duckdb_append_varchar(duckdb_appender appender, const char *val);
duckdb_state duckdb_append_varchar_length(duckdb_appender appender, const char *val, idx_t length);
duckdb_state duckdb_append_blob(duckdb_appender appender, const void *data, idx_t length);
duckdb_state duckdb_append_null(duckdb_appender appender);
duckdb_state duckdb_append_value(duckdb_appender appender, duckdb_value value);
duckdb_state duckdb_append_data_chunk(duckdb_appender appender, duckdb_data_chunk chunk);
```

`duckdb_appender_create`

Creates an appender object.

Note that the object must be destroyed with `duckdb_appender_destroy`.

Syntax

```
           duckdb_state duckdb_appender_create(
  duckdb_connection connection,
  const char *schema,
  const char *table,
  duckdb_appender *out_appender
);
```

Parameters

- `connection`: The connection context to create the appender in.
- `schema`: The schema of the table to append to, or `nullptr` for the default schema.

- `table`: The table name to append to.
- `out_appender`: The resulting appender object.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_appender_create_ext`

Creates an appender object.

Note that the object must be destroyed with `duckdb_appender_destroy`.

Syntax

```
            duckdb_state duckdb_appender_create_ext(
  duckdb_connection connection,
  const char *catalog,
  const char *schema,
  const char *table,
  duckdb_appender *out_appender
);
```

Parameters

- `connection`: The connection context to create the appender in.
- `catalog`: The catalog of the table to append to, or `nullptr` for the default catalog.
- `schema`: The schema of the table to append to, or `nullptr` for the default schema.
- `table`: The table name to append to.
- `out_appender`: The resulting appender object.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_appender_column_count`

Returns the number of columns that belong to the appender. If there is no active column list, then this equals the table's physical columns.

Syntax

```
            idx_t duckdb_appender_column_count(
  duckdb_appender appender
);
```

Parameters

- `appender`: The appender to get the column count from.

Return Value The number of columns in the data chunks.

`duckdb_appender_column_type`

Returns the type of the column at the specified index. This is either a type in the active column list, or the same type as a column in the receiving table.

Note: The resulting type must be destroyed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_appender_column_type(
  duckdb_appender appender,
  idx_t col_idx
);
```

Parameters

- `appender`: The appender to get the column type from.
- `col_idx`: The index of the column to get the type of.

Return Value The `duckdb_logical_type` of the column.

`duckdb_appender_error`

Returns the error message associated with the given appender. If the appender has no error message, this returns `nullptr` instead.

The error message should not be freed. It will be de-allocated when `duckdb_appender_destroy` is called.

Syntax

```
            const char *duckdb_appender_error(
  duckdb_appender appender
);
```

Parameters

- `appender`: The appender to get the error from.

Return Value The error message, or `nullptr` if there is none.

`duckdb_appender_flush`

Flush the appender to the table, forcing the cache of the appender to be cleared. If flushing the data triggers a constraint violation or any other error, then all data is invalidated, and this function returns DuckDBError. It is not possible to append more values. Call duckdb_appender_error to obtain the error message followed by duckdb_appender_destroy to destroy the invalidated appender.

Syntax

```
            duckdb_state duckdb_appender_flush(
  duckdb_appender appender
);
```

Parameters

- `appender`: The appender to flush.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

duckdb_appender_close

Closes the appender by flushing all intermediate states and closing it for further appends. If flushing the data triggers a constraint violation or any other error, then all data is invalidated, and this function returns DuckDBError. Call duckdb_appender_error to obtain the error message followed by duckdb_appender_destroy to destroy the invalidated appender.

Syntax

```
            duckdb_state duckdb_appender_close(
  duckdb_appender appender
);
```

Parameters

- `appender`: The appender to flush and close.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

duckdb_appender_destroy

Closes the appender by flushing all intermediate states to the table and destroying it. By destroying it, this function de-allocates all memory associated with the appender. If flushing the data triggers a constraint violation, then all data is invalidated, and this function returns DuckDBError. Due to the destruction of the appender, it is no longer possible to obtain the specific error message with duckdb_appender_error. Therefore, call duckdb_appender_close before destroying the appender, if you need insights into the specific error.

Syntax

```
            duckdb_state duckdb_appender_destroy(
  duckdb_appender *appender
);
```

Parameters

- `appender`: The appender to flush, close and destroy.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

duckdb_appender_add_column

Appends a column to the active column list of the appender. Immediately flushes all previous data.

The active column list specifies all columns that are expected when flushing the data. Any non-active columns are filled with their default values, or NULL.

Syntax

```
            duckdb_state duckdb_appender_add_column(
  duckdb_appender appender,
  const char *name
);
```

Parameters

- `appender`: The appender to add the column to.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_appender_clear_columns`

Removes all columns from the active column list of the appender, resetting the appender to treat all columns as active. Immediately flushes all previous data.

Syntax

```
duckdb_state duckdb_appender_clear_columns(
  duckdb_appender appender
);
```

Parameters

- `appender`: The appender to clear the columns from.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_appender_begin_row`

A nop function, provided for backwards compatibility reasons. Does nothing. Only `duckdb_appender_end_row` is required.

Syntax

```
duckdb_state duckdb_appender_begin_row(
  duckdb_appender appender
);
```

`duckdb_appender_end_row`

Finish the current row of appends. After end_row is called, the next row can be appended.

Syntax

```
duckdb_state duckdb_appender_end_row(
  duckdb_appender appender
);
```

Parameters

- `appender`: The appender.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_append_default`

Append a DEFAULT value (NULL if DEFAULT not available for column) to the appender.

Syntax

```
            duckdb_state duckdb_append_default(
  duckdb_appender appender
);
```

`duckdb_append_default_to_chunk`

Append a DEFAULT value, at the specified row and column, (NULL if DEFAULT not available for column) to the chunk created from the specified appender. The default value of the column must be a constant value. Non-deterministic expressions like nextval('seq') or random() are not supported.

Syntax

```
            duckdb_state duckdb_append_default_to_chunk(
  duckdb_appender appender,
  duckdb_data_chunk chunk,
  idx_t col,
  idx_t row
);
```

Parameters

- `appender`: The appender to get the default value from.
- `chunk`: The data chunk to append the default value to.
- `col`: The chunk column index to append the default value to.
- `row`: The chunk row index to append the default value to.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_append_bool`

Append a bool value to the appender.

Syntax

```
            duckdb_state duckdb_append_bool(
  duckdb_appender appender,
  bool value
);
```

`duckdb_append_int8`

Append an int8_t value to the appender.

Syntax

```
            duckdb_state duckdb_append_int8(
  duckdb_appender appender,
  int8_t value
);
```

duckdb_append_int16

Append an int16_t value to the appender.

Syntax

```
            duckdb_state duckdb_append_int16(
  duckdb_appender appender,
  int16_t value
);
```

duckdb_append_int32

Append an int32_t value to the appender.

Syntax

```
            duckdb_state duckdb_append_int32(
  duckdb_appender appender,
  int32_t value
);
```

duckdb_append_int64

Append an int64_t value to the appender.

Syntax

```
            duckdb_state duckdb_append_int64(
  duckdb_appender appender,
  int64_t value
);
```

duckdb_append_hugeint

Append a duckdb_hugeint value to the appender.

Syntax

```
            duckdb_state duckdb_append_hugeint(
  duckdb_appender appender,
  duckdb_hugeint value
);
```

duckdb_append_uint8

Append a uint8_t value to the appender.

Syntax

```
            duckdb_state duckdb_append_uint8(
  duckdb_appender appender,
  uint8_t value
);
```

duckdb_append_uint16

Append a uint16_t value to the appender.

Syntax

```
            duckdb_state duckdb_append_uint16(
  duckdb_appender appender,
  uint16_t value
);
```

duckdb_append_uint32

Append a uint32_t value to the appender.

Syntax

```
            duckdb_state duckdb_append_uint32(
  duckdb_appender appender,
  uint32_t value
);
```

duckdb_append_uint64

Append a uint64_t value to the appender.

Syntax

```
            duckdb_state duckdb_append_uint64(
  duckdb_appender appender,
  uint64_t value
);
```

duckdb_append_uhugeint

Append a duckdb_uhugeint value to the appender.

Syntax

```
            duckdb_state duckdb_append_uhugeint(
  duckdb_appender appender,
  duckdb_uhugeint value
);
```

duckdb_append_float

Append a float value to the appender.

Syntax

```
            duckdb_state duckdb_append_float(
  duckdb_appender appender,
  float value
);
```

duckdb_append_double

Append a double value to the appender.

Syntax

```
            duckdb_state duckdb_append_double(
  duckdb_appender appender,
  double value
);
```

duckdb_append_date

Append a duckdb_date value to the appender.

Syntax

```
            duckdb_state duckdb_append_date(
  duckdb_appender appender,
  duckdb_date value
);
```

duckdb_append_time

Append a duckdb_time value to the appender.

Syntax

```
            duckdb_state duckdb_append_time(
  duckdb_appender appender,
  duckdb_time value
);
```

duckdb_append_timestamp

Append a duckdb_timestamp value to the appender.

Syntax

```
            duckdb_state duckdb_append_timestamp(
  duckdb_appender appender,
  duckdb_timestamp value
);
```

duckdb_append_interval

Append a duckdb_interval value to the appender.

Syntax

```
            duckdb_state duckdb_append_interval(
  duckdb_appender appender,
  duckdb_interval value
);
```

duckdb_append_varchar

Append a varchar value to the appender.

Syntax

```
            duckdb_state duckdb_append_varchar(
  duckdb_appender appender,
  const char *val
);
```

duckdb_append_varchar_length

Append a varchar value to the appender.

Syntax

```
            duckdb_state duckdb_append_varchar_length(
  duckdb_appender appender,
  const char *val,
  idx_t length
);
```

duckdb_append_blob

Append a blob value to the appender.

Syntax

```
            duckdb_state duckdb_append_blob(
  duckdb_appender appender,
  const void *data,
  idx_t length
);
```

duckdb_append_null

Append a NULL value to the appender (of any type).

Syntax

```
            duckdb_state duckdb_append_null(
  duckdb_appender appender
);
```

duckdb_append_value

Append a duckdb_value to the appender.

Syntax

```
            duckdb_state duckdb_append_value(
  duckdb_appender appender,
  duckdb_value value
);
```

duckdb_append_data_chunk

Appends a pre-filled data chunk to the specified appender. Attempts casting, if the data chunk types do not match the active appender types.

Syntax

```
            duckdb_state duckdb_append_data_chunk(
  duckdb_appender appender,
  duckdb_data_chunk chunk
);
```

Parameters

- appender: The appender to append to.
- chunk: The data chunk to append.

Return Value DuckDBSuccess on success or DuckDBError on failure.

Table Functions

The table function API can be used to define a table function that can then be called from within DuckDB in the FROM clause of a query.

API Reference Overview

```
            duckdb_table_function duckdb_create_table_function();
void duckdb_destroy_table_function(duckdb_table_function *table_function);
void duckdb_table_function_set_name(duckdb_table_function table_function, const char *name);
void duckdb_table_function_add_parameter(duckdb_table_function table_function, duckdb_logical_type
type);
void duckdb_table_function_add_named_parameter(duckdb_table_function table_function, const char *name,
duckdb_logical_type type);
void duckdb_table_function_set_extra_info(duckdb_table_function table_function, void *extra_info,
duckdb_delete_callback_t destroy);
void duckdb_table_function_set_bind(duckdb_table_function table_function, duckdb_table_function_bind_t
bind);
void duckdb_table_function_set_init(duckdb_table_function table_function, duckdb_table_function_init_t
init);
void duckdb_table_function_set_local_init(duckdb_table_function table_function, duckdb_table_function_
init_t init);
void duckdb_table_function_set_function(duckdb_table_function table_function, duckdb_table_function_t
function);
void duckdb_table_function_supports_projection_pushdown(duckdb_table_function table_function, bool
pushdown);
duckdb_state duckdb_register_table_function(duckdb_connection con, duckdb_table_function function);
```

Table Function Bind

```
            void *duckdb_bind_get_extra_info(duckdb_bind_info info);
void duckdb_bind_add_result_column(duckdb_bind_info info, const char *name, duckdb_logical_type type);
idx_t duckdb_bind_get_parameter_count(duckdb_bind_info info);
duckdb_value duckdb_bind_get_parameter(duckdb_bind_info info, idx_t index);
```

```
duckdb_value duckdb_bind_get_named_parameter(duckdb_bind_info info, const char *name);
void duckdb_bind_set_bind_data(duckdb_bind_info info, void *bind_data, duckdb_delete_callback_t
destroy);
void duckdb_bind_set_cardinality(duckdb_bind_info info, idx_t cardinality, bool is_exact);
void duckdb_bind_set_error(duckdb_bind_info info, const char *error);
```

Table Function Init

```
            void *duckdb_init_get_extra_info(duckdb_init_info info);
void *duckdb_init_get_bind_data(duckdb_init_info info);
void duckdb_init_set_init_data(duckdb_init_info info, void *init_data, duckdb_delete_callback_t
destroy);
idx_t duckdb_init_get_column_count(duckdb_init_info info);
idx_t duckdb_init_get_column_index(duckdb_init_info info, idx_t column_index);
void duckdb_init_set_max_threads(duckdb_init_info info, idx_t max_threads);
void duckdb_init_set_error(duckdb_init_info info, const char *error);
```

Table Function

```
            void *duckdb_function_get_extra_info(duckdb_function_info info);
void *duckdb_function_get_bind_data(duckdb_function_info info);
void *duckdb_function_get_init_data(duckdb_function_info info);
void *duckdb_function_get_local_init_data(duckdb_function_info info);
void duckdb_function_set_error(duckdb_function_info info, const char *error);
```

`duckdb_create_table_function`

Creates a new empty table function.

The return value should be destroyed with `duckdb_destroy_table_function`.

Return Value The table function object.

Syntax

```
            duckdb_table_function duckdb_create_table_function(

);
```

`duckdb_destroy_table_function`

Destroys the given table function object.

Syntax

```
            void duckdb_destroy_table_function(
  duckdb_table_function *table_function
);
```

Parameters

- `table_function`: The table function to destroy

duckdb_table_function_set_name

Sets the name of the given table function.

Syntax

```
            void duckdb_table_function_set_name(
  duckdb_table_function table_function,
  const char *name
);
```

Parameters

- `table_function`: The table function
- `name`: The name of the table function

duckdb_table_function_add_parameter

Adds a parameter to the table function.

Syntax

```
            void duckdb_table_function_add_parameter(
  duckdb_table_function table_function,
  duckdb_logical_type type
);
```

Parameters

- `table_function`: The table function.
- `type`: The parameter type. Cannot contain INVALID.

duckdb_table_function_add_named_parameter

Adds a named parameter to the table function.

Syntax

```
            void duckdb_table_function_add_named_parameter(
  duckdb_table_function table_function,
  const char *name,
  duckdb_logical_type type
);
```

Parameters

- `table_function`: The table function.
- `name`: The parameter name.
- `type`: The parameter type. Cannot contain INVALID.

duckdb_table_function_set_extra_info

Assigns extra information to the table function that can be fetched during binding, etc.

Syntax

```
            void duckdb_table_function_set_extra_info(
  duckdb_table_function table_function,
  void *extra_info,
  duckdb_delete_callback_t destroy
);
```

Parameters

- `table_function`: The table function
- `extra_info`: The extra information
- `destroy`: The callback that will be called to destroy the extra information (if any)

`duckdb_table_function_set_bind`

Sets the bind function of the table function.

Syntax

```
            void duckdb_table_function_set_bind(
  duckdb_table_function table_function,
  duckdb_table_function_bind_t bind
);
```

Parameters

- `table_function`: The table function
- `bind`: The bind function

`duckdb_table_function_set_init`

Sets the init function of the table function.

Syntax

```
            void duckdb_table_function_set_init(
  duckdb_table_function table_function,
  duckdb_table_function_init_t init
);
```

Parameters

- `table_function`: The table function
- `init`: The init function

`duckdb_table_function_set_local_init`

Sets the thread-local init function of the table function.

Syntax

```
            void duckdb_table_function_set_local_init(
  duckdb_table_function table_function,
  duckdb_table_function_init_t init
);
```

Parameters

- `table_function`: The table function
- `init`: The init function

`duckdb_table_function_set_function`

Sets the main function of the table function.

Syntax

```
            void duckdb_table_function_set_function(
  duckdb_table_function table_function,
  duckdb_table_function_t function
);
```

Parameters

- `table_function`: The table function
- `function`: The function

`duckdb_table_function_supports_projection_pushdown`

Sets whether or not the given table function supports projection pushdown.

If this is set to true, the system will provide a list of all required columns in the `init` stage through the `duckdb_init_get_column_count` and `duckdb_init_get_column_index` functions. If this is set to false (the default), the system will expect all columns to be projected.

Syntax

```
            void duckdb_table_function_supports_projection_pushdown(
  duckdb_table_function table_function,
  bool pushdown
);
```

Parameters

- `table_function`: The table function
- `pushdown`: True if the table function supports projection pushdown, false otherwise.

`duckdb_register_table_function`

Register the table function object within the given connection.

The function requires at least a name, a bind function, an init function and a main function.

If the function is incomplete or a function with this name already exists DuckDBError is returned.

Syntax

```
            duckdb_state duckdb_register_table_function(
  duckdb_connection con,
  duckdb_table_function function
);
```

Parameters

- `con`: The connection to register it in.
- `function`: The function pointer

Return Value Whether or not the registration was successful.

`duckdb_bind_get_extra_info`

Retrieves the extra info of the function as set in `duckdb_table_function_set_extra_info`.

Syntax

```
            void *duckdb_bind_get_extra_info(
  duckdb_bind_info info
);
```

Parameters

- `info`: The info object

Return Value The extra info

`duckdb_bind_add_result_column`

Adds a result column to the output of the table function.

Syntax

```
            void duckdb_bind_add_result_column(
  duckdb_bind_info info,
  const char *name,
  duckdb_logical_type type
);
```

Parameters

- `info`: The table function's bind info.
- `name`: The column name.
- `type`: The logical column type.

`duckdb_bind_get_parameter_count`

Retrieves the number of regular (non-named) parameters to the function.

Syntax

```
            idx_t duckdb_bind_get_parameter_count(
  duckdb_bind_info info
);
```

Parameters

- `info`: The info object

Return Value The number of parameters

`duckdb_bind_get_parameter`

Retrieves the parameter at the given index.

The result must be destroyed with `duckdb_destroy_value`.

Syntax

```
            duckdb_value duckdb_bind_get_parameter(
  duckdb_bind_info info,
  idx_t index
);
```

Parameters

- `info`: The info object
- `index`: The index of the parameter to get

Return Value The value of the parameter. Must be destroyed with `duckdb_destroy_value`.

`duckdb_bind_get_named_parameter`

Retrieves a named parameter with the given name.

The result must be destroyed with `duckdb_destroy_value`.

Syntax

```
            duckdb_value duckdb_bind_get_named_parameter(
  duckdb_bind_info info,
  const char *name
);
```

Parameters

- `info`: The info object
- `name`: The name of the parameter

Return Value The value of the parameter. Must be destroyed with `duckdb_destroy_value`.

`duckdb_bind_set_bind_data`

Sets the user-provided bind data in the bind object of the table function. This object can be retrieved again during execution.

Syntax

```
            void duckdb_bind_set_bind_data(
  duckdb_bind_info info,
  void *bind_data,
  duckdb_delete_callback_t destroy
);
```

Parameters

- `info`: The bind info of the table function.
- `bind_data`: The bind data object.
- `destroy`: The callback to destroy the bind data (if any).

`duckdb_bind_set_cardinality`

Sets the cardinality estimate for the table function, used for optimization.

Syntax

```
            void duckdb_bind_set_cardinality(
  duckdb_bind_info info,
  idx_t cardinality,
  bool is_exact
);
```

Parameters

- `info`: The bind data object.
- `is_exact`: Whether or not the cardinality estimate is exact, or an approximation

`duckdb_bind_set_error`

Report that an error has occurred while calling bind on a table function.

Syntax

```
            void duckdb_bind_set_error(
  duckdb_bind_info info,
  const char *error
);
```

Parameters

- `info`: The info object
- `error`: The error message

`duckdb_init_get_extra_info`

Retrieves the extra info of the function as set in `duckdb_table_function_set_extra_info`.

Syntax

```
            void *duckdb_init_get_extra_info(
  duckdb_init_info info
);
```

Parameters

- `info`: The info object

Return Value The extra info

`duckdb_init_get_bind_data`

Gets the bind data set by `duckdb_bind_set_bind_data` during the bind.

Note that the bind data should be considered as read-only. For tracking state, use the init data instead.

Syntax

```
            void *duckdb_init_get_bind_data(
  duckdb_init_info info
);
```

Parameters

- `info`: The info object

Return Value The bind data object

`duckdb_init_set_init_data`

Sets the user-provided init data in the init object. This object can be retrieved again during execution.

Syntax

```
            void duckdb_init_set_init_data(
  duckdb_init_info info,
  void *init_data,
  duckdb_delete_callback_t destroy
);
```

Parameters

- `info`: The info object
- `init_data`: The init data object.
- `destroy`: The callback that will be called to destroy the init data (if any)

`duckdb_init_get_column_count`

Returns the number of projected columns.

This function must be used if projection pushdown is enabled to figure out which columns to emit.

Syntax

```
            idx_t duckdb_init_get_column_count(
  duckdb_init_info info
);
```

Parameters

- `info`: The info object

Return Value The number of projected columns.

`duckdb_init_get_column_index`

Returns the column index of the projected column at the specified position.

This function must be used if projection pushdown is enabled to figure out which columns to emit.

Syntax

```
            idx_t duckdb_init_get_column_index(
  duckdb_init_info info,
  idx_t column_index
);
```

Parameters

- `info`: The info object
- `column_index`: The index at which to get the projected column index, from 0..duckdb_init_get_column_count(info)

Return Value The column index of the projected column.

`duckdb_init_set_max_threads`

Sets how many threads can process this table function in parallel (default: 1)

Syntax

```
            void duckdb_init_set_max_threads(
  duckdb_init_info info,
  idx_t max_threads
);
```

Parameters

- `info`: The info object
- `max_threads`: The maximum amount of threads that can process this table function

`duckdb_init_set_error`

Report that an error has occurred while calling init.

Syntax

```
            void duckdb_init_set_error(
  duckdb_init_info info,
  const char *error
);
```

Parameters

- `info`: The info object
- `error`: The error message

`duckdb_function_get_extra_info`

Retrieves the extra info of the function as set in `duckdb_table_function_set_extra_info`.

Syntax

```
            void *duckdb_function_get_extra_info(
  duckdb_function_info info
);
```

Parameters

- `info`: The info object

Return Value The extra info

`duckdb_function_get_bind_data`

Gets the table function's bind data set by `duckdb_bind_set_bind_data`.

Note that the bind data is read-only. For tracking state, use the init data instead.

Syntax

```
            void *duckdb_function_get_bind_data(
  duckdb_function_info info
);
```

Parameters

- `info`: The function info object.

Return Value The bind data object.

`duckdb_function_get_init_data`

Gets the init data set by `duckdb_init_set_init_data` during the init.

Syntax

```
            void *duckdb_function_get_init_data(
  duckdb_function_info info
);
```

Parameters

- `info`: The info object

Return Value The init data object

`duckdb_function_get_local_init_data`

Gets the thread-local init data set by `duckdb_init_set_init_data` during the local_init.

Syntax

```
            void *duckdb_function_get_local_init_data(
  duckdb_function_info info
);
```

Parameters

- `info`: The info object

Return Value The init data object

`duckdb_function_set_error`

Report that an error has occurred while executing the function.

Syntax

```
            void duckdb_function_set_error(
  duckdb_function_info info,
  const char *error
);
```

Parameters

- `info`: The info object
- `error`: The error message

Replacement Scans

The replacement scan API can be used to register a callback that is called when a table is read that does not exist in the catalog. For example, when a query such as `SELECT * FROM my_table` is executed and `my_table` does not exist, the replacement scan callback will be called with `my_table` as parameter. The replacement scan can then insert a table function with a specific parameter to replace the read of the table.

API Reference Overview

```
            void duckdb_add_replacement_scan(duckdb_database db, duckdb_replacement_callback_t
            replacement, void *extra_data, duckdb_delete_callback_t delete_callback);
void duckdb_replacement_scan_set_function_name(duckdb_replacement_scan_info info, const char *function_
name);
void duckdb_replacement_scan_add_parameter(duckdb_replacement_scan_info info, duckdb_value parameter);
void duckdb_replacement_scan_set_error(duckdb_replacement_scan_info info, const char *error);
```

`duckdb_add_replacement_scan`

Add a replacement scan definition to the specified database.

Syntax

```
            void duckdb_add_replacement_scan(
  duckdb_database db,
  duckdb_replacement_callback_t replacement,
  void *extra_data,
  duckdb_delete_callback_t delete_callback
);
```

Parameters

- db: The database object to add the replacement scan to
- `replacement`: The replacement scan callback
- `extra_data`: Extra data that is passed back into the specified callback
- `delete_callback`: The delete callback to call on the extra data, if any

`duckdb_replacement_scan_set_function_name`

Sets the replacement function name. If this function is called in the replacement callback, the replacement scan is performed. If it is not called, the replacement callback is not performed.

Syntax

```
            void duckdb_replacement_scan_set_function_name(
  duckdb_replacement_scan_info info,
  const char *function_name
);
```

Parameters

- `info`: The info object
- `function_name`: The function name to substitute.

`duckdb_replacement_scan_add_parameter`

Adds a parameter to the replacement scan function.

Syntax

```
            void duckdb_replacement_scan_add_parameter(
  duckdb_replacement_scan_info info,
  duckdb_value parameter
);
```

Parameters

- `info`: The info object
- `parameter`: The parameter to add.

`duckdb_replacement_scan_set_error`

Report that an error has occurred while executing the replacement scan.

Syntax

```
            void duckdb_replacement_scan_set_error(
  duckdb_replacement_scan_info info,
  const char *error
);
```

Parameters

- `info`: The info object
- `error`: The error message

Complete API

This page contains the reference for DuckDB's C API.

> The reference contains several deprecation notices. These concern methods whose long-term availability is not guaranteed as they may be removed in the future. That said, DuckDB's developers plan to carry out deprecations slowly as several of the deprecated methods do not yet have a fully functional alternative. Therefore, they will not removed before the alternative is available, and even then, there will be a grace period of a few minor versions before removing them. The reason that the methods are already deprecated in v1.0 is to denote that they are not part of the v1.0 stable API, which contains methods that are available long-term.

API Reference Overview

Open Connect

```
            duckdb_instance_cache duckdb_create_instance_cache();
duckdb_state duckdb_get_or_create_from_cache(duckdb_instance_cache instance_cache, const char *path,
duckdb_database *out_database, duckdb_config config, char **out_error);
void duckdb_destroy_instance_cache(duckdb_instance_cache *instance_cache);
duckdb_state duckdb_open(const char *path, duckdb_database *out_database);
duckdb_state duckdb_open_ext(const char *path, duckdb_database *out_database, duckdb_config config, char
**out_error);
void duckdb_close(duckdb_database *database);
duckdb_state duckdb_connect(duckdb_database database, duckdb_connection *out_connection);
void duckdb_interrupt(duckdb_connection connection);
duckdb_query_progress_type duckdb_query_progress(duckdb_connection connection);
```

```
void duckdb_disconnect(duckdb_connection *connection);
void duckdb_connection_get_client_context(duckdb_connection connection, duckdb_client_context *out_
context);
idx_t duckdb_client_context_get_connection_id(duckdb_client_context context);
void duckdb_destroy_client_context(duckdb_client_context *context);
const char *duckdb_library_version();
duckdb_value duckdb_get_table_names(duckdb_connection connection, const char *query, bool qualified);
```

Configuration

```
            duckdb_state duckdb_create_config(duckdb_config *out_config);
size_t duckdb_config_count();
duckdb_state duckdb_get_config_flag(size_t index, const char **out_name, const char **out_description);
duckdb_state duckdb_set_config(duckdb_config config, const char *name, const char *option);
void duckdb_destroy_config(duckdb_config *config);
```

Query Execution

```
            duckdb_state duckdb_query(duckdb_connection connection, const char *query, duckdb_result
*out_result);
void duckdb_destroy_result(duckdb_result *result);
const char *duckdb_column_name(duckdb_result *result, idx_t col);
duckdb_type duckdb_column_type(duckdb_result *result, idx_t col);
duckdb_statement_type duckdb_result_statement_type(duckdb_result result);
duckdb_logical_type duckdb_column_logical_type(duckdb_result *result, idx_t col);
idx_t duckdb_column_count(duckdb_result *result);
idx_t duckdb_row_count(duckdb_result *result);
idx_t duckdb_rows_changed(duckdb_result *result);
void *duckdb_column_data(duckdb_result *result, idx_t col);
bool *duckdb_nullmask_data(duckdb_result *result, idx_t col);
const char *duckdb_result_error(duckdb_result *result);
duckdb_error_type duckdb_result_error_type(duckdb_result *result);
```

Result Functions

```
            duckdb_data_chunk duckdb_result_get_chunk(duckdb_result result, idx_t chunk_index);
bool duckdb_result_is_streaming(duckdb_result result);
idx_t duckdb_result_chunk_count(duckdb_result result);
duckdb_result_type duckdb_result_return_type(duckdb_result result);
```

Safe Fetch Functions

```
            bool duckdb_value_boolean(duckdb_result *result, idx_t col, idx_t row);
int8_t duckdb_value_int8(duckdb_result *result, idx_t col, idx_t row);
int16_t duckdb_value_int16(duckdb_result *result, idx_t col, idx_t row);
int32_t duckdb_value_int32(duckdb_result *result, idx_t col, idx_t row);
int64_t duckdb_value_int64(duckdb_result *result, idx_t col, idx_t row);
duckdb_hugeint duckdb_value_hugeint(duckdb_result *result, idx_t col, idx_t row);
duckdb_uhugeint duckdb_value_uhugeint(duckdb_result *result, idx_t col, idx_t row);
duckdb_decimal duckdb_value_decimal(duckdb_result *result, idx_t col, idx_t row);
uint8_t duckdb_value_uint8(duckdb_result *result, idx_t col, idx_t row);
uint16_t duckdb_value_uint16(duckdb_result *result, idx_t col, idx_t row);
uint32_t duckdb_value_uint32(duckdb_result *result, idx_t col, idx_t row);
uint64_t duckdb_value_uint64(duckdb_result *result, idx_t col, idx_t row);
float duckdb_value_float(duckdb_result *result, idx_t col, idx_t row);
```

```
double duckdb_value_double(duckdb_result *result, idx_t col, idx_t row);
duckdb_date duckdb_value_date(duckdb_result *result, idx_t col, idx_t row);
duckdb_time duckdb_value_time(duckdb_result *result, idx_t col, idx_t row);
duckdb_timestamp duckdb_value_timestamp(duckdb_result *result, idx_t col, idx_t row);
duckdb_interval duckdb_value_interval(duckdb_result *result, idx_t col, idx_t row);
char *duckdb_value_varchar(duckdb_result *result, idx_t col, idx_t row);
duckdb_string duckdb_value_string(duckdb_result *result, idx_t col, idx_t row);
char *duckdb_value_varchar_internal(duckdb_result *result, idx_t col, idx_t row);
duckdb_string duckdb_value_string_internal(duckdb_result *result, idx_t col, idx_t row);
duckdb_blob duckdb_value_blob(duckdb_result *result, idx_t col, idx_t row);
bool duckdb_value_is_null(duckdb_result *result, idx_t col, idx_t row);
```

Helpers

```
            void *duckdb_malloc(size_t size);
void duckdb_free(void *ptr);
idx_t duckdb_vector_size();
bool duckdb_string_is_inlined(duckdb_string_t string);
uint32_t duckdb_string_t_length(duckdb_string_t string);
const char *duckdb_string_t_data(duckdb_string_t *string);
```

Date Time Timestamp Helpers

```
            duckdb_date_struct duckdb_from_date(duckdb_date date);
duckdb_date duckdb_to_date(duckdb_date_struct date);
bool duckdb_is_finite_date(duckdb_date date);
duckdb_time_struct duckdb_from_time(duckdb_time time);
duckdb_time_tz duckdb_create_time_tz(int64_t micros, int32_t offset);
duckdb_time_tz_struct duckdb_from_time_tz(duckdb_time_tz micros);
duckdb_time duckdb_to_time(duckdb_time_struct time);
duckdb_timestamp_struct duckdb_from_timestamp(duckdb_timestamp ts);
duckdb_timestamp duckdb_to_timestamp(duckdb_timestamp_struct ts);
bool duckdb_is_finite_timestamp(duckdb_timestamp ts);
bool duckdb_is_finite_timestamp_s(duckdb_timestamp_s ts);
bool duckdb_is_finite_timestamp_ms(duckdb_timestamp_ms ts);
bool duckdb_is_finite_timestamp_ns(duckdb_timestamp_ns ts);
```

Hugeint Helpers

```
            double duckdb_hugeint_to_double(duckdb_hugeint val);
duckdb_hugeint duckdb_double_to_hugeint(double val);
```

Unsigned Hugeint Helpers

```
            double duckdb_uhugeint_to_double(duckdb_uhugeint val);
duckdb_uhugeint duckdb_double_to_uhugeint(double val);
```

Decimal Helpers

```
            duckdb_decimal duckdb_double_to_decimal(double val, uint8_t width, uint8_t scale);
double duckdb_decimal_to_double(duckdb_decimal val);
```

Prepared Statements

```
duckdb_state duckdb_prepare(duckdb_connection connection, const char *query, duckdb_
prepared_statement *out_prepared_statement);
void duckdb_destroy_prepare(duckdb_prepared_statement *prepared_statement);
const char *duckdb_prepare_error(duckdb_prepared_statement prepared_statement);
idx_t duckdb_nparams(duckdb_prepared_statement prepared_statement);
const char *duckdb_parameter_name(duckdb_prepared_statement prepared_statement, idx_t index);
duckdb_type duckdb_param_type(duckdb_prepared_statement prepared_statement, idx_t param_idx);
duckdb_logical_type duckdb_param_logical_type(duckdb_prepared_statement prepared_statement, idx_t param_
idx);
duckdb_state duckdb_clear_bindings(duckdb_prepared_statement prepared_statement);
duckdb_statement_type duckdb_prepared_statement_type(duckdb_prepared_statement statement);
```

Bind Values to Prepared Statements

```
duckdb_state duckdb_bind_value(duckdb_prepared_statement prepared_statement, idx_t param_
idx, duckdb_value val);
duckdb_state duckdb_bind_parameter_index(duckdb_prepared_statement prepared_statement, idx_t *param_idx_
out, const char *name);
duckdb_state duckdb_bind_boolean(duckdb_prepared_statement prepared_statement, idx_t param_idx, bool
val);
duckdb_state duckdb_bind_int8(duckdb_prepared_statement prepared_statement, idx_t param_idx, int8_t
val);
duckdb_state duckdb_bind_int16(duckdb_prepared_statement prepared_statement, idx_t param_idx, int16_t
val);
duckdb_state duckdb_bind_int32(duckdb_prepared_statement prepared_statement, idx_t param_idx, int32_t
val);
duckdb_state duckdb_bind_int64(duckdb_prepared_statement prepared_statement, idx_t param_idx, int64_t
val);
duckdb_state duckdb_bind_hugeint(duckdb_prepared_statement prepared_statement, idx_t param_idx, duckdb_
hugeint val);
duckdb_state duckdb_bind_uhugeint(duckdb_prepared_statement prepared_statement, idx_t param_idx, duckdb_
uhugeint val);
duckdb_state duckdb_bind_decimal(duckdb_prepared_statement prepared_statement, idx_t param_idx, duckdb_
decimal val);
duckdb_state duckdb_bind_uint8(duckdb_prepared_statement prepared_statement, idx_t param_idx, uint8_t
val);
duckdb_state duckdb_bind_uint16(duckdb_prepared_statement prepared_statement, idx_t param_idx, uint16_t
val);
duckdb_state duckdb_bind_uint32(duckdb_prepared_statement prepared_statement, idx_t param_idx, uint32_t
val);
duckdb_state duckdb_bind_uint64(duckdb_prepared_statement prepared_statement, idx_t param_idx, uint64_t
val);
duckdb_state duckdb_bind_float(duckdb_prepared_statement prepared_statement, idx_t param_idx, float
val);
duckdb_state duckdb_bind_double(duckdb_prepared_statement prepared_statement, idx_t param_idx, double
val);
duckdb_state duckdb_bind_date(duckdb_prepared_statement prepared_statement, idx_t param_idx, duckdb_date
val);
duckdb_state duckdb_bind_time(duckdb_prepared_statement prepared_statement, idx_t param_idx, duckdb_time
val);
duckdb_state duckdb_bind_timestamp(duckdb_prepared_statement prepared_statement, idx_t param_idx,
duckdb_timestamp val);
duckdb_state duckdb_bind_timestamp_tz(duckdb_prepared_statement prepared_statement, idx_t param_idx,
duckdb_timestamp val);
duckdb_state duckdb_bind_interval(duckdb_prepared_statement prepared_statement, idx_t param_idx, duckdb_
interval val);
```

```
duckdb_state duckdb_bind_varchar(duckdb_prepared_statement prepared_statement, idx_t param_idx, const
char *val);
duckdb_state duckdb_bind_varchar_length(duckdb_prepared_statement prepared_statement, idx_t param_idx,
const char *val, idx_t length);
duckdb_state duckdb_bind_blob(duckdb_prepared_statement prepared_statement, idx_t param_idx, const void
*data, idx_t length);
duckdb_state duckdb_bind_null(duckdb_prepared_statement prepared_statement, idx_t param_idx);
```

Execute Prepared Statements

```
            duckdb_state duckdb_execute_prepared(duckdb_prepared_statement prepared_statement, duckdb_
result *out_result);
duckdb_state duckdb_execute_prepared_streaming(duckdb_prepared_statement prepared_statement, duckdb_
result *out_result);
```

Extract Statements

```
            idx_t duckdb_extract_statements(duckdb_connection connection, const char *query, duckdb_
extracted_statements *out_extracted_statements);
duckdb_state duckdb_prepare_extracted_statement(duckdb_connection connection, duckdb_extracted_
statements extracted_statements, idx_t index, duckdb_prepared_statement *out_prepared_statement);
const char *duckdb_extract_statements_error(duckdb_extracted_statements extracted_statements);
void duckdb_destroy_extracted(duckdb_extracted_statements *extracted_statements);
```

Pending Result Interface

```
            duckdb_state duckdb_pending_prepared(duckdb_prepared_statement prepared_statement, duckdb_
pending_result *out_result);
duckdb_state duckdb_pending_prepared_streaming(duckdb_prepared_statement prepared_statement, duckdb_
pending_result *out_result);
void duckdb_destroy_pending(duckdb_pending_result *pending_result);
const char *duckdb_pending_error(duckdb_pending_result pending_result);
duckdb_pending_state duckdb_pending_execute_task(duckdb_pending_result pending_result);
duckdb_pending_state duckdb_pending_execute_check_state(duckdb_pending_result pending_result);
duckdb_state duckdb_execute_pending(duckdb_pending_result pending_result, duckdb_result *out_result);
bool duckdb_pending_execution_is_finished(duckdb_pending_state pending_state);
```

Value Interface

```
            void duckdb_destroy_value(duckdb_value *value);
duckdb_value duckdb_create_varchar(const char *text);
duckdb_value duckdb_create_varchar_length(const char *text, idx_t length);
duckdb_value duckdb_create_bool(bool input);
duckdb_value duckdb_create_int8(int8_t input);
duckdb_value duckdb_create_uint8(uint8_t input);
duckdb_value duckdb_create_int16(int16_t input);
duckdb_value duckdb_create_uint16(uint16_t input);
duckdb_value duckdb_create_int32(int32_t input);
duckdb_value duckdb_create_uint32(uint32_t input);
duckdb_value duckdb_create_uint64(uint64_t input);
duckdb_value duckdb_create_int64(int64_t val);
duckdb_value duckdb_create_hugeint(duckdb_hugeint input);
duckdb_value duckdb_create_uhugeint(duckdb_uhugeint input);
duckdb_value duckdb_create_varint(duckdb_varint input);
duckdb_value duckdb_create_decimal(duckdb_decimal input);
```

```
duckdb_value duckdb_create_float(float input);
duckdb_value duckdb_create_double(double input);
duckdb_value duckdb_create_date(duckdb_date input);
duckdb_value duckdb_create_time(duckdb_time input);
duckdb_value duckdb_create_time_tz_value(duckdb_time_tz value);
duckdb_value duckdb_create_timestamp(duckdb_timestamp input);
duckdb_value duckdb_create_timestamp_tz(duckdb_timestamp input);
duckdb_value duckdb_create_timestamp_s(duckdb_timestamp_s input);
duckdb_value duckdb_create_timestamp_ms(duckdb_timestamp_ms input);
duckdb_value duckdb_create_timestamp_ns(duckdb_timestamp_ns input);
duckdb_value duckdb_create_interval(duckdb_interval input);
duckdb_value duckdb_create_blob(const uint8_t *data, idx_t length);
duckdb_value duckdb_create_bit(duckdb_bit input);
duckdb_value duckdb_create_uuid(duckdb_uhugeint input);
bool duckdb_get_bool(duckdb_value val);
int8_t duckdb_get_int8(duckdb_value val);
uint8_t duckdb_get_uint8(duckdb_value val);
int16_t duckdb_get_int16(duckdb_value val);
uint16_t duckdb_get_uint16(duckdb_value val);
int32_t duckdb_get_int32(duckdb_value val);
uint32_t duckdb_get_uint32(duckdb_value val);
int64_t duckdb_get_int64(duckdb_value val);
uint64_t duckdb_get_uint64(duckdb_value val);
duckdb_hugeint duckdb_get_hugeint(duckdb_value val);
duckdb_uhugeint duckdb_get_uhugeint(duckdb_value val);
duckdb_varint duckdb_get_varint(duckdb_value val);
duckdb_decimal duckdb_get_decimal(duckdb_value val);
float duckdb_get_float(duckdb_value val);
double duckdb_get_double(duckdb_value val);
duckdb_date duckdb_get_date(duckdb_value val);
duckdb_time duckdb_get_time(duckdb_value val);
duckdb_time_tz duckdb_get_time_tz(duckdb_value val);
duckdb_timestamp duckdb_get_timestamp(duckdb_value val);
duckdb_timestamp duckdb_get_timestamp_tz(duckdb_value val);
duckdb_timestamp_s duckdb_get_timestamp_s(duckdb_value val);
duckdb_timestamp_ms duckdb_get_timestamp_ms(duckdb_value val);
duckdb_timestamp_ns duckdb_get_timestamp_ns(duckdb_value val);
duckdb_interval duckdb_get_interval(duckdb_value val);
duckdb_logical_type duckdb_get_value_type(duckdb_value val);
duckdb_blob duckdb_get_blob(duckdb_value val);
duckdb_bit duckdb_get_bit(duckdb_value val);
duckdb_uhugeint duckdb_get_uuid(duckdb_value val);
char *duckdb_get_varchar(duckdb_value value);
duckdb_value duckdb_create_struct_value(duckdb_logical_type type, duckdb_value *values);
duckdb_value duckdb_create_list_value(duckdb_logical_type type, duckdb_value *values, idx_t value_
count);
duckdb_value duckdb_create_array_value(duckdb_logical_type type, duckdb_value *values, idx_t value_
count);
duckdb_value duckdb_create_map_value(duckdb_logical_type map_type, duckdb_value *keys, duckdb_value
*values, idx_t entry_count);
duckdb_value duckdb_create_union_value(duckdb_logical_type union_type, idx_t tag_index, duckdb_value
value);
idx_t duckdb_get_map_size(duckdb_value value);
duckdb_value duckdb_get_map_key(duckdb_value value, idx_t index);
duckdb_value duckdb_get_map_value(duckdb_value value, idx_t index);
bool duckdb_is_null_value(duckdb_value value);
duckdb_value duckdb_create_null_value();
idx_t duckdb_get_list_size(duckdb_value value);
duckdb_value duckdb_get_list_child(duckdb_value value, idx_t index);
```

```
duckdb_value duckdb_create_enum_value(duckdb_logical_type type, uint64_t value);
uint64_t duckdb_get_enum_value(duckdb_value value);
duckdb_value duckdb_get_struct_child(duckdb_value value, idx_t index);
char *duckdb_value_to_string(duckdb_value value);
```

Logical Type Interface

```
            duckdb_logical_type duckdb_create_logical_type(duckdb_type type);
char *duckdb_logical_type_get_alias(duckdb_logical_type type);
void duckdb_logical_type_set_alias(duckdb_logical_type type, const char *alias);
duckdb_logical_type duckdb_create_list_type(duckdb_logical_type type);
duckdb_logical_type duckdb_create_array_type(duckdb_logical_type type, idx_t array_size);
duckdb_logical_type duckdb_create_map_type(duckdb_logical_type key_type, duckdb_logical_type value_
type);
duckdb_logical_type duckdb_create_union_type(duckdb_logical_type *member_types, const char **member_
names, idx_t member_count);
duckdb_logical_type duckdb_create_struct_type(duckdb_logical_type *member_types, const char **member_
names, idx_t member_count);
duckdb_logical_type duckdb_create_enum_type(const char **member_names, idx_t member_count);
duckdb_logical_type duckdb_create_decimal_type(uint8_t width, uint8_t scale);
duckdb_type duckdb_get_type_id(duckdb_logical_type type);
uint8_t duckdb_decimal_width(duckdb_logical_type type);
uint8_t duckdb_decimal_scale(duckdb_logical_type type);
duckdb_type duckdb_decimal_internal_type(duckdb_logical_type type);
duckdb_type duckdb_enum_internal_type(duckdb_logical_type type);
uint32_t duckdb_enum_dictionary_size(duckdb_logical_type type);
char *duckdb_enum_dictionary_value(duckdb_logical_type type, idx_t index);
duckdb_logical_type duckdb_list_type_child_type(duckdb_logical_type type);
duckdb_logical_type duckdb_array_type_child_type(duckdb_logical_type type);
idx_t duckdb_array_type_array_size(duckdb_logical_type type);
duckdb_logical_type duckdb_map_type_key_type(duckdb_logical_type type);
duckdb_logical_type duckdb_map_type_value_type(duckdb_logical_type type);
idx_t duckdb_struct_type_child_count(duckdb_logical_type type);
char *duckdb_struct_type_child_name(duckdb_logical_type type, idx_t index);
duckdb_logical_type duckdb_struct_type_child_type(duckdb_logical_type type, idx_t index);
idx_t duckdb_union_type_member_count(duckdb_logical_type type);
char *duckdb_union_type_member_name(duckdb_logical_type type, idx_t index);
duckdb_logical_type duckdb_union_type_member_type(duckdb_logical_type type, idx_t index);
void duckdb_destroy_logical_type(duckdb_logical_type *type);
duckdb_state duckdb_register_logical_type(duckdb_connection con, duckdb_logical_type type, duckdb_
create_type_info info);
```

Data Chunk Interface

```
            duckdb_data_chunk duckdb_create_data_chunk(duckdb_logical_type *types, idx_t column_count);
void duckdb_destroy_data_chunk(duckdb_data_chunk *chunk);
void duckdb_data_chunk_reset(duckdb_data_chunk chunk);
idx_t duckdb_data_chunk_get_column_count(duckdb_data_chunk chunk);
duckdb_vector duckdb_data_chunk_get_vector(duckdb_data_chunk chunk, idx_t col_idx);
idx_t duckdb_data_chunk_get_size(duckdb_data_chunk chunk);
void duckdb_data_chunk_set_size(duckdb_data_chunk chunk, idx_t size);
```

Vector Interface

```
            duckdb_vector duckdb_create_vector(duckdb_logical_type type, idx_t capacity);
void duckdb_destroy_vector(duckdb_vector *vector);
```

```
duckdb_logical_type duckdb_vector_get_column_type(duckdb_vector vector);
void *duckdb_vector_get_data(duckdb_vector vector);
uint64_t *duckdb_vector_get_validity(duckdb_vector vector);
void duckdb_vector_ensure_validity_writable(duckdb_vector vector);
void duckdb_vector_assign_string_element(duckdb_vector vector, idx_t index, const char *str);
void duckdb_vector_assign_string_element_len(duckdb_vector vector, idx_t index, const char *str, idx_t
str_len);
duckdb_vector duckdb_list_vector_get_child(duckdb_vector vector);
idx_t duckdb_list_vector_get_size(duckdb_vector vector);
duckdb_state duckdb_list_vector_set_size(duckdb_vector vector, idx_t size);
duckdb_state duckdb_list_vector_reserve(duckdb_vector vector, idx_t required_capacity);
duckdb_vector duckdb_struct_vector_get_child(duckdb_vector vector, idx_t index);
duckdb_vector duckdb_array_vector_get_child(duckdb_vector vector);
void duckdb_slice_vector(duckdb_vector vector, duckdb_selection_vector selection, idx_t len);
void duckdb_vector_reference_value(duckdb_vector vector, duckdb_value value);
void duckdb_vector_reference_vector(duckdb_vector to_vector, duckdb_vector from_vector);
```

Validity Mask Functions

```
            bool duckdb_validity_row_is_valid(uint64_t *validity, idx_t row);
void duckdb_validity_set_row_validity(uint64_t *validity, idx_t row, bool valid);
void duckdb_validity_set_row_invalid(uint64_t *validity, idx_t row);
void duckdb_validity_set_row_valid(uint64_t *validity, idx_t row);
```

Scalar Functions

```
            duckdb_scalar_function duckdb_create_scalar_function();
void duckdb_destroy_scalar_function(duckdb_scalar_function *scalar_function);
void duckdb_scalar_function_set_name(duckdb_scalar_function scalar_function, const char *name);
void duckdb_scalar_function_set_varargs(duckdb_scalar_function scalar_function, duckdb_logical_type
type);
void duckdb_scalar_function_set_special_handling(duckdb_scalar_function scalar_function);
void duckdb_scalar_function_set_volatile(duckdb_scalar_function scalar_function);
void duckdb_scalar_function_add_parameter(duckdb_scalar_function scalar_function, duckdb_logical_type
type);
void duckdb_scalar_function_set_return_type(duckdb_scalar_function scalar_function, duckdb_logical_type
type);
void duckdb_scalar_function_set_extra_info(duckdb_scalar_function scalar_function, void *extra_info,
duckdb_delete_callback_t destroy);
void duckdb_scalar_function_set_bind(duckdb_scalar_function scalar_function, duckdb_scalar_function_
bind_t bind);
void duckdb_scalar_function_set_bind_data(duckdb_bind_info info, void *bind_data, duckdb_delete_
callback_t destroy);
void duckdb_scalar_function_bind_set_error(duckdb_bind_info info, const char *error);
void duckdb_scalar_function_set_function(duckdb_scalar_function scalar_function, duckdb_scalar_function_
t function);
duckdb_state duckdb_register_scalar_function(duckdb_connection con, duckdb_scalar_function scalar_
function);
void *duckdb_scalar_function_get_extra_info(duckdb_function_info info);
void *duckdb_scalar_function_get_bind_data(duckdb_function_info info);
void duckdb_scalar_function_get_client_context(duckdb_bind_info info, duckdb_client_context *out_
context);
void duckdb_scalar_function_set_error(duckdb_function_info info, const char *error);
duckdb_scalar_function_set duckdb_create_scalar_function_set(const char *name);
void duckdb_destroy_scalar_function_set(duckdb_scalar_function_set *scalar_function_set);
duckdb_state duckdb_add_scalar_function_to_set(duckdb_scalar_function_set set, duckdb_scalar_function
function);
```

```
duckdb_state duckdb_register_scalar_function_set(duckdb_connection con, duckdb_scalar_function_set set);
```

Selection Vector Interface

```
            duckdb_selection_vector duckdb_create_selection_vector(idx_t size);
void duckdb_destroy_selection_vector(duckdb_selection_vector vector);
sel_t *duckdb_selection_vector_get_data_ptr(duckdb_selection_vector vector);
```

Aggregate Functions

```
            duckdb_aggregate_function duckdb_create_aggregate_function();
void duckdb_destroy_aggregate_function(duckdb_aggregate_function *aggregate_function);
void duckdb_aggregate_function_set_name(duckdb_aggregate_function aggregate_function, const char *name);
void duckdb_aggregate_function_add_parameter(duckdb_aggregate_function aggregate_function, duckdb_
logical_type type);
void duckdb_aggregate_function_set_return_type(duckdb_aggregate_function aggregate_function, duckdb_
logical_type type);
void duckdb_aggregate_function_set_functions(duckdb_aggregate_function aggregate_function, duckdb_
aggregate_state_size state_size, duckdb_aggregate_init_t state_init, duckdb_aggregate_update_t update,
duckdb_aggregate_combine_t combine, duckdb_aggregate_finalize_t finalize);
void duckdb_aggregate_function_set_destructor(duckdb_aggregate_function aggregate_function, duckdb_
aggregate_destroy_t destroy);
duckdb_state duckdb_register_aggregate_function(duckdb_connection con, duckdb_aggregate_function
aggregate_function);
void duckdb_aggregate_function_set_special_handling(duckdb_aggregate_function aggregate_function);
void duckdb_aggregate_function_set_extra_info(duckdb_aggregate_function aggregate_function, void *extra_
info, duckdb_delete_callback_t destroy);
void *duckdb_aggregate_function_get_extra_info(duckdb_function_info info);
void duckdb_aggregate_function_set_error(duckdb_function_info info, const char *error);
duckdb_aggregate_function_set duckdb_create_aggregate_function_set(const char *name);
void duckdb_destroy_aggregate_function_set(duckdb_aggregate_function_set *aggregate_function_set);
duckdb_state duckdb_add_aggregate_function_to_set(duckdb_aggregate_function_set set, duckdb_aggregate_
function function);
duckdb_state duckdb_register_aggregate_function_set(duckdb_connection con, duckdb_aggregate_function_set
set);
```

Table Functions

```
            duckdb_table_function duckdb_create_table_function();
void duckdb_destroy_table_function(duckdb_table_function *table_function);
void duckdb_table_function_set_name(duckdb_table_function table_function, const char *name);
void duckdb_table_function_add_parameter(duckdb_table_function table_function, duckdb_logical_type
type);
void duckdb_table_function_add_named_parameter(duckdb_table_function table_function, const char *name,
duckdb_logical_type type);
void duckdb_table_function_set_extra_info(duckdb_table_function table_function, void *extra_info,
duckdb_delete_callback_t destroy);
void duckdb_table_function_set_bind(duckdb_table_function table_function, duckdb_table_function_bind_t
bind);
void duckdb_table_function_set_init(duckdb_table_function table_function, duckdb_table_function_init_t
init);
void duckdb_table_function_set_local_init(duckdb_table_function table_function, duckdb_table_function_
init_t init);
void duckdb_table_function_set_function(duckdb_table_function table_function, duckdb_table_function_t
function);
```

```
void duckdb_table_function_supports_projection_pushdown(duckdb_table_function table_function, bool
pushdown);
duckdb_state duckdb_register_table_function(duckdb_connection con, duckdb_table_function function);
```

Table Function Bind

```
            void *duckdb_bind_get_extra_info(duckdb_bind_info info);
void duckdb_bind_add_result_column(duckdb_bind_info info, const char *name, duckdb_logical_type type);
idx_t duckdb_bind_get_parameter_count(duckdb_bind_info info);
duckdb_value duckdb_bind_get_parameter(duckdb_bind_info info, idx_t index);
duckdb_value duckdb_bind_get_named_parameter(duckdb_bind_info info, const char *name);
void duckdb_bind_set_bind_data(duckdb_bind_info info, void *bind_data, duckdb_delete_callback_t
destroy);
void duckdb_bind_set_cardinality(duckdb_bind_info info, idx_t cardinality, bool is_exact);
void duckdb_bind_set_error(duckdb_bind_info info, const char *error);
```

Table Function Init

```
            void *duckdb_init_get_extra_info(duckdb_init_info info);
void *duckdb_init_get_bind_data(duckdb_init_info info);
void duckdb_init_set_init_data(duckdb_init_info info, void *init_data, duckdb_delete_callback_t
destroy);
idx_t duckdb_init_get_column_count(duckdb_init_info info);
idx_t duckdb_init_get_column_index(duckdb_init_info info, idx_t column_index);
void duckdb_init_set_max_threads(duckdb_init_info info, idx_t max_threads);
void duckdb_init_set_error(duckdb_init_info info, const char *error);
```

Table Function

```
            void *duckdb_function_get_extra_info(duckdb_function_info info);
void *duckdb_function_get_bind_data(duckdb_function_info info);
void *duckdb_function_get_init_data(duckdb_function_info info);
void *duckdb_function_get_local_init_data(duckdb_function_info info);
void duckdb_function_set_error(duckdb_function_info info, const char *error);
```

Replacement Scans

```
            void duckdb_add_replacement_scan(duckdb_database db, duckdb_replacement_callback_t
            replacement, void *extra_data, duckdb_delete_callback_t delete_callback);
void duckdb_replacement_scan_set_function_name(duckdb_replacement_scan_info info, const char *function_
name);
void duckdb_replacement_scan_add_parameter(duckdb_replacement_scan_info info, duckdb_value parameter);
void duckdb_replacement_scan_set_error(duckdb_replacement_scan_info info, const char *error);
```

Profiling Info

```
            duckdb_profiling_info duckdb_get_profiling_info(duckdb_connection connection);
duckdb_value duckdb_profiling_info_get_value(duckdb_profiling_info info, const char *key);
duckdb_value duckdb_profiling_info_get_metrics(duckdb_profiling_info info);
idx_t duckdb_profiling_info_get_child_count(duckdb_profiling_info info);
duckdb_profiling_info duckdb_profiling_info_get_child(duckdb_profiling_info info, idx_t index);
```

Appender

```
duckdb_state duckdb_appender_create(duckdb_connection connection, const char *schema, const
char *table, duckdb_appender *out_appender);
duckdb_state duckdb_appender_create_ext(duckdb_connection connection, const char *catalog, const char
*schema, const char *table, duckdb_appender *out_appender);
idx_t duckdb_appender_column_count(duckdb_appender appender);
duckdb_logical_type duckdb_appender_column_type(duckdb_appender appender, idx_t col_idx);
const char *duckdb_appender_error(duckdb_appender appender);
duckdb_state duckdb_appender_flush(duckdb_appender appender);
duckdb_state duckdb_appender_close(duckdb_appender appender);
duckdb_state duckdb_appender_destroy(duckdb_appender *appender);
duckdb_state duckdb_appender_add_column(duckdb_appender appender, const char *name);
duckdb_state duckdb_appender_clear_columns(duckdb_appender appender);
duckdb_state duckdb_appender_begin_row(duckdb_appender appender);
duckdb_state duckdb_appender_end_row(duckdb_appender appender);
duckdb_state duckdb_append_default(duckdb_appender appender);
duckdb_state duckdb_append_default_to_chunk(duckdb_appender appender, duckdb_data_chunk chunk, idx_t
col, idx_t row);
duckdb_state duckdb_append_bool(duckdb_appender appender, bool value);
duckdb_state duckdb_append_int8(duckdb_appender appender, int8_t value);
duckdb_state duckdb_append_int16(duckdb_appender appender, int16_t value);
duckdb_state duckdb_append_int32(duckdb_appender appender, int32_t value);
duckdb_state duckdb_append_int64(duckdb_appender appender, int64_t value);
duckdb_state duckdb_append_hugeint(duckdb_appender appender, duckdb_hugeint value);
duckdb_state duckdb_append_uint8(duckdb_appender appender, uint8_t value);
duckdb_state duckdb_append_uint16(duckdb_appender appender, uint16_t value);
duckdb_state duckdb_append_uint32(duckdb_appender appender, uint32_t value);
duckdb_state duckdb_append_uint64(duckdb_appender appender, uint64_t value);
duckdb_state duckdb_append_uhugeint(duckdb_appender appender, duckdb_uhugeint value);
duckdb_state duckdb_append_float(duckdb_appender appender, float value);
duckdb_state duckdb_append_double(duckdb_appender appender, double value);
duckdb_state duckdb_append_date(duckdb_appender appender, duckdb_date value);
duckdb_state duckdb_append_time(duckdb_appender appender, duckdb_time value);
duckdb_state duckdb_append_timestamp(duckdb_appender appender, duckdb_timestamp value);
duckdb_state duckdb_append_interval(duckdb_appender appender, duckdb_interval value);
duckdb_state duckdb_append_varchar(duckdb_appender appender, const char *val);
duckdb_state duckdb_append_varchar_length(duckdb_appender appender, const char *val, idx_t length);
duckdb_state duckdb_append_blob(duckdb_appender appender, const void *data, idx_t length);
duckdb_state duckdb_append_null(duckdb_appender appender);
duckdb_state duckdb_append_value(duckdb_appender appender, duckdb_value value);
duckdb_state duckdb_append_data_chunk(duckdb_appender appender, duckdb_data_chunk chunk);
```

Table Description

```
duckdb_state duckdb_table_description_create(duckdb_connection connection, const char
*schema, const char *table, duckdb_table_description *out);
duckdb_state duckdb_table_description_create_ext(duckdb_connection connection, const char *catalog,
const char *schema, const char *table, duckdb_table_description *out);
void duckdb_table_description_destroy(duckdb_table_description *table_description);
const char *duckdb_table_description_error(duckdb_table_description table_description);
duckdb_state duckdb_column_has_default(duckdb_table_description table_description, idx_t index, bool
*out);
char *duckdb_table_description_get_column_name(duckdb_table_description table_description, idx_t index);
```

Arrow Interface

```
            duckdb_state duckdb_query_arrow(duckdb_connection connection, const char *query, duckdb_
arrow *out_result);
duckdb_state duckdb_query_arrow_schema(duckdb_arrow result, duckdb_arrow_schema *out_schema);
duckdb_state duckdb_prepared_arrow_schema(duckdb_prepared_statement prepared, duckdb_arrow_schema *out_
schema);
void duckdb_result_arrow_array(duckdb_result result, duckdb_data_chunk chunk, duckdb_arrow_array *out_
array);
duckdb_state duckdb_query_arrow_array(duckdb_arrow result, duckdb_arrow_array *out_array);
idx_t duckdb_arrow_column_count(duckdb_arrow result);
idx_t duckdb_arrow_row_count(duckdb_arrow result);
idx_t duckdb_arrow_rows_changed(duckdb_arrow result);
const char *duckdb_query_arrow_error(duckdb_arrow result);
void duckdb_destroy_arrow(duckdb_arrow *result);
void duckdb_destroy_arrow_stream(duckdb_arrow_stream *stream_p);
duckdb_state duckdb_execute_prepared_arrow(duckdb_prepared_statement prepared_statement, duckdb_arrow
*out_result);
duckdb_state duckdb_arrow_scan(duckdb_connection connection, const char *table_name, duckdb_arrow_stream
arrow);
duckdb_state duckdb_arrow_array_scan(duckdb_connection connection, const char *table_name, duckdb_arrow_
schema arrow_schema, duckdb_arrow_array arrow_array, duckdb_arrow_stream *out_stream);
```

Threading Information

```
            void duckdb_execute_tasks(duckdb_database database, idx_t max_tasks);
duckdb_task_state duckdb_create_task_state(duckdb_database database);
void duckdb_execute_tasks_state(duckdb_task_state state);
idx_t duckdb_execute_n_tasks_state(duckdb_task_state state, idx_t max_tasks);
void duckdb_finish_execution(duckdb_task_state state);
bool duckdb_task_state_is_finished(duckdb_task_state state);
void duckdb_destroy_task_state(duckdb_task_state state);
bool duckdb_execution_is_finished(duckdb_connection con);
```

Streaming Result Interface

```
            duckdb_data_chunk duckdb_stream_fetch_chunk(duckdb_result result);
duckdb_data_chunk duckdb_fetch_chunk(duckdb_result result);
```

Cast Functions

```
            duckdb_cast_function duckdb_create_cast_function();
void duckdb_cast_function_set_source_type(duckdb_cast_function cast_function, duckdb_logical_type
source_type);
void duckdb_cast_function_set_target_type(duckdb_cast_function cast_function, duckdb_logical_type
target_type);
void duckdb_cast_function_set_implicit_cast_cost(duckdb_cast_function cast_function, int64_t cost);
void duckdb_cast_function_set_function(duckdb_cast_function cast_function, duckdb_cast_function_t
function);
void duckdb_cast_function_set_extra_info(duckdb_cast_function cast_function, void *extra_info, duckdb_
delete_callback_t destroy);
void *duckdb_cast_function_get_extra_info(duckdb_function_info info);
duckdb_cast_mode duckdb_cast_function_get_cast_mode(duckdb_function_info info);
void duckdb_cast_function_set_error(duckdb_function_info info, const char *error);
void duckdb_cast_function_set_row_error(duckdb_function_info info, const char *error, idx_t row, duckdb_
vector output);
```

```
duckdb_state duckdb_register_cast_function(duckdb_connection con, duckdb_cast_function cast_function);
void duckdb_destroy_cast_function(duckdb_cast_function *cast_function);
```

duckdb_create_instance_cache

Creates a new database instance cache. The instance cache is necessary if a client/program (re)opens multiple databases to the same file within the same process. Must be destroyed with 'duckdb_destroy_instance_cache'.

Return Value The database instance cache.

Syntax

```
            duckdb_instance_cache duckdb_create_instance_cache(

);
```

duckdb_get_or_create_from_cache

Creates a new database instance in the instance cache, or retrieves an existing database instance. Must be closed with 'duckdb_close'.

Syntax

```
            duckdb_state duckdb_get_or_create_from_cache(
  duckdb_instance_cache instance_cache,
  const char *path,
  duckdb_database *out_database,
  duckdb_config config,
  char **out_error
);
```

Parameters

- `instance_cache`: The instance cache in which to create the database, or from which to take the database.
- `path`: Path to the database file on disk. Both `nullptr` and `:memory:` open or retrieve an in-memory database.
- `out_database`: The resulting cached database.
- `config`: (Optional) configuration used to create the database.
- `out_error`: If set and the function returns `DuckDBError`, this contains the error message. Note that the error message must be freed using `duckdb_free`.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

duckdb_destroy_instance_cache

Destroys an existing database instance cache and de-allocates its memory.

Syntax

```
            void duckdb_destroy_instance_cache(
  duckdb_instance_cache *instance_cache
);
```

Parameters

- `instance_cache`: The instance cache to destroy.

`duckdb_open`

Creates a new database or opens an existing database file stored at the given path. If no path is given a new in-memory database is created instead. The database must be closed with 'duckdb_close'.

Syntax

```
           duckdb_state duckdb_open(
  const char *path,
  duckdb_database *out_database
);
```

Parameters

- `path`: Path to the database file on disk. Both `nullptr` and `:memory:` open an in-memory database.
- `out_database`: The result database object.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_open_ext`

Extended version of duckdb_open. Creates a new database or opens an existing database file stored at the given path. The database must be closed with 'duckdb_close'.

Syntax

```
           duckdb_state duckdb_open_ext(
  const char *path,
  duckdb_database *out_database,
  duckdb_config config,
  char **out_error
);
```

Parameters

- `path`: Path to the database file on disk. Both `nullptr` and `:memory:` open an in-memory database.
- `out_database`: The result database object.
- `config`: (Optional) configuration used to start up the database.
- `out_error`: If set and the function returns `DuckDBError`, this contains the error message. Note that the error message must be freed using `duckdb_free`.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_close`

Closes the specified database and de-allocates all memory allocated for that database. This should be called after you are done with any database allocated through `duckdb_open` or `duckdb_open_ext`. Note that failing to call `duckdb_close` (in case of e.g., a program crash) will not cause data corruption. Still, it is recommended to always correctly close a database object after you are done with it.

Syntax

```
            void duckdb_close(
  duckdb_database *database
);
```

Parameters

- `database`: The database object to shut down.

`duckdb_connect`

Opens a connection to a database. Connections are required to query the database, and store transactional state associated with the connection. The instantiated connection should be closed using 'duckdb_disconnect'.

Syntax

```
            duckdb_state duckdb_connect(
  duckdb_database database,
  duckdb_connection *out_connection
);
```

Parameters

- `database`: The database file to connect to.
- `out_connection`: The result connection object.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_interrupt`

Interrupt running query

Syntax

```
            void duckdb_interrupt(
  duckdb_connection connection
);
```

Parameters

- `connection`: The connection to interrupt

`duckdb_query_progress`

Get progress of the running query

Syntax

```
            duckdb_query_progress_type duckdb_query_progress(
  duckdb_connection connection
);
```

Parameters

- `connection`: The working connection

Return Value -1 if no progress or a percentage of the progress

`duckdb_disconnect`

Closes the specified connection and de-allocates all memory allocated for that connection.

Syntax

```
            void duckdb_disconnect(
  duckdb_connection *connection
);
```

Parameters

- `connection`: The connection to close.

`duckdb_connection_get_client_context`

Retrieves the client context of the connection.

Syntax

```
            void duckdb_connection_get_client_context(
  duckdb_connection connection,
  duckdb_client_context *out_context
);
```

Parameters

- `connection`: The connection.
- `out_context`: The client context of the connection. Must be destroyed with `duckdb_destroy_client_context`.

`duckdb_client_context_get_connection_id`

Returns the connection id of the client context.

Syntax

```
            idx_t duckdb_client_context_get_connection_id(
  duckdb_client_context context
);
```

Parameters

- `context`: The client context.

Return Value The connection id of the client context.

`duckdb_destroy_client_context`

Destroys the client context and deallocates its memory.

Syntax

```
            void duckdb_destroy_client_context(
  duckdb_client_context *context
);
```

Parameters

- `context`: The client context to destroy.

`duckdb_library_version`

Returns the version of the linked DuckDB, with a version postfix for dev versions

Usually used for developing C extensions that must return this for a compatibility check.

Syntax

```
            const char *duckdb_library_version(

);
```

`duckdb_get_table_names`

Get the list of (fully qualified) table names of the query.

Syntax

```
            duckdb_value duckdb_get_table_names(
  duckdb_connection connection,
  const char *query,
  bool qualified
);
```

Parameters

- `connection`: The connection for which to get the table names.
- `query`: The query for which to get the table names.
- `qualified`: Returns fully qualified table names (catalog.schema.table), if set to true, else only the (not escaped) table names.

Return Value A duckdb_value of type VARCHAR[] containing the (fully qualified) table names of the query. Must be destroyed with duckdb_destroy_value.

`duckdb_create_config`

Initializes an empty configuration object that can be used to provide start-up options for the DuckDB instance through `duckdb_open_ext`. The duckdb_config must be destroyed using 'duckdb_destroy_config'

This will always succeed unless there is a malloc failure.

Note that `duckdb_destroy_config` should always be called on the resulting config, even if the function returns `DuckDBError`.

Syntax

```
            duckdb_state duckdb_create_config(
  duckdb_config *out_config
);
```

Parameters

- `out_config`: The result configuration object.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_config_count`

This returns the total amount of configuration options available for usage with `duckdb_get_config_flag`.

This should not be called in a loop as it internally loops over all the options.

Return Value The amount of config options available.

Syntax

```
            size_t duckdb_config_count(

);
```

`duckdb_get_config_flag`

Obtains a human-readable name and description of a specific configuration option. This can be used to e.g. display configuration options. This will succeed unless `index` is out of range (i.e., `>= duckdb_config_count`).

The result name or description MUST NOT be freed.

Syntax

```
            duckdb_state duckdb_get_config_flag(
  size_t index,
  const char **out_name,
  const char **out_description
);
```

Parameters

- `index`: The index of the configuration option (between 0 and `duckdb_config_count`)
- `out_name`: A name of the configuration flag.
- `out_description`: A description of the configuration flag.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

duckdb_set_config

Sets the specified option for the specified configuration. The configuration option is indicated by name. To obtain a list of config options, see `duckdb_get_config_flag`.

In the source code, configuration options are defined in `config.cpp`.

This can fail if either the name is invalid, or if the value provided for the option is invalid.

Syntax

```
            duckdb_state duckdb_set_config(
  duckdb_config config,
  const char *name,
  const char *option
);
```

Parameters

- `config`: The configuration object to set the option on.
- `name`: The name of the configuration flag to set.
- `option`: The value to set the configuration flag to.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

duckdb_destroy_config

Destroys the specified configuration object and de-allocates all memory allocated for the object.

Syntax

```
            void duckdb_destroy_config(
  duckdb_config *config
);
```

Parameters

- `config`: The configuration object to destroy.

duckdb_query

Executes a SQL query within a connection and stores the full (materialized) result in the out_result pointer. If the query fails to execute, DuckDBError is returned and the error message can be retrieved by calling `duckdb_result_error`.

Note that after running `duckdb_query`, `duckdb_destroy_result` must be called on the result object even if the query fails, otherwise the error stored within the result will not be freed correctly.

Syntax

```
            duckdb_state duckdb_query(
  duckdb_connection connection,
  const char *query,
  duckdb_result *out_result
);
```

Parameters

- `connection`: The connection to perform the query in.
- `query`: The SQL query to run.
- `out_result`: The query result.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_destroy_result`

Closes the result and de-allocates all memory allocated for that result.

Syntax

```
            void duckdb_destroy_result(
  duckdb_result *result
);
```

Parameters

- `result`: The result to destroy.

`duckdb_column_name`

Returns the column name of the specified column. The result should not need to be freed; the column names will automatically be destroyed when the result is destroyed.

Returns `NULL` if the column is out of range.

Syntax

```
            const char *duckdb_column_name(
  duckdb_result *result,
  idx_t col
);
```

Parameters

- `result`: The result object to fetch the column name from.
- `col`: The column index.

Return Value The column name of the specified column.

`duckdb_column_type`

Returns the column type of the specified column.

Returns `DUCKDB_TYPE_INVALID` if the column is out of range.

Syntax

```
            duckdb_type duckdb_column_type(
  duckdb_result *result,
  idx_t col
);
```

Parameters

- `result`: The result object to fetch the column type from.
- `col`: The column index.

Return Value The column type of the specified column.

`duckdb_result_statement_type`

Returns the statement type of the statement that was executed

Syntax

```
            duckdb_statement_type duckdb_result_statement_type(
  duckdb_result result
);
```

Parameters

- `result`: The result object to fetch the statement type from.

Return Value duckdb_statement_type value or DUCKDB_STATEMENT_TYPE_INVALID

`duckdb_column_logical_type`

Returns the logical column type of the specified column.

The return type of this call should be destroyed with `duckdb_destroy_logical_type`.

Returns `NULL` if the column is out of range.

Syntax

```
            duckdb_logical_type duckdb_column_logical_type(
  duckdb_result *result,
  idx_t col
);
```

Parameters

- `result`: The result object to fetch the column type from.
- `col`: The column index.

Return Value The logical column type of the specified column.

`duckdb_column_count`

Returns the number of columns present in a the result object.

Syntax

```
            idx_t duckdb_column_count(
  duckdb_result *result
);
```

Parameters

- `result`: The result object.

Return Value The number of columns present in the result object.

`duckdb_row_count`

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Returns the number of rows present in the result object.

Syntax

```
            idx_t duckdb_row_count(
  duckdb_result *result
);
```

Parameters

- `result`: The result object.

Return Value The number of rows present in the result object.

`duckdb_rows_changed`

Returns the number of rows changed by the query stored in the result. This is relevant only for INSERT/UPDATE/DELETE queries. For other queries the rows_changed will be 0.

Syntax

```
            idx_t duckdb_rows_changed(
  duckdb_result *result
);
```

Parameters

- `result`: The result object.

Return Value The number of rows changed.

`duckdb_column_data`

> **Deprecated.** This method has been deprecated. Prefer using `duckdb_result_get_chunk` instead.

Returns the data of a specific column of a result in columnar format.

The function returns a dense array which contains the result data. The exact type stored in the array depends on the corresponding duckdb_type (as provided by `duckdb_column_type`). For the exact type by which the data should be accessed, see the comments in the types section or the DUCKDB_TYPE enum.

For example, for a column of type DUCKDB_TYPE_INTEGER, rows can be accessed in the following manner:

```
int32_t *data = (int32_t *) duckdb_column_data(&result, 0);
printf("Data for row %d: %d\n", row, data[row]);
```

Syntax

```
            void *duckdb_column_data(
  duckdb_result *result,
  idx_t col
);
```

Parameters

- `result`: The result object to fetch the column data from.
- `col`: The column index.

Return Value The column data of the specified column.

`duckdb_nullmask_data`

> **Deprecated.** This method has been deprecated. Prefer using `duckdb_result_get_chunk` instead.

Returns the nullmask of a specific column of a result in columnar format. The nullmask indicates for every row whether or not the corresponding row is `NULL`. If a row is `NULL`, the values present in the array provided by `duckdb_column_data` are undefined.

```
int32_t *data = (int32_t *) duckdb_column_data(&result, 0);
bool *nullmask = duckdb_nullmask_data(&result, 0);
if (nullmask[row]) {
printf("Data for row %d: NULL\n", row);
} else {
printf("Data for row %d: %d\n", row, data[row]);
}
```

Syntax

```
            bool *duckdb_nullmask_data(
  duckdb_result *result,
  idx_t col
);
```

Parameters

- `result`: The result object to fetch the nullmask from.
- `col`: The column index.

Return Value The nullmask of the specified column.

`duckdb_result_error`

Returns the error message contained within the result. The error is only set if `duckdb_query` returns `DuckDBError`.

The result of this function must not be freed. It will be cleaned up when `duckdb_destroy_result` is called.

Syntax

```
            const char *duckdb_result_error(
  duckdb_result *result
);
```

Parameters

- `result`: The result object to fetch the error from.

Return Value The error of the result.

`duckdb_result_error_type`

Returns the result error type contained within the result. The error is only set if `duckdb_query` returns `DuckDBError`.

Syntax

```
            duckdb_error_type duckdb_result_error_type(
  duckdb_result *result
);
```

Parameters

- `result`: The result object to fetch the error from.

Return Value The error type of the result.

`duckdb_result_get_chunk`

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Fetches a data chunk from the duckdb_result. This function should be called repeatedly until the result is exhausted.

The result must be destroyed with `duckdb_destroy_data_chunk`.

This function supersedes all `duckdb_value` functions, as well as the `duckdb_column_data` and `duckdb_nullmask_data` functions. It results in significantly better performance, and should be preferred in newer code-bases.

If this function is used, none of the other result functions can be used and vice versa (i.e., this function cannot be mixed with the legacy result functions).

Use `duckdb_result_chunk_count` to figure out how many chunks there are in the result.

Syntax

```
            duckdb_data_chunk duckdb_result_get_chunk(
  duckdb_result result,
  idx_t chunk_index
);
```

Parameters

- `result`: The result object to fetch the data chunk from.
- `chunk_index`: The chunk index to fetch from.

Return Value The resulting data chunk. Returns `NULL` if the chunk index is out of bounds.

`duckdb_result_is_streaming`

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Checks if the type of the internal result is StreamQueryResult.

Syntax

```
            bool duckdb_result_is_streaming(
  duckdb_result result
);
```

Parameters

- `result`: The result object to check.

Return Value Whether or not the result object is of the type StreamQueryResult

`duckdb_result_chunk_count`

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Returns the number of data chunks present in the result.

Syntax

```
            idx_t duckdb_result_chunk_count(
  duckdb_result result
);
```

Parameters

- `result`: The result object

Return Value Number of data chunks present in the result.

`duckdb_result_return_type`

Returns the return_type of the given result, or DUCKDB_RETURN_TYPE_INVALID on error

Syntax

```
            duckdb_result_type duckdb_result_return_type(
  duckdb_result result
);
```

Parameters

- `result`: The result object

Return Value The return_type

`duckdb_value_boolean`

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Return Value The boolean value at the specified location, or false if the value cannot be converted.

Syntax

```
            bool duckdb_value_boolean(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

duckdb_value_int8

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Return Value The int8_t value at the specified location, or 0 if the value cannot be converted.

Syntax

```
            int8_t duckdb_value_int8(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

duckdb_value_int16

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Return Value The int16_t value at the specified location, or 0 if the value cannot be converted.

Syntax

```
            int16_t duckdb_value_int16(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

duckdb_value_int32

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Return Value The int32_t value at the specified location, or 0 if the value cannot be converted.

Syntax

```
            int32_t duckdb_value_int32(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

duckdb_value_int64

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Return Value The int64_t value at the specified location, or 0 if the value cannot be converted.

Syntax

```
            int64_t duckdb_value_int64(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

duckdb_value_hugeint

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Return Value The duckdb_hugeint value at the specified location, or 0 if the value cannot be converted.

Syntax

```
            duckdb_hugeint duckdb_value_hugeint(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

duckdb_value_uhugeint

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Return Value The duckdb_uhugeint value at the specified location, or 0 if the value cannot be converted.

Syntax

```
            duckdb_uhugeint duckdb_value_uhugeint(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

duckdb_value_decimal

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Return Value The duckdb_decimal value at the specified location, or 0 if the value cannot be converted.

Syntax

```
            duckdb_decimal duckdb_value_decimal(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

duckdb_value_uint8

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Return Value The uint8_t value at the specified location, or 0 if the value cannot be converted.

Syntax

```
            uint8_t duckdb_value_uint8(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

duckdb_value_uint16

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Return Value The uint16_t value at the specified location, or 0 if the value cannot be converted.

Syntax

```
            uint16_t duckdb_value_uint16(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

duckdb_value_uint32

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Return Value The uint32_t value at the specified location, or 0 if the value cannot be converted.

Syntax

```
            uint32_t duckdb_value_uint32(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

duckdb_value_uint64

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Return Value The uint64_t value at the specified location, or 0 if the value cannot be converted.

Syntax

```
            uint64_t duckdb_value_uint64(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

duckdb_value_float

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Return Value The float value at the specified location, or 0 if the value cannot be converted.

Syntax

```
            float duckdb_value_float(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

duckdb_value_double

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Return Value The double value at the specified location, or 0 if the value cannot be converted.

Syntax

```
            double duckdb_value_double(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

duckdb_value_date

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Return Value The duckdb_date value at the specified location, or 0 if the value cannot be converted.

Syntax

```
            duckdb_date duckdb_value_date(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

duckdb_value_time

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Return Value The duckdb_time value at the specified location, or 0 if the value cannot be converted.

Syntax

```
           duckdb_time duckdb_value_time(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

duckdb_value_timestamp

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Return Value The duckdb_timestamp value at the specified location, or 0 if the value cannot be converted.

Syntax

```
           duckdb_timestamp duckdb_value_timestamp(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

duckdb_value_interval

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Return Value The duckdb_interval value at the specified location, or 0 if the value cannot be converted.

Syntax

```
           duckdb_interval duckdb_value_interval(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

duckdb_value_varchar

> **Deprecated.** This method has been deprecated. Use duckdb_value_string instead. This function does not work correctly if the string contains null bytes.

Return Value The text value at the specified location as a null-terminated string, or nullptr if the value cannot be converted. The result must be freed with `duckdb_free`.

Syntax

```
            char *duckdb_value_varchar(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

`duckdb_value_string`

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

No support for nested types, and for other complex types. The resulting field "string.data" must be freed with `duckdb_free`.

Return Value The string value at the specified location. Attempts to cast the result value to string.

Syntax

```
            duckdb_string duckdb_value_string(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

`duckdb_value_varchar_internal`

> **Deprecated.** This method has been deprecated. Use duckdb_value_string_internal instead. This function does not work correctly if the string contains null bytes.

Return Value The char* value at the specified location. ONLY works on VARCHAR columns and does not auto-cast. If the column is NOT a VARCHAR column this function will return NULL.

The result must NOT be freed.

Syntax

```
            char *duckdb_value_varchar_internal(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

`duckdb_value_string_internal`

> **Deprecated.** This method has been deprecated. Use duckdb_value_string_internal instead. This function does not work correctly if the string contains null bytes.

Return Value The char* value at the specified location. ONLY works on VARCHAR columns and does not auto-cast. If the column is NOT a VARCHAR column this function will return NULL.

The result must NOT be freed.

Syntax

```
            duckdb_string duckdb_value_string_internal(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

`duckdb_value_blob`

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Return Value The duckdb_blob value at the specified location. Returns a blob with blob.data set to nullptr if the value cannot be converted. The resulting field "blob.data" must be freed with `duckdb_free.`

Syntax

```
            duckdb_blob duckdb_value_blob(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

`duckdb_value_is_null`

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Return Value Returns true if the value at the specified index is NULL, and false otherwise.

Syntax

```
            bool duckdb_value_is_null(
  duckdb_result *result,
  idx_t col,
  idx_t row
);
```

`duckdb_malloc`

Allocate `size` bytes of memory using the duckdb internal malloc function. Any memory allocated in this manner should be freed using `duckdb_free`.

Syntax

```
            void *duckdb_malloc(
  size_t size
);
```

Parameters

- `size`: The number of bytes to allocate.

Return Value A pointer to the allocated memory region.

`duckdb_free`

Free a value returned from `duckdb_malloc`, `duckdb_value_varchar`, `duckdb_value_blob`, or `duckdb_value_string`.

Syntax

```
            void duckdb_free(
  void *ptr
);
```

Parameters

- `ptr`: The memory region to de-allocate.

`duckdb_vector_size`

The internal vector size used by DuckDB. This is the amount of tuples that will fit into a data chunk created by `duckdb_create_data_chunk`.

Return Value The vector size.

Syntax

```
            idx_t duckdb_vector_size(

);
```

`duckdb_string_is_inlined`

Whether or not the duckdb_string_t value is inlined. This means that the data of the string does not have a separate allocation.

Syntax

```
            bool duckdb_string_is_inlined(
  duckdb_string_t string
);
```

`duckdb_string_t_length`

Get the string length of a string_t

Syntax

```
            uint32_t duckdb_string_t_length(
  duckdb_string_t string
);
```

Parameters

- `string`: The string to get the length of.

Return Value The length.

duckdb_string_t_data

Get a pointer to the string data of a string_t

Syntax

```
            const char *duckdb_string_t_data(
  duckdb_string_t *string
);
```

Parameters

- `string`: The string to get the pointer to.

Return Value The pointer.

duckdb_from_date

Decompose a `duckdb_date` object into year, month and date (stored as `duckdb_date_struct`).

Syntax

```
            duckdb_date_struct duckdb_from_date(
  duckdb_date date
);
```

Parameters

- `date`: The date object, as obtained from a `DUCKDB_TYPE_DATE` column.

Return Value The `duckdb_date_struct` with the decomposed elements.

duckdb_to_date

Re-compose a `duckdb_date` from year, month and date (`duckdb_date_struct`).

Syntax

```
            duckdb_date duckdb_to_date(
  duckdb_date_struct date
);
```

Parameters

- `date`: The year, month and date stored in a `duckdb_date_struct`.

Return Value The `duckdb_date` element.

duckdb_is_finite_date

Test a `duckdb_date` to see if it is a finite value.

Syntax

```
            bool duckdb_is_finite_date(
  duckdb_date date
);
```

Parameters

- `date`: The date object, as obtained from a `DUCKDB_TYPE_DATE` column.

Return Value True if the date is finite, false if it is ±infinity.

`duckdb_from_time`

Decompose a `duckdb_time` object into hour, minute, second and microsecond (stored as `duckdb_time_struct`).

Syntax

```
            duckdb_time_struct duckdb_from_time(
  duckdb_time time
);
```

Parameters

- `time`: The time object, as obtained from a `DUCKDB_TYPE_TIME` column.

Return Value The `duckdb_time_struct` with the decomposed elements.

`duckdb_create_time_tz`

Create a `duckdb_time_tz` object from micros and a timezone offset.

Syntax

```
            duckdb_time_tz duckdb_create_time_tz(
  int64_t micros,
  int32_t offset
);
```

Parameters

- `micros`: The microsecond component of the time.
- `offset`: The timezone offset component of the time.

Return Value The `duckdb_time_tz` element.

`duckdb_from_time_tz`

Decompose a TIME_TZ objects into micros and a timezone offset.

Use `duckdb_from_time` to further decompose the micros into hour, minute, second and microsecond.

Syntax

```
            duckdb_time_tz_struct duckdb_from_time_tz(
  duckdb_time_tz micros
);
```

Parameters

- `micros`: The time object, as obtained from a `DUCKDB_TYPE_TIME_TZ` column.

`duckdb_to_time`

Re-compose a `duckdb_time` from hour, minute, second and microsecond (`duckdb_time_struct`).

Syntax

```
            duckdb_time duckdb_to_time(
  duckdb_time_struct time
);
```

Parameters

- `time`: The hour, minute, second and microsecond in a `duckdb_time_struct`.

Return Value The `duckdb_time` element.

`duckdb_from_timestamp`

Decompose a `duckdb_timestamp` object into a `duckdb_timestamp_struct`.

Syntax

```
            duckdb_timestamp_struct duckdb_from_timestamp(
  duckdb_timestamp ts
);
```

Parameters

- `ts`: The ts object, as obtained from a `DUCKDB_TYPE_TIMESTAMP` column.

Return Value The `duckdb_timestamp_struct` with the decomposed elements.

`duckdb_to_timestamp`

Re-compose a `duckdb_timestamp` from a duckdb_timestamp_struct.

Syntax

```
            duckdb_timestamp duckdb_to_timestamp(
  duckdb_timestamp_struct ts
);
```

Parameters

- `ts`: The de-composed elements in a `duckdb_timestamp_struct`.

Return Value The `duckdb_timestamp` element.

`duckdb_is_finite_timestamp`

Test a `duckdb_timestamp` to see if it is a finite value.

Syntax

```
            bool duckdb_is_finite_timestamp(
  duckdb_timestamp ts
);
```

Parameters

- `ts`: The duckdb_timestamp object, as obtained from a `DUCKDB_TYPE_TIMESTAMP` column.

Return Value True if the timestamp is finite, false if it is ±infinity.

`duckdb_is_finite_timestamp_s`

Test a `duckdb_timestamp_s` to see if it is a finite value.

Syntax

```
            bool duckdb_is_finite_timestamp_s(
  duckdb_timestamp_s ts
);
```

Parameters

- `ts`: The duckdb_timestamp_s object, as obtained from a `DUCKDB_TYPE_TIMESTAMP_S` column.

Return Value True if the timestamp is finite, false if it is ±infinity.

`duckdb_is_finite_timestamp_ms`

Test a `duckdb_timestamp_ms` to see if it is a finite value.

Syntax

```
            bool duckdb_is_finite_timestamp_ms(
  duckdb_timestamp_ms ts
);
```

Parameters

- `ts`: The duckdb_timestamp_ms object, as obtained from a `DUCKDB_TYPE_TIMESTAMP_MS` column.

Return Value True if the timestamp is finite, false if it is ±infinity.

`duckdb_is_finite_timestamp_ns`

Test a `duckdb_timestamp_ns` to see if it is a finite value.

Syntax

```
            bool duckdb_is_finite_timestamp_ns(
  duckdb_timestamp_ns ts
);
```

Parameters

- `ts`: The duckdb_timestamp_ns object, as obtained from a `DUCKDB_TYPE_TIMESTAMP_NS` column.

Return Value True if the timestamp is finite, false if it is ±infinity.

`duckdb_hugeint_to_double`

Converts a duckdb_hugeint object (as obtained from a `DUCKDB_TYPE_HUGEINT` column) into a double.

Syntax

```
            double duckdb_hugeint_to_double(
  duckdb_hugeint val
);
```

Parameters

- `val`: The hugeint value.

Return Value The converted `double` element.

`duckdb_double_to_hugeint`

Converts a double value to a duckdb_hugeint object.

If the conversion fails because the double value is too big the result will be 0.

Syntax

```
            duckdb_hugeint duckdb_double_to_hugeint(
  double val
);
```

Parameters

- `val`: The double value.

Return Value The converted `duckdb_hugeint` element.

`duckdb_uhugeint_to_double`

Converts a duckdb_uhugeint object (as obtained from a `DUCKDB_TYPE_UHUGEINT` column) into a double.

Syntax

```
            double duckdb_uhugeint_to_double(
  duckdb_uhugeint val
);
```

Parameters

- `val`: The uhugeint value.

Return Value The converted `double` element.

`duckdb_double_to_uhugeint`

Converts a double value to a duckdb_uhugeint object.

If the conversion fails because the double value is too big the result will be 0.

Syntax

```
            duckdb_uhugeint duckdb_double_to_uhugeint(
  double val
);
```

Parameters

- `val`: The double value.

Return Value The converted `duckdb_uhugeint` element.

`duckdb_double_to_decimal`

Converts a double value to a duckdb_decimal object.

If the conversion fails because the double value is too big, or the width/scale are invalid the result will be 0.

Syntax

```
            duckdb_decimal duckdb_double_to_decimal(
  double val,
  uint8_t width,
  uint8_t scale
);
```

Parameters

- `val`: The double value.

Return Value The converted `duckdb_decimal` element.

duckdb_decimal_to_double

Converts a duckdb_decimal object (as obtained from a DUCKDB_TYPE_DECIMAL column) into a double.

Syntax

```
            double duckdb_decimal_to_double(
  duckdb_decimal val
);
```

Parameters

- `val`: The decimal value.

Return Value The converted `double` element.

duckdb_prepare

Create a prepared statement object from a query.

Note that after calling `duckdb_prepare`, the prepared statement should always be destroyed using `duckdb_destroy_prepare`, even if the prepare fails.

If the prepare fails, `duckdb_prepare_error` can be called to obtain the reason why the prepare failed.

Syntax

```
            duckdb_state duckdb_prepare(
  duckdb_connection connection,
  const char *query,
  duckdb_prepared_statement *out_prepared_statement
);
```

Parameters

- `connection`: The connection object
- `query`: The SQL query to prepare
- `out_prepared_statement`: The resulting prepared statement object

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

duckdb_destroy_prepare

Closes the prepared statement and de-allocates all memory allocated for the statement.

Syntax

```
            void duckdb_destroy_prepare(
  duckdb_prepared_statement *prepared_statement
);
```

Parameters

- `prepared_statement`: The prepared statement to destroy.

`duckdb_prepare_error`

Returns the error message associated with the given prepared statement. If the prepared statement has no error message, this returns `nullptr` instead.

The error message should not be freed. It will be de-allocated when `duckdb_destroy_prepare` is called.

Syntax

```
            const char *duckdb_prepare_error(
  duckdb_prepared_statement prepared_statement
);
```

Parameters

- `prepared_statement`: The prepared statement to obtain the error from.

Return Value The error message, or `nullptr` if there is none.

`duckdb_nparams`

Returns the number of parameters that can be provided to the given prepared statement.

Returns 0 if the query was not successfully prepared.

Syntax

```
            idx_t duckdb_nparams(
  duckdb_prepared_statement prepared_statement
);
```

Parameters

- `prepared_statement`: The prepared statement to obtain the number of parameters for.

`duckdb_parameter_name`

Returns the name used to identify the parameter The returned string should be freed using `duckdb_free`.

Returns NULL if the index is out of range for the provided prepared statement.

Syntax

```
            const char *duckdb_parameter_name(
  duckdb_prepared_statement prepared_statement,
  idx_t index
);
```

Parameters

- `prepared_statement`: The prepared statement for which to get the parameter name from.

`duckdb_param_type`

Returns the parameter type for the parameter at the given index.

Returns `DUCKDB_TYPE_INVALID` if the parameter index is out of range or the statement was not successfully prepared.

Syntax

```
           duckdb_type duckdb_param_type(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx
);
```

Parameters

- `prepared_statement`: The prepared statement.
- `param_idx`: The parameter index.

Return Value The parameter type

`duckdb_param_logical_type`

Returns the logical type for the parameter at the given index.

Returns `nullptr` if the parameter index is out of range or the statement was not successfully prepared.

The return type of this call should be destroyed with `duckdb_destroy_logical_type`.

Syntax

```
           duckdb_logical_type duckdb_param_logical_type(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx
);
```

Parameters

- `prepared_statement`: The prepared statement.
- `param_idx`: The parameter index.

Return Value The logical type of the parameter

`duckdb_clear_bindings`

Clear the params bind to the prepared statement.

Syntax

```
           duckdb_state duckdb_clear_bindings(
  duckdb_prepared_statement prepared_statement
);
```

`duckdb_prepared_statement_type`

Returns the statement type of the statement to be executed

Syntax

```
            duckdb_statement_type duckdb_prepared_statement_type(
  duckdb_prepared_statement statement
);
```

Parameters

- `statement`: The prepared statement.

Return Value duckdb_statement_type value or DUCKDB_STATEMENT_TYPE_INVALID

`duckdb_bind_value`

Binds a value to the prepared statement at the specified index.

Syntax

```
            duckdb_state duckdb_bind_value(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  duckdb_value val
);
```

`duckdb_bind_parameter_index`

Retrieve the index of the parameter for the prepared statement, identified by name

Syntax

```
            duckdb_state duckdb_bind_parameter_index(
  duckdb_prepared_statement prepared_statement,
  idx_t *param_idx_out,
  const char *name
);
```

`duckdb_bind_boolean`

Binds a bool value to the prepared statement at the specified index.

Syntax

```
            duckdb_state duckdb_bind_boolean(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  bool val
);
```

duckdb_bind_int8

Binds an int8_t value to the prepared statement at the specified index.

Syntax

```
            duckdb_state duckdb_bind_int8(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  int8_t val
);
```

duckdb_bind_int16

Binds an int16_t value to the prepared statement at the specified index.

Syntax

```
            duckdb_state duckdb_bind_int16(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  int16_t val
);
```

duckdb_bind_int32

Binds an int32_t value to the prepared statement at the specified index.

Syntax

```
            duckdb_state duckdb_bind_int32(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  int32_t val
);
```

duckdb_bind_int64

Binds an int64_t value to the prepared statement at the specified index.

Syntax

```
            duckdb_state duckdb_bind_int64(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  int64_t val
);
```

duckdb_bind_hugeint

Binds a duckdb_hugeint value to the prepared statement at the specified index.

Syntax

```
            duckdb_state duckdb_bind_hugeint(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  duckdb_hugeint val
);
```

duckdb_bind_uhugeint

Binds a duckdb_uhugeint value to the prepared statement at the specified index.

Syntax

```
            duckdb_state duckdb_bind_uhugeint(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  duckdb_uhugeint val
);
```

duckdb_bind_decimal

Binds a duckdb_decimal value to the prepared statement at the specified index.

Syntax

```
            duckdb_state duckdb_bind_decimal(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  duckdb_decimal val
);
```

duckdb_bind_uint8

Binds a uint8_t value to the prepared statement at the specified index.

Syntax

```
            duckdb_state duckdb_bind_uint8(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  uint8_t val
);
```

duckdb_bind_uint16

Binds a uint16_t value to the prepared statement at the specified index.

Syntax

```
            duckdb_state duckdb_bind_uint16(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  uint16_t val
);
```

duckdb_bind_uint32

Binds a uint32_t value to the prepared statement at the specified index.

Syntax

```
            duckdb_state duckdb_bind_uint32(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  uint32_t val
);
```

duckdb_bind_uint64

Binds a uint64_t value to the prepared statement at the specified index.

Syntax

```
            duckdb_state duckdb_bind_uint64(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  uint64_t val
);
```

duckdb_bind_float

Binds a float value to the prepared statement at the specified index.

Syntax

```
            duckdb_state duckdb_bind_float(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  float val
);
```

duckdb_bind_double

Binds a double value to the prepared statement at the specified index.

Syntax

```
            duckdb_state duckdb_bind_double(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  double val
);
```

duckdb_bind_date

Binds a duckdb_date value to the prepared statement at the specified index.

Syntax

```
            duckdb_state duckdb_bind_date(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  duckdb_date val
);
```

`duckdb_bind_time`

Binds a duckdb_time value to the prepared statement at the specified index.

Syntax

```
            duckdb_state duckdb_bind_time(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  duckdb_time val
);
```

`duckdb_bind_timestamp`

Binds a duckdb_timestamp value to the prepared statement at the specified index.

Syntax

```
            duckdb_state duckdb_bind_timestamp(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  duckdb_timestamp val
);
```

`duckdb_bind_timestamp_tz`

Binds a duckdb_timestamp value to the prepared statement at the specified index.

Syntax

```
            duckdb_state duckdb_bind_timestamp_tz(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  duckdb_timestamp val
);
```

`duckdb_bind_interval`

Binds a duckdb_interval value to the prepared statement at the specified index.

Syntax

```
            duckdb_state duckdb_bind_interval(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  duckdb_interval val
);
```

duckdb_bind_varchar

Binds a null-terminated varchar value to the prepared statement at the specified index.

Syntax

```
            duckdb_state duckdb_bind_varchar(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  const char *val
);
```

duckdb_bind_varchar_length

Binds a varchar value to the prepared statement at the specified index.

Syntax

```
            duckdb_state duckdb_bind_varchar_length(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  const char *val,
  idx_t length
);
```

duckdb_bind_blob

Binds a blob value to the prepared statement at the specified index.

Syntax

```
            duckdb_state duckdb_bind_blob(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx,
  const void *data,
  idx_t length
);
```

duckdb_bind_null

Binds a NULL value to the prepared statement at the specified index.

Syntax

```
            duckdb_state duckdb_bind_null(
  duckdb_prepared_statement prepared_statement,
  idx_t param_idx
);
```

duckdb_execute_prepared

Executes the prepared statement with the given bound parameters, and returns a materialized query result.

This method can be called multiple times for each prepared statement, and the parameters can be modified between calls to this function.

Note that the result must be freed with `duckdb_destroy_result`.

Syntax

```
           duckdb_state duckdb_execute_prepared(
  duckdb_prepared_statement prepared_statement,
  duckdb_result *out_result
);
```

Parameters

- `prepared_statement`: The prepared statement to execute.
- `out_result`: The query result.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

duckdb_execute_prepared_streaming

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Executes the prepared statement with the given bound parameters, and returns an optionally-streaming query result. To determine if the resulting query was in fact streamed, use `duckdb_result_is_streaming`

This method can be called multiple times for each prepared statement, and the parameters can be modified between calls to this function.

Note that the result must be freed with `duckdb_destroy_result`.

Syntax

```
           duckdb_state duckdb_execute_prepared_streaming(
  duckdb_prepared_statement prepared_statement,
  duckdb_result *out_result
);
```

Parameters

- `prepared_statement`: The prepared statement to execute.
- `out_result`: The query result.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

duckdb_extract_statements

Extract all statements from a query. Note that after calling `duckdb_extract_statements`, the extracted statements should always be destroyed using `duckdb_destroy_extracted`, even if no statements were extracted.

If the extract fails, `duckdb_extract_statements_error` can be called to obtain the reason why the extract failed.

Syntax

```
            idx_t duckdb_extract_statements(
  duckdb_connection connection,
  const char *query,
  duckdb_extracted_statements *out_extracted_statements
);
```

Parameters

- `connection`: The connection object
- `query`: The SQL query to extract
- `out_extracted_statements`: The resulting extracted statements object

Return Value The number of extracted statements or 0 on failure.

`duckdb_prepare_extracted_statement`

Prepare an extracted statement. Note that after calling `duckdb_prepare_extracted_statement`, the prepared statement should always be destroyed using `duckdb_destroy_prepare`, even if the prepare fails.

If the prepare fails, `duckdb_prepare_error` can be called to obtain the reason why the prepare failed.

Syntax

```
            duckdb_state duckdb_prepare_extracted_statement(
  duckdb_connection connection,
  duckdb_extracted_statements extracted_statements,
  idx_t index,
  duckdb_prepared_statement *out_prepared_statement
);
```

Parameters

- `connection`: The connection object
- `extracted_statements`: The extracted statements object
- `index`: The index of the extracted statement to prepare
- `out_prepared_statement`: The resulting prepared statement object

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_extract_statements_error`

Returns the error message contained within the extracted statements. The result of this function must not be freed. It will be cleaned up when `duckdb_destroy_extracted` is called.

Syntax

```
            const char *duckdb_extract_statements_error(
  duckdb_extracted_statements extracted_statements
);
```

Parameters

- `extracted_statements`: The extracted statements to fetch the error from.

Return Value The error of the extracted statements.

`duckdb_destroy_extracted`

De-allocates all memory allocated for the extracted statements.

Syntax

```
            void duckdb_destroy_extracted(
  duckdb_extracted_statements *extracted_statements
);
```

Parameters

- `extracted_statements`: The extracted statements to destroy.

`duckdb_pending_prepared`

Executes the prepared statement with the given bound parameters, and returns a pending result. The pending result represents an intermediate structure for a query that is not yet fully executed. The pending result can be used to incrementally execute a query, returning control to the client between tasks.

Note that after calling `duckdb_pending_prepared`, the pending result should always be destroyed using `duckdb_destroy_pending`, even if this function returns DuckDBError.

Syntax

```
            duckdb_state duckdb_pending_prepared(
  duckdb_prepared_statement prepared_statement,
  duckdb_pending_result *out_result
);
```

Parameters

- `prepared_statement`: The prepared statement to execute.
- `out_result`: The pending query result.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_pending_prepared_streaming`

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Executes the prepared statement with the given bound parameters, and returns a pending result. This pending result will create a streaming duckdb_result when executed. The pending result represents an intermediate structure for a query that is not yet fully executed.

Note that after calling `duckdb_pending_prepared_streaming`, the pending result should always be destroyed using `duckdb_destroy_pending`, even if this function returns DuckDBError.

Syntax

```
            duckdb_state duckdb_pending_prepared_streaming(
  duckdb_prepared_statement prepared_statement,
  duckdb_pending_result *out_result
);
```

Parameters

- `prepared_statement`: The prepared statement to execute.
- `out_result`: The pending query result.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_destroy_pending`

Closes the pending result and de-allocates all memory allocated for the result.

Syntax

```
            void duckdb_destroy_pending(
  duckdb_pending_result *pending_result
);
```

Parameters

- `pending_result`: The pending result to destroy.

`duckdb_pending_error`

Returns the error message contained within the pending result.

The result of this function must not be freed. It will be cleaned up when `duckdb_destroy_pending` is called.

Syntax

```
            const char *duckdb_pending_error(
  duckdb_pending_result pending_result
);
```

Parameters

- `pending_result`: The pending result to fetch the error from.

Return Value The error of the pending result.

`duckdb_pending_execute_task`

Executes a single task within the query, returning whether or not the query is ready.

If this returns DUCKDB_PENDING_RESULT_READY, the duckdb_execute_pending function can be called to obtain the result. If this returns DUCKDB_PENDING_RESULT_NOT_READY, the duckdb_pending_execute_task function should be called again. If this returns DUCKDB_PENDING_ERROR, an error occurred during execution.

The error message can be obtained by calling duckdb_pending_error on the pending_result.

Syntax

```
            duckdb_pending_state duckdb_pending_execute_task(
  duckdb_pending_result pending_result
);
```

Parameters

- `pending_result`: The pending result to execute a task within.

Return Value The state of the pending result after the execution.

`duckdb_pending_execute_check_state`

If this returns DUCKDB_PENDING_RESULT_READY, the duckdb_execute_pending function can be called to obtain the result. If this returns DUCKDB_PENDING_RESULT_NOT_READY, the duckdb_pending_execute_check_state function should be called again. If this returns DUCKDB_PENDING_ERROR, an error occurred during execution.

The error message can be obtained by calling duckdb_pending_error on the pending_result.

Syntax

```
            duckdb_pending_state duckdb_pending_execute_check_state(
  duckdb_pending_result pending_result
);
```

Parameters

- `pending_result`: The pending result.

Return Value The state of the pending result.

`duckdb_execute_pending`

Fully execute a pending query result, returning the final query result.

If duckdb_pending_execute_task has been called until DUCKDB_PENDING_RESULT_READY was returned, this will return fast. Otherwise, all remaining tasks must be executed first.

Note that the result must be freed with `duckdb_destroy_result`.

Syntax

```
            duckdb_state duckdb_execute_pending(
  duckdb_pending_result pending_result,
  duckdb_result *out_result
);
```

Parameters

- `pending_result`: The pending result to execute.
- `out_result`: The result object.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

duckdb_pending_execution_is_finished

Returns whether a duckdb_pending_state is finished executing. For example if `pending_state` is DUCKDB_PENDING_RESULT_READY, this function will return true.

Syntax

```
            bool duckdb_pending_execution_is_finished(
  duckdb_pending_state pending_state
);
```

Parameters

- `pending_state`: The pending state on which to decide whether to finish execution.

Return Value Boolean indicating pending execution should be considered finished.

duckdb_destroy_value

Destroys the value and de-allocates all memory allocated for that type.

Syntax

```
            void duckdb_destroy_value(
  duckdb_value *value
);
```

Parameters

- `value`: The value to destroy.

duckdb_create_varchar

Creates a value from a null-terminated string

Syntax

```
            duckdb_value duckdb_create_varchar(
  const char *text
);
```

Parameters

- `text`: The null-terminated string

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

duckdb_create_varchar_length

Creates a value from a string

Syntax

```
            duckdb_value duckdb_create_varchar_length(
  const char *text,
  idx_t length
);
```

Parameters

- `text`: The text
- `length`: The length of the text

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_bool`

Creates a value from a boolean

Syntax

```
            duckdb_value duckdb_create_bool(
  bool input
);
```

Parameters

- `input`: The boolean value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_int8`

Creates a value from an int8_t (a tinyint)

Syntax

```
            duckdb_value duckdb_create_int8(
  int8_t input
);
```

Parameters

- `input`: The tinyint value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_uint8`

Creates a value from a uint8_t (a utinyint)

Syntax

```
            duckdb_value duckdb_create_uint8(
  uint8_t input
);
```

Parameters

- `input`: The utinyint value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_int16`

Creates a value from an int16_t (a smallint)

Syntax

```
            duckdb_value duckdb_create_int16(
  int16_t input
);
```

Parameters

- `input`: The smallint value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_uint16`

Creates a value from a uint16_t (a usmallint)

Syntax

```
            duckdb_value duckdb_create_uint16(
  uint16_t input
);
```

Parameters

- `input`: The usmallint value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_int32`

Creates a value from an int32_t (an integer)

Syntax

```
            duckdb_value duckdb_create_int32(
  int32_t input
);
```

Parameters

- `input`: The integer value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_uint32`

Creates a value from a uint32_t (a uinteger)

Syntax

```
            duckdb_value duckdb_create_uint32(
  uint32_t input
);
```

Parameters

- `input`: The uinteger value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_uint64`

Creates a value from a uint64_t (a ubigint)

Syntax

```
            duckdb_value duckdb_create_uint64(
  uint64_t input
);
```

Parameters

- `input`: The ubigint value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_int64`

Creates a value from an int64

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

Syntax

```
            duckdb_value duckdb_create_int64(
  int64_t val
);
```

`duckdb_create_hugeint`

Creates a value from a hugeint

Syntax

```
            duckdb_value duckdb_create_hugeint(
  duckdb_hugeint input
);
```

Parameters

- `input`: The hugeint value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_uhugeint`

Creates a value from a uhugeint

Syntax

```
            duckdb_value duckdb_create_uhugeint(
  duckdb_uhugeint input
);
```

Parameters

- `input`: The uhugeint value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_varint`

Creates a VARINT value from a duckdb_varint

Syntax

```
            duckdb_value duckdb_create_varint(
  duckdb_varint input
);
```

Parameters

- `input`: The duckdb_varint value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_decimal`

Creates a DECIMAL value from a duckdb_decimal

Syntax

```
            duckdb_value duckdb_create_decimal(
  duckdb_decimal input
);
```

Parameters

- `input`: The duckdb_decimal value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_float`

Creates a value from a float

Syntax

```
            duckdb_value duckdb_create_float(
  float input
);
```

Parameters

- `input`: The float value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_double`

Creates a value from a double

Syntax

```
            duckdb_value duckdb_create_double(
  double input
);
```

Parameters

- `input`: The double value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_date`

Creates a value from a date

Syntax

```
            duckdb_value duckdb_create_date(
  duckdb_date input
);
```

Parameters

- `input`: The date value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_time`

Creates a value from a time

Syntax

```
            duckdb_value duckdb_create_time(
  duckdb_time input
);
```

Parameters

- `input`: The time value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_time_tz_value`

Creates a value from a time_tz. Not to be confused with `duckdb_create_time_tz`, which creates a duckdb_time_tz_t.

Syntax

```
            duckdb_value duckdb_create_time_tz_value(
  duckdb_time_tz value
);
```

Parameters

- `value`: The time_tz value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_timestamp`

Creates a TIMESTAMP value from a duckdb_timestamp

Syntax

```
            duckdb_value duckdb_create_timestamp(
  duckdb_timestamp input
);
```

Parameters

- `input`: The duckdb_timestamp value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_timestamp_tz`

Creates a TIMESTAMP_TZ value from a duckdb_timestamp

Syntax

```
            duckdb_value duckdb_create_timestamp_tz(
  duckdb_timestamp input
);
```

Parameters

- `input`: The duckdb_timestamp value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_timestamp_s`

Creates a TIMESTAMP_S value from a duckdb_timestamp_s

Syntax

```
            duckdb_value duckdb_create_timestamp_s(
  duckdb_timestamp_s input
);
```

Parameters

- `input`: The duckdb_timestamp_s value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_timestamp_ms`

Creates a TIMESTAMP_MS value from a duckdb_timestamp_ms

Syntax

```
            duckdb_value duckdb_create_timestamp_ms(
  duckdb_timestamp_ms input
);
```

Parameters

- `input`: The duckdb_timestamp_ms value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_timestamp_ns`

Creates a TIMESTAMP_NS value from a duckdb_timestamp_ns

Syntax

```
            duckdb_value duckdb_create_timestamp_ns(
  duckdb_timestamp_ns input
);
```

Parameters

- `input`: The duckdb_timestamp_ns value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_interval`

Creates a value from an interval

Syntax

```
            duckdb_value duckdb_create_interval(
  duckdb_interval input
);
```

Parameters

- `input`: The interval value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

`duckdb_create_blob`

Creates a value from a blob

Syntax

```
            duckdb_value duckdb_create_blob(
  const uint8_t *data,
  idx_t length
);
```

Parameters

- `data`: The blob data
- `length`: The length of the blob data

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

duckdb_create_bit

Creates a BIT value from a duckdb_bit

Syntax

```
            duckdb_value duckdb_create_bit(
  duckdb_bit input
);
```

Parameters

- `input`: The duckdb_bit value

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

duckdb_create_uuid

Creates a UUID value from a uhugeint

Syntax

```
            duckdb_value duckdb_create_uuid(
  duckdb_uhugeint input
);
```

Parameters

- `input`: The duckdb_uhugeint containing the UUID

Return Value The value. This must be destroyed with `duckdb_destroy_value`.

duckdb_get_bool

Returns the boolean value of the given value.

Syntax

```
            bool duckdb_get_bool(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a boolean

Return Value A boolean, or false if the value cannot be converted

duckdb_get_int8

Returns the int8_t value of the given value.

Syntax

```
            int8_t duckdb_get_int8(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a tinyint

Return Value A int8_t, or MinValue if the value cannot be converted

`duckdb_get_uint8`

Returns the uint8_t value of the given value.

Syntax

```
            uint8_t duckdb_get_uint8(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a utinyint

Return Value A uint8_t, or MinValue if the value cannot be converted

`duckdb_get_int16`

Returns the int16_t value of the given value.

Syntax

```
            int16_t duckdb_get_int16(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a smallint

Return Value A int16_t, or MinValue if the value cannot be converted

`duckdb_get_uint16`

Returns the uint16_t value of the given value.

Syntax

```
            uint16_t duckdb_get_uint16(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a usmallint

Return Value A uint16_t, or MinValue if the value cannot be converted

`duckdb_get_int32`

Returns the int32_t value of the given value.

Syntax

```
            int32_t duckdb_get_int32(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing an integer

Return Value A int32_t, or MinValue if the value cannot be converted

`duckdb_get_uint32`

Returns the uint32_t value of the given value.

Syntax

```
            uint32_t duckdb_get_uint32(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a uinteger

Return Value A uint32_t, or MinValue if the value cannot be converted

`duckdb_get_int64`

Returns the int64_t value of the given value.

Syntax

```
            int64_t duckdb_get_int64(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a bigint

Return Value A int64_t, or MinValue if the value cannot be converted

`duckdb_get_uint64`

Returns the uint64_t value of the given value.

Syntax

```
            uint64_t duckdb_get_uint64(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a ubigint

Return Value A uint64_t, or MinValue if the value cannot be converted

`duckdb_get_hugeint`

Returns the hugeint value of the given value.

Syntax

```
            duckdb_hugeint duckdb_get_hugeint(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a hugeint

Return Value A duckdb_hugeint, or MinValue if the value cannot be converted

`duckdb_get_uhugeint`

Returns the uhugeint value of the given value.

Syntax

```
            duckdb_uhugeint duckdb_get_uhugeint(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a uhugeint

Return Value A duckdb_uhugeint, or MinValue if the value cannot be converted

`duckdb_get_varint`

Returns the duckdb_varint value of the given value. The `data` field must be destroyed with `duckdb_free`.

Syntax

```
            duckdb_varint duckdb_get_varint(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a VARINT

Return Value A duckdb_varint. The `data` field must be destroyed with `duckdb_free`.

`duckdb_get_decimal`

Returns the duckdb_decimal value of the given value.

Syntax

```
            duckdb_decimal duckdb_get_decimal(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a DECIMAL

Return Value A duckdb_decimal, or MinValue if the value cannot be converted

`duckdb_get_float`

Returns the float value of the given value.

Syntax

```
            float duckdb_get_float(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a float

Return Value A float, or NAN if the value cannot be converted

`duckdb_get_double`

Returns the double value of the given value.

Syntax

```
            double duckdb_get_double(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a double

Return Value A double, or NAN if the value cannot be converted

`duckdb_get_date`

Returns the date value of the given value.

Syntax

```
            duckdb_date duckdb_get_date(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a date

Return Value A duckdb_date, or MinValue if the value cannot be converted

`duckdb_get_time`

Returns the time value of the given value.

Syntax

```
            duckdb_time duckdb_get_time(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a time

Return Value A duckdb_time, or MinValue if the value cannot be converted

`duckdb_get_time_tz`

Returns the time_tz value of the given value.

Syntax

```
            duckdb_time_tz duckdb_get_time_tz(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a time_tz

Return Value A duckdb_time_tz, or MinValue if the value cannot be converted

`duckdb_get_timestamp`

Returns the TIMESTAMP value of the given value.

Syntax

```
            duckdb_timestamp duckdb_get_timestamp(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a TIMESTAMP

Return Value A duckdb_timestamp, or MinValue if the value cannot be converted

`duckdb_get_timestamp_tz`

Returns the TIMESTAMP_TZ value of the given value.

Syntax

```
            duckdb_timestamp duckdb_get_timestamp_tz(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a TIMESTAMP_TZ

Return Value A duckdb_timestamp, or MinValue if the value cannot be converted

`duckdb_get_timestamp_s`

Returns the duckdb_timestamp_s value of the given value.

Syntax

```
            duckdb_timestamp_s duckdb_get_timestamp_s(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a TIMESTAMP_S

Return Value A duckdb_timestamp_s, or MinValue if the value cannot be converted

`duckdb_get_timestamp_ms`

Returns the duckdb_timestamp_ms value of the given value.

Syntax

```
            duckdb_timestamp_ms duckdb_get_timestamp_ms(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a TIMESTAMP_MS

Return Value A duckdb_timestamp_ms, or MinValue if the value cannot be converted

`duckdb_get_timestamp_ns`

Returns the duckdb_timestamp_ns value of the given value.

Syntax

```
            duckdb_timestamp_ns duckdb_get_timestamp_ns(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a TIMESTAMP_NS

Return Value A duckdb_timestamp_ns, or MinValue if the value cannot be converted

`duckdb_get_interval`

Returns the interval value of the given value.

Syntax

```
            duckdb_interval duckdb_get_interval(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a interval

Return Value A duckdb_interval, or MinValue if the value cannot be converted

`duckdb_get_value_type`

Returns the type of the given value. The type is valid as long as the value is not destroyed. The type itself must not be destroyed.

Syntax

```
            duckdb_logical_type duckdb_get_value_type(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value

Return Value A duckdb_logical_type.

`duckdb_get_blob`

Returns the blob value of the given value.

Syntax

```
            duckdb_blob duckdb_get_blob(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a blob

Return Value A duckdb_blob

`duckdb_get_bit`

Returns the duckdb_bit value of the given value. The `data` field must be destroyed with `duckdb_free`.

Syntax

```
            duckdb_bit duckdb_get_bit(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a BIT

Return Value A duckdb_bit

`duckdb_get_uuid`

Returns a duckdb_uhugeint representing the UUID value of the given value.

Syntax

```
            duckdb_uhugeint duckdb_get_uuid(
  duckdb_value val
);
```

Parameters

- `val`: A duckdb_value containing a UUID

Return Value A duckdb_uhugeint representing the UUID value

`duckdb_get_varchar`

Obtains a string representation of the given value. The result must be destroyed with `duckdb_free`.

Syntax

```
            char *duckdb_get_varchar(
  duckdb_value value
);
```

Parameters

- `value`: The value

Return Value The string value. This must be destroyed with `duckdb_free`.

`duckdb_create_struct_value`

Creates a struct value from a type and an array of values. Must be destroyed with `duckdb_destroy_value`.

Syntax

```
            duckdb_value duckdb_create_struct_value(
  duckdb_logical_type type,
  duckdb_value *values
);
```

Parameters

- `type`: The type of the struct
- `values`: The values for the struct fields

Return Value The struct value, or nullptr, if any child type is `DUCKDB_TYPE_ANY` or `DUCKDB_TYPE_INVALID`.

`duckdb_create_list_value`

Creates a list value from a child (element) type and an array of values of length `value_count`. Must be destroyed with `duckdb_destroy_value`.

Syntax

```
            duckdb_value duckdb_create_list_value(
  duckdb_logical_type type,
  duckdb_value *values,
  idx_t value_count
);
```

Parameters

- `type`: The type of the list
- `values`: The values for the list
- `value_count`: The number of values in the list

Return Value The list value, or nullptr, if the child type is `DUCKDB_TYPE_ANY` or `DUCKDB_TYPE_INVALID`.

`duckdb_create_array_value`

Creates an array value from a child (element) type and an array of values of length `value_count`. Must be destroyed with `duckdb_destroy_value`.

Syntax

```
            duckdb_value duckdb_create_array_value(
  duckdb_logical_type type,
  duckdb_value *values,
  idx_t value_count
);
```

Parameters

- `type`: The type of the array
- `values`: The values for the array
- `value_count`: The number of values in the array

Return Value The array value, or nullptr, if the child type is `DUCKDB_TYPE_ANY` or `DUCKDB_TYPE_INVALID`.

`duckdb_create_map_value`

Creates a map value from a map type and two arrays, one for the keys and one for the values, each of length `entry_count`. Must be destroyed with `duckdb_destroy_value`.

Syntax

```
            duckdb_value duckdb_create_map_value(
  duckdb_logical_type map_type,
  duckdb_value *keys,
  duckdb_value *values,
  idx_t entry_count
);
```

Parameters

- `map_type`: The map type
- `keys`: The keys of the map
- `values`: The values of the map
- `entry_count`: The number of entrys (key-value pairs) in the map

Return Value The map value, or nullptr, if the parameters are invalid.

`duckdb_create_union_value`

Creates a union value from a union type, a tag index, and a value. Must be destroyed with `duckdb_destroy_value`.

Syntax

```
duckdb_value duckdb_create_union_value(
  duckdb_logical_type union_type,
  idx_t tag_index,
  duckdb_value value
);
```

Parameters

- `union_type`: The union type
- `tag_index`: The index of the tag of the union
- `value`: The value of the union

Return Value The union value, or nullptr, if the parameters are invalid.

`duckdb_get_map_size`

Returns the number of elements in a MAP value.

Syntax

```
idx_t duckdb_get_map_size(
  duckdb_value value
);
```

Parameters

- `value`: The MAP value.

Return Value The number of elements in the map.

`duckdb_get_map_key`

Returns the MAP key at index as a duckdb_value.

Syntax

```
            duckdb_value duckdb_get_map_key(
  duckdb_value value,
  idx_t index
);
```

Parameters

- `value`: The MAP value.
- `index`: The index of the key.

Return Value The key as a duckdb_value.

`duckdb_get_map_value`

Returns the MAP value at index as a duckdb_value.

Syntax

```
            duckdb_value duckdb_get_map_value(
  duckdb_value value,
  idx_t index
);
```

Parameters

- `value`: The MAP value.
- `index`: The index of the value.

Return Value The value as a duckdb_value.

`duckdb_is_null_value`

Returns whether the value's type is SQLNULL or not.

Syntax

```
            bool duckdb_is_null_value(
  duckdb_value value
);
```

Parameters

- `value`: The value to check.

Return Value True, if the value's type is SQLNULL, otherwise false.

`duckdb_create_null_value`

Creates a value of type SQLNULL.

Return Value The duckdb_value representing SQLNULL. This must be destroyed with `duckdb_destroy_value`.

Syntax

```
duckdb_value duckdb_create_null_value(

);
```

`duckdb_get_list_size`

Returns the number of elements in a LIST value.

Syntax

```
idx_t duckdb_get_list_size(
  duckdb_value value
);
```

Parameters

- `value`: The LIST value.

Return Value The number of elements in the list.

`duckdb_get_list_child`

Returns the LIST child at index as a duckdb_value.

Syntax

```
duckdb_value duckdb_get_list_child(
  duckdb_value value,
  idx_t index
);
```

Parameters

- `value`: The LIST value.
- `index`: The index of the child.

Return Value The child as a duckdb_value.

`duckdb_create_enum_value`

Creates an enum value from a type and a value. Must be destroyed with `duckdb_destroy_value`.

Syntax

```
duckdb_value duckdb_create_enum_value(
  duckdb_logical_type type,
  uint64_t value
);
```

Parameters

- `type`: The type of the enum
- `value`: The value for the enum

Return Value The enum value, or nullptr.

`duckdb_get_enum_value`

Returns the enum value of the given value.

Syntax

```
            uint64_t duckdb_get_enum_value(
  duckdb_value value
);
```

Parameters

- `value`: A duckdb_value containing an enum

Return Value A uint64_t, or MinValue if the value cannot be converted

`duckdb_get_struct_child`

Returns the STRUCT child at index as a duckdb_value.

Syntax

```
            duckdb_value duckdb_get_struct_child(
  duckdb_value value,
  idx_t index
);
```

Parameters

- `value`: The STRUCT value.
- `index`: The index of the child.

Return Value The child as a duckdb_value.

`duckdb_value_to_string`

Returns the SQL string representation of the given value.

Syntax

```
            char *duckdb_value_to_string(
  duckdb_value value
);
```

Parameters

- `value`: A duckdb_value.

Return Value The SQL string representation as a null-terminated string. The result must be freed with `duckdb_free`.

`duckdb_create_logical_type`

Creates a `duckdb_logical_type` from a primitive type. The resulting logical type must be destroyed with `duckdb_destroy_logical_type`.

Returns an invalid logical type, if type is: `DUCKDB_TYPE_INVALID`, `DUCKDB_TYPE_DECIMAL`, `DUCKDB_TYPE_ENUM`, `DUCKDB_TYPE_LIST`, `DUCKDB_TYPE_STRUCT`, `DUCKDB_TYPE_MAP`, `DUCKDB_TYPE_ARRAY`, or `DUCKDB_TYPE_UNION`.

Syntax

```
            duckdb_logical_type duckdb_create_logical_type(
  duckdb_type type
);
```

Parameters

- `type`: The primitive type to create.

Return Value The logical type.

`duckdb_logical_type_get_alias`

Returns the alias of a duckdb_logical_type, if set, else `nullptr`. The result must be destroyed with `duckdb_free`.

Syntax

```
            char *duckdb_logical_type_get_alias(
  duckdb_logical_type type
);
```

Parameters

- `type`: The logical type

Return Value The alias or `nullptr`

`duckdb_logical_type_set_alias`

Sets the alias of a duckdb_logical_type.

Syntax

```
            void duckdb_logical_type_set_alias(
  duckdb_logical_type type,
  const char *alias
);
```

Parameters

- `type`: The logical type
- `alias`: The alias to set

`duckdb_create_list_type`

Creates a LIST type from its child type. The return type must be destroyed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_create_list_type(
  duckdb_logical_type type
);
```

Parameters

- `type`: The child type of the list

Return Value The logical type.

`duckdb_create_array_type`

Creates an ARRAY type from its child type. The return type must be destroyed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_create_array_type(
  duckdb_logical_type type,
  idx_t array_size
);
```

Parameters

- `type`: The child type of the array.
- `array_size`: The number of elements in the array.

Return Value The logical type.

`duckdb_create_map_type`

Creates a MAP type from its key type and value type. The return type must be destroyed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_create_map_type(
  duckdb_logical_type key_type,
  duckdb_logical_type value_type
);
```

Parameters

- `key_type`: The map's key type.
- `value_type`: The map's value type.

Return Value The logical type.

`duckdb_create_union_type`

Creates a UNION type from the passed arrays. The return type must be destroyed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_create_union_type(
  duckdb_logical_type *member_types,
  const char **member_names,
  idx_t member_count
);
```

Parameters

- `member_types`: The array of union member types.
- `member_names`: The union member names.
- `member_count`: The number of union members.

Return Value The logical type.

`duckdb_create_struct_type`

Creates a STRUCT type based on the member types and names. The resulting type must be destroyed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_create_struct_type(
  duckdb_logical_type *member_types,
  const char **member_names,
  idx_t member_count
);
```

Parameters

- `member_types`: The array of types of the struct members.
- `member_names`: The array of names of the struct members.
- `member_count`: The number of members of the struct.

Return Value The logical type.

`duckdb_create_enum_type`

Creates an ENUM type from the passed member name array. The resulting type should be destroyed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_create_enum_type(
  const char **member_names,
  idx_t member_count
);
```

Parameters

- `member_names`: The array of names that the enum should consist of.
- `member_count`: The number of elements that were specified in the array.

Return Value The logical type.

`duckdb_create_decimal_type`

Creates a DECIMAL type with the specified width and scale. The resulting type should be destroyed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_create_decimal_type(
  uint8_t width,
  uint8_t scale
);
```

Parameters

- `width`: The width of the decimal type
- `scale`: The scale of the decimal type

Return Value The logical type.

`duckdb_get_type_id`

Retrieves the enum `duckdb_type` of a `duckdb_logical_type`.

Syntax

```
            duckdb_type duckdb_get_type_id(
  duckdb_logical_type type
);
```

Parameters

- `type`: The logical type.

Return Value The `duckdb_type` id.

`duckdb_decimal_width`

Retrieves the width of a decimal type.

Syntax

```
            uint8_t duckdb_decimal_width(
  duckdb_logical_type type
);
```

Parameters

- `type`: The logical type object

Return Value The width of the decimal type

`duckdb_decimal_scale`

Retrieves the scale of a decimal type.

Syntax

```
            uint8_t duckdb_decimal_scale(
  duckdb_logical_type type
);
```

Parameters

- `type`: The logical type object

Return Value The scale of the decimal type

`duckdb_decimal_internal_type`

Retrieves the internal storage type of a decimal type.

Syntax

```
            duckdb_type duckdb_decimal_internal_type(
  duckdb_logical_type type
);
```

Parameters

- `type`: The logical type object

Return Value The internal type of the decimal type

`duckdb_enum_internal_type`

Retrieves the internal storage type of an enum type.

Syntax

```
            duckdb_type duckdb_enum_internal_type(
  duckdb_logical_type type
);
```

Parameters

- `type`: The logical type object

Return Value The internal type of the enum type

`duckdb_enum_dictionary_size`

Retrieves the dictionary size of the enum type.

Syntax

```
            uint32_t duckdb_enum_dictionary_size(
  duckdb_logical_type type
);
```

Parameters

- `type`: The logical type object

Return Value The dictionary size of the enum type

`duckdb_enum_dictionary_value`

Retrieves the dictionary value at the specified position from the enum.

The result must be freed with `duckdb_free`.

Syntax

```
            char *duckdb_enum_dictionary_value(
  duckdb_logical_type type,
  idx_t index
);
```

Parameters

- `type`: The logical type object
- `index`: The index in the dictionary

Return Value The string value of the enum type. Must be freed with `duckdb_free`.

`duckdb_list_type_child_type`

Retrieves the child type of the given LIST type. Also accepts MAP types. The result must be freed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_list_type_child_type(
  duckdb_logical_type type
);
```

Parameters

- `type`: The logical type, either LIST or MAP.

Return Value The child type of the LIST or MAP type.

`duckdb_array_type_child_type`

Retrieves the child type of the given ARRAY type.

The result must be freed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_array_type_child_type(
  duckdb_logical_type type
);
```

Parameters

- `type`: The logical type. Must be ARRAY.

Return Value The child type of the ARRAY type.

`duckdb_array_type_array_size`

Retrieves the array size of the given array type.

Syntax

```
            idx_t duckdb_array_type_array_size(
  duckdb_logical_type type
);
```

Parameters

- `type`: The logical type object

Return Value The fixed number of elements the values of this array type can store.

`duckdb_map_type_key_type`

Retrieves the key type of the given map type.

The result must be freed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_map_type_key_type(
  duckdb_logical_type type
);
```

Parameters

- `type`: The logical type object

Return Value The key type of the map type. Must be destroyed with `duckdb_destroy_logical_type`.

`duckdb_map_type_value_type`

Retrieves the value type of the given map type.

The result must be freed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_map_type_value_type(
  duckdb_logical_type type
);
```

Parameters

- `type`: The logical type object

Return Value The value type of the map type. Must be destroyed with `duckdb_destroy_logical_type`.

`duckdb_struct_type_child_count`

Returns the number of children of a struct type.

Syntax

```
            idx_t duckdb_struct_type_child_count(
  duckdb_logical_type type
);
```

Parameters

- `type`: The logical type object

Return Value The number of children of a struct type.

`duckdb_struct_type_child_name`

Retrieves the name of the struct child.

The result must be freed with `duckdb_free`.

Syntax

```
            char *duckdb_struct_type_child_name(
  duckdb_logical_type type,
  idx_t index
);
```

Parameters

- `type`: The logical type object
- `index`: The child index

Return Value The name of the struct type. Must be freed with `duckdb_free`.

`duckdb_struct_type_child_type`

Retrieves the child type of the given struct type at the specified index.

The result must be freed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_struct_type_child_type(
  duckdb_logical_type type,
  idx_t index
);
```

Parameters

- `type`: The logical type object
- `index`: The child index

Return Value The child type of the struct type. Must be destroyed with `duckdb_destroy_logical_type`.

`duckdb_union_type_member_count`

Returns the number of members that the union type has.

Syntax

```
            idx_t duckdb_union_type_member_count(
  duckdb_logical_type type
);
```

Parameters

- `type`: The logical type (union) object

Return Value The number of members of a union type.

`duckdb_union_type_member_name`

Retrieves the name of the union member.

The result must be freed with `duckdb_free`.

Syntax

```
            char *duckdb_union_type_member_name(
  duckdb_logical_type type,
  idx_t index
);
```

Parameters

- `type`: The logical type object
- `index`: The child index

Return Value The name of the union member. Must be freed with `duckdb_free`.

`duckdb_union_type_member_type`

Retrieves the child type of the given union member at the specified index.

The result must be freed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_union_type_member_type(
  duckdb_logical_type type,
  idx_t index
);
```

Parameters

- `type`: The logical type object
- `index`: The child index

Return Value The child type of the union member. Must be destroyed with `duckdb_destroy_logical_type`.

`duckdb_destroy_logical_type`

Destroys the logical type and de-allocates all memory allocated for that type.

Syntax

```
            void duckdb_destroy_logical_type(
  duckdb_logical_type *type
);
```

Parameters

- `type`: The logical type to destroy.

`duckdb_register_logical_type`

Registers a custom type within the given connection. The type must have an alias

Syntax

```
            duckdb_state duckdb_register_logical_type(
  duckdb_connection con,
  duckdb_logical_type type,
  duckdb_create_type_info info
);
```

Parameters

- `con`: The connection to use
- `type`: The custom type to register

Return Value Whether or not the registration was successful.

`duckdb_create_data_chunk`

Creates an empty data chunk with the specified column types. The result must be destroyed with `duckdb_destroy_data_chunk`.

Syntax

```
            duckdb_data_chunk duckdb_create_data_chunk(
  duckdb_logical_type *types,
  idx_t column_count
);
```

Parameters

- `types`: An array of column types. Column types can not contain ANY and INVALID types.
- `column_count`: The number of columns.

Return Value The data chunk.

`duckdb_destroy_data_chunk`

Destroys the data chunk and de-allocates all memory allocated for that chunk.

Syntax

```
            void duckdb_destroy_data_chunk(
  duckdb_data_chunk *chunk
);
```

Parameters

- `chunk`: The data chunk to destroy.

`duckdb_data_chunk_reset`

Resets a data chunk, clearing the validity masks and setting the cardinality of the data chunk to 0. After calling this method, you must call `duckdb_vector_get_validity` and `duckdb_vector_get_data` to obtain current data and validity pointers

Syntax

```
            void duckdb_data_chunk_reset(
  duckdb_data_chunk chunk
);
```

Parameters

- chunk: The data chunk to reset.

duckdb_data_chunk_get_column_count

Retrieves the number of columns in a data chunk.

Syntax

```
            idx_t duckdb_data_chunk_get_column_count(
  duckdb_data_chunk chunk
);
```

Parameters

- chunk: The data chunk to get the data from

Return Value The number of columns in the data chunk

duckdb_data_chunk_get_vector

Retrieves the vector at the specified column index in the data chunk.

The pointer to the vector is valid for as long as the chunk is alive. It does NOT need to be destroyed.

Syntax

```
            duckdb_vector duckdb_data_chunk_get_vector(
  duckdb_data_chunk chunk,
  idx_t col_idx
);
```

Parameters

- chunk: The data chunk to get the data from

Return Value The vector

duckdb_data_chunk_get_size

Retrieves the current number of tuples in a data chunk.

Syntax

```
            idx_t duckdb_data_chunk_get_size(
  duckdb_data_chunk chunk
);
```

Parameters

- chunk: The data chunk to get the data from

Return Value The number of tuples in the data chunk

duckdb_data_chunk_set_size

Sets the current number of tuples in a data chunk.

Syntax

```
            void duckdb_data_chunk_set_size(
  duckdb_data_chunk chunk,
  idx_t size
);
```

Parameters

- chunk: The data chunk to set the size in
- size: The number of tuples in the data chunk

duckdb_create_vector

Creates a flat vector.

Syntax

```
            duckdb_vector duckdb_create_vector(
  duckdb_logical_type type,
  idx_t capacity
);
```

duckdb_destroy_vector

Destroys the vector and de-allocates all memory allocated for that vector, if unused else where.

Syntax

```
            void duckdb_destroy_vector(
  duckdb_vector *vector
);
```

duckdb_vector_get_column_type

Retrieves the column type of the specified vector.

The result must be destroyed with duckdb_destroy_logical_type.

Syntax

```
            duckdb_logical_type duckdb_vector_get_column_type(
  duckdb_vector vector
);
```

Parameters

- `vector`: The vector get the data from

Return Value The type of the vector

`duckdb_vector_get_data`

Retrieves the data pointer of the vector.

The data pointer can be used to read or write values from the vector. How to read or write values depends on the type of the vector.

Syntax

```
            void *duckdb_vector_get_data(
  duckdb_vector vector
);
```

Parameters

- `vector`: The vector to get the data from

Return Value The data pointer

`duckdb_vector_get_validity`

Retrieves the validity mask pointer of the specified vector.

If all values are valid, this function MIGHT return NULL!

The validity mask is a bitset that signifies null-ness within the data chunk. It is a series of uint64_t values, where each uint64_t value contains validity for 64 tuples. The bit is set to 1 if the value is valid (i.e., not NULL) or 0 if the value is invalid (i.e., NULL).

Validity of a specific value can be obtained like this:

idx_t entry_idx = row_idx / 64; idx_t idx_in_entry = row_idx % 64; bool is_valid = validity_mask[entry_idx] & (1 « idx_in_entry);

Alternatively, the (slower) duckdb_validity_row_is_valid function can be used.

Syntax

```
            uint64_t *duckdb_vector_get_validity(
  duckdb_vector vector
);
```

Parameters

- `vector`: The vector to get the data from

Return Value The pointer to the validity mask, or NULL if no validity mask is present

`duckdb_vector_ensure_validity_writable`

Ensures the validity mask is writable by allocating it.

After this function is called, `duckdb_vector_get_validity` will ALWAYS return non-NULL. This allows NULL values to be written to the vector, regardless of whether a validity mask was present before.

Syntax

```
            void duckdb_vector_ensure_validity_writable(
  duckdb_vector vector
);
```

Parameters

- `vector`: The vector to alter

`duckdb_vector_assign_string_element`

Assigns a string element in the vector at the specified location.

Syntax

```
            void duckdb_vector_assign_string_element(
  duckdb_vector vector,
  idx_t index,
  const char *str
);
```

Parameters

- `vector`: The vector to alter
- `index`: The row position in the vector to assign the string to
- `str`: The null-terminated string

`duckdb_vector_assign_string_element_len`

Assigns a string element in the vector at the specified location. You may also use this function to assign BLOBs.

Syntax

```
            void duckdb_vector_assign_string_element_len(
  duckdb_vector vector,
  idx_t index,
  const char *str,
  idx_t str_len
);
```

Parameters

- `vector`: The vector to alter
- `index`: The row position in the vector to assign the string to
- `str`: The string
- `str_len`: The length of the string (in bytes)

`duckdb_list_vector_get_child`

Retrieves the child vector of a list vector.

The resulting vector is valid as long as the parent vector is valid.

Syntax

```
            duckdb_vector duckdb_list_vector_get_child(
  duckdb_vector vector
);
```

Parameters

- `vector`: The vector

Return Value The child vector

`duckdb_list_vector_get_size`

Returns the size of the child vector of the list.

Syntax

```
            idx_t duckdb_list_vector_get_size(
  duckdb_vector vector
);
```

Parameters

- `vector`: The vector

Return Value The size of the child list

`duckdb_list_vector_set_size`

Sets the total size of the underlying child-vector of a list vector.

Syntax

```
            duckdb_state duckdb_list_vector_set_size(
  duckdb_vector vector,
  idx_t size
);
```

Parameters

- `vector`: The list vector.
- `size`: The size of the child list.

Return Value The duckdb state. Returns DuckDBError if the vector is nullptr.

`duckdb_list_vector_reserve`

Sets the total capacity of the underlying child-vector of a list.

After calling this method, you must call `duckdb_vector_get_validity` and `duckdb_vector_get_data` to obtain current data and validity pointers

Syntax

```
            duckdb_state duckdb_list_vector_reserve(
  duckdb_vector vector,
  idx_t required_capacity
);
```

Parameters

- `vector`: The list vector.
- `required_capacity`: the total capacity to reserve.

Return Value The duckdb state. Returns DuckDBError if the vector is nullptr.

`duckdb_struct_vector_get_child`

Retrieves the child vector of a struct vector.

The resulting vector is valid as long as the parent vector is valid.

Syntax

```
            duckdb_vector duckdb_struct_vector_get_child(
  duckdb_vector vector,
  idx_t index
);
```

Parameters

- `vector`: The vector
- `index`: The child index

Return Value The child vector

`duckdb_array_vector_get_child`

Retrieves the child vector of an array vector.

The resulting vector is valid as long as the parent vector is valid. The resulting vector has the size of the parent vector multiplied by the array size.

Syntax

```
            duckdb_vector duckdb_array_vector_get_child(
  duckdb_vector vector
);
```

Parameters

- `vector`: The vector

Return Value The child vector

`duckdb_slice_vector`

Slice a vector with a selection vector.

The max value in the selection vector must be less than the length of the vector

The resulting vector happens to be a dictionary vector.

Syntax

```
            void duckdb_slice_vector(
  duckdb_vector vector,
  duckdb_selection_vector selection,
  idx_t len
);
```

Parameters

- `vector`: The vector which is to become a dictionary
- `selection`: The selection vector
- `len`: The length of the selection vector

`duckdb_vector_reference_value`

Copies the value from `value` to `vector`.

Syntax

```
            void duckdb_vector_reference_value(
  duckdb_vector vector,
  duckdb_value value
);
```

`duckdb_vector_reference_vector`

References the `from` vector in the `to` vector, this makes take shared ownership of the values buffer

Syntax

```
            void duckdb_vector_reference_vector(
  duckdb_vector to_vector,
  duckdb_vector from_vector
);
```

`duckdb_validity_row_is_valid`

Returns whether or not a row is valid (i.e., not NULL) in the given validity mask.

Syntax

```
            bool duckdb_validity_row_is_valid(
  uint64_t *validity,
  idx_t row
);
```

Parameters

- `validity`: The validity mask, as obtained through `duckdb_vector_get_validity`
- `row`: The row index

Return Value true if the row is valid, false otherwise

`duckdb_validity_set_row_validity`

In a validity mask, sets a specific row to either valid or invalid.

Note that `duckdb_vector_ensure_validity_writable` should be called before calling `duckdb_vector_get_validity`, to ensure that there is a validity mask to write to.

Syntax

```
            void duckdb_validity_set_row_validity(
  uint64_t *validity,
  idx_t row,
  bool valid
);
```

Parameters

- `validity`: The validity mask, as obtained through `duckdb_vector_get_validity`.
- `row`: The row index
- `valid`: Whether or not to set the row to valid, or invalid

`duckdb_validity_set_row_invalid`

In a validity mask, sets a specific row to invalid.

Equivalent to `duckdb_validity_set_row_validity` with valid set to false.

Syntax

```
            void duckdb_validity_set_row_invalid(
  uint64_t *validity,
  idx_t row
);
```

Parameters

- `validity`: The validity mask
- `row`: The row index

`duckdb_validity_set_row_valid`

In a validity mask, sets a specific row to valid.

Equivalent to `duckdb_validity_set_row_validity` with valid set to true.

Syntax

```
            void duckdb_validity_set_row_valid(
  uint64_t *validity,
  idx_t row
);
```

Parameters

- `validity`: The validity mask
- `row`: The row index

`duckdb_create_scalar_function`

Creates a new empty scalar function.

The return value must be destroyed with `duckdb_destroy_scalar_function`.

Return Value The scalar function object.

Syntax

```
            duckdb_scalar_function duckdb_create_scalar_function(

);
```

`duckdb_destroy_scalar_function`

Destroys the given scalar function object.

Syntax

```
            void duckdb_destroy_scalar_function(
  duckdb_scalar_function *scalar_function
);
```

Parameters

- `scalar_function`: The scalar function to destroy

`duckdb_scalar_function_set_name`

Sets the name of the given scalar function.

Syntax

```
            void duckdb_scalar_function_set_name(
  duckdb_scalar_function scalar_function,
  const char *name
);
```

Parameters

- `scalar_function`: The scalar function
- `name`: The name of the scalar function

duckdb_scalar_function_set_varargs

Sets the parameters of the given scalar function to varargs. Does not require adding parameters with duckdb_scalar_function_add_parameter.

Syntax

```
            void duckdb_scalar_function_set_varargs(
  duckdb_scalar_function scalar_function,
  duckdb_logical_type type
);
```

Parameters

- `scalar_function`: The scalar function.
- `type`: The type of the arguments.

Return Value The parameter type. Cannot contain INVALID.

duckdb_scalar_function_set_special_handling

Sets the scalar function's null-handling behavior to special.

Syntax

```
            void duckdb_scalar_function_set_special_handling(
  duckdb_scalar_function scalar_function
);
```

Parameters

- `scalar_function`: The scalar function.

duckdb_scalar_function_set_volatile

Sets the Function Stability of the scalar function to VOLATILE, indicating the function should be re-run for every row. This limits optimization that can be performed for the function.

Syntax

```
            void duckdb_scalar_function_set_volatile(
  duckdb_scalar_function scalar_function
);
```

Parameters

- `scalar_function`: The scalar function.

duckdb_scalar_function_add_parameter

Adds a parameter to the scalar function.

Syntax

```
            void duckdb_scalar_function_add_parameter(
  duckdb_scalar_function scalar_function,
  duckdb_logical_type type
);
```

Parameters

- `scalar_function`: The scalar function.
- `type`: The parameter type. Cannot contain INVALID.

`duckdb_scalar_function_set_return_type`

Sets the return type of the scalar function.

Syntax

```
            void duckdb_scalar_function_set_return_type(
  duckdb_scalar_function scalar_function,
  duckdb_logical_type type
);
```

Parameters

- `scalar_function`: The scalar function
- `type`: Cannot contain INVALID or ANY.

`duckdb_scalar_function_set_extra_info`

Assigns extra information to the scalar function that can be fetched during binding, etc.

Syntax

```
            void duckdb_scalar_function_set_extra_info(
  duckdb_scalar_function scalar_function,
  void *extra_info,
  duckdb_delete_callback_t destroy
);
```

Parameters

- `scalar_function`: The scalar function
- `extra_info`: The extra information
- `destroy`: The callback that will be called to destroy the extra information (if any)

`duckdb_scalar_function_set_bind`

Sets the (optional) bind function of the scalar function.

Syntax

```
            void duckdb_scalar_function_set_bind(
  duckdb_scalar_function scalar_function,
  duckdb_scalar_function_bind_t bind
);
```

Parameters

- `scalar_function`: The scalar function
- `bind`: The bind function

`duckdb_scalar_function_set_bind_data`

Sets the user-provided bind data in the bind object of the scalar function. This object can be retrieved again during execution.

Syntax

```
            void duckdb_scalar_function_set_bind_data(
  duckdb_bind_info info,
  void *bind_data,
  duckdb_delete_callback_t destroy
);
```

Parameters

- `info`: The bind info of the scalar function.
- `bind_data`: The bind data object.
- `destroy`: The callback to destroy the bind data (if any).

`duckdb_scalar_function_bind_set_error`

Report that an error has occurred while calling bind on a scalar function.

Syntax

```
            void duckdb_scalar_function_bind_set_error(
  duckdb_bind_info info,
  const char *error
);
```

Parameters

- `info`: The bind info object
- `error`: The error message

`duckdb_scalar_function_set_function`

Sets the main function of the scalar function.

Syntax

```
            void duckdb_scalar_function_set_function(
  duckdb_scalar_function scalar_function,
  duckdb_scalar_function_t function
);
```

Parameters

- `scalar_function`: The scalar function
- `function`: The function

`duckdb_register_scalar_function`

Register the scalar function object within the given connection.

The function requires at least a name, a function and a return type.

If the function is incomplete or a function with this name already exists DuckDBError is returned.

Syntax

```
            duckdb_state duckdb_register_scalar_function(
  duckdb_connection con,
  duckdb_scalar_function scalar_function
);
```

Parameters

- `con`: The connection to register it in.
- `scalar_function`: The function pointer

Return Value Whether or not the registration was successful.

`duckdb_scalar_function_get_extra_info`

Retrieves the extra info of the function as set in `duckdb_scalar_function_set_extra_info`.

Syntax

```
            void *duckdb_scalar_function_get_extra_info(
  duckdb_function_info info
);
```

Parameters

- `info`: The info object.

Return Value The extra info.

`duckdb_scalar_function_get_bind_data`

Gets the scalar function's bind data set by `duckdb_scalar_function_set_bind_data`.

Note that the bind data is read-only.

Syntax

```
            void *duckdb_scalar_function_get_bind_data(
  duckdb_function_info info
);
```

Parameters

- `info`: The function info.

Return Value The bind data object.

`duckdb_scalar_function_get_client_context`

Retrieves the client context of the bind info of a scalar function.

Syntax

```
            void duckdb_scalar_function_get_client_context(
  duckdb_bind_info info,
  duckdb_client_context *out_context
);
```

Parameters

- `info`: The bind info object of the scalar function.
- `out_context`: The client context of the bind info. Must be destroyed with `duckdb_destroy_client_context`.

`duckdb_scalar_function_set_error`

Report that an error has occurred while executing the scalar function.

Syntax

```
            void duckdb_scalar_function_set_error(
  duckdb_function_info info,
  const char *error
);
```

Parameters

- `info`: The info object.
- `error`: The error message

`duckdb_create_scalar_function_set`

Creates a new empty scalar function set.

The return value must be destroyed with `duckdb_destroy_scalar_function_set`.

Return Value The scalar function set object.

Syntax

```
            duckdb_scalar_function_set duckdb_create_scalar_function_set(
  const char *name
);
```

duckdb_destroy_scalar_function_set

Destroys the given scalar function set object.

Syntax

```
            void duckdb_destroy_scalar_function_set(
  duckdb_scalar_function_set *scalar_function_set
);
```

duckdb_add_scalar_function_to_set

Adds the scalar function as a new overload to the scalar function set.

Returns DuckDBError if the function could not be added, for example if the overload already exists.

Syntax

```
            duckdb_state duckdb_add_scalar_function_to_set(
  duckdb_scalar_function_set set,
  duckdb_scalar_function function
);
```

Parameters

- `set`: The scalar function set
- `function`: The function to add

duckdb_register_scalar_function_set

Register the scalar function set within the given connection.

The set requires at least a single valid overload.

If the set is incomplete or a function with this name already exists DuckDBError is returned.

Syntax

```
            duckdb_state duckdb_register_scalar_function_set(
  duckdb_connection con,
  duckdb_scalar_function_set set
);
```

Parameters

- `con`: The connection to register it in.
- `set`: The function set to register

Return Value Whether or not the registration was successful.

duckdb_create_selection_vector

Creates a new selection vector of size `size`.

Syntax

```
            duckdb_selection_vector duckdb_create_selection_vector(
  idx_t size
);
```

duckdb_destroy_selection_vector

Destroys a selection vector.

Syntax

```
            void duckdb_destroy_selection_vector(
  duckdb_selection_vector vector
);
```

duckdb_selection_vector_get_data_ptr

Access the data pointer of a selection vector.

Syntax

```
            sel_t *duckdb_selection_vector_get_data_ptr(
  duckdb_selection_vector vector
);
```

duckdb_create_aggregate_function

Creates a new empty aggregate function.

The return value should be destroyed with `duckdb_destroy_aggregate_function`.

Return Value The aggregate function object.

Syntax

```
            duckdb_aggregate_function duckdb_create_aggregate_function(

);
```

duckdb_destroy_aggregate_function

Destroys the given aggregate function object.

Syntax

```
            void duckdb_destroy_aggregate_function(
  duckdb_aggregate_function *aggregate_function
);
```

duckdb_aggregate_function_set_name

Sets the name of the given aggregate function.

Syntax

```
            void duckdb_aggregate_function_set_name(
  duckdb_aggregate_function aggregate_function,
  const char *name
);
```

Parameters

- aggregate_function: The aggregate function
- name: The name of the aggregate function

duckdb_aggregate_function_add_parameter

Adds a parameter to the aggregate function.

Syntax

```
            void duckdb_aggregate_function_add_parameter(
  duckdb_aggregate_function aggregate_function,
  duckdb_logical_type type
);
```

Parameters

- aggregate_function: The aggregate function.
- type: The parameter type. Cannot contain INVALID.

duckdb_aggregate_function_set_return_type

Sets the return type of the aggregate function.

Syntax

```
            void duckdb_aggregate_function_set_return_type(
  duckdb_aggregate_function aggregate_function,
  duckdb_logical_type type
);
```

Parameters

- aggregate_function: The aggregate function.
- type: The return type. Cannot contain INVALID or ANY.

duckdb_aggregate_function_set_functions

Sets the main functions of the aggregate function.

Syntax

```
            void duckdb_aggregate_function_set_functions(
  duckdb_aggregate_function aggregate_function,
  duckdb_aggregate_state_size state_size,
  duckdb_aggregate_init_t state_init,
  duckdb_aggregate_update_t update,
  duckdb_aggregate_combine_t combine,
  duckdb_aggregate_finalize_t finalize
);
```

Parameters

- `aggregate_function`: The aggregate function
- `state_size`: state size
- `state_init`: state init function
- `update`: update states
- `combine`: combine states
- `finalize`: finalize states

`duckdb_aggregate_function_set_destructor`

Sets the state destructor callback of the aggregate function (optional)

Syntax

```
            void duckdb_aggregate_function_set_destructor(
  duckdb_aggregate_function aggregate_function,
  duckdb_aggregate_destroy_t destroy
);
```

Parameters

- `aggregate_function`: The aggregate function
- `destroy`: state destroy callback

`duckdb_register_aggregate_function`

Register the aggregate function object within the given connection.

The function requires at least a name, functions and a return type.

If the function is incomplete or a function with this name already exists DuckDBError is returned.

Syntax

```
            duckdb_state duckdb_register_aggregate_function(
  duckdb_connection con,
  duckdb_aggregate_function aggregate_function
);
```

Parameters

- `con`: The connection to register it in.

Return Value Whether or not the registration was successful.

`duckdb_aggregate_function_set_special_handling`

Sets the NULL handling of the aggregate function to SPECIAL_HANDLING.

Syntax

```
            void duckdb_aggregate_function_set_special_handling(
  duckdb_aggregate_function aggregate_function
);
```

Parameters

- `aggregate_function`: The aggregate function

`duckdb_aggregate_function_set_extra_info`

Assigns extra information to the scalar function that can be fetched during binding, etc.

Syntax

```
            void duckdb_aggregate_function_set_extra_info(
  duckdb_aggregate_function aggregate_function,
  void *extra_info,
  duckdb_delete_callback_t destroy
);
```

Parameters

- `aggregate_function`: The aggregate function
- `extra_info`: The extra information
- `destroy`: The callback that will be called to destroy the extra information (if any)

`duckdb_aggregate_function_get_extra_info`

Retrieves the extra info of the function as set in `duckdb_aggregate_function_set_extra_info`.

Syntax

```
            void *duckdb_aggregate_function_get_extra_info(
  duckdb_function_info info
);
```

Parameters

- `info`: The info object

Return Value The extra info

`duckdb_aggregate_function_set_error`

Report that an error has occurred while executing the aggregate function.

Syntax

```
            void duckdb_aggregate_function_set_error(
  duckdb_function_info info,
  const char *error
);
```

Parameters

- `info`: The info object
- `error`: The error message

`duckdb_create_aggregate_function_set`

Creates a new empty aggregate function set.

The return value should be destroyed with `duckdb_destroy_aggregate_function_set`.

Return Value The aggregate function set object.

Syntax

```
            duckdb_aggregate_function_set duckdb_create_aggregate_function_set(
  const char *name
);
```

`duckdb_destroy_aggregate_function_set`

Destroys the given aggregate function set object.

Syntax

```
            void duckdb_destroy_aggregate_function_set(
  duckdb_aggregate_function_set *aggregate_function_set
);
```

`duckdb_add_aggregate_function_to_set`

Adds the aggregate function as a new overload to the aggregate function set.

Returns DuckDBError if the function could not be added, for example if the overload already exists.

Syntax

```
            duckdb_state duckdb_add_aggregate_function_to_set(
  duckdb_aggregate_function_set set,
  duckdb_aggregate_function function
);
```

Parameters

- `set`: The aggregate function set
- `function`: The function to add

`duckdb_register_aggregate_function_set`

Register the aggregate function set within the given connection.

The set requires at least a single valid overload.

If the set is incomplete or a function with this name already exists DuckDBError is returned.

Syntax

```
            duckdb_state duckdb_register_aggregate_function_set(
  duckdb_connection con,
  duckdb_aggregate_function_set set
);
```

Parameters

- `con`: The connection to register it in.
- `set`: The function set to register

Return Value Whether or not the registration was successful.

`duckdb_create_table_function`

Creates a new empty table function.

The return value should be destroyed with `duckdb_destroy_table_function`.

Return Value The table function object.

Syntax

```
            duckdb_table_function duckdb_create_table_function(

);
```

`duckdb_destroy_table_function`

Destroys the given table function object.

Syntax

```
            void duckdb_destroy_table_function(
  duckdb_table_function *table_function
);
```

Parameters

- `table_function`: The table function to destroy

`duckdb_table_function_set_name`

Sets the name of the given table function.

Syntax

```
            void duckdb_table_function_set_name(
  duckdb_table_function table_function,
  const char *name
);
```

Parameters

- `table_function`: The table function
- `name`: The name of the table function

`duckdb_table_function_add_parameter`

Adds a parameter to the table function.

Syntax

```
            void duckdb_table_function_add_parameter(
  duckdb_table_function table_function,
  duckdb_logical_type type
);
```

Parameters

- `table_function`: The table function.
- `type`: The parameter type. Cannot contain INVALID.

`duckdb_table_function_add_named_parameter`

Adds a named parameter to the table function.

Syntax

```
            void duckdb_table_function_add_named_parameter(
  duckdb_table_function table_function,
  const char *name,
  duckdb_logical_type type
);
```

Parameters

- `table_function`: The table function.
- `name`: The parameter name.
- `type`: The parameter type. Cannot contain INVALID.

`duckdb_table_function_set_extra_info`

Assigns extra information to the table function that can be fetched during binding, etc.

Syntax

```
            void duckdb_table_function_set_extra_info(
  duckdb_table_function table_function,
  void *extra_info,
  duckdb_delete_callback_t destroy
);
```

Parameters

- `table_function`: The table function
- `extra_info`: The extra information
- `destroy`: The callback that will be called to destroy the extra information (if any)

`duckdb_table_function_set_bind`

Sets the bind function of the table function.

Syntax

```
            void duckdb_table_function_set_bind(
  duckdb_table_function table_function,
  duckdb_table_function_bind_t bind
);
```

Parameters

- `table_function`: The table function
- `bind`: The bind function

`duckdb_table_function_set_init`

Sets the init function of the table function.

Syntax

```
            void duckdb_table_function_set_init(
  duckdb_table_function table_function,
  duckdb_table_function_init_t init
);
```

Parameters

- `table_function`: The table function
- `init`: The init function

`duckdb_table_function_set_local_init`

Sets the thread-local init function of the table function.

Syntax

```
            void duckdb_table_function_set_local_init(
  duckdb_table_function table_function,
  duckdb_table_function_init_t init
);
```

Parameters

- `table_function`: The table function
- `init`: The init function

`duckdb_table_function_set_function`

Sets the main function of the table function.

Syntax

```
            void duckdb_table_function_set_function(
  duckdb_table_function table_function,
  duckdb_table_function_t function
);
```

Parameters

- `table_function`: The table function
- `function`: The function

`duckdb_table_function_supports_projection_pushdown`

Sets whether or not the given table function supports projection pushdown.

If this is set to true, the system will provide a list of all required columns in the `init` stage through the `duckdb_init_get_column_count` and `duckdb_init_get_column_index` functions. If this is set to false (the default), the system will expect all columns to be projected.

Syntax

```
            void duckdb_table_function_supports_projection_pushdown(
  duckdb_table_function table_function,
  bool pushdown
);
```

Parameters

- `table_function`: The table function
- `pushdown`: True if the table function supports projection pushdown, false otherwise.

`duckdb_register_table_function`

Register the table function object within the given connection.

The function requires at least a name, a bind function, an init function and a main function.

If the function is incomplete or a function with this name already exists DuckDBError is returned.

Syntax

```
            duckdb_state duckdb_register_table_function(
  duckdb_connection con,
  duckdb_table_function function
);
```

Parameters

- `con`: The connection to register it in.
- `function`: The function pointer

Return Value Whether or not the registration was successful.

`duckdb_bind_get_extra_info`

Retrieves the extra info of the function as set in `duckdb_table_function_set_extra_info`.

Syntax

```
            void *duckdb_bind_get_extra_info(
  duckdb_bind_info info
);
```

Parameters

- `info`: The info object

Return Value The extra info

`duckdb_bind_add_result_column`

Adds a result column to the output of the table function.

Syntax

```
            void duckdb_bind_add_result_column(
  duckdb_bind_info info,
  const char *name,
  duckdb_logical_type type
);
```

Parameters

- `info`: The table function's bind info.
- `name`: The column name.
- `type`: The logical column type.

`duckdb_bind_get_parameter_count`

Retrieves the number of regular (non-named) parameters to the function.

Syntax

```
            idx_t duckdb_bind_get_parameter_count(
  duckdb_bind_info info
);
```

Parameters

- `info`: The info object

Return Value The number of parameters

`duckdb_bind_get_parameter`

Retrieves the parameter at the given index.

The result must be destroyed with `duckdb_destroy_value`.

Syntax

```
            duckdb_value duckdb_bind_get_parameter(
  duckdb_bind_info info,
  idx_t index
);
```

Parameters

- `info`: The info object
- `index`: The index of the parameter to get

Return Value The value of the parameter. Must be destroyed with `duckdb_destroy_value`.

`duckdb_bind_get_named_parameter`

Retrieves a named parameter with the given name.

The result must be destroyed with `duckdb_destroy_value`.

Syntax

```
            duckdb_value duckdb_bind_get_named_parameter(
  duckdb_bind_info info,
  const char *name
);
```

Parameters

- `info`: The info object
- `name`: The name of the parameter

Return Value The value of the parameter. Must be destroyed with `duckdb_destroy_value`.

`duckdb_bind_set_bind_data`

Sets the user-provided bind data in the bind object of the table function. This object can be retrieved again during execution.

Syntax

```
            void duckdb_bind_set_bind_data(
  duckdb_bind_info info,
  void *bind_data,
  duckdb_delete_callback_t destroy
);
```

Parameters

- `info`: The bind info of the table function.
- `bind_data`: The bind data object.
- `destroy`: The callback to destroy the bind data (if any).

`duckdb_bind_set_cardinality`

Sets the cardinality estimate for the table function, used for optimization.

Syntax

```
            void duckdb_bind_set_cardinality(
  duckdb_bind_info info,
  idx_t cardinality,
  bool is_exact
);
```

Parameters

- `info`: The bind data object.
- `is_exact`: Whether or not the cardinality estimate is exact, or an approximation

`duckdb_bind_set_error`

Report that an error has occurred while calling bind on a table function.

Syntax

```
            void duckdb_bind_set_error(
  duckdb_bind_info info,
  const char *error
);
```

Parameters

- `info`: The info object
- `error`: The error message

`duckdb_init_get_extra_info`

Retrieves the extra info of the function as set in `duckdb_table_function_set_extra_info`.

Syntax

```
            void *duckdb_init_get_extra_info(
  duckdb_init_info info
);
```

Parameters

- `info`: The info object

Return Value The extra info

`duckdb_init_get_bind_data`

Gets the bind data set by `duckdb_bind_set_bind_data` during the bind.

Note that the bind data should be considered as read-only. For tracking state, use the init data instead.

Syntax

```
            void *duckdb_init_get_bind_data(
  duckdb_init_info info
);
```

Parameters

- `info`: The info object

Return Value The bind data object

`duckdb_init_set_init_data`

Sets the user-provided init data in the init object. This object can be retrieved again during execution.

Syntax

```
            void duckdb_init_set_init_data(
  duckdb_init_info info,
  void *init_data,
  duckdb_delete_callback_t destroy
);
```

Parameters

- `info`: The info object
- `init_data`: The init data object.
- `destroy`: The callback that will be called to destroy the init data (if any)

`duckdb_init_get_column_count`

Returns the number of projected columns.

This function must be used if projection pushdown is enabled to figure out which columns to emit.

Syntax

```
            idx_t duckdb_init_get_column_count(
  duckdb_init_info info
);
```

Parameters

- `info`: The info object

Return Value The number of projected columns.

`duckdb_init_get_column_index`

Returns the column index of the projected column at the specified position.

This function must be used if projection pushdown is enabled to figure out which columns to emit.

Syntax

```
            idx_t duckdb_init_get_column_index(
  duckdb_init_info info,
  idx_t column_index
);
```

Parameters

- `info`: The info object
- `column_index`: The index at which to get the projected column index, from 0..duckdb_init_get_column_count(info)

Return Value The column index of the projected column.

`duckdb_init_set_max_threads`

Sets how many threads can process this table function in parallel (default: 1)

Syntax

```
            void duckdb_init_set_max_threads(
  duckdb_init_info info,
  idx_t max_threads
);
```

Parameters

- `info`: The info object
- `max_threads`: The maximum amount of threads that can process this table function

`duckdb_init_set_error`

Report that an error has occurred while calling init.

Syntax

```
            void duckdb_init_set_error(
  duckdb_init_info info,
  const char *error
);
```

Parameters

- `info`: The info object
- `error`: The error message

`duckdb_function_get_extra_info`

Retrieves the extra info of the function as set in `duckdb_table_function_set_extra_info`.

Syntax

```
            void *duckdb_function_get_extra_info(
  duckdb_function_info info
);
```

Parameters

- `info`: The info object

Return Value The extra info

`duckdb_function_get_bind_data`

Gets the table function's bind data set by `duckdb_bind_set_bind_data`.

Note that the bind data is read-only. For tracking state, use the init data instead.

Syntax

```
            void *duckdb_function_get_bind_data(
  duckdb_function_info info
);
```

Parameters

- `info`: The function info object.

Return Value The bind data object.

`duckdb_function_get_init_data`

Gets the init data set by `duckdb_init_set_init_data` during the init.

Syntax

```
            void *duckdb_function_get_init_data(
  duckdb_function_info info
);
```

Parameters

- `info`: The info object

Return Value The init data object

`duckdb_function_get_local_init_data`

Gets the thread-local init data set by `duckdb_init_set_init_data` during the local_init.

Syntax

```
            void *duckdb_function_get_local_init_data(
  duckdb_function_info info
);
```

Parameters

- `info`: The info object

Return Value The init data object

`duckdb_function_set_error`

Report that an error has occurred while executing the function.

Syntax

```
            void duckdb_function_set_error(
  duckdb_function_info info,
  const char *error
);
```

Parameters

- `info`: The info object
- `error`: The error message

`duckdb_add_replacement_scan`

Add a replacement scan definition to the specified database.

Syntax

```
            void duckdb_add_replacement_scan(
  duckdb_database db,
  duckdb_replacement_callback_t replacement,
  void *extra_data,
  duckdb_delete_callback_t delete_callback
);
```

Parameters

- db: The database object to add the replacement scan to
- `replacement`: The replacement scan callback
- `extra_data`: Extra data that is passed back into the specified callback
- `delete_callback`: The delete callback to call on the extra data, if any

`duckdb_replacement_scan_set_function_name`

Sets the replacement function name. If this function is called in the replacement callback, the replacement scan is performed. If it is not called, the replacement callback is not performed.

Syntax

```
            void duckdb_replacement_scan_set_function_name(
  duckdb_replacement_scan_info info,
  const char *function_name
);
```

Parameters

- `info`: The info object
- `function_name`: The function name to substitute.

`duckdb_replacement_scan_add_parameter`

Adds a parameter to the replacement scan function.

Syntax

```
            void duckdb_replacement_scan_add_parameter(
  duckdb_replacement_scan_info info,
  duckdb_value parameter
);
```

Parameters

- `info`: The info object
- `parameter`: The parameter to add.

`duckdb_replacement_scan_set_error`

Report that an error has occurred while executing the replacement scan.

Syntax

```
            void duckdb_replacement_scan_set_error(
  duckdb_replacement_scan_info info,
  const char *error
);
```

Parameters

- `info`: The info object
- `error`: The error message

`duckdb_get_profiling_info`

Returns the root node of the profiling information. Returns nullptr, if profiling is not enabled.

Syntax

```
            duckdb_profiling_info duckdb_get_profiling_info(
  duckdb_connection connection
);
```

Parameters

- `connection`: A connection object.

Return Value A profiling information object.

`duckdb_profiling_info_get_value`

Returns the value of the metric of the current profiling info node. Returns nullptr, if the metric does not exist or is not enabled. Currently, the value holds a string, and you can retrieve the string by calling the corresponding function: char *duckdb_get_varchar(duckdb_value value).

Syntax

```
            duckdb_value duckdb_profiling_info_get_value(
  duckdb_profiling_info info,
  const char *key
);
```

Parameters

- `info`: A profiling information object.
- `key`: The name of the requested metric.

Return Value The value of the metric. Must be freed with `duckdb_destroy_value`

`duckdb_profiling_info_get_metrics`

Returns the key-value metric map of this profiling node as a MAP duckdb_value. The individual elements are accessible via the duckdb_value MAP functions.

Syntax

```
            duckdb_value duckdb_profiling_info_get_metrics(
  duckdb_profiling_info info
);
```

Parameters

- `info`: A profiling information object.

Return Value The key-value metric map as a MAP duckdb_value.

`duckdb_profiling_info_get_child_count`

Returns the number of children in the current profiling info node.

Syntax

```
            idx_t duckdb_profiling_info_get_child_count(
  duckdb_profiling_info info
);
```

Parameters

- `info`: A profiling information object.

Return Value The number of children in the current node.

`duckdb_profiling_info_get_child`

Returns the child node at the specified index.

Syntax

```
            duckdb_profiling_info duckdb_profiling_info_get_child(
  duckdb_profiling_info info,
  idx_t index
);
```

Parameters

- `info`: A profiling information object.
- `index`: The index of the child node.

Return Value The child node at the specified index.

`duckdb_appender_create`

Creates an appender object.

Note that the object must be destroyed with `duckdb_appender_destroy`.

Syntax

```
            duckdb_state duckdb_appender_create(
  duckdb_connection connection,
  const char *schema,
  const char *table,
  duckdb_appender *out_appender
);
```

Parameters

- `connection`: The connection context to create the appender in.
- `schema`: The schema of the table to append to, or `nullptr` for the default schema.
- `table`: The table name to append to.
- `out_appender`: The resulting appender object.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_appender_create_ext`

Creates an appender object.

Note that the object must be destroyed with `duckdb_appender_destroy`.

Syntax

```
            duckdb_state duckdb_appender_create_ext(
  duckdb_connection connection,
  const char *catalog,
  const char *schema,
  const char *table,
  duckdb_appender *out_appender
);
```

Parameters

- `connection`: The connection context to create the appender in.
- `catalog`: The catalog of the table to append to, or `nullptr` for the default catalog.
- `schema`: The schema of the table to append to, or `nullptr` for the default schema.
- `table`: The table name to append to.
- `out_appender`: The resulting appender object.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_appender_column_count`

Returns the number of columns that belong to the appender. If there is no active column list, then this equals the table's physical columns.

Syntax

```
            idx_t duckdb_appender_column_count(
  duckdb_appender appender
);
```

Parameters

- `appender`: The appender to get the column count from.

Return Value The number of columns in the data chunks.

`duckdb_appender_column_type`

Returns the type of the column at the specified index. This is either a type in the active column list, or the same type as a column in the receiving table.

Note: The resulting type must be destroyed with `duckdb_destroy_logical_type`.

Syntax

```
            duckdb_logical_type duckdb_appender_column_type(
  duckdb_appender appender,
  idx_t col_idx
);
```

Parameters

- `appender`: The appender to get the column type from.
- `col_idx`: The index of the column to get the type of.

Return Value The `duckdb_logical_type` of the column.

`duckdb_appender_error`

Returns the error message associated with the given appender. If the appender has no error message, this returns `nullptr` instead.

The error message should not be freed. It will be de-allocated when `duckdb_appender_destroy` is called.

Syntax

```
            const char *duckdb_appender_error(
  duckdb_appender appender
);
```

Parameters

- `appender`: The appender to get the error from.

Return Value The error message, or `nullptr` if there is none.

`duckdb_appender_flush`

Flush the appender to the table, forcing the cache of the appender to be cleared. If flushing the data triggers a constraint violation or any other error, then all data is invalidated, and this function returns DuckDBError. It is not possible to append more values. Call duckdb_appender_error to obtain the error message followed by duckdb_appender_destroy to destroy the invalidated appender.

Syntax

```
            duckdb_state duckdb_appender_flush(
  duckdb_appender appender
);
```

Parameters

- `appender`: The appender to flush.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_appender_close`

Closes the appender by flushing all intermediate states and closing it for further appends. If flushing the data triggers a constraint violation or any other error, then all data is invalidated, and this function returns DuckDBError. Call duckdb_appender_error to obtain the error message followed by duckdb_appender_destroy to destroy the invalidated appender.

Syntax

```
            duckdb_state duckdb_appender_close(
  duckdb_appender appender
);
```

Parameters

- `appender`: The appender to flush and close.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_appender_destroy`

Closes the appender by flushing all intermediate states to the table and destroying it. By destroying it, this function de-allocates all memory associated with the appender. If flushing the data triggers a constraint violation, then all data is invalidated, and this function returns DuckDBError. Due to the destruction of the appender, it is no longer possible to obtain the specific error message with duckdb_appender_error. Therefore, call duckdb_appender_close before destroying the appender, if you need insights into the specific error.

Syntax

```
            duckdb_state duckdb_appender_destroy(
  duckdb_appender *appender
);
```

Parameters

- `appender`: The appender to flush, close and destroy.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

duckdb_appender_add_column

Appends a column to the active column list of the appender. Immediately flushes all previous data.

The active column list specifies all columns that are expected when flushing the data. Any non-active columns are filled with their default values, or NULL.

Syntax

```
            duckdb_state duckdb_appender_add_column(
  duckdb_appender appender,
  const char *name
);
```

Parameters

- `appender`: The appender to add the column to.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

duckdb_appender_clear_columns

Removes all columns from the active column list of the appender, resetting the appender to treat all columns as active. Immediately flushes all previous data.

Syntax

```
            duckdb_state duckdb_appender_clear_columns(
  duckdb_appender appender
);
```

Parameters

- `appender`: The appender to clear the columns from.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

duckdb_appender_begin_row

A nop function, provided for backwards compatibility reasons. Does nothing. Only `duckdb_appender_end_row` is required.

Syntax

```
            duckdb_state duckdb_appender_begin_row(
  duckdb_appender appender
);
```

duckdb_appender_end_row

Finish the current row of appends. After end_row is called, the next row can be appended.

Syntax

```
            duckdb_state duckdb_appender_end_row(
  duckdb_appender appender
);
```

Parameters

- `appender`: The appender.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_append_default`

Append a DEFAULT value (NULL if DEFAULT not available for column) to the appender.

Syntax

```
            duckdb_state duckdb_append_default(
  duckdb_appender appender
);
```

`duckdb_append_default_to_chunk`

Append a DEFAULT value, at the specified row and column, (NULL if DEFAULT not available for column) to the chunk created from the specified appender. The default value of the column must be a constant value. Non-deterministic expressions like nextval('seq') or random() are not supported.

Syntax

```
            duckdb_state duckdb_append_default_to_chunk(
  duckdb_appender appender,
  duckdb_data_chunk chunk,
  idx_t col,
  idx_t row
);
```

Parameters

- `appender`: The appender to get the default value from.
- `chunk`: The data chunk to append the default value to.
- `col`: The chunk column index to append the default value to.
- `row`: The chunk row index to append the default value to.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_append_bool`

Append a bool value to the appender.

Syntax

```
            duckdb_state duckdb_append_bool(
  duckdb_appender appender,
  bool value
);
```

`duckdb_append_int8`

Append an int8_t value to the appender.

Syntax

```
            duckdb_state duckdb_append_int8(
  duckdb_appender appender,
  int8_t value
);
```

`duckdb_append_int16`

Append an int16_t value to the appender.

Syntax

```
            duckdb_state duckdb_append_int16(
  duckdb_appender appender,
  int16_t value
);
```

`duckdb_append_int32`

Append an int32_t value to the appender.

Syntax

```
            duckdb_state duckdb_append_int32(
  duckdb_appender appender,
  int32_t value
);
```

`duckdb_append_int64`

Append an int64_t value to the appender.

Syntax

```
            duckdb_state duckdb_append_int64(
  duckdb_appender appender,
  int64_t value
);
```

`duckdb_append_hugeint`

Append a duckdb_hugeint value to the appender.

Syntax

```
            duckdb_state duckdb_append_hugeint(
  duckdb_appender appender,
  duckdb_hugeint value
);
```

`duckdb_append_uint8`

Append a uint8_t value to the appender.

Syntax

```
            duckdb_state duckdb_append_uint8(
  duckdb_appender appender,
  uint8_t value
);
```

`duckdb_append_uint16`

Append a uint16_t value to the appender.

Syntax

```
            duckdb_state duckdb_append_uint16(
  duckdb_appender appender,
  uint16_t value
);
```

`duckdb_append_uint32`

Append a uint32_t value to the appender.

Syntax

```
            duckdb_state duckdb_append_uint32(
  duckdb_appender appender,
  uint32_t value
);
```

`duckdb_append_uint64`

Append a uint64_t value to the appender.

Syntax

```
            duckdb_state duckdb_append_uint64(
  duckdb_appender appender,
  uint64_t value
);
```

`duckdb_append_uhugeint`

Append a duckdb_uhugeint value to the appender.

Syntax

```
            duckdb_state duckdb_append_uhugeint(
  duckdb_appender appender,
  duckdb_uhugeint value
);
```

duckdb_append_float

Append a float value to the appender.

Syntax

```
            duckdb_state duckdb_append_float(
  duckdb_appender appender,
  float value
);
```

duckdb_append_double

Append a double value to the appender.

Syntax

```
            duckdb_state duckdb_append_double(
  duckdb_appender appender,
  double value
);
```

duckdb_append_date

Append a duckdb_date value to the appender.

Syntax

```
            duckdb_state duckdb_append_date(
  duckdb_appender appender,
  duckdb_date value
);
```

duckdb_append_time

Append a duckdb_time value to the appender.

Syntax

```
            duckdb_state duckdb_append_time(
  duckdb_appender appender,
  duckdb_time value
);
```

duckdb_append_timestamp

Append a duckdb_timestamp value to the appender.

Syntax

```
            duckdb_state duckdb_append_timestamp(
  duckdb_appender appender,
  duckdb_timestamp value
);
```

`duckdb_append_interval`

Append a duckdb_interval value to the appender.

Syntax

```
            duckdb_state duckdb_append_interval(
  duckdb_appender appender,
  duckdb_interval value
);
```

`duckdb_append_varchar`

Append a varchar value to the appender.

Syntax

```
            duckdb_state duckdb_append_varchar(
  duckdb_appender appender,
  const char *val
);
```

`duckdb_append_varchar_length`

Append a varchar value to the appender.

Syntax

```
            duckdb_state duckdb_append_varchar_length(
  duckdb_appender appender,
  const char *val,
  idx_t length
);
```

`duckdb_append_blob`

Append a blob value to the appender.

Syntax

```
            duckdb_state duckdb_append_blob(
  duckdb_appender appender,
  const void *data,
  idx_t length
);
```

duckdb_append_null

Append a NULL value to the appender (of any type).

Syntax

```
            duckdb_state duckdb_append_null(
  duckdb_appender appender
);
```

duckdb_append_value

Append a duckdb_value to the appender.

Syntax

```
            duckdb_state duckdb_append_value(
  duckdb_appender appender,
  duckdb_value value
);
```

duckdb_append_data_chunk

Appends a pre-filled data chunk to the specified appender. Attempts casting, if the data chunk types do not match the active appender types.

Syntax

```
            duckdb_state duckdb_append_data_chunk(
  duckdb_appender appender,
  duckdb_data_chunk chunk
);
```

Parameters

- `appender`: The appender to append to.
- `chunk`: The data chunk to append.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

duckdb_table_description_create

Creates a table description object. Note that `duckdb_table_description_destroy` should always be called on the resulting table_description, even if the function returns `DuckDBError`.

Syntax

```
            duckdb_state duckdb_table_description_create(
  duckdb_connection connection,
  const char *schema,
  const char *table,
  duckdb_table_description *out
);
```

Parameters

- `connection`: The connection context.
- `schema`: The schema of the table, or `nullptr` for the default schema.
- `table`: The table name.
- `out`: The resulting table description object.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_table_description_create_ext`

Creates a table description object. Note that `duckdb_table_description_destroy` must be called on the resulting table_description, even if the function returns `DuckDBError`.

Syntax

```
           duckdb_state duckdb_table_description_create_ext(
  duckdb_connection connection,
  const char *catalog,
  const char *schema,
  const char *table,
  duckdb_table_description *out
);
```

Parameters

- `connection`: The connection context.
- `catalog`: The catalog (database) name of the table, or `nullptr` for the default catalog.
- `schema`: The schema of the table, or `nullptr` for the default schema.
- `table`: The table name.
- `out`: The resulting table description object.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_table_description_destroy`

Destroy the TableDescription object.

Syntax

```
           void duckdb_table_description_destroy(
  duckdb_table_description *table_description
);
```

Parameters

- `table_description`: The table_description to destroy.

`duckdb_table_description_error`

Returns the error message associated with the given table_description. If the table_description has no error message, this returns `nullptr` instead. The error message should not be freed. It will be de-allocated when `duckdb_table_description_destroy` is called.

Syntax

```
            const char *duckdb_table_description_error(
  duckdb_table_description table_description
);
```

Parameters

- `table_description`: The table_description to get the error from.

Return Value The error message, or `nullptr` if there is none.

`duckdb_column_has_default`

Check if the column at 'index' index of the table has a DEFAULT expression.

Syntax

```
            duckdb_state duckdb_column_has_default(
  duckdb_table_description table_description,
  idx_t index,
  bool *out
);
```

Parameters

- `table_description`: The table_description to query.
- `index`: The index of the column to query.
- `out`: The out-parameter used to store the result.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_table_description_get_column_name`

Obtain the column name at 'index'. The out result must be destroyed with `duckdb_free`.

Syntax

```
            char *duckdb_table_description_get_column_name(
  duckdb_table_description table_description,
  idx_t index
);
```

Parameters

- `table_description`: The table_description to query.
- `index`: The index of the column to query.

Return Value The column name.

`duckdb_query_arrow`

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Executes a SQL query within a connection and stores the full (materialized) result in an arrow structure. If the query fails to execute, DuckDBError is returned and the error message can be retrieved by calling `duckdb_query_arrow_error`.

Note that after running `duckdb_query_arrow`, `duckdb_destroy_arrow` must be called on the result object even if the query fails, otherwise the error stored within the result will not be freed correctly.

Syntax

```
           duckdb_state duckdb_query_arrow(
  duckdb_connection connection,
  const char *query,
  duckdb_arrow *out_result
);
```

Parameters

- `connection`: The connection to perform the query in.
- `query`: The SQL query to run.
- `out_result`: The query result.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_query_arrow_schema`

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Fetch the internal arrow schema from the arrow result. Remember to call release on the respective ArrowSchema object.

Syntax

```
           duckdb_state duckdb_query_arrow_schema(
  duckdb_arrow result,
  duckdb_arrow_schema *out_schema
);
```

Parameters

- `result`: The result to fetch the schema from.
- `out_schema`: The output schema.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_prepared_arrow_schema`

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Fetch the internal arrow schema from the prepared statement. Remember to call release on the respective ArrowSchema object.

Syntax

```
           duckdb_state duckdb_prepared_arrow_schema(
  duckdb_prepared_statement prepared,
  duckdb_arrow_schema *out_schema
);
```

Parameters

- `prepared`: The prepared statement to fetch the schema from.
- `out_schema`: The output schema.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_result_arrow_array`

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Convert a data chunk into an arrow struct array. Remember to call release on the respective ArrowArray object.

Syntax

```
           void duckdb_result_arrow_array(
  duckdb_result result,
  duckdb_data_chunk chunk,
  duckdb_arrow_array *out_array
);
```

Parameters

- `result`: The result object the data chunk have been fetched from.
- `chunk`: The data chunk to convert.
- `out_array`: The output array.

`duckdb_query_arrow_array`

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Fetch an internal arrow struct array from the arrow result. Remember to call release on the respective ArrowArray object.

This function can be called multiple time to get next chunks, which will free the previous out_array. So consume the out_array before calling this function again.

Syntax

```
           duckdb_state duckdb_query_arrow_array(
  duckdb_arrow result,
  duckdb_arrow_array *out_array
);
```

Parameters

- `result`: The result to fetch the array from.
- `out_array`: The output array.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_arrow_column_count`

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Returns the number of columns present in the arrow result object.

Syntax

```
            idx_t duckdb_arrow_column_count(
  duckdb_arrow result
);
```

Parameters

- `result`: The result object.

Return Value The number of columns present in the result object.

`duckdb_arrow_row_count`

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Returns the number of rows present in the arrow result object.

Syntax

```
            idx_t duckdb_arrow_row_count(
  duckdb_arrow result
);
```

Parameters

- `result`: The result object.

Return Value The number of rows present in the result object.

`duckdb_arrow_rows_changed`

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Returns the number of rows changed by the query stored in the arrow result. This is relevant only for INSERT/UPDATE/DELETE queries. For other queries the rows_changed will be 0.

Syntax

```
            idx_t duckdb_arrow_rows_changed(
  duckdb_arrow result
);
```

Parameters

- `result`: The result object.

Return Value The number of rows changed.

`duckdb_query_arrow_error`

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Returns the error message contained within the result. The error is only set if `duckdb_query_arrow` returns `DuckDBError`.

The error message should not be freed. It will be de-allocated when `duckdb_destroy_arrow` is called.

Syntax

```
            const char *duckdb_query_arrow_error(
  duckdb_arrow result
);
```

Parameters

- `result`: The result object to fetch the error from.

Return Value The error of the result.

`duckdb_destroy_arrow`

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Closes the result and de-allocates all memory allocated for the arrow result.

Syntax

```
            void duckdb_destroy_arrow(
  duckdb_arrow *result
);
```

Parameters

- `result`: The result to destroy.

`duckdb_destroy_arrow_stream`

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Releases the arrow array stream and de-allocates its memory.

Syntax

```
            void duckdb_destroy_arrow_stream(
  duckdb_arrow_stream *stream_p
);
```

Parameters

- `stream_p`: The arrow array stream to destroy.

`duckdb_execute_prepared_arrow`

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Executes the prepared statement with the given bound parameters, and returns an arrow query result. Note that after running `duckdb_execute_prepared_arrow`, `duckdb_destroy_arrow` must be called on the result object.

Syntax

```
           duckdb_state duckdb_execute_prepared_arrow(
  duckdb_prepared_statement prepared_statement,
  duckdb_arrow *out_result
);
```

Parameters

- `prepared_statement`: The prepared statement to execute.
- `out_result`: The query result.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_arrow_scan`

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Scans the Arrow stream and creates a view with the given name.

Syntax

```
           duckdb_state duckdb_arrow_scan(
  duckdb_connection connection,
  const char *table_name,
  duckdb_arrow_stream arrow
);
```

Parameters

- `connection`: The connection on which to execute the scan.
- `table_name`: Name of the temporary view to create.
- `arrow`: Arrow stream wrapper.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_arrow_array_scan`

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Scans the Arrow array and creates a view with the given name. Note that after running `duckdb_arrow_array_scan`, `duckdb_destroy_arrow_stream` must be called on the out stream.

Syntax

```
            duckdb_state duckdb_arrow_array_scan(
  duckdb_connection connection,
  const char *table_name,
  duckdb_arrow_schema arrow_schema,
  duckdb_arrow_array arrow_array,
  duckdb_arrow_stream *out_stream
);
```

Parameters

- `connection`: The connection on which to execute the scan.
- `table_name`: Name of the temporary view to create.
- `arrow_schema`: Arrow schema wrapper.
- `arrow_array`: Arrow array wrapper.
- `out_stream`: Output array stream that wraps around the passed schema, for releasing/deleting once done.

Return Value `DuckDBSuccess` on success or `DuckDBError` on failure.

`duckdb_execute_tasks`

Execute DuckDB tasks on this thread.

Will return after `max_tasks` have been executed, or if there are no more tasks present.

Syntax

```
            void duckdb_execute_tasks(
  duckdb_database database,
  idx_t max_tasks
);
```

Parameters

- `database`: The database object to execute tasks for
- `max_tasks`: The maximum amount of tasks to execute

`duckdb_create_task_state`

Creates a task state that can be used with duckdb_execute_tasks_state to execute tasks until `duckdb_finish_execution` is called on the state.

`duckdb_destroy_state` must be called on the result.

Syntax

```
            duckdb_task_state duckdb_create_task_state(
  duckdb_database database
);
```

Parameters

- `database`: The database object to create the task state for

Return Value The task state that can be used with duckdb_execute_tasks_state.

`duckdb_execute_tasks_state`

Execute DuckDB tasks on this thread.

The thread will keep on executing tasks forever, until duckdb_finish_execution is called on the state. Multiple threads can share the same duckdb_task_state.

Syntax

```
            void duckdb_execute_tasks_state(
  duckdb_task_state state
);
```

Parameters

- `state`: The task state of the executor

`duckdb_execute_n_tasks_state`

Execute DuckDB tasks on this thread.

The thread will keep on executing tasks until either duckdb_finish_execution is called on the state, max_tasks tasks have been executed or there are no more tasks to be executed.

Multiple threads can share the same duckdb_task_state.

Syntax

```
            idx_t duckdb_execute_n_tasks_state(
  duckdb_task_state state,
  idx_t max_tasks
);
```

Parameters

- `state`: The task state of the executor
- `max_tasks`: The maximum amount of tasks to execute

Return Value The amount of tasks that have actually been executed

`duckdb_finish_execution`

Finish execution on a specific task.

Syntax

```
            void duckdb_finish_execution(
  duckdb_task_state state
);
```

Parameters

- `state`: The task state to finish execution

duckdb_task_state_is_finished

Check if the provided duckdb_task_state has finished execution

Syntax

```
            bool duckdb_task_state_is_finished(
  duckdb_task_state state
);
```

Parameters

- `state`: The task state to inspect

Return Value Whether or not duckdb_finish_execution has been called on the task state

duckdb_destroy_task_state

Destroys the task state returned from duckdb_create_task_state.

Note that this should not be called while there is an active duckdb_execute_tasks_state running on the task state.

Syntax

```
            void duckdb_destroy_task_state(
  duckdb_task_state state
);
```

Parameters

- `state`: The task state to clean up

duckdb_execution_is_finished

Returns true if the execution of the current query is finished.

Syntax

```
            bool duckdb_execution_is_finished(
  duckdb_connection con
);
```

Parameters

- `con`: The connection on which to check

duckdb_stream_fetch_chunk

> **Warning.** Deprecation notice. This method is scheduled for removal in a future release.

Fetches a data chunk from the (streaming) duckdb_result. This function should be called repeatedly until the result is exhausted.

The result must be destroyed with `duckdb_destroy_data_chunk`.

This function can only be used on duckdb_results created with 'duckdb_pending_prepared_streaming'

If this function is used, none of the other result functions can be used and vice versa (i.e., this function cannot be mixed with the legacy result functions or the materialized result functions).

It is not known beforehand how many chunks will be returned by this result.

Syntax

```
            duckdb_data_chunk duckdb_stream_fetch_chunk(
  duckdb_result result
);
```

Parameters

- `result`: The result object to fetch the data chunk from.

Return Value The resulting data chunk. Returns `NULL` if the result has an error.

`duckdb_fetch_chunk`

Fetches a data chunk from a duckdb_result. This function should be called repeatedly until the result is exhausted.

The result must be destroyed with `duckdb_destroy_data_chunk`.

It is not known beforehand how many chunks will be returned by this result.

Syntax

```
            duckdb_data_chunk duckdb_fetch_chunk(
  duckdb_result result
);
```

Parameters

- `result`: The result object to fetch the data chunk from.

Return Value The resulting data chunk. Returns `NULL` if the result has an error.

`duckdb_create_cast_function`

Creates a new cast function object.

Return Value The cast function object.

Syntax

```
            duckdb_cast_function duckdb_create_cast_function(

);
```

duckdb_cast_function_set_source_type

Sets the source type of the cast function.

Syntax

```
            void duckdb_cast_function_set_source_type(
  duckdb_cast_function cast_function,
  duckdb_logical_type source_type
);
```

Parameters

- `cast_function`: The cast function object.
- `source_type`: The source type to set.

duckdb_cast_function_set_target_type

Sets the target type of the cast function.

Syntax

```
            void duckdb_cast_function_set_target_type(
  duckdb_cast_function cast_function,
  duckdb_logical_type target_type
);
```

Parameters

- `cast_function`: The cast function object.
- `target_type`: The target type to set.

duckdb_cast_function_set_implicit_cast_cost

Sets the ”cost” of implicitly casting the source type to the target type using this function.

Syntax

```
            void duckdb_cast_function_set_implicit_cast_cost(
  duckdb_cast_function cast_function,
  int64_t cost
);
```

Parameters

- `cast_function`: The cast function object.
- `cost`: The cost to set.

duckdb_cast_function_set_function

Sets the actual cast function to use.

Syntax

```
            void duckdb_cast_function_set_function(
  duckdb_cast_function cast_function,
  duckdb_cast_function_t function
);
```

Parameters

- `cast_function`: The cast function object.
- `function`: The function to set.

`duckdb_cast_function_set_extra_info`

Assigns extra information to the cast function that can be fetched during execution, etc.

Syntax

```
            void duckdb_cast_function_set_extra_info(
  duckdb_cast_function cast_function,
  void *extra_info,
  duckdb_delete_callback_t destroy
);
```

Parameters

- `extra_info`: The extra information
- `destroy`: The callback that will be called to destroy the extra information (if any)

`duckdb_cast_function_get_extra_info`

Retrieves the extra info of the function as set in `duckdb_cast_function_set_extra_info`.

Syntax

```
            void *duckdb_cast_function_get_extra_info(
  duckdb_function_info info
);
```

Parameters

- `info`: The info object.

Return Value The extra info.

`duckdb_cast_function_get_cast_mode`

Get the cast execution mode from the given function info.

Syntax

```
            duckdb_cast_mode duckdb_cast_function_get_cast_mode(
  duckdb_function_info info
);
```

Parameters

- `info`: The info object.

Return Value The cast mode.

`duckdb_cast_function_set_error`

Report that an error has occurred while executing the cast function.

Syntax

```
            void duckdb_cast_function_set_error(
  duckdb_function_info info,
  const char *error
);
```

Parameters

- `info`: The info object.
- `error`: The error message.

`duckdb_cast_function_set_row_error`

Report that an error has occurred while executing the cast function, setting the corresponding output row to NULL.

Syntax

```
            void duckdb_cast_function_set_row_error(
  duckdb_function_info info,
  const char *error,
  idx_t row,
  duckdb_vector output
);
```

Parameters

- `info`: The info object.
- `error`: The error message.
- `row`: The index of the row within the output vector to set to NULL.
- `output`: The output vector.

`duckdb_register_cast_function`

Registers a cast function within the given connection.

Syntax

```
            duckdb_state duckdb_register_cast_function(
  duckdb_connection con,
  duckdb_cast_function cast_function
);
```

Parameters

- `con`: The connection to use.
- `cast_function`: The cast function to register.

Return Value Whether or not the registration was successful.

`duckdb_destroy_cast_function`

Destroys the cast function object.

Syntax

```
            void duckdb_destroy_cast_function(
  duckdb_cast_function *cast_function
);
```

Parameters

- `cast_function`: The cast function object.

C++ API

> The latest version of the DuckDB C++ API is 1.3.1.

> **Warning.** DuckDB's C++ API is primarily intended for internal use. It is not guaranteed to be stable and can change without notice. If you would like to build an application on DuckDB, we recommend using the C API.

Installation

The DuckDB C++ API can be installed as part of the `libduckdb` packages. Please see the installation page for details.

Basic API Usage

DuckDB implements a custom C++ API. This is built around the abstractions of a database instance (`DuckDB` class), multiple `Connections` to the database instance and `QueryResult` instances as the result of queries. The header file for the C++ API is `duckdb.hpp`.

Startup & Shutdown

To use DuckDB, you must first initialize a `DuckDB` instance using its constructor. `DuckDB()` takes as parameter the database file to read and write from. The special value `nullptr` can be used to create an **in-memory database**. Note that for an in-memory database no data is persisted to disk (i.e., all data is lost when you exit the process). The second parameter to the `DuckDB` constructor is an optional `DBConfig` object. In `DBConfig`, you can set various database parameters, for example the read/write mode or memory limits. The `DuckDB` constructor may throw exceptions, for example if the database file is not usable.

With the `DuckDB` instance, you can create one or many `Connection` instances using the `Connection()` constructor. While connections should be thread-safe, they will be locked during querying. It is therefore recommended that each thread uses its own connection if you are in a multithreaded environment.

```
DuckDB db(nullptr);
Connection con(db);
```

Querying

Connections expose the `Query()` method to send a SQL query string to DuckDB from C++. `Query()` fully materializes the query result as a `MaterializedQueryResult` in memory before returning at which point the query result can be consumed. There is also a streaming API for queries, see further below.

```
// create a table
con.Query("CREATE TABLE integers (i INTEGER, j INTEGER)");

// insert three rows into the table
con.Query("INSERT INTO integers VALUES (3, 4), (5, 6), (7, NULL)");

auto result = con.Query("SELECT * FROM integers");
if (result->HasError()) {
    cerr << result->GetError() << endl;
```

```
} else {
    cout << result->ToString() << endl;
}
```

The `MaterializedQueryResult` instance contains firstly two fields that indicate whether the query was successful. `Query` will not throw exceptions under normal circumstances. Instead, invalid queries or other issues will lead to the `success` Boolean field in the query result instance to be set to `false`. In this case an error message may be available in `error` as a string. If successful, other fields are set: the type of statement that was just executed (e.g., `StatementType::INSERT_STATEMENT`) is contained in `statement_type`. The high-level ("Logical type"/"SQL type") types of the result set columns are in `types`. The names of the result columns are in the `names` string vector. In case multiple result sets are returned, for example because the result set contained multiple statements, the result set can be chained using the `next` field.

DuckDB also supports prepared statements in the C++ API with the `Prepare()` method. This returns an instance of `PreparedStatement`. This instance can be used to execute the prepared statement with parameters. Below is an example:

```
std::unique_ptr<PreparedStatement> prepare = con.Prepare("SELECT count(*) FROM a WHERE i = $1");
std::unique_ptr<QueryResult> result = prepare->Execute(12);
```

> **Warning.** Do **not** use prepared statements to insert large amounts of data into DuckDB. See the data import documentation for better options.

UDF API

The UDF API allows the definition of user-defined functions. It is exposed in `duckdb:Connection` through the methods: `CreateScalarFunction()`, `CreateVectorizedFunction()`, and variants. These methods created UDFs into the temporary schema (`TEMP_SCHEMA`) of the owner connection that is the only one allowed to use and change them.

CreateScalarFunction

The user can code an ordinary scalar function and invoke the `CreateScalarFunction()` to register and afterward use the UDF in a `SELECT` statement, for instance:

```
bool bigger_than_four(int value) {
    return value > 4;
}

connection.CreateScalarFunction<bool, int>("bigger_than_four", &bigger_than_four);

connection.Query("SELECT bigger_than_four(i) FROM (VALUES(3), (5)) tbl(i)")->Print();
```

The `CreateScalarFunction()` methods automatically creates vectorized scalar UDFs so they are as efficient as built-in functions, we have two variants of this method interface as follows:

1.

```
template<typename TR, typename... Args>
void CreateScalarFunction(string name, TR (*udf_func)(Args…))
```

- template parameters:
  - **TR** is the return type of the UDF function;
  - **Args** are the arguments up to 3 for the UDF function (this method only supports until ternary functions);
- **name**: is the name to register the UDF function;
- **udf_func**: is a pointer to the UDF function.

This method automatically discovers from the template typenames the corresponding LogicalTypes:

- `bool` → `LogicalType::BOOLEAN`

- `int8_t` → `LogicalType::TINYINT`
- `int16_t` → `LogicalType::SMALLINT`
- `int32_t` → `LogicalType::INTEGER`
- `int64_t` →`LogicalType::BIGINT`
- `float` → `LogicalType::FLOAT`
- `double` → `LogicalType::DOUBLE`
- `string_t` → `LogicalType::VARCHAR`

In DuckDB some primitive types, e.g., `int32_t`, are mapped to the same `LogicalType`: `INTEGER`, `TIME` and `DATE`, then for disambiguation the users can use the following overloaded method.

2.

```
template<typename TR, typename... Args>
void CreateScalarFunction(string name, vector<LogicalType> args, LogicalType ret_type, TR (*udf_
func)(Args…))
```

An example of use would be:

```
int32_t udf_date(int32_t a) {
    return a;
}

con.Query("CREATE TABLE dates (d DATE)");
con.Query("INSERT INTO dates VALUES ('1992-01-01')");

con.CreateScalarFunction<int32_t, int32_t>("udf_date", {LogicalType::DATE}, LogicalType::DATE, &udf_
date);

con.Query("SELECT udf_date(d) FROM dates")->Print();
```

- template parameters:
  - **TR** is the return type of the UDF function;
  - **Args** are the arguments up to 3 for the UDF function (this method only supports until ternary functions);
- **name**: is the name to register the UDF function;
- **args**: are the LogicalType arguments that the function uses, which should match with the template Args types;
- **ret_type**: is the LogicalType of return of the function, which should match with the template TR type;
- **udf_func**: is a pointer to the UDF function.

This function checks the template types against the LogicalTypes passed as arguments and they must match as follow:

- LogicalTypeId::BOOLEAN → bool
- LogicalTypeId::TINYINT → int8_t
- LogicalTypeId::SMALLINT → int16_t
- LogicalTypeId::DATE, LogicalTypeId::TIME, LogicalTypeId::INTEGER → int32_t
- LogicalTypeId::BIGINT, LogicalTypeId::TIMESTAMP → int64_t
- LogicalTypeId::FLOAT, LogicalTypeId::DOUBLE, LogicalTypeId::DECIMAL → double
- LogicalTypeId::VARCHAR, LogicalTypeId::CHAR, LogicalTypeId::BLOB → string_t
- LogicalTypeId::VARBINARY → blob_t

CreateVectorizedFunction

The `CreateVectorizedFunction()` methods register a vectorized UDF such as:

```
/*
* This vectorized function copies the input values to the result vector
*/
template<typename TYPE>
```

```
static void udf_vectorized(DataChunk &args, ExpressionState &state, Vector &result) {
    // set the result vector type
    result.vector_type = VectorType::FLAT_VECTOR;
    // get a raw array from the result
    auto result_data = FlatVector::GetData<TYPE>(result);

    // get the solely input vector
    auto &input = args.data[0];
    // now get an orrified vector
    VectorData vdata;
    input.Orrify(args.size(), vdata);

    // get a raw array from the orrified input
    auto input_data = (TYPE *)vdata.data;

    // handling the data
    for (idx_t i = 0; i < args.size(); i++) {
        auto idx = vdata.sel->get_index(i);
        if ((*vdata.nullmask)[idx]) {
            continue;
        }
        result_data[i] = input_data[idx];
    }
}

con.Query("CREATE TABLE integers (i INTEGER)");
con.Query("INSERT INTO integers VALUES (1), (2), (3), (999)");

con.CreateVectorizedFunction<int, int>("udf_vectorized_int", &&udf_vectorized<int>);

con.Query("SELECT udf_vectorized_int(i) FROM integers")->Print();
```

The Vectorized UDF is a pointer of the type *scalar_function_t*:

```
typedef std::function<void(DataChunk &args, ExpressionState &expr, Vector &result)> scalar_function_t;
```

- **args** is a DataChunk that holds a set of input vectors for the UDF that all have the same length;
- **expr** is an ExpressionState that provides information to the query's expression state;
- **result**: is a Vector to store the result values.

There are different vector types to handle in a Vectorized UDF:

- ConstantVector;
- DictionaryVector;
- FlatVector;
- ListVector;
- StringVector;
- StructVector;
- SequenceVector.

The general API of the `CreateVectorizedFunction()` method is as follows:

1.

```
template<typename TR, typename... Args>
void CreateVectorizedFunction(string name, scalar_function_t udf_func, LogicalType varargs =
LogicalType::INVALID)
```

- template parameters:
  - **TR** is the return type of the UDF function;
  - **Args** are the arguments up to 3 for the UDF function.

- **name** is the name to register the UDF function;
- **udf_func** is a *vectorized* UDF function;
- **varargs** The type of varargs to support, or LogicalTypeId::INVALID (default value) if the function does not accept variable length arguments.

This method automatically discovers from the template typenames the corresponding LogicalTypes:

- bool → LogicalType::BOOLEAN;
- int8_t → LogicalType::TINYINT;
- int16_t → LogicalType::SMALLINT
- int32_t → LogicalType::INTEGER
- int64_t → LogicalType::BIGINT
- float → LogicalType::FLOAT
- double → LogicalType::DOUBLE
- string_t → LogicalType::VARCHAR

2.

```
template<typename TR, typename... Args>
void CreateVectorizedFunction(string name, vector<LogicalType> args, LogicalType ret_type, scalar_
function_t udf_func, LogicalType varargs = LogicalType::INVALID)
```

CLI

CLI Client

> The latest version of the DuckDB CLI client is 1.3.1.

Installation

The DuckDB CLI (Command Line Interface) is a single, dependency-free executable. It is precompiled for Windows, Mac, and Linux for both the stable version and for nightly builds produced by GitHub Actions. Please see the installation page under the CLI tab for download links.

The DuckDB CLI is based on the SQLite command line shell, so CLI-client-specific functionality is similar to what is described in the SQLite documentation (although DuckDB's SQL syntax follows PostgreSQL conventions with a few exceptions).

> DuckDB has a tldr page, which summarizes the most common uses of the CLI client. If you have tldr installed, you can display it by running `tldr duckdb`.

Getting Started

Once the CLI executable has been downloaded, unzip it and save it to any directory. Navigate to that directory in a terminal and enter the command `duckdb` to run the executable. If in a PowerShell or POSIX shell environment, use the command `./duckdb` instead.

Usage

The typical usage of the `duckdb` command is the following:

```
duckdb [OPTIONS] [FILENAME]
```

Options

The `[OPTIONS]` part encodes arguments for the CLI client. Common options include:

- `-csv`: sets the output mode to CSV
- `-json`: sets the output mode to JSON
- `-readonly`: open the database in read-only mode (see concurrency in DuckDB)

For a full list of options, see the command line arguments page.

In-Memory vs. Persistent Database

When no `[FILENAME]` argument is provided, the DuckDB CLI will open a temporary in-memory database. You will see DuckDB's version number, the information on the connection and a prompt starting with a `D`.

```
duckdb
```

```
DuckDB v1.3.1 ({{ site.current_duckdb_codename }}) 2063dda3e6
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
D
```

To open or create a persistent database, simply include a path as a command line argument:

```
duckdb my_database.duckdb
```

Running SQL Statements in the CLI

Once the CLI has been opened, enter a SQL statement followed by a semicolon, then hit enter and it will be executed. Results will be displayed in a table in the terminal. If a semicolon is omitted, hitting enter will allow for multi-line SQL statements to be entered.

```
SELECT 'quack' AS my_column;
```

my_column
quack

The CLI supports all of DuckDB's rich SQL syntax including `SELECT`, `CREATE`, and `ALTER` statements.

Editor Features

The CLI supports autocompletion, and has sophisticated editor features and syntax highlighting on certain platforms.

Exiting the CLI

To exit the CLI, press `Ctrl+D` if your platform supports it. Otherwise, press `Ctrl+C` or use the `.exit` command. If you used a persistent database, DuckDB will automatically checkpoint (save the latest edits to disk) and close. This will remove the `.wal` file (the write-ahead log) and consolidate all of your data into the single-file database.

Dot Commands

In addition to SQL syntax, special dot commands may be entered into the CLI client. To use one of these commands, begin the line with a period (`.`) immediately followed by the name of the command you wish to execute. Additional arguments to the command are entered, space separated, after the command. If an argument must contain a space, either single or double quotes may be used to wrap that parameter. Dot commands must be entered on a single line, and no whitespace may occur before the period. No semicolon is required at the end of the line.

Frequently-used configurations can be stored in the file `~/.duckdbrc`, which will be loaded when starting the CLI client. See the Configuring the CLI section below for further information on these options.

> **Tip.** To prevent the DuckDB CLI client from reading the `~/.duckdbrc` file, start it as follows:
>
> ```
> duckdb -init /dev/null
> ```

Below, we summarize a few important dot commands. To see all available commands, see the dot commands page or use the `.help` command.

Opening Database Files

In addition to connecting to a database when opening the CLI, a new database connection can be made by using the `.open` command. If no additional parameters are supplied, a new in-memory database connection is created. This database will not be persisted when the CLI connection is closed.

```
.open
```

The `.open` command optionally accepts several options, but the final parameter can be used to indicate a path to a persistent database (or where one should be created). The special string `:memory:` can also be used to open a temporary in-memory database.

```
.open persistent.duckdb
```

> **Warning.** `.open` closes the current database. To keep the current database, while adding a new database, use the ATTACH statement.

One important option accepted by `.open` is the `--readonly` flag. This disallows any editing of the database. To open in read only mode, the database must already exist. This also means that a new in-memory database can't be opened in read only mode since in-memory databases are created upon connection.

```
.open --readonly preexisting.duckdb
```

Output Formats

The `.mode` dot command may be used to change the appearance of the tables returned in the terminal output. These include the default `duckbox` mode, `csv` and `json` mode for ingestion by other tools, `markdown` and `latex` for documents, and `insert` mode for generating SQL statements.

Writing Results to a File

By default, the DuckDB CLI sends results to the terminal's standard output. However, this can be modified using either the `.output` or `.once` commands. For details, see the documentation for the output dot command.

Reading SQL from a File

The DuckDB CLI can read both SQL commands and dot commands from an external file instead of the terminal using the `.read` command. This allows for a number of commands to be run in sequence and allows command sequences to be saved and reused.

The `.read` command requires only one argument: the path to the file containing the SQL and/or commands to execute. After running the commands in the file, control will revert back to the terminal. Output from the execution of that file is governed by the same `.output` and `.once` commands that have been discussed previously. This allows the output to be displayed back to the terminal, as in the first example below, or out to another file, as in the second example.

In this example, the file `select_example.sql` is located in the same directory as duckdb.exe and contains the following SQL statement:

```
SELECT *
FROM generate_series(5);
```

To execute it from the CLI, the `.read` command is used.

```
.read select_example.sql
```

The output below is returned to the terminal by default. The formatting of the table can be adjusted using the `.output` or `.once` commands.

```
| generate_series |
|----------------:|
| 0               |
| 1               |
| 2               |
| 3               |
| 4               |
| 5               |
```

Multiple commands, including both SQL and dot commands, can also be run in a single `.read` command. In this example, the file `write_markdown_to_file.sql` is located in the same directory as duckdb.exe and contains the following commands:

```
.mode markdown
.output series.md
SELECT *
FROM generate_series(5);
```

To execute it from the CLI, the `.read` command is used as before.

```
.read write_markdown_to_file.sql
```

In this case, no output is returned to the terminal. Instead, the file `series.md` is created (or replaced if it already existed) with the markdown-formatted results shown here:

```
| generate_series |
|----------------:|
| 0               |
| 1               |
| 2               |
| 3               |
| 4               |
| 5               |
```

Configuring the CLI

Several dot commands can be used to configure the CLI. On startup, the CLI reads and executes all commands in the file `~/.duckdbrc`, including dot commands and SQL statements. This allows you to store the configuration state of the CLI. You may also point to a different initialization file using the `-init`.

Setting a Custom Prompt

As an example, a file in the same directory as the DuckDB CLI named `prompt.sql` will change the DuckDB prompt to be a duck head and run a SQL statement. Note that the duck head is built with Unicode characters and does not work in all terminal environments (e.g., in Windows, unless running with WSL and using the Windows Terminal).

```
.prompt '⚫◗ '
```

To invoke that file on initialization, use this command:

```
duckdb -init prompt.sql
```

This outputs:

```
-- Loading resources from prompt.sql
v<version> <git_hash>
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
⚫◗
```

Non-Interactive Usage

To read/process a file and exit immediately, redirect the file contents in to `duckdb`:

```
duckdb < select_example.sql
```

To execute a command with SQL text passed in directly from the command line, call `duckdb` with two arguments: the database location (or `:memory:`), and a string with the SQL statement to execute.

```
duckdb :memory: "SELECT 42 AS the_answer"
```

Loading Extensions

To load extensions, use DuckDB's SQL `INSTALL` and `LOAD` commands as you would other SQL statements.

```
INSTALL fts;
LOAD fts;
```

For details, see the Extension docs.

Reading from stdin and Writing to stdout

When in a Unix environment, it can be useful to pipe data between multiple commands. DuckDB is able to read data from stdin as well as write to stdout using the file location of stdin (`/dev/stdin`) and stdout (`/dev/stdout`) within SQL commands, as pipes act very similarly to file handles.

This command will create an example CSV:

```
COPY (SELECT 42 AS woot UNION ALL SELECT 43 AS woot) TO 'test.csv' (HEADER);
```

First, read a file and pipe it to the `duckdb` CLI executable. As arguments to the DuckDB CLI, pass in the location of the database to open, in this case, an in-memory database, and a SQL command that utilizes `/dev/stdin` as a file location.

```
cat test.csv | duckdb -c "SELECT * FROM read_csv('/dev/stdin')"
```

woot
42
43

To write back to stdout, the copy command can be used with the `/dev/stdout` file location.

```
cat test.csv | \
    duckdb -c "COPY (SELECT * FROM read_csv('/dev/stdin')) TO '/dev/stdout' WITH (FORMAT csv, HEADER)"
```

```
woot
42
43
```

Reading Environment Variables

The `getenv` function can read environment variables.

Examples

To retrieve the home directory's path from the HOME environment variable, use:

```
SELECT getenv('HOME') AS home;
```

home
/Users/user_name

The output of the `getenv` function can be used to set configuration options. For example, to set the `NULL` order based on the environment variable `DEFAULT_NULL_ORDER`, use:

```
SET default_null_order = getenv('DEFAULT_NULL_ORDER');
```

Restrictions for Reading Environment Variables

The `getenv` function can only be run when the `enable_external_access` option is set to `true` (the default setting). It is only available in the CLI client and is not supported in other DuckDB clients.

Prepared Statements

The DuckDB CLI supports executing prepared statements in addition to regular `SELECT` statements. To create and execute a prepared statement in the CLI client, use the `PREPARE` clause and the `EXECUTE` statement.

Command Line Arguments

The table below summarizes DuckDB's command line options. To list all command line options, use the command:

```
duckdb -help
```

For a list of dot commands available in the CLI shell, see the Dot Commands page.

Argument	Description
`-append`	Append the database to the end of the file
`-ascii`	Set output mode to `ascii`
`-bail`	Stop after hitting an error
`-batch`	Force batch I/O
`-box`	Set output mode to `box`
`-column`	Set output mode to `column`
`-cmd COMMAND`	Run `COMMAND` before reading `stdin`
`-c COMMAND`	Run `COMMAND` and exit
`-csv`	Set output mode to `csv`
`-echo`	Print commands before execution
`-f FILENAME`	Run the script in `FILENAME` and exit. Note that the `~/.duckdbrc` is read and executed first (if it exists)

Argument	Description
`-init FILENAME`	Run the script in `FILENAME` upon startup (instead of `~/.duckdbrc`)
`-header`	Turn headers on
`-help`	Show this message
`-html`	Set output mode to HTML
`-interactive`	Force interactive I/O
`-json`	Set output mode to `json`
`-line`	Set output mode to `line`
`-list`	Set output mode to `list`
`-markdown`	Set output mode to `markdown`
`-newline SEP`	Set output row separator. Default: `\n`
`-nofollow`	Refuse to open symbolic links to database files
`-noheader`	Turn headers off
`-no-stdin`	Exit after processing options instead of reading stdin
`-nullvalue TEXT`	Set text string for `NULL` values. Default: `NULL`
`-quote`	Set output mode to `quote`
`-readonly`	Open the database read-only
`-s COMMAND`	Run `COMMAND` and exit
`-separator SEP`	Set output column separator to `SEP`. Default: `|`
`-table`	Set output mode to `table`
`-ui`	Loads and starts the DuckDB UI. If the UI is not yet installed, it installs the `ui` extension
`-unsigned`	Allow loading of unsigned extensions. This option is intended to be used for developing extensions. Consult the Securing DuckDB page for guidelines on how set up DuckDB in a secure manner
`-version`	Show DuckDB version

Passing a Sequence of Arguments

Note that the CLI arguments are processed in order, similarly to the behavior of the SQLite CLI. For example:

```
duckdb -csv -c 'SELECT 42 AS hello' -json -c 'SELECT 84 AS world'
```

Returns the following:

```
hello
42
[{"world":84}]
```

Dot Commands

Dot commands are available in the DuckDB CLI client. To use one of these commands, begin the line with a period (`.`) immediately followed by the name of the command you wish to execute. Additional arguments to the command are entered, space separated, after the command.

If an argument must contain a space, either single or double quotes may be used to wrap that parameter. Dot commands must be entered on a single line, and no whitespace may occur before the period. No semicolon is required at the end of the line. To see available commands, use the `.help` command.

List of Dot Commands

Command	Description
`.bail <on/off>`{:.language-sql .highlight}	Stop after hitting an error. Default: `off`
`.binary <on/off>`{:.language-sql .highlight}	Turn binary output on or `off`. Default: `off`
`.cd <DIRECTORY>`{:.language-sql .highlight}	Change the working directory to `DIRECTORY`
`.changes <on/off>`{:.language-sql .highlight}	Show number of rows changed by SQL
`.columns`{:.language-sql .highlight}	Column-wise rendering of query results
`.constant <COLOR>`{:.language-sql .highlight}	Sets the syntax highlighting color used for constant values
`.constantcode <CODE>`{:.language-sql .highlight}	Sets the syntax highlighting terminal code used for constant values
`.databases`{:.language-sql .highlight}	List names and files of attached databases
`.echo <on/off>`{:.language-sql .highlight}	Turn command echo on or `off`
`.exit <CODE>`{:.language-sql .highlight}	Exit this program with return-code `CODE`
`.headers <on/off>`{:.language-sql .highlight}	Turn display of headers on or `off`. Does not apply to duckbox mode
`.help <-all> <PATTERN>`{:.language-sql .highlight}	Show help text for `PATTERN`
`.highlight <on/off>`{:.language-sql .highlight}	Toggle syntax highlighting in the shell on / `off`. See the query syntax highlighting section for more details
`.highlight_colors <COMPONENT> <COLOR>`{:.language-sql .highlight}	Configure the color of each component in (duckbox only). See the result syntax highlighting section for more details
`.highlight_results <on/off>`{:.language-sql .highlight}	Toggle highlighting in result tables on / `off` (duckbox only). See the result syntaxx highlighting section for more details
`.import <FILE> <TABLE>`{:.language-sql .highlight}	Import data from `FILE` into `TABLE`
`.indexes <TABLE>`{:.language-sql .highlight}	Show names of indexes
`.keyword <COLOR>`{:.language-sql .highlight}	Sets the syntax highlighting color used for keywords

Command	Description
`.keywordcode <CODE>`{:.language-sql .highlight}	Sets the syntax highlighting terminal code used for keywords
`.large_number_rendering <all/footer/off>`{:.language-sql .highlight}	Toggle readable rendering of large numbers (duckbox only, default: `footer`)
`.log <FILE/off>`{:.language-sql .highlight}	Turn logging on or `off`. `FILE` can be `stderr` / `stdout`
`.maxrows <COUNT>`{:.language-sql .highlight}	Sets the maximum number of rows for display. Only for duckbox mode
`.maxwidth <COUNT>`{:.language-sql .highlight}	Sets the maximum width in characters. 0 defaults to terminal width. Only for duckbox mode
`.mode <MODE> <TABLE>`{:.language-sql .highlight}	Set output mode
`.multiline`{:.language-sql .highlight}	Set multi-line mode (default)
`.nullvalue <STRING>`{:.language-sql .highlight}	Use `STRING` in place of `NULL` values. Default: `NULL`
`.once <OPTIONS> <FILE>`{:.language-sql .highlight}	Output for the next SQL command only to `FILE`
`.open <OPTIONS> <FILE>`{:.language-sql .highlight}	Close existing database and reopen `FILE`
`.output <FILE>`{:.language-sql .highlight}	Send output to `FILE` or `stdout` if `FILE` is omitted
`.print <STRING...>`{:.language-sql .highlight}	Print literal `STRING`
`.prompt <MAIN> <CONTINUE>`{:.language-sql .highlight}	Replace the standard prompts
`.quit`{:.language-sql .highlight}	Exit this program
`.read <FILE>`{:.language-sql .highlight}	Read input from `FILE`
`.rows`{:.language-sql .highlight}	Row-wise rendering of query results (default)
`.safe_mode`{:.language-sql .highlight}	Activates safe mode
`.schema <PATTERN>`{:.language-sql .highlight}	Show the `CREATE` statements matching `PATTERN`
`.separator <COL> <ROW>`{:.language-sql .highlight}	Change the column and row separators
`.shell <CMD> <ARGS...>`{:.language-sql .highlight}	Run `CMD` with `ARGS...` in a system shell
`.show`{:.language-sql .highlight}	Show the current values for various settings

Command	Description
`.singleline`{:.language-sql .highlight}	Set single-line mode
`.system <CMD> <ARGS...>`{:.language-sql .highlight}	Run CMD with ARGS... in a system shell
`.tables <TABLE>`{:.language-sql .highlight}	List names of tables matching LIKE pattern TABLE
`.timer <on/off>`{:.language-sql .highlight}	Turn SQL timer on or off. SQL statements separated by ; but *not* separated via newline are measured together
`.width <NUM1> <NUM2> ...`{:.language-sql .highlight}	Set minimum column widths for columnar output

Using the `.help` Command

The `.help` text may be filtered by passing in a text string as the first argument.

```
.help m
```

```
.maxrows COUNT          Sets the maximum number of rows for display (default: 40). Only for duckbox mode.
.maxwidth COUNT          Sets the maximum width in characters. 0 defaults to terminal width. Only for
duckbox mode.
.mode MODE ?TABLE?       Set output mode
```

`.output`: Writing Results to a File

By default, the DuckDB CLI sends results to the terminal's standard output. However, this can be modified using either the `.output` or `.once` commands. Pass in the desired output file location as a parameter. The `.once` command will only output the next set of results and then revert to standard out, but `.output` will redirect all subsequent output to that file location. Note that each result will overwrite the entire file at that destination. To revert back to standard output, enter `.output` with no file parameter.

In this example, the output format is changed to `markdown`, the destination is identified as a Markdown file, and then DuckDB will write the output of the SQL statement to that file. Output is then reverted to standard output using `.output` with no parameter.

```
.mode markdown
.output my_results.md
SELECT 'taking flight' AS output_column;
.output
SELECT 'back to the terminal' AS displayed_column;
```

The file `my_results.md` will then contain:

```
| output_column |
|---------------|
| taking flight |
```

The terminal will then display:

```
|   displayed_column   |
|----------------------|
| back to the terminal |
```

A common output format is CSV, or comma separated values. DuckDB supports SQL syntax to export data as CSV or Parquet, but the CLI-specific commands may be used to write a CSV instead if desired.

```
.mode csv
.once my_output_file.csv
SELECT 1 AS col_1, 2 AS col_2
UNION ALL
SELECT 10 AS col1, 20 AS col_2;
```

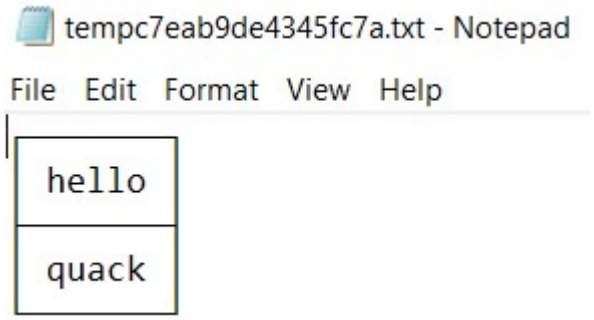
The file `my_output_file.csv` will then contain:

```
col_1,col_2
1,2
10,20
```

By passing special options (flags) to the `.once` command, query results can also be sent to a temporary file and automatically opened in the user's default program. Use either the `-e` flag for a text file (opened in the default text editor), or the `-x` flag for a CSV file (opened in the default spreadsheet editor). This is useful for more detailed inspection of query results, especially if there is a relatively large result set. The `.excel` command is equivalent to `.once -x`.

```
.once -e
SELECT 'quack' AS hello;
```

The results then open in the default text file editor of the system, for example:



> **Tip.** macOS users can copy the results to their clipboards using `pbcopy` by using `.once` to output to `pbcopy` via a pipe: `.once |pbcopy`
>
> Combining this with the `.headers off` and `.mode lines` options can be particularly effective.

Querying the Database Schema

All DuckDB clients support querying the database schema with SQL, but the CLI has additional dot commands that can make it easier to understand the contents of a database. The `.tables` command will return a list of tables in the database. It has an optional argument that will filter the results according to a `LIKE` pattern.

```
CREATE TABLE swimmers AS SELECT 'duck' AS animal;
CREATE TABLE fliers AS SELECT 'duck' AS animal;
CREATE TABLE walkers AS SELECT 'duck' AS animal;
.tables
```

```
fliers    swimmers  walkers
```

For example, to filter to only tables that contain an `l`, use the `LIKE` pattern `%l%`.

```
.tables %l%
```

```
fliers   walkers
```

The `.schema` command will show all of the SQL statements used to define the schema of the database.

```
.schema
```

```
CREATE TABLE fliers (animal VARCHAR);
CREATE TABLE swimmers (animal VARCHAR);
CREATE TABLE walkers (animal VARCHAR);
```

Syntax Highlighters

The DuckDB CLI client has a syntax highlighter for the SQL queries and another for the duckbox-formatted result tables.

Configuring the Query Syntax Highlighter

By default the shell includes support for syntax highlighting. The CLI's syntax highlighter can be configured using the following commands.

To turn off the highlighter:

```
.highlight off
```

To turn on the highlighter:

```
.highlight on
```

To configure the color used to highlight constants:

```
.constant
[red|green|yellow|blue|magenta|cyan|white|brightblack|brightred|brightgreen|brightyellow|brightblue|brightmagenta
```

```
.constantcode <terminal_code>
```

For example:

```
.constantcode 033[31m
```

To configure the color used to highlight keywords:

```
.keyword
[red|green|yellow|blue|magenta|cyan|white|brightblack|brightred|brightgreen|brightyellow|brightblue|brightmagenta
```

```
.keywordcode <terminal_code>
```

For example:

```
.keywordcode 033[31m
```

Configuring the Result Syntax Highlighter

By default, the result highlighting makes a few small modifications:

- Bold column names
- NULL values are greyed out
- Layout elements are grayed out

The highlighting of each of the components can be customized using the `.highlight_colors` command. For example:

```
.highlight_colors layout red
.highlight_colors column_type yellow
.highlight_colors column_name yellow bold_underline
.highlight_colors numeric_value cyan underline
.highlight_colors temporal_value red bold
.highlight_colors string_value green bold
.highlight_colors footer gray
```

The result highlighting can be disabled using `.highlight_results off`.

Shorthands

DuckDB's CLI allows using shorthands for dot commands. Once a sequence of characters can unambiguously completed to a dot command or an argument, the CLI (silently) autocompletes them. For example:

```
.mo ma
```

Is equivalent to:

```
.mode markdown
```

> **Tip.** Avoid using shorthands in SQL scripts to improve readability and ensure that the scripts and futureproof.

Importing Data from CSV

> **Deprecated.** This feature is only included for compatibility reasons and may be removed in the future. Use the `read_csv` function or the COPY statement to load CSV files.

DuckDB supports SQL syntax to directly query or import CSV files, but the CLI-specific commands may be used to import a CSV instead if desired. The `.import` command takes two arguments and also supports several options. The first argument is the path to the CSV file, and the second is the name of the DuckDB table to create. Since DuckDB requires stricter typing than SQLite (upon which the DuckDB CLI is based), the destination table must be created before using the `.import` command. To automatically detect the schema and create a table from a CSV, see the `read_csv` examples in the import docs.

In this example, a CSV file is generated by changing to CSV mode and setting an output file location:

```
.mode csv
.output import_example.csv
SELECT 1 AS col_1, 2 AS col_2 UNION ALL SELECT 10 AS col1, 20 AS col_2;
```

Now that the CSV has been written, a table can be created with the desired schema and the CSV can be imported. The output is reset to the terminal to avoid continuing to edit the output file specified above. The `--skip N` option is used to ignore the first row of data since it is a header row and the table has already been created with the correct column names.

```
.mode csv
.output
CREATE TABLE test_table (col_1 INTEGER, col_2 INTEGER);
.import import_example.csv test_table --skip 1
```

Note that the `.import` command utilizes the current `.mode` and `.separator` settings when identifying the structure of the data to import. The `--csv` option can be used to override that behavior.

```
.import import_example.csv test_table --skip 1 --csv
```

Output Formats

The `.mode` dot command may be used to change the appearance of the tables returned in the terminal output. In addition to customizing the appearance, these modes have additional benefits. This can be useful for presenting DuckDB output elsewhere by redirecting the terminal output to a file. Using the `insert` mode will build a series of SQL statements that can be used to insert the data at a later point. The `markdown` mode is particularly useful for building documentation and the `latex` mode is useful for writing academic papers.

List of Output Formats

Mode	Description
`ascii`	Columns/rows delimited by 0x1F and 0x1E
`box`	Tables using unicode box-drawing characters
`csv`	Comma-separated values
`column`	Output in columns (See `.width`)
`duckbox`	Tables with extensive features (default)
`html`	HTML `<table>` code
`insert <TABLE>{:.language-sql .highlight}`	SQL insert statements for `<TABLE>`{:.language-sql .highlight}
`json`	Results in a JSON array
`jsonlines`	Results in a NDJSON
`latex`	LaTeX tabular environment code
`line`	One value per line
`list`	Values delimited by |
`markdown`	Markdown table format
`quote`	Escape answers as for SQL
`table`	ASCII-art table
`tabs`	Tab-separated values
`tcl`	TCL list elements
`trash`	No output

Changing the Output Format

Use the vanilla `.mode` dot command to query the appearance currently in use.

```
.mode
```

```
current output mode: duckbox
```

Use the `.mode` dot command with an argument to set the output format.

```
.mode markdown
SELECT 'quacking intensifies' AS incoming_ducks;
```

```
|    incoming_ducks    |
|----------------------|
| quacking intensifies |
```

The output appearance can also be adjusted with the `.separator` command. If using an export mode that relies on a separator (`csv` or `tabs` for example), the separator will be reset when the mode is changed. For example, `.mode csv` will set the separator to a comma (`,`). Using `.separator "|"` will then convert the output to be pipe-separated.

```
.mode csv
SELECT 1 AS col_1, 2 AS col_2
UNION ALL
SELECT 10 AS col1, 20 AS col_2;
```

```
col_1,col_2
1,2
10,20
```

```
.separator "|"
SELECT 1 AS col_1, 2 AS col_2
UNION ALL
SELECT 10 AS col1, 20 AS col_2;
```

```
col_1|col_2
1|2
10|20
```

duckbox Mode

By default, DuckDB renders query results in `duckbox` mode, which is a feature-rich ASCII-art style output format.

The duckbox mode supports the `large_number_rendering` option, which allows human-readable rendering of large numbers. It has three levels:

- `off` – All numbers are printed using regular formatting.
- `footer` (default) – Large numbers are augmented with the human-readable format. Only applies to single-row results.
- `all` - All large numbers are replaced with the human-readable format.

See the following examples:

```
.large_number_rendering off
SELECT pi() * 1_000_000_000 AS x;
```

```
┌────────────────────┐
│         x          │
│       double       │
├────────────────────┤
│ 3141592653.589793  │
└────────────────────┘
```

```
.large_number_rendering footer
SELECT pi() * 1_000_000_000 AS x;
```

```
┌────────────────────┐
│         x          │
│       double       │
├────────────────────┤
│ 3141592653.589793  │
│  (3.14 billion)    │
└────────────────────┘
```

```
.large_number_rendering all
SELECT pi() * 1_000_000_000 AS x;
```

```
┌──────────────┐
│      x       │
│    double    │
├──────────────┤
│ 3.14 billion │
└──────────────┘
```

Editing

> The linenoise-based CLI editor is currently only available for macOS and Linux.

DuckDB's CLI uses a line-editing library based on linenoise, which has shortcuts that are based on Emacs mode of readline. Below is a list of available commands.

Moving

Key	Action
`Left`	Move back a character
`Right`	Move forward a character
`Up`	Move up a line. When on the first line, move to previous history entry
`Down`	Move down a line. When on last line, move to next history entry
`Home`	Move to beginning of buffer
`End`	Move to end of buffer
`Ctrl+Left`	Move back a word
`Ctrl+Right`	Move forward a word
`Ctrl+A`	Move to beginning of buffer
`Ctrl+B`	Move back a character
`Ctrl+E`	Move to end of buffer
`Ctrl+F`	Move forward a character
`Alt+Left`	Move back a word
`Alt+Right`	Move forward a word

History

Key	Action
`Ctrl+P`	Move to previous history entry
`Ctrl+N`	Move to next history entry
`Ctrl+R`	Search the history
`Ctrl+S`	Search the history
`Alt+<`	Move to first history entry
`Alt+>`	Move to last history entry
`Alt+N`	Search the history
`Alt+P`	Search the history

Changing Text

Key	Action
`Backspace`	Delete previous character
`Delete`	Delete next character
`Ctrl+D`	Delete next character. When buffer is empty, end editing
`Ctrl+H`	Delete previous character

Key	Action
`Ctrl+K`	Delete everything after the cursor
`Ctrl+T`	Swap current and next character
`Ctrl+U`	Delete all text
`Ctrl+W`	Delete previous word
`Alt+C`	Convert next word to titlecase
`Alt+D`	Delete next word
`Alt+L`	Convert next word to lowercase
`Alt+R`	Delete all text
`Alt+T`	Swap current and next word
`Alt+U`	Convert next word to uppercase
`Alt+Backspace`	Delete previous word
`Alt+\`	Delete spaces around cursor

Completing

Key	Action
`Tab`	Autocomplete. When autocompleting, cycle to next entry
`Shift+Tab`	When autocompleting, cycle to previous entry
`Esc+Esc`	When autocompleting, revert autocompletion

Miscellaneous

Key	Action
`Enter`	Execute query. If query is not complete, insert a newline at the end of the buffer
`Ctrl+J`	Execute query. If query is not complete, insert a newline at the end of the buffer
`Ctrl+C`	Cancel editing of current query
`Ctrl+G`	Cancel editing of current query
`Ctrl+L`	Clear screen
`Ctrl+O`	Cancel editing of current query
`Ctrl+X`	Insert a newline after the cursor
`Ctrl+Z`	Suspend CLI and return to shell, use `fg` to re-open

Using Read-Line

If you prefer, you can use `rlwrap` to use read-line directly with the shell. Then, use `Shift+Enter` to insert a newline and `Enter` to execute the query:

```
rlwrap --substitute-prompt="D " duckdb -batch
```

Safe Mode

The DuckDB CLI client supports "safe mode". In safe mode, the CLI is prevented from accessing external files other than the database file that it was initially connected to and prevented from interacting with the host file system.

This has the following effects:

- The following dot commands are disabled:
  - `.cd`
  - `.excel`
  - `.import`
  - `.log`
  - `.once`
  - `.open`
  - `.output`
  - `.read`
  - `.sh`
  - `.system`
- Auto-complete no longer scans the file system for files to suggest as auto-complete targets.
- The `getenv` function is disabled.
- The `enable_external_access` option is set to `false`. This implies that:
  - `ATTACH` cannot attach to a database in a file.
  - `COPY` cannot read to or write from files.
  - Functions such as `read_csv`, `read_parquet`, `read_json`, etc. cannot read from an external source.

Once safe mode is activated, it cannot be deactivated in the same DuckDB CLI session.

For more information on running DuckDB in secure environments, see the "Securing DuckDB" page.

Autocomplete

The shell offers context-aware autocomplete of SQL queries through the `autocomplete` extension. autocomplete is triggered by pressing `Tab`.

Multiple autocomplete suggestions can be present. You can cycle forwards through the suggestions by repeatedly pressing `Tab`, or `Shift+Tab` to cycle backwards. autocompletion can be reverted by pressing ESC twice.

The shell autocompletes four different groups:

- Keywords
- Table names and table functions
- Column names and scalar functions
- File names

The shell looks at the position in the SQL statement to determine which of these autocompletions to trigger. For example:

```
SELECT s
```

```
student_id
```

```
SELECT student_id F
```

```
FROM
```

```
SELECT student_id FROM g
```

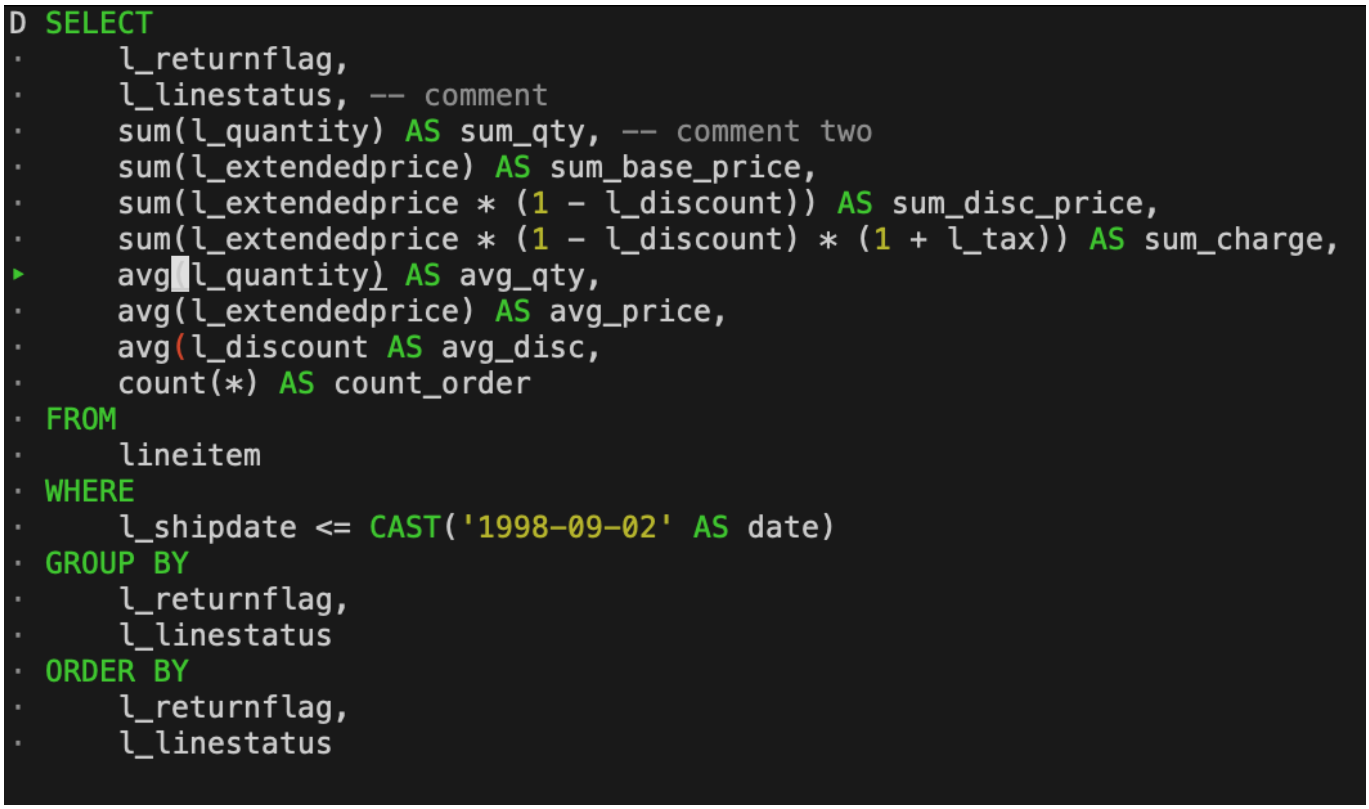
```
grades
```

```
SELECT student_id FROM 'd
'data/
SELECT student_id FROM 'data/
'data/grades.csv
```

Syntax Highlighting

> Syntax highlighting in the CLI is currently only available for macOS and Linux.

SQL queries that are written in the shell are automatically highlighted using syntax highlighting.



There are several components of a query that are highlighted in different colors. The colors can be configured using dot commands. Syntax highlighting can also be disabled entirely using the `.highlight off` command.

Below is a list of components that can be configured.

Type	Command	Default color
Keywords	`.keyword`	`green`
Constants ad literals	`.constant`	`yellow`
Comments	`.comment`	`brightblack`
Errors	`.error`	`red`
Continuation	`.cont`	`brightblack`
Continuation (Selected)	`.cont_sel`	`green`

The components can be configured using either a supported color name (e.g., `.keyword red`), or by directly providing a terminal code to use for rendering (e.g., `.keywordcode \033[31m`). Below is a list of supported color names and their corresponding terminal codes.

Color	Terminal code
red	`\033[31m`
green	`\033[32m`
yellow	`\033[33m`
blue	`\033[34m`
magenta	`\033[35m`
cyan	`\033[36m`
white	`\033[37m`
brightblack	`\033[90m`
brightred	`\033[91m`
brightgreen	`\033[92m`
brightyellow	`\033[93m`
brightblue	`\033[94m`
brightmagenta	`\033[95m`
brightcyan	`\033[96m`
brightwhite	`\033[97m`

For example, here is an alternative set of syntax highlighting colors:

```
.keyword brightred
.constant brightwhite
.comment cyan
.error yellow
.cont blue
.cont_sel brightblue
```

If you wish to start up the CLI with a different set of colors every time, you can place these commands in the `~/.duckdbrc` file that is loaded on start-up of the CLI.

Error Highlighting

The shell has support for highlighting certain errors. In particular, mismatched brackets and unclosed quotes are highlighted in red (or another color if specified). This highlighting is automatically disabled for large queries. In addition, it can be disabled manually using the `.render_errors off` command.

Dart Client

> The latest version of the DuckDB Dart client is {{ site.current_duckdb_dart_version }}.

DuckDB.Dart is the native Dart API for DuckDB.

Installation

DuckDB.Dart can be installed from pub.dev. Please see the API Reference for details.

Use This Package as a Library

Depend on It

Add the dependency with Flutter:

```
flutter pub add dart_duckdb
```

This will add a line like this to your package's `pubspec.yaml` (and run an implicit `flutter pub get`):

```
dependencies:
  dart_duckdb: ^1.1.3
```

Alternatively, your editor might support `flutter pub get`. Check the docs for your editor to learn more.

Import It

Now in your Dart code, you can import it:

```
import 'package:dart_duckdb/dart_duckdb.dart';
```

Usage Examples

See the example projects in the `duckdb-dart` repository:

- `cli`: command-line application
- `duckdbexplorer`: GUI application which builds for desktop operating systems as well as Android and iOS.

Here are some common code snippets for DuckDB.Dart:

Querying an In-Memory Database

```
import 'package:dart_duckdb/dart_duckdb.dart';

void main() {
  final db = duckdb.open(":memory:");
  final connection = duckdb.connect(db);
```

```
  connection.execute('''
    CREATE TABLE users (id INTEGER, name VARCHAR, age INTEGER);
    INSERT INTO users VALUES (1, 'Alice', 30), (2, 'Bob', 25);
  ''');

  final result = connection.query("SELECT * FROM users WHERE age > 28").fetchAll();

  for (final row in result) {
    print(row);
  }

  connection.dispose();
  db.dispose();
}
```

Queries on Background Isolates

```
import 'package:dart_duckdb/dart_duckdb.dart';

void main() {
  final db = duckdb.open(":memory:");
  final connection = duckdb.connect(db);

  await Isolate.spawn(backgroundTask, db.transferrable);

  connection.dispose();
  db.dispose();
}

void backgroundTask(TransferableDatabase transferableDb) {
  final connection = duckdb.connectWithTransferred(transferableDb);
  // Access database ...
  // fetch is needed to send the data back to the main isolate
}
```

Go Client

> The latest version of the DuckDB Go client is {{ site.current_duckdb_go_version }}.

The DuckDB Go driver, `go-duckdb`, allows using DuckDB via the `database/sql` interface. For examples on how to use this interface, see the official documentation and tutorial.

> The `go-duckdb` project, hosted at https://github.com/marcboeker/go-duckdb, is the official DuckDB Go client.

Installation

To install the `go-duckdb` client, run:

```
go get github.com/marcboeker/go-duckdb
```

Importing

To import the DuckDB Go package, add the following entries to your imports:

```
import (
    "database/sql"
    _ "github.com/marcboeker/go-duckdb"
)
```

Appender

The DuckDB Go client supports the DuckDB Appender API for bulk inserts. You can obtain a new Appender by supplying a DuckDB connection to `NewAppenderFromConn()`. For example:

```
connector, err := duckdb.NewConnector("test.db", nil)
if err != nil {
  ...
}
conn, err := connector.Connect(context.Background())
if err != nil {
  ...
}
defer conn.Close()

// Retrieve appender from connection (note that you have to create the table 'test' beforehand).
appender, err := NewAppenderFromConn(conn, "", "test")
if err != nil {
  ...
}
defer appender.Close()

err = appender.AppendRow(...)
if err != nil {
```

```
  ...
}

// Optional, if you want to access the appended rows immediately.
err = appender.Flush()
if err != nil {
  ...
}
```

Examples

Simple Example

An example for using the Go API is as follows:

```
package main

import (
    "database/sql"
    "errors"
    "fmt"
    "log"

    _ "github.com/marcboeker/go-duckdb"
)

func main() {
    db, err := sql.Open("duckdb", "")
    if err != nil {
        log.Fatal(err)
    }
    defer db.Close()

    _, err = db.Exec(` CREATE TABLE people (id INTEGER, name VARCHAR)`)
    if err != nil {
        log.Fatal(err)
    }
    _, err = db.Exec(` INSERT INTO people VALUES (42, 'John')`)
    if err != nil {
        log.Fatal(err)
    }

    var (
        id   int
        name string
    )
    row := db.QueryRow(` SELECT id, name FROM people`)
    err = row.Scan(&id, &name)
    if errors.Is(err, sql.ErrNoRows) {
        log.Println("no rows")
    } else if err != nil {
        log.Fatal(err)
    }

    fmt.Printf("id: %d, name: %s\n", id, name)
}
```

More Examples

For more examples, see the examples in the duckdb-go repository.

Java JDBC Client

> The latest version of the DuckDB Java (JDBC) client is {{ site.current_duckdb_java_short_version }}.

Installation

The DuckDB Java JDBC API can be installed from Maven Central. Please see the installation page for details.

Basic API Usage

DuckDB's JDBC API implements the main parts of the standard Java Database Connectivity (JDBC) API, version 4.1. Describing JDBC is beyond the scope of this page, see the official documentation for details. Below we focus on the DuckDB-specific parts.

Refer to the externally hosted API Reference for more information about our extensions to the JDBC specification, or the below Arrow Methods.

Startup & Shutdown

In JDBC, database connections are created through the standard `java.sql.DriverManager` class. The driver should auto-register in the `DriverManager`, if that does not work for some reason, you can enforce registration using the following statement:

```
Class.forName("org.duckdb.DuckDBDriver");
```

To create a DuckDB connection, call `DriverManager` with the `jdbc:duckdb:` JDBC URL prefix, like so:

```
import java.sql.Connection;
import java.sql.DriverManager;

Connection conn = DriverManager.getConnection("jdbc:duckdb:");
```

To use DuckDB-specific features such as the Appender, cast the object to a `DuckDBConnection`:

```
import java.sql.DriverManager;
import org.duckdb.DuckDBConnection;

DuckDBConnection conn = (DuckDBConnection) DriverManager.getConnection("jdbc:duckdb:");
```

When using the `jdbc:duckdb:` URL alone, an **in-memory database** is created. Note that for an in-memory database no data is persisted to disk (i.e., all data is lost when you exit the Java program). If you would like to access or create a persistent database, append its file name after the path. For example, if your database is stored in `/tmp/my_database`, use the JDBC URL `jdbc:duckdb:/tmp/my_database` to create a connection to it.

It is possible to open a DuckDB database file in **read-only** mode. This is for example useful if multiple Java processes want to read the same database file at the same time. To open an existing database file in read-only mode, set the connection property `duckdb.read_only` like so:

```
Properties readOnlyProperty = new Properties();
readOnlyProperty.setProperty("duckdb.read_only", "true");
Connection conn = DriverManager.getConnection("jdbc:duckdb:/tmp/my_database", readOnlyProperty);
```

Additional connections can be created using the `DriverManager`. A more efficient mechanism is to call the `DuckDBConnection#duplicate()` method:

```
Connection conn2 = ((DuckDBConnection) conn).duplicate();
```

Multiple connections are allowed, but mixing read-write and read-only connections is unsupported.

Configuring Connections

Configuration options can be provided to change different settings of the database system. Note that many of these settings can be changed later on using PRAGMA statements as well.

```
Properties connectionProperties = new Properties();
connectionProperties.setProperty("temp_directory", "/path/to/temp/dir/");
Connection conn = DriverManager.getConnection("jdbc:duckdb:/tmp/my_database", connectionProperties);
```

Querying

DuckDB supports the standard JDBC methods to send queries and retrieve result sets. First a `Statement` object has to be created from the `Connection`, this object can then be used to send queries using `execute` and `executeQuery`. `execute()` is meant for queries where no results are expected like `CREATE TABLE` or `UPDATE` etc. and `executeQuery()` is meant to be used for queries that produce results (e.g., `SELECT`). Below two examples. See also the JDBC `Statement` and `ResultSet` documentations.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

Connection conn = DriverManager.getConnection("jdbc:duckdb:");

// create a table
Statement stmt = conn.createStatement();
stmt.execute("CREATE TABLE items (item VARCHAR, value DECIMAL(10, 2), count INTEGER)");
// insert two items into the table
stmt.execute("INSERT INTO items VALUES ('jeans', 20.0, 1), ('hammer', 42.2, 2)");

try (ResultSet rs = stmt.executeQuery("SELECT * FROM items")) {
    while (rs.next()) {
        System.out.println(rs.getString(1));
        System.out.println(rs.getInt(3));
    }
}
stmt.close();
```

```
jeans
1
hammer
2
```

DuckDB also supports prepared statements as per the JDBC API:

```
import java.sql.PreparedStatement;

try (PreparedStatement stmt = conn.prepareStatement("INSERT INTO items VALUES (?, ?, ?);")) {
    stmt.setString(1, "chainsaw");
    stmt.setDouble(2, 500.0);
    stmt.setInt(3, 42);
    stmt.execute();
```

```
    // more calls to execute() possible
}
```

> **Warning.** Do *not* use prepared statements to insert large amounts of data into DuckDB. See the data import documentation for better options.

Arrow Methods

Refer to the API Reference for type signatures

Arrow Export

The following demonstrates exporting an arrow stream and consuming it using the java arrow bindings

```
import org.apache.arrow.memory.RootAllocator;
import org.apache.arrow.vector.ipc.ArrowReader;
import org.duckdb.DuckDBResultSet;

try (var conn = DriverManager.getConnection("jdbc:duckdb:");
    var stmt = conn.prepareStatement("SELECT * FROM generate_series(2000)");
    var resultset = (DuckDBResultSet) stmt.executeQuery();
    var allocator = new RootAllocator()) {
    try (var reader = (ArrowReader) resultset.arrowExportStream(allocator, 256)) {
        while (reader.loadNextBatch()) {
            System.out.println(reader.getVectorSchemaRoot().getVector("generate_series"));
        }
    }
    stmt.close();
}
```

Arrow Import

The following demonstrates consuming an Arrow stream from the Java Arrow bindings.

```
import org.apache.arrow.memory.RootAllocator;
import org.apache.arrow.vector.ipc.ArrowReader;
import org.duckdb.DuckDBConnection;

// Arrow binding
try (var allocator = new RootAllocator();
     ArrowStreamReader reader = null; // should not be null of course
     var arrow_array_stream = ArrowArrayStream.allocateNew(allocator)) {
    Data.exportArrayStream(allocator, reader, arrow_array_stream);

    // DuckDB setup
    try (var conn = (DuckDBConnection) DriverManager.getConnection("jdbc:duckdb:")) {
        conn.registerArrowStream("asdf", arrow_array_stream);

        // run a query
        try (var stmt = conn.createStatement();
             var rs = (DuckDBResultSet) stmt.executeQuery("SELECT count(*) FROM asdf")) {
            while (rs.next()) {
                System.out.println(rs.getInt(1));
            }
        }
    }
}
```

Streaming Results

Result streaming is opt-in in the JDBC driver – by setting the `jdbc_stream_results` config to `true` before running a query. The easiest way do that is to pass it in the `Properties` object.

```
Properties props = new Properties();
props.setProperty(DuckDBDriver.JDBC_STREAM_RESULTS, String.valueOf(true));

Connection conn = DriverManager.getConnection("jdbc:duckdb:", props);
```

Appender

The Appender is available in the DuckDB JDBC driver via the `org.duckdb.DuckDBAppender` class. The constructor of the class requires the schema name and the table name it is applied to. The Appender is flushed when the `close()` method is called.

Example:

```
import java.sql.DriverManager;
import java.sql.Statement;
import org.duckdb.DuckDBConnection;

DuckDBConnection conn = (DuckDBConnection) DriverManager.getConnection("jdbc:duckdb:");
try (var stmt = conn.createStatement()) {
    stmt.execute("CREATE TABLE tbl (x BIGINT, y FLOAT, s VARCHAR)"
);

// using try-with-resources to automatically close the appender at the end of the scope
try (var appender = conn.createAppender(DuckDBConnection.DEFAULT_SCHEMA, "tbl")) {
    appender.beginRow();
    appender.append(10);
    appender.append(3.2);
    appender.append("hello");
    appender.endRow();
    appender.beginRow();
    appender.append(20);
    appender.append(-8.1);
    appender.append("world");
    appender.endRow();
}
```

Batch Writer

The DuckDB JDBC driver offers batch write functionality. The batch writer supports prepared statements to mitigate the overhead of query parsing.

> The preferred method for bulk inserts is to use the Appender due to its higher performance. However, when using the Appender is not possbile, the batch writer is available as alternative.

Batch Writer with Prepared Statements

```
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import org.duckdb.DuckDBConnection;

DuckDBConnection conn = (DuckDBConnection) DriverManager.getConnection("jdbc:duckdb:");
PreparedStatement stmt = conn.prepareStatement("INSERT INTO test (x, y, z) VALUES (?, ?, ?);");
```

```
stmt.setObject(1, 1);
stmt.setObject(2, 2);
stmt.setObject(3, 3);
stmt.addBatch();

stmt.setObject(1, 4);
stmt.setObject(2, 5);
stmt.setObject(3, 6);
stmt.addBatch();

stmt.executeBatch();
stmt.close();
```

Batch Writer with Vanilla Statements

The batch writer also supports vanilla SQL statements:

```
import java.sql.DriverManager;
import java.sql.Statement;
import org.duckdb.DuckDBConnection;

DuckDBConnection conn = (DuckDBConnection) DriverManager.getConnection("jdbc:duckdb:");
Statement stmt = conn.createStatement();

stmt.execute("CREATE TABLE test (x INTEGER, y INTEGER, z INTEGER)");

stmt.addBatch("INSERT INTO test (x, y, z) VALUES (1, 2, 3);");
stmt.addBatch("INSERT INTO test (x, y, z) VALUES (4, 5, 6);");

stmt.executeBatch();
stmt.close();
```

Troubleshooting

Driver Class Not Found

If the Java application is unable to find the DuckDB, it may throw the following error:

```
Exception in thread "main" java.sql.SQLException: No suitable driver found for jdbc:duckdb:
    at java.sql/java.sql.DriverManager.getConnection(DriverManager.java:706)
    at java.sql/java.sql.DriverManager.getConnection(DriverManager.java:252)
    ...
```

And when trying to load the class manually, it may result in this error:

```
Exception in thread "main" java.lang.ClassNotFoundException: org.duckdb.DuckDBDriver
    at java.base/jdk.internal.loader.BuiltinClassLoader.loadClass(BuiltinClassLoader.java:641)
    at java.base/jdk.internal.loader.ClassLoaders$AppClassLoader.loadClass(ClassLoaders.java:188)
    at java.base/java.lang.ClassLoader.loadClass(ClassLoader.java:520)
    at java.base/java.lang.Class.forName0(Native Method)
    at java.base/java.lang.Class.forName(Class.java:375)
    ...
```

These errors stem from the DuckDB Maven/Gradle dependency not being detected. To ensure that it is detected, force refresh the Maven configuration in your IDE.

Julia Client

The DuckDB Julia package provides a high-performance front-end for DuckDB. Much like SQLite, DuckDB runs in-process within the Julia client, and provides a DBInterface front-end.

The package also supports multi-threaded execution. It uses Julia threads/tasks for this purpose. If you wish to run queries in parallel, you must launch Julia with multi-threading support (by e.g., setting the `JULIA_NUM_THREADS` environment variable).

Installation

Install DuckDB as follows:

```
using Pkg
Pkg.add("DuckDB")
```

Alternatively, enter the package manager using the `]` key, and issue the following command:

```
pkg> add DuckDB
```

Basics

```
using DuckDB

# create a new in-memory database
con = DBInterface.connect(DuckDB.DB, ":memory:")

# create a table
DBInterface.execute(con, "CREATE TABLE integers (i INTEGER)")

# insert data by executing a prepared statement
stmt = DBInterface.prepare(con, "INSERT INTO integers VALUES(?)")
DBInterface.execute(stmt, [42])

# query the database
results = DBInterface.execute(con, "SELECT 42 a")
print(results)
```

Some SQL statements, such as PIVOT and IMPORT DATABASE are executed as multiple prepared statements and will error when using `DuckDB.execute()`. Instead they can be run with `DuckDB.query()` instead of `DuckDB.execute()` and will always return a materialized result.

Scanning DataFrames

The DuckDB Julia package also provides support for querying Julia DataFrames. Note that the DataFrames are directly read by DuckDB – they are not inserted or copied into the database itself.

If you wish to load data from a DataFrame into a DuckDB table you can run a `CREATE TABLE ... AS` or `INSERT INTO` query.

```julia
using DuckDB
using DataFrames

# create a new in-memory dabase
con = DBInterface.connect(DuckDB.DB)

# create a DataFrame
df = DataFrame(a = [1, 2, 3], b = [42, 84, 42])

# register it as a view in the database
DuckDB.register_data_frame(con, df, "my_df")

# run a SQL query over the DataFrame
results = DBInterface.execute(con, "SELECT * FROM my_df")
print(results)
```

Appender API

The DuckDB Julia package also supports the Appender API, which is much faster than using prepared statements or individual `INSERT INTO` statements. Appends are made in row-wise format. For every column, an `append()` call should be made, after which the row should be finished by calling `flush()`. After all rows have been appended, `close()` should be used to finalize the Appender and clean up the resulting memory.

```julia
using DuckDB, DataFrames, Dates
db = DuckDB.DB()
# create a table
DBInterface.execute(db,
    "CREATE OR REPLACE TABLE data (id INTEGER PRIMARY KEY, value FLOAT, timestamp TIMESTAMP, date DATE)")
# create data to insert
len = 100
df = DataFrames.DataFrame(
        id = collect(1:len),
        value = rand(len),
        timestamp = Dates.now() + Dates.Second.(1:len),
        date = Dates.today() + Dates.Day.(1:len)
    )
# append data by row
appender = DuckDB.Appender(db, "data")
for i in eachrow(df)
    for j in i
        DuckDB.append(appender, j)
    end
    DuckDB.end_row(appender)
end
# close the appender after all rows
DuckDB.close(appender)
```

Concurrency

Within a Julia process, tasks are able to concurrently read and write to the database, as long as each task maintains its own connection to the database. In the example below, a single task is spawned to periodically read the database and many tasks are spawned to write to the database using both `INSERT` statements as well as the Appender API.

```julia
using Dates, DataFrames, DuckDB
db = DuckDB.DB()
DBInterface.connect(db)
```

```
DBInterface.execute(db, "CREATE OR REPLACE TABLE data (date TIMESTAMP, id INTEGER)")

function run_reader(db)
    # create a DuckDB connection specifically for this task
    conn = DBInterface.connect(db)
    while true
        println(DBInterface.execute(conn,
                "SELECT id, count(date) AS count, max(date) AS max_date
                FROM data GROUP BY id ORDER BY id") |> DataFrames.DataFrame)
        Threads.sleep(1)
    end
    DBInterface.close(conn)
end
# spawn one reader task
Threads.@spawn run_reader(db)

function run_inserter(db, id)
    # create a DuckDB connection specifically for this task
    conn = DBInterface.connect(db)
    for i in 1:1000
        Threads.sleep(0.01)
        DuckDB.execute(conn, "INSERT INTO data VALUES (current_timestamp, ?)"; id);
    end
    DBInterface.close(conn)
end
# spawn many insert tasks
for i in 1:100
    Threads.@spawn run_inserter(db, 1)
end

function run_appender(db, id)
    # create a DuckDB connection specifically for this task
    appender = DuckDB.Appender(db, "data")
    for i in 1:1000
        Threads.sleep(0.01)
        row = (Dates.now(Dates.UTC), id)
        for j in row
            DuckDB.append(appender, j);
        end
        DuckDB.end_row(appender);
    end
    DuckDB.close(appender);
end
# spawn many appender tasks
for i in 1:100
    Threads.@spawn run_appender(db, 2)
end
```

Original Julia Connector

Credits to kimmolinna for the original DuckDB Julia connector.

Node.js (Deprecated)

Node.js API

> The latest version of the DuckDB Node.js (deprecated) client is {{ site.current_duckdb_nodejs_version }}.

> **Deprecated.** The old DuckDB Node.js package is deprecated. Please use the DuckDB Node Neo package instead.

The duckdb package provides a Node.js API for DuckDB. The API for this client is somewhat compliant to the SQLite Node.js client for easier transition.

Initializing

Load the package and create a database object:

```
const duckdb = require('duckdb');
const db = new duckdb.Database(':memory:'); // or a file name for a persistent DB
```

All options as described on Database configuration can be (optionally) supplied to the `Database` constructor as second argument. The third argument can be optionally supplied to get feedback on the given options.

```
const db = new duckdb.Database(':memory:', {
    "access_mode": "READ_WRITE",
    "max_memory": "512MB",
    "threads": "4"
}, (err) => {
  if (err) {
    console.error(err);
  }
});
```

Running a Query

The following code snippet runs a simple query using the `Database.all()` method.

```
db.all('SELECT 42 AS fortytwo', function(err, res) {
  if (err) {
    console.warn(err);
    return;
  }
  console.log(res[0].fortytwo)
});
```

Other available methods are `each`, where the callback is invoked for each row, `run` to execute a single statement without results and `exec`, which can execute several SQL commands at once but also does not return results. All those commands can work with prepared statements, taking the values for the parameters as additional arguments. For example like so:

```
db.all('SELECT ?::INTEGER AS fortytwo, ?::VARCHAR AS hello', 42, 'Hello, World', function(err, res) {
  if (err) {
    console.warn(err);
    return;
  }
  console.log(res[0].fortytwo)
  console.log(res[0].hello)
});
```

Connections

A database can have multiple `Connections`, those are created using `db.connect()`.

```
const con = db.connect();
```

You can create multiple connections, each with their own transaction context.

`Connection` objects also contain shorthands to directly call `run()`, `all()` and `each()` with parameters and callbacks, respectively, for example:

```
con.all('SELECT 42 AS fortytwo', function(err, res) {
  if (err) {
    console.warn(err);
    return;
  }
  console.log(res[0].fortytwo)
});
```

Prepared Statements

From connections, you can create prepared statements (and only that) using `con.prepare()`:

```
const stmt = con.prepare('SELECT ?::INTEGER AS fortytwo');
```

To execute this statement, you can call for example `all()` on the `stmt` object:

```
stmt.all(42, function(err, res) {
  if (err) {
    console.warn(err);
  } else {
    console.log(res[0].fortytwo)
  }
});
```

You can also execute the prepared statement multiple times. This is for example useful to fill a table with data:

```
con.run('CREATE TABLE a (i INTEGER)');
const stmt = con.prepare('INSERT INTO a VALUES (?)');
for (let i = 0; i < 10; i++) {
  stmt.run(i);
}
stmt.finalize();
con.all('SELECT * FROM a', function(err, res) {
  if (err) {
    console.warn(err);
  } else {
    console.log(res)
  }
});
```

`prepare()` can also take a callback which gets the prepared statement as an argument:

```
const stmt = con.prepare('SELECT ?::INTEGER AS fortytwo', function(err, stmt) {
  stmt.all(42, function(err, res) {
    if (err) {
      console.warn(err);
    } else {
      console.log(res[0].fortytwo)
    }
  });
});
```

Inserting Data via Arrow

Apache Arrow can be used to insert data into DuckDB without making a copy:

```
const arrow = require('apache-arrow');
const db = new duckdb.Database(':memory:');

const jsonData = [
  {"userId":1,"id":1,"title":"delectus aut autem","completed":false},
  {"userId":1,"id":2,"title":"quis ut nam facilis et officia qui","completed":false}
];

// note; doesn't work on Windows yet
db.exec(` INSTALL arrow; LOAD arrow;`, (err) => {
    if (err) {
        console.warn(err);
        return;
    }

    const arrowTable = arrow.tableFromJSON(jsonData);
    db.register_buffer("jsonDataTable", [arrow.tableToIPC(arrowTable)], true, (err, res) => {
        if (err) {
            console.warn(err);
            return;
        }

        // `SELECT * FROM jsonDataTable` would return the entries in `jsonData`
    });
});
```

Loading Unsigned Extensions

To load unsigned extensions, instantiate the database as follows:

```
db = new duckdb.Database(':memory:', {"allow_unsigned_extensions": "true"});
```

Node.js API

Modules

Typedefs

duckdb

Summary: DuckDB is an embeddable SQL OLAP Database Management System

- duckdb
  - ~Connection
    - .run(sql, ...params, callback) ⇒ void
    - .all(sql, ...params, callback) ⇒ void
    - .arrowIPCAll(sql, ...params, callback) ⇒ void
    - .arrowIPCStream(sql, ...params, callback) ⇒
    - .each(sql, ...params, callback) ⇒ void
    - .stream(sql, ...params)
    - .register_udf(name, return_type, fun) ⇒ void
    - .prepare(sql, ...params, callback) ⇒ Statement
    - .exec(sql, ...params, callback) ⇒ void
    - .register_udf_bulk(name, return_type, callback) ⇒ void
    - .unregister_udf(name, return_type, callback) ⇒ void
    - .register_buffer(name, array, force, callback) ⇒ void
    - .unregister_buffer(name, callback) ⇒ void
    - .close(callback) ⇒ void
  - ~Statement
    - .sql ⇒
    - .get()
    - .run(sql, ...params, callback) ⇒ void
    - .all(sql, ...params, callback) ⇒ void
    - .arrowIPCAll(sql, ...params, callback) ⇒ void
    - .each(sql, ...params, callback) ⇒ void
    - .finalize(sql, ...params, callback) ⇒ void
    - .stream(sql, ...params)
    - .columns() ⇒ Array.<ColumnInfo>
  - ~QueryResult
    - .nextChunk() ⇒
    - .nextIpcBuffer() ⇒
    - .asyncIterator()
  - ~Database
    - .close(callback) ⇒ void
    - .close_internal(callback) ⇒ void
    - .wait(callback) ⇒ void
    - .serialize(callback) ⇒ void
    - .parallelize(callback) ⇒ void
    - .connect(path) ⇒ Connection
    - .interrupt(callback) ⇒ void
    - .prepare(sql) ⇒ Statement
    - .run(sql, ...params, callback) ⇒ void

        - .scanArrowIpc(sql, ...params, callback) ⇒ void
        - .each(sql, ...params, callback) ⇒ void
        - .stream(sql, ...params)
        - .all(sql, ...params, callback) ⇒ void
        - .arrowIPCAll(sql, ...params, callback) ⇒ void
        - .arrowIPCStream(sql, ...params, callback) ⇒ void
        - .exec(sql, ...params, callback) ⇒ void
        - .register_udf(name, return_type, fun) ⇒ this
        - .register_buffer(name) ⇒ this
        - .unregister_buffer(name) ⇒ this
        - .unregister_udf(name) ⇒ this
        - .registerReplacementScan(fun) ⇒ this
        - .tokenize(text) ⇒ ScriptTokens
        - .get()
    - ~TokenType
    - ~ERROR : number
    - ~OPEN_READONLY : number
    - ~OPEN_READWRITE : number
    - ~OPEN_CREATE : number
    - ~OPEN_FULLMUTEX : number
    - ~OPEN_SHAREDCACHE : number
    - ~OPEN_PRIVATECACHE : number

duckdb~Connection

Kind: inner class of duckdb

- ~Connection
    - .run(sql, ...params, callback) ⇒ void
    - .all(sql, ...params, callback) ⇒ void
    - .arrowIPCAll(sql, ...params, callback) ⇒ void
    - .arrowIPCStream(sql, ...params, callback) ⇒
    - .each(sql, ...params, callback) ⇒ void
    - .stream(sql, ...params)
    - .register_udf(name, return_type, fun) ⇒ void
    - .prepare(sql, ...params, callback) ⇒ Statement
    - .exec(sql, ...params, callback) ⇒ void
    - .register_udf_bulk(name, return_type, callback) ⇒ void
    - .unregister_udf(name, return_type, callback) ⇒ void
    - .register_buffer(name, array, force, callback) ⇒ void
    - .unregister_buffer(name, callback) ⇒ void
    - .close(callback) ⇒ void

connection.run(sql, ...params, callback) ⇒ void

Run a SQL statement and trigger a callback when done

Kind: instance method of Connection

Param	Type
sql	

Param	Type
...params	*
callback	

connection.all(sql, ...params, callback) ⇒ void

Run a SQL query and triggers the callback once for all result rows

Kind: instance method of Connection

Param	Type
sql	
...params	*
callback	

connection.arrowIPCAll(sql, ...params, callback) ⇒ void

Run a SQL query and serialize the result into the Apache Arrow IPC format (requires arrow extension to be loaded)

Kind: instance method of Connection

Param	Type
sql	
...params	*
callback	

connection.arrowIPCStream(sql, ...params, callback) ⇒

Run a SQL query, returns a IpcResultStreamIterator that allows streaming the result into the Apache Arrow IPC format (requires arrow extension to be loaded)

Kind: instance method of Connection
Returns: Promise

Param	Type
sql	
...params	*
callback	

connection.each(sql, ...params, callback) ⇒ void

Runs a SQL query and triggers the callback for each result row

Kind: instance method of Connection

Param	Type
sql	
...params	*
callback	

connection.stream(sql, ...params)

Kind: instance method of Connection

Param	Type
sql	
...params	*

connection.register_udf(name, return_type, fun) ⇒ void

Register a User Defined Function

Kind: instance method of Connection
Note: this follows the wasm udfs somewhat but is simpler because we can pass data much more cleanly

Param
name
return_type
fun

connection.prepare(sql, ...params, callback) ⇒ Statement

Prepare a SQL query for execution

Kind: instance method of Connection

Param	Type
sql	
...params	*
callback	

connection.exec(sql, ...params, callback) ⇒ void

Execute a SQL query

Kind: instance method of Connection

Param	Type
sql	
...params	*
callback	

connection.register_udf_bulk(name, return_type, callback) ⇒ void

Register a User Defined Function

Kind: instance method of Connection

Param
name
return_type
callback

connection.unregister_udf(name, return_type, callback) ⇒ void

Unregister a User Defined Function

Kind: instance method of Connection

Param
name
return_type
callback

connection.register_buffer(name, array, force, callback) ⇒ void

Register a Buffer to be scanned using the Apache Arrow IPC scanner (requires arrow extension to be loaded)

Kind: instance method of Connection

Param
name
array
force
callback

connection.unregister_buffer(name, callback) ⇒ void

Unregister the Buffer

Kind: instance method of Connection

Param
name
callback

connection.close(callback) ⇒ void

Closes connection

Kind: instance method of Connection

Param
callback

duckdb~Statement

Kind: inner class of duckdb

- ~Statement
  - .sql ⇒
  - .get()
  - .run(sql, ...params, callback) ⇒ void
  - .all(sql, ...params, callback) ⇒ void
  - .arrowIPCAll(sql, ...params, callback) ⇒ void
  - .each(sql, ...params, callback) ⇒ void
  - .finalize(sql, ...params, callback) ⇒ void
  - .stream(sql, ...params)
  - .columns() ⇒ Array.<ColumnInfo>

statement.sql ⇒

Kind: instance property of Statement
Returns: sql contained in statement
Field:

statement.get()

Not implemented

Kind: instance method of Statement

statement.run(sql, ...params, callback) ⇒ void

Kind: instance method of Statement

Param	Type
sql	
...params	*
callback	

statement.all(sql, ...params, callback) ⇒ void

Kind: instance method of Statement

Param	Type
sql	
...params	*
callback	

statement.arrowIPCAll(sql, ...params, callback) ⇒ void

Kind: instance method of Statement

Param	Type
sql	
...params	*
callback	

statement.each(sql, ...params, callback) ⇒ void

Kind: instance method of Statement

Param	Type
sql	
...params	*
callback	

statement.finalize(sql, ...params, callback) ⇒ void

Kind: instance method of Statement

Param	Type
sql	
...params	*
callback	

statement.stream(sql, ...params)

Kind: instance method of Statement

Param	Type
sql	
...params	*

statement.columns() ⇒ Array.<ColumnInfo>

Kind: instance method of Statement
Returns: Array.<ColumnInfo> - - Array of column names and types

duckdb~QueryResult

Kind: inner class of duckdb

- ~QueryResult
  - .nextChunk() ⇒
  - .nextIpcBuffer() ⇒
  - .asyncIterator()

queryResult.nextChunk() ⇒

Kind: instance method of QueryResult
Returns: data chunk

queryResult.nextIpcBuffer() ⇒

Function to fetch the next result blob of an Arrow IPC Stream in a zero-copy way. (requires arrow extension to be loaded)

Kind: instance method of QueryResult
Returns: data chunk

queryResult.asyncIterator()

Kind: instance method of QueryResult

duckdb~Database

Main database interface

Kind: inner property of duckdb

Param	Description
path	path to database file or :memory: for in-memory database
access_mode	access mode
config	the configuration object
callback	callback function

- ~Database
  - .close(callback) ⇒ void
  - .close_internal(callback) ⇒ void
  - .wait(callback) ⇒ void
  - .serialize(callback) ⇒ void
  - .parallelize(callback) ⇒ void
  - .connect(path) ⇒ Connection
  - .interrupt(callback) ⇒ void
  - .prepare(sql) ⇒ Statement
  - .run(sql, ...params, callback) ⇒ void
  - .scanArrowIpc(sql, ...params, callback) ⇒ void
  - .each(sql, ...params, callback) ⇒ void
  - .stream(sql, ...params)
  - .all(sql, ...params, callback) ⇒ void
  - .arrowIPCAll(sql, ...params, callback) ⇒ void
  - .arrowIPCStream(sql, ...params, callback) ⇒ void
  - .exec(sql, ...params, callback) ⇒ void
  - .register_udf(name, return_type, fun) ⇒ this
  - .register_buffer(name) ⇒ this
  - .unregister_buffer(name) ⇒ this
  - .unregister_udf(name) ⇒ this
  - .registerReplacementScan(fun) ⇒ this
  - .tokenize(text) ⇒ ScriptTokens
  - .get()

database.close(callback) ⇒ void

Closes database instance

Kind: instance method of Database

Param
callback

database.close_internal(callback) ⇒ void

Internal method. Do not use, call Connection#close instead

Kind: instance method of Database

Param
callback

database.wait(callback) ⇒ void

Triggers callback when all scheduled database tasks have completed.

Kind: instance method of Database

Param
callback

database.serialize(callback) ⇒ void

Currently a no-op. Provided for SQLite compatibility

Kind: instance method of Database

Param
callback

database.parallelize(callback) ⇒ void

Currently a no-op. Provided for SQLite compatibility

Kind: instance method of Database

Param
callback

database.connect(path) ⇒ Connection

Create a new database connection

Kind: instance method of Database

Param	Description
path	the database to connect to, either a file path, or `:memory:`

database.interrupt(callback) ⇒ void

Supposedly interrupt queries, but currently does not do anything.

Kind: instance method of Database

Param
callback

database.prepare(sql) ⇒ Statement

Prepare a SQL query for execution

Kind: instance method of Database

Param
sql

database.run(sql, ...params, callback) ⇒ void

Convenience method for Connection#run using a built-in default connection

Kind: instance method of Database

Param	Type
sql	
...params	*
callback	

database.scanArrowIpc(sql, ...params, callback) ⇒ void

Convenience method for Connection#scanArrowIpc using a built-in default connection

Kind: instance method of Database

Param	Type
sql	
...params	*
callback	

database.each(sql, ...params, callback) ⇒ void

Kind: instance method of Database

Param	Type
sql	
...params	*
callback	

database.stream(sql, ...params)

Kind: instance method of Database

Param	Type
sql	
...params	*

database.all(sql, ...params, callback) ⇒ void

Convenience method for Connection#apply using a built-in default connection

Kind: instance method of Database

Param	Type
sql	
...params	*
callback	

database.arrowIPCAll(sql, ...params, callback) ⇒ void

Convenience method for Connection#arrowIPCAll using a built-in default connection

Kind: instance method of Database

Param	Type
sql	
...params	*
callback	

database.arrowIPCStream(sql, ...params, callback) ⇒ void

Convenience method for Connection#arrowIPCStream using a built-in default connection

Kind: instance method of Database

Param	Type
sql	
...params	*
callback	

database.exec(sql, ...params, callback) ⇒ void

Kind: instance method of Database

Param	Type
sql	
...params	*

Param	Type
callback	

database.register_udf(name, return_type, fun) ⇒ this

Register a User Defined Function

Convenience method for Connection#register_udf

Kind: instance method of Database

Param
name
return_type
fun

database.register_buffer(name) ⇒ this

Register a buffer containing serialized data to be scanned from DuckDB.

Convenience method for Connection#unregister_buffer

Kind: instance method of Database

Param
name

database.unregister_buffer(name) ⇒ this

Unregister a Buffer

Convenience method for Connection#unregister_buffer

Kind: instance method of Database

Param
name

database.unregister_udf(name) ⇒ this

Unregister a UDF

Convenience method for Connection#unregister_udf

Kind: instance method of Database

Param
name

database.registerReplacementScan(fun) ⇒ this

Register a table replace scan function

Kind: instance method of Database

Param	Description
fun	Replacement scan function

database.tokenize(text) ⇒ ScriptTokens

Return positions and types of tokens in given text

Kind: instance method of Database

Param
text

database.get()

Not implemented

Kind: instance method of Database

duckdb~TokenType

Types of tokens return by `tokenize`.

Kind: inner property of duckdb

duckdb~ERROR : number

Check that errno attribute equals this to check for a duckdb error

Kind: inner constant of duckdb

duckdb~OPEN_READONLY : number

Open database in readonly mode

Kind: inner constant of duckdb

duckdb~OPEN_READWRITE : number

Currently ignored

Kind: inner constant of duckdb

duckdb~OPEN_CREATE : number

Currently ignored

Kind: inner constant of duckdb

duckdb~OPEN_FULLMUTEX : number

Currently ignored

Kind: inner constant of duckdb

duckdb~OPEN_SHAREDCACHE : number

Currently ignored

Kind: inner constant of duckdb

duckdb~OPEN_PRIVATECACHE : number

Currently ignored

Kind: inner constant of duckdb

ColumnInfo : object

Kind: global typedef
Properties

Name	Type	Description
name	string	Column name
type	TypeInfo	Column type

TypeInfo : object

Kind: global typedef
Properties

Name	Type	Description
id	string	Type ID
[alias]	string	SQL type alias
sql_type	string	SQL type name

DuckDbError : object

Kind: global typedef
Properties

Name	Type	Description
errno	number	-1 for DuckDB errors
message	string	Error message
code	string	'DUCKDB_NODEJS_ERROR' for DuckDB errors
errorType	string	DuckDB error type code (eg, HTTP, IO, Catalog)

HTTPError : object

Kind: global typedef
Extends: DuckDbError
Properties

Name	Type	Description
statusCode	number	HTTP response status code
reason	string	HTTP response reason
response	string	HTTP response body
headers	object	HTTP headers

Node.js (Neo)

Node.js Client (Neo)

> The latest version of the DuckDB Node.js (Neo) client is {{ site.current_duckdb_node_neo_version }}.

An API for using DuckDB in Node.js.

The primary package, @duckdb/node-api, is a high-level API meant for applications. It depends on low-level bindings that adhere closely to DuckDB's C API, available separately as @duckdb/node-bindings.

Features

Main Differences from duckdb-node

- Native support for Promises; no need for separate duckdb-async wrapper.
- DuckDB-specific API; not based on the SQLite Node API.
- Lossless & efficent support for values of all DuckDB data types.
- Wraps released DuckDB binaries instead of rebuilding DuckDB.
- Built on DuckDB's C API; exposes more functionality.

Roadmap

Some features are not yet complete:

- Binding and appending the MAP and UNION data types
- Appending default values row-by-row
- User-defined types & functions
- Profiling info
- Table description
- APIs for Arrow

See the issues list on GitHub for the most up-to-date roadmap.

Supported Platforms

- Linux arm64
- Linux x64
- Mac OS X (Darwin) arm64 (Apple Silicon)
- Mac OS X (Darwin) x64 (Intel)
- Windows (Win32) x64

Examples

Get Basic Information

```
import duckdb from '@duckdb/node-api';

console.log(duckdb.version());

console.log(duckdb.configurationOptionDescriptions());
```

Connect

```
import { DuckDBConnection } from '@duckdb/node-api';

const connection = await DuckDBConnection.create();
```

This uses the default instance. For advanced usage, you can create instances explicitly.

Create Instance

```
import { DuckDBInstance } from '@duckdb/node-api';
```

Create with an in-memory database:

```
const instance = await DuckDBInstance.create(':memory:');
```

Equivalent to the above:

```
const instance = await DuckDBInstance.create();
```

Read from and write to a database file, which is created if needed:

```
const instance = await DuckDBInstance.create('my_duckdb.db');
```

Set configuration options:

```
const instance = await DuckDBInstance.create('my_duckdb.db', {
  threads: '4'
});
```

Instance Cache

Multiple instances in the same process should not attach the same database.

To prevent this, an instance cache can be used:

```
const instance = await DuckDBInstance.fromCache('my_duckdb.db');
```

This uses the default instance cache. For advanced usage, you can create instance caches explicitly:

```
import { DuckDBInstanceCache } from '@duckdb/node-api';

const cache = new DuckDBInstanceCache();
const instance = await cache.getOrCreateInstance('my_duckdb.db');
```

Connect to Instance

```
const connection = await instance.connect();
```

Disconnect

Connections will be disconnected automatically soon after their reference is dropped, but you can also disconnect explicitly if and when you want:

```
connection.disconnectSync();
```

or, equivalently:

```
connection.closeSync();
```

Run SQL

```
const result = await connection.run('from test_all_types()');
```

Parameterize SQL

```
const prepared = await connection.prepare('select $1, $2, $3');
prepared.bindVarchar(1, 'duck');
prepared.bindInteger(2, 42);
prepared.bindList(3, listValue([10, 11, 12]), LIST(INTEGER));
const result = await prepared.run();
```

or:

```
const prepared = await connection.prepare('select $a, $b, $c');
prepared.bind({
  'a': 'duck',
  'b': 42,
  'c': listValue([10, 11, 12]),
}, {
  'a': VARCHAR,
  'b': INTEGER,
  'c': LIST(INTEGER),
});
const result = await prepared.run();
```

or even:

```
const result = await connection.run('select $a, $b, $c', {
  'a': 'duck',
  'b': 42,
  'c': listValue([10, 11, 12]),
}, {
  'a': VARCHAR,
  'b': INTEGER,
  'c': LIST(INTEGER),
});
```

Unspecified types will be inferred:

```
const result = await connection.run('select $a, $b, $c', {
  'a': 'duck',
  'b': 42,
  'c': listValue([10, 11, 12]),
});
```

Specifying Values

Values of many data types are represented using one of the JS primitives `boolean`, `number`, `bigint`, or `string`. Also, any type can have `null` values.

Values of some data types need to be constructed using special functions. These are:

Type	Function
ARRAY	arrayValue
BIT	bitValue
BLOB	blobValue
DATE	dateValue
DECIMAL	decimalValue
INTERVAL	intervalValue
LIST	listValue
MAP	mapValue
STRUCT	structValue
TIME	timeValue
TIMETZ	timeTZValue
TIMESTAMP	timestampValue
TIMESTAMPTZ	timestampTZValue
TIMESTAMP_S	timestampSecondsValue
TIMESTAMP_MS	timestampMillisValue
TIMESTAMP_NS	timestampNanosValue
UNION	unionValue
UUID	uuidValue

Stream Results

Streaming results evaluate lazily when rows are read.

```
const result = await connection.stream('from range(10_000)');
```

Inspect Result Metadata

Get column names and types:

```
const columnNames = result.columnNames();
const columnTypes = result.columnTypes();
```

Read Result Data

Run and read all data:

```
const reader = await connection.runAndReadAll('from test_all_types()');
const rows = reader.getRows();
// OR: const columns = reader.getColumns();
```

Stream and read up to (at least) some number of rows:

```
const reader = await connection.streamAndReadUntil(
  'from range(5000)',
  1000
);
const rows = reader.getRows();
// rows.length === 2048. (Rows are read in chunks of 2048.)
```

Read rows incrementally:

```
const reader = await connection.streamAndRead('from range(5000)');
reader.readUntil(2000);
// reader.currentRowCount === 2048 (Rows are read in chunks of 2048.)
// reader.done === false
reader.readUntil(4000);
// reader.currentRowCount === 4096
// reader.done === false
reader.readUntil(6000);
// reader.currentRowCount === 5000
// reader.done === true
```

Get Result Data

Result data can be retrieved in a variety of forms:

```
const reader = await connection.runAndReadAll(
  'from range(3) select range::int as i, 10 + i as n'
);

const rows = reader.getRows();
// [ [0, 10], [1, 11], [2, 12] ]

const rowObjects = reader.getRowObjects();
// [ { i: 0, n: 10 }, { i: 1, n: 11 }, { i: 2, n: 12 } ]

const columns = reader.getColumns();
// [ [0, 1, 2], [10, 11, 12] ]

const columnsObject = reader.getColumnsObject();
// { i: [0, 1, 2], n: [10, 11, 12] }
```

Convert Result Data

By default, data values that cannot be represented as JS built-ins are returned as specialized JS objects; see `Inspect Data Values` below.

To retrieve data in a different form, such as JS built-ins or values that can be losslessly serialized to JSON, use the `JS` or `Json` forms of the above result data methods.

Custom converters can be supplied as well. See the implementations of JSDuckDBValueConverter and JsonDuckDBValueConverters for how to do this.

Examples (using the `Json` forms):

```
const reader = await connection.runAndReadAll(
  'from test_all_types() select bigint, date, interval limit 2'
);
```

```
const rows = reader.getRowsJson();
// [
//   [
//     "-9223372036854775808",
//     "5877642-06-25 (BC)",
//     { "months": 0, "days": 0, "micros": "0" }
//   ],
//   [
//     "9223372036854775807",
//     "5881580-07-10",
//     { "months": 999, "days": 999, "micros": "999999999" }
//   ]
// ]

const rowObjects = reader.getRowObjectsJson();
// [
//   {
//     "bigint": "-9223372036854775808",
//     "date": "5877642-06-25 (BC)",
//     "interval": { "months": 0, "days": 0, "micros": "0" }
//   },
//   {
//     "bigint": "9223372036854775807",
//     "date": "5881580-07-10",
//     "interval": { "months": 999, "days": 999, "micros": "999999999" }
//   }
// ]

const columns = reader.getColumnsJson();
// [
//   [ "-9223372036854775808", "9223372036854775807" ],
//   [ "5877642-06-25 (BC)", "5881580-07-10" ],
//   [
//     { "months": 0, "days": 0, "micros": "0" },
//     { "months": 999, "days": 999, "micros": "999999999" }
//   ]
// ]

const columnsObject = reader.getColumnsObjectJson();
// {
//   "bigint": [ "-9223372036854775808", "9223372036854775807" ],
//   "date": [ "5877642-06-25 (BC)", "5881580-07-10" ],
//   "interval": [
//     { "months": 0, "days": 0, "micros": "0" },
//     { "months": 999, "days": 999, "micros": "999999999" }
//   ]
// }
```

These methods handle nested types as well:

```
const reader = await connection.runAndReadAll(
  'from test_all_types() select int_array, struct, map, "union" limit 2'
);

const rows = reader.getRowsJson();
// [
//   [
//     [],
//     { "a": null, "b": null },
//     [],
```

```
//     { "tag": "name", "value": "Frank" }
//   ],
//   [
//     [ 42, 999, null, null, -42],
//     { "a": 42, "b": "🦆🦆🦆🦆🦆🦆" },
//     [
//       { "key": "key1", "value": "🦆🦆🦆🦆🦆🦆" },
//       { "key": "key2", "value": "goose" }
//     ],
//     { "tag": "age", "value": 5 }
//   ]
// ]

const rowObjects = reader.getRowObjectsJson();
// [
//   {
//     "int_array": [],
//     "struct": { "a": null, "b": null },
//     "map": [],
//     "union": { "tag": "name", "value": "Frank" }
//   },
//   {
//     "int_array": [ 42, 999, null, null, -42 ],
//     "struct": { "a": 42, "b": "🦆🦆🦆🦆🦆🦆" },
//     "map": [
//       { "key": "key1", "value": "🦆🦆🦆🦆🦆🦆" },
//       { "key": "key2", "value": "goose" }
//     ],
//     "union": { "tag": "age", "value": 5 }
//   }
// ]

const columns = reader.getColumnsJson();
// [
//   [
//     [],
//     [42, 999, null, null, -42]
//   ],
//   [
//     { "a": null, "b": null },
//     { "a": 42, "b": "🦆🦆🦆🦆🦆🦆" }
//   ],
//   [
//     [],
//     [
//       { "key": "key1", "value": "🦆🦆🦆🦆🦆🦆" },
//       { "key": "key2", "value": "goose"}
//     ]
//   ],
//   [
//     { "tag": "name", "value": "Frank" },
//     { "tag": "age", "value": 5 }
//   ]
// ]

const columnsObject = reader.getColumnsObjectJson();
// {
//   "int_array": [
//     [],
//     [42, 999, null, null, -42]
```

```
//   ],
//   "struct": [
//     { "a": null, "b": null },
//     { "a": 42, "b": "🦆🦆🦆🦆🦆🦆" }
//   ],
//   "map": [
//     [],
//     [
//       { "key": "key1", "value": "🦆🦆🦆🦆🦆🦆" },
//       { "key": "key2", "value": "goose" }
//     ]
//   ],
//   "union": [
//     { "tag": "name", "value": "Frank" },
//     { "tag": "age", "value": 5 }
//   ]
// }
```

Column names and types can also be serialized to JSON:

```
const columnNamesAndTypes = reader.columnNamesAndTypesJson();
// {
//   "columnNames": [
//     "int_array",
//     "struct",
//     "map",
//     "union"
//   ],
//   "columnTypes": [
//     {
//       "typeId": 24,
//       "valueType": {
//         "typeId": 4
//       }
//     },
//     {
//       "typeId": 25,
//       "entryNames": [
//         "a",
//         "b"
//       ],
//       "entryTypes": [
//         {
//           "typeId": 4
//         },
//         {
//           "typeId": 17
//         }
//       ]
//     },
//     {
//       "typeId": 26,
//       "keyType": {
//         "typeId": 17
//       },
//       "valueType": {
//         "typeId": 17
//       }
//     },
//     {
```

```
//       "typeId": 28,
//       "memberTags": [
//         "name",
//         "age"
//       ],
//       "memberTypes": [
//         {
//           "typeId": 17
//         },
//         {
//           "typeId": 3
//         }
//       ]
//     }
//   ]
// }

const columnNameAndTypeObjects = reader.columnNameAndTypeObjectsJson();
// [
//   {
//     "columnName": "int_array",
//     "columnType": {
//       "typeId": 24,
//       "valueType": {
//         "typeId": 4
//       }
//     }
//   },
//   {
//     "columnName": "struct",
//     "columnType": {
//       "typeId": 25,
//       "entryNames": [
//         "a",
//         "b"
//       ],
//       "entryTypes": [
//         {
//           "typeId": 4
//         },
//         {
//           "typeId": 17
//         }
//       ]
//     }
//   },
//   {
//     "columnName": "map",
//     "columnType": {
//       "typeId": 26,
//       "keyType": {
//         "typeId": 17
//       },
//       "valueType": {
//         "typeId": 17
//       }
//     }
//   },
//   {
//     "columnName": "union",
```

```
//     "columnType": {
//       "typeId": 28,
//       "memberTags": [
//         "name",
//         "age"
//       ],
//       "memberTypes": [
//         {
//           "typeId": 17
//         },
//         {
//           "typeId": 3
//         }
//       ]
//     }
//   }
// ]
```

Fetch Chunks

Fetch all chunks:

```
const chunks = await result.fetchAllChunks();
```

Fetch one chunk at a time:

```
const chunks = [];
while (true) {
  const chunk = await result.fetchChunk();
  // Last chunk will have zero rows.
  if (chunk.rowCount === 0) {
    break;
  }
  chunks.push(chunk);
}
```

For materialized (non-streaming) results, chunks can be read by index:

```
const rowCount = result.rowCount;
const chunkCount = result.chunkCount;
for (let i = 0; i < chunkCount; i++) {
  const chunk = result.getChunk(i);
  // ...
}
```

Get chunk data:

```
const rows = chunk.getRows();

const rowObjects = chunk.getRowObjects(result.deduplicatedColumnNames());

const columns = chunk.getColumns();

const columnsObject =
  chunk.getColumnsObject(result.deduplicatedColumnNames());
```

Get chunk data (one value at a time)

```
const columns = [];
const columnCount = chunk.columnCount;
for (let columnIndex = 0; columnIndex < columnCount; columnIndex++) {
```

```
  const columnValues = [];
  const columnVector = chunk.getColumnVector(columnIndex);
  const itemCount = columnVector.itemCount;
  for (let itemIndex = 0; itemIndex < itemCount; itemIndex++) {
    const value = columnVector.getItem(itemIndex);
    columnValues.push(value);
  }
  columns.push(columnValues);
}
```

Inspect Data Types

```
import { DuckDBTypeId } from '@duckdb/node-api';

if (columnType.typeId === DuckDBTypeId.ARRAY) {
  const arrayValueType = columnType.valueType;
  const arrayLength = columnType.length;
}

if (columnType.typeId === DuckDBTypeId.DECIMAL) {
  const decimalWidth = columnType.width;
  const decimalScale = columnType.scale;
}

if (columnType.typeId === DuckDBTypeId.ENUM) {
  const enumValues = columnType.values;
}

if (columnType.typeId === DuckDBTypeId.LIST) {
  const listValueType = columnType.valueType;
}

if (columnType.typeId === DuckDBTypeId.MAP) {
  const mapKeyType = columnType.keyType;
  const mapValueType = columnType.valueType;
}

if (columnType.typeId === DuckDBTypeId.STRUCT) {
  const structEntryNames = columnType.names;
  const structEntryTypes = columnType.valueTypes;
}

if (columnType.typeId === DuckDBTypeId.UNION) {
  const unionMemberTags = columnType.memberTags;
  const unionMemberTypes = columnType.memberTypes;
}

// For the JSON type (https://duckdb.org/docs/data/json/json_type)
if (columnType.alias === 'JSON') {
  const json = JSON.parse(columnValue);
}
```

Every type implements toString. The result is both human-friendly and readable by DuckDB in an appropriate expression.

```
const typeString = columnType.toString();
```

Inspect Data Values

```
import { DuckDBTypeId } from '@duckdb/node-api';
```

```
if (columnType.typeId === DuckDBTypeId.ARRAY) {
  const arrayItems = columnValue.items; // array of values
  const arrayString = columnValue.toString();
}

if (columnType.typeId === DuckDBTypeId.BIT) {
  const bools = columnValue.toBools(); // array of booleans
  const bits = columnValue.toBits(); // arrary of 0s and 1s
  const bitString = columnValue.toString(); // string of '0's and '1's
}

if (columnType.typeId === DuckDBTypeId.BLOB) {
  const blobBytes = columnValue.bytes; // Uint8Array
  const blobString = columnValue.toString();
}

if (columnType.typeId === DuckDBTypeId.DATE) {
  const dateDays = columnValue.days;
  const dateString = columnValue.toString();
  const { year, month, day } = columnValue.toParts();
}

if (columnType.typeId === DuckDBTypeId.DECIMAL) {
  const decimalWidth = columnValue.width;
  const decimalScale = columnValue.scale;
  // Scaled-up value. Represented number is value/(10^scale).
  const decimalValue = columnValue.value; // bigint
  const decimalString = columnValue.toString();
  const decimalDouble = columnValue.toDouble();
}

if (columnType.typeId === DuckDBTypeId.INTERVAL) {
  const intervalMonths = columnValue.months;
  const intervalDays = columnValue.days;
  const intervalMicros = columnValue.micros; // bigint
  const intervalString = columnValue.toString();
}

if (columnType.typeId === DuckDBTypeId.LIST) {
  const listItems = columnValue.items; // array of values
  const listString = columnValue.toString();
}

if (columnType.typeId === DuckDBTypeId.MAP) {
  const mapEntries = columnValue.entries; // array of { key, value }
  const mapString = columnValue.toString();
}

if (columnType.typeId === DuckDBTypeId.STRUCT) {
  // { name1: value1, name2: value2, ... }
  const structEntries = columnValue.entries;
  const structString = columnValue.toString();
}

if (columnType.typeId === DuckDBTypeId.TIMESTAMP_MS) {
  const timestampMillis = columnValue.milliseconds; // bigint
  const timestampMillisString = columnValue.toString();
}
```

```
if (columnType.typeId === DuckDBTypeId.TIMESTAMP_NS) {
  const timestampNanos = columnValue.nanoseconds; // bigint
  const timestampNanosString = columnValue.toString();
}

if (columnType.typeId === DuckDBTypeId.TIMESTAMP_S) {
  const timestampSecs = columnValue.seconds; // bigint
  const timestampSecsString = columnValue.toString();
}

if (columnType.typeId === DuckDBTypeId.TIMESTAMP_TZ) {
  const timestampTZMicros = columnValue.micros; // bigint
  const timestampTZString = columnValue.toString();
  const {
    date: { year, month, day },
    time: { hour, min, sec, micros },
  } = columnValue.toParts();
}

if (columnType.typeId === DuckDBTypeId.TIMESTAMP) {
  const timestampMicros = columnValue.micros; // bigint
  const timestampString = columnValue.toString();
  const {
    date: { year, month, day },
    time: { hour, min, sec, micros },
  } = columnValue.toParts();
}

if (columnType.typeId === DuckDBTypeId.TIME_TZ) {
  const timeTZMicros = columnValue.micros; // bigint
  const timeTZOffset = columnValue.offset;
  const timeTZString = columnValue.toString();
  const {
    time: { hour, min, sec, micros },
    offset,
  } = columnValue.toParts();
}

if (columnType.typeId === DuckDBTypeId.TIME) {
  const timeMicros = columnValue.micros; // bigint
  const timeString = columnValue.toString();
  const { hour, min, sec, micros } = columnValue.toParts();
}

if (columnType.typeId === DuckDBTypeId.UNION) {
  const unionTag = columnValue.tag;
  const unionValue = columnValue.value;
  const unionValueString = columnValue.toString();
}

if (columnType.typeId === DuckDBTypeId.UUID) {
  const uuidHugeint = columnValue.hugeint; // bigint
  const uuidString = columnValue.toString();
}

// other possible values are: null, boolean, number, bigint, or string
```

Displaying Timezones

Converting a TIMESTAMP_TZ value to a string depends on a timezone offset. By default, this is set to the offset for the local timezone when the Node process is started.

To change it, set the `timezoneOffsetInMinutes` property of `DuckDBTimestampTZValue`:

```
DuckDBTimestampTZValue.timezoneOffsetInMinutes = -8 * 60;
const pst = DuckDBTimestampTZValue.Epoch.toString();
// 1969-12-31 16:00:00-08

DuckDBTimestampTZValue.timezoneOffsetInMinutes = +1 * 60;
const cet = DuckDBTimestampTZValue.Epoch.toString();
// 1970-01-01 01:00:00+01
```

Note that the timezone offset used for this string conversion is distinct from the `TimeZone` setting of DuckDB.

The following sets this offset to match the `TimeZone` setting of DuckDB:

```
const reader = await connection.runAndReadAll(
  `select (timezone(current_timestamp) / 60)::int`
);
DuckDBTimestampTZValue.timezoneOffsetInMinutes =
  reader.getColumns()[0][0];
```

Append To Table

```
await connection.run(
  `create or replace table target_table(i integer, v varchar)`
);

const appender = await connection.createAppender('target_table');

appender.appendInteger(42);
appender.appendVarchar('duck');
appender.endRow();

appender.appendInteger(123);
appender.appendVarchar('mallard');
appender.endRow();

appender.flushSync();

appender.appendInteger(17);
appender.appendVarchar('goose');
appender.endRow();

appender.closeSync(); // also flushes
```

Append Data Chunk

```
await connection.run(
  `create or replace table target_table(i integer, v varchar)`
);

const appender = await connection.createAppender('target_table');

const chunk = DuckDBDataChunk.create([INTEGER, VARCHAR]);
chunk.setColumns([
```

```
  [42, 123, 17],
  ['duck', 'mallad', 'goose'],
]);
// OR:
// chunk.setRows([
//   [42, 'duck'],
//   [123, 'mallard'],
//   [17, 'goose'],
// ]);

appender.appendDataChunk(chunk);
appender.flushSync();
```

See "Specifying Values" above for how to supply values to the appender.

Extract Statements

```
const extractedStatements = await connection.extractStatements(`
  create or replace table numbers as from range(?);
  from numbers where range < ?;
  drop table numbers;
`);
const parameterValues = [10, 7];
const statementCount = extractedStatements.count;
for (let stmtIndex = 0; stmtIndex < statementCount; stmtIndex++) {
  const prepared = await extractedStatements.prepare(stmtIndex);
  let parameterCount = prepared.parameterCount;
  for (let paramIndex = 1; paramIndex <= parameterCount; paramIndex++) {
    prepared.bindInteger(paramIndex, parameterValues.shift());
  }
  const result = await prepared.run();
  // ...
}
```

Control Evaluation of Tasks

```
import { DuckDBPendingResultState } from '@duckdb/node-api';

async function sleep(ms) {
  return new Promise((resolve) => {
    setTimeout(resolve, ms);
  });
}

const prepared = await connection.prepare('from range(10_000_000)');
const pending = prepared.start();
while (pending.runTask() !== DuckDBPendingResultState.RESULT_READY) {
  console.log('not ready');
  await sleep(1);
}
console.log('ready');
const result = await pending.getResult();
// ...
```

Ways to Run SQL

```
// Run to completion but don't yet retrieve any rows.
```

```
// Optionally take values to bind to SQL parameters,
// and (optionally) types of those parameters,
// either as an array (for positional parameters),
// or an object keyed by parameter name.
const result = await connection.run(sql);
const result = await connection.run(sql, values);
const result = await connection.run(sql, values, types);

// Run to completion but don't yet retrieve any rows.
// Wrap in a DuckDBDataReader for convenient data retrieval.
const reader = await connection.runAndRead(sql);
const reader = await connection.runAndRead(sql, values);
const reader = await connection.runAndRead(sql, values, types);

// Run to completion, wrap in a reader, and read all rows.
const reader = await connection.runAndReadAll(sql);
const reader = await connection.runAndReadAll(sql, values);
const reader = await connection.runAndReadAll(sql, values, types);

// Run to completion, wrap in a reader, and read at least
// the given number of rows. (Rows are read in chunks, so more than
// the target may be read.)
const reader = await connection.runAndReadUntil(sql, targetRowCount);
const reader =
  await connection.runAndReadAll(sql, targetRowCount, values);
const reader =
  await connection.runAndReadAll(sql, targetRowCount, values, types);

// Create a streaming result and don't yet retrieve any rows.
const result = await connection.stream(sql);
const result = await connection.stream(sql, values);
const result = await connection.stream(sql, values, types);

// Create a streaming result and don't yet retrieve any rows.
// Wrap in a DuckDBDataReader for convenient data retrieval.
const reader = await connection.streamAndRead(sql);
const reader = await connection.streamAndRead(sql, values);
const reader = await connection.streamAndRead(sql, values, types);

// Create a streaming result, wrap in a reader, and read all rows.
const reader = await connection.streamAndReadAll(sql);
const reader = await connection.streamAndReadAll(sql, values);
const reader = await connection.streamAndReadAll(sql, values, types);

// Create a streaming result, wrap in a reader, and read at least
// the given number of rows.
const reader = await connection.streamAndReadUntil(sql, targetRowCount);
const reader =
  await connection.streamAndReadUntil(sql, targetRowCount, values);
const reader =
  await connection.streamAndReadUntil(sql, targetRowCount, values, types);

// Prepared Statements

// Prepare a possibly-parametered SQL statement to run later.
const prepared = await connection.prepare(sql);

// Bind values to the parameters.
prepared.bind(values);
prepared.bind(values, types);
```

```
// Run the prepared statement. These mirror the methods on the connection.
const result = prepared.run();

const reader = prepared.runAndRead();
const reader = prepared.runAndReadAll();
const reader = prepared.runAndReadUntil(targetRowCount);

const result = prepared.stream();

const reader = prepared.streamAndRead();
const reader = prepared.streamAndReadAll();
const reader = prepared.streamAndReadUntil(targetRowCount);

// Pending Results

// Create a pending result.
const pending = await connection.start(sql);
const pending = await connection.start(sql, values);
const pending = await connection.start(sql, values, types);

// Create a pending, streaming result.
const pending = await connection.startStream(sql);
const pending = await connection.startStream(sql, values);
const pending = await connection.startStream(sql, values, types);

// Create a pending result from a prepared statement.
const pending = await prepared.start();
const pending = await prepared.startStream();

while (pending.runTask() !== DuckDBPendingResultState.RESULT_READY) {
  // optionally sleep or do other work between tasks
}

// Retrieve the result. If not yet READY, will run until it is.
const result = await pending.getResult();

const reader = await pending.read();
const reader = await pending.readAll();
const reader = await pending.readUntil(targetRowCount);
```

Ways to Get Result Data

```
// From a result

// Asynchronously retrieve data for all rows:
const columns = await result.getColumns();
const columnsJson = await result.getColumnsJson();
const columnsObject = await result.getColumnsObject();
const columnsObjectJson = await result.getColumnsObjectJson();
const rows = await result.getRows();
const rowsJson = await result.getRowsJson();
const rowObjects = await result.getRowObjects();
const rowObjectsJson = await result.getRowObjectsJson();

// From a reader

// First, (asynchronously) read some rows:
await reader.readAll();
```

```
// or:
await reader.readUntil(targetRowCount);

// Then, (synchronously) get result data for the rows read:
const columns = reader.getColumns();
const columnsJson = reader.getColumnsJson();
const columnsObject = reader.getColumnsObject();
const columnsObjectJson = reader.getColumnsObjectJson();
const rows = reader.getRows();
const rowsJson = reader.getRowsJson();
const rowObjects = reader.getRowObjects();
const rowObjectsJson = reader.getRowObjectsJson();

// Individual values can also be read directly:
const value = reader.value(columnIndex, rowIndex);

// Using chunks

// If desired, one or more chunks can be fetched from a result:
const chunk = await result.fetchChunk();
const chunks = await result.fetchAllChunks();

// And then data can be retrieved from each chunk:
const columnValues = chunk.getColumnValues(columnIndex);
const columns = chunk.getColumns();
const rowValues = chunk.getRowValues(rowIndex);
const rows = chunk.getRows();

// Or, values can be visited:
chunk.visitColumnValues(columnIndex,
  (value, rowIndex, columnIndex, type) => { /* ... */ }
);
chunk.visitColumns((column, columnIndex, type) => { /* ... */ });
chunk.visitColumnMajor(
  (value, rowIndex, columnIndex, type) => { /* ... */ }
);
chunk.visitRowValues(rowIndex,
  (value, rowIndex, columnIndex, type) => { /* ... */ }
);
chunk.visitRows((row, rowIndex) => { /* ... */ });
chunk.visitRowMajor(
  (value, rowIndex, columnIndex, type) => { /* ... */ }
);

// Or converted:
// The `converter` argument implements `DuckDBValueConverter`,
// which has the single method convertValue(value, type).
const columnValues = chunk.convertColumnValues(columnIndex, converter);
const columns = chunk.convertColumns(converter);
const rowValues = chunk.convertRowValues(rowIndex, converter);
const rows = chunk.convertRows(converter);

// The reader abstracts these low-level chunk manipulations
// and is recommended for most cases.
```

ODBC

ODBC API Overview

> The latest version of the DuckDB ODBC client is {{ site.current_duckdb_odbc_short_version }}.

The ODBC (Open Database Connectivity) is a C-style API that provides access to different flavors of Database Management Systems (DBMSs). The ODBC API consists of the Driver Manager (DM) and the ODBC drivers.

The Driver Manager is part of the system library, e.g., unixODBC, which manages the communications between the user applications and the ODBC drivers. Typically, applications are linked against the DM, which uses Data Source Name (DSN) to look up the correct ODBC driver.

The ODBC driver is a DBMS implementation of the ODBC API, which handles all the internals of that DBMS.

The DM maps user application calls of ODBC functions to the correct ODBC driver that performs the specified function and returns the proper values.

DuckDB ODBC Driver

DuckDB supports the ODBC version 3.0 according to the Core Interface Conformance.

The ODBC driver is available for all operating systems. Visit the installation page for direct links.

ODBC API on Linux

Driver Manager

A driver manager is required to manage communication between applications and the ODBC driver. We tested and support `unixODBC` that is a complete ODBC driver manager for Linux. Users can install it from the command line:

On Debian-based distributions (Ubuntu, Mint, etc.), run:

```
sudo apt-get install unixodbc odbcinst
```

On Fedora-based distributions (Amazon Linux, RHEL, CentOS, etc.), run:

```
sudo yum install unixODBC
```

Setting Up the Driver

1. Download the ODBC Linux Asset corresponding to your architecture:
   - x86_64 (AMD64)
   - arm64 (AArch64)
2. The package contains the following files:
   - `libduckdb_odbc.so`: the DuckDB driver.

- `unixodbc_setup.sh`: a setup script to aid the configuration on Linux.

To extract them, run:

```
mkdir duckdb_odbc && unzip duckdb_odbc-linux-amd64.zip -d duckdb_odbc
```

3. The `unixodbc_setup.sh` script performs the configuration of the DuckDB ODBC Driver. It is based on the unixODBC package that provides some commands to handle the ODBC setup and test like `odbcinst` and `isql`.

   Run the following commands with either option `-u` or `-s` to configure DuckDB ODBC.

   The `-u` option based on the user home directory to setup the ODBC init files.

   ```
   ./unixodbc_setup.sh -u
   ```

   The `-s` option changes the system level files that will be visible for all users, because of that it requires root privileges.

   ```
   sudo ./unixodbc_setup.sh -s
   ```

   The option `--help` shows the usage of `unixodbc_setup.sh` prints the help.

   ```
   ./unixodbc_setup.sh --help
   ```

   ```
   Usage: ./unixodbc_setup.sh <level> [options]

   Example: ./unixodbc_setup.sh -u -db ~/database_path -D ~/driver_path/libduckdb_odbc.so

   Level:
   -s: System-level, using 'sudo' to configure DuckDB ODBC at the system-level, changing the files:
   /etc/odbc[inst].ini
   -u: User-level, configuring the DuckDB ODBC at the user-level, changing the files:
   ~/.odbc[inst].ini.

   Options:
   -db database_path>: the DuckDB database file path, the default is ':memory:' if not provided.
   -D driver_path: the driver file path (i.e., the path for libduckdb_odbc.so), the default is using
   the base script directory
   ```

4. The ODBC setup on Linux is based on the `.odbc.ini` and `.odbcinst.ini` files.

   These files can be placed to the user home directory `/home/<username>`{:.language-sql .highlight} or in the system `/etc`{:.language-sql .highlight} directory. The Driver Manager prioritizes the user configuration files over the system files.

   For the details of the configuration parameters, see the ODBC configuration page.

ODBC API on Windows

Using the DuckDB ODBC API on Windows requires the following steps:

1. The Microsoft Windows requires an ODBC Driver Manager to manage communication between applications and the ODBC drivers. The Driver Manager on Windows is provided in a DLL file `odbccp32.dll`, and other files and tools. For detailed information check out the Common ODBC Component Files.
2. DuckDB releases the ODBC driver as an asset. For Windows, download it from the Windows ODBC asset (x86_64/AMD64).
3. The archive contains the following artifacts:

   - `duckdb_odbc.dll`: the DuckDB driver compiled for Windows.
   - `duckdb_odbc_setup.dll`: a setup DLL used by the Windows ODBC Data Source Administrator tool.
   - `odbc_install.exe`: an installation script to aid the configuration on Windows.

   Decompress the archive to a directory (e.g., `duckdb_odbc`).

4. The `odbc_install.exe` binary performs the configuration of the DuckDB ODBC Driver on Windows. It depends on the `Odbccp32.dll` that provides functions to configure the ODBC registry entries.

   Inside the permanent directory (e.g., `duckdb_odbc`), double-click on the `odbc_install.exe`.

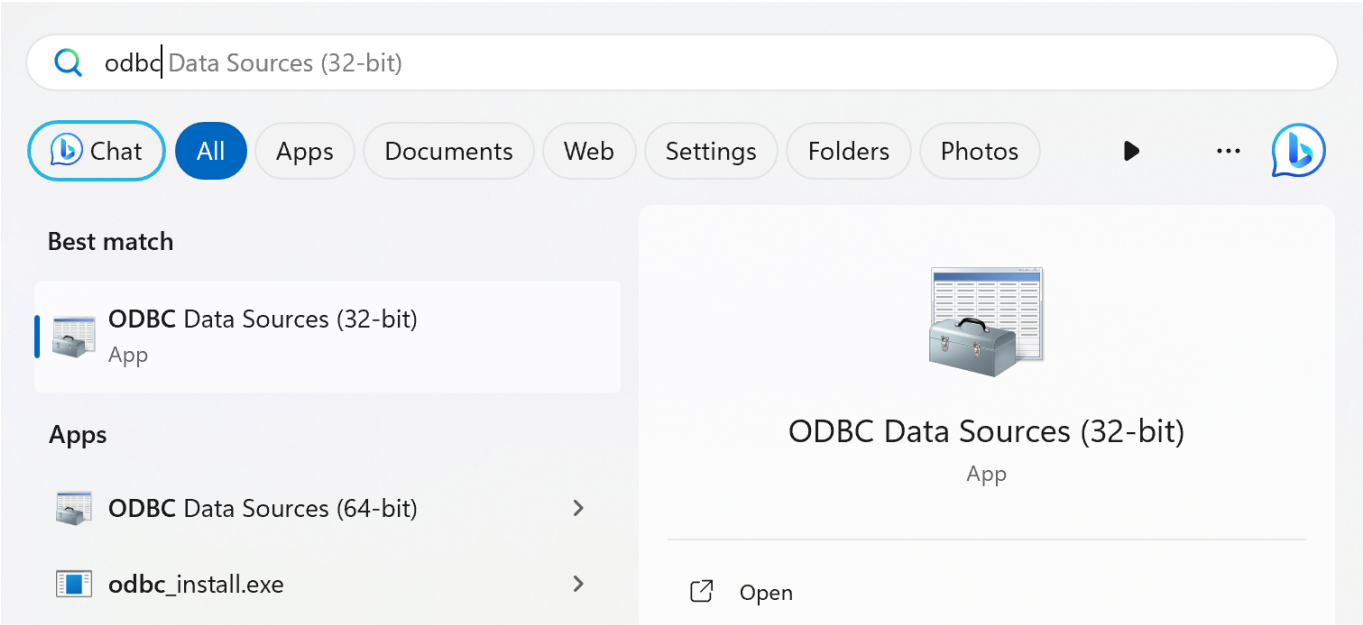
   Windows administrator privileges are required. In case of a non-administrator, a User Account Control prompt will occur.

5. `odbc_install.exe` adds a default DSN configuration into the ODBC registries with a default database `:memory:`.

DSN Windows Setup

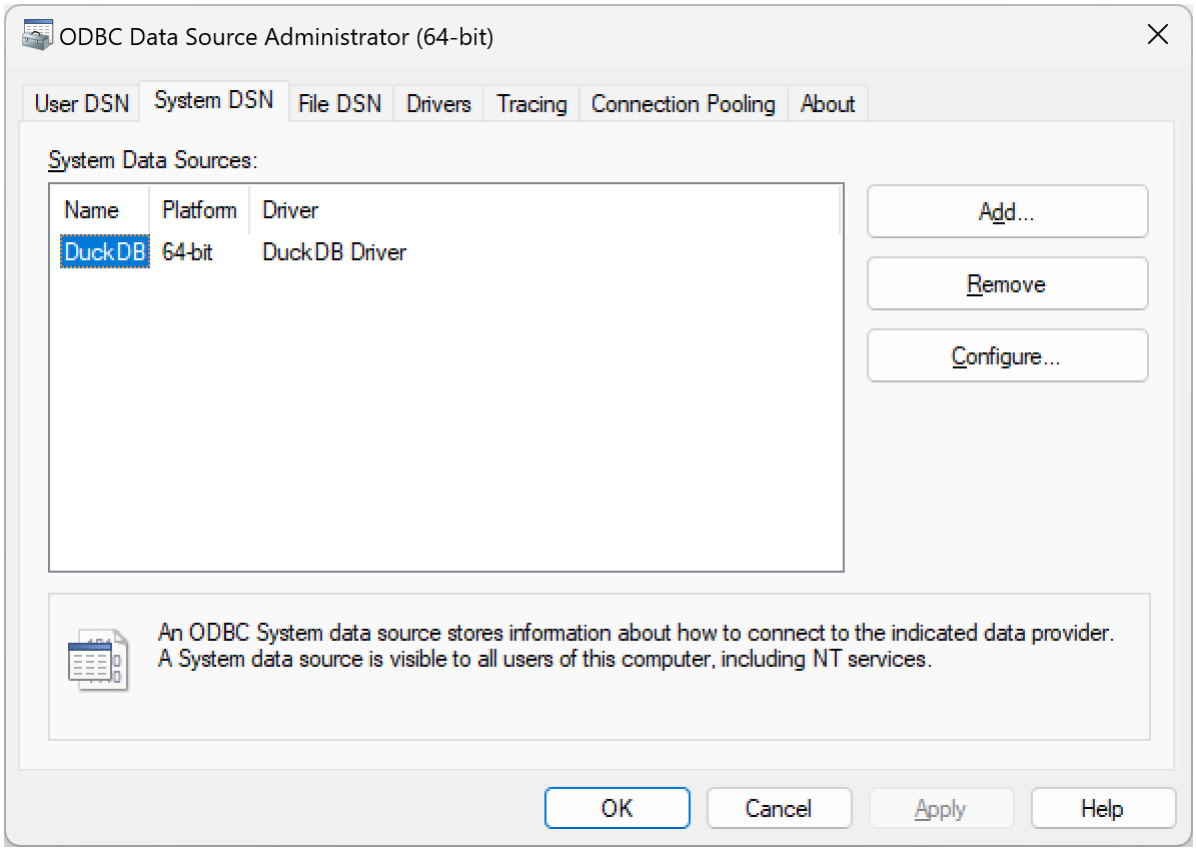
After the installation, it is possible to change the default DSN configuration or add a new one using the Windows ODBC Data Source Administrator tool `odbcad32.exe`.

It also can be launched thought the Windows start:



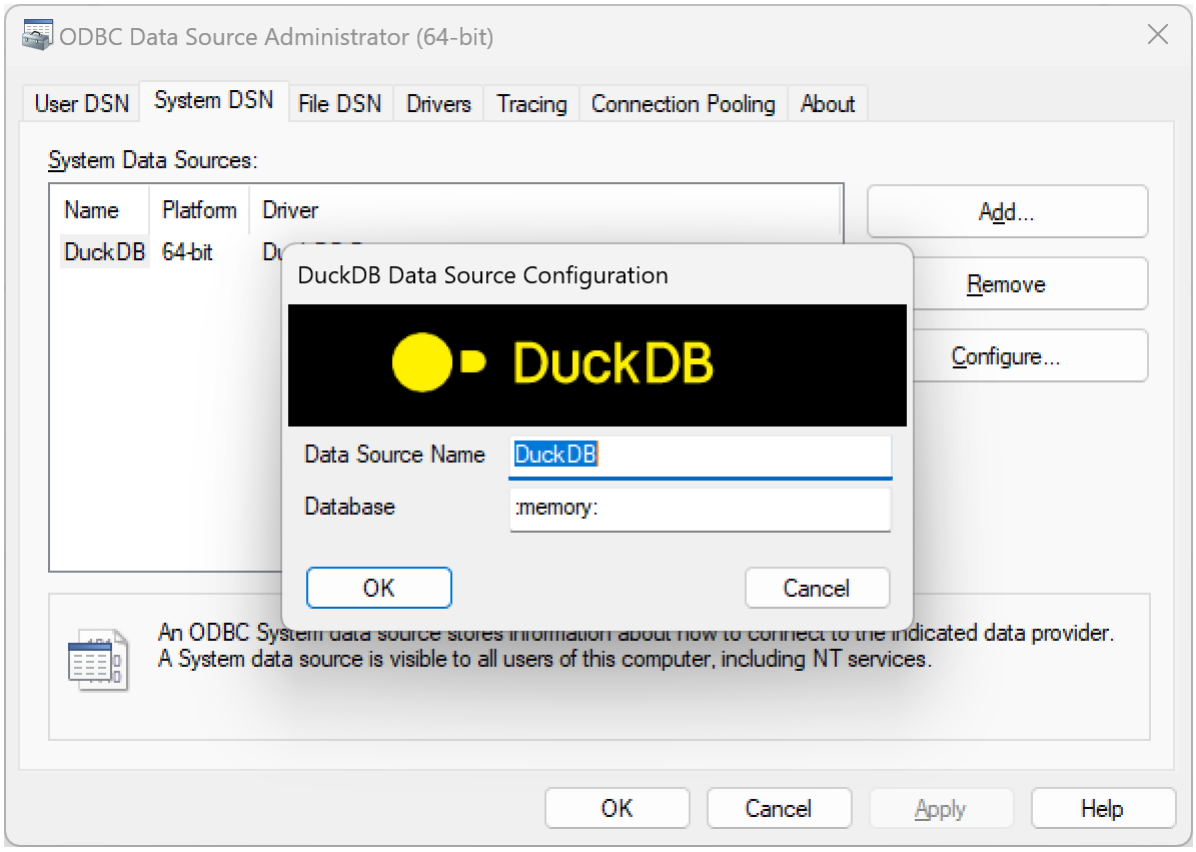
Default DuckDB DSN

The newly installed DSN is visible on the ***System DSN*** in the Windows ODBC Data Source Administrator tool:



Changing DuckDB DSN

When selecting the default DSN (i.e., DuckDB) or adding a new configuration, the following setup window will display:



This window allows you to set the DSN and the database file path associated with that DSN.

More Detailed Windows Setup

There are two ways to configure the ODBC driver, either by altering the registry keys as detailed below, or by connecting with `SQLDriverConnect`. A combination of the two is also possible.

Furthermore, the ODBC driver supports all the configuration options included in DuckDB.

> If a configuration is set in both the connection string passed to `SQLDriverConnect` and in the `odbc.ini` file, the one passed to `SQLDriverConnect` will take precedence.

For the details of the configuration parameters, see the ODBC configuration page.

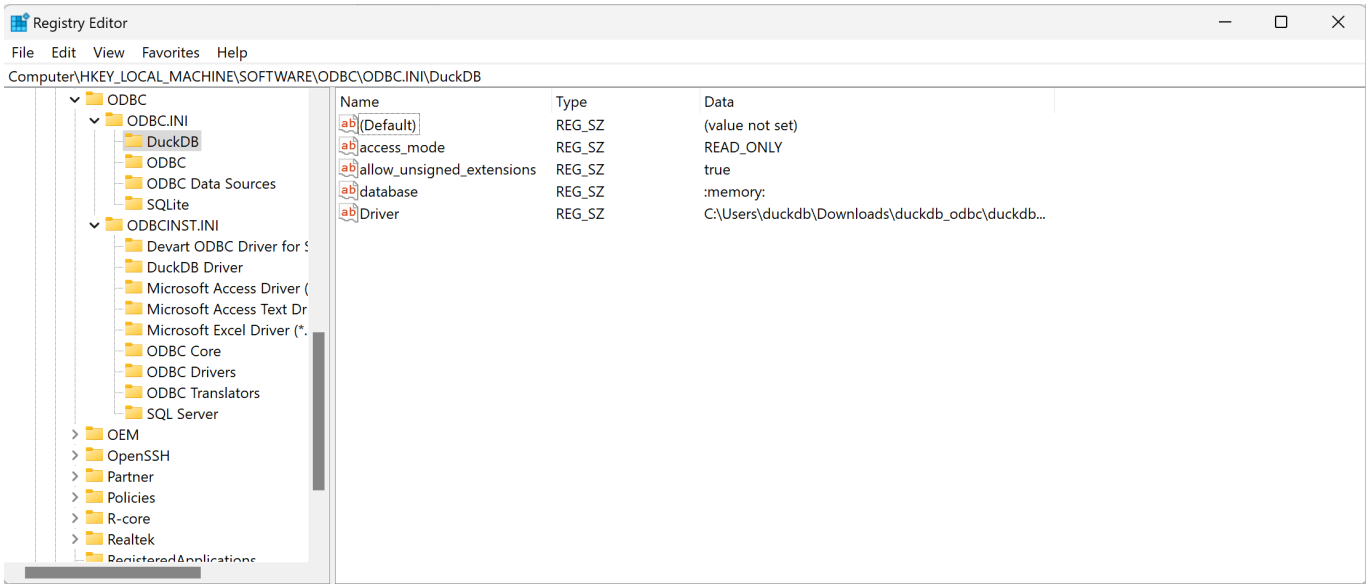
Registry Keys

The ODBC setup on Windows is based on registry keys (see Registry Entries for ODBC Components). The ODBC entries can be placed at the current user registry key (`HKCU`) or the system registry key (`HKLM`).

We have tested and used the system entries based on `HKLM->SOFTWARE->ODBC`. The `odbc_install.exe` changes this entry that has two subkeys: `ODBC.INI` and `ODBCINST.INI`.

The `ODBC.INI` is where users usually insert DSN registry entries for the drivers.

For example, the DSN registry for DuckDB would look like this:



The `ODBCINST.INI` contains one entry for each ODBC driver and other keys predefined for Windows ODBC configuration.

Updating the ODBC Driver

When a new version of the ODBC driver is released, installing the new version will overwrite the existing one. However, the installer doesn't always update the version number in the registry. To ensure the correct version is used, check that `HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBCINST.INI\DuckDB Driver` has the most recent version, and `HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBC.INI\DuckDB\Driver` has the correct path to the new driver.

ODBC API on macOS

1. A driver manager is required to manage communication between applications and the ODBC driver. DuckDB supports `unixODBC`, which is a complete ODBC driver manager for macOS and Linux. Users can install it from the command line via Homebrew:

   ```
   brew install unixodbc
   ```

2. DuckDB releases a universal ODBC driver for macOS (supporting both Intel and Apple Silicon CPUs). To download it, run:

```
wget https://github.com/duckdb/duckdb-odbc/releases/download/v1.3.1.0/duckdb_odbc-osx-universal.zip
```

1. The archive contains the `libduckdb_odbc.dylib` artifact. To extract it to a directory, run:

   ```
   mkdir duckdb_odbc && unzip duckdb_odbc-osx-universal.zip -d duckdb_odbc
   ```

2. There are two ways to configure the ODBC driver, either by initializing via the configuration files, or by connecting with `SQLDriverConnect`. A combination of the two is also possible.

   Furthermore, the ODBC driver supports all the configuration options included in DuckDB.

   > If a configuration is set in both the connection string passed to `SQLDriverConnect` and in the `odbc.ini` file, the one passed to `SQLDriverConnect` will take precedence.

   For the details of the configuration parameters, see the ODBC configuration page.

3. After the configuration, to validate the installation, it is possible to use an ODBC client. unixODBC uses a command line tool called `isql`.

   Use the DSN defined in `odbc.ini` as a parameter of `isql`.

```
isql DuckDB

+---------------------------------------+
| Connected!                            |
|                                       |
| sql-statement                         |
| help [tablename]                      |
| echo [string]                         |
| quit                                  |
|                                       |
+---------------------------------------+

SQL> SELECT 42;

+------------+
| 42         |
+------------+
| 42         |
+------------+

SQLRowCount returns -1
1 rows fetched
```

ODBC Configuration

This page documents the files using the ODBC configuration, `odbc.ini` and `odbcinst.ini`. These are either placed in the home directory as dotfiles (`.odbc.ini` and `.odbcinst.ini`, respectively) or in a system directory. For platform-specific details, see the pages for Linux, macOS, and Windows.

`odbc.ini` and `.odbc.ini`

The `odbc.ini` file contains the DSNs for the drivers, which can have specific knobs. An example of `odbc.ini` with DuckDB:

```
[DuckDB]
Driver = DuckDB Driver
Database = :memory:
access_mode = read_only
```

The lines correspond to the following parameters:

- `[DuckDB]`: between the brackets is a DSN for the DuckDB.
- `Driver`: Describes the driver's name, as well as where to find the configurations in the `odbcinst.ini`.
- `Database`: Describes the database name used by DuckDB, can also be a file path to a `.db` in the system.
- `access_mode`: The mode in which to connect to the database.

`odbcinst.ini` and `.odbcinst.ini`

The `odbcinst.ini` file contains general configurations for the ODBC installed drivers in the system. A driver section starts with the driver name between brackets, and then it follows specific configuration knobs belonging to that driver.

Example of `odbcinst.ini` with the DuckDB:

```
[ODBC]
Trace = yes
TraceFile = /tmp/odbctrace
```

```
[DuckDB Driver]
Driver = /path/to/libduckdb_odbc.dylib
```

The lines correspond to the following parameters:

- `[ODBC]`: The DM configuration section.
- `Trace`: Enables the ODBC trace file using the option `yes`.
- `TraceFile`: The absolute system file path for the ODBC trace file.
- `[DuckDB Driver]`: The section of the DuckDB installed driver.
- `Driver`: The absolute system file path of the DuckDB driver. Change to match your configuration.

Python

Python API

> The latest version of the DuckDB Python client is 1.3.1.

Installation

The DuckDB Python API can be installed using pip: `pip install duckdb`. Please see the installation page: `conda install python-duckdb -c conda-forge`.

Python version: DuckDB requires Python 3.9 or newer.

Basic API Usage

The most straight-forward manner of running SQL queries using DuckDB is using the `duckdb.sql` command.

```
import duckdb

duckdb.sql("SELECT 42").show()
```

This will run queries using an **in-memory database** that is stored globally inside the Python module. The result of the query is returned as a **Relation**. A relation is a symbolic representation of the query. The query is not executed until the result is fetched or requested to be printed to the screen.

Relations can be referenced in subsequent queries by storing them inside variables, and using them as tables. This way queries can be constructed incrementally.

```
import duckdb

r1 = duckdb.sql("SELECT 42 AS i")
duckdb.sql("SELECT i * 2 AS k FROM r1").show()
```

Data Input

DuckDB can ingest data from a wide variety of formats – both on-disk and in-memory. See the data ingestion page for more information.

```
import duckdb

duckdb.read_csv("example.csv")                # read a CSV file into a Relation
duckdb.read_parquet("example.parquet")        # read a Parquet file into a Relation
duckdb.read_json("example.json")              # read a JSON file into a Relation

duckdb.sql("SELECT * FROM 'example.csv'")     # directly query a CSV file
duckdb.sql("SELECT * FROM 'example.parquet'") # directly query a Parquet file
duckdb.sql("SELECT * FROM 'example.json'")    # directly query a JSON file
```

DataFrames

DuckDB can directly query Pandas DataFrames, Polars DataFrames and Arrow tables. Note that these are read-only, i.e., editing these tables via `INSERT` or `UPDATE` statements is not possible.

Pandas

To directly query a Pandas DataFrame, run:

```
import duckdb
import pandas as pd

pandas_df = pd.DataFrame({"a": [42]})
duckdb.sql("SELECT * FROM pandas_df")
```

```
┌─────────┐
│    a    │
│  int64  │
├─────────┤
│      42 │
└─────────┘
```

Polars

To directly query a Polars DataFrame, run:

```
import duckdb
import polars as pl

polars_df = pl.DataFrame({"a": [42]})
duckdb.sql("SELECT * FROM polars_df")
```

```
┌─────────┐
│    a    │
│  int64  │
├─────────┤
│      42 │
└─────────┘
```

PyArrow

To directly query a PyArrow table, run:

```
import duckdb
import pyarrow as pa

arrow_table = pa.Table.from_pydict({"a": [42]})
duckdb.sql("SELECT * FROM arrow_table")
```

```
┌─────────┐
│    a    │
│  int64  │
├─────────┤
│      42 │
└─────────┘
```

Result Conversion

DuckDB supports converting query results efficiently to a variety of formats. See the result conversion page for more information.

```
import duckdb

duckdb.sql("SELECT 42").fetchall()   # Python objects
duckdb.sql("SELECT 42").df()         # Pandas DataFrame
duckdb.sql("SELECT 42").pl()         # Polars DataFrame
duckdb.sql("SELECT 42").arrow()      # Arrow Table
duckdb.sql("SELECT 42").fetchnumpy() # NumPy Arrays
```

Writing Data to Disk

DuckDB supports writing Relation objects directly to disk in a variety of formats. The COPY statement can be used to write data to disk using SQL as an alternative.

```
import duckdb

duckdb.sql("SELECT 42").write_parquet("out.parquet") # Write to a Parquet file
duckdb.sql("SELECT 42").write_csv("out.csv")         # Write to a CSV file
duckdb.sql("COPY (SELECT 42) TO 'out.parquet'")      # Copy to a Parquet file
```

Connection Options

Applications can open a new DuckDB connection via the `duckdb.connect()` method.

Using an In-Memory Database

When using DuckDB through `duckdb.sql()`, it operates on an **in-memory** database, i.e., no tables are persisted on disk. Invoking the `duckdb.connect()` method without arguments returns a connection, which also uses an in-memory database:

```
import duckdb

con = duckdb.connect()
con.sql("SELECT 42 AS x").show()
```

Persistent Storage

The `duckdb.connect(dbname)` creates a connection to a **persistent** database. Any data written to that connection will be persisted, and can be reloaded by reconnecting to the same file, both from Python and from other DuckDB clients.

```
import duckdb

# create a connection to a file called 'file.db'
con = duckdb.connect("file.db")
# create a table and load data into it
con.sql("CREATE TABLE test (i INTEGER)")
con.sql("INSERT INTO test VALUES (42)")
# query the table
con.table("test").show()
# explicitly close the connection
con.close()
# Note: connections also closed implicitly when they go out of scope
```

You can also use a context manager to ensure that the connection is closed:

```
import duckdb

with duckdb.connect("file.db") as con:
    con.sql("CREATE TABLE test (i INTEGER)")
    con.sql("INSERT INTO test VALUES (42)")
    con.table("test").show()
    # the context manager closes the connection automatically
```

Configuration

The `duckdb.connect()` accepts a `config` dictionary, where configuration options can be specified. For example:

```
import duckdb

con = duckdb.connect(config = {'threads': 1})
```

Connection Object and Module

The connection object and the `duckdb` module can be used interchangeably – they support the same methods. The only difference is that when using the `duckdb` module a global in-memory database is used.

> If you are developing a package designed for others to use, and use DuckDB in the package, it is recommend that you create connection objects instead of using the methods on the `duckdb` module. That is because the `duckdb` module uses a shared global database – which can cause hard to debug issues if used from within multiple different packages.

Using Connections in Parallel Python Programs

The `DuckDBPyConnection` object is not thread-safe. If you would like to write to the same database from multiple threads, create a cursor for each thread with the `DuckDBPyConnection.cursor()` method.

Loading and Installing Extensions

DuckDB's Python API provides functions for installing and loading extensions, which perform the equivalent operations to running the `INSTALL` and `LOAD` SQL commands, respectively. An example that installs and loads the `spatial` extension looks like follows:

```
import duckdb

con = duckdb.connect()
con.install_extension("spatial")
con.load_extension("spatial")
```

Community Extensions

To load community extensions, use the `repository="community"` argument with the `install_extension` method.

For example, install and load the `h3` community extension as follows:

```
import duckdb

con = duckdb.connect()
con.install_extension("h3", repository="community")
con.load_extension("h3")
```

Unsigned Extensions

To load unsigned extensions, use the `config = {"allow_unsigned_extensions": "true"}` argument with the `duckdb.connect()` method.

Data Ingestion

This page contains examples for data ingestion to Python using DuckDB. First, import the DuckDB page:

```
import duckdb
```

Then, proceed with any of the following sections.

CSV Files

CSV files can be read using the `read_csv` function, called either from within Python or directly from within SQL. By default, the `read_csv` function attempts to auto-detect the CSV settings by sampling from the provided file.

Read from a file using fully auto-detected settings:

```
duckdb.read_csv("example.csv")
```

Read multiple CSV files from a folder:

```
duckdb.read_csv("folder/*.csv")
```

Specify options on how the CSV is formatted internally:

```
duckdb.read_csv("example.csv", header = False, sep = ",")
```

Override types of the first two columns:

```
duckdb.read_csv("example.csv", dtype = ["int", "varchar"])
```

Directly read a CSV file from within SQL:

```
duckdb.sql("SELECT * FROM 'example.csv'")
```

Call `read_csv` from within SQL:

```
duckdb.sql("SELECT * FROM read_csv('example.csv')")
```

See the CSV Import page for more information.

Parquet Files

Parquet files can be read using the `read_parquet` function, called either from within Python or directly from within SQL.

Read from a single Parquet file:

```
duckdb.read_parquet("example.parquet")
```

Read multiple Parquet files from a folder:

```
duckdb.read_parquet("folder/*.parquet")
```

Read a Parquet file over https:

```
duckdb.read_parquet("https://some.url/some_file.parquet")
```

Read a list of Parquet files:

```
duckdb.read_parquet(["file1.parquet", "file2.parquet", "file3.parquet"])
```

Directly read a Parquet file from within SQL:

```
duckdb.sql("SELECT * FROM 'example.parquet'")
```

Call `read_parquet` from within SQL:

```
duckdb.sql("SELECT * FROM read_parquet('example.parquet')")
```

See the Parquet Loading page for more information.

JSON Files

JSON files can be read using the `read_json` function, called either from within Python or directly from within SQL. By default, the `read_json` function will automatically detect if a file contains newline-delimited JSON or regular JSON, and will detect the schema of the objects stored within the JSON file.

Read from a single JSON file:

```
duckdb.read_json("example.json")
```

Read multiple JSON files from a folder:

```
duckdb.read_json("folder/*.json")
```

Directly read a JSON file from within SQL:

```
duckdb.sql("SELECT * FROM 'example.json'")
```

Call `read_json` from within SQL:

```
duckdb.sql("SELECT * FROM read_json_auto('example.json')")
```

Directly Accessing DataFrames and Arrow Objects

DuckDB is automatically able to query certain Python variables by referring to their variable name (as if it was a table). These types include the following: Pandas DataFrame, Polars DataFrame, Polars LazyFrame, NumPy arrays, relations, and Arrow objects.

Only variables that are visible to Python code at the location of the `sql()` or `execute()` call can be used in this manner. Accessing these variables is made possible by replacement scans. To disable replacement scans entirely, use:

```
SET python_enable_replacements = false;
```

DuckDB supports querying multiple types of Apache Arrow objects including tables, datasets, RecordBatchReaders, and scanners. See the Python guides for more examples.

```
import duckdb
import pandas as pd

test_df = pd.DataFrame.from_dict({"i": [1, 2, 3, 4], "j": ["one", "two", "three", "four"]})
print(duckdb.sql("SELECT * FROM test_df").fetchall())
```

```
[(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
```

DuckDB also supports "registering" a DataFrame or Arrow object as a virtual table, comparable to a SQL `VIEW`. This is useful when querying a DataFrame/Arrow object that is stored in another way (as a class variable, or a value in a dictionary). Below is a Pandas example:

If your Pandas DataFrame is stored in another location, here is an example of manually registering it:

```
import duckdb
import pandas as pd

my_dictionary = {}
my_dictionary["test_df"] = pd.DataFrame.from_dict({"i": [1, 2, 3, 4], "j": ["one", "two", "three",
"four"]})
duckdb.register("test_df_view", my_dictionary["test_df"])
print(duckdb.sql("SELECT * FROM test_df_view").fetchall())
```

```
[(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
```

You can also create a persistent table in DuckDB from the contents of the DataFrame (or the view):

```
# create a new table from the contents of a DataFrame
con.execute("CREATE TABLE test_df_table AS SELECT * FROM test_df")
# insert into an existing table from the contents of a DataFrame
con.execute("INSERT INTO test_df_table SELECT * FROM test_df")
```

Pandas DataFrames – `object` Columns

`pandas.DataFrame` columns of an `object` dtype require some special care, since this stores values of arbitrary type. To convert these columns to DuckDB, we first go through an analyze phase before converting the values. In this analyze phase a sample of all the rows of the column are analyzed to determine the target type. This sample size is by default set to 1000. If the type picked during the analyze step is incorrect, this will result in a "Failed to cast value:" error, in which case you will need to increase the sample size. The sample size can be changed by setting the `pandas_analyze_sample` config option.

```
# example setting the sample size to 100k
duckdb.execute("SET GLOBAL pandas_analyze_sample = 100_000")
```

Registering Objects

You can register Python objects as DuckDB tables using the `DuckDBPyConnection.register()` function.

The precedence of objects with the same name is as follows:

- Objects explicitly registered via `DuckDBPyConnection.register()`
- Native DuckDB tables and views
- Replacement scans

Conversion between DuckDB and Python

This page documents the rules for converting Python objects to DuckDB and DuckDB results to Python.

Object Conversion: Python Object to DuckDB

This is a mapping of Python object types to DuckDB Logical Types:

- `None` → `NULL`
- `bool` → `BOOLEAN`
- `datetime.timedelta` → `INTERVAL`
- `str` → `VARCHAR`
- `bytearray` → `BLOB`
- `memoryview` → `BLOB`
- `decimal.Decimal` → `DECIMAL` / `DOUBLE`

- `uuid.UUID` → `UUID`

The rest of the conversion rules are as follows.

`int`

Since integers can be of arbitrary size in Python, there is not a one-to-one conversion possible for ints. Instead we perform these casts in order until one succeeds:

- `BIGINT`
- `INTEGER`
- `UBIGINT`
- `UINTEGER`
- `DOUBLE`

When using the DuckDB Value class, it's possible to set a target type, which will influence the conversion.

`float`

These casts are tried in order until one succeeds:

- `DOUBLE`
- `FLOAT`

`datetime.datetime`

For `datetime` we will check `pandas.isnull` if it's available and return `NULL` if it returns `true`. We check against `datetime.datetime.min` and `datetime.datetime.max` to convert to `-inf` and `+inf` respectively.

If the `datetime` has tzinfo, we will use `TIMESTAMPTZ`, otherwise it becomes `TIMESTAMP`.

`datetime.time`

If the `time` has tzinfo, we will use `TIMETZ`, otherwise it becomes `TIME`.

`datetime.date`

`date` converts to the `DATE` type. We check against `datetime.date.min` and `datetime.date.max` to convert to `-inf` and `+inf` respectively.

`bytes`

`bytes` converts to `BLOB` by default, when it's used to construct a Value object of type `BITSTRING`, it maps to `BITSTRING` instead.

`list`

`list` becomes a `LIST` type of the "most permissive" type of its children, for example:

```
my_list_value = [
    12345,
    "test"
]
```

Will become VARCHAR[] because 12345 can convert to VARCHAR but test can not convert to INTEGER.

```
[12345, test]
```

dict

The dict object can convert to either STRUCT(...) or MAP(..., ...) depending on its structure. If the dict has a structure similar to:

```
import duckdb

my_map_dict = {
    "key": [
        1, 2, 3
    ],
    "value": [
        "one", "two", "three"
    ]
}

duckdb.values(my_map_dict)
```

Then we'll convert it to a MAP of key-value pairs of the two lists zipped together. The example above becomes a MAP(INTEGER, VARCHAR):

```
┌──────────────────────────┐
│ {1=one, 2=two, 3=three}  │
│  map(integer, varchar)   │
├──────────────────────────┤
│ {1=one, 2=two, 3=three}  │
└──────────────────────────┘
```

If the dict is returned by a function, the function will return a MAP, therefore the function return_type has to be specified. Providing a return type which cannot convert to MAP will raise an error:

```
import duckdb
duckdb_conn = duckdb.connect()

def get_map() -> dict[str,list[str]|list[int]]:
    return {
        "key": [
            1, 2, 3
        ],
        "value": [
            "one", "two", "three"
        ]
    }

duckdb_conn.create_function("get_map", get_map, return_type=dict[int, str])

duckdb_conn.sql("select get_map()").show()

duckdb_conn.create_function("get_map_error", get_map)

duckdb_conn.sql("select get_map_error()").show()
```

```
┌──────────────────────────┐
│        get_map()         │
│  map(bigint, varchar)    │
├──────────────────────────┤
```

```
┌──────────────────────────┐
│ {1=one, 2=two, 3=three}  │
└──────────────────────────┘
```

```
ConversionException: Conversion Error: Type VARCHAR can't be cast as UNION(u1 VARCHAR[], u2 BIGINT[]).
VARCHAR can't be implicitly cast to any of the union member types: VARCHAR[], BIGINT[]
```

> The names of the fields matter and the two lists need to have the same size.

Otherwise we'll try to convert it to a STRUCT.

```
import duckdb

my_struct_dict = {
    1: "one",
    "2": 2,
    "three": [1, 2, 3],
    False: True
}

duckdb.values(my_struct_dict)
```

Becomes:

```
┌──────────────────────────────────────────────────────────────────────┐
│     {'1': 'one', '2': 2, 'three': [1, 2, 3], 'False': true}         │
│ struct("1" varchar, "2" integer, three integer[], "false" boolean)   │
├──────────────────────────────────────────────────────────────────────┤
│ {'1': one, '2': 2, 'three': [1, 2, 3], 'False': true}                │
└──────────────────────────────────────────────────────────────────────┘
```

If the dict is returned by a function, the function will return a MAP, due to automatic conversion. To return a STRUCT, the `return_type` has to be provided:

```
import duckdb
from duckdb.typing import BOOLEAN, INTEGER, VARCHAR
from duckdb import list_type, struct_type

duckdb_conn = duckdb.connect()

my_struct_dict = {
    1: "one",
    "2": 2,
    "three": [1, 2, 3],
    False: True
}

def get_struct() -> dict[str|int|bool,str|int|list[int]|bool]:
    return my_struct_dict

duckdb_conn.create_function("get_struct_as_map", get_struct)

duckdb_conn.sql("select get_struct_as_map()").show()

duckdb_conn.create_function("get_struct", get_struct, return_type=struct_type({
    1: VARCHAR,
    "2": INTEGER,
    "three": list_type(duckdb.typing.INTEGER),
    False: BOOLEAN
}))

duckdb_conn.sql("select get_struct()").show()
```

get_struct_as_map()
map(union(u1 varchar, u2 bigint, u3 boolean), union(u1 varchar, u2 bigint, u3 bigint[], u4 boolean))
{1=one, 2=2, three=[1, 2, 3], false=true}

get_struct()
struct("1" varchar, "2" integer, three integer[], "false" boolean)
{'1': one, '2': 2, 'three': [1, 2, 3], 'False': true}

> Every `key` of the dictionary is converted to string.

`tuple`

`tuple` converts to `LIST` by default, when it's used to construct a Value object of type STRUCT it will convert to STRUCT instead.

`numpy.ndarray` and `numpy.datetime64`

`ndarray` and `datetime64` are converted by calling `tolist()` and converting the result of that.

Result Conversion: DuckDB Results to Python

DuckDB's Python client provides multiple additional methods that can be used to efficiently retrieve data.

NumPy

- `fetchnumpy()` fetches the data as a dictionary of NumPy arrays

Pandas

- `df()` fetches the data as a Pandas DataFrame
- `fetchdf()` is an alias of `df()`
- `fetch_df()` is an alias of `df()`
- `fetch_df_chunk(vector_multiple)` fetches a portion of the results into a DataFrame. The number of rows returned in each chunk is the vector size (2048 by default) * vector_multiple (1 by default).

Apache Arrow

- `arrow()` fetches the data as an Arrow table
- `fetch_arrow_table()` is an alias of `arrow()`
- `fetch_record_batch(chunk_size)` returns an Arrow record batch reader with `chunk_size` rows per batch

Polars

- `pl()` fetches the data as a Polars DataFrame

Examples

Below are some examples using this functionality. See the Python guides for more examples.

Fetch as Pandas DataFrame:

```
df = con.execute("SELECT * FROM items").fetchdf()
print(df)
```

```
       item   value  count
0     jeans    20.0      1
1    hammer    42.2      2
2    laptop  2000.0      1
3  chainsaw   500.0     10
4    iphone   300.0      2
```

Fetch as dictionary of NumPy arrays:

```
arr = con.execute("SELECT * FROM items").fetchnumpy()
print(arr)
```

```
{'item': masked_array(data=['jeans', 'hammer', 'laptop', 'chainsaw', 'iphone'],
             mask=[False, False, False, False, False],
       fill_value='?',
            dtype=object), 'value': masked_array(data=[20.0, 42.2, 2000.0, 500.0, 300.0],
             mask=[False, False, False, False, False],
       fill_value=1e+20), 'count': masked_array(data=[1, 2, 1, 10, 2],
             mask=[False, False, False, False, False],
       fill_value=999999,
            dtype=int32)}
```

Fetch as an Arrow table. Converting to Pandas afterwards just for pretty printing:

```
tbl = con.execute("SELECT * FROM items").fetch_arrow_table()
print(tbl.to_pandas())
```

```
       item    value  count
0     jeans    20.00      1
1    hammer    42.20      2
2    laptop  2000.00      1
3  chainsaw   500.00     10
4    iphone   300.00      2
```

Python DB API

The standard DuckDB Python API provides a SQL interface compliant with the DB-API 2.0 specification described by PEP 249 similar to the SQLite Python API.

Connection

To use the module, you must first create a `DuckDBPyConnection` object that represents a connection to a database. This is done through the `duckdb.connect` method.

The 'config' keyword argument can be used to provide a `dict` that contains key->value pairs referencing settings understood by DuckDB.

In-Memory Connection

The special value `:memory:` can be used to create an **in-memory database**. Note that for an in-memory database no data is persisted to disk (i.e., all data is lost when you exit the Python process).

Named In-memory Connections

The special value `:memory:` can also be postfixed with a name, for example: `:memory:conn3`. When a name is provided, subsequent `duckdb.connect` calls will create a new connection to the same database, sharing the catalogs (views, tables, macros etc..).

Using `:memory:` without a name will always create a new and separate database instance.

Default Connection

By default we create an (unnamed) **in-memory-database** that lives inside the `duckdb` module. Every method of `DuckDBPyConnection` is also available on the `duckdb` module, this connection is what's used by these methods.

The special value `:default:` can be used to get this default connection.

File-Based Connection

If the `database` is a file path, a connection to a persistent database is established. If the file does not exist the file will be created (the extension of the file is irrelevant and can be `.db`, `.duckdb` or anything else).

`read_only` Connections

If you would like to connect in read-only mode, you can set the `read_only` flag to `True`. If the file does not exist, it is **not** created when connecting in read-only mode. Read-only mode is required if multiple Python processes want to access the same database file at the same time.

```
import duckdb

duckdb.execute("CREATE TABLE tbl AS SELECT 42 a")
con = duckdb.connect(":default:")
con.sql("SELECT * FROM tbl")
# or
duckdb.default_connection().sql("SELECT * FROM tbl")
```

```
┌─────────┐
│    a    │
│  int32  │
├─────────┤
│      42 │
└─────────┘
```

```
import duckdb

# to start an in-memory database
con = duckdb.connect(database = ":memory:")
# to use a database file (not shared between processes)
con = duckdb.connect(database = "my-db.duckdb", read_only = False)
# to use a database file (shared between processes)
con = duckdb.connect(database = "my-db.duckdb", read_only = True)
# to explicitly get the default connection
con = duckdb.connect(database = ":default:")
```

If you want to create a second connection to an existing database, you can use the `cursor()` method. This might be useful for example to allow parallel threads running queries independently. A single connection is thread-safe but is locked for the duration of the queries, effectively serializing database access in this case.

Connections are closed implicitly when they go out of scope or if they are explicitly closed using `close()`. Once the last connection to a database instance is closed, the database instance is closed as well.

Querying

SQL queries can be sent to DuckDB using the `execute()` method of connections. Once a query has been executed, results can be retrieved using the `fetchone` and `fetchall` methods on the connection. `fetchall` will retrieve all results and complete the transaction. `fetchone` will retrieve a single row of results each time that it is invoked until no more results are available. The transaction will only close once `fetchone` is called and there are no more results remaining (the return value will be None). As an example, in the case of a query only returning a single row, `fetchone` should be called once to retrieve the results and a second time to close the transaction. Below are some short examples:

```
# create a table
con.execute("CREATE TABLE items (item VARCHAR, value DECIMAL(10, 2), count INTEGER)")
# insert two items into the table
con.execute("INSERT INTO items VALUES ('jeans', 20.0, 1), ('hammer', 42.2, 2)")

# retrieve the items again
con.execute("SELECT * FROM items")
print(con.fetchall())
# [('jeans', Decimal('20.00'), 1), ('hammer', Decimal('42.20'), 2)]

# retrieve the items one at a time
con.execute("SELECT * FROM items")
print(con.fetchone())
# ('jeans', Decimal('20.00'), 1)
print(con.fetchone())
# ('hammer', Decimal('42.20'), 2)
print(con.fetchone()) # This closes the transaction. Any subsequent calls to .fetchone will return None
# None
```

The `description` property of the connection object contains the column names as per the standard.

Prepared Statements

DuckDB also supports prepared statements in the API with the `execute` and `executemany` methods. The values may be passed as an additional parameter after a query that contains `?` or `$1` (dollar symbol and a number) placeholders. Using the `?` notation adds the values in the same sequence as passed within the Python parameter. Using the `$` notation allows for values to be reused within the SQL statement based on the number and index of the value found within the Python parameter. Values are converted according to the conversion rules.

Here are some examples. First, insert a row using a prepared statement:

```
con.execute("INSERT INTO items VALUES (?, ?, ?)", ["laptop", 2000, 1])
```

Second, insert several rows using a prepared statement:

```
con.executemany("INSERT INTO items VALUES (?, ?, ?)", [["chainsaw", 500, 10], ["iphone", 300, 2]] )
```

Query the database using a prepared statement:

```
con.execute("SELECT item FROM items WHERE value > ?", [400])
print(con.fetchall())
```

```
[('laptop',), ('chainsaw',)]
```

Query using the `$` notation for a prepared statement and reused values:

```
con.execute("SELECT $1, $1, $2", ["duck", "goose"])
print(con.fetchall())
```

```
[('duck', 'duck', 'goose')]
```

> **Warning.** Do *not* use `executemany` to insert large amounts of data into DuckDB. See the data ingestion page for better options.

Named Parameters

Besides the standard unnamed parameters, like `$1`, `$2` etc., it's also possible to supply named parameters, like `$my_parameter`. When using named parameters, you have to provide a dictionary mapping of `str` to value in the `parameters` argument. An example use is the following:

```
import duckdb

res = duckdb.execute("""
    SELECT
        $my_param,
        $other_param,
        $also_param
    """,
    {
        "my_param": 5,
        "other_param": "DuckDB",
        "also_param": [42]
    }
).fetchall()
print(res)
```

```
[(5, 'DuckDB', [42])]
```

Relational API

The Relational API is an alternative API that can be used to incrementally construct queries. The API is centered around `DuckDBPyRelation` nodes. The relations can be seen as symbolic representations of SQL queries.

Lazy Evaluation

The relations do not hold any data – and nothing is executed – until a method that triggers execution is called.

For example, we create a relation, which loads 1 billion rows:

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("from range(1_000_000_000)")
```

At the moment of execution, `rel` does not hold any data and no data is retrieved from the database.

By calling `rel.show()` or simply printing `rel` on the terminal, the first 10K rows are fetched. If there are more than 10K rows, the output window will show >9999 rows (as the amount of rows in the relation is unknown).

By calling an output method, the data is retrieved and stored in the specified format:

```
rel.to_table("example_rel")

# 100% ▕████████████████████████████████████████████████████████▏
```

Relation Creation

This section contains the details on how a relation is created. The methods are lazy evaluated.

Name	Description
`from_arrow`	Create a relation object from an Arrow object
`from_csv_auto`	Create a relation object from the CSV file in 'name'
`from_df`	Create a relation object from the DataFrame in df
`from_parquet`	Create a relation object from the Parquet files
`from_query`	Run a SQL query. If it is a SELECT statement, create a relation object from the given SQL query, otherwise run the query as-is.
`query`	Run a SQL query. If it is a SELECT statement, create a relation object from the given SQL query, otherwise run the query as-is.
`read_csv`	Create a relation object from the CSV file in 'name'
`read_json`	Create a relation object from the JSON file in 'name'
`read_parquet`	Create a relation object from the Parquet files
`sql`	Run a SQL query. If it is a SELECT statement, create a relation object from the given SQL query, otherwise run the query as-is.
`table`	Create a relation object for the named table
`table_function`	Create a relation object from the named table function with given parameters
`values`	Create a relation object from the passed values
`view`	Create a relation object for the named view

`from_arrow`

Signature

```
from_arrow(self: duckdb.duckdb.DuckDBPyConnection, arrow_object: object) ->
duckdb.duckdb.DuckDBPyRelation
```

Description Create a relation object from an Arrow object

Parameters

- **arrow_object** : pyarrow.Table, pyarrow.RecordBatch

  Arrow object to create a relation from

Example

```
import duckdb
import pyarrow as pa

ids = pa.array([1], type=pa.int8())
texts = pa.array(['a'], type=pa.string())
example_table = pa.table([ids, texts], names=["id", "text"])

duckdb_conn = duckdb.connect()
```

```
rel = duckdb_conn.from_arrow(example_table)

rel.show()
```

Result

id int8	text varchar
1	a

`from_csv_auto`

Signature

```
from_csv_auto(self: duckdb.duckdb.DuckDBPyConnection, path_or_buffer: object, **kwargs) ->
duckdb.duckdb.DuckDBPyRelation
```

Description Create a relation object from the CSV file in 'name'

Aliases: `read_csv`

Parameters

- **path_or_buffer** : Union[str, StringIO, TextIOBase]

  Path to the CSV file or buffer to read from.

- **header** : Optional[bool], Optional[int]

  Row number(s) to use as the column names, or None if no header.

- **compression** : Optional[str]

  Compression type (e.g., 'gzip', 'bz2').

- **sep** : Optional[str]

  Delimiter to use; defaults to comma.

- **delimiter** : Optional[str]

  Alternative delimiter to use.

- **dtype** : Optional[Dict[str, str]], Optional[List[str]]

  Data types for columns.

- **na_values** : Optional[str], Optional[List[str]]

  Additional strings to recognize as NA/NaN.

- **skiprows** : Optional[int]

  Number of rows to skip at the start.

- **quotechar** : Optional[str]

  Character used to quote fields.

- **escapechar** : Optional[str]

  Character used to escape delimiter or quote characters.

- **encoding** : Optional[str]

  Encoding to use for UTF when reading/writing.

- **parallel** : Optional[bool]

  Enable parallel reading.

- **date_format** : Optional[str]

  Format to parse dates.

- **timestamp_format** : Optional[str]

  Format to parse timestamps.

- **sample_size** : Optional[int]

  Number of rows to sample for schema inference.

- **all_varchar** : Optional[bool]

  Treat all columns as VARCHAR.

- **normalize_names** : Optional[bool]

  Normalize column names to lowercase.

- **null_padding** : Optional[bool]

  Enable null padding for rows with missing columns.

- **names** : Optional[List[str]]

  List of column names to use.

- **lineterminator** : Optional[str]

  Character to break lines on.

- **columns** : Optional[Dict[str, str]]

  Column mapping for schema.

- **auto_type_candidates** : Optional[List[str]]

  List of columns for automatic type inference.

- **max_line_size** : Optional[int]

  Maximum line size in bytes.

- **ignore_errors** : Optional[bool]

  Ignore parsing errors.

- **store_rejects** : Optional[bool]

  Store rejected rows.

- **rejects_table** : Optional[str]

  Table name to store rejected rows.

- **rejects_scan** : Optional[str]

  Scan to use for rejects.

- **rejects_limit** : Optional[int]

  Limit number of rejects stored.

- **force_not_null** : Optional[List[str]]

  List of columns to force as NOT NULL.

- **buffer_size** : Optional[int]

  Buffer size in bytes.

- **decimal** : Optional[str]

  Character to recognize as decimal point.

- **allow_quoted_nulls** : Optional[bool]

  Allow quoted NULL values.

- **filename** : Optional[bool], Optional[str]

  Add filename column or specify filename.

- **hive_partitioning** : Optional[bool]

  Enable Hive-style partitioning.

- **union_by_name** : Optional[bool]

  Union files by column name instead of position.

- **hive_types** : Optional[Dict[str, str]]

  Hive types for columns.

- **hive_types_autocast** : Optional[bool]

  Automatically cast Hive types.

- **connection** : DuckDBPyConnection

  DuckDB connection to use.

Example

```
import csv
import duckdb

duckdb_conn = duckdb.connect()

with open('code_example.csv', 'w', newline='') as csvfile:
    fieldnames = ['id', 'text']
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
    writer.writeheader()
    writer.writerow({'id': '1', 'text': 'a'})

rel = duckdb_conn.from_csv_auto("code_example.csv")

rel.show()
```

Result

```
┌─────────┬─────────┐
│   id    │  text   │
│  int64  │ varchar │
├─────────┼─────────┤
```

1	a

---

from_df

Signature

```
from_df(self: duckdb.duckdb.DuckDBPyConnection, df: pandas.DataFrame) -> duckdb.duckdb.DuckDBPyRelation
```

Description Create a relation object from the DataFrame in df

Parameters

- **df** : pandas.DataFrame

  A pandas DataFrame to be converted into a DuckDB relation.

Example

```
import duckdb
import pandas as pd

df = pd.DataFrame(data = {'id': [1], "text":["a"]})

duckdb_conn = duckdb.connect()

rel = duckdb_conn.from_df(df)

rel.show()
```

Result

id int64	text varchar
1	a

---

from_parquet

Signature

```
from_parquet(*args, **kwargs)
Overloaded function.

1. from_parquet(self: duckdb.duckdb.DuckDBPyConnection, file_glob: str, binary_as_string: bool = False,
*, file_row_number: bool = False, filename: bool = False, hive_partitioning: bool = False, union_by_
name: bool = False, compression: object = None) -> duckdb.duckdb.DuckDBPyRelation

Create a relation object from the Parquet files in file_glob
```

```
2. from_parquet(self: duckdb.duckdb.DuckDBPyConnection, file_globs: list[str], binary_as_string: bool =
False, *, file_row_number: bool = False, filename: bool = False, hive_partitioning: bool = False, union_
by_name: bool = False, compression: object = None) -> duckdb.duckdb.DuckDBPyRelation

Create a relation object from the Parquet files in file_globs
```

Description Create a relation object from the Parquet files

Aliases: `read_parquet`

Parameters

- **file_glob** : str

  File path or glob pattern pointing to Parquet files to be read.

- **binary_as_string** : bool, default: False

  Interpret binary columns as strings instead of blobs.

- **file_row_number** : bool, default: False

  Add a column containing the row number within each file.

- **filename** : bool, default: False

  Add a column containing the name of the file each row came from.

- **hive_partitioning** : bool, default: False

  Enable automatic detection of Hive-style partitions in file paths.

- **union_by_name** : bool, default: False

  Union Parquet files by matching column names instead of positions.

- **compression** : object

  Optional compression codec to use when reading the Parquet files.

Example

```
import duckdb
import pyarrow as pa
import pyarrow.parquet as pq

ids = pa.array([1], type=pa.int8())
texts = pa.array(['a'], type=pa.string())
example_table = pa.table([ids, texts], names=["id", "text"])

pq.write_table(example_table, "code_example.parquet")

duckdb_conn = duckdb.connect()

rel = duckdb_conn.from_parquet("code_example.parquet")

rel.show()
```

Result

id int8	text varchar
1	a

`from_query`

Signature

```
from_query(self: duckdb.duckdb.DuckDBPyConnection, query: object, *, alias: str = '', params: object = None) -> duckdb.duckdb.DuckDBPyRelation
```

Description Run a SQL query. If it is a SELECT statement, create a relation object from the given SQL query, otherwise run the query as-is.

Aliases: `query`, `sql`

Parameters

- **query** : object

  The SQL query or subquery to be executed and converted into a relation.

- **alias** : str, default: ''

  Optional alias name to assign to the resulting relation.

- **params** : object

  Optional query parameters to be used in the SQL query.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.from_query("from range(1,2) tbl(id)")

rel.show()
```

Result

id int64
1

---

query

Signature

```
query(self: duckdb.duckdb.DuckDBPyConnection, query: object, *, alias: str = '', params: object = None)
-> duckdb.duckdb.DuckDBPyRelation
```

Description Run a SQL query. If it is a SELECT statement, create a relation object from the given SQL query, otherwise run the query as-is.

Aliases: `from_query`, `sql`

Parameters

- **query** : object

  The SQL query or subquery to be executed and converted into a relation.

- **alias** : str, default: ''

  Optional alias name to assign to the resulting relation.

- **params** : object

  Optional query parameters to be used in the SQL query.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.query("from range(1,2) tbl(id)")

rel.show()
```

Result

```
┌─────────┐
│   id    │
│  int64  │
├─────────┤
│       1 │
└─────────┘
```

---

read_csv

Signature

```
read_csv(self: duckdb.duckdb.DuckDBPyConnection, path_or_buffer: object, **kwargs) ->
duckdb.duckdb.DuckDBPyRelation
```

Description Create a relation object from the CSV file in 'name'

Aliases: `from_csv_auto`

Parameters

- **path_or_buffer** : Union[str, StringIO, TextIOBase]

  Path to the CSV file or buffer to read from.

- **header** : Optional[bool], Optional[int]

  Row number(s) to use as the column names, or None if no header.

- **compression** : Optional[str]

  Compression type (e.g., 'gzip', 'bz2').

- **sep** : Optional[str]

  Delimiter to use; defaults to comma.

- **delimiter** : Optional[str]

  Alternative delimiter to use.

- **dtype** : Optional[Dict[str, str]], Optional[List[str]]

  Data types for columns.

- **na_values** : Optional[str], Optional[List[str]]

  Additional strings to recognize as NA/NaN.

- **skiprows** : Optional[int]

  Number of rows to skip at the start.

- **quotechar** : Optional[str]

  Character used to quote fields.

- **escapechar** : Optional[str]

  Character used to escape delimiter or quote characters.

- **encoding** : Optional[str]

  Encoding to use for UTF when reading/writing.

- **parallel** : Optional[bool]

  Enable parallel reading.

- **date_format** : Optional[str]

  Format to parse dates.

- **timestamp_format** : Optional[str]

  Format to parse timestamps.

- **sample_size** : Optional[int]

  Number of rows to sample for schema inference.

- **all_varchar** : Optional[bool]

  Treat all columns as VARCHAR.

- **normalize_names** : Optional[bool]

  Normalize column names to lowercase.

- **null_padding** : Optional[bool]

  Enable null padding for rows with missing columns.

- **names** : Optional[List[str]]

  List of column names to use.

- **lineterminator** : Optional[str]

  Character to break lines on.

- **columns** : Optional[Dict[str, str]]

  Column mapping for schema.

- **auto_type_candidates** : Optional[List[str]]

  List of columns for automatic type inference.

- **max_line_size** : Optional[int]

  Maximum line size in bytes.

- **ignore_errors** : Optional[bool]

  Ignore parsing errors.

- **store_rejects** : Optional[bool]

  Store rejected rows.

- **rejects_table** : Optional[str]

  Table name to store rejected rows.

- **rejects_scan** : Optional[str]

  Scan to use for rejects.

- **rejects_limit** : Optional[int]

  Limit number of rejects stored.

- **force_not_null** : Optional[List[str]]

  List of columns to force as NOT NULL.

- **buffer_size** : Optional[int]

  Buffer size in bytes.

- **decimal** : Optional[str]

  Character to recognize as decimal point.

- **allow_quoted_nulls** : Optional[bool]

  Allow quoted NULL values.

- **filename** : Optional[bool], Optional[str]

  Add filename column or specify filename.

- **hive_partitioning** : Optional[bool]

  Enable Hive-style partitioning.

- **union_by_name** : Optional[bool]

  Union files by column name instead of position.

- **hive_types** : Optional[Dict[str, str]]

  Hive types for columns.

- **hive_types_autocast** : Optional[bool]

  Automatically cast Hive types.

- **connection** : DuckDBPyConnection

  DuckDB connection to use.

Example

```
import csv
import duckdb

duckdb_conn = duckdb.connect()

with open('code_example.csv', 'w', newline='') as csvfile:
    fieldnames = ['id', 'text']
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
    writer.writeheader()
    writer.writerow({'id': '1', 'text': 'a'})

rel = duckdb_conn.read_csv("code_example.csv")

rel.show()
```

Result

id int64	text varchar
1	a

`read_json`

Signature

```
read_json(self: duckdb.duckdb.DuckDBPyConnection, path_or_buffer: object, *, columns:
typing.Optional[object] = None, sample_size: typing.Optional[object] = None, maximum_depth:
typing.Optional[object] = None, records: typing.Optional[str] = None, format: typing.Optional[str] =
None, date_format: typing.Optional[object] = None, timestamp_format: typing.Optional[object] = None,
compression: typing.Optional[object] = None, maximum_object_size: typing.Optional[object] = None,
ignore_errors: typing.Optional[object] = None, convert_strings_to_integers: typing.Optional[object] =
None, field_appearance_threshold: typing.Optional[object] = None, map_inference_threshold:
typing.Optional[object] = None, maximum_sample_files: typing.Optional[object] = None, filename:
typing.Optional[object] = None, hive_partitioning: typing.Optional[object] = None, union_by_name:
typing.Optional[object] = None, hive_types: typing.Optional[object] = None, hive_types_autocast:
typing.Optional[object] = None) -> duckdb.duckdb.DuckDBPyRelation
```

Description Create a relation object from the JSON file in 'name'

Parameters

- **path_or_buffer** : object

  File path or file-like object containing JSON data to be read.

- **columns** : object

  Optional list of column names to project from the JSON data.

- **sample_size** : object

  Number of rows to sample for inferring JSON schema.

- **maximum_depth** : object

  Maximum depth to which JSON objects should be parsed.

- **records** : str

  Format string specifying whether JSON is in records mode.

- **format** : str

  Format of the JSON data (e.g., 'auto', 'newline_delimited').

- **date_format** : object

  Format string for parsing date fields.

- **timestamp_format** : object

  Format string for parsing timestamp fields.

- **compression** : object

  Compression codec used on the JSON data (e.g., 'gzip').

- **maximum_object_size** : object

  Maximum size in bytes for individual JSON objects.

- **ignore_errors** : object

  If True, skip over JSON records with parsing errors.

- **convert_strings_to_integers** : object

  If True, attempt to convert strings to integers where appropriate.

- **field_appearance_threshold** : object

  Threshold for inferring optional fields in nested JSON.

- **map_inference_threshold** : object

  Threshold for inferring maps from JSON object patterns.

- **maximum_sample_files** : object

  Maximum number of files to sample for schema inference.

- **filename** : object

  If True, include a column with the source filename for each row.

- **hive_partitioning** : object

  If True, enable Hive partitioning based on directory structure.

- **union_by_name** : object

  If True, align JSON columns by name instead of position.

- **hive_types** : object

  If True, use Hive types from directory structure for schema.

- **hive_types_autocast** : object

  If True, automatically cast data types to match Hive types.

Example

```
import duckdb
import json

with open("code_example.json", mode="w") as f:
    json.dump([{'id': 1, "text":"a"}], f)

duckdb_conn = duckdb.connect()

rel = duckdb_conn.read_json("code_example.json")

rel.show()
```

Result

id int64	text varchar
1	a

---

read_parquet

Signature

```
read_parquet(*args, **kwargs)
Overloaded function.

1. read_parquet(self: duckdb.duckdb.DuckDBPyConnection, file_glob: str, binary_as_string: bool = False,
*, file_row_number: bool = False, filename: bool = False, hive_partitioning: bool = False, union_by_
name: bool = False, compression: object = None) -> duckdb.duckdb.DuckDBPyRelation

Create a relation object from the Parquet files in file_glob

2. read_parquet(self: duckdb.duckdb.DuckDBPyConnection, file_globs: list[str], binary_as_string: bool =
False, *, file_row_number: bool = False, filename: bool = False, hive_partitioning: bool = False, union_
by_name: bool = False, compression: object = None) -> duckdb.duckdb.DuckDBPyRelation

Create a relation object from the Parquet files in file_globs
```

Description Create a relation object from the Parquet files

Aliases: `from_parquet`

Parameters

- **file_glob** : str

  File path or glob pattern pointing to Parquet files to be read.

- **binary_as_string** : bool, default: False

  Interpret binary columns as strings instead of blobs.

- **file_row_number** : bool, default: False

  Add a column containing the row number within each file.

- **filename** : bool, default: False

  Add a column containing the name of the file each row came from.

- **hive_partitioning** : bool, default: False

  Enable automatic detection of Hive-style partitions in file paths.

- **union_by_name** : bool, default: False

  Union Parquet files by matching column names instead of positions.

- **compression** : object

  Optional compression codec to use when reading the Parquet files.

Example

```
import duckdb
import pyarrow as pa
import pyarrow.parquet as pq

ids = pa.array([1], type=pa.int8())
texts = pa.array(['a'], type=pa.string())
example_table = pa.table([ids, texts], names=["id", "text"])

pq.write_table(example_table, "code_example.parquet")

duckdb_conn = duckdb.connect()

rel = duckdb_conn.read_parquet("code_example.parquet")

rel.show()
```

Result

id int8	text varchar
1	a

---

sql

Signature

```
sql(self: duckdb.duckdb.DuckDBPyConnection, query: object, *, alias: str = '', params: object = None) ->
duckdb.duckdb.DuckDBPyRelation
```

Description Run a SQL query. If it is a SELECT statement, create a relation object from the given SQL query, otherwise run the query as-is.

Aliases: `from_query`, `query`

Parameters

- **query** : object

  The SQL query or subquery to be executed and converted into a relation.

- **alias** : str, default: ''

  Optional alias name to assign to the resulting relation.

- **params** : object

  Optional query parameters to be used in the SQL query.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("from range(1,2) tbl(id)")

rel.show()
```

Result

```
┌─────────┐
│   id    │
│  int64  │
├─────────┤
│       1 │
└─────────┘
```

`table`

Signature

```
table(self: duckdb.duckdb.DuckDBPyConnection, table_name: str) -> duckdb.duckdb.DuckDBPyRelation
```

Description Create a relation object for the named table

Parameters

- **table_name** : str

  Name of the table to create a relation from.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

duckdb_conn.sql("create table code_example as select * from range(1,2) tbl(id)")

rel = duckdb_conn.table("code_example")

rel.show()
```

Result

```
┌───────┐
│  id   │
│ int64 │
├───────┤
│     1 │
└───────┘
```

`table_function`

Signature

```
table_function(self: duckdb.duckdb.DuckDBPyConnection, name: str, parameters: object = None) ->
duckdb.duckdb.DuckDBPyRelation
```

Description Create a relation object from the named table function with given parameters

Parameters

- **name** : str

  Name of the table function to call.

- **parameters** : object

  Optional parameters to pass to the table function.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

duckdb_conn.sql("""
    create macro get_record_for(x) as table
    select x*range from range(1,2)
""")

rel = duckdb_conn.table_function(name="get_record_for", parameters=[1])

rel.show()
```

Result

(1 * "range")
int64
1

---

values

Signature

```
values(self: duckdb.duckdb.DuckDBPyConnection, *args) -> duckdb.duckdb.DuckDBPyRelation
```

Description Create a relation object from the passed values

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.values([1, 'a'])

rel.show()
```

Result

col0	col1
int32	varchar
1	a

---

view

Signature

```
view(self: duckdb.duckdb.DuckDBPyConnection, view_name: str) -> duckdb.duckdb.DuckDBPyRelation
```

Description Create a relation object for the named view

Parameters

- **view_name** : str

  Name of the view to create a relation from.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

duckdb_conn.sql("create table code_example as select * from range(1,2) tbl(id)")

rel = duckdb_conn.view("code_example")

rel.show()
```

Result

```
┌─────────┐
│   id    │
│  int64  │
├─────────┤
│       1 │
└─────────┘
```

Relation Definition Details

This section contains the details on how to inspect a relation.

Name	Description
`alias`	Get the name of the current alias
`columns`	Return a list containing the names of the columns of the relation.
`describe`	Gives basic statistics (e.g., min, max) and if NULL exists for each column of the relation.
`description`	Return the description of the result
`dtypes`	Return a list containing the types of the columns of the relation.
`explain`	explain(self: duckdb.duckdb.DuckDBPyRelation, type: duckdb.duckdb.ExplainType = 'standard') -> str
`query`	Run the given SQL query in sql_query on the view named virtual_table_name that refers to the relation object
`set_alias`	Rename the relation object to new alias
`shape`	Tuple of # of rows, # of columns in relation.
`show`	Display a summary of the data
`sql_query`	Get the SQL query that is equivalent to the relation
`type`	Get the type of the relation.
`types`	Return a list containing the types of the columns of the relation.

`alias`

Description Get the name of the current alias

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.alias
```

Result

```
unnamed_relation_43c808c247431be5
```

`columns`

Description Return a list containing the names of the columns of the relation.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.columns
```

Result

```
 ['id', 'description', 'value', 'created_timestamp']
```

`describe`

Signature

```
describe(self: duckdb.duckdb.DuckDBPyRelation) -> duckdb.duckdb.DuckDBPyRelation
```

Description Gives basic statistics (e.g., min, max) and if NULL exists for each column of the relation.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.describe()
```

Result

```
┌─────────┬──────────────────────────────────────┬──────────────┬────────────────────┬──────────────────────┐
│  aggr   │                  id                  │ description  │       value        │      created_        │
│ timestamp       │
│ varchar │               varchar                │   varchar    │       double       │       varchar        │
├─────────┼──────────────────────────────────────┼──────────────┼────────────────────┼──────────────────────┤
│ count   │ 9                                    │ 9            │                9.0 │ 9                    │
│ mean    │ NULL                                 │ NULL         │                5.0 │ NULL                 │
│ stddev  │ NULL                                 │ NULL         │ 2.7386127875258306 │ NULL                 │
│ min     │ 08fdcbf8-4e53-4290-9e81-423af263b518 │ value is even│                1.0 │ 2025-04-09
15:41:20.642+02 │
│ max     │ fb10390e-fad5-4694-91cb-e82728cb6f9f │ value is uneven│              9.0 │ 2025-04-09
15:49:20.642+02 │
│ median  │ NULL                                 │ NULL         │                5.0 │ NULL                 │
└─────────┴──────────────────────────────────────┴──────────────┴────────────────────┴──────────────────────┘
```

`description`

Description Return the description of the result

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
```

```
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.description
```

Result

```
[('id', 'UUID', None, None, None, None, None),
 ('description', 'STRING', None, None, None, None, None),
 ('value', 'NUMBER', None, None, None, None, None),
 ('created_timestamp', 'DATETIME', None, None, None, None, None)]
```

---

dtypes

Description Return a list containing the types of the columns of the relation.

Aliases: `types`

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.dtypes
```

Result

```
 [UUID, VARCHAR, BIGINT, TIMESTAMP WITH TIME ZONE]
```

---

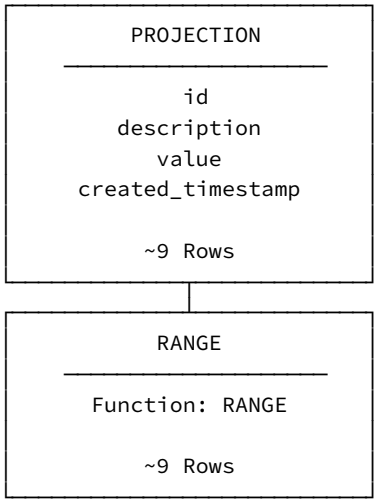
explain

Description explain(self: duckdb.duckdb.DuckDBPyRelation, type: duckdb.duckdb.ExplainType = 'standard') -> str

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.explain()
```

Result



---

query

Signature

```
query(self: duckdb.duckdb.DuckDBPyRelation, virtual_table_name: str, sql_query: str) ->
duckdb.duckdb.DuckDBPyRelation
```

Description Run the given SQL query in sql_query on the view named virtual_table_name that refers to the relation object

Parameters

- **virtual_table_name** : str

  The name to assign to the current relation when referenced in the SQL query.

- **sql_query** : str

  The SQL query string that uses the virtual table name to query the relation.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.query(virtual_table_name="rel_view", sql_query="from rel")

duckdb_conn.sql("show rel_view")
```

Result

column_name varchar	column_type varchar	null varchar	key varchar	default varchar	extra varchar
id	UUID	YES	NULL	NULL	NULL
description	VARCHAR	YES	NULL	NULL	NULL
value	BIGINT	YES	NULL	NULL	NULL
created_timestamp	TIMESTAMP WITH TIME ZONE	YES	NULL	NULL	NULL

---

set_alias

Signature

```
set_alias(self: duckdb.duckdb.DuckDBPyRelation, alias: str) -> duckdb.duckdb.DuckDBPyRelation
```

Description Rename the relation object to new alias

Parameters

- **alias** : str

  The alias name to assign to the relation.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
```

```
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.set_alias('abc').select('abc.id')
```

Result

```
In the SQL query, the alias will be `abc`
```

---

shape

Description Tuple of # of rows, # of columns in relation.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.shape
```

Result

```
(9, 4)
```

---

show

Signature

```
show(self: duckdb.duckdb.DuckDBPyRelation, *, max_width: typing.Optional[int] = None, max_rows:
typing.Optional[int] = None, max_col_width: typing.Optional[int] = None, null_value:
typing.Optional[str] = None, render_mode: object = None) -> None
```

Description Display a summary of the data

Parameters

- **max_width** : int

  Maximum display width for the entire output in characters.

- **max_rows** : int

  Maximum number of rows to display.

- **max_col_width** : int

  Maximum number of characters to display per column.

- **null_value** : str

  String to display in place of NULL values.

- **render_mode** : object

  Render mode for displaying the output.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.show()
```

Result

id uuid	description varchar	value int64	created_timestamp timestamp with time zone
642ea3d7-793d-4867-a759-91c1226c25a0	value is uneven	1	2025-04-09 15:41:20.642+02
6817dd31-297c-40a8-8e40-8521f00b2d08	value is even	2	2025-04-09 15:42:20.642+02
45143f9a-e16e-4e59-91b2-3a0800eed6d6	value is uneven	3	2025-04-09 15:43:20.642+02
fb10390e-fad5-4694-91cb-e82728cb6f9f	value is even	4	2025-04-09 15:44:20.642+02
111ced5c-9155-418e-b087-c331b814db90	value is uneven	5	2025-04-09 15:45:20.642+02
66a870a6-aef0-4085-87d5-5d1b35d21c66	value is even	6	2025-04-09 15:46:20.642+02
a7e8e796-bca0-44cd-a269-1d71090fb5cc	value is uneven	7	2025-04-09 15:47:20.642+02
74908d48-7f2d-4bdd-9c92-1e7920b115b5	value is even	8	2025-04-09 15:48:20.642+02
08fdcbf8-4e53-4290-9e81-423af263b518	value is uneven	9	2025-04-09 15:49:20.642+02

sql_query

Signature

```
sql_query(self: duckdb.duckdb.DuckDBPyRelation) -> str
```

Description Get the SQL query that is equivalent to the relation

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.sql_query()
```

Result

```
SELECT
    gen_random_uuid() AS id,
    concat('value is ', CASE  WHEN ((mod("range", 2) = 0)) THEN ('even') ELSE 'uneven' END) AS
    description,
    "range" AS "value",
    (now() + CAST(concat("range", ' ', 'minutes') AS INTERVAL)) AS created_timestamp
FROM "range"(1, 10)
```

---

type

Description Get the type of the relation.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)
```

```
rel.type
```

Result

```
QUERY_RELATION
```

---

types

Description Return a list containing the types of the columns of the relation.

Aliases: `dtypes`

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.types
```

Result

```
[UUID, VARCHAR, BIGINT, TIMESTAMP WITH TIME ZONE]
```

Transformation

This section contains the methods which can be used to chain queries. The methods are lazy evaluated.

Name	Description
`aggregate`	Compute the aggregate aggr_expr by the optional groups group_expr on the relation
`apply`	Compute the function of a single column or a list of columns by the optional groups on the relation
`cross`	Create cross/cartesian product of two relational objects
`except_`	Create the set except of this relation object with another relation object in other_rel
`filter`	Filter the relation object by the filter in filter_expr
`insert`	Inserts the given values into the relation
`insert_into`	Inserts the relation object into an existing table named table_name
`intersect`	Create the set intersection of this relation object with another relation object in other_rel

Name	Description
`join`	Join the relation object with another relation object in other_rel using the join condition expression in join_condition. Types supported are 'inner', 'left', 'right', 'outer', 'semi' and 'anti'
`limit`	Only retrieve the first n rows from this relation object, starting at offset
`map`	Calls the passed function on the relation
`order`	Reorder the relation object by order_expr
`project`	Project the relation object by the projection in project_expr
`select`	Project the relation object by the projection in project_expr
`sort`	Reorder the relation object by the provided expressions
`union`	Create the set union of this relation object with another relation object in other_rel
`update`	Update the given relation with the provided expressions

aggregate

Signature

```
aggregate(self: duckdb.duckdb.DuckDBPyRelation, aggr_expr: object, group_expr: str = '') ->
duckdb.duckdb.DuckDBPyRelation
```

Description Compute the aggregate aggr_expr by the optional groups group_expr on the relation

Parameters

- **aggr_expr** : str, list[Expression]

  The list of columns and aggregation functions.

- **group_expr** : str, default: ''

  The list of columns to be included in `group_by`. If None, `group by all` is applied.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel = rel.aggregate('max(value)')
```

Result

max("value") int64
9

---

apply

Signature

```
apply(self: duckdb.duckdb.DuckDBPyRelation, function_name: str, function_aggr: str, group_expr: str = '', function_parameter: str = '', projected_columns: str = '') -> duckdb.duckdb.DuckDBPyRelation
```

Description Compute the function of a single column or a list of columns by the optional groups on the relation

Parameters

- **function_name** : str

  Name of the function to apply over the relation.

- **function_aggr** : str

  The list of columns to apply the function over.

- **group_expr** : str, default: ''

  Optional SQL expression for grouping.

- **function_parameter** : str, default: ''

  Optional parameters to pass into the function.

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.apply(
    function_name="count",
```

```
    function_aggr="id",
    group_expr="description",
    projected_columns="description"
)
```

Result

description varchar	count(id) int64
value is uneven	5
value is even	4

`cross`

Signature

```
cross(self: duckdb.duckdb.DuckDBPyRelation, other_rel: duckdb.duckdb.DuckDBPyRelation) ->
duckdb.duckdb.DuckDBPyRelation
```

Description Create cross/cartesian product of two relational objects

Parameters

- **other_rel** : duckdb.duckdb.DuckDBPyRelation

  Another relation to perform a cross product with.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.cross(other_rel=rel.set_alias("other_rel"))
```

Result

```
┌──────────────────────────────────────┬─────────────────┬─────────┬────────────────────────────┬──────────────────────────────────────┬─────────────────┬─────────┬────────────────────────────┐
│                  id                  │   description   │  value  │     created_timestamp      │                  id                  │   description   │  value  │     created_timestamp      │
│                 uuid                 │     varchar     │  int64  │ timestamp with time zone   │                 uuid                 │     varchar     │  int64  │ timestamp with time zone   │
├──────────────────────────────────────┼─────────────────┼─────────┼────────────────────────────┼──────────────────────────────────────┼─────────────────┼─────────┼────────────────────────────┤
```

```
┌──────────────────────────────────────┬──────────────────┬───────┬──────────────────────────────┐
│ cb2b453f-1a06-4f5e-abe1-b…           │ value is uneven  │     1 │ 2025-04-10 09:53:29.78+02    │
cb2b453f-1a06-4f5e-abe1-bfd413581bcf │ value is uneven  │     1 │ 2025-04-10 09:53:29.78+02 │
...
```

---

except_

Signature

```
except_(self: duckdb.duckdb.DuckDBPyRelation, other_rel: duckdb.duckdb.DuckDBPyRelation) ->
duckdb.duckdb.DuckDBPyRelation
```

Description Create the set except of this relation object with another relation object in other_rel

Parameters

- **other_rel** : duckdb.duckdb.DuckDBPyRelation

  The relation to subtract from the current relation (set difference).

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.except_(other_rel=rel.set_alias("other_rel"))
```

Result

The relation query is executed twice, therefore generating different ids and timestamps:

id uuid	description varchar	value int64	created_timestamp timestamp with time zone
f69ed6dd-a7fe-4de2-b6af-1c2418096d69	value is uneven	3	2025-04-10 11:43:05.711+02
08ad11dc-a9c2-4aaa-9272-760b27ad1f5d	value is uneven	7	2025-04-10 11:47:05.711+02

```
...
```

---

filter

Signature

```
filter(self: duckdb.duckdb.DuckDBPyRelation, filter_expr: object) -> duckdb.duckdb.DuckDBPyRelation
```

Description Filter the relation object by the filter in filter_expr

Parameters

- **filter_expr** : str, Expression

  The filter expression to apply over the relation.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.filter("value = 2")
```

Result

id uuid	description varchar	value int64	created_timestamp timestamp with time zone
b0684ab7-fcbf-41c5-8e4a-a51bdde86926	value is even	2	2025-04-10 09:54:29.78+02

`insert`

Signature

```
insert(self: duckdb.duckdb.DuckDBPyRelation, values: object) -> None
```

Description Inserts the given values into the relation

Parameters

- **values** : object

  A tuple of values matching the relation column list, to be inserted.

Example

```
import duckdb

from datetime import datetime
from uuid import uuid4

duckdb_conn = duckdb.connect()

duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
).to_table("code_example")

rel = duckdb_conn.table("code_example")

rel.insert(
    (
        uuid4(),
        'value is even',
        10,
        datetime.now()
    )
)

rel.filter("value = 10")
```

Result

id uuid	description varchar	value int64	created_timestamp timestamp with time zone
c6dfab87-fae6-4213-8f76-1b96a8d179f6	value is even	10	2025-04-10 10:02:24.652218+02

`insert_into`

Signature

```
insert_into(self: duckdb.duckdb.DuckDBPyRelation, table_name: str) -> None
```

Description Inserts the relation object into an existing table named table_name

Parameters

- **table_name** : str

  The table name to insert the data into. The relation must respect the column order of the table.

Example

```
import duckdb

from datetime import datetime
from uuid import uuid4

duckdb_conn = duckdb.connect()

duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
).to_table("code_example")

rel = duckdb_conn.values(
    [
        uuid4(),
        'value is even',
        10,
        datetime.now()
    ]
)

rel.insert_into("code_example")

duckdb_conn.table("code_example").filter("value = 10")
```

Result

id uuid	description varchar	value int64	created_timestamp timestamp with time zone
271c5ddd-c1d5-4638-b5a0-d8c7dc9e8220	value is even	10	2025-04-10 14:29:18.616379+02

`intersect`

Signature

```
intersect(self: duckdb.duckdb.DuckDBPyRelation, other_rel: duckdb.duckdb.DuckDBPyRelation) ->
duckdb.duckdb.DuckDBPyRelation
```

Description Create the set intersection of this relation object with another relation object in other_rel

Parameters

- **other_rel** : duckdb.duckdb.DuckDBPyRelation

  The relation to intersect with the current relation (set intersection).

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.intersect(other_rel=rel.set_alias("other_rel"))
```

Result

```
The relation query is executed once with `rel` and once with `other_rel`,
therefore generating different ids and timestamps:
```

id uuid	description varchar	value int64	created_timestamp timestamp with time zone
0 rows			

---

join

Signature

```
join(self: duckdb.duckdb.DuckDBPyRelation, other_rel: duckdb.duckdb.DuckDBPyRelation, condition: object,
how: str = 'inner') -> duckdb.duckdb.DuckDBPyRelation
```

Description Join the relation object with another relation object in other_rel using the join condition expression in join_condition. Types supported are 'inner', 'left', 'right', 'outer', 'semi' and 'anti'

Depending on how the `condition` parameter is provided, the JOIN clause generated is:

- USING

```
import duckdb

duckdb_conn = duckdb.connect()

rel1 = duckdb_conn.sql("select range as id, concat('dummy 1', range) as text from range(1,10)")
rel2 = duckdb_conn.sql("select range as id, concat('dummy 2', range) as text from range(5,7)")

rel1.join(rel2, condition="id", how="inner").sql_query()
```

with following SQL:

```
SELECT *
FROM (
        SELECT "range" AS id,
            concat('dummy 1', "range") AS "text"
```

```
        FROM "range"(1, 10)
    ) AS unnamed_relation_41bc15e744037078
INNER JOIN (
        SELECT "range" AS id,
        concat('dummy 2', "range") AS "text"
        FROM "range"(5, 7)
    ) AS unnamed_relation_307e245965aa2c2b
USING (id)
```

- ON

```
import duckdb

duckdb_conn = duckdb.connect()

rel1 = duckdb_conn.sql("select range as id, concat('dummy 1', range) as text from range(1,10)")
rel2 = duckdb_conn.sql("select range as id, concat('dummy 2', range) as text from range(5,7)")

rel1.join(rel2, condition=f"{rel1.alias}.id = {rel2.alias}.id", how="inner").sql_query()
```

with the following SQL:

```
SELECT *
FROM (
        SELECT "range" AS id,
            concat('dummy 1', "range") AS "text"
        FROM "range"(1, 10)
    ) AS unnamed_relation_41bc15e744037078
INNER JOIN (
        SELECT "range" AS id,
        concat('dummy 2', "range") AS "text"
        FROM "range"(5, 7)
    ) AS unnamed_relation_307e245965aa2c2b
ON ((unnamed_relation_41bc15e744037078.id = unnamed_relation_307e245965aa2c2b.id))
```

> NATURAL, POSITIONAL and ASOF joins are not provided by the relational API. CROSS joins are provided through the cross method.

Parameters

- **other_rel** : duckdb.duckdb.DuckDBPyRelation

  The relation to join with the current relation.

- **condition** : object

  The join condition, typically a SQL expression or the duplicated column name to join on.

- **how** : str, default: 'inner'

  The type of join to perform: 'inner', 'left', 'right', 'outer', 'semi' and 'anti'.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
```

```
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel = rel.set_alias("rel").join(
    other_rel=rel.set_alias("other_rel"),
    condition="rel.id = other_rel.id",
    how="left"
)

rel.count("*")
```

Result

```
┌──────────────┐
│ count_star() │
│    int64     │
├──────────────┤
│            9 │
└──────────────┘
```

`limit`

Signature

```
limit(self: duckdb.duckdb.DuckDBPyRelation, n: int, offset: int = 0) -> duckdb.duckdb.DuckDBPyRelation
```

Description Only retrieve the first n rows from this relation object, starting at offset

Parameters

- **n** : int

  The maximum number of rows to return.

- **offset** : int, default: 0

  The number of rows to skip before starting to return rows.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.limit(1)
```

Result

id uuid	description varchar	value int64	created_timestamp timestamp with time zone
4135597b-29e7-4cb9-a443-41f3d54f25df	value is uneven	1	2025-04-10 10:52:03.678+02

---

map

Signature

```
map(self: duckdb.duckdb.DuckDBPyRelation, map_function: Callable, *, schema: typing.Optional[object] =
None) -> duckdb.duckdb.DuckDBPyRelation
```

Description Calls the passed function on the relation

Parameters

- **map_function** : Callable

  A Python function that takes a DataFrame and returns a transformed DataFrame.

- **schema** : object, default: None

  Optional schema describing the structure of the output relation.

Example

```
import duckdb
from pandas import DataFrame

def multiply_by_2(df: DataFrame):
    df["id"] = df["id"] * 2
    return df

duckdb_conn = duckdb.connect()
rel = duckdb_conn.sql("select range as id, 'dummy' as text from range(1,3)")

rel.map(multiply_by_2, schema={"id": int, "text": str})
```

Result

id int64	text varchar
2	dummy
4	dummy

---

order

Signature

```
order(self: duckdb.duckdb.DuckDBPyRelation, order_expr: str) -> duckdb.duckdb.DuckDBPyRelation
```

Description Reorder the relation object by order_expr

Parameters

- **order_expr** : str

  SQL expression defining the ordering of the result rows.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.order("value desc").limit(1, offset=4)
```

Result

id uuid	description varchar	value int64	created_timestamp timestamp with time zone
55899131-e3d3-463c-a215-f65cb8aef3bf	value is uneven	5	2025-04-10 10:56:03.678+02

---

project

Signature

```
project(self: duckdb.duckdb.DuckDBPyRelation, *args, groups: str = '') -> duckdb.duckdb.DuckDBPyRelation
```

Description Project the relation object by the projection in project_expr

Aliases: `select`

Parameters

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `group by`.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.project("description").limit(1)
```

Result

```
┌──────────────────┐
│   description    │
│     varchar      │
├──────────────────┤
│ value is uneven  │
└──────────────────┘
```

`select`

Signature

```
select(self: duckdb.duckdb.DuckDBPyRelation, *args, groups: str = '') -> duckdb.duckdb.DuckDBPyRelation
```

Description Project the relation object by the projection in project_expr

Aliases: `project`

Parameters

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `group by`.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
```

```
    """
)

rel.select("description").limit(1)
```

Result

description
varchar
value is uneven

---

sort

Signature

```
sort(self: duckdb.duckdb.DuckDBPyRelation, *args) -> duckdb.duckdb.DuckDBPyRelation
```

Description Reorder the relation object by the provided expressions

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.sort("description")
```

Result

id uuid	description varchar	value int64	created_timestamp timestamp with time zone
5e0dfa8c-de4d-4ccd-8cff-450dabb86bde	value is even	6	2025-04-10 16:52:15.605+02
95f1ad48-facf-4a84-a971-0a4fecce68c7	value is even	2	2025-04-10 16:48:15.605+02

...

---

union

Signature

```
union(self: duckdb.duckdb.DuckDBPyRelation, union_rel: duckdb.duckdb.DuckDBPyRelation) ->
duckdb.duckdb.DuckDBPyRelation
```

Description Create the set union of this relation object with another relation object in other_rel

> The union is `union all`. In order to retrieve distinct values, apply distinct.

Parameters

- **union_rel** : duckdb.duckdb.DuckDBPyRelation

  The relation to union with the current relation (set union).

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel = rel.union(union_rel=rel)

rel.count("*")
```

Result

count_star()
int64
18

---

update

Signature

```
update(self: duckdb.duckdb.DuckDBPyRelation, set: object, *, condition: object = None) -> None
```

Description Update the given relation with the provided expressions

Parameters

- **set** : object

  Mapping of columns to new values for the update operation.

- **condition** : object, default: None

  Optional condition to filter which rows to update.

Example

```
import duckdb

from duckdb import ColumnExpression

duckdb_conn = duckdb.connect()

duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
).to_table("code_example")

rel = duckdb_conn.table("code_example")

rel.update(set={"description":None}, condition=ColumnExpression("value") == 1)

# the update is executed on the table, but not reflected on the relationship
# the relationship has to be recreated to retrieve the modified data
rel = duckdb_conn.table("code_example")

rel.show()
```

Result

```
┌──────────────────────────────────────┬──────────────┬─────────┬──────────────────────────────┐
│                  id                  │ description  │  value  │      created_timestamp       │
│                 uuid                 │   varchar    │  int64  │ timestamp with time zone     │
├──────────────────────────────────────┼──────────────┼─────────┼──────────────────────────────┤
│ 66dcaa14-f4a6-4a55-af3b-7f6aa23ab4ad │ NULL         │       1 │ 2025-04-10 16:54:49.317+02   │
│ c6a18a42-67fb-4c95-827b-c966f2f95b88 │ value is even│       2 │ 2025-04-10 16:55:49.317+02   │
...
```

Functions

This section contains the functions which can be applied to a relation, in order to get a (scalar) result. The functions are lazy evaluated.

Name	Description
`any_value`	Returns the first non-null value from a given column
`arg_max`	Finds the row with the maximum value for a value column and returns the value of that row for an argument column

Name	Description
arg_min	Finds the row with the minimum value for a value column and returns the value of that row for an argument column
avg	Computes the average on a given column
bit_and	Computes the bitwise AND of all bits present in a given column
bit_or	Computes the bitwise OR of all bits present in a given column
bit_xor	Computes the bitwise XOR of all bits present in a given column
bitstring_agg	Computes a bitstring with bits set for each distinct value in a given column
bool_and	Computes the logical AND of all values present in a given column
bool_or	Computes the logical OR of all values present in a given column
count	Computes the number of elements present in a given column
cume_dist	Computes the cumulative distribution within the partition
dense_rank	Computes the dense rank within the partition
distinct	Retrieve distinct rows from this relation object
favg	Computes the average of all values present in a given column using a more accurate floating point summation (Kahan Sum)
first	Returns the first value of a given column
first_value	Computes the first value within the group or partition
fsum	Computes the sum of all values present in a given column using a more accurate floating point summation (Kahan Sum)
geomean	Computes the geometric mean over all values present in a given column
histogram	Computes the histogram over all values present in a given column
lag	Computes the lag within the partition
last	Returns the last value of a given column
last_value	Computes the last value within the group or partition
lead	Computes the lead within the partition
list	Returns a list containing all values present in a given column
max	Returns the maximum value present in a given column
mean	Computes the average on a given column
median	Computes the median over all values present in a given column
min	Returns the minimum value present in a given column
mode	Computes the mode over all values present in a given column
n_tile	Divides the partition as equally as possible into num_buckets
nth_value	Computes the nth value within the partition
percent_rank	Computes the relative rank within the partition
product	Returns the product of all values present in a given column
quantile	Computes the exact quantile value for a given column
quantile_cont	Computes the interpolated quantile value for a given column
quantile_disc	Computes the exact quantile value for a given column
rank	Computes the rank within the partition
rank_dense	Computes the dense rank within the partition

Name	Description
`row_number`	Computes the row number within the partition
`select_dtypes`	Select columns from the relation, by filtering based on type(s)
`select_types`	Select columns from the relation, by filtering based on type(s)
`std`	Computes the sample standard deviation for a given column
`stddev`	Computes the sample standard deviation for a given column
`stddev_pop`	Computes the population standard deviation for a given column
`stddev_samp`	Computes the sample standard deviation for a given column
`string_agg`	Concatenates the values present in a given column with a separator
`sum`	Computes the sum of all values present in a given column
`unique`	Returns the distinct values in a column.
`value_counts`	Computes the number of elements present in a given column, also projecting the original column
`var`	Computes the sample variance for a given column
`var_pop`	Computes the population variance for a given column
`var_samp`	Computes the sample variance for a given column
`variance`	Computes the sample variance for a given column

any_value

Signature

```
any_value(self: duckdb.duckdb.DuckDBPyRelation, column: str, groups: str = '', window_spec: str = '',
projected_columns: str = '') -> duckdb.duckdb.DuckDBPyRelation
```

Description Returns the first non-null value from a given column

Parameters

- **column** : str

  The column name from which to retrieve any value.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `group by`.

- **window_spec** : str, default: ''

  Optional window specification for window functions, provided as `over (partition by ... order by ...)`.

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
```

```
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.any_value('id')
```

Result

any_value(id)
uuid
642ea3d7-793d-4867-a759-91c1226c25a0

---

arg_max

Signature

```
arg_max(self: duckdb.duckdb.DuckDBPyRelation, arg_column: str, value_column: str, groups: str = '',
window_spec: str = '', projected_columns: str = '') -> duckdb.duckdb.DuckDBPyRelation
```

Description Finds the row with the maximum value for a value column and returns the value of that row for an argument column

Parameters

- **arg_column** : str

  The column name for which to find the argument maximizing the value.

- **value_column** : str

  The column name containing values used to determine the maximum.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `group by`.

- **window_spec** : str, default: ''

  Optional window specification for window functions, provided as `over (partition by ... order by ...)`.

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
```

```
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.arg_max(arg_column="value", value_column="value", groups="description", projected_
columns="description")
```

Result

description varchar	arg_max("value", "value") int64
value is uneven	9
value is even	8

---

arg_min

Signature

```
arg_min(self: duckdb.duckdb.DuckDBPyRelation, arg_column: str, value_column: str, groups: str = '',
window_spec: str = '', projected_columns: str = '') -> duckdb.duckdb.DuckDBPyRelation
```

Description Finds the row with the minimum value for a value column and returns the value of that row for an argument column

Parameters

- **arg_column** : str

  The column name for which to find the argument minimizing the value.

- **value_column** : str

  The column name containing values used to determine the minimum.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `group by`.

- **window_spec** : str, default: ''

  Optional window specification for window functions, provided as `over (partition by ... order by ...)`

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()
```

```
rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.arg_min(arg_column="value", value_column="value", groups="description", projected_
columns="description")
```

Result

description varchar	arg_min("value", "value") int64
value is even	2
value is uneven	1

---

avg

Signature

```
avg(self: duckdb.duckdb.DuckDBPyRelation, column: str, groups: str = '', window_spec: str = '',
projected_columns: str = '') -> duckdb.duckdb.DuckDBPyRelation
```

Description Computes the average on a given column

Parameters

- **column** : str

  The column name to calculate the average on.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `group by`.

- **window_spec** : str, default: ''

  Optional window specification for window functions, provided as `over (partition by ... order by ...)`

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
```

```
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.avg('value')
```

Result

```
┌──────────────┐
│ avg("value") │
│    double    │
├──────────────┤
│          5.0 │
└──────────────┘
```

`bit_and`

Signature

```
bit_and(self: duckdb.duckdb.DuckDBPyRelation, column: str, groups: str = '', window_spec: str = '',
projected_columns: str = '') -> duckdb.duckdb.DuckDBPyRelation
```

Description Computes the bitwise AND of all bits present in a given column

Parameters

- **column** : str

  The column name to perform the bitwise AND aggregation on.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `group by`.

- **window_spec** : str, default: ''

  Optional window specification for window functions, provided as `over (partition by ... order by ...)`

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
```

```
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel = rel.select("description, value::bit as value_bit")

rel.bit_and(column="value_bit", groups="description", projected_columns="description")
```

Result

description varchar	bit_and(value_bit) bit
value is uneven	00000000000000000000000000000000000000000000000000000000000000001
value is even	00000000000000000000000000000000000000000000000000000000000000000

---

bit_or

Signature

```
bit_or(self: duckdb.duckdb.DuckDBPyRelation, column: str, groups: str = '', window_spec: str = '',
projected_columns: str = '') -> duckdb.duckdb.DuckDBPyRelation
```

Description Computes the bitwise OR of all bits present in a given column

Parameters

- **column** : str

  The column name to perform the bitwise OR aggregation on.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `group by`.

- **window_spec** : str, default: ''

  Optional window specification for window functions, provided as `over (partition by ... order by ...)`

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
```

```
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel = rel.select("description, value::bit as value_bit")

rel.bit_or(column="value_bit", groups="description", projected_columns="description")
```

Result

description varchar	bit_or(value_bit) bit
value is uneven	00000000000000000000000000000000000000000000000000000000000001111
value is even	00000000000000000000000000000000000000000000000000000000000001110

---

bit_xor

Signature

```
bit_xor(self: duckdb.duckdb.DuckDBPyRelation, column: str, groups: str = '', window_spec: str = '',
projected_columns: str = '') -> duckdb.duckdb.DuckDBPyRelation
```

Description Computes the bitwise XOR of all bits present in a given column

Parameters

- **column** : str

  The column name to perform the bitwise XOR aggregation on.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `group by`.

- **window_spec** : str, default: ''

  Optional window specification for window functions, provided as `over (partition by ... order by ...)`

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
```

```
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel = rel.select("description, value::bit as value_bit")

rel.bit_xor(column="value_bit", groups="description", projected_columns="description")
```

Result

description varchar	bit_xor(value_bit) bit
value is even	0000000000000000000000000000000000000000000000000000000000001000
value is uneven	0000000000000000000000000000000000000000000000000000000000001001

---

bitstring_agg

Signature

```
bitstring_agg(self: duckdb.duckdb.DuckDBPyRelation, column: str, min: typing.Optional[object] = None,
max: typing.Optional[object] = None, groups: str = '', window_spec: str = '', projected_columns: str =
'') -> duckdb.duckdb.DuckDBPyRelation
```

Description Computes a bitstring with bits set for each distinct value in a given column

Parameters

- **column** : str

  The column name to aggregate as a bitstring.

- **min** : object, default: None

  Optional minimum bitstring value for aggregation.

- **max** : object, default: None

  Optional maximum bitstring value for aggregation.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `group by`.

- **window_spec** : str, default: ''

  Optional window specification for window functions, provided as `over (partition by ... order by ...)`

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.bitstring_agg(column="value", groups="description", projected_columns="description", min=1, max=9)
```

Result

description varchar	bitstring_agg("value") bit
value is uneven value is even	101010101 010101010

---

bool_and

Signature

```
bool_and(self: duckdb.duckdb.DuckDBPyRelation, column: str, groups: str = '', window_spec: str = '',
projected_columns: str = '') -> duckdb.duckdb.DuckDBPyRelation
```

Description Computes the logical AND of all values present in a given column

Parameters

- **column** : str

  The column name to perform the boolean AND aggregation on.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `group by`.

- **window_spec** : str, default: ''

  Optional window specification for window functions, provided as `over (partition by ... order by ...)`

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel = rel.select("description, mod(value,2)::boolean as uneven")

rel.bool_and(column="uneven", groups="description", projected_columns="description")
```

Result

description varchar	bool_and(uneven) boolean
value is even	false
value is uneven	true

`bool_or`

Signature

```
bool_or(self: duckdb.duckdb.DuckDBPyRelation, column: str, groups: str = '', window_spec: str = '',
projected_columns: str = '') -> duckdb.duckdb.DuckDBPyRelation
```

Description Computes the logical OR of all values present in a given column

Parameters

- **column** : str

  The column name to perform the boolean OR aggregation on.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `group by`.

- **window_spec** : str, default: ''

  Optional window specification for window functions, provided as `over (partition by ... order by ...)`

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel = rel.select("description, mod(value,2)::boolean as uneven")

rel.bool_or(column="uneven", groups="description", projected_columns="description")
```

Result

description varchar	bool_or(uneven) boolean
value is even	false
value is uneven	true

---

count

Signature

```
count(self: duckdb.duckdb.DuckDBPyRelation, column: str, groups: str = '', window_spec: str = '',
projected_columns: str = '') -> duckdb.duckdb.DuckDBPyRelation
```

Description Computes the number of elements present in a given column

Parameters

- **column** : str

  The column name to perform count on.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `group by`.

- **window_spec** : str, default: ''

  Optional window specification for window functions, provided as `over (partition by ... order by ...)`

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.count("id")
```

Result

```
┌──────────┐
│ count(id) │
│   int64   │
├──────────┤
│         9 │
└──────────┘
```

---

cume_dist

Signature

```
cume_dist(self: duckdb.duckdb.DuckDBPyRelation, window_spec: str, projected_columns: str = '') ->
duckdb.duckdb.DuckDBPyRelation
```

Description Computes the cumulative distribution within the partition

Parameters

- **window_spec** : str

  Optional window specification for window functions, provided as `over (partition by ... order by ...)`

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
```

```
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.cume_dist(window_spec="over (partition by description order by value)", projected_
columns="description, value")
```

Result

description varchar	value int64	cume_dist() OVER (PARTITION BY description ORDER BY "value") double
value is uneven	1	0.2
value is uneven	3	0.4
value is uneven	5	0.6
value is uneven	7	0.8
value is uneven	9	1.0
value is even	2	0.25
value is even	4	0.5
value is even	6	0.75
value is even	8	1.0

---

dense_rank

Signature

```
dense_rank(self: duckdb.duckdb.DuckDBPyRelation, window_spec: str, projected_columns: str = '') ->
duckdb.duckdb.DuckDBPyRelation
```

Description Computes the dense rank within the partition

Aliases: `rank_dense`

Parameters

- **window_spec** : str

  Optional window specification for window functions, provided as `over (partition by ... order by ...)`

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
```

```
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

 rel.dense_rank(window_spec="over (partition by description order by value)", projected_
columns="description, value")
```

Result

description varchar	value int64	dense_rank() OVER (PARTITION BY description ORDER BY "value") int64
value is even	2	1
value is even	4	2
value is even	6	3
value is even	8	4
value is uneven	1	1
value is uneven	3	2
value is uneven	5	3
value is uneven	7	4
value is uneven	9	5

`distinct`

Signature

```
distinct(self: duckdb.duckdb.DuckDBPyRelation) -> duckdb.duckdb.DuckDBPyRelation
```

Description Retrieve distinct rows from this relation object

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("select range from range(1,4)")

rel = rel.union(union_rel=rel)

rel.distinct().order("range")
```

Result

range int64
1
2

```
┌──────────┐
│    3     │
└──────────┘
```

---

favg

Signature

```
favg(self: duckdb.duckdb.DuckDBPyRelation, column: str, groups: str = '', window_spec: str = '',
projected_columns: str = '') -> duckdb.duckdb.DuckDBPyRelation
```

Description Computes the average of all values present in a given column using a more accurate floating point summation (Kahan Sum)

Parameters

- **column** : str

  The column name to calculate the average on.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `group by`.

- **window_spec** : str, default: ''

  Optional window specification for window functions, provided as `over (partition by ... order by ...)`

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.favg(column="value", groups="description", projected_columns="description")
```

Result

description varchar	favg("value") double
value is uneven	5.0
value is even	5.0

`first`

Signature

```
first(self: duckdb.duckdb.DuckDBPyRelation, column: str, groups: str = '', projected_columns: str = '')
-> duckdb.duckdb.DuckDBPyRelation
```

Description Returns the first value of a given column

Parameters

- **column** : str

  The column name from which to retrieve the first value.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `group by`.

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.first(column="value", groups="description", projected_columns="description")
```

Result

description varchar	"first"("value") int64
value is even	2
value is uneven	1

`first_value`

Signature

```
first_value(self: duckdb.duckdb.DuckDBPyRelation, column: str, window_spec: str = '', projected_columns: str = '') -> duckdb.duckdb.DuckDBPyRelation
```

Description Computes the first value within the group or partition

Parameters

- **column** : str

  The column name from which to retrieve the first value.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `group by`.

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.first_value(column="value", window_spec="over (partition by description order by value)", projected_columns="description").distinct()
```

Result

description varchar	first_value("value") OVER (PARTITION BY description ORDER BY "value") int64
value is even	2
value is uneven	1

`fsum`

Signature

```
fsum(self: duckdb.duckdb.DuckDBPyRelation, column: str, groups: str = '', window_spec: str = '', projected_columns: str = '') -> duckdb.duckdb.DuckDBPyRelation
```

Description Computes the sum of all values present in a given column using a more accurate floating point summation (Kahan Sum)

Parameters

- **column** : str

  The column name to calculate the sum on.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `group by`.

- **window_spec** : str, default: ''

  Optional window specification for window functions, provided as `over (partition by ... order by ...)`

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.fsum(column="value", groups="description", projected_columns="description")
```

Result

description varchar	fsum("value") double
value is even	20.0
value is uneven	25.0

---

geomean

Signature

```
geomean(self: duckdb.duckdb.DuckDBPyRelation, column: str, groups: str = '', projected_columns: str =
'') -> duckdb.duckdb.DuckDBPyRelation
```

Description Computes the geometric mean over all values present in a given column

Parameters

- **column** : str

  The column name to calculate the geometric mean on.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `group by`.

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.geomean(column="value", groups="description", projected_columns="description")
```

Result

description varchar	geomean("value") double
value is uneven	3.936283427035351
value is even	4.426727678801287

`histogram`

Signature

```
histogram(self: duckdb.duckdb.DuckDBPyRelation, column: str, groups: str = '', window_spec: str = '',
projected_columns: str = '') -> duckdb.duckdb.DuckDBPyRelation
```

Description Computes the histogram over all values present in a given column

Parameters

- **column** : str

  The column name to calculate the histogram on.

- **window_spec** : str, default: ''

  Optional window specification for window functions, provided as `over (partition by ... order by ...)`

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.histogram(column="value", groups="description", projected_columns="description")
```

Result

description varchar	histogram("value") map(bigint, ubigint)
value is uneven	{1=1, 3=1, 5=1, 7=1, 9=1}
value is even	{2=1, 4=1, 6=1, 8=1}

`lag`

Signature

```
lag(self: duckdb.duckdb.DuckDBPyRelation, column: str, window_spec: str, offset: int = 1, default_value:
str = 'NULL', ignore_nulls: bool = False, projected_columns: str = '') -> duckdb.duckdb.DuckDBPyRelation
```

Description Computes the lag within the partition

Parameters

- **column** : str

  The column name to apply the lag function on.

- **window_spec** : str

  Optional window specification for window functions, provided as `over (partition by ... order by ...)`

- **offset** : int, default: 1

  The number of rows to lag behind.

- **default_value** : str, default: 'NULL'

  The default value to return when the lag offset goes out of bounds.

- **ignore_nulls** : bool, default: False

  Whether to ignore NULL values when computing the lag.

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.lag(column="description", window_spec="over (order by value)", projected_columns="description,
value")
```

Result

description varchar	value int64	lag(description, 1, NULL) OVER (ORDER BY "value") varchar
value is uneven	1	NULL
value is even	2	value is uneven
value is uneven	3	value is even
value is even	4	value is uneven
value is uneven	5	value is even
value is even	6	value is uneven
value is uneven	7	value is even
value is even	8	value is uneven
value is uneven	9	value is even

`last`

Signature

```
last(self: duckdb.duckdb.DuckDBPyRelation, column: str, groups: str = '', projected_columns: str = '')
-> duckdb.duckdb.DuckDBPyRelation
```

Description Returns the last value of a given column

Parameters

- **column** : str

  The column name from which to retrieve the last value.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `group by`.

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.last(column="value", groups="description", projected_columns="description")
```

Result

description varchar	"last"("value") int64
value is even	8
value is uneven	9

`last_value`

Signature

```
last_value(self: duckdb.duckdb.DuckDBPyRelation, column: str, window_spec: str = '', projected_columns:
str = '') -> duckdb.duckdb.DuckDBPyRelation
```

Description Computes the last value within the group or partition

Parameters

- **column** : str

  The column name from which to retrieve the last value within the window.

- **window_spec** : str, default: ''

  Optional window specification for window functions, provided as `over (partition by ... order by ...)`

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.last_value(column="value", window_spec="over (order by description)", projected_
columns="description").distinct()
```

Result

description varchar	last_value("value") OVER (ORDER BY description) int64
value is uneven	9
value is even	8

`lead`

Signature

```
lead(self: duckdb.duckdb.DuckDBPyRelation, column: str, window_spec: str, offset: int = 1, default_value:
str = 'NULL', ignore_nulls: bool = False, projected_columns: str = '') -> duckdb.duckdb.DuckDBPyRelation
```

Description Computes the lead within the partition

Parameters

- **column** : str

  The column name to apply the lead function on.

- **window_spec** : str

  Optional window specification for window functions, provided as `over (partition by ... order by ...)`

- **offset** : int, default: 1

  The number of rows to lead ahead.

- **default_value** : str, default: 'NULL'

  The default value to return when the lead offset goes out of bounds.

- **ignore_nulls** : bool, default: False

  Whether to ignore NULL values when computing the lead.

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.lead(column="description", window_spec="over (order by value)", projected_columns="description,
value")
```

Result

description varchar	value int64	lead(description, 1, NULL) OVER (ORDER BY "value") varchar
value is uneven	1	value is even
value is even	2	value is uneven
value is uneven	3	value is even
value is even	4	value is uneven
value is uneven	5	value is even
value is even	6	value is uneven
value is uneven	7	value is even
value is even	8	value is uneven
value is uneven	9	NULL

`list`

Signature

```
list(self: duckdb.duckdb.DuckDBPyRelation, column: str, groups: str = '', window_spec: str = '',
projected_columns: str = '') -> duckdb.duckdb.DuckDBPyRelation
```

Description Returns a list containing all values present in a given column

Parameters

- **column** : str

  The column name to aggregate values into a list.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `group by`.

- **window_spec** : str, default: ''

  Optional window specification for window functions, provided as `over (partition by ... order by ...)`

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.list(column="value", groups="description", projected_columns="description")
```

Result

description varchar	list("value") int64[]
value is even	[2, 4, 6, 8]
value is uneven	[1, 3, 5, 7, 9]

---

max

Signature

```
max(self: duckdb.duckdb.DuckDBPyRelation, column: str, groups: str = '', window_spec: str = '',
projected_columns: str = '') -> duckdb.duckdb.DuckDBPyRelation
```

Description Returns the maximum value present in a given column

Parameters

- **column** : str

  The column name to calculate the maximum value of.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `group by`.

- **window_spec** : str, default: ''

  Optional window specification for window functions, provided as `over (partition by ... order by ...)`

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

 rel.max(column="value", groups="description", projected_columns="description")
```

Result

description varchar	max("value") int64
value is even	8
value is uneven	9

---

mean

Signature

```
mean(self: duckdb.duckdb.DuckDBPyRelation, column: str, groups: str = '', window_spec: str = '',
projected_columns: str = '') -> duckdb.duckdb.DuckDBPyRelation
```

Description Computes the average on a given column

Parameters

- **column** : str

  The column name to calculate the mean value of.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `group by`.

- **window_spec** : str, default: ''

  Optional window specification for window functions, provided as `over (partition by ... order by ...)`

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.mean(column="value", groups="description", projected_columns="description")
```

Result

description varchar	avg("value") double
value is even	5.0
value is uneven	5.0

---

median

Signature

```
median(self: duckdb.duckdb.DuckDBPyRelation, column: str, groups: str = '', window_spec: str = '',
projected_columns: str = '') -> duckdb.duckdb.DuckDBPyRelation
```

Description Computes the median over all values present in a given column

Parameters

- **column** : str

  The column name to calculate the median value of.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `group by`.

- **window_spec** : str, default: ''

  Optional window specification for window functions, provided as `over (partition by ... order by ...)`

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.median(column="value", groups="description", projected_columns="description")
```

Result

description varchar	median("value") double
value is even	5.0
value is uneven	5.0

`min`

Signature

```
min(self: duckdb.duckdb.DuckDBPyRelation, column: str, groups: str = '', window_spec: str = '',
projected_columns: str = '') -> duckdb.duckdb.DuckDBPyRelation
```

Description Returns the minimum value present in a given column

Parameters

- **column** : str

  The column name to calculate the min value of.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `group by`.

- **window_spec** : str, default: ''

  Optional window specification for window functions, provided as `over (partition by ... order by ...)`

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.min(column="value", groups="description", projected_columns="description")
```

Result

description varchar	min("value") int64
value is uneven	1
value is even	2

---

mode

Signature

```
mode(self: duckdb.duckdb.DuckDBPyRelation, column: str, groups: str = '', window_spec: str = '',
projected_columns: str = '') -> duckdb.duckdb.DuckDBPyRelation
```

Description Computes the mode over all values present in a given column

Parameters

- **column** : str

  The column name to calculate the mode (most frequent value) of.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `group by`.

- **window_spec** : str, default: ''

  Optional window specification for window functions, provided as `over (partition by ... order by ...)`

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.mode(column="value", groups="description", projected_columns="description")
```

Result

description varchar	"mode"("value") int64
value is uneven	1
value is even	2

---

n_tile

Signature

```
n_tile(self: duckdb.duckdb.DuckDBPyRelation, window_spec: str, num_buckets: int, projected_columns: str
= '') -> duckdb.duckdb.DuckDBPyRelation
```

Description Divides the partition as equally as possible into num_buckets

Parameters

- **window_spec** : str

  Optional window specification for window functions, provided as `over (partition by ... order by ...)`

- **num_buckets** : int

  The number of buckets to divide the rows into.

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.n_tile(window_spec="over (partition by description)", num_buckets=2, projected_columns="description,
value")
```

Result

description varchar	value int64	ntile(2) OVER (PARTITION BY description) int64
value is uneven	1	1
value is uneven	3	1
value is uneven	5	1
value is uneven	7	2
value is uneven	9	2
value is even	2	1
value is even	4	1
value is even	6	2
value is even	8	2

---

nth_value

Signature

```
nth_value(self: duckdb.duckdb.DuckDBPyRelation, column: str, window_spec: str, offset: int, ignore_
nulls: bool = False, projected_columns: str = '') -> duckdb.duckdb.DuckDBPyRelation
```

Description Computes the nth value within the partition

Parameters

- **column** : str

  The column name from which to retrieve the nth value within the window.

- **window_spec** : str

  Optional window specification for window functions, provided as `over (partition by ... order by ...)`

- **offset** : int

  The position of the value to retrieve within the window (1-based index).

- **ignore_nulls** : bool, default: False

  Whether to ignore NULL values when computing the nth value.

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.nth_value(column="value", window_spec="over (partition by description)", projected_
columns="description", offset=1)
```

Result

description varchar	nth_value("value", 1) OVER (PARTITION BY description) int64
value is even	2
value is even	2
value is even	2
value is even	2
value is uneven	1
value is uneven	1
value is uneven	1
value is uneven	1
value is uneven	1

percent_rank

Signature

```
percent_rank(self: duckdb.duckdb.DuckDBPyRelation, window_spec: str, projected_columns: str = '') ->
duckdb.duckdb.DuckDBPyRelation
```

Description Computes the relative rank within the partition

Parameters

- **window_spec** : str

  Optional window specification for window functions, provided as `over (partition by ... order by ...)`

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.percent_rank(window_spec="over (partition by description order by value)", projected_
columns="description, value")
```

Result

description varchar	value int64	percent_rank() OVER (PARTITION BY description ORDER BY "value") double
value is even	2	0.0
value is even	4	0.3333333333333333
value is even	6	0.6666666666666666
value is even	8	1.0
value is uneven	1	0.0
value is uneven	3	0.25
value is uneven	5	0.5
value is uneven	7	0.75
value is uneven	9	1.0

`product`

Signature

```
product(self: duckdb.duckdb.DuckDBPyRelation, column: str, groups: str = '', window_spec: str = '',
projected_columns: str = '') -> duckdb.duckdb.DuckDBPyRelation
```

Description Returns the product of all values present in a given column

Parameters

- **column** : str

  The column name to calculate the product of.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `group by`.

- **window_spec** : str, default: ''

  Optional window specification for window functions, provided as `over (partition by ... order by ...)`

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.product(column="value", groups="description", projected_columns="description")
```

Result

description varchar	product("value") double
value is uneven	945.0
value is even	384.0

`quantile`

Signature

```
quantile(self: duckdb.duckdb.DuckDBPyRelation, column: str, q: object = 0.5, groups: str = '', window_
spec: str = '', projected_columns: str = '') -> duckdb.duckdb.DuckDBPyRelation
```

Description Computes the exact quantile value for a given column

Parameters

- **column** : str

  The column name to compute the quantile for.

- **q** : object, default: 0.5

  The quantile value to compute (e.g., 0.5 for median).

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `group by`.

- **window_spec** : str, default: ''

  Optional window specification for window functions, provided as `over (partition by ... order by ...)`

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.quantile(column="value", groups="description", projected_columns="description")
```

Result

description varchar	quantile_disc("value", 0.500000) int64
value is uneven	5
value is even	4

quantile_cont

Signature

```
quantile_cont(self: duckdb.duckdb.DuckDBPyRelation, column: str, q: object = 0.5, groups: str = '',
window_spec: str = '', projected_columns: str = '') -> duckdb.duckdb.DuckDBPyRelation
```

Description Computes the interpolated quantile value for a given column

Parameters

- **column** : str

  The column name to compute the continuous quantile for.

- **q** : object, default: 0.5

  The quantile value to compute (e.g., 0.5 for median).

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `group by`.

- **window_spec** : str, default: ''

  Optional window specification for window functions, provided as `over (partition by ... order by ...)`

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.quantile_cont(column="value", groups="description", projected_columns="description")
```

Result

description varchar	quantile_cont("value", 0.500000) double
value is even	5.0
value is uneven	5.0

quantile_disc

Signature

```
quantile_disc(self: duckdb.duckdb.DuckDBPyRelation, column: str, q: object = 0.5, groups: str = '',
window_spec: str = '', projected_columns: str = '') -> duckdb.duckdb.DuckDBPyRelation
```

Description Computes the exact quantile value for a given column

Parameters

- **column** : str

  The column name to compute the discrete quantile for.

- **q** : object, default: 0.5

  The quantile value to compute (e.g., 0.5 for median).

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `group by`.

- **window_spec** : str, default: ''

  Optional window specification for window functions, provided as `over (partition by ... order by ...)`

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.quantile_disc(column="value", groups="description", projected_columns="description")
```

Result

description varchar	quantile_disc("value", 0.500000) int64
value is even	4
value is uneven	5

`rank`

Signature

```
rank(self: duckdb.duckdb.DuckDBPyRelation, window_spec: str, projected_columns: str = '') ->
duckdb.duckdb.DuckDBPyRelation
```

Description Computes the rank within the partition

Parameters

- **window_spec** : str

  Optional window specification for window functions, provided as `over (partition by ... order by ...)`

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.rank(window_spec="over (partition by description order by value)", projected_columns="description,
value")
```

Result

description varchar	value int64	rank() OVER (PARTITION BY description ORDER BY "value") int64
value is uneven	1	1
value is uneven	3	2
value is uneven	5	3
value is uneven	7	4
value is uneven	9	5
value is even	2	1
value is even	4	2
value is even	6	3
value is even	8	4

rank_dense

Signature

```
rank_dense(self: duckdb.duckdb.DuckDBPyRelation, window_spec: str, projected_columns: str = '') ->
duckdb.duckdb.DuckDBPyRelation
```

Description Computes the dense rank within the partition

Aliases: dense_rank

Parameters

- **window_spec** : str

  Optional window specification for window functions, provided as `over (partition by ... order by ...)`

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

 rel.rank_dense(window_spec="over (partition by description order by value)", projected_
columns="description, value")
```

Result

description varchar	value int64	dense_rank() OVER (PARTITION BY description ORDER BY "value") int64
value is uneven	1	1
value is uneven	3	2
value is uneven	5	3
value is uneven	7	4
value is uneven	9	5
value is even	2	1
value is even	4	2
value is even	6	3
value is even	8	4

`row_number`

Signature

```
row_number(self: duckdb.duckdb.DuckDBPyRelation, window_spec: str, projected_columns: str = '') ->
duckdb.duckdb.DuckDBPyRelation
```

Description Computes the row number within the partition

Parameters

- **window_spec** : str

  Optional window specification for window functions, provided as `over (partition by ... order by ...)`

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.row_number(window_spec="over (partition by description order by value)", projected_
columns="description, value")
```

Result

description varchar	value int64	row_number() OVER (PARTITION BY description ORDER BY "value") int64
value is uneven	1	1
value is uneven	3	2
value is uneven	5	3
value is uneven	7	4
value is uneven	9	5
value is even	2	1
value is even	4	2
value is even	6	3
value is even	8	4

select_dtypes

Signature

```
select_dtypes(self: duckdb.duckdb.DuckDBPyRelation, types: object) -> duckdb.duckdb.DuckDBPyRelation
```

Description Select columns from the relation, by filtering based on type(s)

Aliases: `select_types`

Parameters

- **types** : object

  Data type(s) to select columns by. Can be a single type or a collection of types.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.select_dtypes(types=[duckdb.typing.VARCHAR]).distinct()
```

Result

```
┌─────────────────┐
│   description   │
│     varchar     │
├─────────────────┤
│ value is even   │
│ value is uneven │
└─────────────────┘
```

---

select_types

Signature

```
select_types(self: duckdb.duckdb.DuckDBPyRelation, types: object) -> duckdb.duckdb.DuckDBPyRelation
```

Description Select columns from the relation, by filtering based on type(s)

Aliases: `select_dtypes`

Parameters

- **types** : object

  Data type(s) to select columns by. Can be a single type or a collection of types.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.select_types(types=[duckdb.typing.VARCHAR]).distinct()
```

Result

```
┌──────────────────┐
│   description    │
│     varchar      │
├──────────────────┤
│ value is even    │
│ value is uneven  │
└──────────────────┘
```

`std`

Signature

```
std(self: duckdb.duckdb.DuckDBPyRelation, column: str, groups: str = '', window_spec: str = '',
projected_columns: str = '') -> duckdb.duckdb.DuckDBPyRelation
```

Description Computes the sample standard deviation for a given column

Aliases: `stddev`, `stddev_samp`

Parameters

- **column** : str

  The column name to calculate the standard deviation for.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `group by`.

- **window_spec** : str, default: ''

  Optional window specification for window functions, provided as `over (partition by ... order by ...)`

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.std(column="value", groups="description", projected_columns="description")
```

Result

description varchar	stddev_samp("value") double
value is uneven	3.1622776601683795
value is even	2.581988897471611

---

stddev

Signature

```
stddev(self: duckdb.duckdb.DuckDBPyRelation, column: str, groups: str = '', window_spec: str = '',
projected_columns: str = '') -> duckdb.duckdb.DuckDBPyRelation
```

Description Computes the sample standard deviation for a given column

Aliases: `std`, `stddev_samp`

Parameters

- **column** : str

  The column name to calculate the standard deviation for.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `group by`.

- **window_spec** : str, default: ''

  Optional window specification for window functions, provided as `over (partition by ... order by ...)`

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.stddev(column="value", groups="description", projected_columns="description")
```

Result

description varchar	stddev_samp("value") double
value is even	2.581988897471611
value is uneven	3.1622776601683795

---

stddev_pop

Signature

```
stddev_pop(self: duckdb.duckdb.DuckDBPyRelation, column: str, groups: str = '', window_spec: str = '',
projected_columns: str = '') -> duckdb.duckdb.DuckDBPyRelation
```

Description Computes the population standard deviation for a given column

Parameters

- **column** : str

  The column name to calculate the standard deviation for.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `group by`.

- **window_spec** : str, default: ''

  Optional window specification for window functions, provided as `over (partition by ... order by ...)`

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.stddev_pop(column="value", groups="description", projected_columns="description")
```

Result

description varchar	stddev_pop("value") double
value is even	2.23606797749979
value is uneven	2.8284271247461903

`stddev_samp`

Signature

```
stddev_samp(self: duckdb.duckdb.DuckDBPyRelation, column: str, groups: str = '', window_spec: str = '',
projected_columns: str = '') -> duckdb.duckdb.DuckDBPyRelation
```

Description Computes the sample standard deviation for a given column

Aliases: `stddev`, `std`

Parameters

- **column** : str

  The column name to calculate the standard deviation for.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `group by`.

- **window_spec** : str, default: ''

  Optional window specification for window functions, provided as `over (partition by ... order by ...)`

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.stddev_samp(column="value", groups="description", projected_columns="description")
```

Result

description varchar	stddev_samp("value") double
value is even	2.581988897471611
value is uneven	3.1622776601683795

---

string_agg

Signature

```
string_agg(self: duckdb.duckdb.DuckDBPyRelation, column: str, sep: str = ',', groups: str = '', window_
spec: str = '', projected_columns: str = '') -> duckdb.duckdb.DuckDBPyRelation
```

Description Concatenates the values present in a given column with a separator

Parameters

- **column** : str

  The column name to concatenate values from.

- **sep** : str, default: ','

  Separator string to use between concatenated values.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `group by`.

- **window_spec** : str, default: ''

  Optional window specification for window functions, provided as `over (partition by ... order by ...)`

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.string_agg(column="value", sep=",", groups="description", projected_columns="description")
```

Result

description varchar	string_agg("value", ',') varchar
value is even	2,4,6,8
value is uneven	1,3,5,7,9

---

sum

Signature

```
sum(self: duckdb.duckdb.DuckDBPyRelation, column: str, groups: str = '', window_spec: str = '',
projected_columns: str = '') -> duckdb.duckdb.DuckDBPyRelation
```

Description Computes the sum of all values present in a given column

Parameters

- **column** : str

  The column name to calculate the sum for.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `group by`.

- **window_spec** : str, default: ''

  Optional window specification for window functions, provided as `over (partition by ... order by ...)`

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.sum(column="value", groups="description", projected_columns="description")
```

Result

description varchar	sum("value") int128
value is even	20
value is uneven	25

---

unique

Signature

```
unique(self: duckdb.duckdb.DuckDBPyRelation, unique_aggr: str) -> duckdb.duckdb.DuckDBPyRelation
```

Description Returns the distinct values in a column.

Parameters

- **unique_aggr** : str

  The column to get the distinct values for.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
```

```
)

rel.unique(unique_aggr="description")
```

Result

```
┌──────────────────┐
│   description    │
│     varchar      │
├──────────────────┤
│ value is even    │
│ value is uneven  │
└──────────────────┘
```

`value_counts`

Signature

```
value_counts(self: duckdb.duckdb.DuckDBPyRelation, column: str, groups: str = '') ->
duckdb.duckdb.DuckDBPyRelation
```

Description Computes the number of elements present in a given column, also projecting the original column

Parameters

- **column** : str

  The column name to count values from.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `group by`.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.value_counts(column="description", groups="description")
```

Result

description varchar	count(description) int64
value is uneven	5
value is even	4

---

var

Signature

```
var(self: duckdb.duckdb.DuckDBPyRelation, column: str, groups: str = '', window_spec: str = '',
projected_columns: str = '') -> duckdb.duckdb.DuckDBPyRelation
```

Description Computes the sample variance for a given column

Aliases: `variance`, `var_samp`

Parameters

- **column** : str

  The column name to calculate the sample variance for.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `group by`.

- **window_spec** : str, default: ''

  Optional window specification for window functions, provided as `over (partition by ... order by ...)`

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.var(column="value", groups="description", projected_columns="description")
```

Result

description varchar	var_samp("value") double
value is even	6.666666666666667
value is uneven	10.0

---

var_pop

Signature

```
var_pop(self: duckdb.duckdb.DuckDBPyRelation, column: str, groups: str = '', window_spec: str = '',
projected_columns: str = '') -> duckdb.duckdb.DuckDBPyRelation
```

Description Computes the population variance for a given column

Parameters

- **column** : str

  The column name to calculate the population variance for.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `group by`.

- **window_spec** : str, default: ''

  Optional window specification for window functions, provided as `over (partition by ... order by ...)`

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.var_pop(column="value", groups="description", projected_columns="description")
```

Result

description varchar	var_pop("value") double
value is even	5.0
value is uneven	8.0

`var_samp`

Signature

```
var_samp(self: duckdb.duckdb.DuckDBPyRelation, column: str, groups: str = '', window_spec: str = '',
projected_columns: str = '') -> duckdb.duckdb.DuckDBPyRelation
```

Description Computes the sample variance for a given column

Aliases: `variance`, `var`

Parameters

- **column** : str

  The column name to calculate the sample variance for.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `group by`.

- **window_spec** : str, default: ''

  Optional window specification for window functions, provided as `over (partition by ... order by ...)`

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.var_samp(column="value", groups="description", projected_columns="description")
```

Result

description varchar	var_samp("value") double
value is even	6.666666666666667
value is uneven	10.0

`variance`

Signature

```
variance(self: duckdb.duckdb.DuckDBPyRelation, column: str, groups: str = '', window_spec: str = '',
projected_columns: str = '') -> duckdb.duckdb.DuckDBPyRelation
```

Description Computes the sample variance for a given column

Aliases: `var`, `var_samp`

Parameters

- **column** : str

  The column name to calculate the sample variance for.

- **groups** : str, default: ''

  Comma-separated list of columns to include in the `group by`.

- **window_spec** : str, default: ''

  Optional window specification for window functions, provided as `over (partition by ... order by ...)`

- **projected_columns** : str, default: ''

  Comma-separated list of columns to include in the result.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.variance(column="value", groups="description", projected_columns="description")
```

Result

description varchar	var_samp("value") double
value is even	6.666666666666667
value is uneven	10.0

Output

This section contains the functions which will trigger an SQL execution and retrieve the data.

Name	Description
`arrow`	Execute and fetch all rows as an Arrow Table
`close`	Closes the result
`create`	Creates a new table named table_name with the contents of the relation object
`create_view`	Creates a view named view_name that refers to the relation object
`df`	Execute and fetch all rows as a pandas DataFrame
`execute`	Transform the relation into a result set
`fetch_arrow_reader`	Execute and return an Arrow Record Batch Reader that yields all rows
`fetch_arrow_table`	Execute and fetch all rows as an Arrow Table
`fetch_df_chunk`	Execute and fetch a chunk of the rows
`fetchall`	Execute and fetch all rows as a list of tuples
`fetchdf`	Execute and fetch all rows as a pandas DataFrame
`fetchmany`	Execute and fetch the next set of rows as a list of tuples
`fetchnumpy`	Execute and fetch all rows as a Python dict mapping each column to one numpy arrays
`fetchone`	Execute and fetch a single row as a tuple
`pl`	Execute and fetch all rows as a Polars DataFrame
`record_batch`	Execute and return an Arrow Record Batch Reader that yields all rows
`tf`	Fetch a result as dict of TensorFlow Tensors
`to_arrow_table`	Execute and fetch all rows as an Arrow Table
`to_csv`	Write the relation object to a CSV file in 'file_name'
`to_df`	Execute and fetch all rows as a pandas DataFrame
`to_parquet`	Write the relation object to a Parquet file in 'file_name'
`to_table`	Creates a new table named table_name with the contents of the relation object
`to_view`	Creates a view named view_name that refers to the relation object
`torch`	Fetch a result as dict of PyTorch Tensors
`write_csv`	Write the relation object to a CSV file in 'file_name'
`write_parquet`	Write the relation object to a Parquet file in 'file_name'

arrow

Signature

```
arrow(self: duckdb.duckdb.DuckDBPyRelation, batch_size: int = 1000000) -> pyarrow.lib.Table
```

Description Execute and fetch all rows as an Arrow Table

Aliases: `fetch_arrow_table`, `to_arrow_table`

Parameters

- **batch_size** : int, default: 1000000

  The batch size of writing the data to the Arrow table

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

pa_table = rel.arrow()

pa_table
```

Result

```
pyarrow.Table
id: string
description: string
value: int64
created_timestamp: timestamp[us, tz=Europe/Amsterdam]
----
id: [["3ac9e0ba-8390-4a02-ad72-33b1caea6354","8b844392-1404-4bbc-b731-120f42c8ca27","ca5584ca-8e97-
4fca-a295-ae3c16c32f5b","926d071e-5f64-488f-ae02-d19e315f9f5c","aabeedf0-5783-4eff-9963-
b3967a6ea5d8","1f20db9a-bee8-4b65-b7e8-e7c36b5b8fee","795c678e-3524-4b52-96ec-7b48c24eeab1","9ffbd403-
169f-4fe4-bc41-09751066f1f1","8fdb0a60-29f0-4f5b-afcc-c736a03cd083"]]
description: [["value is uneven","value is even","value is uneven","value is even","value is
uneven","value is even","value is uneven","value is even","value is uneven"]]
value: [[1,2,3,4,5,6,7,8,9]]
created_timestamp: [[2025-04-10 09:07:12.614000Z,2025-04-10 09:08:12.614000Z,2025-04-10
09:09:12.614000Z,2025-04-10 09:10:12.614000Z,2025-04-10 09:11:12.614000Z,2025-04-10
09:12:12.614000Z,2025-04-10 09:13:12.614000Z,2025-04-10 09:14:12.614000Z,2025-04-10 09:15:12.614000Z]]
```

close

Signature

```
close(self: duckdb.duckdb.DuckDBPyRelation) -> None
```

Description Closes the result

---

create

Signature

```
create(self: duckdb.duckdb.DuckDBPyRelation, table_name: str) -> None
```

Description Creates a new table named table_name with the contents of the relation object

Aliases: `to_table`

Parameters

- **table_name** : str

  The name of the table to be created. There shouldn't be any other table with the same name.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.create("table_code_example")

duckdb_conn.table("table_code_example").limit(1)
```

Result

id uuid	description varchar	value int64	created_timestamp timestamp with time zone
3ac9e0ba-8390-4a02-ad72-33b1caea6354	value is uneven	1	2025-04-10 11:07:12.614+02

`create_view`

Signature

```
create_view(self: duckdb.duckdb.DuckDBPyRelation, view_name: str, replace: bool = True) ->
duckdb.duckdb.DuckDBPyRelation
```

Description Creates a view named view_name that refers to the relation object

Aliases: `to_view`

Parameters

- **view_name** : str

  The name of the view to be created.

- **replace** : bool, default: True

  If the view should be created with `CREATE OR REPLACE`. When set to `False`, there shouldn't be another view with the same `view_name`.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.create_view("view_code_example", replace=True)

duckdb_conn.table("view_code_example").limit(1)
```

Result

id uuid	description varchar	value int64	created_timestamp timestamp with time zone
3ac9e0ba-8390-4a02-ad72-33b1caea6354	value is uneven	1	2025-04-10 11:07:12.614+02

`df`

Signature

```
df(self: duckdb.duckdb.DuckDBPyRelation, *, date_as_object: bool = False) -> pandas.DataFrame
```

Description Execute and fetch all rows as a pandas DataFrame

Aliases: `fetchdf`, `to_df`

Parameters

- **date_as_object** : bool, default: False

  If the date columns should be interpreted as Python date objects.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.df()
```

Result

```
                                     id      description  value                created_timestamp
0  3ac9e0ba-8390-4a02-ad72-33b1caea6354  value is uneven      1 2025-04-10 11:07:12.614000+02:00
1  8b844392-1404-4bbc-b731-120f42c8ca27    value is even      2 2025-04-10 11:08:12.614000+02:00
2  ca5584ca-8e97-4fca-a295-ae3c16c32f5b  value is uneven      3 2025-04-10 11:09:12.614000+02:00
...
```

---

execute

Signature

```
execute(self: duckdb.duckdb.DuckDBPyRelation) -> duckdb.duckdb.DuckDBPyRelation
```

Description Transform the relation into a result set

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
```

```
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.execute()
```

Result

id uuid	description varchar	value int64	created_timestamp timestamp with time zone
3ac9e0ba-8390-4a02-ad72-33b1caea6354	value is uneven	1	2025-04-10 11:07:12.614+02
8b844392-1404-4bbc-b731-120f42c8ca27	value is even	2	2025-04-10 11:08:12.614+02
ca5584ca-8e97-4fca-a295-ae3c16c32f5b	value is uneven	3	2025-04-10 11:09:12.614+02

`fetch_arrow_reader`

Signature

```
fetch_arrow_reader(self: duckdb.duckdb.DuckDBPyRelation, batch_size: int = 1000000) ->
pyarrow.lib.RecordBatchReader
```

Description Execute and return an Arrow Record Batch Reader that yields all rows

Parameters

- **batch_size** : int, default: 1000000

  The batch size for fetching the data.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

pa_reader = rel.fetch_arrow_reader(batch_size=1)

pa_reader.read_next_batch()
```

Result

```
pyarrow.RecordBatch
id: string
description: string
value: int64
created_timestamp: timestamp[us, tz=Europe/Amsterdam]
----
id: ["e4ab8cb4-4609-40cb-ad7e-4304ed5ed4bd"]
description: ["value is even"]
value: [2]
created_timestamp: [2025-04-10 09:25:51.259000Z]
```

`fetch_arrow_table`

Signature

```
fetch_arrow_table(self: duckdb.duckdb.DuckDBPyRelation, batch_size: int = 1000000) -> pyarrow.lib.Table
```

Description Execute and fetch all rows as an Arrow Table

Aliases: `arrow`, `to_arrow_table`

Parameters

- **batch_size** : int, default: 1000000

  The batch size for fetching the data.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.fetch_arrow_table()
```

Result

```
pyarrow.Table
id: string
description: string
value: int64
created_timestamp: timestamp[us, tz=Europe/Amsterdam]
----
```

```
id: [["1587b4b0-3023-49fe-82cf-06303ca136ac","e4ab8cb4-4609-40cb-ad7e-4304ed5ed4bd","3f8ad67a-290f-
4a22-b41b-0173b8e45afa","9a4e37ef-d8bd-46dd-ab01-51cf4973549f","12baa624-ebc9-45ae-b73e-
6f4029e31d2d","56d41292-53cc-48be-a1b8-e1f5d6ca5581","1accca18-c950-47c1-9108-aef8afbd5249","56d8db75-
72c4-4d40-90d2-a3c840579c37","e19f6201-8646-401c-b019-e37c42c39632"]]
description: [["value is uneven","value is even","value is uneven","value is even","value is
uneven","value is even","value is uneven","value is even","value is uneven"]]
value: [[1,2,3,4,5,6,7,8,9]]
created_timestamp: [[2025-04-10 09:24:51.259000Z,2025-04-10 09:25:51.259000Z,2025-04-10
09:26:51.259000Z,2025-04-10 09:27:51.259000Z,2025-04-10 09:28:51.259000Z,2025-04-10
09:29:51.259000Z,2025-04-10 09:30:51.259000Z,2025-04-10 09:31:51.259000Z,2025-04-10 09:32:51.259000Z]]
```

`fetch_df_chunk`

Signature

```
fetch_df_chunk(self: duckdb.duckdb.DuckDBPyRelation, vectors_per_chunk: int = 1, *, date_as_object: bool
= False) -> pandas.DataFrame
```

Description Execute and fetch a chunk of the rows

Parameters

- **vectors_per_chunk** : int, default: 1

  Number of data chunks to be processed before converting to dataframe.

- **date_as_object** : bool, default: False

  If the date columns should be interpreted as Python date objects.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.fetch_df_chunk()
```

Result

```
                                     id      description  value                created_timestamp
0  1587b4b0-3023-49fe-82cf-06303ca136ac  value is uneven      1 2025-04-10 11:24:51.259000+02:00
1  e4ab8cb4-4609-40cb-ad7e-4304ed5ed4bd    value is even      2 2025-04-10 11:25:51.259000+02:00
2  3f8ad67a-290f-4a22-b41b-0173b8e45afa  value is uneven      3 2025-04-10 11:26:51.259000+02:00
...
```

`fetchall`

Signature

```
fetchall(self: duckdb.duckdb.DuckDBPyRelation) -> list
```

Description Execute and fetch all rows as a list of tuples

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.limit(1).fetchall()
```

Result

```
[(UUID('1587b4b0-3023-49fe-82cf-06303ca136ac'),
  'value is uneven',
  1,
  datetime.datetime(2025, 4, 10, 11, 24, 51, 259000, tzinfo=<DstTzInfo 'Europe/Amsterdam' CEST+2:00:00
DST>))]
```

`fetchdf`

Signature

```
fetchdf(self: duckdb.duckdb.DuckDBPyRelation, *, date_as_object: bool = False) -> pandas.DataFrame
```

Description Execute and fetch all rows as a pandas DataFrame

Aliases: `df`, `to_df`

Parameters

- **date_as_object** : bool, default: False

  If the date columns should be interpreted as Python date objects.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.fetchdf()
```

Result

```
                                     id      description  value                created_timestamp
0  1587b4b0-3023-49fe-82cf-06303ca136ac  value is uneven      1 2025-04-10 11:24:51.259000+02:00
1  e4ab8cb4-4609-40cb-ad7e-4304ed5ed4bd    value is even      2 2025-04-10 11:25:51.259000+02:00
2  3f8ad67a-290f-4a22-b41b-0173b8e45afa  value is uneven      3 2025-04-10 11:26:51.259000+02:00
...
```

`fetchmany`

Signature

```
fetchmany(self: duckdb.duckdb.DuckDBPyRelation, size: int = 1) -> list
```

Description Execute and fetch the next set of rows as a list of tuples

> Warning Executing any operation during the retrieval of the data from an aggregate relation, will close the result set.
>
> ```
> import duckdb
>
> duckdb_conn = duckdb.connect()
>
> rel = duckdb_conn.sql("""
>       select
>           gen_random_uuid() as id,
>          concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
>           range as value,
>           now() + concat(range,' ', 'minutes')::interval as created_timestamp
>       from range(1, 10)
>    """
> )
>
> agg_rel = rel.aggregate("value")
>
> while res := agg_rel.fetchmany(size=1):
>    print(res)
>    rel.show()
> ```

Parameters

- **size** : int, default: 1

  The number of records to be fetched.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

while res := rel.fetchmany(size=1):
    print(res)
```

Result

```
[(UUID('cf4c5e32-d0aa-4699-a3ee-0092e900f263'), 'value is uneven', 1, datetime.datetime(2025, 4, 30, 16,
23, 5, 310000, tzinfo=<DstTzInfo 'Europe/Amsterdam' CEST+2:00:00 DST>))]
[(UUID('cec335ac-24ac-49a3-ae9a-bb35f71fc88d'), 'value is even', 2, datetime.datetime(2025, 4, 30, 16,
24, 5, 310000, tzinfo=<DstTzInfo 'Europe/Amsterdam' CEST+2:00:00 DST>))]
[(UUID('2423295d-9bb0-453c-a385-21bdacba03b6'), 'value is uneven', 3, datetime.datetime(2025, 4, 30, 16,
25, 5, 310000, tzinfo=<DstTzInfo 'Europe/Amsterdam' CEST+2:00:00 DST>))]
[(UUID('88806b21-192d-41e7-a293-c789aad636ba'), 'value is even', 4, datetime.datetime(2025, 4, 30, 16,
26, 5, 310000, tzinfo=<DstTzInfo 'Europe/Amsterdam' CEST+2:00:00 DST>))]
[(UUID('05837a28-dacf-4121-88a6-a374aefb8a07'), 'value is uneven', 5, datetime.datetime(2025, 4, 30, 16,
27, 5, 310000, tzinfo=<DstTzInfo 'Europe/Amsterdam' CEST+2:00:00 DST>))]
[(UUID('b9c1f7e9-6156-4554-b80e-67d3b5d810bb'), 'value is even', 6, datetime.datetime(2025, 4, 30, 16,
28, 5, 310000, tzinfo=<DstTzInfo 'Europe/Amsterdam' CEST+2:00:00 DST>))]
[(UUID('4709c7fa-d286-4864-bb48-69748b447157'), 'value is uneven', 7, datetime.datetime(2025, 4, 30, 16,
29, 5, 310000, tzinfo=<DstTzInfo 'Europe/Amsterdam' CEST+2:00:00 DST>))]
[(UUID('30e48457-b103-4fa5-95cf-1c7f0143335b'), 'value is even', 8, datetime.datetime(2025, 4, 30, 16,
30, 5, 310000, tzinfo=<DstTzInfo 'Europe/Amsterdam' CEST+2:00:00 DST>))]
[(UUID('036b7f4b-bd78-4ffb-a351-964d93f267b7'), 'value is uneven', 9, datetime.datetime(2025, 4, 30, 16,
31, 5, 310000, tzinfo=<DstTzInfo 'Europe/Amsterdam' CEST+2:00:00 DST>))]
```

`fetchnumpy`

Signature

```
fetchnumpy(self: duckdb.duckdb.DuckDBPyRelation) -> dict
```

Description Execute and fetch all rows as a Python dict mapping each column to one numpy arrays

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.fetchnumpy()
```

Result

```
{'id': array([UUID('1587b4b0-3023-49fe-82cf-06303ca136ac'),
        UUID('e4ab8cb4-4609-40cb-ad7e-4304ed5ed4bd'),
        UUID('3f8ad67a-290f-4a22-b41b-0173b8e45afa'),
        UUID('9a4e37ef-d8bd-46dd-ab01-51cf4973549f'),
        UUID('12baa624-ebc9-45ae-b73e-6f4029e31d2d'),
        UUID('56d41292-53cc-48be-a1b8-e1f5d6ca5581'),
        UUID('1accca18-c950-47c1-9108-aef8afbd5249'),
        UUID('56d8db75-72c4-4d40-90d2-a3c840579c37'),
        UUID('e19f6201-8646-401c-b019-e37c42c39632')], dtype=object),
 'description': array(['value is uneven', 'value is even', 'value is uneven',
        'value is even', 'value is uneven', 'value is even',
        'value is uneven', 'value is even', 'value is uneven'],
       dtype=object),
 'value': array([1, 2, 3, 4, 5, 6, 7, 8, 9]),
 'created_timestamp': array(['2025-04-10T09:24:51.259000', '2025-04-10T09:25:51.259000',
        '2025-04-10T09:26:51.259000', '2025-04-10T09:27:51.259000',
        '2025-04-10T09:28:51.259000', '2025-04-10T09:29:51.259000',
        '2025-04-10T09:30:51.259000', '2025-04-10T09:31:51.259000',
        '2025-04-10T09:32:51.259000'], dtype='datetime64[us]')}
```

`fetchone`

Signature

```
fetchone(self: duckdb.duckdb.DuckDBPyRelation) -> typing.Optional[tuple]
```

Description Execute and fetch a single row as a tuple

> Warning Executing any operation during the retrieval of the data from an aggregate relation, will close the result set.
>
> ```
> import duckdb
>
> duckdb_conn = duckdb.connect()
>
> rel = duckdb_conn.sql("""
>       select
> ```

```
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

agg_rel = rel.aggregate("value")

while res := agg_rel.fetchone():
    print(res)
    rel.show()
```

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

while res := rel.fetchone():
    print(res)
```

Result

```
(UUID('fe036411-f4c7-4f52-9ddd-80cd2bb56613'), 'value is uneven', 1, datetime.datetime(2025, 4, 30, 12,
59, 8, 912000, tzinfo=<DstTzInfo 'Europe/Amsterdam' CEST+2:00:00 DST>))
(UUID('466c9b43-e9f0-4237-8f26-155f259a5b59'), 'value is even', 2, datetime.datetime(2025, 4, 30, 13, 0,
8, 912000, tzinfo=<DstTzInfo 'Europe/Amsterdam' CEST+2:00:00 DST>))
(UUID('5755cf16-a94f-41ef-a16d-21e856d71f9f'), 'value is uneven', 3, datetime.datetime(2025, 4, 30, 13,
1, 8, 912000, tzinfo=<DstTzInfo 'Europe/Amsterdam' CEST+2:00:00 DST>))
(UUID('05b52c93-bd68-45e1-b02a-a08d682c33d5'), 'value is even', 4, datetime.datetime(2025, 4, 30, 13, 2,
8, 912000, tzinfo=<DstTzInfo 'Europe/Amsterdam' CEST+2:00:00 DST>))
(UUID('cf61ef13-2840-4541-900d-f493767d7622'), 'value is uneven', 5, datetime.datetime(2025, 4, 30, 13,
3, 8, 912000, tzinfo=<DstTzInfo 'Europe/Amsterdam' CEST+2:00:00 DST>))
(UUID('033e7c68-e800-4ee8-9787-6cf50aabc27b'), 'value is even', 6, datetime.datetime(2025, 4, 30, 13, 4,
8, 912000, tzinfo=<DstTzInfo 'Europe/Amsterdam' CEST+2:00:00 DST>))
(UUID('8b8d6545-ff54-45d6-b69a-97edb63dfe43'), 'value is uneven', 7, datetime.datetime(2025, 4, 30, 13,
5, 8, 912000, tzinfo=<DstTzInfo 'Europe/Amsterdam' CEST+2:00:00 DST>))
(UUID('7da79dfe-b29c-462b-a414-9d5e3cc80139'), 'value is even', 8, datetime.datetime(2025, 4, 30, 13, 6,
8, 912000, tzinfo=<DstTzInfo 'Europe/Amsterdam' CEST+2:00:00 DST>))
(UUID('f83ffff2-33b9-4f86-9d14-46974b546bab'), 'value is uneven', 9, datetime.datetime(2025, 4, 30, 13,
7, 8, 912000, tzinfo=<DstTzInfo 'Europe/Amsterdam' CEST+2:00:00 DST>))
```

---

pl

Signature

```
pl(self: duckdb.duckdb.DuckDBPyRelation, batch_size: int = 1000000) -> duckdb::PolarsDataFrame
```

Description Execute and fetch all rows as a Polars DataFrame

Parameters

- **batch_size** : int, default: 1000000

  The number of records to be fetched per batch.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.pl(batch_size=1)
```

Result

```
shape: (9, 4)
```

id --- str	description --- str	value --- i64	created_timestamp --- datetime[μs, Europe/Amsterdam]
b2f92c3c-9372-49f3-897f-2c86fc…	value is uneven	1	2025-04-10 11:49:51.886 CEST

`record_batch`

Signature

```
record_batch(self: duckdb.duckdb.DuckDBPyRelation, batch_size: int = 1000000) ->
pyarrow.lib.RecordBatchReader
```

Description Execute and return an Arrow Record Batch Reader that yields all rows

Parameters

- **batch_size** : int, default: 1000000

  The batch size for fetching the data.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

pa_batch = rel.record_batch(batch_size=1)

pa_batch.read_next_batch()
```

Result

```
pyarrow.RecordBatch
id: string
description: string
value: int64
created_timestamp: timestamp[us, tz=Europe/Amsterdam]
----
id: ["908cf67c-a086-4b94-9017-2089a83e4a6c"]
description: ["value is uneven"]
value: [1]
created_timestamp: [2025-04-10 09:52:55.249000Z]
```

---

tf

Signature

```
tf(self: duckdb.duckdb.DuckDBPyRelation) -> dict
```

Description Fetch a result as dict of TensorFlow Tensors

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)
```

```
rel.select("description, value").tf()
```

Result

```
{'description': <tf.Tensor: shape=(9,), dtype=string, numpy=
 array([b'value is uneven', b'value is even', b'value is uneven',
        b'value is even', b'value is uneven', b'value is even',
        b'value is uneven', b'value is even', b'value is uneven'],
       dtype=object)>,
 'value': <tf.Tensor: shape=(9,), dtype=int64, numpy=array([1, 2, 3, 4, 5, 6, 7, 8, 9])>}
```

---

to_arrow_table

Signature

```
to_arrow_table(self: duckdb.duckdb.DuckDBPyRelation, batch_size: int = 1000000) -> pyarrow.lib.Table
```

Description Execute and fetch all rows as an Arrow Table

Aliases: `fetch_arrow_table`, `arrow`

Parameters

- **batch_size** : int, default: 1000000

  The batch size for fetching the data.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.to_arrow_table()
```

Result

```
pyarrow.Table
id: string
description: string
value: int64
created_timestamp: timestamp[us, tz=Europe/Amsterdam]
----
```

```
id: [["86b2011d-3818-426f-a41e-7cd5c7321f79","07fa4f89-0bba-4049-9acd-c933332a66d5","f2f1479e-f582-
4fe4-b82f-9b753b69634c","529d3c63-5961-4adb-b0a8-8249188fc82a","aa9eea7d-7fac-4dcf-8f32-
4a0b5d64f864","4852aa32-03f2-40d3-8006-b8213904775a","c0127203-f2e3-4925-9810-655bc02a3c19","2a1356ba-
5707-44d6-a492-abd0a67e5efb","800a1c24-231c-4dae-bd68-627654c8a110"]]
description: [["value is uneven","value is even","value is uneven","value is even","value is
uneven","value is even","value is uneven","value is even","value is uneven"]]
value: [[1,2,3,4,5,6,7,8,9]]
created_timestamp: [[2025-04-10 09:54:24.015000Z,2025-04-10 09:55:24.015000Z,2025-04-10
09:56:24.015000Z,2025-04-10 09:57:24.015000Z,2025-04-10 09:58:24.015000Z,2025-04-10
09:59:24.015000Z,2025-04-10 10:00:24.015000Z,2025-04-10 10:01:24.015000Z,2025-04-10 10:02:24.015000Z]]
```

`to_csv`

Signature

```
to_csv(self: duckdb.duckdb.DuckDBPyRelation, file_name: str, *, sep: object = None, na_rep: object =
None, header: object = None, quotechar: object = None, escapechar: object = None, date_format: object =
None, timestamp_format: object = None, quoting: object = None, encoding: object = None, compression:
object = None, overwrite: object = None, per_thread_output: object = None, use_tmp_file: object = None,
partition_by: object = None, write_partition_columns: object = None) -> None
```

Description Write the relation object to a CSV file in 'file_name'

Aliases: `write_csv`

Parameters

- **file_name** : str

  The name of the output CSV file.

- **sep** : str, default: ','

  Field delimiter for the output file.

- **na_rep** : str, default: ''

  Missing data representation.

- **header** : bool, default: True

  Whether to write column headers.

- **quotechar** : str, default: '"'

  Character used to quote fields containing special characters.

- **escapechar** : str, default: None

  Character used to escape the delimiter if quoting is set to QUOTE_NONE.

- **date_format** : str, default: None

  Custom format string for DATE values.

- **timestamp_format** : str, default: None

  Custom format string for TIMESTAMP values.

- **quoting** : int, default: csv.QUOTE_MINIMAL

  Control field quoting behavior (e.g., QUOTE_MINIMAL, QUOTE_ALL).

- **encoding** : str, default: 'utf-8'

  Character encoding for the output file.

- **compression** : str, default: auto

  Compression type (e.g., 'gzip', 'bz2', 'zstd').

- **overwrite** : bool, default: False

  When true, all existing files inside targeted directories will be removed (not supported on remote filesystems). Only has an effect when used with `partition_by`.

- **per_thread_output** : bool, default: False

  When `true`, write one file per thread, rather than one file in total. This allows for faster parallel writing.

- **use_tmp_file** : bool, default: False

  Write to a temporary file before renaming to final name to avoid partial writes.

- **partition_by** : list[str], default: None

  List of column names to partition output by (creates folder structure).

- **write_partition_columns** : bool, default: False

  Whether or not to write partition columns into files. Only has an effect when used with `partition_by`.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.to_csv("code_example.csv")
```

Result

```
The data is exported to a CSV file, named code_example.csv
```

---

to_df

Signature

```
to_df(self: duckdb.duckdb.DuckDBPyRelation, *, date_as_object: bool = False) -> pandas.DataFrame
```

Description Execute and fetch all rows as a pandas DataFrame

Aliases: `fetchdf`, `df`

Parameters

- **date_as_object** : bool, default: False

  If the date columns should be interpreted as Python date objects.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.to_df()
```

Result

```
                                     id      description  value                created_timestamp
0  e1f79925-60fd-4ee2-ae67-5eff6b0543d1  value is uneven      1 2025-04-10 11:56:04.452000+02:00
1  caa619d4-d79c-4c00-b82e-9319b086b6f8    value is even      2 2025-04-10 11:57:04.452000+02:00
2  64c68032-99b9-4e8f-b4a3-6c522d5419b3  value is uneven      3 2025-04-10 11:58:04.452000+02:00
...
```

---

to_parquet

Signature

```
to_parquet(self: duckdb.duckdb.DuckDBPyRelation, file_name: str, *, compression: object = None, field_
ids: object = None, row_group_size_bytes: object = None, row_group_size: object = None, overwrite:
object = None, per_thread_output: object = None, use_tmp_file: object = None, partition_by: object =
None, write_partition_columns: object = None, append: object = None) -> None
```

Description Write the relation object to a Parquet file in 'file_name'

Aliases: `write_parquet`

Parameters

- **file_name** : str

  The name of the output Parquet file.

- **compression** : str, default: 'snappy'

  The compression format to use (`uncompressed`, `snappy`, `gzip`, `zstd`, `brotli`, `lz4`, `lz4_raw`).

- **field_ids** : STRUCT

  The field_id for each column. Pass auto to attempt to infer automatically.

- **row_group_size_bytes** : int, default: row_group_size * 1024

  The target size of each row group. You can pass either a human-readable string, e.g., 2MB, or an integer, i.e., the number of bytes. This option is only used when you have issued `SET preserve_insertion_order = false;`, otherwise, it is ignored.

- **row_group_size** : int, default: 122880

  The target size, i.e., number of rows, of each row group.

- **overwrite** : bool, default: False

  If True, overwrite the file if it exists.

- **per_thread_output** : bool, default: False

  When `True`, write one file per thread, rather than one file in total. This allows for faster parallel writing.

- **use_tmp_file** : bool, default: False

  Write to a temporary file before renaming to final name to avoid partial writes.

- **partition_by** : list[str], default: None

  List of column names to partition output by (creates folder structure).

- **write_partition_columns** : bool, default: False

  Whether or not to write partition columns into files. Only has an effect when used with `partition_by`.

- **append** : bool, default: False

  When `True`, in the event a filename pattern is generated that already exists, the path will be regenerated to ensure no existing files are overwritten. Only has an effect when used with `partition_by`.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.to_parquet("code_example.parquet")
```

Result

```
The data is exported to a Parquet file, named code_example.parquet
```

---

to_table

Signature

```
to_table(self: duckdb.duckdb.DuckDBPyRelation, table_name: str) -> None
```

Description Creates a new table named table_name with the contents of the relation object

Aliases: `create`

Parameters

- **table_name** : str

  The name of the table to be created. There shouldn't be any other table with the same name.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.to_table("table_code_example")
```

Result

```
A table, named table_code_example, is created with the data of the relation
```

`to_view`

Signature

```
to_view(self: duckdb.duckdb.DuckDBPyRelation, view_name: str, replace: bool = True) ->
duckdb.duckdb.DuckDBPyRelation
```

Description Creates a view named view_name that refers to the relation object

Aliases: `create_view`

Parameters

- **view_name** : str

  The name of the view to be created.

- **replace** : bool, default: True

  If the view should be created with `CREATE OR REPLACE`. When set to `False`, there shouldn't be another view with the same `view_name`.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.to_view("view_code_example", replace=True)
```

Result

```
A view, named view_code_example, is created with the query definition of the relation
```

`torch`

Signature

```
torch(self: duckdb.duckdb.DuckDBPyRelation) -> dict
```

Description Fetch a result as dict of PyTorch Tensors

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.select("value").torch()
```

Result

```
{'value': tensor([1, 2, 3, 4, 5, 6, 7, 8, 9])}
```

`write_csv`

Signature

```
write_csv(self: duckdb.duckdb.DuckDBPyRelation, file_name: str, *, sep: object = None, na_rep: object = None, header: object = None, quotechar: object = None, escapechar: object = None, date_format: object = None, timestamp_format: object = None, quoting: object = None, encoding: object = None, compression: object = None, overwrite: object = None, per_thread_output: object = None, use_tmp_file: object = None, partition_by: object = None, write_partition_columns: object = None) -> None
```

Description Write the relation object to a CSV file in 'file_name'

Aliases: `to_csv`

Parameters

- **file_name** : str

  The name of the output CSV file.

- **sep** : str, default: ','

  Field delimiter for the output file.

- **na_rep** : str, default: ''

  Missing data representation.

- **header** : bool, default: True

  Whether to write column headers.

- **quotechar** : str, default: '"'

  Character used to quote fields containing special characters.

- **escapechar** : str, default: None

  Character used to escape the delimiter if quoting is set to QUOTE_NONE.

- **date_format** : str, default: None

  Custom format string for DATE values.

- **timestamp_format** : str, default: None

  Custom format string for TIMESTAMP values.

- **quoting** : int, default: csv.QUOTE_MINIMAL

  Control field quoting behavior (e.g., QUOTE_MINIMAL, QUOTE_ALL).

- **encoding** : str, default: 'utf-8'

  Character encoding for the output file.

- **compression** : str, default: auto

  Compression type (e.g., 'gzip', 'bz2', 'zstd').

- **overwrite** : bool, default: False

  When true, all existing files inside targeted directories will be removed (not supported on remote filesystems). Only has an effect when used with `partition_by`.

- **per_thread_output** : bool, default: False

  When `true`, write one file per thread, rather than one file in total. This allows for faster parallel writing.

- **use_tmp_file** : bool, default: False

  Write to a temporary file before renaming to final name to avoid partial writes.

- **partition_by** : list[str], default: None

  List of column names to partition output by (creates folder structure).

- **write_partition_columns** : bool, default: False

  Whether or not to write partition columns into files. Only has an effect when used with `partition_by`.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.write_csv("code_example.csv")
```

Result

```
The data is exported to a CSV file, named code_example.csv
```

`write_parquet`

Signature

```
write_parquet(self: duckdb.duckdb.DuckDBPyRelation, file_name: str, *, compression: object = None,
field_ids: object = None, row_group_size_bytes: object = None, row_group_size: object = None, overwrite:
object = None, per_thread_output: object = None, use_tmp_file: object = None, partition_by: object =
None, write_partition_columns: object = None, append: object = None) -> None
```

Description Write the relation object to a Parquet file in 'file_name'

Aliases: `to_parquet`

Parameters

- **file_name** : str

  The name of the output Parquet file.

- **compression** : str, default: 'snappy'

  The compression format to use (`uncompressed`, `snappy`, `gzip`, `zstd`, `brotli`, `lz4`, `lz4_raw`).

- **field_ids** : STRUCT

  The field_id for each column. Pass auto to attempt to infer automatically.

- **row_group_size_bytes** : int, default: row_group_size * 1024

  The target size of each row group. You can pass either a human-readable string, e.g., 2MB, or an integer, i.e., the number of bytes. This option is only used when you have issued `SET preserve_insertion_order = false;`, otherwise, it is ignored.

- **row_group_size** : int, default: 122880

  The target size, i.e., number of rows, of each row group.

- **overwrite** : bool, default: False

  If True, overwrite the file if it exists.

- **per_thread_output** : bool, default: False

  When `True`, write one file per thread, rather than one file in total. This allows for faster parallel writing.

- **use_tmp_file** : bool, default: False

  Write to a temporary file before renaming to final name to avoid partial writes.

- **partition_by** : list[str], default: None

  List of column names to partition output by (creates folder structure).

- **write_partition_columns** : bool, default: False

  Whether or not to write partition columns into files. Only has an effect when used with `partition_by`.

- **append** : bool, default: False

  When `True`, in the event a filename pattern is generated that already exists, the path will be regenerated to ensure no existing files are overwritten. Only has an effect when used with `partition_by`.

Example

```
import duckdb

duckdb_conn = duckdb.connect()

rel = duckdb_conn.sql("""
        select
            gen_random_uuid() as id,
            concat('value is ', case when mod(range,2)=0 then 'even' else 'uneven' end) as description,
            range as value,
            now() + concat(range,' ', 'minutes')::interval as created_timestamp
        from range(1, 10)
    """
)

rel.write_parquet("code_example.parquet")
```

Result

```
The data is exported to a Parquet file, named code_example.parquet
```

Python Function API

You can create a DuckDB user-defined function (UDF) from a Python function so it can be used in SQL queries. Similarly to regular functions, they need to have a name, a return type and parameter types.

Here is an example using a Python function that calls a third-party library.

```
import duckdb
from duckdb.typing import *
from faker import Faker

def generate_random_name():
    fake = Faker()
    return fake.name()

duckdb.create_function("random_name", generate_random_name, [], VARCHAR)
res = duckdb.sql("SELECT random_name()").fetchall()
print(res)
```

```
[('Gerald Ashley',)]
```

Creating Functions

To register a Python UDF, use the `create_function` method from a DuckDB connection. Here is the syntax:

```
import duckdb
con = duckdb.connect()
con.create_function(name, function, parameters, return_type)
```

The `create_function` method takes the following parameters:

1. `name` A string representing the unique name of the UDF within the connection catalog.
2. `function` The Python function you wish to register as a UDF.
3. `parameters` Scalar functions can operate on one or more columns. This parameter takes a list of column types used as input.
4. `return_type` Scalar functions return one element per row. This parameter specifies the return type of the function.
5. `type` (optional): DuckDB supports both built-in Python types and PyArrow Tables. By default, built-in types are assumed, but you can specify `type = 'arrow'` to use PyArrow Tables.
6. `null_handling` (optional): By default, `NULL` values are automatically handled as `NULL`-in `NULL`-out. Users can specify a desired behavior for `NULL` values by setting `null_handling = 'special'`.
7. `exception_handling` (optional): By default, when an exception is thrown from the Python function, it will be re-thrown in Python. Users can disable this behavior, and instead return `NULL`, by setting this parameter to `'return_null'`
8. `side_effects` (optional): By default, functions are expected to produce the same result for the same input. If the result of a function is impacted by any type of randomness, `side_effects` must be set to `True`.

To unregister a UDF, you can call the `remove_function` method with the UDF name:

```
con.remove_function(name)
```

Using Partial Functions

DuckDB UDFs can also be created with Python partial functions.

In the below example, we show how a custom logger will return the concatenation of the execution datetime in ISO format, always followed by argument passed at UDF creation and the input parameter provided to the function call:

```
from datetime import datetime
import duckdb
import functools


def get_datetime_iso_format() -> str:
    return datetime.now().isoformat()
```

```python
def logger_udf(func, arg1: str, arg2: int) -> str:
    return ' '.join([func(), arg1, str(arg2)])


with duckdb.connect() as con:
    con.sql("select * from range(10) tbl(id)").to_table("example_table")

    con.create_function(
        'custom_logger',
        functools.partial(logger_udf, get_datetime_iso_format, 'logging data')
    )
    rel = con.sql("SELECT custom_logger(id) from example_table;")
    rel.show()

    con.create_function(
        'another_custom_logger',
        functools.partial(logger_udf, get_datetime_iso_format, ':')
    )
    rel = con.sql("SELECT another_custom_logger(id) from example_table;")
    rel.show()
```

```
┌──────────────────────────────────────────┐
│            custom_logger(id)            │
│                  varchar                 │
├──────────────────────────────────────────┤
│ 2025-03-27T12:07:56.811251 logging data 0 │
│ 2025-03-27T12:07:56.811264 logging data 1 │
│ 2025-03-27T12:07:56.811266 logging data 2 │
│ 2025-03-27T12:07:56.811268 logging data 3 │
│ 2025-03-27T12:07:56.811269 logging data 4 │
│ 2025-03-27T12:07:56.811270 logging data 5 │
│ 2025-03-27T12:07:56.811271 logging data 6 │
│ 2025-03-27T12:07:56.811272 logging data 7 │
│ 2025-03-27T12:07:56.811274 logging data 8 │
│ 2025-03-27T12:07:56.811275 logging data 9 │
├──────────────────────────────────────────┤
│                 10 rows                  │
└──────────────────────────────────────────┘
```

```
┌──────────────────────────────────┐
│   another_custom_logger(id)      │
│             varchar              │
├──────────────────────────────────┤
│ 2025-03-27T12:07:56.812106 : 0   │
│ 2025-03-27T12:07:56.812116 : 1   │
│ 2025-03-27T12:07:56.812118 : 2   │
│ 2025-03-27T12:07:56.812119 : 3   │
│ 2025-03-27T12:07:56.812121 : 4   │
│ 2025-03-27T12:07:56.812122 : 5   │
│ 2025-03-27T12:07:56.812123 : 6   │
│ 2025-03-27T12:07:56.812124 : 7   │
│ 2025-03-27T12:07:56.812126 : 8   │
│ 2025-03-27T12:07:56.812127 : 9   │
├──────────────────────────────────┤
│            10 rows               │
└──────────────────────────────────┘
```

Type Annotation

When the function has type annotation it's often possible to leave out all of the optional parameters. Using `DuckDBPyType` we can implicitly convert many known types to DuckDBs type system. For example:

```
import duckdb

def my_function(x: int) -> str:
    return x

duckdb.create_function("my_func", my_function)
print(duckdb.sql("SELECT my_func(42)"))
```

```
┌──────────────┐
│ my_func(42)  │
│   varchar    │
├──────────────┤
│ 42           │
└──────────────┘
```

If only the parameter list types can be inferred, you'll need to pass in None as `parameters`.

NULL Handling

By default when functions receive a `NULL` value, this instantly returns `NULL`, as part of the default `NULL`-handling. When this is not desired, you need to explicitly set this parameter to `"special"`.

```
import duckdb
from duckdb.typing import *

def dont_intercept_null(x):
    return 5

duckdb.create_function("dont_intercept", dont_intercept_null, [BIGINT], BIGINT)
res = duckdb.sql("SELECT dont_intercept(NULL)").fetchall()
print(res)
```

```
[(None,)]
```

With `null_handling="special"`:

```
import duckdb
from duckdb.typing import *

def dont_intercept_null(x):
    return 5

duckdb.create_function("dont_intercept", dont_intercept_null, [BIGINT], BIGINT, null_handling="special")
res = duckdb.sql("SELECT dont_intercept(NULL)").fetchall()
print(res)
```

```
[(5,)]
```

> Always use `null_handling="special"` when the function can return NULL.

```
import duckdb
from duckdb.typing import VARCHAR
```

```python
def return_str_or_none(x: str) -> str | None:
    if not x:
        return None

    return x

duckdb.create_function(
    "return_str_or_none",
    return_str_or_none,
    [VARCHAR],
    VARCHAR,
    null_handling="special"
)
res = duckdb.sql("SELECT return_str_or_none('')").fetchall()
print(res)
```

```
[(None,)]
```

Exception Handling

By default, when an exception is thrown from the Python function, we'll forward (re-throw) the exception. If you want to disable this behavior, and instead return `NULL`, you'll need to set this parameter to `"return_null"`.

```python
import duckdb
from duckdb.typing import *

def will_throw():
    raise ValueError("ERROR")

duckdb.create_function("throws", will_throw, [], BIGINT)
try:
    res = duckdb.sql("SELECT throws()").fetchall()
except duckdb.InvalidInputException as e:
    print(e)

duckdb.create_function("doesnt_throw", will_throw, [], BIGINT, exception_handling="return_null")
res = duckdb.sql("SELECT doesnt_throw()").fetchall()
print(res)
```

```
Invalid Input Error: Python exception occurred while executing the UDF: ValueError: ERROR

At:
  ...(5): will_throw
  ...(9): <module>

[(None,)]
```

Side Effects

By default DuckDB will assume the created function is a *pure* function, meaning it will produce the same output when given the same input. If your function does not follow that rule, for example when your function makes use of randomness, then you will need to mark this function as having `side_effects`.

For example, this function will produce a new count for every invocation.

```python
def count() -> int:
    old = count.counter;
```

```
    count.counter += 1
    return old

count.counter = 0
```

If we create this function without marking it as having side effects, the result will be the following:

```
con = duckdb.connect()
con.create_function("my_counter", count, side_effects=False)
res = con.sql("SELECT my_counter() FROM range(10)").fetchall()
print(res)
```

```
[(0,), (0,), (0,), (0,), (0,), (0,), (0,), (0,), (0,), (0,)]
```

Which is obviously not the desired result, when we add `side_effects=True`, the result is as we would expect:

```
con.remove_function("my_counter")
count.counter = 0
con.create_function("my_counter", count, side_effects=True)
res = con.sql("SELECT my_counter() FROM range(10)").fetchall()
print(res)
```

```
[(0,), (1,), (2,), (3,), (4,), (5,), (6,), (7,), (8,), (9,)]
```

Python Function Types

Currently, two function types are supported, `native` (default) and `arrow`.

Arrow

If the function is expected to receive arrow arrays, set the `type` parameter to `'arrow'`.

This will let the system know to provide arrow arrays of up to `STANDARD_VECTOR_SIZE` tuples to the function, and also expect an array of the same amount of tuples to be returned from the function.

Native

When the function type is set to `native` the function will be provided with a single tuple at a time, and expect only a single value to be returned. This can be useful to interact with Python libraries that don't operate on Arrow, such as `faker`:

```
import duckdb

from duckdb.typing import *
from faker import Faker

def random_date():
    fake = Faker()
    return fake.date_between()

duckdb.create_function("random_date", random_date, [], DATE, type="native")
res = duckdb.sql("SELECT random_date()").fetchall()
print(res)
```

```
[(datetime.date(2019, 5, 15),)]
```

Types API

The `DuckDBPyType` class represents a type instance of our data types.

Converting from Other Types

To make the API as easy to use as possible, we have added implicit conversions from existing type objects to a DuckDBPyType instance. This means that wherever a DuckDBPyType object is expected, it is also possible to provide any of the options listed below.

Python Built-Ins

The table below shows the mapping of Python Built-in types to DuckDB type.

Built-in types	DuckDB type
bool	BOOLEAN
bytearray	BLOB
bytes	BLOB
float	DOUBLE
int	BIGINT
str	VARCHAR

Numpy DTypes

The table below shows the mapping of Numpy DType to DuckDB type.

Type	DuckDB type
bool	BOOLEAN
float32	FLOAT
float64	DOUBLE
int16	SMALLINT
int32	INTEGER
int64	BIGINT
int8	TINYINT
uint16	USMALLINT
uint32	UINTEGER
uint64	UBIGINT
uint8	UTINYINT

Nested Types

`list[child_type]`

`list` type objects map to a `LIST` type of the child type. Which can also be arbitrarily nested.

```
import duckdb
from typing import Union

duckdb.typing.DuckDBPyType(list[dict[Union[str, int], str]])
```

```
MAP(UNION(u1 VARCHAR, u2 BIGINT), VARCHAR)[]
```

`dict[key_type, value_type]`

`dict` type objects map to a `MAP` type of the key type and the value type.

```
import duckdb

print(duckdb.typing.DuckDBPyType(dict[str, int]))

MAP(VARCHAR, BIGINT)
```

`{'a': field_one, 'b': field_two, .., 'n': field_n}`

`dict` objects map to a `STRUCT` composed of the keys and values of the dict.

```
import duckdb

print(duckdb.typing.DuckDBPyType({'a': str, 'b': int}))

STRUCT(a VARCHAR, b BIGINT)
```

`Union[type_1, ... type_n]`

`typing.Union` objects map to a `UNION` type of the provided types.

```
import duckdb
from typing import Union

print(duckdb.typing.DuckDBPyType(Union[int, str, bool, bytearray]))

UNION(u1 BIGINT, u2 VARCHAR, u3 BOOLEAN, u4 BLOB)
```

Creation Functions

For the built-in types, you can use the constants defined in `duckdb.typing`:

DuckDB type
BIGINT
BIT
BLOB
BOOLEAN
DATE
DOUBLE
FLOAT
HUGEINT
INTEGER
INTERVAL
SMALLINT
SQLNULL
TIME_TZ
TIME

DuckDB type
TIMESTAMP_MS
TIMESTAMP_NS
TIMESTAMP_S
TIMESTAMP_TZ
TIMESTAMP
TINYINT
UBIGINT
UHUGEINT
UINTEGER
USMALLINT
UTINYINT
UUID
VARCHAR

For the complex types there are methods available on the `DuckDBPyConnection` object or the `duckdb` module. Anywhere a `DuckDBPyType` is accepted, we will also accept one of the type objects that can implicitly convert to a `DuckDBPyType`.

`list_type|array_type`

Parameters:

- `child_type: DuckDBPyType`

`struct_type|row_type`

Parameters:

- `fields: Union[list[DuckDBPyType], dict[str, DuckDBPyType]]`

`map_type`

Parameters:

- `key_type: DuckDBPyType`
- `value_type: DuckDBPyType`

`decimal_type`

Parameters:

- `width: int`
- `scale: int`

`union_type`

Parameters:

- `members: Union[list[DuckDBPyType], dict[str, DuckDBPyType]]`

string_type

Parameters:

- collation: Optional[str]

Expression API

The Expression class represents an instance of an expression.

Why Would I Use the Expression API?

Using this API makes it possible to dynamically build up expressions, which are typically created by the parser from the query string. This allows you to skip that and have more fine-grained control over the used expressions.

Below is a list of currently supported expressions that can be created through the API.

Column Expression

This expression references a column by name.

```
import duckdb
import pandas as pd

df = pd.DataFrame({
    'a': [1, 2, 3, 4],
    'b': [True, None, False, True],
    'c': [42, 21, 13, 14]
})
```

Selecting a single column:

```
col = duckdb.ColumnExpression('a')
duckdb.df(df).select(col).show()
```

```
┌─────────┐
│    a    │
│  int64  │
├─────────┤
│       1 │
│       2 │
│       3 │
│       4 │
└─────────┘
```

Selecting multiple columns:

```
col_list = [
        duckdb.ColumnExpression('a') * 10,
        duckdb.ColumnExpression('b').isnull(),
        duckdb.ColumnExpression('c') + 5
    ]
duckdb.df(df).select(*col_list).show()
```

(a * 10) int64	(b IS NULL) boolean	(c + 5) int64
10	false	47
20	true	26
30	false	18
40	false	19

Star Expression

This expression selects all columns of the input source.

Optionally it's possible to provide an `exclude` list to filter out columns of the table. This `exclude` list can contain either strings or Expressions.

```
import duckdb
import pandas as pd

df = pd.DataFrame({
    'a': [1, 2, 3, 4],
    'b': [True, None, False, True],
    'c': [42, 21, 13, 14]
})

star = duckdb.StarExpression(exclude = ['b'])
duckdb.df(df).select(star).show()
```

a int64	c int64
1	42
2	21
3	13
4	14

Constant Expression

This expression contains a single value.

```
import duckdb
import pandas as pd

df = pd.DataFrame({
    'a': [1, 2, 3, 4],
    'b': [True, None, False, True],
    'c': [42, 21, 13, 14]
})

const = duckdb.ConstantExpression('hello')
duckdb.df(df).select(const).show()
```

'hello'

```
┌─────────┐
│ varchar │
├─────────┤
│ hello   │
│ hello   │
│ hello   │
│ hello   │
└─────────┘
```

Case Expression

This expression contains a `CASE WHEN (...) THEN (...) ELSE (...) END` expression. By default `ELSE` is `NULL` and it can be set using `.else(value = ...)`. Additional `WHEN (...) THEN (...)` blocks can be added with `.when(condition = ..., value = ...)`.

```
import duckdb
import pandas as pd
from duckdb import (
    ConstantExpression,
    ColumnExpression,
    CaseExpression
)

df = pd.DataFrame({
    'a': [1, 2, 3, 4],
    'b': [True, None, False, True],
    'c': [42, 21, 13, 14]
})

hello = ConstantExpression('hello')
world = ConstantExpression('world')

case = \
    CaseExpression(condition = ColumnExpression('b') == False, value = world) \
    .otherwise(hello)
duckdb.df(df).select(case).show()
```

```
┌──────────────────────────────────────────────────────────┐
│ CASE  WHEN ((b = false)) THEN ('world') ELSE 'hello' END │
│                          varchar                         │
├──────────────────────────────────────────────────────────┤
│ hello                                                    │
│ hello                                                    │
│ world                                                    │
│ hello                                                    │
└──────────────────────────────────────────────────────────┘
```

Function Expression

This expression contains a function call. It can be constructed by providing the function name and an arbitrary amount of Expressions as arguments.

```
import duckdb
import pandas as pd
from duckdb import (
    ConstantExpression,
    ColumnExpression,
```

```
    FunctionExpression
)

df = pd.DataFrame({
    'a': [1, 2, 3, 4],
    'b': [True, None, False, True],
    'c': [42, 21, 13, 14]
})

multiply_by_2 = FunctionExpression('multiply', ColumnExpression('a'), ConstantExpression(2))
duckdb.df(df).select(multiply_by_2).show()
```

```
┌──────────────┐
│ multiply(a, 2) │
│     int64      │
├──────────────┤
│            2 │
│            4 │
│            6 │
│            8 │
└──────────────┘
```

SQL Expression

This expression contains any valid SQL expression.

```
import duckdb
import pandas as pd

from duckdb import SQLExpression

df = pd.DataFrame({
    'a': [1, 2, 3, 4],
    'b': [True, None, False, True],
    'c': [42, 21, 13, 14]
})

duckdb.df(df).filter(
    SQLExpression("b is true")
).select(
    SQLExpression("a").alias("selecting_column_a"),
    SQLExpression("case when a = 1 then 1 else 0 end").alias("selecting_case_expression"),
    SQLExpression("1").alias("constant_numeric_column"),
    SQLExpression("'hello'").alias("constant_text_column")
).aggregate(
    aggr_expr=[
        SQLExpression("SUM(selecting_column_a)").alias("sum_a"),
        "selecting_case_expression" ,
        "constant_numeric_column",
        "constant_text_column"
    ],
).show()
```

sum_a	selecting_case_expression	constant_numeric_column	constant_text_column
int128	int32	int32	varchar
4	0	1	hello
1	1	1	hello

└─────────┴──────────────────────────┴──────────────────────┴──────────────────────┘

Common Operations

The Expression class also contains many operations that can be applied to any Expression type.

Operation	Description
`.alias(name: str)`	Applies an alias to the expression
`.cast(type: DuckDBPyType)`	Applies a cast to the provided type on the expression
`.isin(*exprs: Expression)`	Creates an `IN` expression against the provided expressions as the list
`.isnotin(*exprs: Expression)`	Creates a `NOT IN` expression against the provided expressions as the list
`.isnotnull()`	Checks whether the expression is not `NULL`
`.isnull()`	Checks whether the expression is `NULL`

Order Operations

When expressions are provided to `DuckDBPyRelation.order()`, the following order operations can be applied.

Operation	Description
`.asc()`	Indicates that this expression should be sorted in ascending order
`.desc()`	Indicates that this expression should be sorted in descending order
`.nulls_first()`	Indicates that the nulls in this expression should precede the non-null values
`.nulls_last()`	Indicates that the nulls in this expression should come after the non-null values

Spark API

The DuckDB Spark API implements the PySpark API, allowing you to use the familiar Spark API to interact with DuckDB. All statements are translated to DuckDB's internal plans using our relational API and executed using DuckDB's query engine.

> **Warning.** The DuckDB Spark API is currently experimental and features are still missing. We are very interested in feedback. Please report any functionality that you are missing, either through Discord or on GitHub.

Example

```
from duckdb.experimental.spark.sql import SparkSession as session
from duckdb.experimental.spark.sql.functions import lit, col
import pandas as pd

spark = session.builder.getOrCreate()

pandas_df = pd.DataFrame({
```

```
    'age': [34, 45, 23, 56],
    'name': ['Joan', 'Peter', 'John', 'Bob']
})

df = spark.createDataFrame(pandas_df)
df = df.withColumn(
    'location', lit('Seattle')
)
res = df.select(
    col('age'),
    col('location')
).collect()

print(res)
```

```
[
    Row(age=34, location='Seattle'),
    Row(age=45, location='Seattle'),
    Row(age=23, location='Seattle'),
    Row(age=56, location='Seattle')
]
```

Contribution Guidelines

Contributions to the experimental Spark API are welcome. When making a contribution, please follow these guidelines:

- Instead of using temporary files, use our `pytest` testing framework.
- When adding new functions, ensure that method signatures comply with those in the PySpark API.

Python Client API

Known Python Issues

Troubleshooting

Running EXPLAIN Renders Newlines

In Python, the output of the `EXPLAIN` statement contains hard line breaks (`\n`):

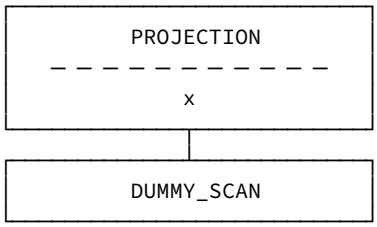
```
In [1]: import duckdb
   ...: duckdb.sql("EXPLAIN SELECT 42 AS x")
```

```
Out[1]:
┌───────────────┬──────────────────────────────────────────────────────────────
│  explain_key  │                          explain_value
│    varchar    │                             varchar
├───────────────┼──────────────────────────────────────────────────────────────
│ physical_plan │ ┌───────────────────────────┐\n│         PROJECTION        │\n│   ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─
│\n│                x      … │
└───────────────┴──────────────────────────────────────────────────────────────
```

To work around this, `print` the output of the `explain()` function:

```
In [2]: print(duckdb.sql("SELECT 42 AS x").explain())
```

```
Out[2]:
```



Please also check out the Jupyter guide for tips on using Jupyter with JupySQL.

Crashes and Errors on Windos

When importing DuckDB on Windows, the Python runtime may crash or return an error upon import or first use:

```
import duckdb

duckdb.sql("...")
```

```
ImportError: DLL load failed while importing duckdb: The specified module could not be found.
```

```
Windows fatal exception: access violation

Current thread 0x0000311c (most recent call first):
  File "<stdin>", line 1 in <module>

Process finished with exit code -1073741819 (0xC0000005)
```

The problem is likely caused by using an outdated Microsoft Visual C++ (MSVC) Redistributable package. The solution is to install the latest MSVC Redistributable package. Alternatively, you can instruct `pip` to compile the package from source as follows:

```
python3 -m pip install duckdb --no-binary duckdb
```

Known Issues

Unfortunately there are some issues that are either beyond our control or are very elusive / hard to track down. Below is a list of these issues that you might have to be aware of, depending on your workflow.

Numpy Import Multithreading

When making use of multi threading and fetching results either directly as Numpy arrays or indirectly through a Pandas DataFrame, it might be necessary to ensure that `numpy.core.multiarray` is imported. If this module has not been imported from the main thread, and a different thread during execution attempts to import it this causes either a deadlock or a crash.

To avoid this, it's recommended to `import numpy.core.multiarray` before starting up threads.

DESCRIBE and SUMMARIZE Return Empty Tables in Jupyter

The `DESCRIBE` and `SUMMARIZE` statements return an empty table:

```
%sql
CREATE OR REPLACE TABLE tbl AS (SELECT 42 AS x);
DESCRIBE tbl;
```

To work around this, wrap them into a subquery:

```
%sql
CREATE OR REPLACE TABLE tbl AS (SELECT 42 AS x);
FROM (DESCRIBE tbl);
```

Protobuf Error for JupySQL in IPython

Loading the JupySQL extension in IPython fails:

```
In [1]: %load_ext sql
```

```
ImportError: cannot import name 'builder' from 'google.protobuf.internal' (unknown location)
```

The solution is to fix the `protobuf` package. This may require uninstalling conflicting packages, e.g.:

```
%pip uninstall tensorflow
%pip install protobuf
```

R Client

> The latest version of the DuckDB R client is {{ site.current_duckdb_r_version }}.

Installation

duckdb: R Client

The DuckDB R client can be installed using the following command:

```
install.packages("duckdb")
```

Please see the installation page for details.

duckplyr: dplyr Client

DuckDB offers a dplyr-compatible API via the `duckplyr` package. It can be installed using `install.packages("duckplyr")`. For details, see the `duckplyr` documentation.

Reference Manual

The reference manual for the DuckDB R client is available at r.duckdb.org.

Basic Client Usage

The standard DuckDB R client implements the DBI interface for R. If you are not familiar with DBI yet, see the Using DBI page for an introduction.

Startup & Shutdown

To use DuckDB, you must first create a connection object that represents the database. The connection object takes as parameter the database file to read and write from. If the database file does not exist, it will be created (the file extension may be `.db`, `.duckdb`, or anything else). The special value `:memory:` (the default) can be used to create an **in-memory database**. Note that for an in-memory database no data is persisted to disk (i.e., all data is lost when you exit the R process). If you would like to connect to an existing database in read-only mode, set the `read_only` flag to TRUE. Read-only mode is required if multiple R processes want to access the same database file at the same time.

```
library("duckdb")
# to start an in-memory database
con <- dbConnect(duckdb())
# or
con <- dbConnect(duckdb(), dbdir = ":memory:")
# to use a database file (not shared between processes)
con <- dbConnect(duckdb(), dbdir = "my-db.duckdb", read_only = FALSE)
# to use a database file (shared between processes)
con <- dbConnect(duckdb(), dbdir = "my-db.duckdb", read_only = TRUE)
```

Connections are closed implicitly when they go out of scope or if they are explicitly closed using `dbDisconnect()`. To shut down the database instance associated with the connection, use `dbDisconnect(con, shutdown = TRUE)`

Querying

DuckDB supports the standard DBI methods to send queries and retrieve result sets. `dbExecute()` is meant for queries where no results are expected like `CREATE TABLE` or `UPDATE` etc. and `dbGetQuery()` is meant to be used for queries that produce results (e.g., `SELECT`). Below an example.

```
# create a table
dbExecute(con, "CREATE TABLE items (item VARCHAR, value DECIMAL(10, 2), count INTEGER)")
# insert two items into the table
dbExecute(con, "INSERT INTO items VALUES ('jeans', 20.0, 1), ('hammer', 42.2, 2)")

# retrieve the items again
res <- dbGetQuery(con, "SELECT * FROM items")
print(res)
#     item value count
# 1  jeans  20.0     1
# 2 hammer  42.2     2
```

DuckDB also supports prepared statements in the R client with the `dbExecute` and `dbGetQuery` methods. Here is an example:

```
# prepared statement parameters are given as a list
dbExecute(con, "INSERT INTO items VALUES (?, ?, ?)", list('laptop', 2000, 1))

# if you want to reuse a prepared statement multiple times, use dbSendStatement() and dbBind()
stmt <- dbSendStatement(con, "INSERT INTO items VALUES (?, ?, ?)")
dbBind(stmt, list('iphone', 300, 2))
dbBind(stmt, list('android', 3.5, 1))
dbClearResult(stmt)

# query the database using a prepared statement
res <- dbGetQuery(con, "SELECT item FROM items WHERE value > ?", list(400))
print(res)
#       item
# 1 laptop
```

> **Warning.** Do **not** use prepared statements to insert large amounts of data into DuckDB. See below for better options.

Efficient Transfer

To write a R data frame into DuckDB, use the standard DBI function `dbWriteTable()`. This creates a table in DuckDB and populates it with the data frame contents. For example:

```
dbWriteTable(con, "iris_table", iris)
res <- dbGetQuery(con, "SELECT * FROM iris_table LIMIT 1")
print(res)
#   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
# 1          5.1         3.5          1.4         0.2  setosa
```

It is also possible to "register" a R data frame as a virtual table, comparable to a SQL `VIEW`. This *does not actually transfer data* into DuckDB yet. Below is an example:

```
duckdb_register(con, "iris_view", iris)
res <- dbGetQuery(con, "SELECT * FROM iris_view LIMIT 1")
print(res)
#   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
# 1          5.1         3.5          1.4         0.2  setosa
```

> DuckDB keeps a reference to the R data frame after registration. This prevents the data frame from being garbage-collected. The reference is cleared when the connection is closed, but can also be cleared manually using the `duckdb_unregister()` method.

Also refer to the data import documentation for more options of efficiently importing data.

dbplyr

DuckDB also plays well with the dbplyr / dplyr packages for programmatic query construction from R. Here is an example:

```r
library("duckdb")
library("dplyr")
con <- dbConnect(duckdb())
duckdb_register(con, "flights", nycflights13::flights)

tbl(con, "flights") |>
  group_by(dest) |>
  summarise(delay = mean(dep_time, na.rm = TRUE)) |>
  collect()
```

When using dbplyr, CSV and Parquet files can be read using the `dplyr::tbl` function.

```r
# Establish a CSV for the sake of this example
write.csv(mtcars, "mtcars.csv")

# Summarize the dataset in DuckDB to avoid reading the entire CSV into R's memory
tbl(con, "mtcars.csv") |>
  group_by(cyl) |>
  summarise(across(disp:wt, .fns = mean)) |>
  collect()

# Establish a set of Parquet files
dbExecute(con, "COPY flights TO 'dataset' (FORMAT parquet, PARTITION_BY (year, month))")

# Summarize the dataset in DuckDB to avoid reading 12 Parquet files into R's memory
tbl(con, "read_parquet('dataset/**/*.parquet', hive_partitioning = true)") |>
  filter(month == "3") |>
  summarise(delay = mean(dep_time, na.rm = TRUE)) |>
  collect()
```

Memory Limit

You can use the `memory_limit` configuration option to limit the memory use of DuckDB, e.g.:

```r
SET memory_limit = '2GB';
```

Note that this limit is only applied to the memory DuckDB uses and it does not affect the memory use of other R libraries. Therefore, the total memory used by the R process may be higher than the configured `memory_limit`.

Troubleshooting

Warning When Installing on macOS

On macOS, installing DuckDB may result in a warning `unable to load shared object '.../R_X11.so'`:

```
Warning message:
In doTryCatch(return(expr), name, parentenv, handler) :
  unable to load shared object '/Library/Frameworks/R.framework/Resources/modules//R_X11.so':
  dlopen(/Library/Frameworks/R.framework/Resources/modules//R_X11.so, 0x0006): Library not loaded:
/opt/X11/lib/libSM.6.dylib
  Referenced from: <31EADEB5-0A17-3546-9944-9B3747071FE8>
/Library/Frameworks/R.framework/Versions/4.4-arm64/Resources/modules/R_X11.so
  Reason: tried: '/opt/X11/lib/libSM.6.dylib' (no such file) ...
> ')
```

Note that this is just a warning, so the simplest solution is to ignore it. Alternatively, you can install DuckDB from the R-universe:

```r
install.packages("duckdb", repos = c("https://duckdb.r-universe.dev", "https://cloud.r-project.org"))
```

You may also install the optional `xquartz` dependency via Homebrew.

Rust Client

> The latest version of the DuckDB Rust client is {{ site.current_duckdb_rust_version }}.

Installation

The DuckDB Rust client can be installed from crates.io. Please see the docs.rs for details.

Basic API Usage

duckdb-rs is an ergonomic wrapper based on the DuckDB C API, please refer to the README for details.

Startup & Shutdown

To use duckdb, you must first initialize a `Connection` handle using `Connection::open()`. `Connection::open()` takes as parameter the database file to read and write from. If the database file does not exist, it will be created (the file extension may be `.db`, `.duckdb`, or anything else). You can also use `Connection::open_in_memory()` to create an **in-memory database**. Note that for an in-memory database no data is persisted to disk (i.e., all data is lost when you exit the process).

```
use duckdb::{params, Connection, Result};
let conn = Connection::open_in_memory()?;
```

The `Connection` will automatically close the underlying db connection for you when it goes out of scope (via `Drop`). You can also explicitly close the `Connection` with `conn.close()`. This is not much difference between these in the typical case, but in case there is an error, you'll have the chance to handle it with the explicit close.

Querying

SQL queries can be sent to DuckDB using the `execute()` method of connections, or we can also prepare the statement and then query on that.

```
#[derive(Debug)]
struct Person {
    id: i32,
    name: String,
    data: Option<Vec<u8>>,
}

conn.execute(
    "INSERT INTO person (name, data) VALUES (?, ?)",
    params![me.name, me.data],
)?;

let mut stmt = conn.prepare("SELECT id, name, data FROM person")?;
let person_iter = stmt.query_map([], |row| {
    Ok(Person {
        id: row.get(0)?,
```

```
        name: row.get(1)?,
        data: row.get(2)?,
    })
})?;

for person in person_iter {
    println!("Found person {:?}", person.unwrap());
}
```

Appender

The Rust client supports the DuckDB Appender API for bulk inserts. For example:

```
fn insert_rows(conn: &Connection) -> Result<()> {
    let mut app = conn.appender("foo")?;
    app.append_rows([[1, 2], [3, 4], [5, 6], [7, 8], [9, 10]])?;
    Ok(())
}
```

Swift Client

DuckDB has a Swift client. See the announcement post for details.

Instantiating DuckDB

DuckDB supports both in-memory and persistent databases. To work with an in-memory datatabase, run:

```
let database = try Database(store: .inMemory)
```

To work with a persistent database, run:

```
let database = try Database(store: .file(at: "test.db"))
```

Queries can be issued through a database connection.

```
let connection = try database.connect()
```

DuckDB supports multiple connections per database.

Application Example

The rest of the page is based on the example of our announcement post, which uses raw data from NASA's Exoplanet Archive loaded directly into DuckDB.

Creating an Application-Specific Type

We first create an application-specific type that we'll use to house our database and connection and through which we'll eventually define our app-specific queries.

```
import DuckDB

final class ExoplanetStore {

  let database: Database
  let connection: Connection

  init(database: Database, connection: Connection) {
    self.database = database
    self.connection = connection
  }
}
```

Loading a CSV File

We load the data from NASA's Exoplanet Archive:

```
wget https://exoplanetarchive.ipac.caltech.edu/TAP/sync?query=select+pl_name+,+disc_
year+from+pscomppars&format=csv -O downloaded_exoplanets.csv
```

Once we have our CSV downloaded locally, we can use the following SQL command to load it as a new table to DuckDB:

```
CREATE TABLE exoplanets AS
    SELECT * FROM read_csv('downloaded_exoplanets.csv');
```

Let's package this up as a new asynchronous factory method on our `ExoplanetStore` type:

```
import DuckDB
import Foundation

final class ExoplanetStore {

  // Factory method to create and prepare a new ExoplanetStore
  static func create() async throws -> ExoplanetStore {

  // Create our database and connection as described above
    let database = try Database(store: .inMemory)
    let connection = try database.connect()

  // Download the CSV from the exoplanet archive
  let (csvFileURL, _) = try await URLSession.shared.download(
    from: URL(string: "https://exoplanetarchive.ipac.caltech.edu/TAP/sync?query=select+pl_name+,+disc_
year+from+pscomppars&format=csv")!)

  // Issue our first query to DuckDB
  try connection.execute("""
      CREATE TABLE exoplanets AS
          SELECT * FROM read_csv('\(csvFileURL.path)');
  """)

  // Create our pre-populated ExoplanetStore instance
    return ExoplanetStore(
    database: database,
      connection: connection
  )
  }

  // Let's make the initializer we defined previously
  // private. This prevents anyone accidentally instantiating
  // the store without having pre-loaded our Exoplanet CSV
  // into the database
  private init(database: Database, connection: Connection) {
  ...
  }
}
```

Querying the Database

The following example queires DuckDB from within Swift via an async function. This means the callee won't be blocked while the query is executing. We'll then cast the result columns to Swift native types using DuckDB's `ResultSet` `cast(to:)` family of methods, before finally wrapping them up in a `DataFrame` from the TabularData framework.

```
...

import TabularData

extension ExoplanetStore {

  // Retrieves the number of exoplanets discovered by year
```

```
  func groupedByDiscoveryYear() async throws -> DataFrame {

  // Issue the query we described above
    let result = try connection.query("""
      SELECT disc_year, count(disc_year) AS Count
        FROM exoplanets
        GROUP BY disc_year
        ORDER BY disc_year
      """)

    // Cast our DuckDB columns to their native Swift
    // equivalent types
    let discoveryYearColumn = result[0].cast(to: Int.self)
    let countColumn = result[1].cast(to: Int.self)

    // Use our DuckDB columns to instantiate TabularData
    // columns and populate a TabularData DataFrame
    return DataFrame(columns: [
      TabularData.Column(discoveryYearColumn).eraseToAnyColumn(),
      TabularData.Column(countColumn).eraseToAnyColumn(),
    ])
  }
}
```

Complete Project

For the complete example project, clone the DuckDB Swift repository and open up the runnable app project located in `Examples/SwiftUI/ExoplanetExplorer.xcodeproj`.

Wasm

DuckDB Wasm

> The latest version of the DuckDB WebAssembly client is {{ site.current_duckdb_wasm_version }}.

DuckDB has been compiled to WebAssembly, so it can run inside any browser on any device.

DuckDB-Wasm offers a layered API, it can be embedded as a JavaScript + WebAssembly library, as a Web shell, or built from source according to your needs.

Getting Started with DuckDB-Wasm

A great starting point is to read the DuckDB-Wasm launch blog post!

Another great resource is the GitHub repository.

For details, see the full DuckDB-Wasm API Documentation.

Limitations

- By default, the WebAssembly client only uses a single thread.
- The WebAssembly client has a limited amount of memory available. WebAssembly limits the amount of available memory to 4 GB and browsers may impose even stricter limits.

Instantiation

DuckDB-Wasm has multiple ways to be instantiated depending on the use case.

cdn(jsdelivr)

```
import * as duckdb from '@duckdb/duckdb-wasm';

const JSDELIVR_BUNDLES = duckdb.getJsDelivrBundles();

// Select a bundle based on browser checks
const bundle = await duckdb.selectBundle(JSDELIVR_BUNDLES);

const worker_url = URL.createObjectURL(
  new Blob([`importScripts("${bundle.mainWorker!}");`], {type: 'text/javascript'})
);

// Instantiate the asynchronus version of DuckDB-Wasm
const worker = new Worker(worker_url);
const logger = new duckdb.ConsoleLogger();
```

```
const db = new duckdb.AsyncDuckDB(logger, worker);
await db.instantiate(bundle.mainModule, bundle.pthreadWorker);
URL.revokeObjectURL(worker_url);
```

webpack

```
import * as duckdb from '@duckdb/duckdb-wasm';
import duckdb_wasm from '@duckdb/duckdb-wasm/dist/duckdb-mvp.wasm';
import duckdb_wasm_next from '@duckdb/duckdb-wasm/dist/duckdb-eh.wasm';
const MANUAL_BUNDLES: duckdb.DuckDBBundles = {
    mvp: {
        mainModule: duckdb_wasm,
        mainWorker: new URL('@duckdb/duckdb-wasm/dist/duckdb-browser-mvp.worker.js',
import.meta.url).toString(),
    },
    eh: {
        mainModule: duckdb_wasm_next,
        mainWorker: new URL('@duckdb/duckdb-wasm/dist/duckdb-browser-eh.worker.js',
import.meta.url).toString(),
    },
};
// Select a bundle based on browser checks
const bundle = await duckdb.selectBundle(MANUAL_BUNDLES);
// Instantiate the asynchronus version of DuckDB-Wasm
const worker = new Worker(bundle.mainWorker!);
const logger = new duckdb.ConsoleLogger();
const db = new duckdb.AsyncDuckDB(logger, worker);
await db.instantiate(bundle.mainModule, bundle.pthreadWorker);
```

vite

```
import * as duckdb from '@duckdb/duckdb-wasm';
import duckdb_wasm from '@duckdb/duckdb-wasm/dist/duckdb-mvp.wasm?url';
import mvp_worker from '@duckdb/duckdb-wasm/dist/duckdb-browser-mvp.worker.js?url';
import duckdb_wasm_eh from '@duckdb/duckdb-wasm/dist/duckdb-eh.wasm?url';
import eh_worker from '@duckdb/duckdb-wasm/dist/duckdb-browser-eh.worker.js?url';

const MANUAL_BUNDLES: duckdb.DuckDBBundles = {
    mvp: {
        mainModule: duckdb_wasm,
        mainWorker: mvp_worker,
    },
    eh: {
        mainModule: duckdb_wasm_eh,
        mainWorker: eh_worker,
    },
};
// Select a bundle based on browser checks
const bundle = await duckdb.selectBundle(MANUAL_BUNDLES);
// Instantiate the asynchronus version of DuckDB-wasm
const worker = new Worker(bundle.mainWorker!);
const logger = new duckdb.ConsoleLogger();
const db = new duckdb.AsyncDuckDB(logger, worker);
await db.instantiate(bundle.mainModule, bundle.pthreadWorker);
```

Statically Served

It is possible to manually download the files from https://cdn.jsdelivr.net/npm/@duckdb/duckdb-wasm/dist/.

```
import * as duckdb from '@duckdb/duckdb-wasm';

const MANUAL_BUNDLES: duckdb.DuckDBBundles = {
    mvp: {
        mainModule: 'change/me/../duckdb-mvp.wasm',
        mainWorker: 'change/me/../duckdb-browser-mvp.worker.js',
    },
    eh: {
        mainModule: 'change/m/../duckdb-eh.wasm',
        mainWorker: 'change/m/../duckdb-browser-eh.worker.js',
    },
};
// Select a bundle based on browser checks
const bundle = await duckdb.selectBundle(MANUAL_BUNDLES);
// Instantiate the asynchronous version of DuckDB-Wasm
const worker = new Worker(bundle.mainWorker!);
const logger = new duckdb.ConsoleLogger();
const db = new duckdb.AsyncDuckDB(logger, worker);
await db.instantiate(bundle.mainModule, bundle.pthreadWorker);
```

Data Ingestion

DuckDB-Wasm has multiple ways to import data, depending on the format of the data.

There are two steps to import data into DuckDB.

First, the data file is imported into a local file system using register functions (registerEmptyFileBuffer, registerFileBuffer, registerFileHandle, registerFileText, registerFileURL).

Then, the data file is imported into DuckDB using insert functions (insertArrowFromIPCStream, insertArrowTable, insertCSVFromPath, insertJSONFromPath) or directly using FROM SQL query (using extensions like Parquet or Wasm-flavored httpfs).

Insert statements can also be used to import data.

Data Import

Open & Close Connection

```
// Create a new connection
const c = await db.connect();

// ... import data

// Close the connection to release memory
await c.close();
```

Apache Arrow

```
// Data can be inserted from an existing arrow.Table
// More Example https://arrow.apache.org/docs/js/
import { tableFromArrays } from 'apache-arrow';
```

```
// EOS signal according to Arrorw IPC streaming format
// See https://arrow.apache.org/docs/format/Columnar.html#ipc-streaming-format
const EOS = new Uint8Array([255, 255, 255, 255, 0, 0, 0, 0]);

const arrowTable = tableFromArrays({
  id: [1, 2, 3],
  name: ['John', 'Jane', 'Jack'],
  age: [20, 21, 22],
});

await c.insertArrowTable(arrowTable, { name: 'arrow_table' });
// Write EOS
await c.insertArrowTable(EOS, { name: 'arrow_table' });

// ..., from a raw Arrow IPC stream
const streamResponse = await fetch(` someapi`);
const streamReader = streamResponse.body.getReader();
const streamInserts = [];
while (true) {
    const { value, done } = await streamReader.read();
    if (done) break;
    streamInserts.push(c.insertArrowFromIPCStream(value, { name: 'streamed' }));
}

// Write EOS
streamInserts.push(c.insertArrowFromIPCStream(EOS, { name: 'streamed' }));

await Promise.all(streamInserts);
```

CSV

```
// ..., from CSV files
// (interchangeable: registerFile{Text,Buffer,URL,Handle})
const csvContent = '1|foo\n2|bar\n';
await db.registerFileText(` data.csv`, csvContent);
// ... with typed insert options
await c.insertCSVFromPath('data.csv', {
    schema: 'main',
    name: 'foo',
    detect: false,
    header: false,
    delimiter: '|',
    columns: {
        col1: new arrow.Int32(),
        col2: new arrow.Utf8(),
    },
});
```

JSON

```
// ..., from JSON documents in row-major format
const jsonRowContent = [
    { "col1": 1, "col2": "foo" },
    { "col1": 2, "col2": "bar" },
];
await db.registerFileText(
```

```
    'rows.json',
    JSON.stringify(jsonRowContent),
);
await c.insertJSONFromPath('rows.json', { name: 'rows' });

// ... or column-major format
const jsonColContent = {
    "col1": [1, 2],
    "col2": ["foo", "bar"]
};
await db.registerFileText(
    'columns.json',
    JSON.stringify(jsonColContent),
);
await c.insertJSONFromPath('columns.json', { name: 'columns' });

// From API
const streamResponse = await fetch(` someapi/content.json`);
await db.registerFileBuffer('file.json', new Uint8Array(await streamResponse.arrayBuffer()))
await c.insertJSONFromPath('file.json', { name: 'JSONContent' });
```

Parquet

```
// from Parquet files
// ...Local
const pickedFile: File = letUserPickFile();
await db.registerFileHandle('local.parquet', pickedFile, DuckDBDataProtocol.BROWSER_FILEREADER, true);
// ...Remote
await db.registerFileURL('remote.parquet', 'https://origin/remote.parquet', DuckDBDataProtocol.HTTP,
false);
// ... Using Fetch
const res = await fetch('https://origin/remote.parquet');
await db.registerFileBuffer('buffer.parquet', new Uint8Array(await res.arrayBuffer()));

// ..., by specifying URLs in the SQL text
await c.query(`
    CREATE TABLE direct AS
        SELECT * FROM 'https://origin/remote.parquet'
`);
// ..., or by executing raw insert statements
await c.query(`
    INSERT INTO existing_table
    VALUES (1, 'foo'), (2, 'bar')`);
```

httpfs (Wasm-Flavored)

```
// ..., by specifying URLs in the SQL text
await c.query(`
    CREATE TABLE direct AS
        SELECT * FROM 'https://origin/remote.parquet'
`);
```

> **Tip.** If you encounter a Network Error (`Failed to execute 'send' on 'XMLHttpRequest'`) when you try to query files from S3, configure the S3 permission CORS header. For example:

```
[
    {
```

```
        "AllowedHeaders": [
            "*"
        ],
        "AllowedMethods": [
            "GET",
            "HEAD"
        ],
        "AllowedOrigins": [
            "*"
        ],
        "ExposeHeaders": [],
        "MaxAgeSeconds": 3000
    }
]
```

Insert Statement

```
// ..., or by executing raw insert statements
await c.query(`
    INSERT INTO existing_table
    VALUES (1, 'foo'), (2, 'bar')`);
```

Query

DuckDB-Wasm provides functions for querying data. Queries are run sequentially.

First, a connection need to be created by calling connect. Then, queries can be run by calling query or send.

Query Execution

```
// Create a new connection
const conn = await db.connect();

// Either materialize the query result
await conn.query<{ v: arrow.Int }>(`
    SELECT * FROM generate_series(1, 100) t(v)
`);
// ..., or fetch the result chunks lazily
for await (const batch of await conn.send<{ v: arrow.Int }>(`
    SELECT * FROM generate_series(1, 100) t(v)
`)) {
    // ...
}

// Close the connection to release memory
await conn.close();
```

Prepared Statements

```
// Create a new connection
const conn = await db.connect();
// Prepare query
const stmt = await conn.prepare(` SELECT v + ? FROM generate_series(0, 10_000) t(v);`);
```

```
// ... and run the query with materialized results
await stmt.query(234);
// ... or result chunks
for await (const batch of await stmt.send(234)) {
    // ...
}
// Close the statement to release memory
await stmt.close();
// Closing the connection will release statements as well
await conn.close();
```

Arrow Table to JSON

```
// Create a new connection
const conn = await db.connect();

// Query
const arrowResult = await conn.query<{ v: arrow.Int }>(`
    SELECT * FROM generate_series(1, 100) t(v)
`);

// Convert arrow table to json
const result = arrowResult.toArray().map((row) => row.toJSON());

// Close the connection to release memory
await conn.close();
```

Export Parquet

```
// Create a new connection
const conn = await db.connect();

// Export Parquet
conn.send(` COPY (SELECT * FROM tbl) TO 'result-snappy.parquet' (FORMAT parquet);`);
const parquet_buffer = await this._db.copyFileToBuffer('result-snappy.parquet');

// Generate a download link
const link = URL.createObjectURL(new Blob([parquet_buffer]));

// Close the connection to release memory
await conn.close();
```

Extensions

DuckDB-Wasm's (dynamic) extension loading is modeled after the regular DuckDB's extension loading, with a few relevant differences due to the difference in platform.

Format

Extensions in DuckDB are binaries to be dynamically loaded via `dlopen`. A cryptographical signature is appended to the binary. An extension in DuckDB-Wasm is a regular Wasm file to be dynamically loaded via Emscripten's `dlopen`. A cryptographical signature is appended to the Wasm file as a WebAssembly custom section called `duckdb_signature`. This ensures the file remains a valid WebAssembly file.

> Currently, we require this custom section to be the last one, but this can be potentially relaxed in the future.

`INSTALL` and `LOAD`

The `INSTALL` semantic in native embeddings of DuckDB is to fetch, decompress from `gzip` and store data in local disk. The `LOAD` semantic in native embeddings of DuckDB is to (optionally) perform signature checks *and* dynamic load the binary with the main DuckDB binary.

In DuckDB-Wasm, `INSTALL` is a no-op given there is no durable cross-session storage. The `LOAD` operation will fetch (and decompress on the fly), perform signature checks *and* dynamically load via the Emscripten implementation of `dlopen`.

Autoloading

Autoloading, i.e., the possibility for DuckDB to add extension functionality on-the-fly, is enabled by default in DuckDB-Wasm.

List of Officially Available Extensions

Extension name	Description	Aliases
autocomplete	Adds support for autocomplete in the shell	
excel	Adds support for Excel-like format strings	
fts	Adds support for Full-Text Search Indexes	
icu	Adds support for time zones and collations using the ICU library	
inet	Adds support for IP-related data types and functions	
json	Adds support for JSON operations	
parquet	Adds support for reading and writing Parquet files	
sqlite	Adds support for reading SQLite database files	sqlite, sqlite3
sqlsmith		
tpcds	Adds TPC-DS data generation and query support	
tpch	Adds TPC-H data generation and query support	

WebAssembly is basically an additional platform, and there might be platform-specific limitations that make some extensions not able to match their native capabilities or to perform them in a different way. We will document here relevant differences for DuckDB-hosted extensions.

HTTPFS

The HTTPFS extension is, at the moment, not available in DuckDB-Wasm. Https protocol capabilities needs to go through an additional layer, the browser, which adds both differences and some restrictions to what is doable from native.

Instead, DuckDB-Wasm has a separate implementation that for most purposes is interchangeable, but does not support all use cases (as it must follow security rules imposed by the browser, such as CORS). Due to this CORS restriction, any requests for data made using the HTTPFS extension must be to websites that allow (using CORS headers) the website hosting the DuckDB-Wasm instance to access that data. The MDN website is a great resource for more information regarding CORS.

Extension Signing

As with regular DuckDB extensions, DuckDB-Wasm extension are by default checked on LOAD to verify the signature confirm the extension has not been tampered with. Extension signature verification can be disabled via a configuration option. Signing is a property of the binary itself, so copying a DuckDB extension (say to serve it from a different location) will still keep a valid signature (e.g., for local development).

Fetching DuckDB-Wasm Extensions

Official DuckDB extensions are served at `extensions.duckdb.org`, and this is also the default value for the `default_extension_repository` option. When installing extensions, a relevant URL will be built that will look like `extensions.duckdb.org/$duckdb_version_hash/$duckdb_platform/$name.duckdb_extension.gz`.

DuckDB-Wasm extension are fetched only on load, and the URL will look like: `extensions.duckdb.org/duckdb-wasm/$duckdb_version_hash/$duckdb_platform/$name.duckdb_extension.wasm`.

Note that an additional `duckdb-wasm` is added to the folder structure, and the file is served as a `.wasm` file.

DuckDB-Wasm extensions are served pre-compressed using Brotli compression. While fetched from a browser, extensions will be transparently uncompressed. If you want to fetch the `duckdb-wasm` extension manually, you can use `curl --compress extensions.duckdb.org/<...>/icu.duckdb_extension.wasm`.

Serving Extensions from a Third-Party Repository

As with regular DuckDB, if you use `SET custom_extension_repository = some.url.com`, subsequent loads will be attempted at `some.url.com/duckdb-wasm/$duckdb_version_hash/$duckdb_platform/$name.duckdb_extension.wasm`.

Note that GET requests on the extensions needs to be CORS enabled for a browser to allow the connection.

Tooling

Both DuckDB-Wasm and its extensions have been compiled using latest packaged Emscripten toolchain.

SQL

SQL Introduction

Here we provide an overview of how to perform simple operations in SQL. This tutorial is only intended to give you an introduction and is in no way a complete tutorial on SQL. This tutorial is adapted from the PostgreSQL tutorial.

> DuckDB's SQL dialect closely follows the conventions of the PostgreSQL dialect. The few exceptions to this are listed on the PostgreSQL compatibility page.

In the examples that follow, we assume that you have installed the DuckDB Command Line Interface (CLI) shell. See the installation page for information on how to install the CLI.

Concepts

DuckDB is a relational database management system (RDBMS). That means it is a system for managing data stored in relations. A relation is essentially a mathematical term for a table.

Each table is a named collection of rows. Each row of a given table has the same set of named columns, and each column is of a specific data type. Tables themselves are stored inside schemas, and a collection of schemas constitutes the entire database that you can access.

Creating a New Table

You can create a new table by specifying the table name, along with all column names and their types:

```
CREATE TABLE weather (
    city    VARCHAR,
    temp_lo INTEGER, -- minimum temperature on a day
    temp_hi INTEGER, -- maximum temperature on a day
    prcp    FLOAT,
    date    DATE
);
```

You can enter this into the shell with the line breaks. The command is not terminated until the semicolon.

White space (i.e., spaces, tabs, and newlines) can be used freely in SQL commands. That means you can type the command aligned differently than above, or even all on one line. Two dash characters (`--`) introduce comments. Whatever follows them is ignored up to the end of the line. SQL is case-insensitive about keywords and identifiers. When returning identifiers, their original cases are preserved.

In the SQL command, we first specify the type of command that we want to perform: `CREATE TABLE`. After that follows the parameters for the command. First, the table name, `weather`, is given. Then the column names and column types follow.

`city VARCHAR` specifies that the table has a column called `city` that is of type VARCHAR. VARCHAR specifies a data type that can store text of arbitrary length. The temperature fields are stored in an `INTEGER` type, a type that stores integer numbers (i.e., whole numbers without a decimal point). `FLOAT` columns store single precision floating-point numbers (i.e., numbers with a decimal point). `DATE` stores a date (i.e., year, month, day combination). `DATE` only stores the specific day, not a time associated with that day.

DuckDB supports the standard SQL types `INTEGER`, `SMALLINT`, `FLOAT`, `DOUBLE`, `DECIMAL`, `CHAR(n)`, `VARCHAR(n)`, `DATE`, `TIME` and `TIMESTAMP`.

The second example will store cities and their associated geographical location:

```
CREATE TABLE cities (
    name VARCHAR,
    lat  DECIMAL,
    lon  DECIMAL
);
```

Finally, it should be mentioned that if you don't need a table any longer or want to recreate it differently you can remove it using the following command:

```
DROP TABLE <tablename>;
```

Populating a Table with Rows

The insert statement is used to populate a table with rows:

```
INSERT INTO weather
VALUES ('San Francisco', 46, 50, 0.25, '1994-11-27');
```

Constants that are not numeric values (e.g., text and dates) must be surrounded by single quotes (`''`), as in the example. Input dates for the date type must be formatted as `'YYYY-MM-DD'`.

We can insert into the `cities` table in the same manner.

```
INSERT INTO cities
VALUES ('San Francisco', -194.0, 53.0);
```

The syntax used so far requires you to remember the order of the columns. An alternative syntax allows you to list the columns explicitly:

```
INSERT INTO weather (city, temp_lo, temp_hi, prcp, date)
VALUES ('San Francisco', 43, 57, 0.0, '1994-11-29');
```

You can list the columns in a different order if you wish or even omit some columns, e.g., if the `prcp` is unknown:

```
INSERT INTO weather (date, city, temp_hi, temp_lo)
VALUES ('1994-11-29', 'Hayward', 54, 37);
```

> **Tip.** Many developers consider explicitly listing the columns better style than relying on the order implicitly.

Please enter all the commands shown above so you have some data to work with in the following sections.

Alternatively, you can use the `COPY` statement. This is faster for large amounts of data because the `COPY` command is optimized for bulk loading while allowing less flexibility than `INSERT`. An example with `weather.csv` would be:

```
COPY weather
FROM 'weather.csv';
```

Where the file name for the source file must be available on the machine running the process. There are many other ways of loading data into DuckDB, see the corresponding documentation section for more information.

Querying a Table

To retrieve data from a table, the table is queried. A SQL `SELECT` statement is used to do this. The statement is divided into a select list (the part that lists the columns to be returned), a table list (the part that lists the tables from which to retrieve the data), and an optional qualification (the part that specifies any restrictions). For example, to retrieve all the rows of table weather, type:

```
SELECT *
FROM weather;
```

Here `*` is a shorthand for “all columns”. So the same result would be had with:

```
SELECT city, temp_lo, temp_hi, prcp, date
FROM weather;
```

The output should be:

city	temp_lo	temp_hi	prcp	date
San Francisco	46	50	0.25	1994-11-27
San Francisco	43	57	0.0	1994-11-29
Hayward	37	54	NULL	1994-11-29

You can write expressions, not just simple column references, in the select list. For example, you can do:

```
SELECT city, (temp_hi + temp_lo) / 2 AS temp_avg, date
FROM weather;
```

This should give:

city	temp_avg	date
San Francisco	48.0	1994-11-27
San Francisco	50.0	1994-11-29
Hayward	45.5	1994-11-29

Notice how the AS clause is used to relabel the output column. (The AS clause is optional.)

A query can be "qualified" by adding a WHERE clause that specifies which rows are wanted. The WHERE clause contains a Boolean (truth value) expression, and only rows for which the Boolean expression is true are returned. The usual Boolean operators (AND, OR, and NOT) are allowed in the qualification. For example, the following retrieves the weather of San Francisco on rainy days:

```
SELECT *
FROM weather
WHERE city = 'San Francisco'
  AND prcp > 0.0;
```

Result:

city	temp_lo	temp_hi	prcp	date
San Francisco	46	50	0.25	1994-11-27

You can request that the results of a query be returned in sorted order:

```
SELECT *
FROM weather
ORDER BY city;
```

city	temp_lo	temp_hi	prcp	date
Hayward	37	54	NULL	1994-11-29
San Francisco	43	57	0.0	1994-11-29
San Francisco	46	50	0.25	1994-11-27

In this example, the sort order isn't fully specified, and so you might get the San Francisco rows in either order. But you'd always get the results shown above if you do:

```
SELECT *
FROM weather
ORDER BY city, temp_lo;
```

You can request that duplicate rows be removed from the result of a query:

```
SELECT DISTINCT city
FROM weather;
```

city
San Francisco
Hayward

Here again, the result row ordering might vary. You can ensure consistent results by using `DISTINCT` and `ORDER BY` together:

```
SELECT DISTINCT city
FROM weather
ORDER BY city;
```

Joins between Tables

Thus far, our queries have only accessed one table at a time. Queries can access multiple tables at once, or access the same table in such a way that multiple rows of the table are being processed at the same time. A query that accesses multiple rows of the same or different tables at one time is called a join query. As an example, say you wish to list all the weather records together with the location of the associated city. To do that, we need to compare the city column of each row of the `weather` table with the name column of all rows in the `cities` table, and select the pairs of rows where these values match.

This would be accomplished by the following query:

```
SELECT *
FROM weather, cities
WHERE city = name;
```

city	temp_lo	temp_hi	prcp	date	name	lat	lon
San Francisco	46	50	0.25	1994-11-27	San Francisco	-194.000	53.000
San Francisco	43	57	0.0	1994-11-29	San Francisco	-194.000	53.000

Observe two things about the result set:

- There is no result row for the city of Hayward. This is because there is no matching entry in the `cities` table for Hayward, so the join ignores the unmatched rows in the `weather` table. We will see shortly how this can be fixed.
- There are two columns containing the city name. This is correct because the lists of columns from the `weather` and `cities` tables are concatenated. In practice this is undesirable, though, so you will probably want to list the output columns explicitly rather than using `*`:

```
SELECT city, temp_lo, temp_hi, prcp, date, lon, lat
FROM weather, cities
WHERE city = name;
```

city	temp_lo	temp_hi	prcp	date	lon	lat
San Francisco	46	50	0.25	1994-11-27	53.000	-194.000
San Francisco	43	57	0.0	1994-11-29	53.000	-194.000

Since the columns all had different names, the parser automatically found which table they belong to. If there were duplicate column names in the two tables you'd need to qualify the column names to show which one you meant, as in:

```
SELECT weather.city, weather.temp_lo, weather.temp_hi,
       weather.prcp, weather.date, cities.lon, cities.lat
FROM weather, cities
WHERE cities.name = weather.city;
```

It is widely considered good style to qualify all column names in a join query, so that the query won't fail if a duplicate column name is later added to one of the tables.

Join queries of the kind seen thus far can also be written in this alternative form:

```
SELECT *
FROM weather
INNER JOIN cities ON weather.city = cities.name;
```

This syntax is not as commonly used as the one above, but we show it here to help you understand the following topics.

Now we will figure out how we can get the Hayward records back in. What we want the query to do is to scan the `weather` table and for each row to find the matching cities row(s). If no matching row is found we want some “empty values” to be substituted for the `cities` table's columns. This kind of query is called an outer join. (The joins we have seen so far are inner joins.) The command looks like this:

```
SELECT *
FROM weather
LEFT OUTER JOIN cities ON weather.city = cities.name;
```

city	temp_lo	temp_hi	prcp	date	name	lat	lon
San Francisco	46	50	0.25	1994-11-27	San Francisco	-194.000	53.000
San Francisco	43	57	0.0	1994-11-29	San Francisco	-194.000	53.000
Hayward	37	54	NULL	1994-11-29	NULL	NULL	NULL

This query is called a left outer join because the table mentioned on the left of the join operator will have each of its rows in the output at least once, whereas the table on the right will only have those rows output that match some row of the left table. When outputting a left-table row for which there is no right-table match, empty (null) values are substituted for the right-table columns.

Aggregate Functions

Like most other relational database products, DuckDB supports aggregate functions. An aggregate function computes a single result from multiple input rows. For example, there are aggregates to compute the `count`, `sum`, `avg` (average), `max` (maximum) and `min` (minimum) over a set of rows.

As an example, we can find the highest low-temperature reading anywhere with:

```
SELECT max(temp_lo)
FROM weather;
```

max(temp_lo)
46

If we wanted to know what city (or cities) that reading occurred in, we might try:

```
SELECT city
FROM weather
WHERE temp_lo = max(temp_lo);
```

But this will not work since the aggregate max cannot be used in the WHERE clause:

```
Binder Error:
WHERE clause cannot contain aggregates!
```

This restriction exists because the WHERE clause determines which rows will be included in the aggregate calculation; so obviously it has to be evaluated before aggregate functions are computed. However, as is often the case the query can be restated to accomplish the desired result, here by using a subquery:

```
SELECT city
FROM weather
WHERE temp_lo = (SELECT max(temp_lo) FROM weather);
```

city
San Francisco

This is OK because the subquery is an independent computation that computes its own aggregate separately from what is happening in the outer query.

Aggregates are also very useful in combination with GROUP BY clauses. For example, we can get the maximum low temperature observed in each city with:

```
SELECT city, max(temp_lo)
FROM weather
GROUP BY city;
```

city	max(temp_lo)
San Francisco	46
Hayward	37

Which gives us one output row per city. Each aggregate result is computed over the table rows matching that city. We can filter these grouped rows using HAVING:

```
SELECT city, max(temp_lo)
FROM weather
GROUP BY city
HAVING max(temp_lo) < 40;
```

city	max(temp_lo)
Hayward	37

which gives us the same results for only the cities that have all temp_lo values below 40. Finally, if we only care about cities whose names begin with S, we can use the LIKE operator:

```
SELECT city, max(temp_lo)
FROM weather
WHERE city LIKE 'S%'            -- (1)
GROUP BY city
HAVING max(temp_lo) < 40;
```

More information about the LIKE operator can be found in the pattern matching page.

It is important to understand the interaction between aggregates and SQL's WHERE and HAVING clauses. The fundamental difference between WHERE and HAVING is this: WHERE selects input rows before groups and aggregates are computed (thus, it controls which rows go into the aggregate computation), whereas HAVING selects group rows after groups and aggregates are computed. Thus, the WHERE clause must not contain aggregate functions; it makes no sense to try to use an aggregate to determine which rows will be inputs to the aggregates. On the other hand, the HAVING clause always contains aggregate functions.

In the previous example, we can apply the city name restriction in WHERE, since it needs no aggregate. This is more efficient than adding the restriction to HAVING, because we avoid doing the grouping and aggregate calculations for all rows that fail the WHERE check.

Updates

You can update existing rows using the UPDATE command. Suppose you discover the temperature readings are all off by 2 degrees after November 28. You can correct the data as follows:

```
UPDATE weather
SET temp_hi = temp_hi - 2,  temp_lo = temp_lo - 2
WHERE date > '1994-11-28';
```

Look at the new state of the data:

```
SELECT *
FROM weather;
```

city	temp_lo	temp_hi	prcp	date
San Francisco	46	50	0.25	1994-11-27
San Francisco	41	55	0.0	1994-11-29
Hayward	35	52	NULL	1994-11-29

Deletions

Rows can be removed from a table using the DELETE command. Suppose you are no longer interested in the weather of Hayward. Then you can do the following to delete those rows from the table:

```
DELETE FROM weather
WHERE city = 'Hayward';
```

All weather records belonging to Hayward are removed.

```
SELECT *
FROM weather;
```

city	temp_lo	temp_hi	prcp	date
San Francisco	46	50	0.25	1994-11-27
San Francisco	41	55	0.0	1994-11-29

One should be cautious when issuing statements of the following form:

```
DELETE FROM <table_name>;
```

> **Warning.** Without a qualification, DELETE will remove all rows from the given table, leaving it empty. The system will not request confirmation before doing this.

Statements

Statements Overview

ANALYZE Statement

The ANALYZE statement recomputes the statistics on DuckDB's tables.

Usage

The statistics recomputed by the ANALYZE statement are only used for join order optimization. It is therefore recommended to recompute these statistics for improved join orders, especially after performing large updates (inserts and/or deletes).

To recompute the statistics, run:

```
ANALYZE;
```

ALTER TABLE Statement

The ALTER TABLE statement changes the schema of an existing table in the catalog.

Examples

```
CREATE TABLE integers (i INTEGER, j INTEGER);
```

Add a new column with name k to the table integers, it will be filled with the default value NULL:

```
ALTER TABLE integers
ADD COLUMN k INTEGER;
```

Add a new column with name l to the table integers, it will be filled with the default value 10:

```
ALTER TABLE integers
ADD COLUMN l INTEGER DEFAULT 10;
```

Drop the column k from the table integers:

```
ALTER TABLE integers
DROP k;
```

Change the type of the column i to the type VARCHAR using a standard cast:

```
ALTER TABLE integers
ALTER i TYPE VARCHAR;
```

Change the type of the column i to the type VARCHAR, using the specified expression to convert the data for each row:

```
ALTER TABLE integers
ALTER i SET DATA TYPE VARCHAR USING concat(i, '_', j);
```

Set the default value of a column:

```
ALTER TABLE integers
ALTER COLUMN i SET DEFAULT 10;
```

Drop the default value of a column:

```
ALTER TABLE integers
ALTER COLUMN i DROP DEFAULT;
```

Make a column not nullable:

```
ALTER TABLE integers
ALTER COLUMN i SET NOT NULL;
```

Drop the not-NULL constraint:

```
ALTER TABLE integers
ALTER COLUMN i DROP NOT NULL;
```

Rename a table:

```
ALTER TABLE integers
RENAME TO integers_old;
```

Rename a column of a table:

```
ALTER TABLE integers
RENAME i TO ii;
```

Add a primary key to a column of a table:

```
ALTER TABLE integers
ADD PRIMARY KEY (i);
```

Syntax

`ALTER TABLE` changes the schema of an existing table. All the changes made by `ALTER TABLE` fully respect the transactional semantics, i.e., they will not be visible to other transactions until committed, and can be fully reverted through a rollback.

`RENAME TABLE`

Rename a table:

```
ALTER TABLE integers
RENAME TO integers_old;
```

The `RENAME TO` clause renames an entire table, changing its name in the schema. Note that any views that rely on the table are **not** automatically updated.

`RENAME COLUMN`

To rename a column of a table, use the `RENAME` or `RENAME COLUMN` clauses:

```
ALTER TABLE integers
RENAME COLUMN i TO j;
```

```
ALTER TABLE integers
RENAME i TO j;
```

The `RENAME [COLUMN]` clause renames a single column within a table. Any constraints that rely on this name (e.g., `CHECK` constraints) are automatically updated. However, note that any views that rely on this column name are **not** automatically updated.

ADD COLUMN

To add a column of a table, use the ADD or ADD COLUMN clauses.

E.g., to add a new column with name k to the table `integers`, it will be filled with the default value NULL:

```
ALTER TABLE integers
ADD COLUMN k INTEGER;
```

Or:

```
ALTER TABLE integers
ADD k INTEGER;
```

Add a new column with name `l` to the table integers, it will be filled with the default value 10:

```
ALTER TABLE integers
ADD COLUMN l INTEGER DEFAULT 10;
```

The ADD [COLUMN] clause can be used to add a new column of a specified type to a table. The new column will be filled with the specified default value, or NULL if none is specified.

DROP COLUMN

To drop a column of a table, use the DROP or DROP COLUMN clause:

E.g., to drop the column k from the table `integers`:

```
ALTER TABLE integers
DROP COLUMN k;
```

Or:

```
ALTER TABLE integers
DROP k;
```

The DROP [COLUMN] clause can be used to remove a column from a table. Note that columns can only be removed if they do not have any indexes that rely on them. This includes any indexes created as part of a PRIMARY KEY or UNIQUE constraint. Columns that are part of multi-column check constraints cannot be dropped either. If you attempt to drop a column with an index on it, DuckDB will return the following error message:

```
Dependency Error:
Cannot alter entry "..." because there are entries that depend on it.
```

[SET [DATA]] TYPE

Change the type of the column `i` to the type VARCHAR using a standard cast:

```
ALTER TABLE integers
ALTER i TYPE VARCHAR;
```

> Instead of ALTER <column_name> TYPE <type>{:.language-sql .highlight}, you can also use the equivalent ALTER <column_name> SET TYPE <type>{:.language-sql .highlight} and the ALTER <column_name> SET DATA TYPE <type>{:.language-sql .highlight} clauses.

Change the type of the column `i` to the type VARCHAR, using the specified expression to convert the data for each row:

```
ALTER TABLE integers
ALTER i SET DATA TYPE VARCHAR USING concat(i, '_', j);
```

The `[SET [DATA]] TYPE` clause changes the type of a column in a table. Any data present in the column is converted according to the provided expression in the `USING` clause, or, if the `USING` clause is absent, cast to the new data type. Note that columns can only have their type changed if they do not have any indexes that rely on them and are not part of any `CHECK` constraints.

Handling Structs

There are two options to change the sub-schema of a STRUCT-typed column.

`ALTER TABLE` with `struct_insert`

You can use `ALTER TABLE` with the `struct_insert` function. For example:

```
CREATE TABLE tbl (col STRUCT(i INTEGER));
ALTER TABLE tbl
ALTER col TYPE USING struct_insert(col, a := 42, b := NULL::VARCHAR);
```

`ALTER TABLE` with `ADD COLUMN` / `DROP COLUMN` / `RENAME COLUMN`

Starting with DuckDB v1.3.0, `ALTER TABLE` supports the `ADD COLUMN`, `DROP COLUMN` and `RENAME COLUMN` clauses to update the sub-schema of a STRUCT.

`SET` / `DROP DEFAULT`

Set the default value of a column:

```
ALTER TABLE integers
ALTER COLUMN i SET DEFAULT 10;
```

Drop the default value of a column:

```
ALTER TABLE integers
ALTER COLUMN i DROP DEFAULT;
```

The `SET`/`DROP DEFAULT` clause modifies the `DEFAULT` value of an existing column. Note that this does not modify any existing data in the column. Dropping the default is equivalent to setting the default value to NULL.

> **Warning.** At the moment DuckDB will not allow you to alter a table if there are any dependencies. That means that if you have an index on a column you will first need to drop the index, alter the table, and then recreate the index. Otherwise, you will get a `Dependency Error`.

`ADD PRIMARY KEY`

Add a primary key to a column of a table:

```
ALTER TABLE integers
ADD PRIMARY KEY (i);
```

Add a primary key to multiple columns of a table:

```
ALTER TABLE integers
ADD PRIMARY KEY (i, j);
```

ADD / DROP CONSTRAINT

> ADD CONSTRAINT and DROP CONSTRAINT clauses are not yet supported in DuckDB.

Limitations

ALTER COLUMN fails if values of conflicting types have occurred in the table at any point, even if they have been deleted:

```
CREATE TABLE tbl (col VARCHAR);

INSERT INTO tbl
VALUES ('asdf'), ('42');

DELETE FROM tbl
WHERE col = 'asdf';

ALTER TABLE tbl
ALTER COLUMN col TYPE INTEGER;
```

```
Conversion Error:
Could not convert string 'asdf' to INT32
```

Currently, this is expect behavior. As a workaround, you can create a copy of the table:

```
CREATE OR REPLACE TABLE tbl AS FROM tbl;
```

ALTER VIEW Statement

The ALTER VIEW statement changes the schema of an existing view in the catalog.

Examples

Rename a view:

```
ALTER VIEW v1 RENAME TO v2;
```

ALTER VIEW changes the schema of an existing table. All the changes made by ALTER VIEW fully respect the transactional semantics, i.e., they will not be visible to other transactions until committed, and can be fully reverted through a rollback. Note that other views that rely on the table are **not** automatically updated.

ATTACH and DETACH Statements

DuckDB allows attaching to and detaching from database files.

Examples

Attach the database `file.db` with the alias inferred from the name (`file`):

```
ATTACH 'file.db';
```

Attach the database `file.db` with an explicit alias (`file_db`):

```
ATTACH 'file.db' AS file_db;
```

Attach the database `file.db` in read only mode:

```
ATTACH 'file.db' (READ_ONLY);
```

Attach the database `file.db` with a block size of 16 kB:

```
ATTACH 'file.db' (BLOCK_SIZE 16_384);
```

Attach a SQLite database for reading and writing (see the `sqlite` extension for more information):

```
ATTACH 'sqlite_file.db' AS sqlite_db (TYPE sqlite);
```

Attach the database `file.db` if inferred database alias `file` does not yet exist:

```
ATTACH IF NOT EXISTS 'file.db';
```

Attach the database `file.db` if explicit database alias `file_db` does not yet exist:

```
ATTACH IF NOT EXISTS 'file.db' AS file_db;
```

Attach the database `file2.db` as alias `file_db` detaching and replacing the existing alias if it exists:

```
ATTACH OR REPLACE 'file2.db' AS file_db;
```

Create a table in the attached database with alias `file`:

```
CREATE TABLE file.new_table (i INTEGER);
```

Detach the database with alias `file`:

```
DETACH file;
```

Show a list of all attached databases:

```
SHOW DATABASES;
```

Change the default database that is used to the database `file`:

```
USE file;
```

ATTACH

The ATTACH statement adds a new database file to the catalog that can be read from and written to. Note that attachment definitions are not persisted between sessions: when a new session is launched, you have to re-attach to all databases.

Attach Syntax

ATTACH allows DuckDB to operate on multiple database files, and allows for transfer of data between different database files.

ATTACH supports HTTP and S3 endpoints. For these, it creates a read-only connection by default. Therefore, the following two commands are equivalent:

```
ATTACH 'https://blobs.duckdb.org/databases/stations.duckdb' AS stations_db;
ATTACH 'https://blobs.duckdb.org/databases/stations.duckdb' AS stations_db (READ_ONLY);
```

Similarly, the following two commands connecting to S3 are equivalent:

```
ATTACH 's3://duckdb-blobs/databases/stations.duckdb' AS stations_db;
ATTACH 's3://duckdb-blobs/databases/stations.duckdb' AS stations_db (READ_ONLY);
```

> Prior to DuckDB version 1.1.0, it was necessary to specify the READ_ONLY flag for HTTP and S3 endpoints.

Explicit Storage Versions

DuckDB v1.2.0 introduced the STORAGE_VERSION option, which allows explicilty specifying the storage version. Using this, you can opt-in to newer forwards-incompatible features:

```
ATTACH 'file.db' (STORAGE_VERSION 'v1.2.0');
```

This setting specifies the minimum DuckDB version that should be able to read the database file. When database files are written with this option, the resulting files cannot be opened by older DuckDB released versions than the specified version. They can be read by the specified version and all newer versions of DuckDB.

For more details, see the "Storage" page.

DETACH

The DETACH statement allows previously attached database files to be closed and detached, releasing any locks held on the database file.

Note that it is not possible to detach from the default database: if you would like to do so, issue the USE statement to change the default database to another one. For example, if you are connected to a persistent database, you may change to an in-memory database by issuing:

```
ATTACH ':memory:' AS memory_db;
USE memory_db;
```

> **Warning.** Closing the connection, e.g., invoking the `close()` function in Python, does not release the locks held on the database files as the file handles are held by the main DuckDB instance (in Python's case, the `duckdb` module).

Detach Syntax

Options

Name	Description	Type	Default value
`access_mode`	Access mode of the database (**AUTOMATIC**, **READ_ONLY**, or **READ_WRITE**).	VARCHAR	`automatic`
`type`	The file type (**DUCKDB** or **SQLITE**), or deduced from the input string literal (MySQL, PostgreSQL).	VARCHAR	DUCKDB
`block_size`	The block size of a new database file. Must be a power of two and within [16384, 262144]. Cannot be set for existing files.	UBIGINT	262144

Name Qualification

The fully qualified name of catalog objects contains the *catalog*, the *schema* and the *name* of the object. For example:

Attach the database new_db:

```
ATTACH 'new_db.db';
```

Create the schema `my_schema` in the database `new_db`:

```
CREATE SCHEMA new_db.my_schema;
```

Create the table `my_table` in the schema `my_schema`:

```
CREATE TABLE new_db.my_schema.my_table (col INTEGER);
```

Refer to the column `col` inside the table `my_table`:

```
SELECT new_db.my_schema.my_table.col FROM new_db.my_schema.my_table;
```

Note that often the fully qualified name is not required. When a name is not fully qualified, the system looks for which entries to reference using the *catalog search path*. The default catalog search path includes the system catalog, the temporary catalog and the initially attached database together with the `main` schema.

Also note the rules on identifiers and database names in particular.

Default Database and Schema

When a table is created without any qualifications, the table is created in the default schema of the default database. The default database is the database that is launched when the system is created – and the default schema is `main`.

Create the table `my_table` in the default database:

```
CREATE TABLE my_table (col INTEGER);
```

Changing the Default Database and Schema

The default database and schema can be changed using the USE command.

Set the default database schema to `new_db.main`:

```
USE new_db;
```

Set the default database schema to `new_db.my_schema`:

```
USE new_db.my_schema;
```

Resolving Conflicts

When providing only a single qualification, the system can interpret this as *either* a catalog *or* a schema, as long as there are no conflicts. For example:

```
ATTACH 'new_db.db';
CREATE SCHEMA my_schema;
```

Creates the table `new_db.main.tbl`:

```
CREATE TABLE new_db.tbl (i INTEGER);
```

Creates the table `default_db.my_schema.tbl`:

```
CREATE TABLE my_schema.tbl (i INTEGER);
```

If we create a conflict (i.e., we have both a schema and a catalog with the same name) the system requests that a fully qualified path is used instead:

```
CREATE SCHEMA new_db;
CREATE TABLE new_db.tbl (i INTEGER);
```

```
Binder Error:
Ambiguous reference to catalog or schema "new_db" - use a fully qualified path like "memory.new_db"
```

Changing the Catalog Search Path

The catalog search path can be adjusted by setting the `search_path` configuration option, which uses a comma-separated list of values that will be on the search path. The following example demonstrates searching in two databases:

```
ATTACH ':memory:' AS db1;
ATTACH ':memory:' AS db2;
CREATE table db1.tbl1 (i INTEGER);
CREATE table db2.tbl2 (j INTEGER);
```

Reference the tables using their fully qualified name:

```
SELECT * FROM db1.tbl1;
SELECT * FROM db2.tbl2;
```

Or set the search path and reference the tables using their name:

```
SET search_path = 'db1,db2';
SELECT * FROM tbl1;
SELECT * FROM tbl2;
```

Transactional Semantics

When running queries on multiple databases, the system opens separate transactions per database. The transactions are started *lazily* by default – when a given database is referenced for the first time in a query, a transaction for that database will be started. `SET immediate_transaction_mode = true` can be toggled to change this behavior to eagerly start transactions in all attached databases instead.

While multiple transactions can be active at a time – the system only supports *writing* to a single attached database in a single transaction. If you try to write to multiple attached databases in a single transaction the following error will be thrown:

```
Attempting to write to database "db2" in a transaction that has already modified database "db1" –
a single transaction can only write to a single attached database.
```

The reason for this restriction is that the system does not maintain atomicity for transactions across attached databases. Transactions are only atomic *within* each database file. By restricting the global transaction to write to only a single database file the atomicity guarantees are maintained.

CALL Statement

The `CALL` statement invokes the given table function and returns the results.

Examples

Invoke the 'duckdb_functions' table function:

```
CALL duckdb_functions();
```

Invoke the 'pragma_table_info' table function:

```
CALL pragma_table_info('pg_am');
```

Select only the functions where the name starts with ST_:

```
SELECT function_name, parameters, parameter_types, return_type
FROM duckdb_functions()
WHERE function_name LIKE 'ST_%';
```

Syntax

CHECKPOINT Statement

The `CHECKPOINT` statement synchronizes data in the write-ahead log (WAL) to the database data file. For in-memory databases this statement will succeed with no effect.

Examples

Synchronize data in the default database:

```
CHECKPOINT;
```

Synchronize data in the specified database:

```
CHECKPOINT file_db;
```

Abort any in-progress transactions to synchronize the data:

```
FORCE CHECKPOINT;
```

Syntax

Checkpoint operations happen automatically based on the WAL size (see Configuration). This statement is for manual checkpoint actions.

Behavior

The default `CHECKPOINT` command will fail if there are any running transactions. Including `FORCE` will abort any transactions and execute the checkpoint operation.

Also see the related PRAGMA option for further behavior modification.

Reclaiming Space

When performing a checkpoint (automatic or otherwise), the space occupied by deleted rows is partially reclaimed. Note that this does not remove all deleted rows, but rather merges row groups that have a significant amount of deletes together. In the current implementation this requires ~25% of rows to be deleted in adjacent row groups.

When running in in-memory mode, checkpointing has no effect, hence it does not reclaim space after deletes in in-memory databases.

> **Warning.** The VACUUM statement does *not* trigger vacuuming deletes and hence does not reclaim space.

COMMENT ON Statement

The `COMMENT ON` statement allows adding metadata to catalog entries (tables, columns, etc.). It follows the PostgreSQL syntax.

Examples

Create a comment on a TABLE:

```
COMMENT ON TABLE test_table IS 'very nice table';
```

Create a comment on a COLUMN:

```
COMMENT ON COLUMN test_table.test_table_column IS 'very nice column';
```

Create a comment on a VIEW:

```
COMMENT ON VIEW test_view IS 'very nice view';
```

Create a comment on an INDEX:

```
COMMENT ON INDEX test_index IS 'very nice index';
```

Create a comment on a SEQUENCE:

```
COMMENT ON SEQUENCE test_sequence IS 'very nice sequence';
```

Create a comment on a TYPE:

```
COMMENT ON TYPE test_type IS 'very nice type';
```

Create a comment on a MACRO:

```
COMMENT ON MACRO test_macro IS 'very nice macro';
```

Create a comment on a MACRO TABLE:

```
COMMENT ON MACRO TABLE test_table_macro IS 'very nice table macro';
```

To unset a comment, set it to NULL, e.g.:

```
COMMENT ON TABLE test_table IS NULL;
```

Reading Comments

Comments can be read by querying the `comment` column of the respective metadata functions:

List comments on TABLEs:

```
SELECT comment FROM duckdb_tables();
```

List comments on COLUMNs:

```
SELECT comment FROM duckdb_columns();
```

List comments on VIEWs:

```
SELECT comment FROM duckdb_views();
```

List comments on INDEXs:

```
SELECT comment FROM duckdb_indexes();
```

List comments on SEQUENCEs:

```
SELECT comment FROM duckdb_sequences();
```

List comments on TYPEs:

```
SELECT comment FROM duckdb_types();
```

List comments on MACROs:

```
SELECT comment FROM duckdb_functions();
```

List comments on MACRO TABLEs:

```
SELECT comment FROM duckdb_functions();
```

Limitations

The COMMENT ON statement currently has the following limitations:

- It is not possible to comment on schemas or databases.
- It is not possible to comment on things that have a dependency (e.g., a table with an index).

Syntax

COPY Statement

Examples

Read a CSV file into the `lineitem` table, using auto-detected CSV options:

```
COPY lineitem FROM 'lineitem.csv';
```

Read a CSV file into the `lineitem` table, using manually specified CSV options:

```
COPY lineitem FROM 'lineitem.csv' (DELIMITER '|');
```

Read a Parquet file into the `lineitem` table:

```
COPY lineitem FROM 'lineitem.pq' (FORMAT parquet);
```

Read a JSON file into the `lineitem` table, using auto-detected options:

```
COPY lineitem FROM 'lineitem.json' (FORMAT json, AUTO_DETECT true);
```

Read a CSV file into the `lineitem` table, using double quotes:

```
COPY lineitem FROM "lineitem.csv";
```

Read a CSV file into the `lineitem` table, omitting quotes:

```
COPY lineitem FROM lineitem.csv;
```

Write a table to a CSV file:

```
COPY lineitem TO 'lineitem.csv' (FORMAT csv, DELIMITER '|', HEADER);
```

Write a table to a CSV file, using double quotes:

```
COPY lineitem TO "lineitem.csv";
```

Write a table to a CSV file, omitting quotes:

```
COPY lineitem TO lineitem.csv;
```

Write the result of a query to a Parquet file:

```
COPY (SELECT l_orderkey, l_partkey FROM lineitem) TO 'lineitem.parquet' (COMPRESSION zstd);
```

Copy the entire content of database db1 to database db2:

```
COPY FROM DATABASE db1 TO db2;
```

Copy only the schema (catalog elements) but not any data:

```
COPY FROM DATABASE db1 TO db2 (SCHEMA);
```

Overview

COPY moves data between DuckDB and external files. COPY ... FROM imports data into DuckDB from an external file. COPY ... TO writes data from DuckDB to an external file. The COPY command can be used for CSV, PARQUET and JSON files.

COPY ... FROM

COPY ... FROM imports data from an external file into an existing table. The data is appended to whatever data is in the table already. The amount of columns inside the file must match the amount of columns in the table table_name, and the contents of the columns must be convertible to the column types of the table. In case this is not possible, an error will be thrown.

If a list of columns is specified, COPY will only copy the data in the specified columns from the file. If there are any columns in the table that are not in the column list, COPY ... FROM will insert the default values for those columns

Copy the contents of a comma-separated file test.csv without a header into the table test:

```
COPY test FROM 'test.csv';
```

Copy the contents of a comma-separated file with a header into the category table:

```
COPY category FROM 'categories.csv' (HEADER);
```

Copy the contents of lineitem.tbl into the lineitem table, where the contents are delimited by a pipe character (|):

```
COPY lineitem FROM 'lineitem.tbl' (DELIMITER '|');
```

Copy the contents of lineitem.tbl into the lineitem table, where the delimiter, quote character, and presence of a header are automatically detected:

```
COPY lineitem FROM 'lineitem.tbl' (AUTO_DETECT true);
```

Read the contents of a comma-separated file names.csv into the name column of the category table. Any other columns of this table are filled with their default value:

```
COPY category(name) FROM 'names.csv';
```

Read the contents of a Parquet file lineitem.parquet into the lineitem table:

```
COPY lineitem FROM 'lineitem.parquet' (FORMAT parquet);
```

Read the contents of a newline-delimited JSON file lineitem.ndjson into the lineitem table:

```
COPY lineitem FROM 'lineitem.ndjson' (FORMAT json);
```

Read the contents of a JSON file lineitem.json into the lineitem table:

```
COPY lineitem FROM 'lineitem.json' (FORMAT json, ARRAY true);
```

Syntax

> To ensure compatibility with PostgreSQL, DuckDB accepts `COPY ... FROM` statements that do not fully comply with the railroad diagram shown here. For example, the following is a valid statement:
>
> ```
> COPY tbl FROM 'tbl.csv' WITH DELIMITER '|' CSV HEADER;
> ```

`COPY ... TO`

`COPY ... TO` exports data from DuckDB to an external CSV or Parquet file. It has mostly the same set of options as `COPY ... FROM`, however, in the case of `COPY ... TO` the options specify how the file should be written to disk. Any file created by `COPY ... TO` can be copied back into the database by using `COPY ... FROM` with a similar set of options.

The `COPY ... TO` function can be called specifying either a table name, or a query. When a table name is specified, the contents of the entire table will be written into the resulting file. When a query is specified, the query is executed and the result of the query is written to the resulting file.

Copy the contents of the `lineitem` table to a CSV file with a header:

```
COPY lineitem TO 'lineitem.csv';
```

Copy the contents of the `lineitem` table to the file `lineitem.tbl`, where the columns are delimited by a pipe character (`|`), including a header line:

```
COPY lineitem TO 'lineitem.tbl' (DELIMITER '|');
```

Use tab separators to create a TSV file without a header:

```
COPY lineitem TO 'lineitem.tsv' (DELIMITER '\t', HEADER false);
```

Copy the l_orderkey column of the `lineitem` table to the file `orderkey.tbl`:

```
COPY lineitem(l_orderkey) TO 'orderkey.tbl' (DELIMITER '|');
```

Copy the result of a query to the file `query.csv`, including a header with column names:

```
COPY (SELECT 42 AS a, 'hello' AS b) TO 'query.csv' (DELIMITER ',');
```

Copy the result of a query to the Parquet file `query.parquet`:

```
COPY (SELECT 42 AS a, 'hello' AS b) TO 'query.parquet' (FORMAT parquet);
```

Copy the result of a query to the newline-delimited JSON file `query.ndjson`:

```
COPY (SELECT 42 AS a, 'hello' AS b) TO 'query.ndjson' (FORMAT json);
```

Copy the result of a query to the JSON file `query.json`:

```
COPY (SELECT 42 AS a, 'hello' AS b) TO 'query.json' (FORMAT json, ARRAY true);
```

Return the files and their column statistics that were written as part of the COPY statement:

```
COPY (SELECT l_orderkey, l_comment FROM lineitem) TO 'lineitem_part.parquet' (RETURN_STATS);
```

filename	count	file_size_bytes	footer_size_bytes	column_statistics	partition_keys
lineitem_part.parquet	600572	8579141	1445	{'”l_comment”'={column_size_bytes=7642227, max=zzle. slyly, min=' Tiresias above the blit', null_count=0}, '”l_orderkey”'={column_size_bytes=935457, max=600000, min=1, null_count=0}}	NULL

Note: for nested columns (e.g., structs) the column statistics are defined for each part. For example, if we have a column `name STRUCT(field1 INTEGER, field2 INTEGER)` the column statistics will have stats for `name.field1` and `name.field2`.

COPY ... TO Options

Zero or more copy options may be provided as a part of the copy operation. The `WITH` specifier is optional, but if any options are specified, the parentheses are required. Parameter values can be passed in with or without wrapping in single quotes.

Any option that is a Boolean can be enabled or disabled in multiple ways. You can write `true`, `ON`, or `1` to enable the option, and `false`, `OFF`, or `0` to disable it. The `BOOLEAN` value can also be omitted, e.g., by only passing `(HEADER)`, in which case `true` is assumed.

With few exceptions, the below options are applicable to all formats written with `COPY`.

Name	Description	Type	Default
`FORMAT`	Specifies the copy function to use. The default is selected from the file extension (e.g., `.parquet` results in a Parquet file being written/read). If the file extension is unknown `CSV` is selected. Vanilla DuckDB provides `CSV`, `PARQUET` and `JSON` but additional copy functions can be added by `extensions`.	`VARCHAR`	`auto`
`USE_TMP_FILE`	Whether or not to write to a temporary file first if the original file exists (`target.csv.tmp`). This prevents overwriting an existing file with a broken file in case the writing is cancelled.	`BOOL`	`auto`
`OVERWRITE_OR_IGNORE`	Whether or not to allow overwriting files if they already exist. Only has an effect when used with `PARTITION_BY`.	`BOOL`	`false`
`OVERWRITE`	When `true`, all existing files inside targeted directories will be removed (not supported on remote filesystems). Only has an effect when used with `PARTITION_BY`.	`BOOL`	`false`
`APPEND`	When `true`, in the event a filename pattern is generated that already exists, the path will be regenerated to ensure no existing files are overwritten. Only has an effect when used with `PARTITION_BY`.	`BOOL`	`false`
`FILENAME_PATTERN`	Set a pattern to use for the filename, can optionally contain `{uuid}` / `{uuidv4}` or `{uuidv7}` to be filled in with a generated UUID (v4 or v7, respectively), and `{i}`, which is replaced by an incrementing index. Only has an effect when used with `PARTITION_BY`.	`VARCHAR`	`auto`
`FILE_EXTENSION`	Set the file extension that should be assigned to the generated file(s).	`VARCHAR`	`auto`
`PER_THREAD_OUTPUT`	When `true`, the `COPY` command generates one file per thread, rather than one file in total. This allows for faster parallel writing.	`BOOL`	`false`
`FILE_SIZE_BYTES`	If this parameter is set, the `COPY` process creates a directory which will contain the exported files. If a file exceeds the set limit (specified as bytes such as `1000` or in human-readable format such as `1k`), the process creates a new file in the directory. This parameter works in combination with `PER_THREAD_OUTPUT`. Note that the size is used as an approximation, and files can be occasionally slightly over the limit.	`VARCHAR` or `BIGINT`	(empty)
`PARTITION_BY`	The columns to partition by using a Hive partitioning scheme, see the partitioned writes section.	`VARCHAR[]`	(empty)

Name	Description	Type	Default
PRESERVE_ORDER	Whether or not to preserve order during the copy operation. Defaults to the value of the `preserve_insertion_order` configuration option.	BOOL	(*)
RETURN_FILES	Whether or not to include the created filepath(s) (as a `files VARCHAR[]` column) in the query result.	BOOL	`false`
RETURN_STATS	Whether or not to return the files and their column statistics that were written as part of the COPY statement.	BOOL	`false`
WRITE_PARTITION_COLUMNS	Whether or not to write partition columns into files. Only has an effect when used with PARTITION_BY.	BOOL	`false`

Syntax

> To ensure compatibility with PostgreSQL, DuckDB accepts `COPY ... TO` statements that do not fully comply with the railroad diagram shown here. For example, the following is a valid statement:
>
> ```
> COPY (SELECT 42 AS x, 84 AS y) TO 'out.csv' WITH DELIMITER '|' CSV HEADER;
> ```

COPY FROM DATABASE ... TO

The `COPY FROM DATABASE ... TO` statement copies the entire content from one attached database to another attached database. This includes the schema, including constraints, indexes, sequences, macros, and the data itself.

```
ATTACH 'db1.db' AS db1;
CREATE TABLE db1.tbl AS SELECT 42 AS x, 3 AS y;
CREATE MACRO db1.two_x_plus_y(x, y) AS 2 * x + y;

ATTACH 'db2.db' AS db2;
COPY FROM DATABASE db1 TO db2;
SELECT db2.two_x_plus_y(x, y) AS z FROM db2.tbl;
```

z
87

To only copy the **schema** of db1 to db2 but omit copying the data, add SCHEMA to the statement:

```
COPY FROM DATABASE db1 TO db2 (SCHEMA);
```

Syntax

Format-Specific Options

CSV Options

The below options are applicable when writing CSV files.

Name	Description	Type	Default
COMPRESSION	The compression type for the file. By default this will be detected automatically from the file extension (e.g., `file.csv.gz` will use `gzip`, `file.csv.zst` will use `zstd`, and `file.csv` will use none). Options are none, `gzip`, `zstd`.	VARCHAR	`auto`
DATEFORMAT	Specifies the date format to use when writing dates. See Date Format.	VARCHAR	(empty)
DELIM or SEP	The character that is written to separate columns within each row.	VARCHAR	`,`
ESCAPE	The character that should appear before a character that matches the `quote` value.	VARCHAR	"
FORCE_QUOTE	The list of columns to always add quotes to, even if not required.	VARCHAR[]	`[]`
HEADER	Whether or not to write a header for the CSV file.	BOOL	`true`
NULLSTR	The string that is written to represent a `NULL` value.	VARCHAR	(empty)
PREFIX	Prefixes the CSV file with a specified string. This option must be used in conjunction with `SUFFIX` and requires HEADER to be set to `false`.	VARCHAR	(empty)
SUFFIX	Appends a specified string as a suffix to the CSV file. This option must be used in conjunction with `PREFIX` and requires HEADER to be set to `false`.	VARCHAR	(empty)
QUOTE	The quoting character to be used when a data value is quoted.	VARCHAR	"
TIMESTAMPFORMAT	Specifies the date format to use when writing timestamps. See Date Format.	VARCHAR	(empty)

Parquet Options

The below options are applicable when writing Parquet files.

Name	Description	Type	Default
COMPRESSION	The compression format to use (`uncompressed`, `snappy`, `gzip`, `zstd`, `brotli`, `lz4`, `lz4_raw`).	VARCHAR	`snappy`
COMPRESSION_LEVEL	Compression level, set between 1 (lowest compression, fastest) and 22 (highest compression, slowest). Only supported for zstd compression.	BIGINT	3
FIELD_IDS	The `field_id` for each column. Pass `auto` to attempt to infer automatically.	STRUCT	(empty)
ROW_GROUP_SIZE_BYTES	The target size of each row group. You can pass either a human-readable string, e.g., 2MB, or an integer, i.e., the number of bytes. This option is only used when you have issued `SET preserve_insertion_order = false;`, otherwise, it is ignored.	BIGINT	`row_group_size * 1024`
ROW_GROUP_SIZE	The target size, i.e., number of rows, of each row group.	BIGINT	122880

Name	Description	Type	Default
ROW_GROUPS_PER_FILE	Create a new Parquet file if the current one has a specified number of row groups. If multiple threads are active, the number of row groups in a file may slightly exceed the specified number of row groups to limit the amount of locking – similarly to the behaviour of FILE_SIZE_BYTES. However, if per_thread_output is set, only one thread writes to each file, and it becomes accurate again.	BIGINT	(empty)
PARQUET_VERSION	The parquet version to use (V1, V2).	VARCHAR	V1

Some examples of FIELD_IDS are as follows.

Assign field_ids automatically:

```
COPY
    (SELECT 128 AS i)
    TO 'my.parquet'
    (FIELD_IDS 'auto');
```

Sets the field_id of column i to 42:

```
COPY
    (SELECT 128 AS i)
    TO 'my.parquet'
    (FIELD_IDS {i: 42});
```

Sets the field_id of column i to 42, and column j to 43:

```
COPY
    (SELECT 128 AS i, 256 AS j)
    TO 'my.parquet'
    (FIELD_IDS {i: 42, j: 43});
```

Sets the field_id of column my_struct to 43, and column i (nested inside my_struct) to 43:

```
COPY
    (SELECT {i: 128} AS my_struct)
    TO 'my.parquet'
    (FIELD_IDS {my_struct: {__duckdb_field_id: 42, i: 43}});
```

Sets the field_id of column my_list to 42, and column element (default name of list child) to 43:

```
COPY
    (SELECT [128, 256] AS my_list)
    TO 'my.parquet'
    (FIELD_IDS {my_list: {__duckdb_field_id: 42, element: 43}});
```

Sets the field_id of column my_map to 42, and columns key and value (default names of map children) to 43 and 44:

```
COPY
    (SELECT MAP {'key1' : 128, 'key2': 256} my_map)
    TO 'my.parquet'
    (FIELD_IDS {my_map: {__duckdb_field_id: 42, key: 43, value: 44}});
```

JSON Options

The below options are applicable when writing JSON files.

Name	Description	Type	Default
ARRAY	Whether to write a JSON array. If `true`, a JSON array of records is written, if `false`, newline-delimited JSON is written	BOOL	`false`
COMPRESSION	The compression type for the file. By default this will be detected automatically from the file extension (e.g., `file.json.gz` will use `gzip`, `file.json.zst` will use `zstd`, and `file.json` will use none). Options are none, `gzip`, `zstd`.	VARCHAR	`auto`
DATEFORMAT	Specifies the date format to use when writing dates. See Date Format.	VARCHAR	(empty)
TIMESTAMPFORMAT	Specifies the date format to use when writing timestamps. See Date Format.	VARCHAR	(empty)

Limitations

COPY does not support copying between tables. To copy between tables, use an `INSERT statement`:

```
INSERT INTO tbl2
    FROM tbl1;
```

CREATE MACRO Statement

The `CREATE MACRO` statement can create a scalar or table macro (function) in the catalog. A macro may only be a single `SELECT` statement (similar to a `VIEW`), but it has the benefit of accepting parameters.

For a scalar macro, `CREATE MACRO` is followed by the name of the macro, and optionally parameters within a set of parentheses. The keyword `AS` is next, followed by the text of the macro. By design, a scalar macro may only return a single value. For a table macro, the syntax is similar to a scalar macro except `AS` is replaced with `AS TABLE`. A table macro may return a table of arbitrary size and shape.

> If a `MACRO` is temporary, it is only usable within the same database connection and is deleted when the connection is closed.

Examples

Scalar Macros

Create a macro that adds two expressions (a and b):

```
CREATE MACRO add(a, b) AS a + b;
```

Create a macro, replacing possible existing definitions:

```
CREATE OR REPLACE MACRO add(a, b) AS a + b;
```

Create a macro if it does not already exist, else do nothing:

```
CREATE MACRO IF NOT EXISTS add(a, b) AS a + b;
```

Create a macro for a `CASE` expression:

```
CREATE MACRO ifelse(a, b, c) AS CASE WHEN a THEN b ELSE c END;
```

Create a macro that does a subquery:

```
CREATE MACRO one() AS (SELECT 1);
```

Macros are schema-dependent, and have an alias, `FUNCTION`:

```
CREATE FUNCTION main.my_avg(x) AS sum(x) / count(x);
```

Create a macro with default constant parameters:

```
CREATE MACRO add_default(a, b := 5) AS a + b;
```

Create a macro `arr_append` (with a functionality equivalent to `array_append`):

```
CREATE MACRO arr_append(l, e) AS list_concat(l, list_value(e));
```

Table Macros

Create a table macro without parameters:

```
CREATE MACRO static_table() AS TABLE
    SELECT 'Hello' AS column1, 'World' AS column2;
```

Create a table macro with parameters (that can be of any type):

```
CREATE MACRO dynamic_table(col1_value, col2_value) AS TABLE
    SELECT col1_value AS column1, col2_value AS column2;
```

Create a table macro that returns multiple rows. It will be replaced if it already exists, and it is temporary (will be automatically deleted when the connection ends):

```
CREATE OR REPLACE TEMP MACRO dynamic_table(col1_value, col2_value) AS TABLE
    SELECT col1_value AS column1, col2_value AS column2
    UNION ALL
    SELECT 'Hello' AS col1_value, 456 AS col2_value;
```

Pass an argument as a list:

```
CREATE MACRO get_users(i) AS TABLE
    SELECT * FROM users WHERE uid IN (SELECT unnest(i));
```

An example for how to use the `get_users` table macro is the following:

```
CREATE TABLE users AS
    SELECT *
    FROM (VALUES (1, 'Ada'), (2, 'Bob'), (3, 'Carl'), (4, 'Dan'), (5, 'Eve')) t(uid, name);
SELECT * FROM get_users([1, 5]);
```

To define macros on arbitrary tables, use the `query_table` function. For example, the following macro computes a column-wise checksum on a table:

```
CREATE MACRO checksum(table_name) AS TABLE
    SELECT bit_xor(md5_number(COLUMNS(*)::VARCHAR))
    FROM query_table(table_name);

CREATE TABLE tbl AS SELECT unnest([42, 43]) AS x, 100 AS y;
SELECT * FROM checksum('tbl');
```

Overloading

It is possible to overload a macro based on the amount of parameters it takes, this works for both scalar and table macros.

By providing overloads we can have both `add_x(a, b)` and `add_x(a, b, c)` with different function bodies.

```
CREATE MACRO add_x
    (a, b) AS a + b,
    (a, b, c) AS a + b + c;
```

```
SELECT
    add_x(21, 42) AS two_args,
    add_x(21, 42, 21) AS three_args;
```

two_args	three_args
63	84

Syntax

Macros allow you to create shortcuts for combinations of expressions.

```
CREATE MACRO add(a) AS a + b;
```

```
Binder Error:
Referenced column "b" not found in FROM clause!
```

This works:

```
CREATE MACRO add(a, b) AS a + b;
```

Usage example:

```
SELECT add(1, 2) AS x;
```

x
3

However, this fails:

```
SELECT add('hello', 3);
```

```
Binder Error:
Could not choose a best candidate function for the function call "add(STRING_LITERAL, INTEGER_LITERAL)".
In order to select one, please add explicit type casts.
    Candidate functions:
    add(DATE, INTEGER) -> DATE
    add(INTEGER, INTEGER) -> INTEGER
```

Macros can have default parameters. Unlike some languages, default parameters must be named when the macro is invoked.

b is a default parameter:

```
CREATE MACRO add_default(a, b := 5) AS a + b;
```

The following will result in 42:

```
SELECT add_default(37);
```

The following will throw an error:

```
SELECT add_default(40, 2);
```

```
Binder Error:
Macro function 'add_default(a)' requires a single positional argument, but 2 positional arguments were provided.
```

Default parameters must used by assigning them like the following:

```
SELECT add_default(40, b := 2) AS x;
```

x
42

However, the following fails:

```
SELECT add_default(b := 2, 40);
```

```
Binder Error:
Positional parameters cannot come after parameters with a default value!
```

The order of default parameters does not matter:

```
CREATE MACRO triple_add(a, b := 5, c := 10) AS a + b + c;
```

```
SELECT triple_add(40, c := 1, b := 1) AS x;
```

x
42

When macros are used, they are expanded (i.e., replaced with the original expression), and the parameters within the expanded expression are replaced with the supplied arguments. Step by step:

The add macro we defined above is used in a query:

```
SELECT add(40, 2) AS x;
```

Internally, add is replaced with its definition of `a + b`:

```
SELECT a + b; AS x
```

Then, the parameters are replaced by the supplied arguments:

```
SELECT 40 + 2 AS x;
```

Limitations

Using Named Parameters

Currently, positional macro parameters can only be used positionally, and named parameters can only be used by supplying their name. Therefore, the following will not work:

```
CREATE MACRO my_macro(a, b := 42) AS (a + b);
SELECT my_macro(32, 52);
```

```
Binder Error:
Macro function 'my_macro(a)' requires a single positional argument, but 2 positional arguments were provided.
```

Using Subquery Macros

If a MACRO is defined as a subquery, it cannot be invoked in a table function. DuckDB will return the following error:

```
Binder Error:
Table function cannot contain subqueries
```

Overloads

Overloads for macro functions have to be set at creation, it is not possible to define a macro by the same name twice without first removing the first definition.

Recursive Functions

Defining recursive functions is not supported. For example, the following macro – supposed to compute the *n*th number of the Fibonacci sequence – fails:

```
CREATE OR REPLACE FUNCTION fibo(n) AS (SELECT 1);
CREATE OR REPLACE FUNCTION fibo(n) AS (
    CASE
        WHEN n <= 1 THEN 1
        ELSE fibo(n - 1)
    END
);
SELECT fibo(3);
```

```
Binder Error:
Max expression depth limit of 1000 exceeded. Use "SET max_expression_depth TO x" to increase the maximum
expression depth.
```

CREATE SCHEMA Statement

The `CREATE SCHEMA` statement creates a schema in the catalog. The default schema is `main`.

Examples

Create a schema:

```
CREATE SCHEMA s1;
```

Create a schema if it does not exist yet:

```
CREATE SCHEMA IF NOT EXISTS s2;
```

Create a schema or replace a schema if it exists:

```
CREATE OR REPLACE SCHEMA s2;
```

Create table in the schemas:

```
CREATE TABLE s1.t (id INTEGER PRIMARY KEY, other_id INTEGER);
CREATE TABLE s2.t (id INTEGER PRIMARY KEY, j VARCHAR);
```

Compute a join between tables from two schemas:

```
SELECT *
FROM s1.t s1t, s2.t s2t
WHERE s1t.other_id = s2t.id;
```

Syntax

CREATE SECRET Statement

The `CREATE SECRET` statement creates a new secret in the Secrets Manager.

Syntax for CREATE SECRET

Syntax for DROP SECRET

CREATE SEQUENCE Statement

The `CREATE SEQUENCE` statement creates a new sequence number generator.

Examples

Generate an ascending sequence starting from 1:

```
CREATE SEQUENCE serial;
```

Generate sequence from a given start number:

```
CREATE SEQUENCE serial START 101;
```

Generate odd numbers using `INCREMENT BY`:

```
CREATE SEQUENCE serial START WITH 1 INCREMENT BY 2;
```

Generate a descending sequence starting from 99:

```
CREATE SEQUENCE serial START WITH 99 INCREMENT BY -1 MAXVALUE 99;
```

By default, cycles are not allowed and will result in error, e.g.:

```
Sequence Error:
nextval: reached maximum value of sequence "serial" (10)
```

```
CREATE SEQUENCE serial START WITH 1 MAXVALUE 10;
```

`CYCLE` allows cycling through the same sequence repeatedly:

```
CREATE SEQUENCE serial START WITH 1 MAXVALUE 10 CYCLE;
```

Creating and Dropping Sequences

Sequences can be created and dropped similarly to other catalogue items.

Overwrite an existing sequence:

```
CREATE OR REPLACE SEQUENCE serial;
```

Only create sequence if no such sequence exists yet:

```
CREATE SEQUENCE IF NOT EXISTS serial;
```

Remove sequence:

```
DROP SEQUENCE serial;
```

Remove sequence if exists:

```
DROP SEQUENCE IF EXISTS serial;
```

Using Sequences for Primary Keys

Sequences can provide an integer primary key for a table. For example:

```
CREATE SEQUENCE id_sequence START 1;
CREATE TABLE tbl (id INTEGER DEFAULT nextval('id_sequence'), s VARCHAR);
INSERT INTO tbl (s) VALUES ('hello'), ('world');
SELECT * FROM tbl;
```

The script results in the following table:

id	s
1	hello
2	world

Sequences can also be added using the `ALTER TABLE` statement. The following example adds an `id` column and fills it with values generated by the sequence:

```
CREATE TABLE tbl (s VARCHAR);
INSERT INTO tbl VALUES ('hello'), ('world');
CREATE SEQUENCE id_sequence START 1;
ALTER TABLE tbl ADD COLUMN id INTEGER DEFAULT nextval('id_sequence');
SELECT * FROM tbl;
```

This script results in the same table as the previous example.

Selecting the Next Value

To select the next number from a sequence, use `nextval`:

```
CREATE SEQUENCE serial START 1;
SELECT nextval('serial') AS nextval;
```

nextval
1

Using this sequence in an `INSERT` command:

```
INSERT INTO distributors VALUES (nextval('serial'), 'nothing');
```

Selecting the Current Value

You may also view the current number from the sequence. Note that the `nextval` function must have already been called before calling `currval`, otherwise a Serialization Error (`sequence is not yet defined in this session`) will be thrown.

```
CREATE SEQUENCE serial START 1;
SELECT nextval('serial') AS nextval;
SELECT currval('serial') AS currval;
```

currval
1

Syntax

`CREATE SEQUENCE` creates a new sequence number generator.

If a schema name is given then the sequence is created in the specified schema. Otherwise it is created in the current schema. Temporary sequences exist in a special schema, so a schema name may not be given when creating a temporary sequence. The sequence name must be distinct from the name of any other sequence in the same schema.

After a sequence is created, you use the function `nextval` to operate on the sequence.

Parameters

Name	Description
`CYCLE` or `NO CYCLE`	The `CYCLE` option allows the sequence to wrap around when the `maxvalue` or `minvalue` has been reached by an ascending or descending sequence respectively. If the limit is reached, the next number generated will be the `minvalue` or `maxvalue`, respectively. If `NO CYCLE` is specified, any calls to `nextval` after the sequence has reached its maximum value will return an error. If neither `CYCLE` or `NO CYCLE` are specified, `NO CYCLE` is the default.
`increment`	The optional clause `INCREMENT BY increment` specifies which value is added to the current sequence value to create a new value. A positive value will make an ascending sequence, a negative one a descending sequence. The default value is 1.
`maxvalue`	The optional clause `MAXVALUE maxvalue` determines the maximum value for the sequence. If this clause is not supplied or `NO MAXVALUE` is specified, then default values will be used. The defaults are 2^63 - 1 and -1 for ascending and descending sequences, respectively.
`minvalue`	The optional clause `MINVALUE minvalue` determines the minimum value a sequence can generate. If this clause is not supplied or `NO MINVALUE` is specified, then defaults will be used. The defaults are 1 and -(2^63 - 1) for ascending and descending sequences, respectively.
`name`	The name (optionally schema-qualified) of the sequence to be created.
`start`	The optional clause `START WITH start` allows the sequence to begin anywhere. The default starting value is `minvalue` for ascending sequences and `maxvalue` for descending ones.
`TEMPORARY` or `TEMP`	If specified, the sequence object is created only for this session, and is automatically dropped on session exit. Existing permanent sequences with the same name are not visible (in this session) while the temporary sequence exists, unless they are referenced with schema-qualified names.

> Sequences are based on `BIGINT` arithmetic, so the range cannot exceed the range of an eight-byte integer (-9223372036854775808 to 9223372036854775807).

Limitations

Due to limitations in DuckDB's dependency manager, `DROP SEQUENCE` will fail in some corner cases. For example, deleting a column that uses a sequence should allow the sequence to be dropped but this currently returns an error:

```
CREATE SEQUENCE id_sequence START 1;
CREATE TABLE tbl (id INTEGER DEFAULT nextval('id_sequence'), s VARCHAR);

ALTER TABLE tbl DROP COLUMN id;
DROP SEQUENCE id_sequence;
```

```
Dependency Error:
Cannot drop entry "id_sequence" because there are entries that depend on it.
table "tbl" depends on index "id_sequence".
Use DROP...CASCADE to drop all dependents.
```

This can be worked around by using the `CASCADE` modifier. The following command drops the sequence:

```
DROP SEQUENCE id_sequence CASCADE;
```

CREATE TABLE Statement

The `CREATE TABLE` statement creates a table in the catalog.

Examples

Create a table with two integer columns (`i` and `j`):

```
CREATE TABLE t1 (i INTEGER, j INTEGER);
```

Create a table with a primary key:

```
CREATE TABLE t1 (id INTEGER PRIMARY KEY, j VARCHAR);
```

Create a table with a composite primary key:

```
CREATE TABLE t1 (id INTEGER, j VARCHAR, PRIMARY KEY (id, j));
```

Create a table with various different types and constraints:

```
CREATE TABLE t1 (
    i INTEGER NOT NULL,
    decimalnr DOUBLE CHECK (decimalnr < 10),
    date DATE UNIQUE,
    time TIMESTAMP
);
```

Create table with `CREATE TABLE ... AS SELECT` (CTAS):

```
CREATE TABLE t1 AS
    SELECT 42 AS i, 84 AS j;
```

Create a table from a CSV file (automatically detecting column names and types):

```
CREATE TABLE t1 AS
    SELECT *
    FROM read_csv('path/file.csv');
```

We can use the FROM-first syntax to omit `SELECT *`:

```
CREATE TABLE t1 AS
    FROM read_csv('path/file.csv');
```

Copy the schema of `t2` to `t1`:

```
CREATE TABLE t1 AS
    FROM t2
    LIMIT 0;
```

Note that only the column names and types are copied to `t2`, other pieces of information (indexes, constraints, default values, etc.) are not copied.

Temporary Tables

Temporary tables can be created using the `CREATE TEMP TABLE` or the `CREATE TEMPORARY TABLE` statement (see diagram below). Temporary tables are session scoped (similar to PostgreSQL for example), meaning that only the specific connection that created them can access them, and once the connection to DuckDB is closed they will be automatically dropped. Temporary tables reside in memory rather than on disk (even when connecting to a persistent DuckDB), but if the `temp_directory` configuration is set when connecting or with a `SET` command, data will be spilled to disk if memory becomes constrained.

Create a temporary table from a CSV file (automatically detecting column names and types):

```
CREATE TEMP TABLE t1 AS
    SELECT *
    FROM read_csv('path/file.csv');
```

Allow temporary tables to off-load excess memory to disk:

```
SET temp_directory = '/path/to/directory/';
```

Temporary tables are part of the `temp.main` schema. While discouraged, their names can overlap with the names of the regular database tables. In these cases, use their fully qualified name, e.g., `temp.main.t1`, for disambiguation.

CREATE OR REPLACE

The `CREATE OR REPLACE` syntax allows a new table to be created or for an existing table to be overwritten by the new table. This is shorthand for dropping the existing table and then creating the new one.

Create a table with two integer columns (i and j) even if t1 already exists:

```
CREATE OR REPLACE TABLE t1 (i INTEGER, j INTEGER);
```

IF NOT EXISTS

The `IF NOT EXISTS` syntax will only proceed with the creation of the table if it does not already exist. If the table already exists, no action will be taken and the existing table will remain in the database.

Create a table with two integer columns (`i` and `j`) only if `t1` does not exist yet:

```
CREATE TABLE IF NOT EXISTS t1 (i INTEGER, j INTEGER);
```

CREATE TABLE ... AS SELECT (CTAS)

DuckDB supports the `CREATE TABLE ... AS SELECT` syntax, also known as "CTAS":

```
CREATE TABLE nums AS
    SELECT i
    FROM range(0, 3) t(i);
```

This syntax can be used in combination with the CSV reader, the shorthand to read directly from CSV files without specifying a function, the FROM-first syntax, and the HTTP(S) support, yielding concise SQL commands such as the following:

```
CREATE TABLE flights AS
    FROM 'https://duckdb.org/data/flights.csv';
```

The CTAS construct also works with the `OR REPLACE` modifier, yielding `CREATE OR REPLACE TABLE ... AS` statements:

```
CREATE OR REPLACE TABLE flights AS
    FROM 'https://duckdb.org/data/flights.csv';
```

Copying the Schema

You can create a copy of the table's schema (column names and types only) as follows:

```
CREATE TABLE t1 AS
    FROM t2
    WITH NO DATA;
```

Or:

```
CREATE TABLE t1 AS
    FROM t2
    LIMIT 0;
```

It is not possible to create tables using CTAS statements with constraints (primary keys, check constraints, etc.).

Check Constraints

A CHECK constraint is an expression that must be satisfied by the values of every row in the table.

```
CREATE TABLE t1 (
    id INTEGER PRIMARY KEY,
    percentage INTEGER CHECK (0 <= percentage AND percentage <= 100)
);
INSERT INTO t1 VALUES (1, 5);
INSERT INTO t1 VALUES (2, -1);
```

```
Constraint Error:
CHECK constraint failed: t1
```

```
INSERT INTO t1 VALUES (3, 101);
```

```
Constraint Error:
CHECK constraint failed: t1
```

```
CREATE TABLE t2 (id INTEGER PRIMARY KEY, x INTEGER, y INTEGER CHECK (x < y));
INSERT INTO t2 VALUES (1, 5, 10);
INSERT INTO t2 VALUES (2, 5, 3);
```

```
Constraint Error:
CHECK constraint failed: t2
```

CHECK constraints can also be added as part of the CONSTRAINTS clause:

```
CREATE TABLE t3 (
    id INTEGER PRIMARY KEY,
    x INTEGER,
    y INTEGER,
    CONSTRAINT x_smaller_than_y CHECK (x < y)
);
INSERT INTO t3 VALUES (1, 5, 10);
INSERT INTO t3 VALUES (2, 5, 3);
```

```
Constraint Error:
CHECK constraint failed: t3
```

Foreign Key Constraints

A FOREIGN KEY is a column (or set of columns) that references another table's primary key. Foreign keys check referential integrity, i.e., the referred primary key must exist in the other table upon insertion.

```
CREATE TABLE t1 (id INTEGER PRIMARY KEY, j VARCHAR);
CREATE TABLE t2 (
    id INTEGER PRIMARY KEY,
    t1_id INTEGER,
    FOREIGN KEY (t1_id) REFERENCES t1 (id)
);
```

Example:

```
INSERT INTO t1 VALUES (1, 'a');
INSERT INTO t2 VALUES (1, 1);
INSERT INTO t2 VALUES (2, 2);
```

```
Constraint Error:
Violates foreign key constraint because key "id: 2" does not exist in the referenced table
```

Foreign keys can be defined on composite primary keys:

```
CREATE TABLE t3 (id INTEGER, j VARCHAR, PRIMARY KEY (id, j));
CREATE TABLE t4 (
    id INTEGER PRIMARY KEY, t3_id INTEGER, t3_j VARCHAR,
    FOREIGN KEY (t3_id, t3_j) REFERENCES t3(id, j)
);
```

Example:

```
INSERT INTO t3 VALUES (1, 'a');
INSERT INTO t4 VALUES (1, 1, 'a');
INSERT INTO t4 VALUES (2, 1, 'b');
```

```
Constraint Error:
Violates foreign key constraint because key "id: 1, j: b" does not exist in the referenced table
```

Foreign keys can also be defined on unique columns:

```
CREATE TABLE t5 (id INTEGER UNIQUE, j VARCHAR);
CREATE TABLE t6 (
    id INTEGER PRIMARY KEY,
    t5_id INTEGER,
    FOREIGN KEY (t5_id) REFERENCES t5(id)
);
```

Limitations

Foreign keys have the following limitations.

Foreign keys with cascading deletes (`FOREIGN KEY ... REFERENCES ... ON DELETE CASCADE`) are not supported.

Inserting into tables with self-referencing foreign keys is currently not supported and will result in the following error:

```
Constraint Error:
Violates foreign key constraint because key "..." does not exist in the referenced table.
```

Generated Columns

The `[type] [GENERATED ALWAYS] AS (expr) [VIRTUAL|STORED]` syntax will create a generated column. The data in this kind of column is generated from its expression, which can reference other (regular or generated) columns of the table. Since they are produced by calculations, these columns can not be inserted into directly.

DuckDB can infer the type of the generated column based on the expression's return type. This allows you to leave out the type when declaring a generated column. It is possible to explicitly set a type, but insertions into the referenced columns might fail if the type can not be cast to the type of the generated column.

Generated columns come in two varieties: `VIRTUAL` and `STORED`. The data of virtual generated columns is not stored on disk, instead it is computed from the expression every time the column is referenced (through a select statement).

The data of stored generated columns is stored on disk and is computed every time the data of their dependencies change (through an `INSERT` / `UPDATE` / `DROP` statement).

Currently, only the `VIRTUAL` kind is supported, and it is also the default option if the last field is left blank.

The simplest syntax for a generated column:

The type is derived from the expression, and the variant defaults to `VIRTUAL`:

```
CREATE TABLE t1 (x FLOAT, two_x AS (2 * x));
```

Fully specifying the same generated column for completeness:

```
CREATE TABLE t1 (x FLOAT, two_x FLOAT GENERATED ALWAYS AS (2 * x) VIRTUAL);
```

Syntax

CREATE VIEW Statement

The `CREATE VIEW` statement defines a new view in the catalog.

Examples

Create a simple view:

```
CREATE VIEW v1 AS SELECT * FROM tbl;
```

Create a view or replace it if a view with that name already exists:

```
CREATE OR REPLACE VIEW v1 AS SELECT 42;
```

Create a view and replace the column names:

```
CREATE VIEW v1(a) AS SELECT 42;
```

The SQL query behind an existing view can be read using the duckdb_views() function like this:

```
SELECT sql FROM duckdb_views() WHERE view_name = 'v1';
```

Syntax

CREATE VIEW defines a view of a query. The view is not physically materialized. Instead, the query is run every time the view is referenced in a query.

CREATE OR REPLACE VIEW is similar, but if a view of the same name already exists, it is replaced.

If a schema name is given then the view is created in the specified schema. Otherwise, it is created in the current schema. Temporary views exist in a special schema, so a schema name cannot be given when creating a temporary view. The name of the view must be distinct from the name of any other view or table in the same schema.

CREATE TYPE Statement

The CREATE TYPE statement defines a new type in the catalog.

Examples

Create a simple ENUM type:

```
CREATE TYPE mood AS ENUM ('happy', 'sad', 'curious');
```

Create a simple STRUCT type:

```
CREATE TYPE many_things AS STRUCT(k INTEGER, l VARCHAR);
```

Create a simple UNION type:

```
CREATE TYPE one_thing AS UNION(number INTEGER, string VARCHAR);
```

Create a type alias:

```
CREATE TYPE x_index AS INTEGER;
```

Syntax

The CREATE TYPE clause defines a new data type available to this DuckDB instance. These new types can then be inspected in the duckdb_types table.

Limitations

- Extending types to support custom operators (such as the PostgreSQL && operator) is not possible via plain SQL. Instead, it requires adding additional C++ code. To do this, create an extension.
- The CREATE TYPE clause does not support the OR REPLACE modifier.

DELETE Statement

The DELETE statement removes rows from the table identified by the table-name. If the WHERE clause is not present, all records in the table are deleted. If a WHERE clause is supplied, then only those rows for which the WHERE clause results in true are deleted. Rows for which the expression is false or NULL are retained.

Examples

Remove the rows matching the condition `i = 2` from the database:

```
DELETE FROM tbl WHERE i = 2;
```

Delete all rows in the table `tbl`:

```
DELETE FROM tbl;
```

USING Clause

The USING clause allows deleting based on the content of other tables or subqueries.

RETURNING Clause

The RETURNING clause allows returning the deletes values. It uses the same syntax as the SELECT clause except the DISTINCT modifier is not supported.

```
CREATE TABLE employees(name VARCHAR, age INTEGER);
INSERT INTO employees VALUES ('Kat', 32);
DELETE FROM employees RETURNING name, 2025 - age AS approx_birthyear;
```

name	approx_birthyear
Kat	1993

Syntax

TheTRUNCATE Statement

The TRUNCATE statement removes all rows from a table, acting as an alias for DELETE FROM without a WHERE clause:

```
TRUNCATE tbl;
```

Limitations on Reclaiming Memory and Disk Space

Running DELETE does not mean space is reclaimed. In general, rows are only marked as deleted. DuckDB reclaims space upon performing a CHECKPOINT. VACUUM currently does not reclaim space.

DESCRIBE Statement

The DESCRIBE statement shows the schema of a table, view or query.

Usage

```
DESCRIBE tbl;
```

In order to summarize a query, prepend DESCRIBE to a query.

```
DESCRIBE SELECT * FROM tbl;
```

Alias

The SHOW statement is an alias for DESCRIBE.

See Also

For more examples, see the guide on DESCRIBE.

DROP Statement

The DROP statement removes a catalog entry added previously with the CREATE command.

Examples

Delete the table with the name tbl:

```
DROP TABLE tbl;
```

Drop the view with the name v1; do not throw an error if the view does not exist:

```
DROP VIEW IF EXISTS v1;
```

Drop function fn:

```
DROP FUNCTION fn;
```

Drop index idx:

```
DROP INDEX idx;
```

Drop schema sch:

```
DROP SCHEMA sch;
```

Drop sequence seq:

```
DROP SEQUENCE seq;
```

Drop macro mcr:

```
DROP MACRO mcr;
```

Drop macro table mt:

```
DROP MACRO TABLE mt;
```

Drop type typ:

```
DROP TYPE typ;
```

Syntax

Dependencies of Dropped Objects

DuckDB performs limited dependency tracking for some object types. By default or if the `RESTRICT` clause is provided, the entry will not be dropped if there are any other objects that depend on it. If the `CASCADE` clause is provided then all the objects that are dependent on the object will be dropped as well.

```
CREATE SCHEMA myschema;
CREATE TABLE myschema.t1 (i INTEGER);
DROP SCHEMA myschema;
```

```
Dependency Error:
Cannot drop entry "myschema" because there are entries that depend on it.
table "t1" depends on schema "myschema".
Use DROP...CASCADE to drop all dependents.
```

The `CASCADE` modifier drops both myschema and `myschema.t1`:

```
CREATE SCHEMA myschema;
CREATE TABLE myschema.t1 (i INTEGER);
DROP SCHEMA myschema CASCADE;
```

The following dependencies are tracked and thus will raise an error if the user tries to drop the depending object without the `CASCADE` modifier.

Depending object type	Dependant object type
SCHEMA	FUNCTION
SCHEMA	INDEX
SCHEMA	MACRO TABLE
SCHEMA	MACRO
SCHEMA	SCHEMA
SCHEMA	SEQUENCE
SCHEMA	TABLE
SCHEMA	TYPE
SCHEMA	VIEW
TABLE	INDEX

Limitations

Dependencies on Views

Currently, dependencies are not tracked for views. For example, if a view is created that references a table and the table is dropped, then the view will be in an invalid state:

```
CREATE TABLE tbl (i INTEGER);
CREATE VIEW v AS
    SELECT i FROM tbl;
DROP TABLE tbl RESTRICT;
SELECT * FROM v;
```

```
Catalog Error:
Table with name tbl does not exist!
```

Limitations on Reclaiming Disk Space

Running `DROP TABLE` should free the memory used by the table, but not always disk space. Even if disk space does not decrease, the free blocks will be marked as `free`. For example, if we have a 2 GB file and we drop a 1 GB table, the file might still be 2 GB, but it should have 1 GB of free blocks in it. To check this, use the following PRAGMA and check the number of `free_blocks` in the output:

```
PRAGMA database_size;
```

For instruction on reclaiming space after dropping a table, refer to the "Reclaiming space" page.

EXPORT and IMPORT DATABASE Statements

The `EXPORT DATABASE` command allows you to export the contents of the database to a specific directory. The `IMPORT DATABASE` command allows you to then read the contents again.

Examples

Export the database to the target directory 'target_directory' as CSV files:

```
EXPORT DATABASE 'target_directory';
```

Export to directory 'target_directory', using the given options for the CSV serialization:

```
EXPORT DATABASE 'target_directory' (FORMAT csv, DELIMITER '|');
```

Export to directory 'target_directory', tables serialized as Parquet:

```
EXPORT DATABASE 'target_directory' (FORMAT parquet);
```

Export to directory 'target_directory', tables serialized as Parquet, compressed with ZSTD, with a row_group_size of 100,000:

```
EXPORT DATABASE 'target_directory' (
    FORMAT parquet,
    COMPRESSION zstd,
    ROW_GROUP_SIZE 100_000
);
```

Reload the database again:

```
IMPORT DATABASE 'source_directory';
```

Alternatively, use a PRAGMA:

```
PRAGMA import_database('source_directory');
```

For details regarding the writing of Parquet files, see the Parquet Files page in the Data Import section and the COPY Statement page.

EXPORT DATABASE

The `EXPORT DATABASE` command exports the full contents of the database – including schema information, tables, views and sequences – to a specific directory that can then be loaded again. The created directory will be structured as follows:

```
target_directory/schema.sql
target_directory/load.sql
target_directory/t_1.csv
...
target_directory/t_n.csv
```

The `schema.sql` file contains the schema statements that are found in the database. It contains any `CREATE SCHEMA`, `CREATE TABLE`, `CREATE VIEW` and `CREATE SEQUENCE` commands that are necessary to re-construct the database.

The `load.sql` file contains a set of `COPY` statements that can be used to read the data from the CSV files again. The file contains a single `COPY` statement for every table found in the schema.

Syntax

`IMPORT DATABASE`

The database can be reloaded by using the `IMPORT DATABASE` command again, or manually by running `schema.sql` followed by `load.sql` to re-load the data.

Syntax

INSERT Statement

The `INSERT` statement inserts new data into a table.

Examples

Insert the values 1, 2, 3 into `tbl`:

```
INSERT INTO tbl
    VALUES (1), (2), (3);
```

Insert the result of a query into a table:

```
INSERT INTO tbl
    SELECT * FROM other_tbl;
```

Insert values into the `i` column, inserting the default value into other columns:

```
INSERT INTO tbl (i)
    VALUES (1), (2), (3);
```

Explicitly insert the default value into a column:

```
INSERT INTO tbl (i)
    VALUES (1), (DEFAULT), (3);
```

Assuming `tbl` has a primary key/unique constraint, do nothing on conflict:

```
INSERT OR IGNORE INTO tbl (i)
    VALUES (1);
```

Or update the table with the new values instead:

```
INSERT OR REPLACE INTO tbl (i)
    VALUES (1);
```

Syntax

`INSERT INTO` inserts new rows into a table. One can insert one or more rows specified by value expressions, or zero or more rows resulting from a query.

Insert Column Order

It's possible to provide an optional insert column order, this can either be `BY POSITION` (the default) or `BY NAME`. Each column not present in the explicit or implicit column list will be filled with a default value, either its declared default value or `NULL` if there is none.

If the expression for any column is not of the correct data type, automatic type conversion will be attempted.

`INSERT INTO ... [BY POSITION]`

The order that values are inserted into the columns of the table is determined by the order that the columns were declared in. That is, the values supplied by the `VALUES` clause or query are associated with the column list left-to-right. This is the default option, that can be explicitly specified using the `BY POSITION` option. For example:

```
CREATE TABLE tbl (a INTEGER, b INTEGER);
INSERT INTO tbl
    VALUES (5, 42);
```

Specifying `BY POSITION` is optional and is equivalent to the default behavior:

```
INSERT INTO tbl
    BY POSITION
    VALUES (5, 42);
```

To use a different order, column names can be provided as part of the target, for example:

```
CREATE TABLE tbl (a INTEGER, b INTEGER);
INSERT INTO tbl (b, a)
    VALUES (5, 42);
```

Adding `BY POSITION` results in the same behavior:

```
INSERT INTO tbl
    BY POSITION (b, a)
    VALUES (5, 42);
```

This will insert 5 into b and 42 into a.

`INSERT INTO ... BY NAME`

Using the `BY NAME` modifier, the names of the column list of the `SELECT` statement are matched against the column names of the table to determine the order that values should be inserted into the table. This allows inserting even in cases when the order of the columns in the table differs from the order of the values in the `SELECT` statement or certain columns are missing.

For example:

```
CREATE TABLE tbl (a INTEGER, b INTEGER);
INSERT INTO tbl BY NAME (SELECT 42 AS b, 32 AS a);
INSERT INTO tbl BY NAME (SELECT 22 AS b);
SELECT * FROM tbl;
```

a	b
32	42
NULL	22

It's important to note that when using `INSERT INTO ... BY NAME`, the column names specified in the `SELECT` statement must match the column names in the table. If a column name is misspelled or does not exist in the table, an error will occur. Columns that are missing from the `SELECT` statement will be filled with the default value.

ON CONFLICT Clause

An ON CONFLICT clause can be used to perform a certain action on conflicts that arise from UNIQUE or PRIMARY KEY constraints. An example for such a conflict is shown in the following example:

```
CREATE TABLE tbl (i INTEGER PRIMARY KEY, j INTEGER);
INSERT INTO tbl
    VALUES (1, 42);
INSERT INTO tbl
    VALUES (1, 84);
```

This raises as an error:

```
Constraint Error:
Duplicate key "i: 1" violates primary key constraint.
```

The table will contain the row that was first inserted:

```
SELECT * FROM tbl;
```

i	j
1	42

These error messages can be avoided by explicitly handling conflicts. DuckDB supports two such clauses: ON CONFLICT DO NOTHING and ON CONFLICT DO UPDATE SET ....

DO NOTHING Clause

The DO NOTHING clause causes the error(s) to be ignored, and the values are not inserted or updated. For example:

```
CREATE TABLE tbl (i INTEGER PRIMARY KEY, j INTEGER);
INSERT INTO tbl
    VALUES (1, 42);
INSERT INTO tbl
    VALUES (1, 84)
    ON CONFLICT DO NOTHING;
```

These statements finish successfully and leaves the table with the row <i: 1, j: 42>.

INSERT OR IGNORE INTO

The INSERT OR IGNORE INTO ... statement is a shorter syntax alternative to INSERT INTO ... ON CONFLICT DO NOTHING. For example, the following statements are equivalent:

```
INSERT OR IGNORE INTO tbl
    VALUES (1, 84);
INSERT INTO tbl
    VALUES (1, 84) ON CONFLICT DO NOTHING;
```

DO UPDATE Clause (Upsert)

The DO UPDATE clause causes the INSERT to turn into an UPDATE on the conflicting row(s) instead. The SET expressions that follow determine how these rows are updated. The expressions can use the special virtual table EXCLUDED, which contains the conflicting values for the row. Optionally you can provide an additional WHERE clause that can exclude certain rows from the update. The conflicts that don't meet this condition are ignored instead.

Because we need a way to refer to both the **to-be-inserted** tuple and the **existing** tuple, we introduce the special `EXCLUDED` qualifier. When the `EXCLUDED` qualifier is provided, the reference refers to the **to-be-inserted** tuple, otherwise, it refers to the **existing** tuple. This special qualifier can be used within the `WHERE` clauses and `SET` expressions of the `ON CONFLICT` clause.

```
CREATE TABLE tbl (i INTEGER PRIMARY KEY, j INTEGER);
INSERT INTO tbl VALUES (1, 42);
INSERT INTO tbl VALUES (1, 52), (1, 62) ON CONFLICT DO UPDATE SET j = EXCLUDED.j;
```

Examples

An example using `DO UPDATE` is the following:

```
CREATE TABLE tbl (i INTEGER PRIMARY KEY, j INTEGER);
INSERT INTO tbl
    VALUES (1, 42);
INSERT INTO tbl
    VALUES (1, 84)
    ON CONFLICT DO UPDATE SET j = EXCLUDED.j;
SELECT * FROM tbl;
```

i	j
1	84

Rearranging columns and using `BY NAME` is also possible:

```
CREATE TABLE tbl (i INTEGER PRIMARY KEY, j INTEGER);
INSERT INTO tbl
    VALUES (1, 42);
INSERT INTO tbl (j, i)
    VALUES (168, 1)
    ON CONFLICT DO UPDATE SET j = EXCLUDED.j;
INSERT INTO tbl
    BY NAME (SELECT 1 AS i, 336 AS j)
    ON CONFLICT DO UPDATE SET j = EXCLUDED.j;
SELECT * FROM tbl;
```

i	j
1	336

`INSERT OR REPLACE INTO`

The `INSERT OR REPLACE INTO ...` statement is a shorter syntax alternative to `INSERT INTO ... DO UPDATE SET c1 = EXCLUDED.c1, c2 = EXCLUDED.c2, ...`. That is, it updates every column of the **existing** row to the new values of the **to-be-inserted** row. For example, given the following input table:

```
CREATE TABLE tbl (i INTEGER PRIMARY KEY, j INTEGER);
INSERT INTO tbl
    VALUES (1, 42);
```

These statements are equivalent:

```
INSERT OR REPLACE INTO tbl
    VALUES (1, 84);
INSERT INTO tbl
```

```
    VALUES (1, 84)
    ON CONFLICT DO UPDATE SET j = EXCLUDED.j;
INSERT INTO tbl (j, i)
    VALUES (84, 1)
    ON CONFLICT DO UPDATE SET j = EXCLUDED.j;
INSERT INTO tbl BY NAME
    (SELECT 84 AS j, 1 AS i)
    ON CONFLICT DO UPDATE SET j = EXCLUDED.j;
```

Limitations

When the `ON CONFLICT ... DO UPDATE` clause is used and a conflict occurs, DuckDB internally assigns `NULL` values to the row's columns that are unaffected by the conflict, then re-assigns their values. If the affected columns use a `NOT NULL` constraint, this will trigger a `NOT NULL constraint failed` error. For example:

```
CREATE TABLE t1 (id INTEGER PRIMARY KEY, val1 DOUBLE, val2 DOUBLE NOT NULL);
CREATE TABLE t2 (id INTEGER PRIMARY KEY, val1 DOUBLE);
INSERT INTO t1
    VALUES (1, 2, 3);
INSERT INTO t2
    VALUES (1, 5);

INSERT INTO t1 BY NAME (SELECT id, val1 FROM t2)
    ON CONFLICT DO UPDATE
    SET val1 = EXCLUDED.val1;
```

This fails with the following error:

```
Constraint Error:
NOT NULL constraint failed: t1.val2
```

Composite Primary Key

When multiple columns need to be part of the uniqueness constraint, use a single `PRIMARY KEY` clause including all relevant columns:

```
CREATE TABLE t1 (id1 INTEGER, id2 INTEGER, val1 DOUBLE, PRIMARY KEY(id1, id2));
INSERT OR REPLACE INTO t1
    VALUES (1, 2, 3);
INSERT OR REPLACE INTO t1
    VALUES (1, 2, 4);
```

Defining a Conflict Target

A conflict target may be provided as `ON CONFLICT (conflict_target)`. This is a group of columns that an index or uniqueness/key constraint is defined on. If the conflict target is omitted, or `PRIMARY KEY` constraint(s) on the table are targeted.

Specifying a conflict target is optional unless using a `DO UPDATE` and there are multiple unique/primary key constraints on the table.

```
CREATE TABLE tbl (i INTEGER PRIMARY KEY, j INTEGER UNIQUE, k INTEGER);
INSERT INTO tbl
    VALUES (1, 20, 300);
SELECT * FROM tbl;
```

i	j	k
1	20	300

```
INSERT INTO tbl
    VALUES (1, 40, 700)
    ON CONFLICT (i) DO UPDATE SET k = 2 * EXCLUDED.k;
```

i	j	k
1	20	1400

```
INSERT INTO tbl
    VALUES (1, 20, 900)
    ON CONFLICT (j) DO UPDATE SET k = 5 * EXCLUDED.k;
```

i	j	k
1	20	4500

When a conflict target is provided, you can further filter this with a `WHERE` clause, that should be met by all conflicts.

```
INSERT INTO tbl
    VALUES (1, 40, 700)
    ON CONFLICT (i) DO UPDATE SET k = 2 * EXCLUDED.k WHERE k < 100;
```

`RETURNING` Clause

The `RETURNING` clause may be used to return the contents of the rows that were inserted. This can be useful if some columns are calculated upon insert. For example, if the table contains an automatically incrementing primary key, then the `RETURNING` clause will include the automatically created primary key. This is also useful in the case of generated columns.

Some or all columns can be explicitly chosen to be returned and they may optionally be renamed using aliases. Arbitrary non-aggregating expressions may also be returned instead of simply returning a column. All columns can be returned using the `*` expression, and columns or expressions can be returned in addition to all columns returned by the `*`.

For example:

```
CREATE TABLE t1 (i INTEGER);
INSERT INTO t1
    SELECT 42
    RETURNING *;
```

i
42

A more complex example that includes an expression in the `RETURNING` clause:

```
CREATE TABLE t2 (i INTEGER, j INTEGER);
INSERT INTO t2
    SELECT 2 AS i, 3 AS j
    RETURNING *, i * j AS i_times_j;
```

i	j	i_times_j
2	3	6

The next example shows a situation where the RETURNING clause is more helpful. First, a table is created with a primary key column. Then a sequence is created to allow for that primary key to be incremented as new rows are inserted. When we insert into the table, we do not already know the values generated by the sequence, so it is valuable to return them. For additional information, see the CREATE SEQUENCE page.

```
CREATE TABLE t3 (i INTEGER PRIMARY KEY, j INTEGER);
CREATE SEQUENCE 't3_key';
INSERT INTO t3
    SELECT nextval('t3_key') AS i, 42 AS j
    UNION ALL
    SELECT nextval('t3_key') AS i, 43 AS j
    RETURNING *;
```

i	j
1	42
2	43

LOAD / INSTALL Statements

INSTALL

The INSTALL statement downloads an extension so it can be loaded into a DuckDB session.

Examples

Install the httpfs extension:

```
INSTALL httpfs;
```

Install the h3 community extension:

```
INSTALL h3 FROM community;
```

Syntax

LOAD

The LOAD statement loads an installed DuckDB extension into the current session.

Examples

Load the httpfs extension:

```
LOAD httpfs;
```

Load the spatial extension:

```
LOAD spatial;
```

Syntax

PIVOT Statement

The PIVOT statement allows distinct values within a column to be separated into their own columns. The values within those new columns are calculated using an aggregate function on the subset of rows that match each distinct value.

DuckDB implements both the SQL Standard PIVOT syntax and a simplified PIVOT syntax that automatically detects the columns to create while pivoting. PIVOT_WIDER may also be used in place of the PIVOT keyword.

For details on how the PIVOT statement is implemented, see the Pivot Internals site.

> The UNPIVOT statement is the inverse of the PIVOT statement.

Simplified PIVOT Syntax

The full syntax diagram is below, but the simplified PIVOT syntax can be summarized using spreadsheet pivot table naming conventions as:

```
PIVOT <dataset>
ON <columns>
USING <values>
GROUP BY <rows>
ORDER BY <columns_with_order_directions>
LIMIT <number_of_rows>;
```

The ON, USING, and GROUP BY clauses are each optional, but they may not all be omitted.

Example Data

All examples use the dataset produced by the queries below:

```
CREATE TABLE cities (
    country VARCHAR, name VARCHAR, year INTEGER, population INTEGER
);
INSERT INTO cities VALUES
    ('NL', 'Amsterdam', 2000, 1005),
    ('NL', 'Amsterdam', 2010, 1065),
    ('NL', 'Amsterdam', 2020, 1158),
    ('US', 'Seattle', 2000, 564),
    ('US', 'Seattle', 2010, 608),
    ('US', 'Seattle', 2020, 738),
    ('US', 'New York City', 2000, 8015),
    ('US', 'New York City', 2010, 8175),
    ('US', 'New York City', 2020, 8772);

SELECT *
FROM cities;
```

country	name	year	population
NL	Amsterdam	2000	1005
NL	Amsterdam	2010	1065
NL	Amsterdam	2020	1158
US	Seattle	2000	564

country	name	year	population
US	Seattle	2010	608
US	Seattle	2020	738
US	New York City	2000	8015
US	New York City	2010	8175
US	New York City	2020	8772

`PIVOT ON` and `USING`

Use the `PIVOT` statement below to create a separate column for each year and calculate the total population in each. The `ON` clause specifies which column(s) to split into separate columns. It is equivalent to the columns parameter in a spreadsheet pivot table.

The `USING` clause determines how to aggregate the values that are split into separate columns. This is equivalent to the values parameter in a spreadsheet pivot table. If the `USING` clause is not included, it defaults to `count(*)`.

```
PIVOT cities
ON year
USING sum(population);
```

country	name	2000	2010	2020
NL	Amsterdam	1005	1065	1158
US	Seattle	564	608	738
US	New York City	8015	8175	8772

In the above example, the `sum` aggregate is always operating on a single value. If we only want to change the orientation of how the data is displayed without aggregating, use the `first` aggregate function. In this example, we are pivoting numeric values, but the `first` function works very well for pivoting out a text column. (This is something that is difficult to do in a spreadsheet pivot table, but easy in DuckDB!)

This query produces a result that is identical to the one above:

```
PIVOT cities
ON year
USING first(population);
```

> **Note.** The SQL syntax permits `FILTER` clauses with aggregate functions in the `USING` clause. In DuckDB, the `PIVOT` statement currently does not support these and they are silently ignored.

`PIVOT ON`, `USING`, and `GROUP BY`

By default, the `PIVOT` statement retains all columns not specified in the `ON` or `USING` clauses. To include only certain columns and further aggregate, specify columns in the `GROUP BY` clause. This is equivalent to the rows parameter of a spreadsheet pivot table.

In the below example, the `name` column is no longer included in the output, and the data is aggregated up to the `country` level.

```
PIVOT cities
ON year
USING sum(population)
GROUP BY country;
```

country	2000	2010	2020
NL	1005	1065	1158
US	8579	8783	9510

`IN` Filter for `ON` Clause

To only create a separate column for specific values within a column in the ON clause, use an optional `IN` expression. Let's say for example that we wanted to forget about the year 2020 for no particular reason...

```
PIVOT cities
ON year IN (2000, 2010)
USING sum(population)
GROUP BY country;
```

country	2000	2010
NL	1005	1065
US	8579	8783

Multiple Expressions per Clause

Multiple columns can be specified in the ON and GROUP BY clauses, and multiple aggregate expressions can be included in the `USING` clause.

Multiple `ON` Columns and `ON` Expressions

Multiple columns can be pivoted out into their own columns. DuckDB will find the distinct values in each ON clause column and create one new column for all combinations of those values (a Cartesian product).

In the below example, all combinations of unique countries and unique cities receive their own column. Some combinations may not be present in the underlying data, so those columns are populated with `NULL` values.

```
PIVOT cities
ON country, name
USING sum(population);
```

year	NL_Amsterdam	NL_New York City	NL_Seattle	US_Amsterdam	US_New York City	US_Seattle
2000	1005	NULL	NULL	NULL	8015	564
2010	1065	NULL	NULL	NULL	8175	608
2020	1158	NULL	NULL	NULL	8772	738

To pivot only the combinations of values that are present in the underlying data, use an expression in the ON clause. Multiple expressions and/or columns may be provided.

Here, `country` and `name` are concatenated together and the resulting concatenations each receive their own column. Any arbitrary non-aggregating expression may be used. In this case, concatenating with an underscore is used to imitate the naming convention the `PIVOT` clause uses when multiple ON columns are provided (like in the prior example).

```
PIVOT cities
ON country || '_' || name
USING sum(population);
```

year	NL_Amsterdam	US_New York City	US_Seattle
2000	1005	8015	564
2010	1065	8175	608
2020	1158	8772	738

Multiple USING Expressions

An alias may also be included for each expression in the USING clause. It will be appended to the generated column names after an underscore (_). This makes the column naming convention much cleaner when multiple expressions are included in the USING clause.

In this example, both the sum and max of the population column are calculated for each year and are split into separate columns.

```
PIVOT cities
ON year
USING sum(population) AS total, max(population) AS max
GROUP BY country;
```

country	2000_total	2000_max	2010_total	2010_max	2020_total	2020_max
US	8579	8015	8783	8175	9510	8772
NL	1005	1005	1065	1065	1158	1158

Multiple GROUP BY Columns

Multiple GROUP BY columns may also be provided. Note that column names must be used rather than column positions (1, 2, etc.), and that expressions are not supported in the GROUP BY clause.

```
PIVOT cities
ON year
USING sum(population)
GROUP BY country, name;
```

country	name	2000	2010	2020
NL	Amsterdam	1005	1065	1158
US	Seattle	564	608	738
US	New York City	8015	8175	8772

Using PIVOT within a SELECT Statement

The PIVOT statement may be included within a SELECT statement as a CTE (a Common Table Expression, or WITH clause), or a subquery. This allows for a PIVOT to be used alongside other SQL logic, as well as for multiple PIVOTs to be used in one query.

No SELECT is needed within the CTE, the PIVOT keyword can be thought of as taking its place.

```
WITH pivot_alias AS (
    PIVOT cities
    ON year
    USING sum(population)
    GROUP BY country
)
SELECT * FROM pivot_alias;
```

A PIVOT may be used in a subquery and must be wrapped in parentheses. Note that this behavior is different than the SQL Standard Pivot, as illustrated in subsequent examples.

```
SELECT *
FROM (
    PIVOT cities
    ON year
    USING sum(population)
    GROUP BY country
) pivot_alias;
```

Multiple PIVOT Statements

Each PIVOT can be treated as if it were a SELECT node, so they can be joined together or manipulated in other ways.

For example, if two PIVOT statements share the same GROUP BY expression, they can be joined together using the columns in the GROUP BY clause into a wider pivot.

```
SELECT *
FROM (PIVOT cities ON year USING sum(population) GROUP BY country) year_pivot
JOIN (PIVOT cities ON name USING sum(population) GROUP BY country) name_pivot
USING (country);
```

country	2000	2010	2020	Amsterdam	New York City	Seattle
NL	1005	1065	1158	3228	NULL	NULL
US	8579	8783	9510	NULL	24962	1910

Simplified PIVOT Full Syntax Diagram

Below is the full syntax diagram of the PIVOT statement.

SQL Standard PIVOT Syntax

The full syntax diagram is below, but the SQL Standard PIVOT syntax can be summarized as:

```
SELECT *
FROM <dataset>
PIVOT (
    <values>
    FOR
        <column_1> IN (<in_list>)
        <column_2> IN (<in_list>)
        ...
    GROUP BY <rows>
);
```

Unlike the simplified syntax, the `IN` clause must be specified for each column to be pivoted. If you are interested in dynamic pivoting, the simplified syntax is recommended.

Note that no commas separate the expressions in the `FOR` clause, but that `value` and `GROUP BY` expressions must be comma-separated!

Examples

This example uses a single value expression, a single column expression, and a single row expression:

```
SELECT *
FROM cities
PIVOT (
    sum(population)
    FOR
        year IN (2000, 2010, 2020)
    GROUP BY country
);
```

country	2000	2010	2020
NL	1005	1065	1158
US	8579	8783	9510

This example is somewhat contrived, but serves as an example of using multiple value expressions and multiple columns in the `FOR` clause.

```
SELECT *
FROM cities
PIVOT (
    sum(population) AS total,
    count(population) AS count
    FOR
        year IN (2000, 2010)
        country IN ('NL', 'US')
);
```

name	2000_NL_total	2000_NL_count	2000_US_total	2000_US_count	2010_NL_total	2010_NL_count	2010_US_total	2010_US_count
Amsterdam	1005	1	NULL	0	1065	1	NULL	0
Seattle	NULL	0	564	1	NULL	0	608	1
New York City	NULL	0	8015	1	NULL	0	8175	1

SQL Standard `PIVOT` Full Syntax Diagram

Below is the full syntax diagram of the SQL Standard version of the `PIVOT` statement.

Limitations

`PIVOT` currently only accepts an aggregate function, expressions are not allowed. For example, the following query attempts to get the population as the number of people instead of thousands of people (i.e., instead of 564, get 564000):

```
PIVOT cities
ON year
USING sum(population) * 1000;
```

However, it fails with the following error:

```
Catalog Error:
* is not an aggregate function
```

To work around this limitation, perform the `PIVOT` with the aggregation only, then use the `COLUMNS` expression:

```
SELECT country, name, 1000 * COLUMNS(* EXCLUDE (country, name))
FROM (
    PIVOT cities
    ON year
    USING sum(population)
);
```

Profiling Queries

DuckDB supports profiling queries via the `EXPLAIN` and `EXPLAIN ANALYZE` statements.

EXPLAIN

To see the query plan of a query without executing it, run:

```
EXPLAIN <query>;
```

The output of `EXPLAIN` contains the estimated cardinalities for each operator.

EXPLAIN ANALYZE

To profile a query, run:

```
EXPLAIN ANALYZE <query>;
```

The `EXPLAIN ANALYZE` statement runs the query, and shows the actual cardinalities for each operator, as well as the cumulative wall-clock time spent in each operator.

SELECT Statement

The `SELECT` statement retrieves rows from the database.

Examples

Select all columns from the table `tbl`:

```
SELECT * FROM tbl;
```

Select the rows from `tbl`:

```
SELECT j FROM tbl WHERE i = 3;
```

Perform an aggregate grouped by the column `i`:

```
SELECT i, sum(j) FROM tbl GROUP BY i;
```

Select only the top 3 rows from the `tbl`:

```
SELECT * FROM tbl ORDER BY i DESC LIMIT 3;
```

Join two tables together using the `USING` clause:

```
SELECT * FROM t1 JOIN t2 USING (a, b);
```

Use column indexes to select the first and third column from the table `tbl`:

```
SELECT #1, #3 FROM tbl;
```

Select all unique cities from the addresses table:

```
SELECT DISTINCT city FROM addresses;
```

Return a STRUCT by using a row variable:

```
SELECT d
FROM (SELECT 1 AS a, 2 AS b) d;
```

Syntax

The `SELECT` statement retrieves rows from the database. The canonical order of a `SELECT` statement is as follows, with less common clauses being indented:

```
SELECT <select_list>
FROM <tables>
    USING SAMPLE <sample_expression>
WHERE <condition>
GROUP BY <groups>
HAVING <group_filter>
    WINDOW <window_expression>
    QUALIFY <qualify_filter>
ORDER BY <order_expression>
LIMIT <n>;
```

Optionally, the `SELECT` statement can be prefixed with a `WITH` clause.

As the `SELECT` statement is so complex, we have split up the syntax diagrams into several parts. The full syntax diagram can be found at the bottom of the page.

SELECT Clause

The `SELECT` clause specifies the list of columns that will be returned by the query. While it appears first in the clause, *logically* the expressions here are executed only at the end. The `SELECT` clause can contain arbitrary expressions that transform the output, as well as aggregates and window functions. The `DISTINCT` keyword ensures that only unique tuples are returned.

> Column names are case-insensitive. See the Rules for Case Sensitivity for more details.

FROM Clause

The FROM clause specifies the *source* of the data on which the remainder of the query should operate. Logically, the FROM clause is where the query starts execution. The FROM clause can contain a single table, a combination of multiple tables that are joined together, or another SELECT query inside a subquery node.

SAMPLE Clause

The SAMPLE clause allows you to run the query on a sample from the base table. This can significantly speed up processing of queries, at the expense of accuracy in the result. Samples can also be used to quickly see a snapshot of the data when exploring a data set. The SAMPLE clause is applied right after anything in the FROM clause (i.e., after any joins, but before the where clause or any aggregates). See the Samples page for more information.

WHERE Clause

The WHERE clause specifies any filters to apply to the data. This allows you to select only a subset of the data in which you are interested. Logically the WHERE clause is applied immediately after the FROM clause.

GROUP BY and HAVING Clauses

The GROUP BY clause specifies which grouping columns should be used to perform any aggregations in the SELECT clause. If the GROUP BY clause is specified, the query is always an aggregate query, even if no aggregations are present in the SELECT clause.

WINDOW Clause

The WINDOW clause allows you to specify named windows that can be used within window functions. These are useful when you have multiple window functions, as they allow you to avoid repeating the same window clause.

QUALIFY Clause

The QUALIFY clause is used to filter the result of WINDOW functions.

ORDER BY, LIMIT and OFFSET Clauses

ORDER BY, LIMIT and OFFSET are output modifiers. Logically they are applied at the very end of the query. The ORDER BY clause sorts the rows on the sorting criteria in either ascending or descending order. The LIMIT clause restricts the amount of rows fetched, while the OFFSET clause indicates at which position to start reading the values.

VALUES List

A VALUES list is a set of values that is supplied instead of a SELECT statement.

Row IDs

For each table, the `rowid` pseudocolumn returns the row identifiers based on the physical storage.

```
CREATE TABLE t (id INTEGER, content VARCHAR);
INSERT INTO t VALUES (42, 'hello'), (43, 'world');
SELECT rowid, id, content FROM t;
```

rowid	id	content
0	42	hello
1	43	world

In the current storage, these identifiers are contiguous unsigned integers (0, 1, ...) if no rows were deleted. Deletions introduce gaps in the rowids which may be reclaimed later:

```
CREATE OR REPLACE TABLE t AS (FROM range(10) r(i));
DELETE FROM t WHERE i % 2 = 0;
SELECT rowid FROM t;
```

rowid
1
3
5
7
9

The `rowid` values are stable within a transaction.

> **Best practice.** It is strongly advised to avoid using rowids as identifiers.

> If there is a user-defined column named `rowid`, it shadows the `rowid` pseudocolumn.

Common Table Expressions

Full Syntax Diagram

Below is the full syntax diagram of the `SELECT` statement:

SET and RESET Statements

The `SET` statement modifies the provided DuckDB configuration option at the specified scope.

Examples

Update the `memory_limit` configuration value:

```
SET memory_limit = '10GB';
```

Configure the system to use `1` thread:

```
SET threads = 1;
```

Or use the `TO` keyword:

```
SET threads TO 1;
```

Change configuration option to default value:

```
RESET threads;
```

Retrieve configuration value:

```
SELECT current_setting('threads');
```

Set the default sort order globally:

```
SET GLOBAL sort_order = 'desc';
```

Set the default collation for the session:

```
SET SESSION default_collation = 'nocase';
```

Syntax

`SET` updates a DuckDB configuration option to the provided value.

RESET

The `RESET` statement changes the given DuckDB configuration option to the default value.

Scopes

Configuration options can have different scopes:

- `GLOBAL`: Configuration value is used (or reset) across the entire DuckDB instance.
- `SESSION`: Configuration value is used (or reset) only for the current session attached to a DuckDB instance.
- `LOCAL`: Not yet implemented.

When not specified, the default scope for the configuration option is used. For most options this is `GLOBAL`.

Configuration

See the Configuration page for the full list of configuration options.

SET VARIABLE and RESET VARIABLE Statements

DuckDB supports the definition of SQL-level variables using the `SET VARIABLE` and `RESET VARIABLE` statements.

SET VARIABLE

The `SET VARIABLE` statement assigns a value to a variable, which can be accessed using the `getvariable` call:

```
SET VARIABLE my_var = 30;
SELECT 20 + getvariable('my_var') AS total;
```

total
50

If `SET VARIABLE` is invoked on an existing variable, it will overwrite its value:

```
SET VARIABLE my_var = 30;
SET VARIABLE my_var = 100;
SELECT 20 + getvariable('my_var') AS total;
```

total
120

Variables can have different types:

```
SET VARIABLE my_date = DATE '2018-07-13';
SET VARIABLE my_string = 'Hello world';
SET VARIABLE my_map = MAP {'k1': 10, 'k2': 20};
```

Variables can also be assigned to results of queries:

```
-- write some CSV files
COPY (SELECT 42 AS a) TO 'test1.csv';
COPY (SELECT 84 AS a) TO 'test2.csv';

-- add a list of CSV files to a table
CREATE TABLE csv_files (file VARCHAR);
INSERT INTO csv_files VALUES ('test1.csv'), ('test2.csv');

-- initialize a variable with the list of csv files
SET VARIABLE list_of_files = (SELECT list(file) FROM csv_files);

-- read the CSV files
SELECT * FROM read_csv(getvariable('list_of_files'), filename := True);
```

a	filename
42	test.csv
84	test2.csv

If a variable is not set, the `getvariable` function returns `NULL`:

```
SELECT getvariable('undefined_var') AS result;
```

result
NULL

The `getvariable` function can also be used in a COLUMNS expression:

```
SET VARIABLE column_to_exclude = 'col1';
CREATE TABLE tbl AS SELECT 12 AS col0, 34 AS col1, 56 AS col2;
SELECT COLUMNS(c -> c != getvariable('column_to_exclude')) FROM tbl;
```

col0	col2
12	56

Syntax

RESET VARIABLE

The `RESET VARIABLE` statement unsets a variable.

```
SET VARIABLE my_var = 30;
RESET VARIABLE my_var;
SELECT getvariable('my_var') AS my_var;
```

my_var
NULL

Syntax

SUMMARIZE Statement

The `SUMMARIZE` statement returns summary statistics for a table, view or a query.

Usage

```
SUMMARIZE tbl;
```

In order to summarize a query, prepend `SUMMARIZE` to a query.

```
SUMMARIZE SELECT * FROM tbl;
```

See Also

For more examples, see the guide on `SUMMARIZE`.

Transaction Management

DuckDB supports ACID database transactions. Transactions provide isolation, i.e., changes made by a transaction are not visible from concurrent transactions until it is committed. A transaction can also be aborted, which discards any changes it made so far.

Statements

DuckDB provides the following statements for transaction management.

Starting a Transaction

To start a transaction, run:

```
BEGIN TRANSACTION;
```

Committing a Transaction

You can commit a transaction to make it visible to other transactions and to write it to persistent storage (if using DuckDB in persistent mode). To commit a transaction, run:

```
COMMIT;
```

If you are not in an active transaction, the `COMMIT` statement will fail.

Rolling Back a Transaction

You can abort a transaction. This operation, also known as rolling back, will discard any changes the transaction made to the database. To abort a transaction, run:

```
ROLLBACK;
```

You can also use the abort command, which has an identical behavior:

```
ABORT;
```

If you are not in an active transaction, the `ROLLBACK` and `ABORT` statements will fail.

Example

We illustrate the use of transactions through a simple example.

```
CREATE TABLE person (name VARCHAR, age BIGINT);

BEGIN TRANSACTION;
INSERT INTO person VALUES ('Ada', 52);
COMMIT;

BEGIN TRANSACTION;
DELETE FROM person WHERE name = 'Ada';
INSERT INTO person VALUES ('Bruce', 39);
ROLLBACK;

SELECT * FROM person;
```

The first transaction (inserting "Ada") was committed but the second (deleting "Ada" and inserting "Bruce") was aborted. Therefore, the resulting table will only contain `<'Ada', 52>`.

UNPIVOT Statement

The UNPIVOT statement allows multiple columns to be stacked into fewer columns. In the basic case, multiple columns are stacked into two columns: a NAME column (which contains the name of the source column) and a VALUE column (which contains the value from the source column).

DuckDB implements both the SQL Standard UNPIVOT syntax and a simplified UNPIVOT syntax. Both can utilize a COLUMNS expression to automatically detect the columns to unpivot. PIVOT_LONGER may also be used in place of the UNPIVOT keyword.

For details on how the UNPIVOT statement is implemented, see the Pivot Internals site.

> The PIVOT statement is the inverse of the UNPIVOT statement.

Simplified UNPIVOT Syntax

The full syntax diagram is below, but the simplified UNPIVOT syntax can be summarized using spreadsheet pivot table naming conventions as:

```
UNPIVOT <dataset>
ON <column(s)>
INTO
    NAME <name_column_name>
    VALUE <value_column_name(s)>
ORDER BY <column(s)_with_order_direction(s)>
LIMIT <number_of_rows>;
```

Example Data

All examples use the dataset produced by the queries below:

```
CREATE OR REPLACE TABLE monthly_sales
    (empid INTEGER, dept TEXT, Jan INTEGER, Feb INTEGER, Mar INTEGER, Apr INTEGER, May INTEGER, Jun
INTEGER);
INSERT INTO monthly_sales VALUES
    (1, 'electronics', 1, 2, 3, 4, 5, 6),
    (2, 'clothes', 10, 20, 30, 40, 50, 60),
    (3, 'cars', 100, 200, 300, 400, 500, 600);

FROM monthly_sales;
```

empid	dept	Jan	Feb	Mar	Apr	May	Jun
1	electronics	1	2	3	4	5	6
2	clothes	10	20	30	40	50	60
3	cars	100	200	300	400	500	600

UNPIVOT Manually

The most typical UNPIVOT transformation is to take already pivoted data and re-stack it into a column each for the name and value. In this case, all months will be stacked into a `month` column and a `sales` column.

```
UNPIVOT monthly_sales
ON jan, feb, mar, apr, may, jun
INTO
    NAME month
    VALUE sales;
```

empid	dept	month	sales
1	electronics	Jan	1
1	electronics	Feb	2
1	electronics	Mar	3
1	electronics	Apr	4
1	electronics	May	5
1	electronics	Jun	6
2	clothes	Jan	10
2	clothes	Feb	20
2	clothes	Mar	30
2	clothes	Apr	40
2	clothes	May	50
2	clothes	Jun	60
3	cars	Jan	100
3	cars	Feb	200
3	cars	Mar	300
3	cars	Apr	400
3	cars	May	500
3	cars	Jun	600

`UNPIVOT` Dynamically Using Columns Expression

In many cases, the number of columns to unpivot is not easy to predetermine ahead of time. In the case of this dataset, the query above would have to change each time a new month is added. The COLUMNS expression can be used to select all columns that are not `empid` or `dept`. This enables dynamic unpivoting that will work regardless of how many months are added. The query below returns identical results to the one above.

```
UNPIVOT monthly_sales
ON COLUMNS(* EXCLUDE (empid, dept))
INTO
    NAME month
    VALUE sales;
```

empid	dept	month	sales
1	electronics	Jan	1
1	electronics	Feb	2
1	electronics	Mar	3
1	electronics	Apr	4
1	electronics	May	5
1	electronics	Jun	6
2	clothes	Jan	10
2	clothes	Feb	20
2	clothes	Mar	30

empid	dept	month	sales
2	clothes	Apr	40
2	clothes	May	50
2	clothes	Jun	60
3	cars	Jan	100
3	cars	Feb	200
3	cars	Mar	300
3	cars	Apr	400
3	cars	May	500
3	cars	Jun	600

UNPIVOT into Multiple Value Columns

The UNPIVOT statement has additional flexibility: more than 2 destination columns are supported. This can be useful when the goal is to reduce the extent to which a dataset is pivoted, but not completely stack all pivoted columns. To demonstrate this, the query below will generate a dataset with a separate column for the number of each month within the quarter (month 1, 2, or 3), and a separate row for each quarter. Since there are fewer quarters than months, this does make the dataset longer, but not as long as the above.

To accomplish this, multiple sets of columns are included in the ON clause. The q1 and q2 aliases are optional. The number of columns in each set of columns in the ON clause must match the number of columns in the VALUE clause.

```
UNPIVOT monthly_sales
    ON (jan, feb, mar) AS q1, (apr, may, jun) AS q2
    INTO
        NAME quarter
        VALUE month_1_sales, month_2_sales, month_3_sales;
```

empid	dept	quarter	month_1_sales	month_2_sales	month_3_sales
1	electronics	q1	1	2	3
1	electronics	q2	4	5	6
2	clothes	q1	10	20	30
2	clothes	q2	40	50	60
3	cars	q1	100	200	300
3	cars	q2	400	500	600

Using UNPIVOT within a SELECT Statement

The UNPIVOT statement may be included within a SELECT statement as a CTE (a Common Table Expression, or WITH clause), or a subquery. This allows for an UNPIVOT to be used alongside other SQL logic, as well as for multiple UNPIVOTs to be used in one query.

No SELECT is needed within the CTE, the UNPIVOT keyword can be thought of as taking its place.

```
WITH unpivot_alias AS (
    UNPIVOT monthly_sales
    ON COLUMNS(* EXCLUDE (empid, dept))
    INTO
        NAME month
        VALUE sales
```

```
)
SELECT * FROM unpivot_alias;
```

An UNPIVOT may be used in a subquery and must be wrapped in parentheses. Note that this behavior is different than the SQL Standard Unpivot, as illustrated in subsequent examples.

```
SELECT *
FROM (
    UNPIVOT monthly_sales
    ON COLUMNS(* EXCLUDE (empid, dept))
    INTO
        NAME month
        VALUE sales
) unpivot_alias;
```

Expressions within UNPIVOT Statements

DuckDB allows expressions within the UNPIVOT statements, provided that they only involve a single column. These can be used to perform computations as well as explicit casts. For example:

```
UNPIVOT
    (SELECT 42 AS col1, 'woot' AS col2)
    ON
        (col1 * 2)::VARCHAR,
        col2;
```

name	value
col1	84
col2	woot

Simplified UNPIVOT Full Syntax Diagram

Below is the full syntax diagram of the UNPIVOT statement.

SQL Standard UNPIVOT Syntax

The full syntax diagram is below, but the SQL Standard UNPIVOT syntax can be summarized as:

```
FROM [dataset]
UNPIVOT [INCLUDE NULLS] (
    [value-column-name(s)]
    FOR [name-column-name] IN [column(s)]
);
```

Note that only one column can be included in the `name-column-name` expression.

SQL Standard UNPIVOT Manually

To complete the basic UNPIVOT operation using the SQL standard syntax, only a few additions are needed.

```
FROM monthly_sales UNPIVOT (
    sales
    FOR month IN (jan, feb, mar, apr, may, jun)
);
```

empid	dept	month	sales
1	electronics	Jan	1
1	electronics	Feb	2
1	electronics	Mar	3
1	electronics	Apr	4
1	electronics	May	5
1	electronics	Jun	6
2	clothes	Jan	10
2	clothes	Feb	20
2	clothes	Mar	30
2	clothes	Apr	40
2	clothes	May	50
2	clothes	Jun	60
3	cars	Jan	100
3	cars	Feb	200
3	cars	Mar	300
3	cars	Apr	400
3	cars	May	500
3	cars	Jun	600

SQL Standard `UNPIVOT` Dynamically Using the `COLUMNS` Expression

The `COLUMNS` expression can be used to determine the `IN` list of columns dynamically. This will continue to work even if additional `month` columns are added to the dataset. It produces the same result as the query above.

```
FROM monthly_sales UNPIVOT (
    sales
    FOR month IN (columns(* EXCLUDE (empid, dept)))
);
```

SQL Standard `UNPIVOT` into Multiple Value Columns

The `UNPIVOT` statement has additional flexibility: more than 2 destination columns are supported. This can be useful when the goal is to reduce the extent to which a dataset is pivoted, but not completely stack all pivoted columns. To demonstrate this, the query below will generate a dataset with a separate column for the number of each month within the quarter (month 1, 2, or 3), and a separate row for each quarter. Since there are fewer quarters than months, this does make the dataset longer, but not as long as the above.

To accomplish this, multiple columns are included in the `value-column-name` portion of the `UNPIVOT` statement. Multiple sets of columns are included in the `IN` clause. The `q1` and `q2` aliases are optional. The number of columns in each set of columns in the `IN` clause must match the number of columns in the `value-column-name` portion.

```
FROM monthly_sales
UNPIVOT (
    (month_1_sales, month_2_sales, month_3_sales)
    FOR quarter IN (
        (jan, feb, mar) AS q1,
        (apr, may, jun) AS q2
    )
```

```
);
```

empid	dept	quarter	month_1_sales	month_2_sales	month_3_sales
1	electronics	q1	1	2	3
1	electronics	q2	4	5	6
2	clothes	q1	10	20	30
2	clothes	q2	40	50	60
3	cars	q1	100	200	300
3	cars	q2	400	500	600

SQL Standard `UNPIVOT` Full Syntax Diagram

Below is the full syntax diagram of the SQL Standard version of the `UNPIVOT` statement.

UPDATE Statement

The `UPDATE` statement modifies the values of rows in a table.

Examples

For every row where `i` is `NULL`, set the value to 0 instead:

```
UPDATE tbl
SET i = 0
WHERE i IS NULL;
```

Set all values of `i` to 1 and all values of `j` to 2:

```
UPDATE tbl
SET i = 1, j = 2;
```

Syntax

`UPDATE` changes the values of the specified columns in all rows that satisfy the condition. Only the columns to be modified need be mentioned in the `SET` clause; columns not explicitly modified retain their previous values.

Update from Other Table

A table can be updated based upon values from another table. This can be done by specifying a table in a `FROM` clause, or using a sub-select statement. Both approaches have the benefit of completing the `UPDATE` operation in bulk for increased performance.

```
CREATE OR REPLACE TABLE original AS
    SELECT 1 AS key, 'original value' AS value
    UNION ALL
    SELECT 2 AS key, 'original value 2' AS value;

CREATE OR REPLACE TABLE new AS
```

```
    SELECT 1 AS key, 'new value' AS value
    UNION ALL
    SELECT 2 AS key, 'new value 2' AS value;

SELECT *
FROM original;
```

key	value
1	original value
2	original value 2

```
UPDATE original
    SET value = new.value
    FROM new
    WHERE original.key = new.key;
```

Or:

```
UPDATE original
    SET value = (
        SELECT
            new.value
        FROM new
        WHERE original.key = new.key
    );

SELECT *
FROM original;
```

key	value
1	new value
2	new value 2

Update from Same Table

The only difference between this case and the above is that a different table alias must be specified on both the target table and the source table. In this example `AS true_original` and `AS new` are both required.

```
UPDATE original AS true_original
    SET value = (
        SELECT
            new.value || ' a change!' AS value
        FROM original AS new
        WHERE true_original.key = new.key
    );
```

Update Using Joins

To select the rows to update, `UPDATE` statements can use the `FROM` clause and express joins via the `WHERE` clause. For example:

```
CREATE TABLE city (name VARCHAR, revenue BIGINT, country_code VARCHAR);
CREATE TABLE country (code VARCHAR, name VARCHAR);
INSERT INTO city VALUES ('Paris', 700, 'FR'), ('Lyon', 200, 'FR'), ('Brussels', 400, 'BE');
INSERT INTO country VALUES ('FR', 'France'), ('BE', 'Belgium');
```

To increase the revenue of all cities in France, join the `city` and the `country` tables, and filter on the latter:

```
UPDATE city
SET revenue = revenue + 100
FROM country
WHERE city.country_code = country.code
  AND country.name = 'France';
```

```
SELECT *
FROM city;
```

name	revenue	country_code
Paris	800	FR
Lyon	300	FR
Brussels	400	BE

Upsert (Insert or Update)

See the Insert documentation for details.

USE Statement

The USE statement selects a database and optional schema to use as the default.

Examples

```
--- Sets the 'memory' database as the default. Will use 'main' schema implicitly or error
--- if it does not exist.
USE memory;
--- Sets the 'duck.main' database and schema as the default
USE duck.main;
```

Syntax

The USE statement sets a default database or database/schema combination to use for future operations. For instance, tables created without providing a fully qualified table name will be created in the default database.

VACUUM Statement

The VACUUM statement only has basic support in DuckDB and is mostly provided for PostgreSQL-compatibility.

Some variants of it, such as when calling for a given column, recompute the distinct statistics (the amount if distinct entities) if they have become stale due to updates.

> **Warning.** The behavior of VACUUM is not consistent with PostgreSQL semantics and it is likely going to change in the future.

Examples

No-op:

```
VACUUM;
```

No-op:

```
VACUUM ANALYZE;
```

Calling VACUUM on a given table-column pair rebuilds statistics for the table and column:

```
VACUUM my_table(my_column);
```

Rebuild statistics for the table and column:

```
VACUUM ANALYZE my_table(my_column);
```

The following operation is not supported:

```
VACUUM FULL;
```

```
Not implemented Error:
Full vacuum option
```

Reclaiming Space

The VACUUM statement does not reclaim space. For instruction on reclaiming space, refer to the "Reclaiming space" page.

Syntax

Query Syntax

SELECT Clause

The SELECT clause specifies the list of columns that will be returned by the query. While it appears first in the clause, *logically* the expressions here are executed only at the end. The SELECT clause can contain arbitrary expressions that transform the output, as well as aggregates and window functions.

Examples

Select all columns from the table called `table_name`:

```
SELECT * FROM table_name;
```

Perform arithmetic on the columns in a table, and provide an alias:

```
SELECT col1 + col2 AS res, sqrt(col1) AS root FROM table_name;
```

Use prefix aliases:

```
SELECT
    res: col1 + col2,
    root: sqrt(col1)
FROM table_name;
```

Select all unique cities from the `addresses` table:

```
SELECT DISTINCT city FROM addresses;
```

Return the total number of rows in the `addresses` table:

```
SELECT count(*) FROM addresses;
```

Select all columns except the city column from the `addresses` table:

```
SELECT * EXCLUDE (city) FROM addresses;
```

Select all columns from the `addresses` table, but replace `city` with `lower(city)`:

```
SELECT * REPLACE (lower(city) AS city) FROM addresses;
```

Select all columns matching the given regular expression from the table:

```
SELECT COLUMNS('number\d+') FROM addresses;
```

Compute a function on all given columns of a table:

```
SELECT min(COLUMNS(*)) FROM addresses;
```

To select columns with spaces or special characters, use double quotes (`"`):

```
SELECT "Some Column Name" FROM tbl;
```

Syntax

SELECT List

The SELECT clause contains a list of expressions that specify the result of a query. The select list can refer to any columns in the FROM clause, and combine them using expressions. As the output of a SQL query is a table – every expression in the SELECT clause also has a name. The expressions can be explicitly named using the AS clause (e.g., `expr AS name`). If a name is not provided by the user the expressions are named automatically by the system.

> Column names are case-insensitive. See the Rules for Case Sensitivity for more details.

Star Expressions

Select all columns from the table called `table_name`:

```
SELECT *
FROM table_name;
```

Select all columns matching the given regular expression from the table:

```
SELECT COLUMNS('number\d+')
FROM addresses;
```

The star expression is a special expression that expands to *multiple expressions* based on the contents of the FROM clause. In the simplest case, `*` expands to **all** expressions in the FROM clause. Columns can also be selected using regular expressions or lambda functions. See the star expression page for more details.

DISTINCT Clause

Select all unique cities from the addresses table:

```
SELECT DISTINCT city
FROM addresses;
```

The DISTINCT clause can be used to return **only** the unique rows in the result – so that any duplicate rows are filtered out.

> Queries starting with `SELECT DISTINCT` run deduplication, which is an expensive operation. Therefore, only use DISTINCT if necessary.

DISTINCT ON Clause

Select only the highest population city for each country:

```
SELECT DISTINCT ON(country) city, population
FROM cities
ORDER BY population DESC;
```

The `DISTINCT ON` clause returns only one row per unique value in the set of expressions as defined in the ON clause. If an `ORDER BY` clause is present, the row that is returned is the first row that is encountered as per the `ORDER BY` criteria. If an `ORDER BY` clause is not present, the first row that is encountered is not defined and can be any row in the table.

> When querying large data sets, using DISTINCT on all columns can be expensive. Therefore, consider using `DISTINCT ON` on a column (or a set of columns) which guaranetees a sufficient degree of uniqueness for your results. For example, using `DISTINCT ON` on the key column(s) of a table guarantees full uniqueness.

Aggregates

Return the total number of rows in the addresses table:

```
SELECT count(*)
FROM addresses;
```

Return the total number of rows in the addresses table grouped by city:

```
SELECT city, count(*)
FROM addresses
GROUP BY city;
```

Aggregate functions are special functions that *combine* multiple rows into a single value. When aggregate functions are present in the `SELECT` clause, the query is turned into an aggregate query. In an aggregate query, **all** expressions must either be part of an aggregate function, or part of a group (as specified by the `GROUP BY clause`).

Window Functions

Generate a `row_number` column containing incremental identifiers for each row:

```
SELECT row_number() OVER ()
FROM sales;
```

Compute the difference between the current amount, and the previous amount, by order of time:

```
SELECT amount - lag(amount) OVER (ORDER BY time)
FROM sales;
```

Window functions are special functions that allow the computation of values relative to *other rows* in a result. Window functions are marked by the `OVER` clause which contains the *window specification*. The window specification defines the frame or context in which the window function is computed. See the window functions page for more information.

`unnest` Function

Unnest an array by one level:

```
SELECT unnest([1, 2, 3]);
```

Unnest a struct by one level:

```
SELECT unnest({'a': 42, 'b': 84});
```

The `unnest` function is a special function that can be used together with arrays, lists, or structs. The unnest function strips one level of nesting from the type. For example, `INTEGER[]` is transformed into `INTEGER`. `STRUCT(a INTEGER, b INTEGER)` is transformed into `a INTEGER, b INTEGER`. The unnest function can be used to transform nested types into regular scalar types, which makes them easier to operate on.

FROM and JOIN Clauses

The `FROM` clause specifies the *source* of the data on which the remainder of the query should operate. Logically, the `FROM` clause is where the query starts execution. The `FROM` clause can contain a single table, a combination of multiple tables that are joined together using `JOIN` clauses, or another `SELECT` query inside a subquery node. DuckDB also has an optional `FROM`-first syntax which enables you to also query without a `SELECT` statement.

Examples

Select all columns from the table called `table_name`:

```
SELECT *
FROM table_name;
```

Select all columns from the table using the FROM-first syntax:

```
FROM table_name
SELECT *;
```

Select all columns using the FROM-first syntax and omitting the `SELECT` clause:

```
FROM table_name;
```

Select all columns from the table called `table_name` through an alias `tn`:

```
SELECT tn.*
FROM table_name tn;
```

Use a prefix alias:

```
SELECT tn.*
FROM tn: table_name;
```

Select all columns from the table `table_name` in the schema `schema_name`:

```
SELECT *
FROM schema_name.table_name;
```

Select the column `i` from the table function `range`, where the first column of the range function is renamed to `i`:

```
SELECT t.i
FROM range(100) AS t(i);
```

Select all columns from the CSV file called `test.csv`:

```
SELECT *
FROM 'test.csv';
```

Select all columns from a subquery:

```
SELECT *
FROM (SELECT * FROM table_name);
```

Select the entire row of the table as a struct:

```
SELECT t
FROM t;
```

Select the entire row of the subquery as a struct (i.e., a single column):

```
SELECT t
FROM (SELECT unnest(generate_series(41, 43)) AS x, 'hello' AS y) t;
```

Join two tables together:

```
SELECT *
FROM table_name
JOIN other_table
  ON table_name.key = other_table.key;
```

Select a 10% sample from a table:

```
SELECT *
FROM table_name
TABLESAMPLE 10%;
```

Select a sample of 10 rows from a table:

```
SELECT *
FROM table_name
TABLESAMPLE 10 ROWS;
```

Use the FROM-first syntax with WHERE clause and aggregation:

```
FROM range(100) AS t(i)
SELECT sum(t.i)
WHERE i % 2 = 0;
```

Joins

Joins are a fundamental relational operation used to connect two tables or relations horizontally. The relations are referred to as the *left* and *right* sides of the join based on how they are written in the join clause. Each result row has the columns from both relations.

A join uses a rule to match pairs of rows from each relation. Often this is a predicate, but there are other implied rules that may be specified.

Outer Joins

Rows that do not have any matches can still be returned if an OUTER join is specified. Outer joins can be one of:

- LEFT (All rows from the left relation appear at least once)
- RIGHT (All rows from the right relation appear at least once)
- FULL (All rows from both relations appear at least once)

A join that is not OUTER is INNER (only rows that get paired are returned).

When an unpaired row is returned, the attributes from the other table are set to NULL.

Cross Product Joins (Cartesian Product)

The simplest type of join is a CROSS JOIN. There are no conditions for this type of join, and it just returns all the possible pairs.

Return all pairs of rows:

```
SELECT a.*, b.*
FROM a
CROSS JOIN b;
```

This is equivalent to omitting the JOIN clause:

```
SELECT a.*, b.*
FROM a, b;
```

Conditional Joins

Most joins are specified by a predicate that connects attributes from one side to attributes from the other side. The conditions can be explicitly specified using an ON clause with the join (clearer) or implied by the WHERE clause (old-fashioned).

We use the l_regions and the l_nations tables from the TPC-H schema:

```
CREATE TABLE l_regions (
    r_regionkey INTEGER NOT NULL PRIMARY KEY,
    r_name      CHAR(25) NOT NULL,
    r_comment   VARCHAR(152)
);

CREATE TABLE l_nations (
    n_nationkey INTEGER NOT NULL PRIMARY KEY,
    n_name      CHAR(25) NOT NULL,
    n_regionkey INTEGER NOT NULL,
    n_comment   VARCHAR(152),
    FOREIGN KEY (n_regionkey) REFERENCES l_regions(r_regionkey)
);
```

Return the regions for the nations:

```
SELECT n.*, r.*
FROM l_nations n
JOIN l_regions r ON (n_regionkey = r_regionkey);
```

If the column names are the same and are required to be equal, then the simpler `USING` syntax can be used:

```
CREATE TABLE l_regions (regionkey INTEGER NOT NULL PRIMARY KEY,
                        name      CHAR(25) NOT NULL,
                        comment   VARCHAR(152));

CREATE TABLE l_nations (nationkey INTEGER NOT NULL PRIMARY KEY,
                        name      CHAR(25) NOT NULL,
                        regionkey INTEGER NOT NULL,
                        comment   VARCHAR(152),
                        FOREIGN KEY (regionkey) REFERENCES l_regions(regionkey));
```

Return the regions for the nations:

```
SELECT n.*, r.*
FROM l_nations n
JOIN l_regions r USING (regionkey);
```

The expressions do not have to be equalities – any predicate can be used:

Return the pairs of jobs where one ran longer but cost less:

```
SELECT s1.t_id, s2.t_id
FROM west s1, west s2
WHERE s1.time > s2.time
  AND s1.cost < s2.cost;
```

Natural Joins

Natural joins join two tables based on attributes that share the same name.

For example, take the following example with cities, airport codes and airport names. Note that both tables are intentionally incomplete, i.e., they do not have a matching pair in the other table.

```
CREATE TABLE city_airport (city_name VARCHAR, iata VARCHAR);
CREATE TABLE airport_names (iata VARCHAR, airport_name VARCHAR);
INSERT INTO city_airport VALUES
    ('Amsterdam', 'AMS'),
    ('Rotterdam', 'RTM'),
    ('Eindhoven', 'EIN'),
    ('Groningen', 'GRQ');
INSERT INTO airport_names VALUES
```

```
    ('AMS', 'Amsterdam Airport Schiphol'),
    ('RTM', 'Rotterdam The Hague Airport'),
    ('MST', 'Maastricht Aachen Airport');
```

To join the tables on their shared `IATA` attributes, run:

```
SELECT *
FROM city_airport
NATURAL JOIN airport_names;
```

This produces the following result:

city_name	iata	airport_name
Amsterdam	AMS	Amsterdam Airport Schiphol
Rotterdam	RTM	Rotterdam The Hague Airport

Note that only rows where the same `iata` attribute was present in both tables were included in the result.

We can also express query using the vanilla `JOIN` clause with the `USING` keyword:

```
SELECT *
FROM city_airport
JOIN airport_names
USING (iata);
```

Semi and Anti Joins

Semi joins return rows from the left table that have at least one match in the right table. Anti joins return rows from the left table that have *no* matches in the right table. When using a semi or anti join the result will never have more rows than the left hand side table. Semi joins provide the same logic as the `IN` operator statement. Anti joins provide the same logic as the `NOT IN` operator, except anti joins ignore `NULL` values from the right table.

Semi Join Example

Return a list of city-airport code pairs from the `city_airport` table where the airport name **is available** in the `airport_names` table:

```
SELECT *
FROM city_airport
SEMI JOIN airport_names
    USING (iata);
```

city_name	iata
Amsterdam	AMS
Rotterdam	RTM

This query is equivalent with:

```
SELECT *
FROM city_airport
WHERE iata IN (SELECT iata FROM airport_names);
```

Anti Join Example

Return a list of city–airport code pairs from the `city_airport` table where the airport name **is not available** in the `airport_names` table:

```
SELECT *
FROM city_airport
ANTI JOIN airport_names
    USING (iata);
```

city_name	iata
Eindhoven	EIN
Groningen	GRQ

This query is equivalent with:

```
SELECT *
FROM city_airport
WHERE iata NOT IN (SELECT iata FROM airport_names WHERE iata IS NOT NULL);
```

Lateral Joins

The LATERAL keyword allows subqueries in the FROM clause to refer to previous subqueries. This feature is also known as a *lateral join*.

```
SELECT *
FROM range(3) t(i), LATERAL (SELECT i + 1) t2(j);
```

i	j
0	1
2	3
1	2

Lateral joins are a generalization of correlated subqueries, as they can return multiple values per input value rather than only a single value.

```
SELECT *
FROM
    generate_series(0, 1) t(i),
    LATERAL (SELECT i + 10 UNION ALL SELECT i + 100) t2(j);
```

i	j
0	10
1	11
0	100
1	101

It may be helpful to think about LATERAL as a loop where we iterate through the rows of the first subquery and use it as input to the second (LATERAL) subquery. In the examples above, we iterate through table `t` and refer to its column `i` from the definition of table `t2`. The rows of `t2` form column `j` in the result.

It is possible to refer to multiple attributes from the LATERAL subquery. Using the table from the first example:

```
CREATE TABLE t1 AS
    SELECT *
    FROM range(3) t(i), LATERAL (SELECT i + 1) t2(j);

SELECT *
    FROM t1, LATERAL (SELECT i + j) t2(k)
    ORDER BY ALL;
```

i	j	k
0	1	1
1	2	3
2	3	5

> DuckDB detects when LATERAL joins should be used, making the use of the LATERAL keyword optional.

Positional Joins

When working with data frames or other embedded tables of the same size, the rows may have a natural correspondence based on their physical order. In scripting languages, this is easily expressed using a loop:

```
for (i = 0; i < n; i++) {
    f(t1.a[i], t2.b[i]);
}
```

It is difficult to express this in standard SQL because relational tables are not ordered, but imported tables such as data frames or disk files (like CSVs or Parquet files) do have a natural ordering.

Connecting them using this ordering is called a *positional join:*

```
CREATE TABLE t1 (x INTEGER);
CREATE TABLE t2 (s VARCHAR);

INSERT INTO t1 VALUES (1), (2), (3);
INSERT INTO t2 VALUES ('a'), ('b');

SELECT *
FROM t1
POSITIONAL JOIN t2;
```

x	s
1	a
2	b
3	NULL

Positional joins are always `FULL OUTER` joins, i.e., missing values (the last values in the shorter column) are set to `NULL`.

As-Of Joins

A common operation when working with temporal or similarly-ordered data is to find the nearest (first) event in a reference table (such as prices). This is called an *as-of join:*

Attach prices to stock trades:

```
SELECT t.*, p.price
FROM trades t
ASOF JOIN prices p
       ON t.symbol = p.symbol AND t.when >= p.when;
```

The ASOF join requires at least one inequality condition on the ordering field. The inequality can be any inequality condition (>=, >, <=, <) on any data type, but the most common form is >= on a temporal type. Any other conditions must be equalities (or `NOT DISTINCT`). This means that the left/right order of the tables is significant.

ASOF joins each left side row with at most one right side row. It can be specified as an OUTER join to find unpaired rows (e.g., trades without prices or prices which have no trades.)

Attach prices or NULLs to stock trades:

```
SELECT *
FROM trades t
ASOF LEFT JOIN prices p
            ON t.symbol = p.symbol
           AND t.when >= p.when;
```

ASOF joins can also specify join conditions on matching column names with the `USING` syntax, but the *last* attribute in the list must be the inequality, which will be greater than or equal to (>=):

```
SELECT *
FROM trades t
ASOF JOIN prices p USING (symbol, "when");
```

Returns symbol, trades.when, price (but NOT prices.when):

If you combine `USING` with a `SELECT *` like this, the query will return the left side (probe) column values for the matches, not the right side (build) column values. To get the `prices` times in the example, you will need to list the columns explicitly:

```
SELECT t.symbol, t.when AS trade_when, p.when AS price_when, price
FROM trades t
ASOF LEFT JOIN prices p USING (symbol, "when");
```

Self-Joins

DuckDB allows self-joins for all types of joins. Note that tables need to be aliased, using the same table name without aliases will result in an error:

```
CREATE TABLE t (x INTEGER);
SELECT * FROM t JOIN t USING(x);
```

```
Binder Error:
Duplicate alias "t" in query!
```

Adding the aliases allows the query to parse successfully:

```
SELECT * FROM t AS t t1 JOIN t t2 USING(x);
```

Shorthands in the `JOIN` Clause

You can specify column names in the `JOIN` clause:

```
CREATE TABLE t1 (x INTEGER);
CREATE TABLE t2 (y INTEGER);
INSERT INTO t1 VALUES (1), (2), (4);
INSERT INTO t2 VALUES (2), (3);
SELECT * FROM t1 NATURAL JOIN t2 t2(x);
```

x
2

You can also use the `VALUES` clause in the `JOIN` clause:

```
SELECT * FROM t1 NATURAL JOIN (VALUES (2), (4)) _(x);
```

x
2
4

FROM-First Syntax

DuckDB's SQL supports the `FROM`-first syntax, i.e., it allows putting the `FROM` clause before the `SELECT` clause or completely omitting the `SELECT` clause. We use the following example to demonstrate it:

```
CREATE TABLE tbl AS
    SELECT *
    FROM (VALUES ('a'), ('b')) t1(s), range(1, 3) t2(i);
```

FROM-First Syntax with a SELECT Clause

The following statement demonstrates the use of the `FROM`-first syntax:

```
FROM tbl
SELECT i, s;
```

This is equivalent to:

```
SELECT i, s
FROM tbl;
```

i	s
1	a
2	a
1	b
2	b

FROM-First Syntax without a SELECT Clause

The following statement demonstrates the use of the optional `SELECT` clause:

```
FROM tbl;
```

This is equivalent to:

```
SELECT *
FROM tbl;
```

s	i
a	1
a	2
b	1
b	2

Time Travel Using AT

DuckDB v1.3.0 introduced support for time travel queries on data lake formats such as Delta, DuckLake and Iceberg.

To specify a time travel query, use the `AT` modifier in the `FROM` clause. Time travel queries can use either a version number or the timestamp specified with `VERSION => <version>`{:.language-sql .highlight} and `TIMESTAMP => <timestamp or date>`{:.language-sql .highlight}, respectively. For example:

```
FROM my_ducklake.demo AT (VERSION => 2);
FROM my_ducklake.demo AT (TIMESTAMP => DATE '2025-05-26');
```

Syntax

WHERE Clause

The `WHERE` clause specifies any filters to apply to the data. This allows you to select only a subset of the data in which you are interested. Logically the `WHERE` clause is applied immediately after the `FROM` clause.

Examples

Select all rows that where the `id` is equal to 3:

```
SELECT *
FROM table_name
WHERE id = 3;
```

Select all rows that match the given **case-sensitive** `LIKE` expression:

```
SELECT *
FROM table_name
WHERE name LIKE '%mark%';
```

Select all rows that match the given **case-insensitive** expression formulated with the `ILIKE` operator:

```
SELECT *
FROM table_name
WHERE name ILIKE '%mark%';
```

Select all rows that match the given composite expression:

```
SELECT *
FROM table_name
WHERE id = 3 OR id = 7;
```

Syntax

GROUP BY Clause

The GROUP BY clause specifies which grouping columns should be used to perform any aggregations in the SELECT clause. If the GROUP BY clause is specified, the query is always an aggregate query, even if no aggregations are present in the SELECT clause.

When a GROUP BY clause is specified, all tuples that have matching data in the grouping columns (i.e., all tuples that belong to the same group) will be combined. The values of the grouping columns themselves are unchanged, and any other columns can be combined using an aggregate function (such as `count`, `sum`, `avg`, etc).

GROUP BY ALL

Use GROUP BY ALL to GROUP BY all columns in the SELECT statement that are not wrapped in aggregate functions. This simplifies the syntax by allowing the columns list to be maintained in a single location, and prevents bugs by keeping the SELECT granularity aligned to the GROUP BY granularity (e.g., it prevents duplication). See examples below and additional examples in the "Friendlier SQL with DuckDB" blog post.

Multiple Dimensions

Normally, the GROUP BY clause groups along a single dimension. Using the GROUPING SETS, CUBE or ROLLUP clauses it is possible to group along multiple dimensions. See the GROUPING SETS page for more information.

Examples

Count the number of entries in the `addresses` table that belong to each different city:

```
SELECT city, count(*)
FROM addresses
GROUP BY city;
```

Compute the average income per city per street_name:

```
SELECT city, street_name, avg(income)
FROM addresses
GROUP BY city, street_name;
```

GROUP BY ALL Examples

Group by city and street_name to remove any duplicate values:

```
SELECT city, street_name
FROM addresses
GROUP BY ALL;
```

Compute the average income per city per street_name. Since income is wrapped in an aggregate function, do not include it in the GROUP BY:

```
SELECT city, street_name, avg(income)
FROM addresses
GROUP BY ALL;
-- GROUP BY city, street_name:
```

Syntax

GROUPING SETS

GROUPING SETS, ROLLUP and CUBE can be used in the GROUP BY clause to perform a grouping over multiple dimensions within the same query. Note that this syntax is not compatible with `GROUP BY ALL`.

Examples

Compute the average income along the provided four different dimensions:

```
-- the syntax () denotes the empty set (i.e., computing an ungrouped aggregate)
SELECT city, street_name, avg(income)
FROM addresses
GROUP BY GROUPING SETS ((city, street_name), (city), (street_name), ());
```

Compute the average income along the same dimensions:

```
SELECT city, street_name, avg(income)
FROM addresses
GROUP BY CUBE (city, street_name);
```

Compute the average income along the dimensions `(city, street_name)`, `(city)` and `()`:

```
SELECT city, street_name, avg(income)
FROM addresses
GROUP BY ROLLUP (city, street_name);
```

Description

GROUPING SETS perform the same aggregate across different GROUP BY `clauses` in a single query.

```
CREATE TABLE students (course VARCHAR, type VARCHAR);
INSERT INTO students (course, type)
VALUES
    ('CS', 'Bachelor'), ('CS', 'Bachelor'), ('CS', 'PhD'), ('Math', 'Masters'),
    ('CS', NULL), ('CS', NULL), ('Math', NULL);
```

```
SELECT course, type, count(*)
FROM students
GROUP BY GROUPING SETS ((course, type), course, type, ());
```

course	type	count_star()
Math	NULL	1
NULL	NULL	7
CS	PhD	1
CS	Bachelor	2
Math	Masters	1
CS	NULL	2
Math	NULL	2
CS	NULL	5
NULL	NULL	3
NULL	Masters	1
NULL	Bachelor	2
NULL	PhD	1

In the above query, we group across four different sets: `course, type`, `course`, `type` and `()` (the empty group). The result contains `NULL` for a group which is not in the grouping set for the result, i.e., the above query is equivalent to the following statement of `UNION ALL` clauses:

```
-- Group by course, type:
SELECT course, type, count(*)
FROM students
GROUP BY course, type
UNION ALL
-- Group by type:
SELECT NULL AS course, type, count(*)
FROM students
GROUP BY type
UNION ALL
-- Group by course:
SELECT course, NULL AS type, count(*)
FROM students
GROUP BY course
UNION ALL
-- Group by nothing:
SELECT NULL AS course, NULL AS type, count(*)
FROM students;
```

CUBE and ROLLUP are syntactic sugar to easily produce commonly used grouping sets.

The ROLLUP clause will produce all "sub-groups" of a grouping set, e.g., `ROLLUP (country, city, zip)` produces the grouping sets `(country, city, zip)`, `(country, city)`, `(country)`, `()`. This can be useful for producing different levels of detail of a group by clause. This produces `n+1` grouping sets where n is the amount of terms in the `ROLLUP` clause.

CUBE produces grouping sets for all combinations of the inputs, e.g., `CUBE (country, city, zip)` will produce `(country, city, zip)`, `(country, city)`, `(country, zip)`, `(city, zip)`, `(country)`, `(city)`, `(zip)`, `()`. This produces `2^n` grouping sets.

Identifying Grouping Sets with GROUPING_ID()

The super-aggregate rows generated by GROUPING SETS, ROLLUP and CUBE can often be identified by NULL-values returned for the respective column in the grouping. But if the columns used in the grouping can themselves contain actual NULL-values, then it can be challenging to distinguish whether the value in the resultset is a "real" NULL-value coming out of the data itself, or a NULL-value generated by the grouping construct. The GROUPING_ID() or GROUPING() function is designed to identify which groups generated the super-aggregate rows in the result.

GROUPING_ID() is an aggregate function that takes the column expressions that make up the grouping(s). It returns a BIGINT value. The return value is 0 for the rows that are not super-aggregate rows. But for the super-aggregate rows, it returns an integer value that identifies the combination of expressions that make up the group for which the super-aggregate is generated. At this point, an example might help. Consider the following query:

```
WITH days AS (
    SELECT
        year("generate_series")    AS y,
        quarter("generate_series") AS q,
        month("generate_series")   AS m
    FROM generate_series(DATE '2023-01-01', DATE '2023-12-31', INTERVAL 1 DAY)
)
SELECT y, q, m, GROUPING_ID(y, q, m) AS "grouping_id()"
FROM days
GROUP BY GROUPING SETS (
    (y, q, m),
    (y, q),
    (y),
    ()
)
ORDER BY y, q, m;
```

These are the results:

y	q	m	grouping_id()
2023	1	1	0
2023	1	2	0
2023	1	3	0
2023	1	NULL	1
2023	2	4	0
2023	2	5	0
2023	2	6	0
2023	2	NULL	1
2023	3	7	0
2023	3	8	0
2023	3	9	0
2023	3	NULL	1
2023	4	10	0
2023	4	11	0
2023	4	12	0
2023	4	NULL	1
2023	NULL	NULL	3

y	q	m	grouping_id()
NULL	NULL	NULL	7

In this example, the lowest level of grouping is at the month level, defined by the grouping set `(y, q, m)`. Result rows corresponding to that level are simply aggregate rows and the `GROUPING_ID(y, q, m)` function returns `0` for those. The grouping set `(y, q)` results in super-aggregate rows over the month level, leaving a `NULL`-value for the `m` column, and for which `GROUPING_ID(y, q, m)` returns `1`. The grouping set `(y)` results in super-aggregate rows over the quarter level, leaving `NULL`-values for the `m` and `q` column, for which `GROUPING_ID(y, q, m)` returns `3`. Finally, the `()` grouping set results in one super-aggregate row for the entire resultset, leaving `NULL`-values for `y`, `q` and `m` and for which `GROUPING_ID(y, q, m)` returns `7`.

To understand the relationship between the return value and the grouping set, you can think of `GROUPING_ID(y, q, m)` writing to a bitfield, where the first bit corresponds to the last expression passed to `GROUPING_ID()`, the second bit to the one-but-last expression passed to `GROUPING_ID()`, and so on. This may become clearer by casting `GROUPING_ID()` to `BIT`:

```
WITH days AS (
    SELECT
        year("generate_series")    AS y,
        quarter("generate_series") AS q,
        month("generate_series")   AS m
    FROM generate_series(DATE '2023-01-01', DATE '2023-12-31', INTERVAL 1 DAY)
)
SELECT
    y, q, m,
    GROUPING_ID(y, q, m) AS "grouping_id(y, q, m)",
    right(GROUPING_ID(y, q, m)::BIT::VARCHAR, 3) AS "y_q_m_bits"
FROM days
GROUP BY GROUPING SETS (
    (y, q, m),
    (y, q),
    (y),
    ()
)
ORDER BY y, q, m;
```

Which returns these results:

y	q	m	grouping_id(y, q, m)	y_q_m_bits
2023	1	1	0	000
2023	1	2	0	000
2023	1	3	0	000
2023	1	NULL	1	001
2023	2	4	0	000
2023	2	5	0	000
2023	2	6	0	000
2023	2	NULL	1	001
2023	3	7	0	000
2023	3	8	0	000
2023	3	9	0	000
2023	3	NULL	1	001
2023	4	10	0	000

y	q	m	grouping_id(y, q, m)	y_q_m_bits
2023	4	11	0	000
2023	4	12	0	000
2023	4	NULL	1	001
2023	NULL	NULL	3	011
NULL	NULL	NULL	7	111

Note that the number of expressions passed to `GROUPING_ID()`, or the order in which they are passed is independent from the actual group definitions appearing in the `GROUPING SETS`-clause (or the groups implied by `ROLLUP` and `CUBE`). As long as the expressions passed to `GROUPING_ID()` are expressions that appear some where in the `GROUPING SETS`-clause, `GROUPING_ID()` will set a bit corresponding to the position of the expression whenever that expression is rolled up to a super-aggregate.

Syntax

HAVING Clause

The `HAVING` clause can be used after the `GROUP BY` clause to provide filter criteria *after* the grouping has been completed. In terms of syntax the `HAVING` clause is identical to the `WHERE` clause, but while the `WHERE` clause occurs before the grouping, the `HAVING` clause occurs after the grouping.

Examples

Count the number of entries in the `addresses` table that belong to each different `city`, filtering out cities with a count below 50:

```
SELECT city, count(*)
FROM addresses
GROUP BY city
HAVING count(*) >= 50;
```

Compute the average income per city per `street_name`, filtering out cities with an average `income` bigger than twice the median `income`:

```
SELECT city, street_name, avg(income)
FROM addresses
GROUP BY city, street_name
HAVING avg(income) > 2 * median(income);
```

Syntax

ORDER BY Clause

`ORDER BY` is an output modifier. Logically it is applied near the very end of the query (just prior to `LIMIT` or `OFFSET`, if present). The `ORDER BY` clause sorts the rows on the sorting criteria in either ascending or descending order. In addition, every order clause can specify whether `NULL` values should be moved to the beginning or to the end.

The `ORDER BY` clause may contain one or more expressions, separated by commas. An error will be thrown if no expressions are included, since the `ORDER BY` clause should be removed in that situation. The expressions may begin with either an arbitrary scalar expression

(which could be a column name), a column position number (where the indexing starts from 1), or the keyword `ALL`. Each expression can optionally be followed by an order modifier (`ASC` or `DESC`, default is `ASC`), and/or a `NULL` order modifier (`NULLS FIRST` or `NULLS LAST`, default is `NULLS LAST`).

`ORDER BY ALL`

The `ALL` keyword indicates that the output should be sorted by every column in order from left to right. The direction of this sort may be modified using either `ORDER BY ALL ASC` or `ORDER BY ALL DESC` and/or `NULLS FIRST` or `NULLS LAST`. Note that `ALL` may not be used in combination with other expressions in the `ORDER BY` clause – it must be by itself. See examples below.

NULL Order Modifier

By default, DuckDB sorts `ASC` and `NULLS LAST`, i.e., the values are sorted in ascending order and `NULL` values are placed last. This is identical to the default sort order of PostgreSQL. The default sort order can be changed with the following configuration options.

Use the `default_null_order` option to change the default `NULL` sorting order to either `NULLS_FIRST`, `NULLS_LAST`, `NULLS_FIRST_ON_ASC_LAST_ON_DESC` or `NULLS_LAST_ON_ASC_FIRST_ON_DESC`:

```
SET default_null_order = 'NULLS_FIRST';
```

Use the `default_order` to change the direction of the default sorting order to either `DESC` or `ASC`:

```
SET default_order = 'DESC';
```

Collations

Text is sorted using the binary comparison collation by default, which means values are sorted on their binary UTF-8 values. While this works well for ASCII text (e.g., for English language data), the sorting order can be incorrect for other languages. For this purpose, DuckDB provides collations. For more information on collations, see the Collation page.

Examples

All examples use this example table:

```
CREATE OR REPLACE TABLE addresses AS
    SELECT '123 Quack Blvd' AS address, 'DuckTown' AS city, '11111' AS zip
    UNION ALL
    SELECT '111 Duck Duck Goose Ln', 'DuckTown', '11111'
    UNION ALL
    SELECT '111 Duck Duck Goose Ln', 'Duck Town', '11111'
    UNION ALL
    SELECT '111 Duck Duck Goose Ln', 'Duck Town', '11111-0001';
```

Select the addresses, ordered by city name using the default `NULL` order and default order:

```
SELECT *
FROM addresses
ORDER BY city;
```

Select the addresses, ordered by city name in descending order with nulls at the end:

```
SELECT *
FROM addresses
ORDER BY city DESC NULLS LAST;
```

Order by city and then by zip code, both using the default orderings:

```
SELECT *
FROM addresses
ORDER BY city, zip;
```

Order by city using German collation rules:

```
SELECT *
FROM addresses
ORDER BY city COLLATE DE;
```

ORDER BY ALL Examples

Order from left to right (by address, then by city, then by zip) in ascending order:

```
SELECT *
FROM addresses
ORDER BY ALL;
```

address	city	zip
111 Duck Duck Goose Ln	Duck Town	11111
111 Duck Duck Goose Ln	Duck Town	11111-0001
111 Duck Duck Goose Ln	DuckTown	11111
123 Quack Blvd	DuckTown	11111

Order from left to right (by address, then by city, then by zip) in descending order:

```
SELECT *
FROM addresses
ORDER BY ALL DESC;
```

address	city	zip
123 Quack Blvd	DuckTown	11111
111 Duck Duck Goose Ln	DuckTown	11111
111 Duck Duck Goose Ln	Duck Town	11111-0001
111 Duck Duck Goose Ln	Duck Town	11111

Syntax

LIMIT and OFFSET Clauses

`LIMIT` is an output modifier. Logically it is applied at the very end of the query. The `LIMIT` clause restricts the amount of rows fetched. The `OFFSET` clause indicates at which position to start reading the values, i.e., the first `OFFSET` values are ignored.

Note that while `LIMIT` can be used without an `ORDER BY` clause, the results might not be deterministic without the `ORDER BY` clause. This can still be useful, however, for example when you want to inspect a quick snapshot of the data.

Examples

Select the first 5 rows from the addresses table:

```
SELECT *
FROM addresses
LIMIT 5;
```

Select the 5 rows from the addresses table, starting at position 5 (i.e., ignoring the first 5 rows):

```
SELECT *
FROM addresses
LIMIT 5
OFFSET 5;
```

Select the top 5 cities with the highest population:

```
SELECT city, count(*) AS population
FROM addresses
GROUP BY city
ORDER BY population DESC
LIMIT 5;
```

Select 10% of the rows from the addresses table:

```
SELECT *
FROM addresses
LIMIT 10%;
```

Syntax

SAMPLE Clause

The SAMPLE clause allows you to run the query on a sample from the base table. This can significantly speed up processing of queries, at the expense of accuracy in the result. Samples can also be used to quickly see a snapshot of the data when exploring a data set. The sample clause is applied right after anything in the FROM clause (i.e., after any joins, but before the WHERE clause or any aggregates). See the SAMPLE page for more information.

Examples

Select a sample of 1% of the addresses table using default (system) sampling:

```
SELECT *
FROM addresses
USING SAMPLE 1%;
```

Select a sample of 1% of the addresses table using bernoulli sampling:

```
SELECT *
FROM addresses
USING SAMPLE 1% (bernoulli);
```

Select a sample of 10 rows from the subquery:

```
SELECT *
FROM (SELECT * FROM addresses)
USING SAMPLE 10 ROWS;
```

Syntax

Unnesting

Examples

Unnest a list, generating 3 rows (1, 2, 3):

```
SELECT unnest([1, 2, 3]);
```

Unnesting a struct, generating two columns (a, b):

```
SELECT unnest({'a': 42, 'b': 84});
```

Recursive unnest of a list of structs:

```
SELECT unnest([{'a': 42, 'b': 84}, {'a': 100, 'b': NULL}], recursive := true);
```

Limit depth of recursive unnest using `max_depth`:

```
SELECT unnest([[[1, 2], [3, 4]], [[5, 6], [7, 8, 9], []], [[10, 11]]], max_depth := 2);
```

The `unnest` special function is used to unnest lists or structs by one level. The function can be used as a regular scalar function, but only in the `SELECT` clause. Invoking `unnest` with the `recursive` parameter will unnest lists and structs of multiple levels. The depth of unnesting can be limited using the `max_depth` parameter (which assumes `recursive` unnesting by default).

Unnesting Lists

Unnest a list, generating 3 rows (1, 2, 3):

```
SELECT unnest([1, 2, 3]);
```

Unnest a list, generating 3 rows ((1, 10), (2, 10), (3, 10)):

```
SELECT unnest([1, 2, 3]), 10;
```

Unnest two lists of different sizes, generating 3 rows ((1, 10), (2, 11), (3, NULL)):

```
SELECT unnest([1, 2, 3]), unnest([10, 11]);
```

Unnest a list column from a subquery:

```
SELECT unnest(l) + 10 FROM (VALUES ([1, 2, 3]), ([4, 5])) tbl(l);
```

Empty result:

```
SELECT unnest([]);
```

Empty result:

```
SELECT unnest(NULL);
```

Using `unnest` on a list emits one row per list entry. Regular scalar expressions in the same `SELECT` clause are repeated for every emitted row. When multiple lists are unnested in the same `SELECT` clause, the lists are unnested side-by-side. If one list is longer than the other, the shorter list is padded with `NULL` values.

Empty and `NULL` lists both unnest to zero rows.

Unnesting Structs

Unnesting a struct, generating two columns (a, b):

```
SELECT unnest({'a': 42, 'b': 84});
```

Unnesting a struct, generating two columns (a, b):

```
SELECT unnest({'a': 42, 'b': {'x': 84}});
```

`unnest` on a struct will emit one column per entry in the struct.

Recursive Unnest

Unnesting a list of lists recursively, generating 5 rows (1, 2, 3, 4, 5):

```
SELECT unnest([[1, 2, 3], [4, 5]], recursive := true);
```

Unnesting a list of structs recursively, generating two rows of two columns (a, b):

```
SELECT unnest([{'a': 42, 'b': 84}, {'a': 100, 'b': NULL}], recursive := true);
```

Unnesting a struct, generating two columns (a, b):

```
SELECT unnest({'a': [1, 2, 3], 'b': 88}, recursive := true);
```

Calling `unnest` with the `recursive` setting will fully unnest lists, followed by fully unnesting structs. This can be useful to fully flatten columns that contain lists within lists, or lists of structs. Note that lists *within* structs are not unnested.

Setting the Maximum Depth of Unnesting

The `max_depth` parameter allows limiting the maximum depth of recursive unnesting (which is assumed by default and does not have to be specified separately). For example, unnestig to `max_depth` of 2 yields the following:

```
SELECT unnest([[[1, 2], [3, 4]], [[5, 6], [7, 8, 9], []], [[10, 11]]], max_depth := 2) AS x;
```

x
[1, 2]
[3, 4]
[5, 6]
[7, 8, 9]
[]
[10, 11]

Meanwhile, unnesting to `max_depth` of 3 results in:

```
SELECT unnest([[[1, 2], [3, 4]], [[5, 6], [7, 8, 9], []], [[10, 11]]], max_depth := 3) AS x;
```

x
1
2
3

x
4
5
6
7
8
9
10
11

Keeping Track of List Entry Positions

To keep track of each entry's position within the original list, `unnest` may be combined with `generate_subscripts`:

```
SELECT unnest(l) AS x, generate_subscripts(l, 1) AS index
FROM (VALUES ([1, 2, 3]), ([4, 5])) tbl(l);
```

x	index
1	1
2	2
3	3
4	1
5	2

WITH Clause

The `WITH` clause allows you to specify common table expressions (CTEs). Regular (non-recursive) common-table-expressions are essentially views that are limited in scope to a particular query. CTEs can reference each-other and can be nested. Recursive CTEs can reference themselves.

Basic CTE Examples

Create a CTE called `cte` and use it in the main query:

```
WITH cte AS (SELECT 42 AS x)
SELECT * FROM cte;
```

x
42

Create two CTEs `cte1` and `cte2`, where the second CTE references the first CTE:

```
WITH
    cte1 AS (SELECT 42 AS i),
    cte2 AS (SELECT i * 100 AS x FROM cte1)
SELECT * FROM cte2;
```

x
4200

You can specify column names for CTEs:

```
WITH cte(j) AS (SELECT 42 AS i)
FROM cte;
```

CTE Materialization

DuckDB can employ CTE materialization, instead of inlining CTEs into the main query. This is performed using heuristics: if the CTE performs a grouped aggregation and is queried more than once, it is materialized. Materialization can be explicitly activated by defining the CTE using `AS MATERIALIZED` and disabled by using `AS NOT MATERIALIZED`.

Take the following query for example, which invokes the same CTE three times:

```
WITH t(x) AS (<complex_query>)
SELECT *
FROM
    t AS t1,
    t AS t2,
    t AS t3;
```

Inlining duplicates the definition of `t` for each reference which results in the following query:

```
SELECT *
FROM
    (<complex_query>) AS t1(x),
    (<complex_query>) AS t2(x),
    (<complex_query>) AS t3(x);
```

If `complex_query` is expensive, materializing it with the `MATERIALIZED` keyword can improve performance. In this case, `complex_query` is evaluated only once.

```
WITH t(x) AS MATERIALIZED (<complex_query>)
SELECT *
FROM
    t AS t1,
    t AS t2,
    t AS t3;
```

If one wants to disable materialization, use `NOT MATERIALIZED`:

```
WITH t(x) AS NOT MATERIALIZED (<complex_query>)
SELECT *
FROM
    t AS t1,
    t AS t2,
    t AS t3;
```

Recursive CTEs

`WITH RECURSIVE` allows the definition of CTEs which can refer to themselves. Note that the query must be formulated in a way that ensures termination, otherwise, it may run into an infinite loop.

Example: Fibonacci Sequence

`WITH RECURSIVE` can be used to make recursive calculations. For example, here is how `WITH RECURSIVE` could be used to calculate the first ten Fibonacci numbers:

```
WITH RECURSIVE FibonacciNumbers (
    RecursionDepth, FibonacciNumber, NextNumber
) AS (
        -- Base case
        SELECT
            0 AS RecursionDepth,
            0 AS FibonacciNumber,
            1 AS NextNumber
        UNION ALL
        -- Recursive step
        SELECT
            fib.RecursionDepth + 1 AS RecursionDepth,
            fib.NextNumber AS FibonacciNumber,
            fib.FibonacciNumber + fib.NextNumber AS NextNumber
        FROM
            FibonacciNumbers fib
        WHERE
            fib.RecursionDepth + 1 < 10
    )
SELECT
    fn.RecursionDepth AS FibonacciNumberIndex,
    fn.FibonacciNumber
FROM
    FibonacciNumbers fn;
```

FibonacciNumberIndex	FibonacciNumber
0	0
1	1
2	1
3	2
4	3
5	5
6	8
7	13
8	21
9	34

Example: Tree Traversal

`WITH RECURSIVE` can be used to traverse trees. For example, take a hierarchy of tags:

```
CREATE TABLE tag (id INTEGER, name VARCHAR, subclassof INTEGER);
INSERT INTO tag VALUES
    (1, 'U2',     5),
    (2, 'Blur',   5),
    (3, 'Oasis',  5),
    (4, '2Pac',   6),
    (5, 'Rock',   7),
    (6, 'Rap',    7),
    (7, 'Music',  9),
    (8, 'Movies', 9),
    (9, 'Art', NULL);
```

The following query returns the path from the node `Oasis` to the root of the tree (`Art`).

```
WITH RECURSIVE tag_hierarchy(id, source, path) AS (
        SELECT id, name, [name] AS path
        FROM tag
        WHERE subclassof IS NULL
    UNION ALL
        SELECT tag.id, tag.name, list_prepend(tag.name, tag_hierarchy.path)
        FROM tag, tag_hierarchy
        WHERE tag.subclassof = tag_hierarchy.id
    )
SELECT path
FROM tag_hierarchy
WHERE source = 'Oasis';
```

path
[Oasis, Rock, Music, Art]

Graph Traversal

The `WITH RECURSIVE` clause can be used to express graph traversal on arbitrary graphs. However, if the graph has cycles, the query must perform cycle detection to prevent infinite loops. One way to achieve this is to store the path of a traversal in a list and, before extending the path with a new edge, check whether its endpoint has been visited before (see the example later).

Take the following directed graph from the LDBC Graphalytics benchmark:

```
CREATE TABLE edge (node1id INTEGER, node2id INTEGER);
INSERT INTO edge VALUES
    (1, 3), (1, 5), (2, 4), (2, 5), (2, 10), (3, 1),
    (3, 5), (3, 8), (3, 10), (5, 3), (5, 4), (5, 8),
    (6, 3), (6, 4), (7, 4), (8, 1), (9, 4);
```

Note that the graph contains directed cycles, e.g., between nodes 1, 5 and 8.

Enumerate All Paths from a Node

The following query returns **all paths** starting in node 1:

```
WITH RECURSIVE paths(startNode, endNode, path) AS (
        SELECT -- Define the path as the first edge of the traversal
            node1id AS startNode,
            node2id AS endNode,
            [node1id, node2id] AS path
        FROM edge
        WHERE startNode = 1
```

```
        UNION ALL
        SELECT -- Concatenate new edge to the path
            paths.startNode AS startNode,
            node2id AS endNode,
            array_append(path, node2id) AS path
        FROM paths
        JOIN edge ON paths.endNode = node1id
        -- Prevent adding a repeated node to the path.
        -- This ensures that no cycles occur.
        WHERE list_position(paths.path, node2id) IS NULL
    )
SELECT startNode, endNode, path
FROM paths
ORDER BY length(path), path;
```

startNode	endNode	path
1	3	[1, 3]
1	5	[1, 5]
1	5	[1, 3, 5]
1	8	[1, 3, 8]
1	10	[1, 3, 10]
1	3	[1, 5, 3]
1	4	[1, 5, 4]
1	8	[1, 5, 8]
1	4	[1, 3, 5, 4]
1	8	[1, 3, 5, 8]
1	8	[1, 5, 3, 8]
1	10	[1, 5, 3, 10]

Note that the result of this query is not restricted to shortest paths, e.g., for node 5, the results include paths `[1, 5]` and `[1, 3, 5]`.

Enumerate Unweighted Shortest Paths from a Node

In most cases, enumerating all paths is not practical or feasible. Instead, only the **(unweighted) shortest paths** are of interest. To find these, the second half of the `WITH RECURSIVE` query should be adjusted such that it only includes a node if it has not yet been visited. This is implemented by using a subquery that checks if any of the previous paths includes the node:

```
WITH RECURSIVE paths(startNode, endNode, path) AS (
        SELECT -- Define the path as the first edge of the traversal
            node1id AS startNode,
            node2id AS endNode,
            [node1id, node2id] AS path
        FROM edge
        WHERE startNode = 1
        UNION ALL
        SELECT -- Concatenate new edge to the path
            paths.startNode AS startNode,
            node2id AS endNode,
            array_append(path, node2id) AS path
        FROM paths
        JOIN edge ON paths.endNode = node1id
```

```
        -- Prevent adding a node that was visited previously by any path.
        -- This ensures that (1) no cycles occur and (2) only nodes that
        -- were not visited by previous (shorter) paths are added to a path.
        WHERE NOT EXISTS (
                FROM paths previous_paths
                WHERE list_contains(previous_paths.path, node2id)
              )
    )
SELECT startNode, endNode, path
FROM paths
ORDER BY length(path), path;
```

startNode	endNode	path
1	3	[1, 3]
1	5	[1, 5]
1	8	[1, 3, 8]
1	10	[1, 3, 10]
1	4	[1, 5, 4]
1	8	[1, 5, 8]

Enumerate Unweighted Shortest Paths between Two Nodes

`WITH RECURSIVE` can also be used to find **all (unweighted) shortest paths between two nodes**. To ensure that the recursive query is stopped as soon as we reach the end node, we use a window function which checks whether the end node is among the newly added nodes.

The following query returns all unweighted shortest paths between nodes 1 (start node) and 8 (end node):

```
WITH RECURSIVE paths(startNode, endNode, path, endReached) AS (
   SELECT -- Define the path as the first edge of the traversal
        node1id AS startNode,
        node2id AS endNode,
        [node1id, node2id] AS path,
        (node2id = 8) AS endReached
     FROM edge
     WHERE startNode = 1
   UNION ALL
   SELECT -- Concatenate new edge to the path
        paths.startNode AS startNode,
        node2id AS endNode,
        array_append(path, node2id) AS path,
        max(CASE WHEN node2id = 8 THEN 1 ELSE 0 END)
            OVER (ROWS BETWEEN UNBOUNDED PRECEDING
                           AND UNBOUNDED FOLLOWING) AS endReached
     FROM paths
     JOIN edge ON paths.endNode = node1id
    WHERE NOT EXISTS (
            FROM paths previous_paths
            WHERE list_contains(previous_paths.path, node2id)
          )
      AND paths.endReached = 0
)
SELECT startNode, endNode, path
FROM paths
WHERE endNode = 8
```

```
ORDER BY length(path), path;
```

startNode	endNode	path
1	8	[1, 3, 8]
1	8	[1, 5, 8]

Recursive CTEs with `USING KEY`

`USING KEY` alters the behavior of a regular recursive CTE.

In each iteration, a regular recursive CTE appends result rows to the union table, which ultimately defines the overall result of the CTE. In contrast, a CTE with `USING KEY` has the ability to update rows that have been placed in the union table in an earlier iteration: if the current iteration produces a row with key k, it replaces a row with the same key k in the union table (like a dictionary). If no such row exists in the union table yet, the new row is appended to the union table as usual.

This allows a CTE to exercise fine-grained control over the union table contents. Avoiding the append-only behavior can lead to significantly smaller union table sizes. This helps query runtime, memory consumption, and makes it feasible to access the union table while the iteration is still ongoing (this is impossible for regular recursive CTEs): in a CTE `WITH RECURSIVE T(...) USING KEY ...`, table `T` denotes the rows added by the last iteration (as is usual for recursive CTEs), while table `recurring.T` denotes the union table built so far. References to `recurring.T` allow for the elegant and idiomatic translation of rather complex algorithms into readable SQL code.

Example: `USING KEY`

This is a recursive CTE where `USING KEY` has a key column (a) and a payload column (b). The payload columns correspond to the columns to be overwritten. In the first iteration we have two different keys, `1` and `2`. These two keys will generate two new rows, `(1, 3)` and `(2, 4)`. In the next iteration we produce a new key, `3`, which generates a new row. We also generate the row `(2, 3)`, where `2` is a key that already exists from the previous iteration. This will overwrite the old payload `4` with the new payload `3`.

```
WITH RECURSIVE tbl(a, b) USING KEY (a) AS (
    SELECT a, b
    FROM (VALUES (1, 3), (2, 4)) t(a, b)
        UNION
    SELECT a + 1, b
    FROM tbl
    WHERE a < 3
)
SELECT *
FROM tbl;
```

a	b
1	3
2	3
3	3

Using VALUES

You can use the `VALUES` clause for the initial (anchor) part of the CTE:

```
WITH RECURSIVE tbl(a, b) USING KEY (a) AS (
    VALUES (1, 3), (2, 4)
        UNION
    SELECT a + 1, b
    FROM tbl
    WHERE a < 3
)
SELECT *
FROM tbl;
```

Example: `USING KEY` References Union Table

As well as using the union table as a dictionary, we can now reference it in queries. This allows you to use results from not just the previous iteration, but also earlier ones. This new feature makes certain algorithms easier to implement.

One example is the connected components algorithm. For each node, the algorithm determines the node with the lowest ID to which it is connected. To achieve this, we use the entries in the union table to track the lowest ID found for a node. If a new incoming row contains a lower ID, we update this value.

```
CREATE TABLE nodes (id INTEGER);
INSERT INTO nodes VALUES (1), (2), (3), (4), (5), (6), (7), (8);
CREATE TABLE edges (node1id INTEGER, node2id INTEGER);
INSERT INTO edges VALUES
    (1, 3), (2, 3), (3, 7), (7, 8), (5, 4),
    (6, 4);
```

```
WITH RECURSIVE cc(id, comp) USING KEY (id) AS (
    SELECT n.id, n.id AS comp
    FROM nodes AS n
        UNION
    (SELECT DISTINCT ON (u.id) u.id, v.comp
    FROM recurring.cc AS u, cc AS v, edges AS e
    WHERE ((e.node1id, e.node2id) = (u.id, v.id)
       OR (e.node2id, e.node1id) = (u.id, v.id))
      AND v.comp < u.comp
    ORDER BY u.id ASC, v.comp ASC)
)
TABLE cc
ORDER BY id;
```

id	comp
1	1
2	1
3	1
4	4
5	4
6	4
7	1
8	1

Limitations

DuckDB does not support mutually recursive CTEs. See the related issue and discussion in the DuckDB repository.

Syntax

WINDOW Clause

The WINDOW clause allows you to specify named windows that can be used within window functions. These are useful when you have multiple window functions, as they allow you to avoid repeating the same window clause.

Syntax

QUALIFY Clause

The QUALIFY clause is used to filter the results of WINDOW functions. This filtering of results is similar to how a HAVING clause filters the results of aggregate functions applied based on the GROUP BY clause.

The QUALIFY clause avoids the need for a subquery or WITH clause to perform this filtering (much like HAVING avoids a subquery). An example using a WITH clause instead of QUALIFY is included below the QUALIFY examples.

Note that this is filtering based on WINDOW functions, not necessarily based on the WINDOW clause. The WINDOW clause is optional and can be used to simplify the creation of multiple WINDOW function expressions.

The position of where to specify a QUALIFY clause is following the WINDOW clause in a SELECT statement (WINDOW does not need to be specified), and before the ORDER BY.

Examples

Each of the following examples produce the same output, located below.

Filter based on a window function defined in the QUALIFY clause:

```
SELECT
    schema_name,
    function_name,
    -- In this example the function_rank column in the select clause is for reference
    row_number() OVER (PARTITION BY schema_name ORDER BY function_name) AS function_rank
FROM duckdb_functions()
QUALIFY
    row_number() OVER (PARTITION BY schema_name ORDER BY function_name) < 3;
```

Filter based on a window function defined in the SELECT clause:

```
SELECT
    schema_name,
    function_name,
    row_number() OVER (PARTITION BY schema_name ORDER BY function_name) AS function_rank
FROM duckdb_functions()
QUALIFY
    function_rank < 3;
```

Filter based on a window function defined in the QUALIFY clause, but using the WINDOW clause:

```
SELECT
    schema_name,
    function_name,
    -- In this example the function_rank column in the select clause is for reference
    row_number() OVER my_window AS function_rank
FROM duckdb_functions()
```

```
WINDOW
    my_window AS (PARTITION BY schema_name ORDER BY function_name)
QUALIFY
    row_number() OVER my_window < 3;
```

Filter based on a window function defined in the `SELECT` clause, but using the `WINDOW` clause:

```
SELECT
    schema_name,
    function_name,
    row_number() OVER my_window AS function_rank
FROM duckdb_functions()
WINDOW
    my_window AS (PARTITION BY schema_name ORDER BY function_name)
QUALIFY
    function_rank < 3;
```

Equivalent query based on a `WITH` clause (without a `QUALIFY` clause):

```
WITH ranked_functions AS (
    SELECT
        schema_name,
        function_name,
        row_number() OVER (PARTITION BY schema_name ORDER BY function_name) AS function_rank
    FROM duckdb_functions()
)
SELECT
    *
FROM ranked_functions
WHERE
    function_rank < 3;
```

schema_name	function_name	function_rank
main	!__postfix	1
main	!~~	2
pg_catalog	col_description	1
pg_catalog	format_pg_type	2

Syntax

VALUES Clause

The `VALUES` clause is used to specify a fixed number of rows. The `VALUES` clause can be used as a stand-alone statement, as part of the `FROM` clause, or as input to an `INSERT INTO` statement.

Examples

Generate two rows and directly return them:

```
VALUES ('Amsterdam', 1), ('London', 2);
```

Generate two rows as part of a `FROM` clause, and rename the columns:

```
SELECT *
FROM (VALUES ('Amsterdam', 1), ('London', 2)) cities(name, id);
```

Generate two rows and insert them into a table:

```
INSERT INTO cities
VALUES ('Amsterdam', 1), ('London', 2);
```

Create a table directly from a `VALUES` clause:

```
CREATE TABLE cities AS
    SELECT *
    FROM (VALUES ('Amsterdam', 1), ('London', 2)) cities(name, id);
```

Syntax

FILTER Clause

The `FILTER` clause may optionally follow an aggregate function in a `SELECT` statement. This will filter the rows of data that are fed into the aggregate function in the same way that a `WHERE` clause filters rows, but localized to the specific aggregate function.

There are multiple types of situations where this is useful, including when evaluating multiple aggregates with different filters, and when creating a pivoted view of a dataset. `FILTER` provides a cleaner syntax for pivoting data when compared with the more traditional `CASE WHEN` approach discussed below.

Some aggregate functions also do not filter out `NULL` values, so using a `FILTER` clause will return valid results when at times the `CASE WHEN` approach will not. This occurs with the functions `first` and `last`, which are desirable in a non-aggregating pivot operation where the goal is to simply re-orient the data into columns rather than re-aggregate it. `FILTER` also improves `NULL` handling when using the `list` and `array_agg` functions, as the `CASE WHEN` approach will include `NULL` values in the list result, while the `FILTER` clause will remove them.

Examples

Return the following:

- The total number of rows
- The number of rows where `i <= 5`
- The number of rows where `i` is odd

```
SELECT
    count() AS total_rows,
    count() FILTER (i <= 5) AS lte_five,
    count() FILTER (i % 2 = 1) AS odds
FROM generate_series(1, 10) tbl(i);
```

total_rows	lte_five	odds
10	5	5

> Simply counting rows that satisfy a condition can also be achieved without `FILTER` clause, using the boolean `sum` aggregate function, e.g., `sum(i <= 5)`.

Different aggregate functions may be used, and multiple `WHERE` expressions are also permitted:

```
SELECT
    sum(i) FILTER (i <= 5) AS lte_five_sum,
    median(i) FILTER (i % 2 = 1) AS odds_median,
    median(i) FILTER (i % 2 = 1 AND i <= 5) AS odds_lte_five_median
FROM generate_series(1, 10) tbl(i);
```

lte_five_sum	odds_median	odds_lte_five_median
15	5.0	3.0

The `FILTER` clause can also be used to pivot data from rows into columns. This is a static pivot, as columns must be defined prior to runtime in SQL. However, this kind of statement can be dynamically generated in a host programming language to leverage DuckDB's SQL engine for rapid, larger than memory pivoting.

First generate an example dataset:

```
CREATE TEMP TABLE stacked_data AS
    SELECT
        i,
        CASE WHEN i <= rows * 0.25  THEN 2022
             WHEN i <= rows * 0.5   THEN 2023
             WHEN i <= rows * 0.75  THEN 2024
             WHEN i <= rows * 0.875 THEN 2025
             ELSE NULL
             END AS year
    FROM (
        SELECT
            i,
            count(*) OVER () AS rows
        FROM generate_series(1, 100_000_000) tbl(i)
    ) tbl;
```

"Pivot" the data out by year (move each year out to a separate column):

```
SELECT
    count(i) FILTER (year = 2022) AS "2022",
    count(i) FILTER (year = 2023) AS "2023",
    count(i) FILTER (year = 2024) AS "2024",
    count(i) FILTER (year = 2025) AS "2025",
    count(i) FILTER (year IS NULL) AS "NULLs"
FROM stacked_data;
```

This syntax produces the same results as the `FILTER` clauses above:

```
SELECT
    count(CASE WHEN year = 2022 THEN i END) AS "2022",
    count(CASE WHEN year = 2023 THEN i END) AS "2023",
    count(CASE WHEN year = 2024 THEN i END) AS "2024",
    count(CASE WHEN year = 2025 THEN i END) AS "2025",
    count(CASE WHEN year IS NULL THEN i END) AS "NULLs"
FROM stacked_data;
```

2022	2023	2024	2025	NULLs
25000000	25000000	25000000	12500000	12500000

However, the `CASE WHEN` approach will not work as expected when using an aggregate function that does not ignore `NULL` values. The `first` function falls into this category, so `FILTER` is preferred in this case.

"Pivot" the data out by year (move each year out to a separate column):

```
SELECT
    first(i) FILTER (year = 2022) AS "2022",
    first(i) FILTER (year = 2023) AS "2023",
    first(i) FILTER (year = 2024) AS "2024",
    first(i) FILTER (year = 2025) AS "2025",
    first(i) FILTER (year IS NULL) AS "NULLs"
FROM stacked_data;
```

2022	2023	2024	2025	NULLs
1474561	25804801	50749441	76431361	87500001

This will produce NULL values whenever the first evaluation of the CASE WHEN clause returns a NULL:

```
SELECT
    first(CASE WHEN year = 2022 THEN i END) AS "2022",
    first(CASE WHEN year = 2023 THEN i END) AS "2023",
    first(CASE WHEN year = 2024 THEN i END) AS "2024",
    first(CASE WHEN year = 2025 THEN i END) AS "2025",
    first(CASE WHEN year IS NULL THEN i END) AS "NULLs"
FROM stacked_data;
```

2022	2023	2024	2025	NULLs
1228801	NULL	NULL	NULL	NULL

Aggregate Function Syntax (Including FILTER Clause)

Set Operations

Set operations allow queries to be combined according to set operation semantics. Set operations refer to the UNION [ALL], INTERSECT [ALL] and EXCEPT [ALL] clauses. The vanilla variants use set semantics, i.e., they eliminate duplicates, while the variants with ALL use bag semantics.

Traditional set operations unify queries **by column position**, and require the to-be-combined queries to have the same number of input columns. If the columns are not of the same type, casts may be added. The result will use the column names from the first query.

DuckDB also supports UNION [ALL] BY NAME, which joins columns by name instead of by position. UNION BY NAME does not require the inputs to have the same number of columns. NULL values will be added in case of missing columns.

UNION

The UNION clause can be used to combine rows from multiple queries. The queries are required to return the same number of columns. Implicit casting to one of the returned types is performed to combine columns of different types where necessary. If this is not possible, the UNION clause throws an error.

Vanilla `UNION` (Set Semantics)

The vanilla `UNION` clause follows set semantics, therefore it performs duplicate elimination, i.e., only unique rows will be included in the result.

```
SELECT * FROM range(2) t1(x)
UNION
SELECT * FROM range(3) t2(x);
```

x
2
1
0

`UNION ALL` (Bag Semantics)

`UNION ALL` returns all rows of both queries following bag semantics, i.e., *without* duplicate elimination.

```
SELECT * FROM range(2) t1(x)
UNION ALL
SELECT * FROM range(3) t2(x);
```

x
0
1
0
1
2

`UNION [ALL] BY NAME`

The `UNION [ALL] BY NAME` clause can be used to combine rows from different tables by name, instead of by position. `UNION BY NAME` does not require both queries to have the same number of columns. Any columns that are only found in one of the queries are filled with `NULL` values for the other query.

Take the following tables for example:

```
CREATE TABLE capitals (city VARCHAR, country VARCHAR);
INSERT INTO capitals VALUES
    ('Amsterdam', 'NL'),
    ('Berlin', 'Germany');
CREATE TABLE weather (city VARCHAR, degrees INTEGER, date DATE);
INSERT INTO weather VALUES
    ('Amsterdam', 10, '2022-10-14'),
    ('Seattle', 8, '2022-10-12');
```

```
SELECT * FROM capitals
UNION BY NAME
SELECT * FROM weather;
```

city	country	degrees	date
Seattle	NULL	8	2022-10-12
Amsterdam	NL	NULL	NULL
Berlin	Germany	NULL	NULL
Amsterdam	NULL	10	2022-10-14

`UNION BY NAME` follows set semantics (therefore it performs duplicate elimination), whereas `UNION ALL BY NAME` follows bag semantics.

INTERSECT

The `INTERSECT` clause can be used to select all rows that occur in the result of **both** queries.

Vanilla INTERSECT (Set Semantics)

Vanilla `INTERSECT` performs duplicate elimination, so only unique rows are returned.

```
SELECT * FROM range(2) t1(x)
INTERSECT
SELECT * FROM range(6) t2(x);
```

x
0
1

INTERSECT ALL (Bag Semantics)

`INTERSECT ALL` follows bag semantics, so duplicates are returned.

```
SELECT unnest([5, 5, 6, 6, 6, 6, 7, 8]) AS x
INTERSECT ALL
SELECT unnest([5, 6, 6, 7, 7, 9]);
```

x
5
6
6
7

EXCEPT

The EXCEPT clause can be used to select all rows that **only** occur in the left query.

Vanilla EXCEPT (Set Semantics)

Vanilla EXCEPT follows set semantics, therefore, it performs duplicate elimination, so only unique rows are returned.

```
SELECT * FROM range(5) t1(x)
EXCEPT
SELECT * FROM range(2) t2(x);
```

x
2
3
4

EXCEPT ALL (Bag Semantics)

EXCEPT ALL uses bag semantics:

```
SELECT unnest([5, 5, 6, 6, 6, 6, 7, 8]) AS x
EXCEPT ALL
SELECT unnest([5, 6, 6, 7, 7, 9]);
```

x
5
8
6
6

Syntax

Prepared Statements

DuckDB supports prepared statements where parameters are substituted when the query is executed. This can improve readability and is useful for preventing SQL injections.

Syntax

There are three syntaxes for denoting parameters in prepared statements: auto-incremented (`?`), positional (`$1`), and named (`$param`). Note that not all clients support all of these syntaxes, e.g., the JDBC client only supports auto-incremented parameters in prepared statements.

Example Data Set

In the following, we introduce the three different syntaxes and illustrate them with examples using the following table.

```
CREATE TABLE person (name VARCHAR, age BIGINT);
INSERT INTO person VALUES ('Alice', 37), ('Ana', 35), ('Bob', 41), ('Bea', 25);
```

In our example query, we'll look for people whose name starts with a B and are at least 40 years old. This will return a single row `<'Bob', 41>`.

Auto-Incremented Parameters: ?

DuckDB support using prepared statements with auto-incremented indexing, i.e., the position of the parameters in the query corresponds to their position in the execution statement. For example:

```
PREPARE query_person AS
    SELECT *
    FROM person
    WHERE starts_with(name, ?)
      AND age >= ?;
```

Using the CLI client, the statement is executed as follows.

```
EXECUTE query_person('B', 40);
```

Positional Parameters: $1

Prepared statements can use positional parameters, where parameters are denoted with an integer (`$1`, `$2`). For example:

```
PREPARE query_person AS
    SELECT *
    FROM person
    WHERE starts_with(name, $2)
      AND age >= $1;
```

Using the CLI client, the statement is executed as follows. Note that the first parameter corresponds to `$1`, the second to `$2`, and so on.

```
EXECUTE query_person(40, 'B');
```

Named Parameters: `$parameter`

DuckDB also supports names parameters where parameters are denoted with `$parameter_name`. For example:

```
PREPARE query_person AS
    SELECT *
    FROM person
    WHERE starts_with(name, $name_start_letter)
      AND age >= $minimum_age;
```

Using the CLI client, the statement is executed as follows.

```
EXECUTE query_person(name_start_letter := 'B', minimum_age := 40);
```

Dropping Prepared Statements: DEALLOCATE

To drop a prepared statement, use the `DEALLOCATE` statement:

```
DEALLOCATE query_person;
```

Alternatively, use:

```
DEALLOCATE PREPARE query_person;
```

Data Types

Data Types

General-Purpose Data Types

The table below shows all the built-in general-purpose data types. The alternatives listed in the aliases column can be used to refer to these types as well, however, note that the aliases are not part of the SQL standard and hence might not be accepted by other database engines.

Name	Aliases	Description
`BIGINT`	`INT8`, `LONG`	Signed eight-byte integer
`BIT`	`BITSTRING`	String of 1s and 0s
`BLOB`	`BYTEA`, `BINARY`, `VARBINARY`	Variable-length binary data
`BOOLEAN`	`BOOL`, `LOGICAL`	Logical Boolean (`true` / `false`)
`DATE`		Calendar date (year, month day)
`DECIMAL(prec, scale)`	`NUMERIC(prec, scale)`	Fixed-precision number with the given width (precision) and scale, defaults to `prec = 18` and `scale = 3`
`DOUBLE`	`FLOAT8`,	Double precision floating-point number (8 bytes)
`FLOAT`	`FLOAT4`, `REAL`	Single precision floating-point number (4 bytes)
`HUGEINT`		Signed sixteen-byte integer
`INTEGER`	`INT4`, `INT`, `SIGNED`	Signed four-byte integer
`INTERVAL`		Date / time delta
`JSON`		JSON object (via the `json` extension)
`SMALLINT`	`INT2`, `SHORT`	Signed two-byte integer
`TIME`		Time of day (no time zone)
`TIMESTAMP WITH TIME ZONE`	`TIMESTAMPTZ`	Combination of time and date that uses the current time zone
`TIMESTAMP`	`DATETIME`	Combination of time and date
`TINYINT`	`INT1`	Signed one-byte integer
`UBIGINT`		Unsigned eight-byte integer
`UHUGEINT`		Unsigned sixteen-byte integer
`UINTEGER`		Unsigned four-byte integer
`USMALLINT`		Unsigned two-byte integer
`UTINYINT`		Unsigned one-byte integer
`UUID`		UUID data type

Name	Aliases	Description
VARCHAR	CHAR, BPCHAR, TEXT, STRING	Variable-length character string

Implicit and explicit typecasting is possible between numerous types, see the Typecasting page for details.

Nested / Composite Types

DuckDB supports five nested data types: `ARRAY`, `LIST`, `MAP`, `STRUCT`, and `UNION`. Each supports different use cases and has a different structure.

Name	Description	Rules when used in a column	Build from values	Define in DDL/CREATE
ARRAY	An ordered, fixed-length sequence of data values of the same type.	Each row must have the same data type within each instance of the ARRAY and the same number of elements.	`[1, 2, 3]`	`INTEGER[3]`
LIST	An ordered sequence of data values of the same type.	Each row must have the same data type within each instance of the LIST, but can have any number of elements.	`[1, 2, 3]`	`INTEGER[]`
MAP	A dictionary of multiple named values, each key having the same type and each value having the same type. Keys and values can be any type and can be different types from one another.	Rows may have different keys.	`map([1, 2], ['a', 'b'])`	`MAP(INTEGER, VARCHAR)`
STRUCT	A dictionary of multiple named values, where each key is a string, but the value can be a different type for each key.	Each row must have the same keys.	`{'i': 42, 'j': 'a'}`	`STRUCT(i INTEGER, j VARCHAR)`
UNION	A union of multiple alternative data types, storing one of them in each value at a time. A union also contains a discriminator "tag" value to inspect and access the currently set member type.	Rows may be set to different member types of the union.	`union_value(num := 2)`	`UNION(num INTEGER, text VARCHAR)`

Rules for Case Sensitivity

The keys of MAPs are case-sensitive, while keys of UNIONs and STRUCTs are case-insensitive. For examples, see the Rules for Case Sensitivity section.

Updating Values of Nested Types

When performing *updates* on values of nested types, DuckDB performs a *delete* operation followed by an *insert* operation. When used in a table with ART indexes (either via explicit indexes or primary keys/unique constraints), this can lead to unexpected constraint violations.

Nesting

`ARRAY`, `LIST`, `MAP`, `STRUCT`, and `UNION` types can be arbitrarily nested to any depth, so long as the type rules are observed.

Struct with `LIST`s:

```
SELECT {'birds': ['duck', 'goose', 'heron'], 'aliens': NULL, 'amphibians': ['frog', 'toad']};
```

Struct with list of `MAP`s:

```
SELECT {'test': [MAP([1, 5], [42.1, 45]), MAP([1, 5], [42.1, 45])]};
```

A list of `UNION`s:

```
SELECT [union_value(num := 2), union_value(str := 'ABC')::UNION(str VARCHAR, num INTEGER)];
```

Performance Implications

The choice of data types can have a strong effect on performance. Please consult the Performance Guide for details.

Array Type

An ARRAY column stores fixed-sized arrays. All fields in the column must have the same length and the same underlying type. Arrays are typically used to store arrays of numbers, but can contain any uniform data type, including ARRAY, `LIST` and `STRUCT` types.

Arrays can be used to store vectors such as word embeddings or image embeddings.

To store variable-length lists, use the `LIST` type. See the data types overview for a comparison between nested data types.

> The ARRAY type in PostgreSQL allows variable-length fields. DuckDB's ARRAY type is fixed-length.

Creating Arrays

Arrays can be created using the `array_value(expr, ...)` function.

Construct with the `array_value` function:

```
SELECT array_value(1, 2, 3);
```

You can always implicitly cast an array to a list (and use list functions, like `list_extract`, `[i]`):

```
SELECT array_value(1, 2, 3)[2];
```

You can cast from a list to an array (the dimensions have to match):

```
SELECT [3, 2, 1]::INTEGER[3];
```

Arrays can be nested:

```
SELECT array_value(array_value(1, 2), array_value(3, 4), array_value(5, 6));
```

Arrays can store structs:

```
SELECT array_value({'a': 1, 'b': 2}, {'a': 3, 'b': 4});
```

Defining an Array Field

Arrays can be created using the `<TYPE_NAME>[<LENGTH>]{:.language-sql .highlight}` syntax. For example, to create an array field for 3 integers, run:

```
CREATE TABLE array_table (id INTEGER, arr INTEGER[3]);
INSERT INTO array_table VALUES (10, [1, 2, 3]), (20, [4, 5, 6]);
```

Retrieving Values from Arrays

Retrieving one or more values from an array can be accomplished using brackets and slicing notation, or through list functions like `list_extract` and `array_extract`. Using the example in Defining an Array Field.

The following queries for extracting the second element of an array are equivalent:

```
SELECT id, arr[1] AS element FROM array_table;
SELECT id, list_extract(arr, 1) AS element FROM array_table;
SELECT id, array_extract(arr, 1) AS element FROM array_table;
```

id	element
10	1
20	4

Using the slicing notation returns a `LIST`:

```
SELECT id, arr[1:2] AS elements FROM array_table;
```

id	elements
10	[1, 2]
20	[4, 5]

Functions

All `LIST` functions work with the ARRAY type. Additionally, several ARRAY-native functions are also supported. See the ARRAY functions.

Examples

Create sample data:

```
CREATE TABLE x (i INTEGER, v FLOAT[3]);
CREATE TABLE y (i INTEGER, v FLOAT[3]);
INSERT INTO x VALUES (1, array_value(1.0::FLOAT, 2.0::FLOAT, 3.0::FLOAT));
INSERT INTO y VALUES (1, array_value(2.0::FLOAT, 3.0::FLOAT, 4.0::FLOAT));
```

Compute cross product:

```
SELECT array_cross_product(x.v, y.v)
FROM x, y
WHERE x.i = y.i;
```

Compute cosine similarity:

```
SELECT array_cosine_similarity(x.v, y.v)
FROM x, y
WHERE x.i = y.i;
```

Ordering

The ordering of ARRAY instances is defined using a lexicographical order. NULL values compare greater than all other values and are considered equal to each other.

See Also

For more functions, see List Functions.

Bitstring Type

Name	Aliases	Description
BITSTRING	BIT	Variable-length strings of 1s and 0s

Bitstrings are strings of 1s and 0s. The bit type data is of variable length. A bitstring value requires 1 byte for each group of 8 bits, plus a fixed amount to store some metadata.

By default bitstrings will not be padded with zeroes. Bitstrings can be very large, having the same size restrictions as BLOBs.

Creating a Bitstring

A string encoding a bitstring can be cast to a BITSTRING:

```
SELECT '101010'::BITSTRING AS b;
```

b
101010

Create a BITSTRING with predefined length is possible with the `bitstring` function. The resulting bitstring will be left-padded with zeroes.

```
SELECT bitstring('0101011', 12) AS b;
```

b
000000101011

Numeric values (integer and float values) can also be converted to a BITSTRING via casting. For example:

```
SELECT 123::BITSTRING AS b;
```

b
00000000000000000000000001111011

Functions

See Bitstring Functions.

Blob Type

Name	Aliases	Description
BLOB	BYTEA, BINARY, VARBINARY	Variable-length binary data

The blob (**B**inary **L**arge **OB**ject) type represents an arbitrary binary object stored in the database system. The blob type can contain any type of binary data with no restrictions. What the actual bytes represent is opaque to the database system.

Create a BLOB value with a single byte (170):

```
SELECT '\xAA'::BLOB;
```

Create a BLOB value with three bytes (170, 171, 172):

```
SELECT '\xAA\xAB\xAC'::BLOB;
```

Create a BLOB value with two bytes (65, 66):

```
SELECT 'AB'::BLOB;
```

Blobs are typically used to store non-textual objects that the database does not provide explicit support for, such as images. While blobs can hold objects up to 4 GB in size, typically it is not recommended to store very large objects within the database system. In many situations it is better to store the large file on the file system, and store the path to the file in the database system in a VARCHAR field.

Functions

See Blob Functions.

Boolean Type

Name	Aliases	Description
BOOLEAN	BOOL	Logical Boolean (`true` / `false`)

The BOOLEAN type represents a statement of truth (“true” or “false”). In SQL, the BOOLEAN field can also have a third state “unknown” which is represented by the SQL NULL value.

Select the three possible values of a BOOLEAN column:

```
SELECT true, false, NULL::BOOLEAN;
```

Boolean values can be explicitly created using the literals `true` and `false`. However, they are most often created as a result of comparisons or conjunctions. For example, the comparison `i > 10` results in a Boolean value. Boolean values can be used in the `WHERE` and `HAVING` clauses of a SQL statement to filter out tuples from the result. In this case, tuples for which the predicate evaluates to `true` will pass the filter, and tuples for which the predicate evaluates to `false` or `NULL` will be filtered out. Consider the following example:

Create a table with the values 5, 15 and `NULL`:

```
CREATE TABLE integers (i INTEGER);
INSERT INTO integers VALUES (5), (15), (NULL);
```

Select all entries where `i > 10`:

```
SELECT * FROM integers WHERE i > 10;
```

In this case 5 and `NULL` are filtered out (`5 > 10` is `false` and `NULL > 10` is `NULL`):

i
15

Conjunctions

The `AND` / `OR` conjunctions can be used to combine Boolean values.

Below is the truth table for the `AND` conjunction (i.e., `x AND y`).

X	X AND true	X AND false	X AND NULL
true	true	false	NULL
false	false	false	false
NULL	NULL	false	NULL

Below is the truth table for the `OR` conjunction (i.e., `x OR y`).

X	X OR true	X OR false	X OR NULL
true	true	true	true
false	true	false	NULL
NULL	true	NULL	NULL

Expressions

See Logical Operators and Comparison Operators.

Date Types

Name	Aliases	Description
DATE		Calendar date (year, month day)

A date specifies a combination of year, month and day. DuckDB follows the SQL standard's lead by counting dates exclusively in the Gregorian calendar, even for years before that calendar was in use. Dates can be created using the `DATE` keyword, where the data must be formatted according to the ISO 8601 format (`YYYY-MM-DD`).

```
SELECT DATE '1992-09-20';
```

Special Values

There are also three special date values that can be used on input:

Input string	Description
epoch	1970-01-01 (Unix system day zero)
infinity	Later than all other dates
-infinity	Earlier than all other dates

The values `infinity` and `-infinity` are specially represented inside the system and will be displayed unchanged; but `epoch` is simply a notational shorthand that will be converted to the date value when read.

```
SELECT
    '-infinity'::DATE AS negative,
    'epoch'::DATE AS epoch,
    'infinity'::DATE AS positive;
```

negative	epoch	positive
-infinity	1970-01-01	infinity

Functions

See Date Functions.

Enum Data Type

Name	Description
ENUM	Dictionary representing all possible string values of a column

The enum type represents a dictionary data structure with all possible unique values of a column. For example, a column storing the days of the week can be an enum holding all possible days. Enums are particularly interesting for string columns with low cardinality (i.e., fewer distinct values). This is because the column only stores a numerical reference to the string in the enum dictionary, resulting in immense savings in disk storage and faster query performance.

Enum Definition

Enum types are created from either a hardcoded set of values or from a select statement that returns a single column of VARCHARs. The set of values in the select statement will be deduplicated, but if the enum is created from a hardcoded set there may not be any duplicates.

Create enum using hardcoded values:

```
CREATE TYPE <enum_name> AS ENUM (<value_1>, <value_2>, ...);
```

Create enum using a SELECT statement that returns a single column of VARCHARs:

```
CREATE TYPE <enum_name> AS ENUM (<select_expression>);
```

Enums can also be created on the fly during casting:

```
SELECT 'some_string'::ENUM (<value_1>, <value_2>, ...);
```

Examples

Creates new user defined type mood as an enum:

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
```

Create an enum from a select statement. First create an example table of values:

```
CREATE TABLE my_inputs AS
    FROM (VALUES ('duck'), ('duck'), ('goose')) t(my_varchar);
```

Create an anonymous enum value on the fly:

```
SELECT 'happy'::ENUM ('sad', 'ok', 'happy');
```

This statement will fail since enums cannot hold NULL values:

```
CREATE TYPE breed AS ENUM ('maltese', NULL);
```

This statement will fail since enum values must be unique:

```
CREATE TYPE breed AS ENUM ('maltese', 'maltese');
```

Create an enum using the unique string values in the `my_varchar` column:

```
CREATE TYPE birds AS ENUM (SELECT my_varchar FROM my_inputs);
```

Show the available values in the `birds` enum using the `enum_range` function:

```
SELECT enum_range(NULL::birds) AS my_enum_range;
```

my_enum_range
[duck, goose]

Enum Usage

After an enum has been created, it can be used anywhere a standard built-in type is used. For example, we can create a table with a column that references the enum.

Creates a table `person`, with attributes `name` (string type) and `current_mood` (mood type):

```
CREATE TABLE person (
    name TEXT,
    current_mood mood
);
```

Inserts tuples in the `person` table:

```
INSERT INTO person
VALUES ('Pedro', 'happy'), ('Mark', NULL), ('Pagliacci', 'sad'), ('Mr. Mackey', 'ok');
```

The following query will fail since the mood type does not have `quackity-quack` value.

```
INSERT INTO person
VALUES ('Hannes', 'quackity-quack');
```

The string `sad` is cast to the type `mood`, returning a numerical reference value. This makes the comparison a numerical comparison instead of a string comparison.

```
SELECT *
FROM person
WHERE current_mood = 'sad';
```

name	current_mood
Pagliacci	sad

If you are importing data from a file, you can create an enum for a VARCHAR column before importing. Given this, the following subquery selects automatically selects only distinct values:

```
CREATE TYPE mood AS ENUM (SELECT mood FROM 'path/to/file.csv');
```

Then you can create a table with the enum type and import using any data import statement:

```
CREATE TABLE person (name TEXT, current_mood mood);
COPY person FROM 'path/to/file.csv';
```

Enums vs. Strings

DuckDB enums are automatically cast to VARCHAR types whenever necessary. This characteristic allows for enum columns to be used in any VARCHAR function. In addition, it also allows for comparisons between different enum columns, or an enum and a VARCHAR column.

For example:

Regexp_matches is a function that takes a VARCHAR, hence current_mood is cast to VARCHAR:

```
SELECT regexp_matches(current_mood, '.*a.*') AS contains_a
FROM person;
```

contains_a
true
NULL
true
false

Create a new mood and table:

```
CREATE TYPE new_mood AS ENUM ('happy', 'anxious');
CREATE TABLE person_2 (
    name text,
    current_mood mood,
    future_mood new_mood,
    past_mood VARCHAR
);
```

Since the `current_mood` and `future_mood` columns are constructed on different enum types, DuckDB will cast both enums to strings and perform a string comparison:

```
SELECT *
FROM person_2
WHERE current_mood = future_mood;
```

When comparing the `past_mood` column (string), DuckDB will cast the `current_mood` enum to VARCHAR and perform a string comparison:

```
SELECT *
FROM person_2
WHERE current_mood = past_mood;
```

Enum Removal

Enum types are stored in the catalog, and a catalog dependency is added to each table that uses them. It is possible to drop an enum from the catalog using the following command:

```
DROP TYPE <enum_name>;
```

Currently, it is possible to drop enums that are used in tables without affecting the tables.

> **Warning.** This behavior of the enum removal feature is subject to change. In future releases, it is expected that any dependent columns must be removed before dropping the enum, or the enum must be dropped with the additional CASCADE parameter.

Comparison of Enums

Enum values are compared according to their order in the enum's definition. For example:

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
```

```
SELECT 'sad'::mood < 'ok'::mood AS comp;
```

comp
true

```
SELECT unnest(['ok'::mood, 'happy'::mood, 'sad'::mood]) AS m
ORDER BY m;
```

m
sad
ok
happy

Functions

See Enum Functions.

Interval Type

`INTERVAL`s represent periods of time that can be added to or subtracted from `DATE`, `TIMESTAMP`, `TIMESTAMPTZ`, or `TIME` values.

Name	Description
`INTERVAL`	Period of time

An `INTERVAL` can be constructed by providing amounts together with units. Units that aren't *months*, *days*, or *microseconds* are converted to equivalent amounts in the next smaller of these three basis units.

```
SELECT
    INTERVAL 1 YEAR, -- single unit using YEAR keyword; stored as 12 months
    INTERVAL (random() * 10) YEAR, -- parentheses necessary for variable amounts;
                                   -- stored as integer number of months
    INTERVAL '1 month 1 day', -- string type necessary for multiple units; stored as (1 month, 1 day)
    '16 months'::INTERVAL, -- string cast supported; stored as 16 months
    '48:00:00'::INTERVAL, -- HH::MM::SS string supported; stored as (48 * 60 * 60 * 1e6 microseconds)
;
```

> **Warning.** Decimal values are truncated to integers when used with unit keywords (unless the unit is `SECONDS` or `MILLISECONDS`).
>
> ```
> SELECT INTERVAL '1.5' YEARS;
> -- Returns 12 months; equivalent to `to_years(CAST(trunc(1.5) AS INTEGER))`
> ```
>
> For more precision, include the unit in the string or use a more granular unit; e.g., `INTERVAL '1.5 years'` or `INTERVAL 18 MONTHS`.

Three basis units are necessary because a month does not correspond to a fixed amount of days (February has fewer days than March) and a day doesn't correspond to a fixed amount of microseconds. The division into components makes the `INTERVAL` class suitable for adding or subtracting specific time units to a date. For example, we can generate a table with the first day of every month using the following SQL query:

```
SELECT DATE '2000-01-01' + INTERVAL (i) MONTH
FROM range(12) t(i);
```

When `INTERVAL`s are deconstructed via the `datepart` function, the *months* component is additionally split into years and months, and the *microseconds* component is split into hours, minutes, and microseconds. The *days* component is not split into additional units. To demonstrate this, the following query generates an `INTERVAL` called `period` by summing random amounts of the three basis units. It then extracts the aforementioned six parts from `period`, adds them back together, and confirms that the result is always equal to the original `period`.

```
SELECT
    period = list_reduce(
        [INTERVAL (datepart(part, period) || part) FOR part IN
             ['year', 'month', 'day', 'hour', 'minute', 'microsecond']
        ],
        (i1, i2) -> i1 + i2
    ) -- always true
FROM (
    VALUES (
        INTERVAL (random() * 123_456_789_123) MICROSECONDS
```

```
        + INTERVAL (random() * 12_345) DAYS
        + INTERVAL (random() * 12_345) MONTHS
    )
) _(period);
```

> **Warning.** The *microseconds* component is split only into hours, minutes, and microseconds, rather than hours, minutes, *seconds*, and microseconds.

Additionally, the amounts of centuries, decades, quarters, seconds, and milliseconds in an `INTERVAL`, rounded down to the nearest integer, can be extracted via the `datepart` function. However, these components are not required to reassemble the original `INTERVAL`. In fact, if the previous query additionally extracted decades or seconds, then the sum of extracted parts would generally be larger than the original `period` since this would double count the months and microseconds components, respectively.

> All units use 0-based indexing, except for quarters, which use 1-based indexing.

For example:

```
SELECT
    datepart('decade', INTERVAL 12 YEARS), -- returns 1
    datepart('year', INTERVAL 12 YEARS), -- returns 12
    datepart('second', INTERVAL 1_234 MILLISECONDS), -- returns 1
    datepart('microsecond', INTERVAL 1_234 MILLISECONDS), -- returns 1_234_000
;
```

Arithmetic with Timestamps, Dates and Intervals

`INTERVAL`s can be added to and subtracted from `TIMESTAMP`s, `TIMESTAMPTZ`s, `DATE`s, and `TIME`s using the + and − operators.

```
SELECT
    DATE '2000-01-01' + INTERVAL 1 YEAR,
    TIMESTAMP '2000-01-01 01:33:30' - INTERVAL '1 month 13 hours',
    TIME '02:00:00' - INTERVAL '3 days 23 hours', -- wraps; equals TIME '03:00:00'
;
```

> Adding an `INTERVAL` to a `DATE` returns a `TIMESTAMP` even when the `INTERVAL` has no microseconds component. The result is the same as if the `DATE` was cast to a `TIMESTAMP` (which sets the time component to `00:00:00`) before adding the `INTERVAL`.

Conversely, subtracting two `TIMESTAMP`s or two `TIMESTAMPTZ`s from one another creates an `INTERVAL` describing the difference between the timestamps with only the *days and microseconds* components. For example:

```
SELECT
    TIMESTAMP '2000-02-06 12:00:00' - TIMESTAMP '2000-01-01 11:00:00', -- 36 days 1 hour
    TIMESTAMP '2000-02-01' + (TIMESTAMP '2000-02-01' - TIMESTAMP '2000-01-01'), -- '2000-03-03', NOT
    '2000-03-01'
;
```

Subtracting two `DATE`s from one another does not create an `INTERVAL` but rather returns the number of days between the given dates as integer value.

> **Warning.** Extracting a component of the `INTERVAL` difference between two `TIMESTAMP`s is not equivalent to computing the number of partition boundaries between the two `TIMESTAMP`s for the corresponding unit, as computed by the `datediff` function:
>
> ```
> SELECT
>     datediff('day', TIMESTAMP '2020-01-01 01:00:00', TIMESTAMP '2020-01-02 00:00:00'), -- 1
>     datepart('day', TIMESTAMP '2020-01-02 00:00:00' - TIMESTAMP '2020-01-01 01:00:00'), -- 0
> ;
> ```

Equality and Comparison

For equality and ordering comparisons only, the total number of microseconds in an `INTERVAL` is computed by converting the days basis unit to `24 * 60 * 60 * 1e6` microseconds and the months basis unit to 30 days, or `30 * 24 * 60 * 60 * 1e6` microseconds.

As a result, `INTERVAL`s can compare equal even when they are functionally different, and the ordering of `INTERVAL`s is not always preserved when they are added to dates or timestamps.

For example:

- `INTERVAL 30 DAYS = INTERVAL 1 MONTH`
- but `DATE '2020-01-01' + INTERVAL 30 DAYS != DATE '2020-01-01' + INTERVAL 1 MONTH`.

and

- `INTERVAL '30 days 12 hours' > INTERVAL 1 MONTH`
- but `DATE '2020-01-01' + INTERVAL '30 days 12 hours' < DATE '2020-01-01' + INTERVAL 1 MONTH`.

Functions

See the Date Part Functions page for a list of available date parts for use with an `INTERVAL`.

See the Interval Operators page for functions that operate on intervals.

List Type

A `LIST` column encodes lists of values. Fields in the column can have values with different lengths, but they must all have the same underlying type. `LIST`s are typically used to store arrays of numbers, but can contain any uniform data type, including other `LIST`s and `STRUCT`s.

`LIST`s are similar to PostgreSQL's ARRAY type. DuckDB uses the `LIST` terminology, but some `array_` functions are provided for PostgreSQL compatibility.

See the data types overview for a comparison between nested data types.

> For storing fixed-length lists, DuckDB uses the ARRAY type.

Creating Lists

Lists can be created using the `list_value(expr, ...)` function or the equivalent bracket notation `[expr, ...]`. The expressions can be constants or arbitrary expressions. To create a list from a table column, use the `list` aggregate function.

List of integers:

```
SELECT [1, 2, 3];
```

List of strings with a `NULL` value:

```
SELECT ['duck', 'goose', NULL, 'heron'];
```

List of lists with `NULL` values:

```
SELECT [['duck', 'goose', 'heron'], NULL, ['frog', 'toad'], []];
```

Create a list with the list_value function:

```
SELECT list_value(1, 2, 3);
```

Create a table with an `INTEGER` list column and a `VARCHAR` list column:

```
CREATE TABLE list_table (int_list INTEGER[], varchar_list VARCHAR[]);
```

Retrieving from Lists

Retrieving one or more values from a list can be accomplished using brackets and slicing notation, or through list functions like `list_extract`. Multiple equivalent functions are provided as aliases for compatibility with systems that refer to lists as arrays. For example, the function `array_slice`.

Example	Result
SELECT ['a', 'b', 'c'][3]	'c'
SELECT ['a', 'b', 'c'][-1]	'c'
SELECT ['a', 'b', 'c'][2 + 1]	'c'
SELECT list_extract(['a', 'b', 'c'], 3)	'c'
SELECT ['a', 'b', 'c'][1:2]	['a', 'b']
SELECT ['a', 'b', 'c'][:2]	['a', 'b']
SELECT ['a', 'b', 'c'][-2:]	['b', 'c']
SELECT list_slice(['a', 'b', 'c'], 2, 3)	['b', 'c']

Comparison and Ordering

The `LIST` type can be compared using all the comparison operators. These comparisons can be used in logical expressions such as `WHERE` and `HAVING` clauses, and return BOOLEAN values.

The `LIST` ordering is defined positionally using the following rules, where `min_len = min(len(l1), len(l2))`.

- **Equality.** `l1` and `l2` are equal, if for each `i` in `[1, min_len]`: `l1[i] = l2[i]`.
- **Less Than**. For the first index `i` in `[1, min_len]` where `l1[i] != l2[i]`: If `l1[i] < l2[i]`, `l1` is less than `l2`.

`NULL` values are compared following PostgreSQL's semantics. Lower nesting levels are used for tie-breaking.

Here are some queries returning `true` for the comparison.

```
SELECT [1, 2] < [1, 3] AS result;
```

```
SELECT [[1], [2, 4, 5]] < [[2]] AS result;
```

```
SELECT [ ] < [1] AS result;
```

These queries return `false`.

```
SELECT [ ] < [ ] AS result;
```

```
SELECT [1, 2] < [1] AS result;
```

These queries return `NULL`.

```
SELECT [1, 2] < [1, NULL, 4] AS result;
```

Functions

See List Functions.

Literal Types

DuckDB has special literal types for representing `NULL`, integer and string literals in queries. These have their own binding and conversion rules.

> Prior to DuckDB version 0.10.0, integer and string literals behaved identically to the `INTEGER` and `VARCHAR` types.

Null Literals

The `NULL` literal is denoted with the keyword `NULL`. The `NULL` literal can be implicitly converted to any other type.

Integer Literals

Integer literals are denoted as a sequence of one or more digits. At runtime, these result in values of the `INTEGER_LITERAL` type. `INTEGER_LITERAL` types can be implicitly converted to any integer type in which the value fits. For example, the integer literal `42` can be implicitly converted to a `TINYINT`, but the integer literal `1000` cannot be.

Other Numeric Literals

Non-integer numeric literals can be denoted with decimal notation, using the period character (`.`) to separate the integer part and the decimal part of the number. Either the integer part or the decimal part may be omitted:

```
SELECT 1.5;          -- 1.5
SELECT .50;          -- 0.5
SELECT 2.;           -- 2.0
```

Non-integer numeric literals can also be denoted using *E notation*. In E notation, an integer or decimal literal is followed by and exponential part, which is denoted by e or E, followed by a literal integer indicating the exponent. The exponential part indicates that the preceding value should be multiplied by 10 raised to the power of the exponent:

```
SELECT 1e2;           -- 100
SELECT 6.02214e23;    -- Avogadro's constant
SELECT 1e-10;         -- 1 ångström
```

Underscores in Numeric Literals

DuckDB's SQL dialect allows using the underscore character _ in numeric literals as an optional separator. The rules for using underscores are as follows:

- Underscores are allowed in integer, decimal, hexadecimal and binary notation.
- Underscores can not be the first or last character in a literal.
- Underscores have to have an integer/numeric part on either side of them, i.e., there can not be multiple underscores in a row and not immediately before/after a decimal or exponent.

Examples:

```
SELECT 100_000_000;          -- 100000000
SELECT '0xFF_FF'::INTEGER;   -- 65535
SELECT 1_2.1_2E0_1;          -- 121.2
SELECT '0b0_1_0_1'::INTEGER; -- 5
```

String Literals

String literals are delimited using single quotes (', apostrophe) and result in `STRING_LITERAL` values. Note that double quotes (") cannot be used as string delimiter character: instead, double quotes are used to delimit quoted identifiers.

Implicit String Literal Concatenation

Consecutive single-quoted string literals separated only by whitespace that contains at least one newline are implicitly concatenated:

```
SELECT 'Hello'
    ' '
    'World' AS greeting;
```

is equivalent to:

```
SELECT 'Hello'
    || ' '
    || 'World' AS greeting;
```

They both return the following result:

greeting
Hello World

Note that implicit concatenation only works if there is at least one newline between the literals. Using adjacent string literals separated by whitespace without a newline results in a syntax error:

```
SELECT 'Hello' ' ' 'World' AS greeting;
```

```
Parser Error:
syntax error at or near "' '"

LINE 1: SELECT 'Hello' ' ' 'World' AS greeting;
                       ^
```

Also note that implicit concatenation only works with single-quoted string literals, and does not work with other kinds of string values.

Implicit String Conversion

`STRING_LITERAL` instances can be implicitly converted to *any* other type.

For example, we can compare string literals with dates:

```
SELECT d > '1992-01-01' AS result
FROM (VALUES (DATE '1992-01-01')) t(d);
```

result
false

However, we cannot compare VARCHAR values with dates.

```
SELECT d > '1992-01-01'::VARCHAR
FROM (VALUES (DATE '1992-01-01')) t(d);
```

```
Binder Error:
Cannot compare values of type DATE and type VARCHAR - an explicit cast is required
```

Escape String Literals

To escape a single quote (apostrophe) character in a string literal, use `''`. For example, `SELECT '''' AS s` returns `'`.

To enable some common escape sequences, such as `\n` for the newline character, prefix a string literal with `e` (or `E`).

```
SELECT e'Hello\nworld' AS msg;
```

msg
varchar
Hello\nworld

The following backslash escape sequences are supported:

Escape sequence	Name	ASCII code
`\b`	backspace	8
`\f`	form feed	12
`\n`	newline	10
`\r`	carriage return	13
`\t`	tab	9

Dollar-Quoted String Literals

DuckDB supports dollar-quoted string literals, which are surrounded by double-dollar symbols (`$$`):

```
SELECT $$Hello
world$$ AS msg;
```

msg
varchar
Hello\nworld

```
SELECT $$The price is $9.95$$ AS msg;
```

msg
The price is $9.95

Even more, you can insert alphanumeric tags in the double-dollar symbols to allow for the use of regular double-dollar symbols *within* the string literal:

```
SELECT $tag$ this string can contain newlines,
'single quotes',
"double quotes",
and $$dollar quotes$$ $tag$ AS msg;
```

msg
varchar

```
┌──────────────────────────────────────────────────────────────────────────────────────────────────┐
│ this string can contain newlines,\n'single quotes',\n"double quotes",\nand $$dollar quotes$$ │
└──────────────────────────────────────────────────────────────────────────────────────────────────┘
```

Implicit concatenation only works for single-quoted string literals, not with dollar-quoted ones.

Map Type

MAPs are similar to STRUCTs in that they are an ordered list of "entries" where a key maps to a value. However, MAPs do not need to have the same keys present for each row, and thus are suitable for other use cases. MAPs are useful when the schema is unknown beforehand or when the schema varies per row; their flexibility is a key differentiator.

MAPs must have a single type for all keys, and a single type for all values. Keys and values can be any type, and the type of the keys does not need to match the type of the values (e.g., a MAP of VARCHAR to INT is valid). MAPs may not have duplicate keys. MAPs return an empty list if a key is not found rather than throwing an error as structs do.

In contrast, STRUCTs must have string keys, but each key may have a value of a different type. See the data types overview for a comparison between nested data types.

To construct a MAP, use the bracket syntax preceded by the MAP keyword.

Creating Maps

A map with VARCHAR keys and INTEGER values. This returns `{key1=10, key2=20, key3=30}`:

```
SELECT MAP {'key1': 10, 'key2': 20, 'key3': 30};
```

Alternatively use the map_from_entries function. This returns `{key1=10, key2=20, key3=30}`:

```
SELECT map_from_entries([('key1', 10), ('key2', 20), ('key3', 30)]);
```

A map can be also created using two lists: keys and values. This returns `{key1=10, key2=20, key3=30}`:

```
SELECT MAP(['key1', 'key2', 'key3'], [10, 20, 30]);
```

A map can also use INTEGER keys and NUMERIC values. This returns `{1=42.001, 5=-32.100}`:

```
SELECT MAP {1: 42.001, 5: -32.1};
```

Keys and/or values can also be nested types. This returns `{[a, b]=[1.1, 2.2], [c, d]=[3.3, 4.4]}`:

```
SELECT MAP {['a', 'b']: [1.1, 2.2], ['c', 'd']: [3.3, 4.4]};
```

Create a table with a map column that has INTEGER keys and DOUBLE values:

```
CREATE TABLE tbl (col MAP(INTEGER, DOUBLE));
```

Retrieving from Maps

MAPs use bracket notation for retrieving values. Selecting from a MAP returns a LIST rather than an individual value, with an empty LIST meaning that the key was not found.

Use bracket notation to retrieve the value at a key's location. Note that the expression in bracket notation must match the type of the map's key:

```
SELECT MAP {'key1': 5, 'key2': 43}['key1'];
```

```
5
```

If the element is not in the map, a NULL value will be returned.

```
SELECT MAP {'key1': 5, 'key2': 43}['key3'];
```

```
NULL
```

The `element_at` function can be used to retrieve a map value as a list:

```
SELECT element_at(MAP {'key1': 5, 'key2': 43}, 'key1');
```

```
[5]
```

Comparison Operators

Nested types can be compared using all the comparison operators. These comparisons can be used in logical expressions for both WHERE and HAVING clauses, as well as for creating Boolean values.

The ordering is defined positionally in the same way that words can be ordered in a dictionary. NULL values compare greater than all other values and are considered equal to each other.

At the top level, NULL nested values obey standard SQL NULL comparison rules: comparing a NULL nested value to a non-NULL nested value produces a NULL result. Comparing nested value *members*, however, uses the internal nested value rules for NULLs, and a NULL nested value member will compare above a non-NULL nested value member.

Functions

See Map Functions.

NULL Values

NULL values are special values that are used to represent missing data in SQL. Columns of any type can contain NULL values. Logically, a NULL value can be seen as "the value of this field is unknown".

A NULL value can be inserted to any field that does not have the NOT NULL qualifier:

```
CREATE TABLE integers (i INTEGER);
INSERT INTO integers VALUES (NULL);
```

NULL values have special semantics in many parts of the query as well as in many functions:

> Any comparison with a NULL value returns NULL, including NULL = NULL.

You can use IS NOT DISTINCT FROM to perform an equality comparison where NULL values compare equal to each other. Use IS (NOT) NULL to check if a value is NULL.

```
SELECT NULL = NULL;
```

```
NULL
```

```
SELECT NULL IS NOT DISTINCT FROM NULL;
```

```
true
```

```
SELECT NULL IS NULL;
```

```
true
```

NULL and Functions

A function that has input argument as `NULL` **usually** returns `NULL`.

```
SELECT cos(NULL);
```

```
NULL
```

The `coalesce` function is an exception to this: it takes any number of arguments, and returns for each row the first argument that is not `NULL`. If all arguments are `NULL`, `coalesce` also returns `NULL`.

```
SELECT coalesce(NULL, NULL, 1);
```

```
1
```

```
SELECT coalesce(10, 20);
```

```
10
```

```
SELECT coalesce(NULL, NULL);
```

```
NULL
```

The `ifnull` function is a two-argument version of `coalesce`.

```
SELECT ifnull(NULL, 'default_string');
```

```
default_string
```

```
SELECT ifnull(1, 'default_string');
```

```
1
```

NULL and AND / OR

`NULL` values have special behavior when used with `AND` and `OR`. For details, see the Boolean Type documentation.

NULL and IN / NOT IN

The behavior of `... IN <something with a NULL>` is different from `... IN <something with no NULLs>`. For details, see the `IN` documentation.

NULL and Aggregate Functions

`NULL` values are ignored in most aggregate functions.

Aggregate functions that do not ignore `NULL` values include: `first`, `last`, `list`, and `array_agg`. To exclude `NULL` values from those aggregate functions, the `FILTER` clause can be used.

```
CREATE TABLE integers (i INTEGER);
INSERT INTO integers VALUES (1), (10), (NULL);
```

```
SELECT min(i) FROM integers;
```

```
1
```

```
SELECT max(i) FROM integers;
```

```
10
```

Numeric Types

Integer Types

The types `TINYINT`, `SMALLINT`, `INTEGER`, `BIGINT` and `HUGEINT` store whole numbers, that is, numbers without fractional components, of various ranges. Attempts to store values outside of the allowed range will result in an error. The types `UTINYINT`, `USMALLINT`, `UINTEGER`, `UBIGINT` and `UHUGEINT` store whole unsigned numbers. Attempts to store negative numbers or values outside of the allowed range will result in an error.

Name	Aliases	Min	Max	Size in bytes
`TINYINT`	`INT1`	-2^7	2^7 - 1	1
`SMALLINT`	`INT2`, `INT16`, `SHORT`	-2^15	2^15 - 1	2
`INTEGER`	`INT4`, `INT32`, `INT`, `SIGNED`	-2^31	2^31 - 1	4
`BIGINT`	`INT8`, `INT64`, `LONG`	-2^63	2^63 - 1	8
`HUGEINT`	`INT128`	-2^127	2^127 - 1	16
`UTINYINT`	`UINT8`	0	2^8 - 1	1
`USMALLINT`	`UINT16`	0	2^16 - 1	2
`UINTEGER`	`UINT32`	0	2^32 - 1	4
`UBIGINT`	`UINT64`	0	2^64 - 1	8
`UHUGEINT`	`UINT128`	0	2^128 - 1	16

The type integer is the common choice, as it offers the best balance between range, storage size, and performance. The `SMALLINT` type is generally only used if disk space is at a premium. The `BIGINT` and `HUGEINT` types are designed to be used when the range of the integer type is insufficient.

Variable Integer

The previously mentioned integer types all have in common that the numbers in the minimum and maximum range all have the same storage size, `UTINYINT` is 1 byte, `SMALLINT` is 2 bytes, etc. But sometimes you need numbers that are even bigger than what is supported by a `HUGEINT`! For these situations the `VARINT` type can come in handy, as the `VARINT` type has a *much* bigger limit (the value can consist of up to 1,262,612 digits). The minimum storage size for a `VARINT` is 4 bytes, every digit takes up an extra bit, rounded up to 8 (12 digits take 12 bits, rounded up to 16, becomes two extra bytes).

Both negative and positive values are supported by the `VARINT` type.

Fixed-Point Decimals

The data type `DECIMAL(WIDTH, SCALE)` (also available under the alias `NUMERIC(WIDTH, SCALE)`) represents an exact fixed-point decimal value. When creating a value of type `DECIMAL`, the `WIDTH` and `SCALE` can be specified to define which size of decimal values can be held in the field. The `WIDTH` field determines how many digits can be held, and the `scale` determines the amount of digits after the decimal point. For example, the type `DECIMAL(3, 2)` can fit the value `1.23`, but cannot fit the value `12.3` or the value `1.234`. The default `WIDTH` and `SCALE` is `DECIMAL(18, 3)`, if none are specified.

Addition, subtraction, and multiplication of two fixed-point decimals returns another fixed-point decimal with the required `WIDTH` and `SCALE` to contain the exact result, or throws an error if the required `WIDTH` would exceed the maximal supported `WIDTH`, which is currently 38.

Division of fixed-point decimals does not typically produce numbers with finite decimal expansion. Therefore, DuckDB uses approximate floating-point arithmetic for all divisions that involve fixed-point decimals and accordingly returns floating-point data types.

Internally, decimals are represented as integers depending on their specified `WIDTH`.

Width	Internal	Size (bytes)
1-4	`INT16`	2
5-9	`INT32`	4
10-18	`INT64`	8
19-38	`INT128`	16

Performance can be impacted by using too large decimals when not required. In particular decimal values with a width above 19 are slow, as arithmetic involving the `INT128` type is much more expensive than operations involving the `INT32` or `INT64` types. It is therefore recommended to stick with a `WIDTH` of 18 or below, unless there is a good reason for why this is insufficient.

Floating-Point Types

The data types `FLOAT` and `DOUBLE` precision are variable-precision numeric types. In practice, these types are usually implementations of IEEE Standard 754 for Binary Floating-Point Arithmetic (single and double precision, respectively), to the extent that the underlying processor, operating system, and compiler support it.

Name	Aliases	Description
`FLOAT`	`FLOAT4`, `REAL`	Single precision floating-point number (4 bytes)
`DOUBLE`	`FLOAT8`	Double precision floating-point number (8 bytes)

Like for fixed-point data types, conversion from literals or casts from other datatypes to floating-point types stores inputs that cannot be represented exactly as approximations. However, it can be harder to predict what inputs are affected by this. For example, it is not surprising that `1.3::DECIMAL(1, 0) - 0.7::DECIMAL(1, 0) != 0.6::DECIMAL(1, 0)` but it may he surprising that `1.3::FLOAT - 0.7::FLOAT != 0.6::FLOAT`.

Additionally, whereas multiplication, addition, and subtraction of fixed-point decimal data types is exact, these operations are only approximate on floating-point binary data types.

For more complex mathematical operations, however, floating-point arithmetic is used internally and more precise results can be obtained if intermediate steps are *not* cast to fixed point formats of the same width as in- and outputs. For example, `(10::FLOAT / 3::FLOAT)::FLOAT * 3 = 10` whereas `(10::DECIMAL(18, 3) / 3::DECIMAL(18, 3))::DECIMAL(18, 3) * 3 = 9.999`.

In general, we advise that:

- If you require exact storage of numbers with a known number of decimal digits and require exact additions, subtractions, and multiplications (such as for monetary amounts), use the `DECIMAL` data type or its `NUMERIC` alias instead.
- If you want to do fast or complicated calculations, the floating-point data types may be more appropriate. However, if you use the results for anything important, you should evaluate your implementation carefully for corner cases (ranges, infinities, underflows, invalid operations) that may be handled differently from what you expect and you should familiarize yourself with common floating-point pitfalls. The article "What Every Computer Scientist Should Know About Floating-Point Arithmetic" by David Goldberg and the floating point series on Bruce Dawson's blog provide excellent starting points.

On most platforms, the `FLOAT` type has a range of at least 1E-37 to 1E+37 with a precision of at least 6 decimal digits. The `DOUBLE` type typically has a range of around 1E-307 to 1E+308 with a precision of at least 15 digits. Positive numbers outside of these ranges (and negative numbers ourside the mirrored ranges) may cause errors on some platforms but will usually be converted to zero or infinity, respectively.

In addition to ordinary numeric values, the floating-point types have several special values representing IEEE 754 special values:

- `Infinity`: infinity
- `-Infinity`: negative infinity
- NaN: not a number

On machines with the required CPU/FPU support, DuckDB follows the IEEE 754 specification regarding these special values, with two exceptions:

- NaN compares equal to NaN and greater than any other floating point number.
- Some floating point functions, like `sqrt` / `sin` / `asin` throw errors rather than return NaN for values outside their ranges of definition.

To insert these values as literals in a SQL command, you must put quotes around them, you may abbreviate `Infinity` as `Inf`, and you may use any capitalization. For example:

```
SELECT
    sqrt(2) > '-inf',
    'nan' > sqrt(2);
```

(sqrt(2) > '-inf')	('nan' > sqrt(2))
true	true

Universally Unique Identifiers (UUIDs)

DuckDB supports universally unique identifiers (UUIDs) through the `UUID` type. These use 128 bits and are represented internally as `HUGEINT` values. When printed, they are shown with lowercase hexadecimal characters, separated by dashes as follows: `<12345678>-<1234>-<1234>-<1234>-<1234567890ab>{:.language-sql .highlight}` (using 36 characters in total including the dashes). For example, `4ac7a9e9-607c-4c8a-84f3-843f0191e3fd` is a valid UUID.

DuckDB supports generating UUIDv4 and UUIDv7 identifiers. To retrieve the version of a UUID value, use the `uuid_extract_version` function.

UUIDv4

To generate a UUIDv4 value, use the `uuid()` function or it aliases the `uuidv4()` and `gen_random_uuid()` functions.

UUIDv7

To generate a UUIDv7 value, use the `uuidv7()` function. To retrieve the timestamp from a UUIDv7 value, use the `uuid_extract_timestamp` function:

```
SELECT uuid_extract_timestamp(uuidv7()) AS ts;
```

ts
2025-04-19 15:51:20.07+00

Functions

See Numeric Functions and Operators.

Struct Data Type

Conceptually, a STRUCT column contains an ordered list of columns called "entries". The entries are referenced by name using strings. This document refers to those entry names as keys. Each row in the STRUCT column must have the same keys. The names of the struct entries are part of the *schema*. Each row in a STRUCT column must have the same layout. The names of the struct entries are case-insensitive.

STRUCTs are typically used to nest multiple columns into a single column, and the nested column can be of any type, including other STRUCTs and `LIST`s.

STRUCTs are similar to PostgreSQL's `ROW` type. The key difference is that DuckDB STRUCTs require the same keys in each row of a STRUCT column. This allows DuckDB to provide significantly improved performance by fully utilizing its vectorized execution engine, and also enforces type consistency for improved correctness. DuckDB includes a `row` function as a special way to produce a STRUCT, but does not have a `ROW` data type. See an example below and the STRUCT functions documentation for details.

See the data types overview for a comparison between nested data types.

Creating Structs

Structs can be created using the `struct_pack(name := expr, ...)` function, the equivalent array notation `{'name': expr, ...}`, using a row variable, or using the `row` function.

Create a struct using the `struct_pack` function. Note the lack of single quotes around the keys and the use of the `:=` operator:

```
SELECT struct_pack(key1 := 'value1', key2 := 42) AS s;
```

Create a struct using the array notation:

```
SELECT {'key1': 'value1', 'key2': 42} AS s;
```

Create a struct using a row variable:

```
SELECT d AS s FROM (SELECT 'value1' AS key1, 42 AS key2) d;
```

Create a struct of integers:

```
SELECT {'x': 1, 'y': 2, 'z': 3} AS s;
```

Create a struct of strings with a `NULL` value:

```
SELECT {'yes': 'duck', 'maybe': 'goose', 'huh': NULL, 'no': 'heron'} AS s;
```

Create a struct with a different type for each key:

```
SELECT {'key1': 'string', 'key2': 1, 'key3': 12.345} AS s;
```

Create a struct of structs with `NULL` values:

```
SELECT {
        'birds': {'yes': 'duck', 'maybe': 'goose', 'huh': NULL, 'no': 'heron'},
        'aliens': NULL,
        'amphibians': {'yes': 'frog', 'maybe': 'salamander', 'huh': 'dragon', 'no': 'toad'}
    } AS s;
```

Adding Field(s)/Value(s) to Structs

Add to a struct of integers:

```
SELECT struct_insert({'a': 1, 'b': 2, 'c': 3}, d := 4) AS s;
```

Retrieving from Structs

Retrieving a value from a struct can be accomplished using dot notation, bracket notation, or through struct functions like `struct_extract`.

Use dot notation to retrieve the value at a key's location. In the following query, the subquery generates a struct column a, which we then query with `a.x`.

```
SELECT a.x FROM (SELECT {'x': 1, 'y': 2, 'z': 3} AS a);
```

If a key contains a space, simply wrap it in double quotes (").

```
SELECT a."x space" FROM (SELECT {'x space': 1, 'y': 2, 'z': 3} AS a);
```

Bracket notation may also be used. Note that this uses single quotes (') since the goal is to specify a certain string key and only constant expressions may be used inside the brackets (no expressions):

```
SELECT a['x space'] FROM (SELECT {'x space': 1, 'y': 2, 'z': 3} AS a);
```

The `struct_extract` function is also equivalent. This returns 1:

```
SELECT struct_extract({'x space': 1, 'y': 2, 'z': 3}, 'x space');
```

`unnest` / `STRUCT.*`

Rather than retrieving a single key from a struct, the `unnest` special function can be used to retrieve all keys from a struct as separate columns. This is particularly useful when a prior operation creates a struct of unknown shape, or if a query must handle any potential struct keys:

```
SELECT unnest(a)
FROM (SELECT {'x': 1, 'y': 2, 'z': 3} AS a);
```

x	y	z
1	2	3

The same can be achieved with the star notation (*), which additionally allows modifications of the returned columns:

```
SELECT a.* EXCLUDE ('y')
FROM (SELECT {'x': 1, 'y': 2, 'z': 3} AS a);
```

x	z
1	3

> **Warning.** The star notation is currently limited to top-level struct columns and non-aggregate expressions.

Dot Notation Order of Operations

Referring to structs with dot notation can be ambiguous with referring to schemas and tables. In general, DuckDB looks for columns first, then for struct keys within columns. DuckDB resolves references in these orders, using the first match to occur:

No Dots

```
SELECT part1
FROM tbl;
```

1. `part1` is a column

One Dot

```
SELECT part1.part2
FROM tbl;
```

1. `part1` is a table, `part2` is a column
2. `part1` is a column, `part2` is a property of that column

Two (or More) Dots

```
SELECT part1.part2.part3
FROM tbl;
```

1. `part1` is a schema, `part2` is a table, `part3` is a column
2. `part1` is a table, `part2` is a column, `part3` is a property of that column
3. `part1` is a column, `part2` is a property of that column, `part3` is a property of that column

Any extra parts (e.g., `.part4.part5`, etc.) are always treated as properties

Creating Structs with the `row` Function

The `row` function can be used to automatically convert multiple columns to a single struct column. When using `row` the keys will be empty strings allowing for easy insertion into a table with a struct column. Columns, however, cannot be initialized with the `row` function, and must be explicitly named. For example, inserting values into a struct column using the `row` function:

```
CREATE TABLE t1 (s STRUCT(v VARCHAR, i INTEGER));
INSERT INTO t1 VALUES (row('a', 42));
SELECT * FROM t1;
```

The table will contain a single entry:

```
{'v': a, 'i': 42}
```

The following produces the same result as above:

```
CREATE TABLE t1 AS (
    SELECT row('a', 42)::STRUCT(v VARCHAR, i INTEGER)
);
```

Initializing a struct column with the `row` function will fail:

```
CREATE TABLE t2 AS SELECT row('a');
```

```
Invalid Input Error:
A table cannot be created from an unnamed struct
```

When casting between structs, the names of at least one field have to match. Therefore, the following query will fail:

```
SELECT a::STRUCT(y INTEGER) AS b
FROM
    (SELECT {'x': 42} AS a);
```

```
Binder Error:
STRUCT to STRUCT cast must have at least one matching member
```

A workaround for this is to use `struct_pack` instead:

```
SELECT struct_pack(y := a.x) AS b
FROM
    (SELECT {'x': 42} AS a);
```

The `row` function can be used to return unnamed structs. For example:

```
SELECT row(x, x + 1, y) FROM (SELECT 1 AS x, 'a' AS y) AS s;
```

This produces `(1, 2, a)`.

If using multiple expressions when creating a struct, the `row` function is optional. The following query returns the same result as the previous one:

```
SELECT (x, x + 1, y) AS s FROM (SELECT 1 AS x, 'a' AS y);
```

Comparison and Ordering

The STRUCT type can be compared using all the comparison operators. These comparisons can be used in logical expressions such as `WHERE` and `HAVING` clauses, and return BOOLEAN values.

For comparisons, the keys of a STRUCT have a fixed positional order, from left to right. Comparisons behave the same as row comparisons, therefore, matching keys must be at identical positions.

Specifically, for any STRUCT comparison, the following rules apply:

- **Equality.** `s1` and `s2` are equal, if all respective values are equal.
- **Less Than**. For the first index `i` where `s1.value[i] != s2.value[i]`: If `s1.value[i] < s2.value[i]`, `s1` is less than `s2`.

`NULL` values are compared following PostgreSQL's semantics. Lower nesting levels are used for tie-breaking.

Here are some queries returning `true` for the comparison.

```
SELECT {'k1': 2, 'k2': 3} < {'k1': 2, 'k2': 4} AS result;
```

```
SELECT {'k1': 'hello'} < {'k1': 'world'} AS result;
```

These queries return `false`.

```
SELECT {'k2': 4, 'k1': 3} < {'k2': 2, 'k1': 4} AS result;
```

```
SELECT {'k1': [4, 3]} < {'k1': [3, 6, 7]} AS result;
```

These queries return `NULL`.

```
SELECT {'k1': 2, 'k2': 3} < {'k1': 2, 'k2': NULL} AS result;
```

Updating the Schema

Starting with DuckDB v1.3.0, it's possible to update the sub-schema of structs using the `ALTER TABLE` clause.

To follow the examples, initialize the `test` table as follows:

```
CREATE TABLE test(s STRUCT(i INTEGER, j INTEGER));
INSERT INTO test VALUES (ROW(1, 1)), (ROW(2, 2));
```

Adding a Field

Add field `k INTEGER` to struct `s` in table `test`:

```
ALTER TABLE test ADD COLUMN s.k INTEGER;
FROM test;
```

s
struct(i integer, j integer, k integer)
{'i': 1, 'j': 1, 'k': NULL}
{'i': 2, 'j': 2, 'k': NULL}

Dropping a Field

Drop field `i` from struct `s` in table `test`:

```
ALTER TABLE test DROP COLUMN s.i;
FROM test;
```

s
struct(j integer, k integer)
{'j': 1, 'k': NULL}
{'j': 2, 'k': NULL}

Renaming a Field

Renaming field `j` of struct `s` to `v1` in table test‘:

```
ALTER TABLE test RENAME s.j TO v1;
FROM test;
```

s
struct(v1 integer, k integer)
{'v1': 1, 'k': NULL}
{'v1': 2, 'k': NULL}

Functions

See Struct Functions.

Text Types

In DuckDB, strings can be stored in the VARCHAR field. The field allows storage of Unicode characters. Internally, the data is encoded as UTF-8.

Name	Aliases	Description
VARCHAR	CHAR, BPCHAR, STRING, TEXT	Variable-length character string
VARCHAR(n)	CHAR(n), BPCHAR(n), STRING(n), TEXT(n)	Variable-length character string. The maximum length n has no effect and is only provided for compatibility

Specifying a Length Limit

Specifying the length for the VARCHAR, STRING, and TEXT types is not required and has no effect on the system. Specifying the length will not improve performance or reduce storage space of the strings in the database. These variants variant is supported for compatibility reasons with other systems that do require a length to be specified for strings.

If you wish to restrict the number of characters in a VARCHAR column for data integrity reasons the CHECK constraint should be used, for example:

```
CREATE TABLE strings (
    val VARCHAR CHECK (length(val) <= 10) -- val has a maximum length of 10
);
```

The VARCHAR field allows storage of Unicode characters. Internally, the data is encoded as UTF-8.

Specifying a Compression Type

You can specify a compression type for a string with the `USING COMPRESSION` clause. For example, to apply zstd compression, run:

```
CREATE TABLE tbl(s VARCHAR USING COMPRESSION zstd);
```

Text Type Values

Values of the text type are character strings, also known as string values or simply strings. At runtime, string values are constructed in one of the following ways:

- referencing columns whose declared or implied type is the text data type
- string literals
- casting expressions to a text type
- applying a string operator, or invoking a function that returns a text type value

Strings with Special Characters

To use special characters in string, use escape string literals or dollar-quoted string literals. Alternatively, you can use concatenation and the `chr` character function:

```
SELECT 'Hello' || chr(10) || 'world' AS msg;
```

```
┌─────────────┐
│     msg     │
│   varchar   │
├─────────────┤
│ Hello\nworld │
└─────────────┘
```

Functions

See Text Functions and Pattern Matching.

Time Types

The TIME and TIMETZ types specify the hour, minute, second, microsecond of a day.

Name	Aliases	Description
TIME	TIME WITHOUT TIME ZONE	Time of day (ignores time zone)
TIMETZ	TIME WITH TIME ZONE	Time of day (uses time zone)

Instances can be created using the type names as a keyword, where the data must be formatted according to the ISO 8601 format (hh:mm:ss[.zzzzzz][+-TT[:tt]]).

```
SELECT TIME '1992-09-20 11:30:00.123456';
```

```
11:30:00.123456
```

```
SELECT TIMETZ '1992-09-20 11:30:00.123456';
```

```
11:30:00.123456+00
```

```
SELECT TIMETZ '1992-09-20 11:30:00.123456-02:00';
```

```
13:30:00.123456+00
```

```
SELECT TIMETZ '1992-09-20 11:30:00.123456+05:30';
```

```
06:00:00.123456+00
```

> **Warning.** The TIME type should only be used in rare cases, where the date part of the timestamp can be disregarded. Most applications should use the TIMESTAMP types to represent their timestamps.

Timestamp Types

Timestamps represent points in time. As such, they combine DATE and TIME information. They can be created using the type name followed by a string formatted according to the ISO 8601 format, YYYY-MM-DD hh:mm:ss[.zzzzzzzzz][+-TT[:tt]], which is also the format we use in this documentation. Decimal places beyond the supported precision are ignored.

Timestamp Types

Name	Aliases	Description
TIMESTAMP_NS		Naive timestamp with nanosecond precision
TIMESTAMP	DATETIME, TIMESTAMP WITHOUT TIME ZONE	Naive timestamp with microsecond precision
TIMESTAMP_MS		Naive timestamp with millisecond precision

Name	Aliases	Description
`TIMESTAMP_S`		Naive timestamp with second precision
`TIMESTAMPTZ`	`TIMESTAMP WITH TIME ZONE`	Time zone aware timestamp with microsecond precision

> **Warning.** Since there is not currently a `TIMESTAMP_NS WITH TIME ZONE` data type, external columns with nanosecond precision and `WITH TIME ZONE` semantics, e.g., Parquet timestamp columns with `isAdjustedToUTC=true`, are converted to `TIMESTAMP WITH TIME ZONE` and thus lose precision when read using DuckDB.

```
SELECT TIMESTAMP_NS '1992-09-20 11:30:00.123456789';
```

```
1992-09-20 11:30:00.123456789
```

```
SELECT TIMESTAMP '1992-09-20 11:30:00.123456789';
```

```
1992-09-20 11:30:00.123456
```

```
SELECT TIMESTAMP_MS '1992-09-20 11:30:00.123456789';
```

```
1992-09-20 11:30:00.123
```

```
SELECT TIMESTAMP_S '1992-09-20 11:30:00.123456789';
```

```
1992-09-20 11:30:00
```

```
SELECT TIMESTAMPTZ '1992-09-20 11:30:00.123456789';
```

```
1992-09-20 11:30:00.123456+00
```

```
SELECT TIMESTAMPTZ '1992-09-20 12:30:00.123456789+01:00';
```

```
1992-09-20 11:30:00.123456+00
```

DuckDB distinguishes timestamps `WITHOUT TIME ZONE` and `WITH TIME ZONE` (of which the only current representative is `TIMESTAMP WITH TIME ZONE`).

Despite the name, a `TIMESTAMP WITH TIME ZONE` does not store time zone information. Instead, it only stores the `INT64` number of non-leap microseconds since the Unix epoch `1970-01-01 00:00:00+00`, and thus unambiguously identifies a point in absolute time, or *instant*. The reason for the labels *time zone aware* and `WITH TIME ZONE` is that timestamp arithmetic, *binning*, and string formatting for this type are performed in a configured time zone, which defaults to the system time zone and is just `UTC+00:00` in the examples above.

The corresponding `TIMESTAMP WITHOUT TIME ZONE` stores the same `INT64`, but arithmetic, binning, and string formatting follow the straightforward rules of Coordinated Universal Time (UTC) without offsets or time zones. Accordingly, `TIMESTAMP`s could be interpreted as UTC timestamps, but more commonly they are used to represent *local* observations of time recorded in an unspecified time zone, and operations on these types can be interpreted as simply manipulating tuple fields following nominal temporal logic. It is a common data cleaning problem to disambiguate such observations, which may also be stored in raw strings without time zone specification or UTC offsets, into unambiguous `TIMESTAMP WITH TIME ZONE` instants. One possible solution to this is to append UTC offsets to strings, followed by an explicit cast to `TIMESTAMP WITH TIME ZONE`. Alternatively, a `TIMESTAMP WITHOUT TIME ZONE` may be created first and then be combined with a time zone specification to obtain a time zone aware `TIMESTAMP WITH TIME ZONE`.

Conversion between Strings and Naïve / Time Zone-Aware Timestamps

The conversion between strings *without* UTC offsets or IANA time zone names and `WITHOUT TIME ZONE` types is unambiguous and straightforward. The conversion between strings *with* UTC offsets or time zone names and `WITH TIME ZONE` types is also unambiguous, but requires the `ICU` extension to handle time zone names.

When strings *without* UTC offsets or time zone names are converted to a `WITH TIME ZONE` type, the string is interpreted in the configured time zone. Conversely, when strings *with* UTC offsets are passed to a `WITHOUT TIME ZONE` type, the local time in the configured time zone at the instant specified by the string is stored.

Finally, when `WITH TIME ZONE` and `WITHOUT TIME ZONE` types are converted to each other via explicit or implicit casts, the translation uses the configured time zone. To use an alternative time zone, the `timezone` function provided by the `ICU` extension may be used:

```
SELECT
    timezone('America/Denver', TIMESTAMP '2001-02-16 20:38:40') AS aware1,
    timezone('America/Denver', TIMESTAMPTZ '2001-02-16 04:38:40') AS naive1,
    timezone('UTC', TIMESTAMP '2001-02-16 20:38:40+00:00') AS aware2,
    timezone('UTC', TIMESTAMPTZ '2001-02-16 04:38:40 Europe/Berlin') AS naive2;
```

aware1	naive1	aware2	naive2
2001-02-17 04:38:40+01	2001-02-15 20:38:40	2001-02-16 21:38:40+01	2001-02-16 03:38:40

Note that `TIMESTAMP`s are displayed without time zone specification in the results, following ISO 8601 rules for local times, while time-zone aware `TIMESTAMPTZ`s are displayed with the UTC offset of the configured time zone, which is `'Europe/Berlin'` in the example. The UTC offsets of `'America/Denver'` and `'Europe/Berlin'` at all involved instants are `-07:00` and `+01:00`, respectively.

Special Values

Three special strings can be used to create timestamps:

Input string	Description
`epoch`	1970-01-01 00:00:00[+00] (Unix system time zero)
`infinity`	Later than all other timestamps
`-infinity`	Earlier than all other timestamps

The values `infinity` and `-infinity` are special cased and are displayed unchanged, whereas the value `epoch` is simply a notational shorthand that is converted to the corresponding timestamp value when read.

```
SELECT '-infinity'::TIMESTAMP, 'epoch'::TIMESTAMP, 'infinity'::TIMESTAMP;
```

Negative	Epoch	Positive
-infinity	1970-01-01 00:00:00	infinity

Functions

See Timestamp Functions.

Time Zones

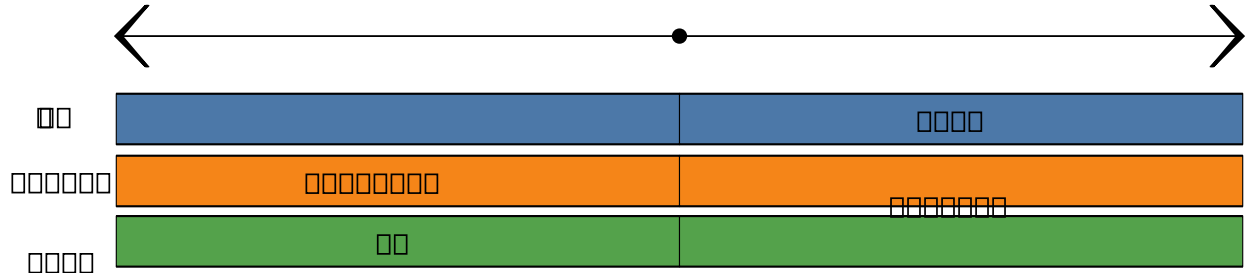
To understand time zones and the `WITH TIME ZONE` types, it helps to start with two concepts: *instants* and *temporal binning*.

Instants

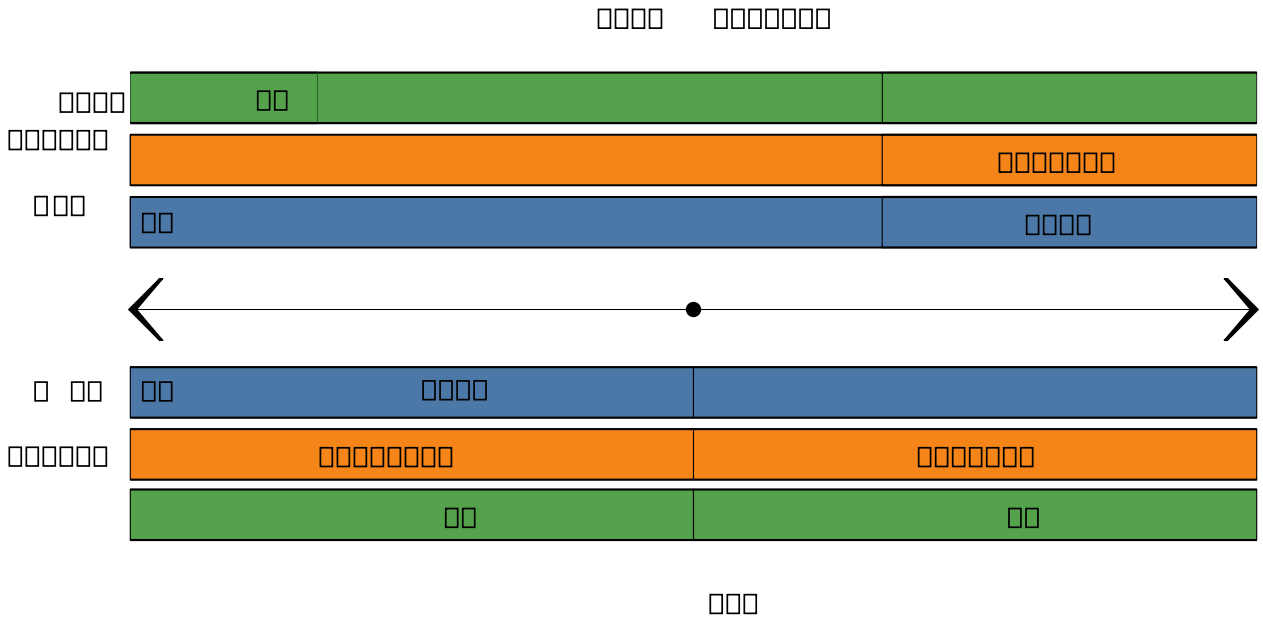
An instant is a point in absolute time, usually given as a count of some time increment from a fixed point in time (called the *epoch*). This is similar to how positions on the earth's surface are given using latitude and longitude relative to the equator and the Greenwich Meridian. In DuckDB, the fixed point is the Unix epoch `1970-01-01 00:00:00+00:00`, and the increment is in seconds, milliseconds, microseconds, or nanoseconds, depending on the specific data type.

Temporal Binning

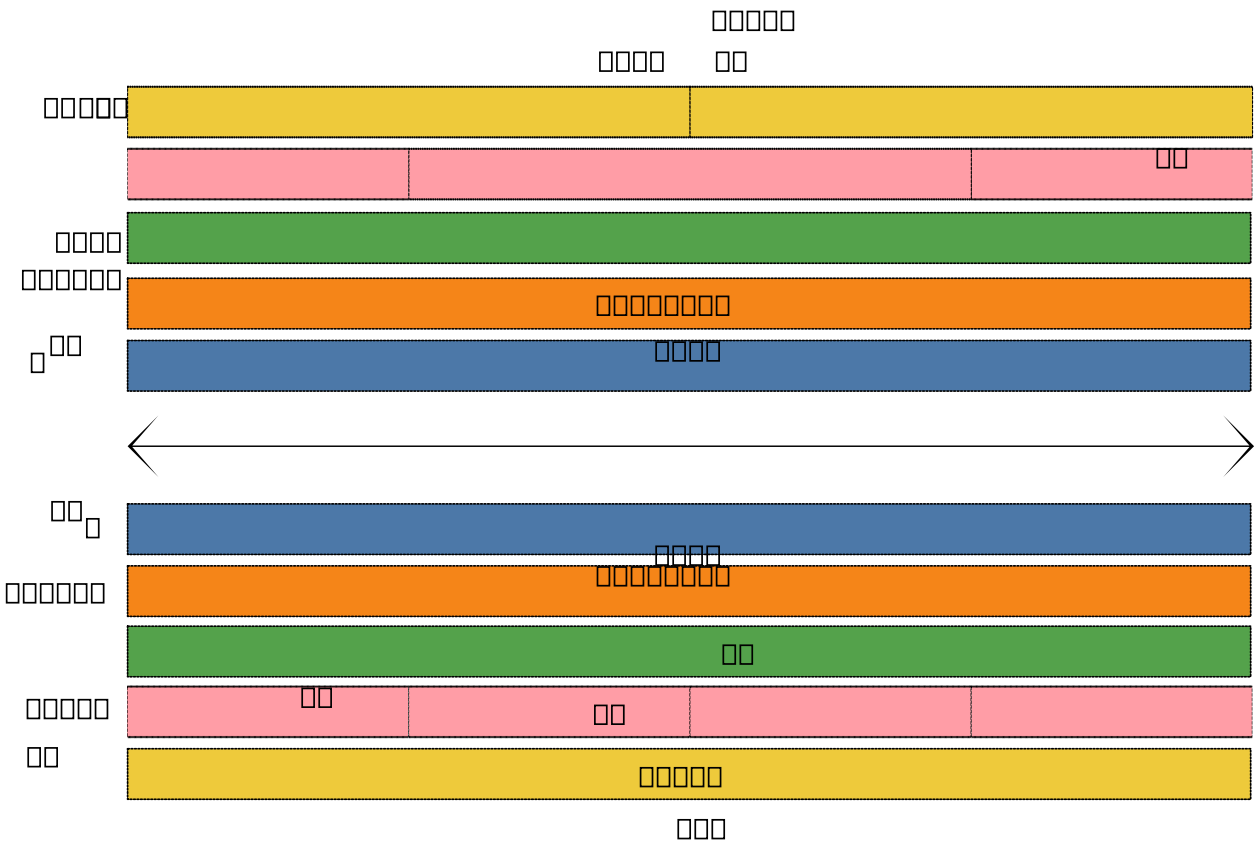
Binning is a common practice with continuous data: A range of possible values is broken up into contiguous subsets and the binning operation maps actual values to the *bin* they fall into. *Temporal binning* is simply applying this practice to instants; for example, by binning instants into years, months, and days.



Temporal binning rules are complex, and generally come in two sets: *time zones* and *calendars*. For most tasks, the calendar will just be the widely used Gregorian calendar, but time zones apply locale-specific rules and can vary widely. For example, here is what binning for the `'America/Los_Angeles'` time zone looks like near the epoch:



The most common temporal binning problem occurs when daylight savings time changes. The example below contains a daylight savings time change where the ”hour” bin is two hours long. To distinguish the two hours, another range of bins containing the offset from UTC is needed:



Time Zone Support

The `TIMESTAMPTZ` type can be binned into calendar and clock bins using a suitable extension. The built-in ICU extension implements all the binning and arithmetic functions using the International Components for Unicode time zone and calendar functions.

To set the time zone to use, first load the ICU extension. The ICU extension comes pre-bundled with several DuckDB clients (including Python, R, JDBC, and ODBC), so this step can be skipped in those cases. In other cases you might first need to install and load the ICU extension.

```
INSTALL icu;
LOAD icu;
```

Next, use the `SET TimeZone` command:

```
SET TimeZone = 'America/Los_Angeles';
```

Time binning operations for `TIMESTAMPTZ` will then be implemented using the given time zone.

A list of available time zones can be pulled from the `pg_timezone_names()` table function:

```
SELECT
    name,
    abbrev,
    utc_offset
FROM pg_timezone_names()
ORDER BY
    name;
```

You can also find a reference table of available time zones.

Calendar Support

The ICU extension also supports non-Gregorian calendars using the `SET Calendar` command. Note that the `INSTALL` and `LOAD` steps are only required if the DuckDB client does not bundle the ICU extension.

```
INSTALL icu;
LOAD icu;
SET Calendar = 'japanese';
```

Time binning operations for `TIMESTAMPTZ` will then be implemented using the given calendar. In this example, the `era` part will now report the Japanese imperial era number.

A list of available calendars can be pulled from the `icu_calendar_names()` table function:

```
SELECT name
FROM icu_calendar_names()
ORDER BY 1;
```

Settings

The current value of the `TimeZone` and `Calendar` settings are determined by ICU when it starts up. They can be queried from in the `duckdb_settings()` table function:

```
SELECT *
FROM duckdb_settings()
WHERE name = 'TimeZone';
```

name	value	description	input_type
TimeZone	Europe/Amsterdam	The current time zone	VARCHAR

```
SELECT *
FROM duckdb_settings()
WHERE name = 'Calendar';
```

name	value	description	input_type
Calendar	gregorian	The current calendar	VARCHAR

> If you find that your binning operations are not behaving as you expect, check the `TimeZone` and `Calendar` values and adjust them if needed.

Time Zone Reference List

An up-to-date version of this list can be pulled from the `pg_timezone_names()` table function:

```
SELECT name, abbrev
FROM pg_timezone_names()
ORDER BY name;
```

name	abbrev
ACT	ACT
AET	AET
AGT	AGT
ART	ART
AST	AST
Africa/Abidjan	Iceland
Africa/Accra	Iceland
Africa/Addis_Ababa	EAT
Africa/Algiers	Africa/Algiers
Africa/Asmara	EAT
Africa/Asmera	EAT
Africa/Bamako	Iceland
Africa/Bangui	Africa/Bangui
Africa/Banjul	Iceland
Africa/Bissau	Africa/Bissau
Africa/Blantyre	CAT
Africa/Brazzaville	Africa/Brazzaville
Africa/Bujumbura	CAT
Africa/Cairo	ART
Africa/Casablanca	Africa/Casablanca
Africa/Ceuta	Africa/Ceuta
Africa/Conakry	Iceland
Africa/Dakar	Iceland
Africa/Dar_es_Salaam	EAT
Africa/Djibouti	EAT
Africa/Douala	Africa/Douala
Africa/El_Aaiun	Africa/El_Aaiun
Africa/Freetown	Iceland
Africa/Gaborone	CAT
Africa/Harare	CAT
Africa/Johannesburg	Africa/Johannesburg
Africa/Juba	Africa/Juba
Africa/Kampala	EAT
Africa/Khartoum	Africa/Khartoum
Africa/Kigali	CAT
Africa/Kinshasa	Africa/Kinshasa
Africa/Lagos	Africa/Lagos
Africa/Libreville	Africa/Libreville
Africa/Lome	Iceland

name	abbrev
Africa/Luanda	Africa/Luanda
Africa/Lubumbashi	CAT
Africa/Lusaka	CAT
Africa/Malabo	Africa/Malabo
Africa/Maputo	CAT
Africa/Maseru	Africa/Maseru
Africa/Mbabane	Africa/Mbabane
Africa/Mogadishu	EAT
Africa/Monrovia	Africa/Monrovia
Africa/Nairobi	EAT
Africa/Ndjamena	Africa/Ndjamena
Africa/Niamey	Africa/Niamey
Africa/Nouakchott	Iceland
Africa/Ouagadougou	Iceland
Africa/Porto-Novo	Africa/Porto-Novo
Africa/Sao_Tome	Africa/Sao_Tome
Africa/Timbuktu	Iceland
Africa/Tripoli	Libya
Africa/Tunis	Africa/Tunis
Africa/Windhoek	Africa/Windhoek
America/Adak	America/Adak
America/Anchorage	AST
America/Anguilla	PRT
America/Antigua	PRT
America/Araguaina	America/Araguaina
America/Argentina/Buenos_Aires	AGT
America/Argentina/Catamarca	America/Argentina/Catamarca
America/Argentina/ComodRivadavia	America/Argentina/ComodRivadavia
America/Argentina/Cordoba	America/Argentina/Cordoba
America/Argentina/Jujuy	America/Argentina/Jujuy
America/Argentina/La_Rioja	America/Argentina/La_Rioja
America/Argentina/Mendoza	America/Argentina/Mendoza
America/Argentina/Rio_Gallegos	America/Argentina/Rio_Gallegos
America/Argentina/Salta	America/Argentina/Salta
America/Argentina/San_Juan	America/Argentina/San_Juan
America/Argentina/San_Luis	America/Argentina/San_Luis
America/Argentina/Tucuman	America/Argentina/Tucuman
America/Argentina/Ushuaia	America/Argentina/Ushuaia
America/Aruba	PRT

name	abbrev
America/Asuncion	America/Asuncion
America/Atikokan	EST
America/Atka	America/Atka
America/Bahia	America/Bahia
America/Bahia_Banderas	America/Bahia_Banderas
America/Barbados	America/Barbados
America/Belem	America/Belem
America/Belize	America/Belize
America/Blanc-Sablon	PRT
America/Boa_Vista	America/Boa_Vista
America/Bogota	America/Bogota
America/Boise	America/Boise
America/Buenos_Aires	AGT
America/Cambridge_Bay	America/Cambridge_Bay
America/Campo_Grande	America/Campo_Grande
America/Cancun	America/Cancun
America/Caracas	America/Caracas
America/Catamarca	America/Catamarca
America/Cayenne	America/Cayenne
America/Cayman	EST
America/Chicago	CST
America/Chihuahua	America/Chihuahua
America/Ciudad_Juarez	America/Ciudad_Juarez
America/Coral_Harbour	EST
America/Cordoba	America/Cordoba
America/Costa_Rica	America/Costa_Rica
America/Creston	MST
America/Cuiaba	America/Cuiaba
America/Curacao	PRT
America/Danmarkshavn	America/Danmarkshavn
America/Dawson	America/Dawson
America/Dawson_Creek	America/Dawson_Creek
America/Denver	Navajo
America/Detroit	America/Detroit
America/Dominica	PRT
America/Edmonton	America/Edmonton
America/Eirunepe	America/Eirunepe
America/El_Salvador	America/El_Salvador
America/Ensenada	America/Ensenada

name	abbrev
America/Fort_Nelson	America/Fort_Nelson
America/Fort_Wayne	IET
America/Fortaleza	America/Fortaleza
America/Glace_Bay	America/Glace_Bay
America/Godthab	America/Godthab
America/Goose_Bay	America/Goose_Bay
America/Grand_Turk	America/Grand_Turk
America/Grenada	PRT
America/Guadeloupe	PRT
America/Guatemala	America/Guatemala
America/Guayaquil	America/Guayaquil
America/Guyana	America/Guyana
America/Halifax	America/Halifax
America/Havana	Cuba
America/Hermosillo	America/Hermosillo
America/Indiana/Indianapolis	IET
America/Indiana/Knox	America/Indiana/Knox
America/Indiana/Marengo	America/Indiana/Marengo
America/Indiana/Petersburg	America/Indiana/Petersburg
America/Indiana/Tell_City	America/Indiana/Tell_City
America/Indiana/Vevay	America/Indiana/Vevay
America/Indiana/Vincennes	America/Indiana/Vincennes
America/Indiana/Winamac	America/Indiana/Winamac
America/Indianapolis	IET
America/Inuvik	America/Inuvik
America/Iqaluit	America/Iqaluit
America/Jamaica	Jamaica
America/Jujuy	America/Jujuy
America/Juneau	America/Juneau
America/Kentucky/Louisville	America/Kentucky/Louisville
America/Kentucky/Monticello	America/Kentucky/Monticello
America/Knox_IN	America/Knox_IN
America/Kralendijk	PRT
America/La_Paz	America/La_Paz
America/Lima	America/Lima
America/Los_Angeles	PST
America/Louisville	America/Louisville
America/Lower_Princes	PRT
America/Maceio	America/Maceio

name	abbrev
America/Managua	America/Managua
America/Manaus	America/Manaus
America/Marigot	PRT
America/Martinique	America/Martinique
America/Matamoros	America/Matamoros
America/Mazatlan	America/Mazatlan
America/Mendoza	America/Mendoza
America/Menominee	America/Menominee
America/Merida	America/Merida
America/Metlakatla	America/Metlakatla
America/Mexico_City	America/Mexico_City
America/Miquelon	America/Miquelon
America/Moncton	America/Moncton
America/Monterrey	America/Monterrey
America/Montevideo	America/Montevideo
America/Montreal	America/Montreal
America/Montserrat	PRT
America/Nassau	America/Nassau
America/New_York	EST5EDT
America/Nipigon	America/Nipigon
America/Nome	America/Nome
America/Noronha	America/Noronha
America/North_Dakota/Beulah	America/North_Dakota/Beulah
America/North_Dakota/Center	America/North_Dakota/Center
America/North_Dakota/New_Salem	America/North_Dakota/New_Salem
America/Nuuk	America/Nuuk
America/Ojinaga	America/Ojinaga
America/Panama	EST
America/Pangnirtung	America/Pangnirtung
America/Paramaribo	America/Paramaribo
America/Phoenix	MST
America/Port-au-Prince	America/Port-au-Prince
America/Port_of_Spain	PRT
America/Porto_Acre	America/Porto_Acre
America/Porto_Velho	America/Porto_Velho
America/Puerto_Rico	PRT
America/Punta_Arenas	America/Punta_Arenas
America/Rainy_River	America/Rainy_River
America/Rankin_Inlet	America/Rankin_Inlet

name	abbrev
America/Recife	America/Recife
America/Regina	America/Regina
America/Resolute	America/Resolute
America/Rio_Branco	America/Rio_Branco
America/Rosario	America/Rosario
America/Santa_Isabel	America/Santa_Isabel
America/Santarem	America/Santarem
America/Santiago	America/Santiago
America/Santo_Domingo	America/Santo_Domingo
America/Sao_Paulo	BET
America/Scoresbysund	America/Scoresbysund
America/Shiprock	Navajo
America/Sitka	America/Sitka
America/St_Barthelemy	PRT
America/St_Johns	CNT
America/St_Kitts	PRT
America/St_Lucia	PRT
America/St_Thomas	PRT
America/St_Vincent	PRT
America/Swift_Current	America/Swift_Current
America/Tegucigalpa	America/Tegucigalpa
America/Thule	America/Thule
America/Thunder_Bay	America/Thunder_Bay
America/Tijuana	America/Tijuana
America/Toronto	America/Toronto
America/Tortola	PRT
America/Vancouver	America/Vancouver
America/Virgin	PRT
America/Whitehorse	America/Whitehorse
America/Winnipeg	America/Winnipeg
America/Yakutat	America/Yakutat
America/Yellowknife	America/Yellowknife
Antarctica/Casey	Antarctica/Casey
Antarctica/Davis	Antarctica/Davis
Antarctica/DumontDUrville	Antarctica/DumontDUrville
Antarctica/Macquarie	Antarctica/Macquarie
Antarctica/Mawson	Antarctica/Mawson
Antarctica/McMurdo	NZ
Antarctica/Palmer	Antarctica/Palmer

name	abbrev
Antarctica/Rothera	Antarctica/Rothera
Antarctica/South_Pole	NZ
Antarctica/Syowa	Antarctica/Syowa
Antarctica/Troll	Antarctica/Troll
Antarctica/Vostok	Antarctica/Vostok
Arctic/Longyearbyen	Arctic/Longyearbyen
Asia/Aden	Asia/Aden
Asia/Almaty	Asia/Almaty
Asia/Amman	Asia/Amman
Asia/Anadyr	Asia/Anadyr
Asia/Aqtau	Asia/Aqtau
Asia/Aqtobe	Asia/Aqtobe
Asia/Ashgabat	Asia/Ashgabat
Asia/Ashkhabad	Asia/Ashkhabad
Asia/Atyrau	Asia/Atyrau
Asia/Baghdad	Asia/Baghdad
Asia/Bahrain	Asia/Bahrain
Asia/Baku	Asia/Baku
Asia/Bangkok	Asia/Bangkok
Asia/Barnaul	Asia/Barnaul
Asia/Beirut	Asia/Beirut
Asia/Bishkek	Asia/Bishkek
Asia/Brunei	Asia/Brunei
Asia/Calcutta	IST
Asia/Chita	Asia/Chita
Asia/Choibalsan	Asia/Choibalsan
Asia/Chongqing	CTT
Asia/Chungking	CTT
Asia/Colombo	Asia/Colombo
Asia/Dacca	BST
Asia/Damascus	Asia/Damascus
Asia/Dhaka	BST
Asia/Dili	Asia/Dili
Asia/Dubai	Asia/Dubai
Asia/Dushanbe	Asia/Dushanbe
Asia/Famagusta	Asia/Famagusta
Asia/Gaza	Asia/Gaza
Asia/Harbin	CTT
Asia/Hebron	Asia/Hebron

name	abbrev
Asia/Ho_Chi_Minh	VST
Asia/Hong_Kong	Hongkong
Asia/Hovd	Asia/Hovd
Asia/Irkutsk	Asia/Irkutsk
Asia/Istanbul	Turkey
Asia/Jakarta	Asia/Jakarta
Asia/Jayapura	Asia/Jayapura
Asia/Jerusalem	Israel
Asia/Kabul	Asia/Kabul
Asia/Kamchatka	Asia/Kamchatka
Asia/Karachi	PLT
Asia/Kashgar	Asia/Kashgar
Asia/Kathmandu	Asia/Kathmandu
Asia/Katmandu	Asia/Katmandu
Asia/Khandyga	Asia/Khandyga
Asia/Kolkata	IST
Asia/Krasnoyarsk	Asia/Krasnoyarsk
Asia/Kuala_Lumpur	Singapore
Asia/Kuching	Asia/Kuching
Asia/Kuwait	Asia/Kuwait
Asia/Macao	Asia/Macao
Asia/Macau	Asia/Macau
Asia/Magadan	Asia/Magadan
Asia/Makassar	Asia/Makassar
Asia/Manila	Asia/Manila
Asia/Muscat	Asia/Muscat
Asia/Nicosia	Asia/Nicosia
Asia/Novokuznetsk	Asia/Novokuznetsk
Asia/Novosibirsk	Asia/Novosibirsk
Asia/Omsk	Asia/Omsk
Asia/Oral	Asia/Oral
Asia/Phnom_Penh	Asia/Phnom_Penh
Asia/Pontianak	Asia/Pontianak
Asia/Pyongyang	Asia/Pyongyang
Asia/Qatar	Asia/Qatar
Asia/Qostanay	Asia/Qostanay
Asia/Qyzylorda	Asia/Qyzylorda
Asia/Rangoon	Asia/Rangoon
Asia/Riyadh	Asia/Riyadh

name	abbrev
Asia/Saigon	VST
Asia/Sakhalin	Asia/Sakhalin
Asia/Samarkand	Asia/Samarkand
Asia/Seoul	ROK
Asia/Shanghai	CTT
Asia/Singapore	Singapore
Asia/Srednekolymsk	Asia/Srednekolymsk
Asia/Taipei	ROC
Asia/Tashkent	Asia/Tashkent
Asia/Tbilisi	Asia/Tbilisi
Asia/Tehran	Iran
Asia/Tel_Aviv	Israel
Asia/Thimbu	Asia/Thimbu
Asia/Thimphu	Asia/Thimphu
Asia/Tokyo	JST
Asia/Tomsk	Asia/Tomsk
Asia/Ujung_Pandang	Asia/Ujung_Pandang
Asia/Ulaanbaatar	Asia/Ulaanbaatar
Asia/Ulan_Bator	Asia/Ulan_Bator
Asia/Urumqi	Asia/Urumqi
Asia/Ust-Nera	Asia/Ust-Nera
Asia/Vientiane	Asia/Vientiane
Asia/Vladivostok	Asia/Vladivostok
Asia/Yakutsk	Asia/Yakutsk
Asia/Yangon	Asia/Yangon
Asia/Yekaterinburg	Asia/Yekaterinburg
Asia/Yerevan	NET
Atlantic/Azores	Atlantic/Azores
Atlantic/Bermuda	Atlantic/Bermuda
Atlantic/Canary	Atlantic/Canary
Atlantic/Cape_Verde	Atlantic/Cape_Verde
Atlantic/Faeroe	Atlantic/Faeroe
Atlantic/Faroe	Atlantic/Faroe
Atlantic/Jan_Mayen	Atlantic/Jan_Mayen
Atlantic/Madeira	Atlantic/Madeira
Atlantic/Reykjavik	Iceland
Atlantic/South_Georgia	Atlantic/South_Georgia
Atlantic/St_Helena	Iceland
Atlantic/Stanley	Atlantic/Stanley

name	abbrev
Australia/ACT	AET
Australia/Adelaide	Australia/Adelaide
Australia/Brisbane	Australia/Brisbane
Australia/Broken_Hill	Australia/Broken_Hill
Australia/Canberra	AET
Australia/Currie	Australia/Currie
Australia/Darwin	ACT
Australia/Eucla	Australia/Eucla
Australia/Hobart	Australia/Hobart
Australia/LHI	Australia/LHI
Australia/Lindeman	Australia/Lindeman
Australia/Lord_Howe	Australia/Lord_Howe
Australia/Melbourne	Australia/Melbourne
Australia/NSW	AET
Australia/North	ACT
Australia/Perth	Australia/Perth
Australia/Queensland	Australia/Queensland
Australia/South	Australia/South
Australia/Sydney	AET
Australia/Tasmania	Australia/Tasmania
Australia/Victoria	Australia/Victoria
Australia/West	Australia/West
Australia/Yancowinna	Australia/Yancowinna
BET	BET
BST	BST
Brazil/Acre	Brazil/Acre
Brazil/DeNoronha	Brazil/DeNoronha
Brazil/East	BET
Brazil/West	Brazil/West
CAT	CAT
CET	CET
CNT	CNT
CST	CST
CST6CDT	CST
CTT	CTT
Canada/Atlantic	Canada/Atlantic
Canada/Central	Canada/Central
Canada/East-Saskatchewan	Canada/East-Saskatchewan
Canada/Eastern	Canada/Eastern

name	abbrev
Canada/Mountain	Canada/Mountain
Canada/Newfoundland	CNT
Canada/Pacific	Canada/Pacific
Canada/Saskatchewan	Canada/Saskatchewan
Canada/Yukon	Canada/Yukon
Chile/Continental	Chile/Continental
Chile/EasterIsland	Chile/EasterIsland
Cuba	Cuba
EAT	EAT
ECT	ECT
EET	EET
EST	EST
EST5EDT	EST5EDT
Egypt	ART
Eire	Eire
Etc/GMT	GMT
Etc/GMT+0	GMT
Etc/GMT+1	Etc/GMT+1
Etc/GMT+10	Etc/GMT+10
Etc/GMT+11	Etc/GMT+11
Etc/GMT+12	Etc/GMT+12
Etc/GMT+2	Etc/GMT+2
Etc/GMT+3	Etc/GMT+3
Etc/GMT+4	Etc/GMT+4
Etc/GMT+5	Etc/GMT+5
Etc/GMT+6	Etc/GMT+6
Etc/GMT+7	Etc/GMT+7
Etc/GMT+8	Etc/GMT+8
Etc/GMT+9	Etc/GMT+9
Etc/GMT-0	GMT
Etc/GMT-1	Etc/GMT-1
Etc/GMT-10	Etc/GMT-10
Etc/GMT-11	Etc/GMT-11
Etc/GMT-12	Etc/GMT-12
Etc/GMT-13	Etc/GMT-13
Etc/GMT-14	Etc/GMT-14
Etc/GMT-2	Etc/GMT-2
Etc/GMT-3	Etc/GMT-3
Etc/GMT-4	Etc/GMT-4

name	abbrev
Etc/GMT-5	Etc/GMT-5
Etc/GMT-6	Etc/GMT-6
Etc/GMT-7	Etc/GMT-7
Etc/GMT-8	Etc/GMT-8
Etc/GMT-9	Etc/GMT-9
Etc/GMT0	GMT
Etc/Greenwich	GMT
Etc/UCT	UCT
Etc/UTC	UCT
Etc/Universal	UCT
Etc/Zulu	UCT
Europe/Amsterdam	CET
Europe/Andorra	Europe/Andorra
Europe/Astrakhan	Europe/Astrakhan
Europe/Athens	EET
Europe/Belfast	GB
Europe/Belgrade	Europe/Belgrade
Europe/Berlin	Europe/Berlin
Europe/Bratislava	Europe/Bratislava
Europe/Brussels	CET
Europe/Bucharest	Europe/Bucharest
Europe/Budapest	Europe/Budapest
Europe/Busingen	Europe/Busingen
Europe/Chisinau	Europe/Chisinau
Europe/Copenhagen	Europe/Copenhagen
Europe/Dublin	Eire
Europe/Gibraltar	Europe/Gibraltar
Europe/Guernsey	GB
Europe/Helsinki	Europe/Helsinki
Europe/Isle_of_Man	GB
Europe/Istanbul	Turkey
Europe/Jersey	GB
Europe/Kaliningrad	Europe/Kaliningrad
Europe/Kiev	Europe/Kiev
Europe/Kirov	Europe/Kirov
Europe/Kyiv	Europe/Kyiv
Europe/Lisbon	WET
Europe/Ljubljana	Europe/Ljubljana
Europe/London	GB

name	abbrev
Europe/Luxembourg	CET
Europe/Madrid	Europe/Madrid
Europe/Malta	Europe/Malta
Europe/Mariehamn	Europe/Mariehamn
Europe/Minsk	Europe/Minsk
Europe/Monaco	ECT
Europe/Moscow	W-SU
Europe/Nicosia	Europe/Nicosia
Europe/Oslo	Europe/Oslo
Europe/Paris	ECT
Europe/Podgorica	Europe/Podgorica
Europe/Prague	Europe/Prague
Europe/Riga	Europe/Riga
Europe/Rome	Europe/Rome
Europe/Samara	Europe/Samara
Europe/San_Marino	Europe/San_Marino
Europe/Sarajevo	Europe/Sarajevo
Europe/Saratov	Europe/Saratov
Europe/Simferopol	Europe/Simferopol
Europe/Skopje	Europe/Skopje
Europe/Sofia	Europe/Sofia
Europe/Stockholm	Europe/Stockholm
Europe/Tallinn	Europe/Tallinn
Europe/Tirane	Europe/Tirane
Europe/Tiraspol	Europe/Tiraspol
Europe/Ulyanovsk	Europe/Ulyanovsk
Europe/Uzhgorod	Europe/Uzhgorod
Europe/Vaduz	Europe/Vaduz
Europe/Vatican	Europe/Vatican
Europe/Vienna	Europe/Vienna
Europe/Vilnius	Europe/Vilnius
Europe/Volgograd	Europe/Volgograd
Europe/Warsaw	Poland
Europe/Zagreb	Europe/Zagreb
Europe/Zaporozhye	Europe/Zaporozhye
Europe/Zurich	Europe/Zurich
Factory	Factory
GB	GB
GB-Eire	GB

name	abbrev
GMT	GMT
GMT+0	GMT
GMT-0	GMT
GMT0	GMT
Greenwich	GMT
HST	HST
Hongkong	Hongkong
IET	IET
IST	IST
Iceland	Iceland
Indian/Antananarivo	EAT
Indian/Chagos	Indian/Chagos
Indian/Christmas	Indian/Christmas
Indian/Cocos	Indian/Cocos
Indian/Comoro	EAT
Indian/Kerguelen	Indian/Kerguelen
Indian/Mahe	Indian/Mahe
Indian/Maldives	Indian/Maldives
Indian/Mauritius	Indian/Mauritius
Indian/Mayotte	EAT
Indian/Reunion	Indian/Reunion
Iran	Iran
Israel	Israel
JST	JST
Jamaica	Jamaica
Japan	JST
Kwajalein	Kwajalein
Libya	Libya
MET	CET
MIT	MIT
MST	MST
MST7MDT	Navajo
Mexico/BajaNorte	Mexico/BajaNorte
Mexico/BajaSur	Mexico/BajaSur
Mexico/General	Mexico/General
NET	NET
NST	NZ
NZ	NZ
NZ-CHAT	NZ-CHAT

name	abbrev
Navajo	Navajo
PLT	PLT
PNT	MST
PRC	CTT
PRT	PRT
PST	PST
PST8PDT	PST
Pacific/Apia	MIT
Pacific/Auckland	NZ
Pacific/Bougainville	Pacific/Bougainville
Pacific/Chatham	NZ-CHAT
Pacific/Chuuk	Pacific/Chuuk
Pacific/Easter	Pacific/Easter
Pacific/Efate	Pacific/Efate
Pacific/Enderbury	Pacific/Enderbury
Pacific/Fakaofo	Pacific/Fakaofo
Pacific/Fiji	Pacific/Fiji
Pacific/Funafuti	Pacific/Funafuti
Pacific/Galapagos	Pacific/Galapagos
Pacific/Gambier	Pacific/Gambier
Pacific/Guadalcanal	SST
Pacific/Guam	Pacific/Guam
Pacific/Honolulu	HST
Pacific/Johnston	HST
Pacific/Kanton	Pacific/Kanton
Pacific/Kiritimati	Pacific/Kiritimati
Pacific/Kosrae	Pacific/Kosrae
Pacific/Kwajalein	Kwajalein
Pacific/Majuro	Pacific/Majuro
Pacific/Marquesas	Pacific/Marquesas
Pacific/Midway	Pacific/Midway
Pacific/Nauru	Pacific/Nauru
Pacific/Niue	Pacific/Niue
Pacific/Norfolk	Pacific/Norfolk
Pacific/Noumea	Pacific/Noumea
Pacific/Pago_Pago	Pacific/Pago_Pago
Pacific/Palau	Pacific/Palau
Pacific/Pitcairn	Pacific/Pitcairn
Pacific/Pohnpei	SST

name	abbrev
Pacific/Ponape	SST
Pacific/Port_Moresby	Pacific/Port_Moresby
Pacific/Rarotonga	Pacific/Rarotonga
Pacific/Saipan	Pacific/Saipan
Pacific/Samoa	Pacific/Samoa
Pacific/Tahiti	Pacific/Tahiti
Pacific/Tarawa	Pacific/Tarawa
Pacific/Tongatapu	Pacific/Tongatapu
Pacific/Truk	Pacific/Truk
Pacific/Wake	Pacific/Wake
Pacific/Wallis	Pacific/Wallis
Pacific/Yap	Pacific/Yap
Poland	Poland
Portugal	WET
ROC	ROC
ROK	ROK
SST	SST
Singapore	Singapore
SystemV/AST4	SystemV/AST4
SystemV/AST4ADT	SystemV/AST4ADT
SystemV/CST6	SystemV/CST6
SystemV/CST6CDT	SystemV/CST6CDT
SystemV/EST5	SystemV/EST5
SystemV/EST5EDT	SystemV/EST5EDT
SystemV/HST10	SystemV/HST10
SystemV/MST7	SystemV/MST7
SystemV/MST7MDT	SystemV/MST7MDT
SystemV/PST8	SystemV/PST8
SystemV/PST8PDT	SystemV/PST8PDT
SystemV/YST9	SystemV/YST9
SystemV/YST9YDT	SystemV/YST9YDT
Turkey	Turkey
UCT	UCT
US/Alaska	AST
US/Aleutian	US/Aleutian
US/Arizona	MST
US/Central	CST
US/East-Indiana	IET
US/Eastern	EST5EDT

name	abbrev
US/Hawaii	HST
US/Indiana-Starke	US/Indiana-Starke
US/Michigan	US/Michigan
US/Mountain	Navajo
US/Pacific	PST
US/Pacific-New	PST
US/Samoa	US/Samoa
UTC	UCT
Universal	UCT
VST	VST
W-SU	W-SU
WET	WET
Zulu	UCT

Union Type

A `UNION` *type* (not to be confused with the SQL `UNION` operator) is a nested type capable of holding one of multiple "alternative" values, much like the `union` in C. The main difference being that these `UNION` types are *tagged unions* and thus always carry a discriminator "tag" which signals which alternative it is currently holding, even if the inner value itself is null. `UNION` types are thus more similar to C++17's `std::variant`, Rust's `Enum` or the "sum type" present in most functional languages.

`UNION` types must always have at least one member, and while they can contain multiple members of the same type, the tag names must be unique. `UNION` types can have at most 256 members.

Under the hood, `UNION` types are implemented on top of `STRUCT` types, and simply keep the "tag" as the first entry.

`UNION` values can be created with the `union_value(tag := expr)` function or by casting from a member type.

Example

Create a table with a `UNION` column:

```
CREATE TABLE tbl1 (u UNION(num INTEGER, str VARCHAR));
INSERT INTO tbl1 VALUES (1), ('two'), (union_value(str := 'three'));
```

Any type can be implicitly cast to a `UNION` containing the type. Any `UNION` can also be implicitly cast to another `UNION` if the source `UNION` members are a subset of the target's (if the cast is unambiguous).

`UNION` uses the member types' `VARCHAR` cast functions when casting to `VARCHAR`:

```
SELECT u FROM tbl1;
```

u
1
two
three

Select all the `str` members:

```
SELECT union_extract(u, 'str') AS str
FROM tbl1;
```

str
NULL
two
three

Alternatively, you can use 'dot syntax' similarly to STRUCTs.

```
SELECT u.str
FROM tbl1;
```

str
NULL
two
three

Select the currently active tag from the `UNION` as an `ENUM`.

```
SELECT union_tag(u) AS t
FROM tbl1;
```

t
num
str
str

Union Casts

Compared to other nested types, `UNION`s allow a set of implicit casts to facilitate unintrusive and natural usage when working with their members as "subtypes". However, these casts have been designed with two principles in mind, to avoid ambiguity and to avoid casts that could lead to loss of information. This prevents `UNION`s from being completely "transparent", while still allowing `UNION` types to have a "supertype" relationship with their members.

Thus `UNION` types can't be implicitly cast to any of their member types in general, since the information in the other members not matching the target type would be "lost". If you want to coerce a `UNION` into one of its members, you should use the `union_extract` function explicitly instead.

The only exception to this is when casting a `UNION` to `VARCHAR`, in which case the members will all use their corresponding `VARCHAR` casts. Since everything can be cast to `VARCHAR`, this is "safe" in a sense.

Casting to Unions

A type can always be implicitly cast to a `UNION` if it can be implicitly cast to one of the `UNION` member types.

- If there are multiple candidates, the built in implicit casting priority rules determine the target type. For example, a `FLOAT` → `UNION(i INTEGER, v VARCHAR)` cast will always cast the `FLOAT` to the `INTEGER` member before `VARCHAR`.
- If the cast still is ambiguous, i.e., there are multiple candidates with the same implicit casting priority, an error is raised. This usually happens when the `UNION` contains multiple members of the same type, e.g., a `FLOAT` → `UNION(i INTEGER, num INTEGER)` is always ambiguous.

So how do we disambiguate if we want to create a `UNION` with multiple members of the same type? By using the `union_value` function, which takes a keyword argument specifying the tag. For example, `union_value(num := 2::INTEGER)` will create a `UNION` with a single member of type `INTEGER` with the tag `num`. This can then be used to disambiguate in an explicit (or implicit, read on below!) `UNION` to `UNION` cast, like `CAST(union_value(b := 2) AS UNION(a INTEGER, b INTEGER))`.

Casting between Unions

`UNION` types can be cast between each other if the source type is a "subset" of the target type. In other words, all the tags in the source `UNION` must be present in the target `UNION`, and all the types of the matching tags must be implicitly castable between source and target. In essence, this means that `UNION` types are covariant with respect to their members.

Ok	Source	Target	Comments
✅	`UNION(a A, b B)`	`UNION(a A, b B, c C)`	
✅	`UNION(a A, b B)`	`UNION(a A, b C)`	if B can be implicitly cast to C
❌	`UNION(a A, b B, c C)`	`UNION(a A, b B)`	
❌	`UNION(a A, b B)`	`UNION(a A, b C)`	if B can't be implicitly cast to C
❌	`UNION(A, B, D)`	`UNION(A, B, C)`	

Comparison and Sorting

Since `UNION` types are implemented on top of `STRUCT` types internally, they can be used with all the comparison operators as well as in both `WHERE` and `HAVING` clauses with the same semantics as `STRUCT`s. The "tag" is always stored as the first struct entry, which ensures that the `UNION` types are compared and ordered by "tag" first.

Functions

See Union Functions.

Typecasting

Typecasting is an operation that converts a value in one particular data type to the closest corresponding value in another data type. Like other SQL engines, DuckDB supports both implicit and explicit typecasting.

Explicit Casting

Explicit typecasting is performed by using a `CAST` expression. For example, `CAST(col AS VARCHAR)` or `col::VARCHAR` explicitly cast the column `col` to `VARCHAR`. See the cast page for more information.

Implicit Casting

In many situations, the system will add casts by itself. This is called *implicit* casting and happens, for example, when a function is called with an argument that does not match the type of the function but can be cast to the required type.

Implicit casts can only be added for a number of type combinations, and is generally only possible when the cast cannot fail. For example, an implicit cast can be added from `INTEGER` to `DOUBLE` – but not from `DOUBLE` to `INTEGER`.

Consider the function `sin(DOUBLE)`. This function takes as input argument a column of type `DOUBLE`, however, it can be called with an integer as well: `sin(1)`. The integer is converted into a double before being passed to the `sin` function.

> **Tip.** To check whether a type can be implicitly cast to another type, use the `can_cast_implicitly` function.

Combination Casting

When values of different types need to be combined to an unspecified joint parent type, the system will perform implicit casts to an automatically selected parent type. For example, `list_value(1::INT64, 1::UINT64)` creates a list of type `INT128[]`. The implicit casts performed in this situation are sometimes more lenient than regular implicit casts. For example, a `BOOL` value may be cast to `INT` (with `true` mapping to `1` and `false` to `0`) even though this is not possible for regular implicit casts.

This *combination casting* occurs for comparisons (= / < / >), set operations (`UNION` / `EXCEPT` / `INTERSECT`), and nested type constructors (`list_value` / `[...]` / `MAP`).

Casting Operations Matrix

Values of a particular data type cannot always be cast to any arbitrary target data type. The only exception is the `NULL` value – which can always be converted between types. The following matrix describes which conversions are supported. When implicit casting is allowed, it implies that explicit casting is also possible.

from \ to	BLOB	INTERVAL	DATE	TIME	TIMESTAMP	BOOLEAN	BIT	DOUBLE	FLOAT	HUGEINT	BIGINT	INTEGER	SMALLINT	TINYINT	UBIGINT	UINTEGER	USMALLINT	UTINYINT	DECIMAL	UUID	VARCHAR
BLOB		✗	✗	✗	✗	✗	■	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	■
INTERVAL	✗		✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	●
DATE	✗	✗		✗	●	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	●
TIME	✗	✗	✗		✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	●
TIMESTAMP	✗	✗	■	■		✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	●
BOOLEAN	✗	✗	✗	✗	✗		■	■	■	■	■	■	■	■	■	■	■	■	■	✗	●
BIT	■	✗	✗	✗	✗	✗		■	■	■	■	■	✗	✗	■	■	✗	✗	✗	✗	●
DOUBLE	✗	✗	✗	✗	✗	■	■		✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	●
FLOAT	✗	✗	✗	✗	✗	■	■	●		✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	●
HUGEINT	✗	✗	✗	✗	✗	■	■	●	●		✗	✗	✗	✗	✗	✗	✗	✗	●	✗	●
BIGINT	✗	✗	✗	✗	✗	■	■	●	●	●		✗	✗	✗	✗	✗	✗	✗	●	✗	●
INTEGER	✗	✗	✗	✗	✗	■	■	●	●	●	●		✗	✗	✗	✗	✗	✗	●	✗	●
SMALLINT	✗	✗	✗	✗	✗	■	■	●	●	●	●	●		✗	✗	✗	✗	✗	●	✗	●
TINYINT	✗	✗	✗	✗	✗	■	■	●	●	●	●	●	●		✗	✗	✗	✗	●	✗	●
UBIGINT	✗	✗	✗	✗	✗	■	■	●	●	●	■	■	■	■		■	■	■	●	✗	●
UINTEGER	✗	✗	✗	✗	✗	■	■	●	●	●	●	■	■	■	●		■	■	●	✗	●
USMALLINT	✗	✗	✗	✗	✗	■	■	●	●	●	●	●	■	■	●	●		■	●	✗	●
UTINYINT	✗	✗	✗	✗	✗	■	■	●	●	●	●	●	●	■	●	●	●		●	✗	●
DECIMAL	✗	✗	✗	✗	✗	■	✗	●	●	■	■	■	■	✗	✗	✗	✗	✗		✗	●
UUID	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗		●
VARCHAR	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	

Even though a casting operation is supported based on the source and target data type, it does not necessarily mean the cast operation will succeed at runtime.

> **Deprecated.** Prior to version 0.10.0, DuckDB allowed any type to be implicitly cast to VARCHAR during function binding. Version 0.10.0 introduced a breaking change which no longer allows implicit casts to VARCHAR. The `old_implicit_casting` configuration option setting can be used to revert to the old behavior. However, please note that this flag will be deprecated in the future.

Lossy Casts

Casting operations that result in loss of precision are allowed. For example, it is possible to explicitly cast a numeric type with fractional digits – such as `DECIMAL`, `FLOAT` or `DOUBLE` – to an integral type like `INTEGER` or `BIGINT`. The number will be rounded.

```
SELECT CAST(3.1 AS INTEGER);  -- 3
SELECT CAST(3.5 AS INTEGER);  -- 4
SELECT CAST(-1.7 AS INTEGER); -- -2
```

Overflows

Casting operations that would result in a value overflow throw an error. For example, the value `999` is too large to be represented by the `TINYINT` data type. Therefore, an attempt to cast that value to that type results in a runtime error:

```
SELECT CAST(999 AS TINYINT);
```

```
Conversion Error:
Type INT32 with value 999 can't be cast because the value is out of range for the destination type INT8
```

So even though the cast operation from `INTEGER` to `TINYINT` is supported, it is not possible for this particular value. TRY_CAST can be used to convert the value into `NULL` instead of throwing an error.

Varchar

The VARCHAR type acts as a univeral target: any arbitrary value of any arbitrary type can always be cast to the VARCHAR type. This type is also used for displaying values in the shell.

```
SELECT CAST(42.5 AS VARCHAR);
```

Casting from VARCHAR to another data type is supported, but can raise an error at runtime if DuckDB cannot parse and convert the provided text to the target data type.

```
SELECT CAST('NotANumber' AS INTEGER);
```

In general, casting to VARCHAR is a lossless operation and any type can be cast back to the original type after being converted into text.

```
SELECT CAST(CAST([1, 2, 3] AS VARCHAR) AS INTEGER[]);
```

Literal Types

Integer literals (such as `42`) and string literals (such as `'string'`) have special implicit casting rules. See the literal types page for more information.

Lists / Arrays

Lists can be explicitly cast to other lists using the same casting rules. The cast is applied to the children of the list. For example, if we convert an `INTEGER[]` list to a `VARCHAR[]` list, the child `INTEGER` elements are individually cast to `VARCHAR` and a new list is constructed.

```
SELECT CAST([1, 2, 3] AS VARCHAR[]);
```

Arrays

Arrays follow the same casting rules as lists. In addition, arrays can be implicitly cast to lists of the same type. For example, an `INTEGER[3]` array can be implicitly cast to an `INTEGER[]` list.

Structs

Structs can be cast to other structs as long as they share at least one field.

> The rationale behind this requirement is to help avoid unintended errors. If two structs do not have any fields in common, then the cast was likely not intended.

```
SELECT CAST({'a': 42} AS STRUCT(a VARCHAR));
```

Fields that exist in the target struct, but that do not exist in the source struct, default to `NULL`.

```
SELECT CAST({'a': 42} AS STRUCT(a VARCHAR, b VARCHAR));
```

Fields that only exist in the source struct are ignored.

```
SELECT CAST({'a': 42, 'b': 43} AS STRUCT(a VARCHAR));
```

The names of the struct can also be in a different order. The fields of the struct will be reshuffled based on the names of the structs.

```
SELECT CAST({'a': 42, 'b': 84} AS STRUCT(b VARCHAR, a VARCHAR));
```

Struct casting behaves differently when combined with the `UNION [ALL] BY NAME` operation. In that case, the fields of the resulting struct are the superset of all fields of the input structs. This logic also applies recursively to potentially nested structs.

```
SELECT {'outer1': {'inner1': 42, 'inner2': 42}} AS c
UNION ALL BY NAME
SELECT {'outer1': {'inner2': 'hello', 'inner3': 'world'}, 'outer2': '100'} AS c;
```

```
SELECT [{'a': 42}, {'b': 84}];
```

Unions

Union casting rules can be found on the `UNION type page`.

Expressions

Expressions

An expression is a combination of values, operators and functions. Expressions are highly composable, and range from very simple to arbitrarily complex. They can be found in many different parts of SQL statements. In this section, we provide the different types of operators and functions that can be used within expressions.

CASE Expression

The CASE expression performs a switch based on a condition. The basic form is identical to the ternary condition used in many programming languages (`CASE WHEN cond THEN a ELSE b END` is equivalent to `cond ? a : b`). With a single condition this can be expressed with `IF(cond, a, b)`.

```
CREATE OR REPLACE TABLE integers AS SELECT unnest([1, 2, 3]) AS i;
SELECT i, CASE WHEN i > 2 THEN 1 ELSE 0 END AS test
FROM integers;
```

i	test
1	0
2	0
3	1

This is equivalent to:

```
SELECT i, IF(i > 2, 1, 0) AS test
FROM integers;
```

The `WHEN cond THEN expr` part of the CASE expression can be chained, whenever any of the conditions returns true for a single tuple, the corresponding expression is evaluated and returned.

```
CREATE OR REPLACE TABLE integers AS SELECT unnest([1, 2, 3]) AS i;
SELECT i, CASE WHEN i = 1 THEN 10 WHEN i = 2 THEN 20 ELSE 0 END AS test
FROM integers;
```

i	test
1	10
2	20
3	0

The `ELSE` clause of the CASE expression is optional. If no `ELSE` clause is provided and none of the conditions match, the CASE expression will return `NULL`.

```
CREATE OR REPLACE TABLE integers AS SELECT unnest([1, 2, 3]) AS i;
SELECT i, CASE WHEN i = 1 THEN 10 END AS test
FROM integers;
```

i	test
1	10
2	NULL
3	NULL

It is also possible to provide an individual expression after the CASE but before the WHEN. When this is done, the CASE expression is effectively transformed into a switch statement.

```
CREATE OR REPLACE TABLE integers AS SELECT unnest([1, 2, 3]) AS i;
SELECT i, CASE i WHEN 1 THEN 10 WHEN 2 THEN 20 WHEN 3 THEN 30 END AS test
FROM integers;
```

i	test
1	10
2	20
3	30

This is equivalent to:

```
SELECT i, CASE WHEN i = 1 THEN 10 WHEN i = 2 THEN 20 WHEN i = 3 THEN 30 END AS test
FROM integers;
```

Casting

Casting refers to the operation of converting a value in a particular data type to the corresponding value in another data type. Casting can occur either implicitly or explicitly. The syntax described here performs an explicit cast. More information on casting can be found on the typecasting page.

Explicit Casting

The standard SQL syntax for explicit casting is `CAST(expr AS TYPENAME)`, where TYPENAME is a name (or alias) of one of DuckDB's data types. DuckDB also supports the shorthand `expr::TYPENAME`, which is also present in PostgreSQL.

```
SELECT CAST(i AS VARCHAR) AS i
FROM generate_series(1, 3) tbl(i);
```

i
1
2
3

```
SELECT i::DOUBLE AS i
FROM generate_series(1, 3) tbl(i);
```

i
1.0
2.0
3.0

Casting Rules

Not all casts are possible. For example, it is not possible to convert an `INTEGER` to a `DATE`. Casts may also throw errors when the cast could not be successfully performed. For example, trying to cast the string `'hello'` to an `INTEGER` will result in an error being thrown.

```
SELECT CAST('hello' AS INTEGER);
```

```
Conversion Error:
Could not convert string 'hello' to INT32
```

The exact behavior of the cast depends on the source and destination types. For example, when casting from `VARCHAR` to any other type, the string will be attempted to be converted.

TRY_CAST

`TRY_CAST` can be used when the preferred behavior is not to throw an error, but instead to return a `NULL` value. `TRY_CAST` will never throw an error, and will instead return `NULL` if a cast is not possible.

```
SELECT TRY_CAST('hello' AS INTEGER) AS i;
```

i
NULL

`cast_to_type` Function

The `cast_to_type` function allows generating a cast from an expression to the type of another column. For example:

```
SELECT cast_to_type('42', NULL::INTEGER) AS result;
```

res
int32
42

This function is primarily useful in macros, as it allows you to maintain types. This helps with making generic macros that operate on different types. For example, the following macro adds to a number if the input is an `INTEGER`:

```
CREATE TABLE tbl(i INT, s VARCHAR);
INSERT INTO tbl VALUES (42, 'hello world');

CREATE MACRO conditional_add(col, nr) AS
    CASE
```

```
        WHEN typeof(col) == 'INTEGER' THEN cast_to_type(col::INTEGER + nr, col)
        ELSE col
    END;
SELECT conditional_add(COLUMNS(*), 100) FROM tbl;
```

i	s
int32	varchar
142	hello world

Note that the CASE statement needs to return the same type in all code paths. We can perform the addition on any input column by adding a cast to the desired type – but we need to cast the result of the addition back to the source type to make the binding work.

Collations

Collations provide rules for how text should be sorted or compared in the execution engine. Collations are useful for localization, as the rules for how text should be ordered are different for different languages or for different countries. These orderings are often incompatible with one another. For example, in English the letter `y` comes between `x` and `z`. However, in Lithuanian the letter `y` comes between the `i` and `j`. For that reason, different collations are supported. The user must choose which collation they want to use when performing sorting and comparison operations.

By default, the `BINARY` collation is used. That means that strings are ordered and compared based only on their binary contents. This makes sense for standard ASCII characters (i.e., the letters A-Z and numbers 0-9), but generally does not make much sense for special unicode characters. It is, however, by far the fastest method of performing ordering and comparisons. Hence it is recommended to stick with the `BINARY` collation unless required otherwise.

> The `BINARY` collation is also available under the aliases `C` and `POSIX`.

> **Warning.** Collation support in DuckDB has some known limitations and has several planned improvements.

Using Collations

In the stand-alone installation of DuckDB three collations are included: `NOCASE`, `NOACCENT` and `NFC`. The `NOCASE` collation compares characters as equal regardless of their casing. The `NOACCENT` collation compares characters as equal regardless of their accents. The `NFC` collation performs NFC-normalized comparisons, see Unicode normalization for more information.

```
SELECT 'hello' = 'hElLO';
```

```
false
```

```
SELECT 'hello' COLLATE NOCASE = 'hElLO';
```

```
true
```

```
SELECT 'hello' = 'hëllo';
```

```
false
```

```
SELECT 'hello' COLLATE NOACCENT = 'hëllo';
```

```
true
```

Collations can be combined by chaining them using the dot operator. Note, however, that not all collations can be combined together. In general, the `NOCASE` collation can be combined with any other collator, but most other collations cannot be combined.

```
SELECT 'hello' COLLATE NOCASE = 'hElLÖ';
```

```
false
```

```
SELECT 'hello' COLLATE NOACCENT = 'hElLÖ';
```

```
false
```

```
SELECT 'hello' COLLATE NOCASE.NOACCENT = 'hElLÖ';
```

```
true
```

Default Collations

The collations we have seen so far have all been specified *per expression*. It is also possible to specify a default collator, either on the global database level or on a base table column. The PRAGMA `default_collation` can be used to specify the global default collator. This is the collator that will be used if no other one is specified.

```
SET default_collation = NOCASE;
SELECT 'hello' = 'HeLlo';
```

```
true
```

Collations can also be specified per-column when creating a table. When that column is then used in a comparison, the per-column collation is used to perform that comparison.

```
CREATE TABLE names (name VARCHAR COLLATE NOACCENT);
INSERT INTO names VALUES ('hännes');
```

```
SELECT name
FROM names
WHERE name = 'hannes';
```

```
hännes
```

Be careful here, however, as different collations cannot be combined. This can be problematic when you want to compare columns that have a different collation specified.

```
SELECT name
FROM names
WHERE name = 'hannes' COLLATE NOCASE;
```

```
ERROR: Cannot combine types with different collation!
```

```
CREATE TABLE other_names (name VARCHAR COLLATE NOCASE);
INSERT INTO other_names VALUES ('HÄNNES');
```

```
SELECT names.name AS name, other_names.name AS other_name
FROM names, other_names
WHERE names.name = other_names.name;
```

```
ERROR: Cannot combine types with different collation!
```

We need to manually overwrite the collation:

```
SELECT names.name AS name, other_names.name AS other_name
FROM names, other_names
WHERE names.name COLLATE NOACCENT.NOCASE = other_names.name COLLATE NOACCENT.NOCASE;
```

name	other_name
hännes	HÄNNES

ICU Collations

The collations we have seen so far are not region-dependent, and do not follow any specific regional rules. If you wish to follow the rules of a specific region or language, you will need to use one of the ICU collations. For that, you need to load the ICU extension.

Loading this extension will add a number of language and region specific collations to your database. These can be queried using PRAGMA `collations` command, or by querying the `pragma_collations` function.

```
PRAGMA collations;
SELECT list(collname) FROM pragma_collations();
```

```
[af, am, ar, ar_sa, as, az, be, bg, bn, bo, br, bs, ca, ceb, chr, cs, cy, da, de, de_at, dsb, dz, ee, el,
en, en_us, eo, es, et, fa, fa_af, ff, fi, fil, fo, fr, fr_ca, fy, ga, gl, gu, ha, haw, he, he_il, hi, hr,
hsb, hu, hy, icu_noaccent, id, id_id, ig, is, it, ja, ka, kk, kl, km, kn, ko, kok, ku, ky, lb, lkt, ln,
lo, lt, lv, mk, ml, mn, mr, ms, mt, my, nb, nb_no, ne, nfc, nl, nn, noaccent, nocase, om, or, pa, pa_in,
pl, ps, pt, ro, ru, sa, se, si, sk, sl, smn, sq, sr, sr_ba, sr_me, sr_rs, sv, sw, ta, te, th, tk, to, tr,
ug, uk, ur, uz, vi, wae, wo, xh, yi, yo, yue, yue_cn, zh, zh_cn, zh_hk, zh_mo, zh_sg, zh_tw, zu]
```

These collations can then be used as the other collations would be used before. They can also be combined with the NOCASE collation. For example, to use the German collation rules you could use the following code snippet:

```
CREATE TABLE strings (s VARCHAR COLLATE DE);
INSERT INTO strings VALUES ('Gabel'), ('Göbel'), ('Goethe'), ('Goldmann'), ('Göthe'), ('Götz');
SELECT * FROM strings ORDER BY s;
```

```
"Gabel", "Göbel", "Goethe", "Goldmann", "Göthe", "Götz"
```

Comparisons

Comparison Operators

The table below shows the standard comparison operators. Whenever either of the input arguments is NULL, the output of the comparison is NULL.

Operator	Description	Example	Result
<	less than	`2 < 3`	`true`
>	greater than	`2 > 3`	`false`
<=	less than or equal to	`2 <= 3`	`true`
>=	greater than or equal to	`4 >= NULL`	`NULL`
= or ==	equal	`NULL = NULL`	`NULL`
<> or !=	not equal	`2 <> 2`	`false`

The table below shows the standard distinction operators. These operators treat NULL values as equal.

Operator	Description	Example	Result
`IS DISTINCT FROM`	not equal, including NULL	`2 IS DISTINCT FROM NULL`	`true`
`IS NOT DISTINCT FROM`	equal, including NULL	`NULL IS NOT DISTINCT FROM NULL`	`true`

Combination Casting

When performing comparison on different types, DuckDB performs Combination Casting. These casts were introduced to make interactive querying more convenient and are in line with the casts performed by several programming languages but are often not compatible with PostgreSQL's behavior. For example, the following expressions evaluate and return `true` in DuckDB but fail in PostgreSQL.

```
SELECT 1 = true;
SELECT 1 = '1.1';
```

> It is not possible to enforce stricter type-checking for DuckDB's comparison operators. If you require stricter type-checking, we recommend creating a macro with the `typeof` function or implementing a user-defined function.

`BETWEEN` and `IS [NOT] NULL`

Besides the standard comparison operators there are also the `BETWEEN` and `IS (NOT) NULL` operators. These behave much like operators, but have special syntax mandated by the SQL standard. They are shown in the table below.

Note that `BETWEEN` and `NOT BETWEEN` are only equivalent to the examples below in the cases where both `a`, `x` and `y` are of the same type, as `BETWEEN` will cast all of its inputs to the same type.

Predicate	Description
`a BETWEEN x AND y`	equivalent to `x <= a AND a <= y`
`a NOT BETWEEN x AND y`	equivalent to `x > a OR a > y`
`expression IS NULL`	`true` if expression is `NULL`, `false` otherwise
`expression ISNULL`	alias for `IS NULL` (non-standard)
`expression IS NOT NULL`	`false` if expression is `NULL`, `true` otherwise
`expression NOTNULL`	alias for `IS NOT NULL` (non-standard)

> For the expression `BETWEEN x AND y`, `x` is used as the lower bound and `y` is used as the upper bound. Therefore, if `x > y`, the result will always be `false`.

IN Operator

The `IN` operator checks containment of the left expression inside the *collection* on the right hand side (RHS). Supported collections on the RHS are tuples, lists, maps and subqueries that return a single column.

`IN (val1, val2, ...)` (Tuple)

The `IN` operator on a tuple `(val1, val2, ...)` returns `true` if the expression is present in the RHS, `false` if the expression is not in the RHS and the RHS has no `NULL` values, or `NULL` if the expression is not in the RHS and the RHS has `NULL` values.

```
SELECT 'Math' IN ('CS', 'Math');
```

```
true
```

```
SELECT 'English' IN ('CS', 'Math');
```

```
false
```

```
SELECT 'Math' IN ('CS', 'Math', NULL);
```

```
true
```

```
SELECT 'English' IN ('CS', 'Math', NULL);
```

```
NULL
```

`IN [val1, val2, ...]` (List)

The `IN` operator works on lists according to the semantics used in Python. Unlike for the `IN tuple` operator, the presence of `NULL` values on the right hand side of the expression does not make a difference in the result:

```
SELECT 'Math' IN ['CS', 'Math', NULL];
```

```
true
```

```
SELECT 'English' IN ['CS', 'Math', NULL];
```

```
false
```

`IN` Map

The `IN` operator works on maps according to the semantics used in Python, i.e., it checks for the presence of keys (not values):

```
SELECT 'key1' IN MAP {'key1': 50, 'key2': 75};
```

```
true
```

```
SELECT 'key3' IN MAP {'key1': 50, 'key2': 75};
```

```
false
```

`IN` Subquery

The `IN` operator works with subqueries that return a single column. For example:

```
SELECT 42 IN (SELECT unnest([32, 42, 52]) AS x);
```

```
true
```

If the subquery returns more than one column, a Binder Error is thrown:

```
SELECT 42 IN (SELECT unnest([32, 42, 52]) AS x, 'a' AS y);
```

```
Binder Error:
Subquery returns 2 columns - expected 1
```

`NOT IN`

`NOT IN` can be used to check if an element is not present in the set. `x NOT IN y` is equivalent to `NOT (x IN y)`.

Logical Operators

The following logical operators are available: `AND`, `OR` and `NOT`. SQL uses a three-valuad logic system with `true`, `false` and `NULL`. Note that logical operators involving `NULL` do not always evaluate to `NULL`. For example, `NULL AND false` will evaluate to `false`, and `NULL OR true` will evaluate to `true`. Below are the complete truth tables.

Binary Operators: AND and OR

a	b	a AND b	a OR b
true	true	true	true
true	false	false	true
true	NULL	NULL	true
false	false	false	false
false	NULL	false	NULL
NULL	NULL	NULL	NULL

Unary Operator: NOT

a	NOT a
true	false
false	true
NULL	NULL

The operators AND and OR are commutative, that is, you can switch the left and right operand without affecting the result.

Star Expression

Syntax

The * expression can be used in a SELECT statement to select all columns that are projected in the FROM clause.

```
SELECT *
FROM tbl;
```

TABLE.* and STRUCT.*

The * expression can be prepended by a table name to select only columns from that table.

```
SELECT table_name.*
FROM table_name
JOIN other_table_name USING (id);
```

Similarly, the * expression can also be used to retrieve all keys from a struct as separate columns. This is particularly useful when a prior operation creates a struct of unknown shape, or if a query must handle any potential struct keys. See the STRUCT data type and STRUCT functions pages for more details on working with structs.

For example:

```
SELECT st.* FROM (SELECT {'x': 1, 'y': 2, 'z': 3} AS st);
```

x	y	z
1	2	3

EXCLUDE Clause

EXCLUDE allows you to exclude specific columns from the * expression.

```
SELECT * EXCLUDE (col)
FROM tbl;
```

REPLACE Clause

REPLACE allows you to replace specific columns by alternative expressions.

```
SELECT * REPLACE (col1 / 1_000 AS col1, col2 / 1_000 AS col2)
FROM tbl;
```

RENAME Clause

RENAME allows you to replace specific columns.

```
SELECT * RENAME (col1 AS height, col2 AS width)
FROM tbl;
```

Column Filtering via Pattern Matching Operators

The pattern matching operators `LIKE`, `GLOB`, `SIMILAR TO` and their variants allow you to select columns by matching their names to patterns.

```
SELECT * LIKE 'col%'
FROM tbl;
```

```
SELECT * GLOB 'col*'
FROM tbl;
```

```
SELECT * SIMILAR TO 'col.'
FROM tbl;
```

COLUMNS Expression

The COLUMNS expression is similar to the regular star expression, but additionally allows you to execute the same expression on the resulting columns.

```
CREATE TABLE numbers (id INTEGER, number INTEGER);
INSERT INTO numbers VALUES (1, 10), (2, 20), (3, NULL);
SELECT min(COLUMNS(*)), count(COLUMNS(*)) FROM numbers;
```

id	number	id	number
1	10	3	2

```
SELECT
    min(COLUMNS(* REPLACE (number + id AS number))),
    count(COLUMNS(* EXCLUDE (number)))
FROM numbers;
```

id	min(number := (number + id))	id
1	11	3

COLUMNS expressions can also be combined, as long as they contain the same star expression:

```
SELECT COLUMNS(*) + COLUMNS(*) FROM numbers;
```

id	number
2	20
4	40
6	NULL

COLUMNS Expression in a WHERE Clause

COLUMNS expressions can also be used in WHERE clauses. The conditions are applied to all columns and are combined using the logical AND operator.

```
SELECT *
FROM (
    SELECT 0 AS x, 1 AS y, 2 AS z
    UNION ALL
    SELECT 1 AS x, 2 AS y, 3 AS z
    UNION ALL
    SELECT 2 AS x, 3 AS y, 4 AS z
)
WHERE COLUMNS(*) > 1; -- equivalent to: x > 1 AND y > 1 AND z > 1
```

x	y	z
2	3	4

Regular Expressions in a COLUMNS Expression

COLUMNS expressions don't currently support the pattern matching operators, but they do supports regular expression matching by simply passing a string constant in place of the star:

```
SELECT COLUMNS('(id|numbers?)') FROM numbers;
```

id	number
1	10
2	20
3	NULL

Renaming Columns with Regular Expressions in a COLUMNS Expression

The matches of capture groups in regular expressions can be used to rename matching columns. The capture groups are one-indexed; `\0` is the original column name.

For example, to select the first three letters of colum names, run:

```
SELECT COLUMNS('(\w{3}).*') AS '\1' FROM numbers;
```

id	num
1	10
2	20
3	NULL

To remove a colon (`:`) character in the middle of a column name, run:

```
CREATE TABLE tbl ("Foo:Bar" INTEGER, "Foo:Baz" INTEGER, "Foo:Qux" INTEGER);
SELECT COLUMNS('(\w*):(\w*)') AS '\1\2' FROM tbl;
```

To add the original column name to the expression alias, run:

```
SELECT min(COLUMNS(*)) AS "min_\0" FROM numbers;
```

min_id	min_number
1	10

COLUMNS Lambda Function

COLUMNS also supports passing in a lambda function. The lambda function will be evaluated for all columns present in the FROM clause, and only columns that match the lambda function will be returned. This allows the execution of arbitrary expressions in order to select and rename columns.

```
SELECT COLUMNS(c -> c LIKE '%num%') FROM numbers;
```

number
10
20
NULL

COLUMNS List

COLUMNS also supports passing in a list of column names.

```
SELECT COLUMNS(['id', 'num']) FROM numbers;
```

id	num
1	10

id	num
2	20
3	NULL

*COLUMNS Unpacked Columns

The *COLUMNS clause is a variation of COLUMNS, which supports all of the previously mentioned capabilities. The difference is in how the expression expands.

*COLUMNS will expand in-place, much like the iterable unpacking behavior in Python, which inspired the * syntax. This implies that the expression expands into the parent expression. An example that shows this difference between COLUMNS and *COLUMNS:

With COLUMNS:

```
SELECT coalesce(COLUMNS(['a', 'b', 'c'])) AS result
FROM (SELECT NULL a, 42 b, true c);
```

result	result	result
NULL	42	true

With *COLUMNS, the expression expands in its parent expression `coalesce`, resulting in a single result column:

```
SELECT coalesce(*COLUMNS(['a', 'b', 'c'])) AS result
FROM (SELECT NULL AS a, 42 AS b, true AS c);
```

result
42

*COLUMNS also works with the `(*)` argument:

```
SELECT coalesce(*COLUMNS(*)) AS result
FROM (SELECT NULL a, 42 AS b, true AS c);
```

result
42

STRUCT.*

The * expression can also be used to retrieve all keys from a struct as separate columns. This is particularly useful when a prior operation creates a struct of unknown shape, or if a query must handle any potential struct keys. See the STRUCT data type and STRUCT functions pages for more details on working with structs.

For example:

```
SELECT st.* FROM (SELECT {'x': 1, 'y': 2, 'z': 3} AS st);
```

x	y	z
1	2	3

Subqueries

Subqueries are parenthesized query expressions that appear as part of a larger, outer query. Subqueries are usually based on `SELECT ... FROM`, but in DuckDB other query constructs such as `PIVOT` can also appear as a subquery.

Scalar Subquery

Scalar subqueries are subqueries that return a single value. They can be used anywhere where an expression can be used. If a scalar subquery returns more than a single value, an error is raised (unless `scalar_subquery_error_on_multiple_rows` is set to `false`, in which case a row is selected randomly).

Consider the following table:

Grades

grade	course
7	Math
9	Math
8	CS

```
CREATE TABLE grades (grade INTEGER, course VARCHAR);
INSERT INTO grades VALUES (7, 'Math'), (9, 'Math'), (8, 'CS');
```

We can run the following query to obtain the minimum grade:

```
SELECT min(grade) FROM grades;
```

min(grade)
7

By using a scalar subquery in the `WHERE` clause, we can figure out for which course this grade was obtained:

```
SELECT course FROM grades WHERE grade = (SELECT min(grade) FROM grades);
```

course
Math

Subquery Comparisons: ALL, ANY and SOME

In the section on scalar subqueries, a scalar expression was compared directly to a subquery using the equality comparison operator (=). Such direct comparisons only make sense with scalar subqueries.

Scalar expressions can still be compared to single-column subqueries returning multiple rows by specifying a quantifier. Available quantifiers are ALL, ANY and SOME. The quantifiers ANY and SOME are equivalent.

ALL

The ALL quantifier specifies that the comparison as a whole evaluates to `true` when the individual comparison results of *the expression at the left hand side of the comparison operator* with each of the values from *the subquery at the right hand side of the comparison operator* **all** evaluate to `true`:

```
SELECT 6 <= ALL (SELECT grade FROM grades) AS adequate;
```

returns:

adequate
true

because 6 is less than or equal to each of the subquery results 7, 8 and 9.

However, the following query

```
SELECT 8 >= ALL (SELECT grade FROM grades) AS excellent;
```

returns

excellent
false

because 8 is not greater than or equal to the subquery result 7. And thus, because not all comparisons evaluate `true`, `>= ALL` as a whole evaluates to `false`.

ANY

The ANY quantifier specifies that the comparison as a whole evaluates to `true` when at least one of the individual comparison results evaluates to `true`. For example:

```
SELECT 5 >= ANY (SELECT grade FROM grades) AS fail;
```

returns

fail
false

because no result of the subquery is less than or equal to 5.

The quantifier SOME maybe used instead of ANY: ANY and SOME are interchangeable.

EXISTS

The EXISTS operator tests for the existence of any row inside the subquery. It returns either true when the subquery returns one or more records, and false otherwise. The EXISTS operator is generally the most useful as a *correlated* subquery to express semijoin operations. However, it can be used as an uncorrelated subquery as well.

For example, we can use it to figure out if there are any grades present for a given course:

```
SELECT EXISTS (FROM grades WHERE course = 'Math') AS math_grades_present;
```

math_grades_present
true

```
SELECT EXISTS (FROM grades WHERE course = 'History') AS history_grades_present;
```

history_grades_present
false

> The subqueries in the examples above make use of the fact that you can omit the SELECT * in DuckDB thanks to the FROM-first syntax. The SELECT clause is required in subqueries by other SQL systems but cannot fulfil any purpose in EXISTS and NOT EXISTS subqueries.

NOT EXISTS

The NOT EXISTS operator tests for the absence of any row inside the subquery. It returns either true when the subquery returns an empty result, and false otherwise. The NOT EXISTS operator is generally the most useful as a *correlated* subquery to express antijoin operations. For example, to find Person nodes without an interest:

```
CREATE TABLE Person (id BIGINT, name VARCHAR);
CREATE TABLE interest (PersonId BIGINT, topic VARCHAR);

INSERT INTO Person VALUES (1, 'Jane'), (2, 'Joe');
INSERT INTO interest VALUES (2, 'Music');

SELECT *
FROM Person
WHERE NOT EXISTS (FROM interest WHERE interest.PersonId = Person.id);
```

id	name
1	Jane

> DuckDB automatically detects when a NOT EXISTS query expresses an antijoin operation. There is no need to manually rewrite such queries to use LEFT OUTER JOIN ... WHERE ... IS NULL.

IN Operator

The IN operator checks containment of the left expression inside the result defined by the subquery or the set of expressions on the right hand side (RHS). The IN operator returns true if the expression is present in the RHS, false if the expression is not in the RHS and the RHS has no NULL values, or NULL if the expression is not in the RHS and the RHS has NULL values.

We can use the `IN` operator in a similar manner as we used the `EXISTS` operator:

```
SELECT 'Math' IN (SELECT course FROM grades) AS math_grades_present;
```

math_grades_present
true

Correlated Subqueries

All the subqueries presented here so far have been **uncorrelated** subqueries, where the subqueries themselves are entirely self-contained and can be run without the parent query. There exists a second type of subqueries called **correlated** subqueries. For correlated subqueries, the subquery uses values from the parent subquery.

Conceptually, the subqueries are run once for every single row in the parent query. Perhaps a simple way of envisioning this is that the correlated subquery is a **function** that is applied to every row in the source data set.

For example, suppose that we want to find the minimum grade for every course. We could do that as follows:

```
SELECT *
FROM grades grades_parent
WHERE grade =
    (SELECT min(grade)
     FROM grades
     WHERE grades.course = grades_parent.course);
```

grade	course
7	Math
8	CS

The subquery uses a column from the parent query (`grades_parent.course`). Conceptually, we can see the subquery as a function where the correlated column is a parameter to that function:

```
SELECT min(grade)
FROM grades
WHERE course = ?;
```

Now when we execute this function for each of the rows, we can see that for `Math` this will return `7`, and for `CS` it will return `8`. We then compare it against the grade for that actual row. As a result, the row `(Math, 9)` will be filtered out, as `9 <> 7`.

Returning Each Row of the Subquery as a Struct

Using the name of a subquery in the `SELECT` clause (without referring to a specific column) turns each row of the subquery into a struct whose fields correspond to the columns of the subquery. For example:

```
SELECT t
FROM (SELECT unnest(generate_series(41, 43)) AS x, 'hello' AS y) t;
```

t
{'x': 41, 'y': hello}

t
{'x': 42, 'y': hello}
{'x': 43, 'y': hello}

TRY expression

The TRY expression ensures that errors caused by the input rows in the child (scalar) expression result in `NULL` for those rows, instead of causing the query to throw an error.

> The TRY expression was inspired by the TRY_CAST expression.

Examples

The following calls return errors when invoked without the TRY expression. When they are wrapped into as TRY expression, they return `NULL`:

Casting

Without TRY

```
SELECT 'abc'::INTEGER;
```

```
Conversion Error:
Could not convert string 'abc' to INT32
```

With TRY

```
SELECT TRY('abc'::INTEGER);
```

```
NULL
```

Logarithm on Zero

Without TRY

```
SELECT ln(0);
```

```
Out of Range Error:
cannot take logarithm of zero
```

With TRY

```
SELECT TRY(ln(0));
```

```
NULL
```

Casting Multiple Rows

Without TRY

```
WITH cte AS (FROM (VALUES ('123'), ('test'), ('235')) t(a))
SELECT a::INTEGER AS x FROM cte;
```

```
Conversion Error:
Could not convert string 'test' to INT32
```

With TRY

```
WITH cte AS (FROM (VALUES ('123'), ('test'), ('235')) t(a))
SELECT TRY(a::INTEGER) AS x FROM cte;
```

x
123
NULL
235

Limitations

TRY cannot be used in combination with a volatile function or with a scalar subquery. For example:

```
SELECT TRY(random())
```

```
Binder Error:
TRY can not be used in combination with a volatile function
```

Functions

Functions

Function Syntax

Function Chaining via the Dot Operator

DuckDB supports the dot syntax for function chaining. This allows the function call `fn(arg1, arg2, arg3, ...)` to be rewritten as `arg1.fn(arg2, arg3, ...)`. For example, take the following use of the `replace` function:

```
SELECT replace(goose_name, 'goose', 'duck') AS duck_name
FROM unnest(['African goose', 'Faroese goose', 'Hungarian goose', 'Pomeranian goose']) breed(goose_name);
```

This can be rewritten as follows:

```
SELECT goose_name.replace('goose', 'duck') AS duck_name
FROM unnest(['African goose', 'Faroese goose', 'Hungarian goose', 'Pomeranian goose']) breed(goose_name);
```

Using with Literals and Arrays

To apply function chaining to literals and following array access operations, you must surround the argument with parentheses, e.g.:

```
SELECT ('hello world').replace(' ', '_');
```

```
SELECT (2).sqrt();
```

```
SELECT (m[1]).map_entries()
FROM (VALUES ([MAP {'hello': 42}, MAP {'world': 42}])) t(m);
```

In the absence of these parentheses, DuckDB will return a `Parser Error` for the function call:

```
Parser Error:
syntax error at or near "("
```

Limitations

Function chaining via the dot operator is limited to *scalar* functions and is not supported for *table* functions. For example, the following call returns a `Parser Error`:

```
SELECT * FROM ('my_file.parquet').read_parquet(); -- does not work
```

Query Functions

The `duckdb_functions()` table function shows the list of functions currently built into the system.

```
SELECT DISTINCT ON(function_name)
    function_name,
    function_type,
    return_type,
    parameters,
    parameter_types,
    description
FROM duckdb_functions()
WHERE function_type = 'scalar'
  AND function_name LIKE 'b%'
ORDER BY function_name;
```

function_name	function_type	return_type	parameters	parameter_types	description
bar	scalar	VARCHAR	[x, min, max, width]	[DOUBLE, DOUBLE, DOUBLE, DOUBLE]	Draws a band whose width is proportional to (x - min) and equal to width characters when x = max. width defaults to 80
base64	scalar	VARCHAR	[blob]	[BLOB]	Convert a blob to a base64 encoded string
bin	scalar	VARCHAR	[value]	[VARCHAR]	Converts the value to binary representation
bit_count	scalar	TINYINT	[x]	[TINYINT]	Returns the number of bits that are set
bit_length	scalar	BIGINT	[col0]	[VARCHAR]	NULL
bit_position	scalar	INTEGER	[substring, bitstring]	[BIT, BIT]	Returns first starting index of the specified substring within bits, or zero if it is not present. The first (leftmost) bit is indexed 1
bitstring	scalar	BIT	[bitstring, length]	[VARCHAR, INTEGER]	Pads the bitstring until the specified length

> Currently, the description and parameter names of functions are not available in the `duckdb_functions()` function.

Aggregate Functions

Examples

Produce a single row containing the sum of the `amount` column:

```
SELECT sum(amount)
FROM sales;
```

Produce one row per unique region, containing the sum of `amount` for each group:

```
SELECT region, sum(amount)
FROM sales
GROUP BY region;
```

Return only the regions that have a sum of `amount` higher than 100:

```
SELECT region
FROM sales
GROUP BY region
HAVING sum(amount) > 100;
```

Return the number of unique values in the `region` column:

```
SELECT count(DISTINCT region)
FROM sales;
```

Return two values, the total sum of `amount` and the sum of `amount` minus columns where the region is `north` using the `FILTER` clause:

```
SELECT sum(amount), sum(amount) FILTER (region != 'north')
FROM sales;
```

Returns a list of all regions in order of the `amount` column:

```
SELECT list(region ORDER BY amount DESC)
FROM sales;
```

Returns the amount of the first sale using the `first()` aggregate function:

```
SELECT first(amount ORDER BY date ASC)
FROM sales;
```

Syntax

Aggregates are functions that *combine* multiple rows into a single value. Aggregates are different from scalar functions and window functions because they change the cardinality of the result. As such, aggregates can only be used in the `SELECT` and `HAVING` clauses of a SQL query.

`DISTINCT` Clause in Aggregate Functions

When the `DISTINCT` clause is provided, only distinct values are considered in the computation of the aggregate. This is typically used in combination with the `count` aggregate to get the number of distinct elements; but it can be used together with any aggregate function in the system. There are some aggregates that are insensitive to duplicate values (e.g., `min` and `max`) and for them this clause is parsed and ignored.

`ORDER BY` Clause in Aggregate Functions

An `ORDER BY` clause can be provided after the last argument of the function call. Note the lack of the comma separator before the clause.

```
SELECT <aggregate_function>(<arg>, <sep> ORDER BY <ordering_criteria>);
```

This clause ensures that the values being aggregated are sorted before applying the function. Most aggregate functions are order-insensitive, and for them this clause is parsed and discarded. However, there are some order-sensitive aggregates that can have non-deterministic results without ordering, e.g., `first`, `last`, `list` and `string_agg` / `group_concat` / `listagg`. These can be made deterministic by ordering the arguments.

For example:

```
CREATE TABLE tbl AS
    SELECT s FROM range(1, 4) r(s);

SELECT string_agg(s, ', ' ORDER BY s DESC) AS countdown
FROM tbl;
```

countdown
3, 2, 1

Handling NULL Values

All general aggregate functions ignore NULLs, except for `list` (`array_agg`), `first` (`arbitrary`) and `last`. To exclude NULLs from `list`, you can use a FILTER clause. To ignore NULLs from `first`, you can use the `any_value` aggregate.

All general aggregate functions except `count` return `NULL` on empty groups. In particular, `list` does *not* return an empty list, `sum` does *not* return zero, and `string_agg` does *not* return an empty string in this case.

General Aggregate Functions

The table below shows the available general aggregate functions.

Function	Description
`any_value(arg)`	Returns the first non-null value from `arg`. This function is affected by ordering.
`arbitrary(arg)`	Returns the first value (null or non-null) from `arg`. This function is affected by ordering.
`arg_max(arg, val)`	Finds the row with the maximum `val` and calculates the `arg` expression at that row. Rows where the value of the `arg` or `val` expression is `NULL` are ignored. This function is affected by ordering.
`arg_max(arg, val, n)`	The generalized case of `arg_max` for n values: returns a `LIST` containing the `arg` expressions for the top n rows ordered by `val` descending. This function is affected by ordering.
`arg_max_null(arg, val)`	Finds the row with the maximum `val` and calculates the `arg` expression at that row. Rows where the `val` expression evaluates to `NULL` are ignored. This function is affected by ordering.
`arg_min(arg, val)`	Finds the row with the minimum `val` and calculates the `arg` expression at that row. Rows where the value of the `arg` or `val` expression is `NULL` are ignored. This function is affected by ordering.
`arg_min(arg, val, n)`	Returns a `LIST` containing the `arg` expressions for the ”bottom” n rows ordered by `val` ascending. This function is affected by ordering.
`arg_min_null(arg, val)`	Finds the row with the minimum `val` and calculates the `arg` expression at that row. Rows where the `val` expression evaluates to `NULL` are ignored. This function is affected by ordering.
`array_agg(arg)`	Returns a `LIST` containing all the values of a column. This function is affected by ordering.
`avg(arg)`	Calculates the average of all non-null values in `arg`.
`bit_and(arg)`	Returns the bitwise AND of all bits in a given expression.
`bit_or(arg)`	Returns the bitwise OR of all bits in a given expression.
`bit_xor(arg)`	Returns the bitwise XOR of all bits in a given expression.
`bitstring_agg(arg)`	Returns a bitstring whose length corresponds to the range of the non-null (integer) values, with bits set at the location of each (distinct) value.
`bool_and(arg)`	Returns `true` if every input value is `true`, otherwise `false`.
`bool_or(arg)`	Returns `true` if any input value is `true`, otherwise `false`.
`count()`	Returns the number of rows in a group.
`count(arg)`	Returns the number of non-`NULL` values in `arg`.
`countif(arg)`	Returns the number of rows in a group where `arg` is `true`.
`favg(arg)`	Calculates the average using a more accurate floating point summation (Kahan Sum).
`first(arg)`	Returns the first value (null or non-null) from `arg`. This function is affected by ordering.
`fsum(arg)`	Calculates the sum using a more accurate floating point summation (Kahan Sum).
`geomean(arg)`	Calculates the geometric mean of all non-null values in `arg`.
`histogram(arg)`	Returns a `MAP` of key-value pairs representing buckets and counts.

Function	Description
`histogram(arg, boundaries)`	Returns a `MAP` of key-value pairs representing the provided upper `boundaries` and counts of elements in the corresponding bins (left-open and right-closed partitions) of the datatype. A boundary at the largest value of the datatype is automatically added when elements larger than all provided `boundaries` appear, see `is_histogram_other_bin`. Boundaries may be provided, e.g., via `equi_width_bins`.
`histogram_exact(arg, elements)`	Returns a `MAP` of key-value pairs representing the requested elements and their counts. A catch-all element specific to the data-type is automatically added to count other elements when they appear, see `is_histogram_other_bin`.
`histogram_values(source, boundaries)`	Returns the upper boundaries of the bins and their counts.
`kahan_sum(arg)`	Calculates the sum using a more accurate floating point summation (Kahan Sum).
`last(arg)`	Returns the last value of a column. This function is affected by ordering.
`list(arg)`	Returns a `LIST` containing all the values of a column. This function is affected by ordering.
`max(arg)`	Returns the maximum value present in `arg`. This function is unaffected by distinctness.
`max(arg, n)`	Returns a `LIST` containing the `arg` values for the "top" n rows ordered by `arg` descending.
`max_by(arg, val)`	Finds the row with the maximum `val`. Calculates the `arg` expression at that row. This function is affected by ordering.
`max_by(arg, val, n)`	Returns a `LIST` containing the `arg` expressions for the "top" n rows ordered by `val` descending.
`min(arg)`	Returns the minimum value present in `arg`. This function is unaffected by distinctness.
`min(arg, n)`	Returns a `LIST` containing the `arg` values for the "bottom" n rows ordered by `arg` ascending.
`min_by(arg, val)`	Finds the row with the minimum `val`. Calculates the `arg` expression at that row. This function is affected by ordering.
`min_by(arg, val, n)`	Returns a `LIST` containing the `arg` expressions for the "bottom" n rows ordered by `val` ascending.
`product(arg)`	Calculates the product of all non-null values in `arg`.
`string_agg(arg)`	Concatenates the column string values with a comma separator (`,`). This function is affected by ordering.
`string_agg(arg, sep)`	Concatenates the column string values with a separator. This function is affected by ordering.
`sum(arg)`	Calculates the sum of all non-null values in `arg` / counts `true` values when `arg` is boolean.
`sumkahan(arg)`	Calculates the sum using a more accurate floating point summation (Kahan Sum).
`weighted_avg(arg, weight)`	Calculates the weighted average all non-null values in `arg`, where each value is scaled by its corresponding `weight`. If `weight` is `NULL`, the corresponding `arg` value will be skipped.

`any_value(arg)`

Description	Returns the first non-`NULL` value from `arg`. This function is affected by ordering.
Example	`any_value(A)`
Alias(es)	-

`arbitrary(arg)`

Description	Returns the first value (NULL or non-NULL) from arg. This function is affected by ordering.
Example	arbitrary(A)
Alias(es)	first(A)

arg_max(arg, val)

Description	Finds the row with the maximum val and calculates the arg expression at that row. Rows where the value of the arg or val expression is NULL are ignored. This function is affected by ordering.
Example	arg_max(A, B)
Alias(es)	argmax(arg, val),max_by(arg, val)

arg_max(arg, val, n)

Description	The generalized case of arg_max for n values: returns a LIST containing the arg expressions for the top n rows ordered by val descending. This function is affected by ordering.
Example	arg_max(A, B, 2)
Alias(es)	argmax(arg, val, n),max_by(arg, val, n)

arg_max_null(arg, val)

Description	Finds the row with the maximum val and calculates the arg expression at that row. Rows where the val expression evaluates to NULL are ignored. This function is affected by ordering.
Example	arg_max_null(A, B)
Alias(es)	-

arg_min(arg, val)

Description	Finds the row with the minimum val and calculates the arg expression at that row. Rows where the value of the arg or val expression is NULL are ignored. This function is affected by ordering.
Example	arg_min(A, B)
Alias(es)	argmin(arg, val),min_by(arg, val)

arg_min(arg, val, n)

Description	The generalized case of arg_min for n values: returns a LIST containing the arg expressions for the top n rows ordered by val descending. This function is affected by ordering.
Example	arg_min(A, B, 2)
Alias(es)	argmin(arg, val, n),min_by(arg, val, n)

arg_min_null(arg, val)

Description	Finds the row with the minimum `val` and calculates the `arg` expression at that row. Rows where the `val` expression evaluates to `NULL` are ignored. This function is affected by ordering.
Example	`arg_min_null(A, B)`
Alias(es)	-

array_agg(arg)

Description	Returns a `LIST` containing all the values of a column. This function is affected by ordering.
Example	`array_agg(A)`
Alias(es)	`list`

avg(arg)

Description	Calculates the average of all non-null values in `arg`.
Example	`avg(A)`
Alias(es)	`mean`

bit_and(arg)

Description	Returns the bitwise AND of all bits in a given expression.
Example	`bit_and(A)`
Alias(es)	-

bit_or(arg)

Description	Returns the bitwise OR of all bits in a given expression.
Example	`bit_or(A)`
Alias(es)	-

bit_xor(arg)

Description	Returns the bitwise XOR of all bits in a given expression.
Example	`bit_xor(A)`
Alias(es)	-

`bitstring_agg(arg)`

Description	Returns a bitstring whose length corresponds to the range of the non-null (integer) values, with bits set at the location of each (distinct) value.
Example	`bitstring_agg(A)`
Alias(es)	-

`bool_and(arg)`

Description	Returns `true` if every input value is `true`, otherwise `false`.
Example	`bool_and(A)`
Alias(es)	-

`bool_or(arg)`

Description	Returns `true` if any input value is `true`, otherwise `false`.
Example	`bool_or(A)`
Alias(es)	-

`count()`

Description	Returns the number of rows in a group.
Example	`count()`
Alias(es)	`count(*)`

`count(arg)`

Description	Returns the number of non-`NULL` values in `arg`.
Example	`count(A)`
Alias(es)	-

`countif(arg)`

Description	Returns the number of rows in a group where `arg` is `true`.
Example	`countif(A)`
Alias(es)	-

`favg(arg)`

Description	Calculates the average using a more accurate floating point summation (Kahan Sum).
Example	`favg(A)`
Alias(es)	-

`first(arg)`

Description	Returns the first value (null or non-null) from `arg`. This function is affected by ordering.
Example	`first(A)`
Alias(es)	`arbitrary(A)`

`fsum(arg)`

Description	Calculates the sum using a more accurate floating point summation (Kahan Sum).
Example	`fsum(A)`
Alias(es)	`sumkahan`, `kahan_sum`

`geomean(arg)`

Description	Calculates the geometric mean of all non-null values in `arg`.
Example	`geomean(A)`
Alias(es)	`geometric_mean(A)`

`histogram(arg)`

Description	Returns a `MAP` of key-value pairs representing buckets and counts.
Example	`histogram(A)`
Alias(es)	-

`histogram(arg, boundaries)`

Description	Returns a `MAP` of key-value pairs representing the provided upper `boundaries` and counts of elements in the corresponding bins (left-open and right-closed partitions) of the datatype. A boundary at the largest value of the datatype is automatically added when elements larger than all provided `boundaries` appear, see `is_histogram_other_bin`. Boundaries may be provided, e.g., via `equi_width_bins`.
Example	`histogram(A, [0, 1, 10])`
Alias(es)	-

histogram_exact(arg, elements)

Description	Returns a MAP of key-value pairs representing the requested elements and their counts. A catch-all element specific to the data-type is automatically added to count other elements when they appear, see is_histogram_other_bin.
Example	histogram_exact(A, [0, 1, 10])
Alias(es)	-

histogram_values(source, col_name, technique, bin_count)

Description	Returns the upper boundaries of the bins and their counts.
Example	histogram_values(integers, i, bin_count := 2)
Alias(es)	-

last(arg)

Description	Returns the last value of a column. This function is affected by ordering.
Example	last(A)
Alias(es)	-

list(arg)

Description	Returns a LIST containing all the values of a column. This function is affected by ordering.
Example	list(A)
Alias(es)	array_agg

max(arg)

Description	Returns the maximum value present in arg. This function is unaffected by distinctness.
Example	max(A)
Alias(es)	-

max(arg, n)

Description	Returns a LIST containing the arg values for the ”top” n rows ordered by arg descending.
Example	max(A, 2)
Alias(es)	-

max_by(arg, val)

Description	Finds the row with the maximum `val`. Calculates the `arg` expression at that row. This function is affected by ordering.
Example	`max_by(A, B)`
Alias(es)	`argmax(arg, val)`,`arg_max(arg, val)`

max_by(arg, val, n)

Description	Returns a `LIST` containing the `arg` expressions for the "top" n rows ordered by `val` descending.
Example	`max_by(A, B, 2)`
Alias(es)	`argmax(arg, val, n)`,`arg_max(arg, val, n)`

min(arg)

Description	Returns the minimum value present in `arg`. This function is unaffected by distinctness.
Example	`min(A)`
Alias(es)	-

min(arg, n)

Description	Returns a `LIST` containing the `arg` values for the "bottom" n rows ordered by `arg` ascending.
Example	`min(A, 2)`
Alias(es)	-

min_by(arg, val)

Description	Finds the row with the minimum `val`. Calculates the `arg` expression at that row. This function is affected by ordering.
Example	`min_by(A, B)`
Alias(es)	`argMin(arg, val)`,`arg_min(arg, val)`

min_by(arg, val, n)

Description	Returns a `LIST` containing the `arg` expressions for the "bottom" n rows ordered by `val` ascending.
Example	`min_by(A, B, 2)`
Alias(es)	`argMin(arg, val, n)`,`arg_min(arg, val, n)`

`product(arg)`

Description	Calculates the product of all non-null values in `arg`.
Example	`product(A)`
Alias(es)	-

`string_agg(arg)`

Description	Concatenates the column string values with a comma separator (`,`). This function is affected by ordering.
Example	`string_agg(S, ',')`
Alias(es)	`group_concat(arg, sep)`,`listagg(arg, sep)`

`string_agg(arg, sep)`

Description	Concatenates the column string values with a separator. This function is affected by ordering.
Example	`string_agg(S, ',')`
Alias(es)	`group_concat(arg, sep)`,`listagg(arg, sep)`

`sum(arg)`

Description	Calculates the sum of all non-null values in `arg` / counts `true` values when `arg` is boolean.
Example	`sum(A)`
Alias(es)	-

`weighted_avg(arg, weight)`

Description	Calculates the weighted average of all non-null values in `arg`, where each value is scaled by its corresponding `weight`. If `weight` is `NULL`, the value will be skipped.
Example	`weighted_avg(A, W)`
Alias(es)	`wavg(arg, weight)`

Approximate Aggregates

The table below shows the available approximate aggregate functions.

Function	Description	Example
`approx_count_distinct(x)`	Calculates the approximate count of distinct elements using HyperLogLog.	`approx_count_distinct(A)`
`approx_quantile(x, pos)`	Calculates the approximate quantile using T-Digest.	`approx_quantile(A, 0.5)`
`approx_top_k(arg, k)`	Calculates a `LIST` of the `k` approximately most frequent values of `arg` using Filtered Space-Saving.	
`reservoir_quantile(x, quantile, sample_size = 8192)`	Calculates the approximate quantile using reservoir sampling, the sample size is optional and uses 8192 as a default size.	`reservoir_quantile(A, 0.5, 1024)`

Statistical Aggregates

The table below shows the available statistical aggregate functions. They all ignore `NULL` values (in the case of a single input column `x`), or pairs where either input is `NULL` (in the case of two input columns `y` and `x`).

Function	Description
`corr(y, x)`	The correlation coefficient.
`covar_pop(y, x)`	The population covariance, which does not include bias correction.
`covar_samp(y, x)`	The sample covariance, which includes Bessel's bias correction.
`entropy(x)`	The log-2 entropy.
`kurtosis_pop(x)`	The excess kurtosis (Fisher's definition) without bias correction.
`kurtosis(x)`	The excess kurtosis (Fisher's definition) with bias correction according to the sample size.
`mad(x)`	The median absolute deviation. Temporal types return a positive `INTERVAL`.
`median(x)`	The middle value of the set. For even value counts, quantitative values are averaged and ordinal values return the lower value.
`mode(x)`	The most frequent value. This function is affected by ordering.
`quantile_cont(x, pos)`	The interpolated `pos`-quantile of `x` for `0 <= pos <= 1`. Returns the `pos * (n_nonnull_values - 1)`th (zero-indexed, in the specified order) value of `x` or an interpolation between the adjacent values if the index is not an integer. Intuitively, arranges the values of `x` as equispaced *points* on a line, starting at 0 and ending at 1, and returns the (interpolated) value at `pos`. If `pos` is a `LIST` of `FLOAT`s, then the result is a `LIST` of the corresponding interpolated quantiles.
`quantile_disc(x, pos)`	The discrete `pos`-quantile of `x` for `0 <= pos <= 1`. Returns the `greatest(ceil(pos * n_nonnull_values) - 1, 0)`th (zero-indexed, in the specified order) value of `x`. Intuitively, assigns to each value of `x` an equisized *sub-interval* (left-open and right-closed except for the initial interval) of the interval `[0, 1]`, and picks the value of the sub-interval that contains `pos`. If `pos` is a `LIST` of `FLOAT`s, then the result is a `LIST` of the corresponding discrete quantiles.
`regr_avgx(y, x)`	The average of the independent variable for non-`NULL` pairs, where x is the independent variable and y is the dependent variable.
`regr_avgy(y, x)`	The average of the dependent variable for non-`NULL` pairs, where x is the independent variable and y is the dependent variable.
`regr_count(y, x)`	The number of non-`NULL` pairs.

Function	Description
`regr_intercept(y, x)`	The intercept of the univariate linear regression line, where x is the independent variable and y is the dependent variable.
`regr_r2(y, x)`	The squared Pearson correlation coefficient between y and x. Also: The coefficient of determination in a linear regression, where x is the independent variable and y is the dependent variable.
`regr_slope(y, x)`	The slope of the linear regression line, where x is the independent variable and y is the dependent variable.
`regr_sxx(y, x)`	The population variance, which includes Bessel's bias correction, of the independent variable for non-NULL pairs, where x is the independent variable and y is the dependent variable.
`regr_sxy(y, x)`	The population covariance, which includes Bessel's bias correction.
`regr_syy(y, x)`	The population variance, which includes Bessel's bias correction, of the dependent variable for non-NULL pairs , where x is the independent variable and y is the dependent variable.
`skewness(x)`	The skewness.
`sem(x)`	The standard error of the mean.
`stddev_pop(x)`	The population standard deviation.
`stddev_samp(x)`	The sample standard deviation.
`var_pop(x)`	The population variance, which does not include bias correction.
`var_samp(x)`	The sample variance, which includes Bessel's bias correction.

`corr(y, x)`

Description	The correlation coefficient.
Formula	`covar_pop(y, x) / (stddev_pop(x) * stddev_pop(y))`
Alias(es)	-

`covar_pop(y, x)`

Description	The population covariance, which does not include bias correction.
Formula	`(sum(x*y) - sum(x) * sum(y) / regr_count(y, x)) / regr_count(y, x)`, `covar_samp(y, x) * (1 - 1 / regr_count(y, x))`
Alias(es)	-

`covar_samp(y, x)`

Description	The sample covariance, which includes Bessel's bias correction.
Formula	`(sum(x*y) - sum(x) * sum(y) / regr_count(y, x)) / (regr_count(y, x) - 1)`,`covar_pop(y, x) / (1 - 1 / regr_count(y, x))`
Alias(es)	`regr_sxy(y, x)`

`entropy(x)`

Description	The log-2 entropy.
Formula	-
Alias(es)	-

`kurtosis_pop(x)`

Description	The excess kurtosis (Fisher's definition) without bias correction.
Formula	-
Alias(es)	-

`kurtosis(x)`

Description	The excess kurtosis (Fisher's definition) with bias correction according to the sample size.
Formula	-
Alias(es)	-

`mad(x)`

Description	The median absolute deviation. Temporal types return a positive `INTERVAL`.
Formula	`median(abs(x - median(x)))`
Alias(es)	-

`median(x)`

Description	The middle value of the set. For even value counts, quantitative values are averaged and ordinal values return the lower value.
Formula	`quantile_cont(x, 0.5)`
Alias(es)	-

`mode(x)`

Description	The most frequent value. This function is affected by ordering.
Formula	-
Alias(es)	-

`quantile_cont(x, pos)`

Description	The interpolated `pos`-quantile of `x` for `0 <= pos <= 1`. Returns the `pos * (n_nonnull_values - 1)`th (zero-indexed, in the specified order) value of `x` or an interpolation between the adjacent values if the index is not an integer. Intuitively, arranges the values of `x` as equispaced *points* on a line, starting at 0 and ending at 1, and returns the (interpolated) value at `pos`. If `pos` is a `LIST` of `FLOAT`s, then the result is a `LIST` of the corresponding interpolated quantiles.
Formula	-
Alias(es)	-

`quantile_disc(x, pos)`

Description	The discrete `pos`-quantile of `x` for `0 <= pos <= 1`. Returns the `greatest(ceil(pos * n_nonnull_values) - 1, 0)`th (zero-indexed, in the specified order) value of `x`. Intuitively, assigns to each value of `x` an equisized *sub-interval* (left-open and right-closed except for the initial interval) of the interval `[0, 1]`, and picks the value of the sub-interval that contains `pos`. If `pos` is a `LIST` of `FLOAT`s, then the result is a `LIST` of the corresponding discrete quantiles.
Formula	-
Alias(es)	`quantile`

`regr_avgx(y, x)`

Description	The average of the independent variable for non-`NULL` pairs, where x is the independent variable and y is the dependent variable.
Formula	-
Alias(es)	-

`regr_avgy(y, x)`

Description	The average of the dependent variable for non-`NULL` pairs, where x is the independent variable and y is the dependent variable.
Formula	-
Alias(es)	-

`regr_count(y, x)`

Description	The number of non-`NULL` pairs.
Formula	-
Alias(es)	-

`regr_intercept(y, x)`

Description	The intercept of the univariate linear regression line, where x is the independent variable and y is the dependent variable.
Formula	`regr_avgy(y, x) - regr_slope(y, x) * regr_avgx(y, x)`
Alias(es)	-

`regr_r2(y, x)`

Description	The squared Pearson correlation coefficient between y and x. Also: The coefficient of determination in a linear regression, where x is the independent variable and y is the dependent variable.
Formula	-
Alias(es)	-

`regr_slope(y, x)`

Description	Returns the slope of the linear regression line, where x is the independent variable and y is the dependent variable.
Formula	`regr_sxy(y, x) / regr_sxx(y, x)`
Alias(es)	-

`regr_sxx(y, x)`

Description	The population variance, which includes Bessel's bias correction, of the independent variable for non-`NULL` pairs, where x is the independent variable and y is the dependent variable.
Formula	-
Alias(es)	-

`regr_sxy(y, x)`

Description	The population covariance, which includes Bessel's bias correction.
Formula	-
Alias(es)	-

`regr_syy(y, x)`

Description	The population variance, which includes Bessel's bias correction, of the dependent variable for non-`NULL` pairs, where x is the independent variable and y is the dependent variable.
Formula	-

Alias(es)	-

sem(x)

Description	The standard error of the mean.
Formula	-
Alias(es)	-

skewness(x)

Description	The skewness.
Formula	-
Alias(es)	-

stddev_pop(x)

Description	The population standard deviation.
Formula	`sqrt(var_pop(x))`
Alias(es)	-

stddev_samp(x)

Description	The sample standard deviation.
Formula	`sqrt(var_samp(x))`
Alias(es)	`stddev(x)`

var_pop(x)

Description	The population variance, which does not include bias correction.
Formula	`(sum(x^2) - sum(x)^2 / count(x)) / count(x),var_samp(y, x) * (1 - 1 / count(x))`
Alias(es)	-

var_samp(x)

Description	The sample variance, which includes Bessel's bias correction.

Formula	`(sum(x^2) - sum(x)^2 / count(x)) / (count(x) - 1),var_pop(y, x) / (1 - 1 / count(x))`
Alias(es)	`variance(arg, val)`

Ordered Set Aggregate Functions

The table below shows the available "ordered set" aggregate functions. These functions are specified using the `WITHIN GROUP (ORDER BY sort_expression)` syntax, and they are converted to an equivalent aggregate function that takes the ordering expression as the first argument.

Function	Equivalent
mode() WITHIN GROUP (ORDER BY column [(ASC|DESC)])	mode(column ORDER BY column [(ASC|DESC)])
percentile_cont(fraction) WITHIN GROUP (ORDER BY column [(ASC|DESC)])	quantile_cont(column, fraction ORDER BY column [(ASC|DESC)])
percentile_cont(fractions) WITHIN GROUP (ORDER BY column [(ASC|DESC)])	quantile_cont(column, fractions ORDER BY column [(ASC|DESC)])
percentile_disc(fraction) WITHIN GROUP (ORDER BY column [(ASC|DESC)])	quantile_disc(column, fraction ORDER BY column [(ASC|DESC)])
percentile_disc(fractions) WITHIN GROUP (ORDER BY column [(ASC|DESC)])	quantile_disc(column, fractions ORDER BY column [(ASC|DESC)])

Miscellaneous Aggregate Functions

Function	Description	Alias
`grouping()`	For queries with GROUP BY and either ROLLUP or GROUPING SETS: Returns an integer identifying which of the argument expressions where used to group on to create the current supper-aggregate row.	`grouping_id()`

Array Functions

All LIST functions work with the ARRAY data type. Additionally, several ARRAY-native functions are also supported.

Array-Native Functions

Function	Description
`array_value(arg1, ...)`	Create an ARRAY containing the argument values.

Function	Description
`array_cross_product(array1, array2)`	Compute the cross product of two arrays of size 3. The array elements can not be `NULL`.
`array_cosine_similarity(array1, array2)`	Compute the cosine similarity between two arrays of the same size. The array elements can not be `NULL`. The arrays can have any size as long as the size is the same for both arguments.
`array_cosine_distance(array1, array2)`	Compute the cosine distance between two arrays of the same size. The array elements can not be `NULL`. The arrays can have any size as long as the size is the same for both arguments. This is equivalent to `1.0 - array_cosine_similarity`.
`array_distance(array1, array2)`	Compute the distance between two arrays of the same size. The array elements can not be `NULL`. The arrays can have any size as long as the size is the same for both arguments.
`array_inner_product(array1, array2)`	Compute the inner product between two arrays of the same size. The array elements can not be `NULL`. The arrays can have any size as long as the size is the same for both arguments.
`array_negative_inner_product(array1, array2)`	Compute the negative inner product between two arrays of the same size. The array elements can not be `NULL`. The arrays can have any size as long as the size is the same for both arguments. This is equivalent to `-array_inner_product`.
`array_dot_product(array1, array2)`	Alias for `array_inner_product(array1, array2)`.
`array_negative_dot_product(array1, array2)`	Alias for `array_negative_inner_product(array1, array2)`.

`array_value(arg1, ..)`

Description	Create an ARRAY containing the argument values.
Example	`array_value(1.0::FLOAT, 2.0::FLOAT, 3.0::FLOAT)`
Result	`[1.0, 2.0, 3.0]`

`array_cross_product(array1, array2)`

Description	Compute the cross product of two arrays of size 3. The array elements can not be `NULL`.
Example	`array_cross_product(array_value(1.0::FLOAT, 2.0::FLOAT, 3.0::FLOAT), array_value(2.0::FLOAT, 3.0::FLOAT, 4.0::FLOAT))`
Result	`[-1.0, 2.0, -1.0]`

`array_cosine_similarity(array1, array2)`

Description	Compute the cosine similarity between two arrays of the same size. The array elements can not be `NULL`. The arrays can have any size as long as the size is the same for both arguments.
Example	`array_cosine_similarity(array_value(1.0::FLOAT, 2.0::FLOAT, 3.0::FLOAT), array_value(2.0::FLOAT, 3.0::FLOAT, 4.0::FLOAT))`
Result	`0.9925833`

`array_cosine_distance(array1, array2)`

Description	Compute the cosine distance between two arrays of the same size. The array elements can not be `NULL`. The arrays can have any size as long as the size is the same for both arguments. This is equivalent to `1.0 - array_cosine_similarity`.
Example	`array_cosine_distance(array_value(1.0::FLOAT, 2.0::FLOAT, 3.0::FLOAT), array_value(2.0::FLOAT, 3.0::FLOAT, 4.0::FLOAT))`
Result	`0.007416606`

`array_distance(array1, array2)`

Description	Compute the distance between two arrays of the same size. The array elements can not be `NULL`. The arrays can have any size as long as the size is the same for both arguments.
Example	`array_distance(array_value(1.0::FLOAT, 2.0::FLOAT, 3.0::FLOAT), array_value(2.0::FLOAT, 3.0::FLOAT, 4.0::FLOAT))`
Result	`1.7320508`

`array_inner_product(array1, array2)`

Description	Compute the inner product between two arrays of the same size. The array elements can not be NULL. The arrays can have any size as long as the size is the same for both arguments.
Example	`array_inner_product(array_value(1.0::FLOAT, 2.0::FLOAT, 3.0::FLOAT), array_value(2.0::FLOAT, 3.0::FLOAT, 4.0::FLOAT))`
Result	`20.0`

`array_negative_inner_product(array1, array2)`

Description	Compute the negative inner product between two arrays of the same size. The array elements can not be NULL. The arrays can have any size as long as the size is the same for both arguments. This is equivalent to `-array_inner_product`
Example	`array_inner_product(array_value(1.0::FLOAT, 2.0::FLOAT, 3.0::FLOAT), array_value(2.0::FLOAT, 3.0::FLOAT, 4.0::FLOAT))`
Result	`-20.0`

`array_dot_product(array1, array2)`

Description	Alias for `array_inner_product(array1, array2)`.
Example	`array_dot_product(l1, l2)`
Result	`20.0`

`array_negative_dot_product(array1, array2)`

Description	Alias for `array_negative_inner_product(array1, array2)`.
Example	`array_negative_dot_product(l1, l2)`
Result	`-20.0`

Bitstring Functions

This section describes functions and operators for examining and manipulating `BITSTRING` values. Bitstrings must be of equal length when performing the bitwise operands AND, OR and XOR. When bit shifting, the original length of the string is preserved.

Bitstring Operators

The table below shows the available mathematical operators for `BIT` type.

Operator	Description	Example	Result
&	Bitwise AND	`'10101'::BITSTRING & '10001'::BITSTRING`	`10001`

Operator	Description	Example	Result
`|`	Bitwise OR	`'1011'::BITSTRING | '0001'::BITSTRING`	1011
`xor`	Bitwise XOR	`xor('101'::BITSTRING, '001'::BITSTRING)`	100
`~`	Bitwise NOT	`~('101'::BITSTRING)`	010
`<<`	Bitwise shift left	`'1001011'::BITSTRING << 3`	1011000
`>>`	Bitwise shift right	`'1001011'::BITSTRING >> 3`	0001001

Bitstring Functions

The table below shows the available scalar functions for `BIT` type.

Name	Description
`bit_count(bitstring)`	Returns the number of set bits in the bitstring.
`bit_length(bitstring)`	Returns the number of bits in the bitstring.
`bit_position(substring, bitstring)`	Returns first starting index of the specified substring within bits, or zero if it's not present. The first (leftmost) bit is indexed 1.
`bitstring(bitstring, length)`	Returns a bitstring of determined length.
`get_bit(bitstring, index)`	Extracts the nth bit from bitstring; the first (leftmost) bit is indexed 0.
`length(bitstring)`	Alias for `bit_length`.
`octet_length(bitstring)`	Returns the number of bytes in the bitstring.
`set_bit(bitstring, index, new_value)`	Sets the nth bit in bitstring to newvalue; the first (leftmost) bit is indexed 0. Returns a new bitstring.

`bit_count(bitstring)`

Description	Returns the number of set bits in the bitstring.
Example	`bit_count('1101011'::BITSTRING)`
Result	5

`bit_length(bitstring)`

Description	Returns the number of bits in the bitstring.
Example	`bit_length('1101011'::BITSTRING)`
Result	7

bit_position(substring, bitstring)

Description	Returns first starting index of the specified substring within bits, or zero if it's not present. The first (leftmost) bit is indexed 1
Example	`bit_position('010'::BITSTRING, '1110101'::BITSTRING)`
Result	4

bitstring(bitstring, length)

Description	Returns a bitstring of determined length.
Example	`bitstring('1010'::BITSTRING, 7)`
Result	0001010

get_bit(bitstring, index)

Description	Extracts the nth bit from bitstring; the first (leftmost) bit is indexed 0.
Example	`get_bit('0110010'::BITSTRING, 2)`
Result	1

length(bitstring)

Description	Alias for `bit_length`.
Example	`length('1101011'::BITSTRING)`
Result	7

octet_length(bitstring)

Description	Returns the number of bytes in the bitstring.
Example	`octet_length('1101011'::BITSTRING)`
Result	1

set_bit(bitstring, index, new_value)

Description	Sets the nth bit in bitstring to newvalue; the first (leftmost) bit is indexed 0. Returns a new bitstring.
Example	`set_bit('0110010'::BITSTRING, 2, 0)`
Result	0100010

Bitstring Aggregate Functions

These aggregate functions are available for `BIT` type.

Name	Description
`bit_and(arg)`	Returns the bitwise AND operation performed on all bitstrings in a given expression.
`bit_or(arg)`	Returns the bitwise OR operation performed on all bitstrings in a given expression.
`bit_xor(arg)`	Returns the bitwise XOR operation performed on all bitstrings in a given expression.
`bitstring_agg(arg)`	Returns a bitstring with bits set for each distinct position defined in `arg`.
`bitstring_agg(arg, min, max)`	Returns a bitstring with bits set for each distinct position defined in `arg`. All positions must be within the range [`min`, `max`] or an `Out of Range Error` will be thrown.

`bit_and(arg)`

Description	Returns the bitwise AND operation performed on all bitstrings in a given expression.
Example	`bit_and(A)`

`bit_or(arg)`

Description	Returns the bitwise OR operation performed on all bitstrings in a given expression.
Example	`bit_or(A)`

`bit_xor(arg)`

Description	Returns the bitwise XOR operation performed on all bitstrings in a given expression.
Example	`bit_xor(A)`

`bitstring_agg(arg)`

Description	The `bitstring_agg` function takes any integer type as input and returns a bitstring with bits set for each distinct value. The left-most bit represents the smallest value in the column and the right-most bit the maximum value. If possible, the min and max are retrieved from the column statistics. Otherwise, it is also possible to provide the min and max values.
Example	`bitstring_agg(A)`

> **Tip.** The combination of `bit_count` and `bitstring_agg` can be used as an alternative to `count(DISTINCT ...)`, with possible performance improvements in cases of low cardinality and dense values.

`bitstring_agg(arg, min, max)`

Description	Returns a bitstring with bits set for each distinct position defined in `arg`. All positions must be within the range [`min`, `max`] or an `Out of Range Error` will be thrown.
Example	`bitstring_agg(A, 1, 42)`

Blob Functions

This section describes functions and operators for examining and manipulating BLOB values.

Name	Description
`arg1 || arg2`	Concatenates two strings, lists, or blobs. Any NULL input results in NULL. See also `concat(arg1, arg2, ...)` and `list_concat(list1, list2)`.
`base64(blob)`	Converts a `blob` to a base64 encoded string.
`concat(value, ...)`	Concatenates multiple strings, lists, or blobs. NULL inputs are skipped. See also operator `||`.
`decode(blob)`	Converts `blob` to VARCHAR. Fails if `blob` is not valid UTF-8.
`encode(string)`	Converts the `string` to BLOB. Converts UTF-8 characters into literal encoding.
`from_base64(string)`	Converts a base64 encoded `string` to a character string (BLOB).
`from_binary(value)`	Converts a `value` from binary representation to a blob.
`from_hex(value)`	Converts a `value` from hexadecimal representation to a blob.
`hex(blob)`	Converts `blob` to VARCHAR using hexadecimal encoding.
`md5(blob)`	Returns the MD5 hash of the `blob` as a VARCHAR.
`md5_number(blob)`	Returns the MD5 hash of the `blob` as a HUGEINT.
`octet_length(blob)`	Number of bytes in `blob`.
`read_blob(source)`	Returns the content from `source` (a filename, a list of filenames, or a glob pattern) as a BLOB. See the `read_blob` guide for more details.
`repeat(blob, count)`	Repeats the `blob` `count` number of times.
`sha1(blob)`	Returns a VARCHAR with the SHA-1 hash of the `blob`.
`sha256(blob)`	Returns a VARCHAR with the SHA-256 hash of the `blob`.
`to_base64(blob)`	Converts a `blob` to a base64 encoded string.
`to_hex(blob)`	Converts `blob` to VARCHAR using hexadecimal encoding.
`unbin(value)`	Converts a `value` from binary representation to a blob.
`unhex(value)`	Converts a `value` from hexadecimal representation to a blob.

`arg1 || arg2`

Description	Concatenates two strings, lists, or blobs. Any NULL input results in NULL. See also `concat(arg1, arg2, ...)` and `list_concat(list1, list2)`.
Example 1	`'Duck' || 'DB'`
Result	DuckDB
Example 2	`[1, 2, 3] || [4, 5, 6]`
Result	`[1, 2, 3, 4, 5, 6]`

Example 3	`'\xAA'::BLOB || '\xBB'::BLOB`
Result	`\xAA\xBB`

`base64(blob)`

Description	Converts a `blob` to a base64 encoded string.
Example	`base64('A'::BLOB)`
Result	`QQ==`
Alias	`to_base64`

`concat(value, ...)`

Description	Concatenates multiple strings, lists, or blobs. `NULL` inputs are skipped. See also operator `||`.
Example	`concat('Hello', ' ', 'World')`
Result	`Hello World`

`decode(blob)`

Description	Converts `blob` to VARCHAR. Fails if `blob` is not valid UTF-8.
Example	`decode('\xC3\xBC'::BLOB)`
Result	ü

`encode(string)`

Description	Converts the `string` to `BLOB`. Converts UTF-8 characters into literal encoding.
Example	`encode('my_string_with_ü')`
Result	`my_string_with_\xC3\xBC`

`from_base64(string)`

Description	Converts a base64 encoded `string` to a character string (`BLOB`).
Example	`from_base64('QQ==')`
Result	A

`from_binary(value)`

Description	Converts a `value` from binary representation to a blob.
Example	`from_binary('0110')`
Result	`\x06`
Alias	`unbin`

`from_hex(value)`

Description	Converts a `value` from hexadecimal representation to a blob.
Example	`from_hex('2A')`
Result	`*`
Alias	`unhex`

`hex(blob)`

Description	Converts `blob` to VARCHAR using hexadecimal encoding.
Example	`hex('\xAA\xBB'::BLOB)`
Result	AABB
Alias	`to_hex`

`md5(blob)`

Description	Returns the MD5 hash of the `blob` as a VARCHAR.
Example	`md5('\xAA\xBB'::BLOB)`
Result	`58cea1f6b2b06520613e09af90dc1c47`

`md5_number(blob)`

Description	Returns the MD5 hash of the `blob` as a `HUGEINT`.
Example	`md5_number('\xAA\xBB'::BLOB)`
Result	`94525045605907259200829535064523132504`

`octet_length(blob)`

Description	Number of bytes in `blob`.
Example	`octet_length('\xAA\xBB'::BLOB)`
Result	2

`read_blob(source)`

Description	Returns the content from `source` (a filename, a list of filenames, or a glob pattern) as a BLOB. See the `read_blob` guide for more details.
Example	`read_blob('hello.bin')`
Result	`hello\x0A`

`repeat(blob, count)`

Description	Repeats the `blob` `count` number of times.
Example	`repeat('\xAA\xBB'::BLOB, 5)`
Result	`\xAA\xBB\xAA\xBB\xAA\xBB\xAA\xBB\xAA\xBB`

`sha1(blob)`

Description	Returns a VARCHAR with the SHA-1 hash of the `blob`.
Example	`sha1('\xAA\xBB'::BLOB)`
Result	`65b1e351a6cbfeb41c927222bc9ef53aad3396b0`

`sha256(blob)`

Description	Returns a VARCHAR with the SHA-256 hash of the `blob`.
Example	`sha256('\xAA\xBB'::BLOB)`
Result	`d798d1fac6bd4bb1c11f50312760351013379a0ab6f0a8c0af8a506b96b2525a`

`to_base64(blob)`

Description	Converts a `blob` to a base64 encoded string.
Example	`to_base64('A'::BLOB)`
Result	`QQ==`

Alias	base64

to_hex(blob)

Description	Converts blob to VARCHAR using hexadecimal encoding.
Example	to_hex('\xAA\xBB'::BLOB)
Result	AABB
Alias	hex

unbin(value)

Description	Converts a value from binary representation to a blob.
Example	unbin('0110')
Result	\x06
Alias	from_binary

unhex(value)

Description	Converts a value from hexadecimal representation to a blob.
Example	unhex('2A')
Result	*
Alias	from_hex

Date Format Functions

The strftime and strptime functions can be used to convert between DATE / TIMESTAMP values and strings. This is often required when parsing CSV files, displaying output to the user or transferring information between programs. Because there are many possible date representations, these functions accept a format string that describes how the date or timestamp should be structured.

strftime Examples

The strftime(timestamp, format) converts timestamps or dates to strings according to the specified pattern.

```
SELECT strftime(DATE '1992-03-02', '%d/%m/%Y');
```

```
02/03/1992
```

```
SELECT strftime(TIMESTAMP '1992-03-02 20:32:45', '%A, %-d %B %Y - %I:%M:%S %p');
```

```
Monday, 2 March 1992 - 08:32:45 PM
```

`strptime` Examples

The `strptime(text, format)` function converts strings to timestamps according to the specified pattern.

```
SELECT strptime('02/03/1992', '%d/%m/%Y');
```

```
1992-03-02 00:00:00
```

```
SELECT strptime('Monday, 2 March 1992 - 08:32:45 PM', '%A, %-d %B %Y - %I:%M:%S %p');
```

```
1992-03-02 20:32:45
```

The `strptime` function throws an error on failure:

```
SELECT strptime('02/50/1992', '%d/%m/%Y') AS x;
```

```
Invalid Input Error: Could not parse string "02/50/1992" according to format specifier "%d/%m/%Y"
02/50/1992
   ^
Error: Month out of range, expected a value between 1 and 12
```

To return `NULL` on failure, use the `try_strptime` function:

```
NULL
```

CSV Parsing

The date formats can also be specified during CSV parsing, either in the COPY statement or in the `read_csv` function. This can be done by either specifying a `DATEFORMAT` or a `TIMESTAMPFORMAT` (or both). `DATEFORMAT` will be used for converting dates, and `TIMESTAMPFORMAT` will be used for converting timestamps. Below are some examples for how to use this.

In a COPY statement:

```
COPY dates FROM 'test.csv' (DATEFORMAT '%d/%m/%Y', TIMESTAMPFORMAT '%A, %-d %B %Y - %I:%M:%S %p');
```

In a `read_csv` function:

```
SELECT *
FROM read_csv('test.csv', dateformat = '%m/%d/%Y', timestampformat = '%A, %-d %B %Y - %I:%M:%S %p');
```

Format Specifiers

Below is a full list of all available format specifiers.

Specifier	Description	Example
%a	Abbreviated weekday name.	Sun, Mon, ...
%A	Full weekday name.	Sunday, Monday, ...
%b	Abbreviated month name.	Jan, Feb, ..., Dec
%B	Full month name.	January, February, ...
%c	ISO date and time representation	1992-03-02 10:30:20
%d	Day of the month as a zero-padded decimal.	01, 02, ..., 31
%-d	Day of the month as a decimal number.	1, 2, ..., 30
%f	Microsecond as a decimal number, zero-padded on the left.	000000 - 999999
%g	Millisecond as a decimal number, zero-padded on the left.	000 - 999

Specifier	Description	Example
%G	ISO 8601 year with century representing the year that contains the greater part of the ISO week (see %V).	0001, 0002, ..., 2013, 2014, ..., 9998, 9999
%H	Hour (24-hour clock) as a zero-padded decimal number.	00, 01, ..., 23
%-H	Hour (24-hour clock) as a decimal number.	0, 1, ..., 23
%I	Hour (12-hour clock) as a zero-padded decimal number.	01, 02, ..., 12
%-I	Hour (12-hour clock) as a decimal number.	1, 2, ... 12
%j	Day of the year as a zero-padded decimal number.	001, 002, ..., 366
%-j	Day of the year as a decimal number.	1, 2, ..., 366
%m	Month as a zero-padded decimal number.	01, 02, ..., 12
%-m	Month as a decimal number.	1, 2, ..., 12
%M	Minute as a zero-padded decimal number.	00, 01, ..., 59
%-M	Minute as a decimal number.	0, 1, ..., 59
%n	Nanosecond as a decimal number, zero-padded on the left.	000000000 - 999999999
%p	Locale's AM or PM.	AM, PM
%S	Second as a zero-padded decimal number.	00, 01, ..., 59
%-S	Second as a decimal number.	0, 1, ..., 59
%u	ISO 8601 weekday as a decimal number where 1 is Monday.	1, 2, ..., 7
%U	Week number of the year. Week 01 starts on the first Sunday of the year, so there can be week 00. Note that this is not compliant with the week date standard in ISO-8601.	00, 01, ..., 53
%V	ISO 8601 week as a decimal number with Monday as the first day of the week. Week 01 is the week containing Jan 4. Note that %V is incompatible with year directive %Y. Use the ISO year %G instead.	01, ..., 53
%w	Weekday as a decimal number.	0, 1, ..., 6
%W	Week number of the year. Week 01 starts on the first Monday of the year, so there can be week 00. Note that this is not compliant with the week date standard in ISO-8601.	00, 01, ..., 53
%x	ISO date representation	1992-03-02
%X	ISO time representation	10:30:20
%y	Year without century as a zero-padded decimal number.	00, 01, ..., 99
%-y	Year without century as a decimal number.	0, 1, ..., 99
%Y	Year with century as a decimal number.	2013, 2019 etc.
%z	Time offset from UTC in the form ±HH:MM, ±HHMM, or ±HH.	-0700
%Z	Time zone name.	Europe/Amsterdam
%%	A literal % character.	%

Date Functions

This section describes functions and operators for examining and manipulating DATE values.

Date Operators

The table below shows the available mathematical operators for DATE types.

Operator	Description	Example	Result
+	addition of days (integers)	`DATE '1992-03-22' + 5`	`1992-03-27`
+	addition of an INTERVAL	`DATE '1992-03-22' + INTERVAL 5 DAY`	`1992-03-27 00:00:00`
+	addition of a variable INTERVAL	`SELECT DATE '1992-03-22' + INTERVAL (d.days) DAY FROM (VALUES (5), (11)) d(days)`	`1992-03-27 00:00:00` and `1992-04-02 00:00:00`
-	subtraction of DATEs	`DATE '1992-03-27' - DATE '1992-03-22'`	`5`
-	subtraction of an INTERVAL	`DATE '1992-03-27' - INTERVAL 5 DAY`	`1992-03-22 00:00:00`
-	subtraction of a variable INTERVAL	`SELECT DATE '1992-03-27' - INTERVAL (d.days) DAY FROM (VALUES (5), (11)) d(days)`	`1992-03-22 00:00:00` and `1992-03-16 00:00:00`

Adding to or subtracting from infinite values produces the same infinite value.

Date Functions

The table below shows the available functions for DATE types. Dates can also be manipulated with the timestamp functions through type promotion.

Name	Description
`current_date`	Current date (at start of current transaction) in the local time zone. Note that parentheses should be omitted from the function call.
`date_add(date, interval)`	Add the interval to the date and return a DATETIME value.
`date_diff(part, startdate, enddate)`	The number of partition boundaries between the dates.
`date_part(part, date)`	Get the subfield (equivalent to `extract`).
`date_sub(part, startdate, enddate)`	The number of complete partitions between the dates.
`date_trunc(part, date)`	Truncate to specified precision.
`datediff(part, startdate, enddate)`	The number of partition boundaries between the dates. Alias of `date_diff`.
`datepart(part, date)`	Get the subfield (equivalent to `extract`). Alias of `date_part`.
`datesub(part, startdate, enddate)`	The number of complete partitions between the dates. Alias of `date_sub`.
`datetrunc(part, date)`	Truncate to specified precision. Alias of `date_trunc`.
`dayname(date)`	The (English) name of the weekday.
`extract(part from date)`	Get subfield from a date.

Name	Description
`greatest(date, date)`	The later of two dates.
`isfinite(date)`	Returns true if the date is finite, false otherwise.
`isinf(date)`	Returns true if the date is infinite, false otherwise.
`julian(date)`	Extract the Julian Day number from a date.
`last_day(date)`	The last day of the corresponding month in the date.
`least(date, date)`	The earlier of two dates.
`make_date(year, month, day)`	The date for the given parts.
`monthname(date)`	The (English) name of the month.
`strftime(date, format)`	Converts a date to a string according to the format string.
`time_bucket(bucket_width, date[, offset])`	Truncate `date` to a grid of width `bucket_width`. The grid is anchored at `2000-01-01[ + offset]` when `bucket_width` is a number of months or coarser units, else `2000-01-03[ + offset]`. Note that `2000-01-03` is a Monday.
`time_bucket(bucket_width, date[, origin])`	Truncate `timestamptz` to a grid of width `bucket_width`. The grid is anchored at the `origin` timestamp, which defaults to `2000-01-01` when `bucket_width` is a number of months or coarser units, else `2000-01-03`. Note that `2000-01-03` is a Monday.
`today()`	Current date (start of current transaction) in UTC.

`current_date`

Description	Current date (at start of current transaction) in the local time zone. Note that parentheses should be omitted from the function call.
Example	`current_date`
Result	`2022-10-08`

`date_add(date, interval)`

Description	Add the interval to the date and return a `DATETIME` value.
Example	`date_add(DATE '1992-09-15', INTERVAL 2 MONTH)`
Result	`1992-11-15 00:00:00`

`date_diff(part, startdate, enddate)`

Description	The number of partition boundaries between the dates.
Example	`date_diff('month', DATE '1992-09-15', DATE '1992-11-14')`
Result	2

`date_part(part, date)`

Description	Get the subfield (equivalent to `extract`).
Example	`date_part('year', DATE '1992-09-20')`
Result	`1992`

`date_sub(part, startdate, enddate)`

Description	The number of complete partitions between the dates.
Example	`date_sub('month', DATE '1992-09-15', DATE '1992-11-14')`
Result	`1`

`date_trunc(part, date)`

Description	Truncate to specified precision.
Example	`date_trunc('month', DATE '1992-03-07')`
Result	`1992-03-01`

`datediff(part, startdate, enddate)`

Description	The number of partition boundaries between the dates.
Example	`datediff('month', DATE '1992-09-15', DATE '1992-11-14')`
Result	`2`
Alias	`date_diff`.

`datepart(part, date)`

Description	Get the subfield (equivalent to `extract`).
Example	`datepart('year', DATE '1992-09-20')`
Result	`1992`
Alias	`date_part`.

`datesub(part, startdate, enddate)`

Description	The number of complete partitions between the dates.
Example	`datesub('month', DATE '1992-09-15', DATE '1992-11-14')`
Result	`1`
Alias	`date_sub`.

`datetrunc(part, date)`

Description	Truncate to specified precision.
Example	`datetrunc('month', DATE '1992-03-07')`
Result	`1992-03-01`
Alias	`date_trunc`.

`dayname(date)`

Description	The (English) name of the weekday.
Example	`dayname(DATE '1992-09-20')`
Result	Sunday

`extract(part from date)`

Description	Get subfield from a date.
Example	`extract('year' FROM DATE '1992-09-20')`
Result	`1992`

`greatest(date, date)`

Description	The later of two dates.
Example	`greatest(DATE '1992-09-20', DATE '1992-03-07')`
Result	`1992-09-20`

`isfinite(date)`

Description	Returns `true` if the date is finite, false otherwise.
Example	`isfinite(DATE '1992-03-07')`
Result	`true`

`isinf(date)`

Description	Returns `true` if the date is infinite, false otherwise.
Example	`isinf(DATE '-infinity')`
Result	`true`

julian(date)

Description	Extract the Julian Day number from a date.
Example	`julian(DATE '1992-09-20')`
Result	`2448886.0`

last_day(date)

Description	The last day of the corresponding month in the date.
Example	`last_day(DATE '1992-09-20')`
Result	`1992-09-30`

least(date, date)

Description	The earlier of two dates.
Example	`least(DATE '1992-09-20', DATE '1992-03-07')`
Result	`1992-03-07`

make_date(year, month, day)

Description	The date for the given parts.
Example	`make_date(1992, 9, 20)`
Result	`1992-09-20`

monthname(date)

Description	The (English) name of the month.
Example	`monthname(DATE '1992-09-20')`
Result	`September`

strftime(date, format)

Description	Converts a date to a string according to the format string.
Example	`strftime(DATE '1992-01-01', '%a, %-d %B %Y')`
Result	`Wed, 1 January 1992`

`time_bucket(bucket_width, date[, offset])`

Description	Truncate `date` to a grid of width `bucket_width`. The grid is anchored at `2000-01-01[ + offset]` when `bucket_width` is a number of months or coarser units, else `2000-01-03[ + offset]`. Note that `2000-01-03` is a Monday.
Example	`time_bucket(INTERVAL '2 months', DATE '1992-04-20', INTERVAL '1 month')`
Result	`1992-04-01`

`time_bucket(bucket_width, date[, origin])`

Description	Truncate `timestamptz` to a grid of width `bucket_width`. The grid is anchored at the `origin` timestamp, which defaults to `2000-01-01` when `bucket_width` is a number of months or coarser units, else `2000-01-03`. Note that `2000-01-03` is a Monday.
Example	`time_bucket(INTERVAL '2 weeks', DATE '1992-04-20', DATE '1992-04-01')`
Result	`1992-04-15`

`today()`

Description	Current date (start of current transaction) in UTC.
Example	`today()`
Result	`2022-10-08`

Date Part Extraction Functions

There are also dedicated extraction functions to get the subfields. A few examples include extracting the day from a date, or the day of the week from a date.

Functions applied to infinite dates will either return the same infinite dates (e.g., `greatest`) or `NULL` (e.g., `date_part`) depending on what "makes sense". In general, if the function needs to examine the parts of the infinite date, the result will be `NULL`.

Date Part Functions

The `date_part` and `date_diff` and `date_trunc` functions can be used to manipulate the fields of temporal types such as `DATE` and `TIMESTAMP`. The fields are specified as strings that contain the part name of the field.

Below is a full list of all available date part specifiers. The examples are the corresponding parts of the timestamp `2021-08-03 11:59:44.123456`.

Part Specifiers Usable as Date Part Specifiers and in Intervals

Specifier	Description	Synonyms	Example
`century`	Gregorian century	`cent`, `centuries`, `c`	21
`day`	Gregorian day	`days`, `d`, `dayofmonth`	3
`decade`	Gregorian decade	`dec`, `decades`, `decs`	202
`hour`	Hours	`hr`, `hours`, `hrs`, `h`	11
`microseconds`	Sub-minute microseconds	`microsecond`, `us`, `usec`, `usecs`, `usecond`, `useconds`	44123456
`millennium`	Gregorian millennium	`mil`, `millenniums`, `millenia`, `mils`, `millenium`	3
`milliseconds`	Sub-minute milliseconds	`millisecond`, `ms`, `msec`, `msecs`, `msecond`, `mseconds`	44123
`minute`	Minutes	`min`, `minutes`, `mins`, `m`	59
`month`	Gregorian month	`mon`, `months`, `mons`	8
`quarter`	Quarter of the year (1-4)	`quarters`	3
`second`	Seconds	`sec`, `seconds`, `secs`, `s`	44
`year`	Gregorian year	`yr`, `y`, `years`, `yrs`	2021

Part Specifiers Only Usable as Date Part Specifiers

Specifier	Description	Synonyms	Example
`dayofweek`	Day of the week (Sunday = 0, Saturday = 6)	`weekday`, `dow`	2
`dayofyear`	Day of the year (1-365/366)	`doy`	215
`epoch`	Seconds since 1970-01-01		1627991984
`era`	Gregorian era (CE/AD, BCE/BC)		1
`isodow`	ISO day of the week (Monday = 1, Sunday = 7)		2
`isoyear`	ISO Year number (Starts on Monday of week containing Jan 4th)		2021
`timezone_hour`	Time zone offset hour portion		0
`timezone_minute`	Time zone offset minute portion		0
`timezone`	Time zone offset in seconds		0
`week`	Week number	`weeks`, `w`	31
`yearweek`	ISO year and week number in `YYYYWW` format		202131

Note that the time zone parts are all zero unless a time zone extension such as ICU has been installed to support `TIMESTAMP WITH TIME ZONE`.

Part Functions

There are dedicated extraction functions to get certain subfields:

Name	Description
`century(date)`	Century.
`day(date)`	Day.
`dayofmonth(date)`	Day (synonym).
`dayofweek(date)`	Numeric weekday (Sunday = 0, Saturday = 6).
`dayofyear(date)`	Day of the year (starts from 1, i.e., January 1 = 1).
`decade(date)`	Decade (year / 10).
`epoch(date)`	Seconds since 1970-01-01.
`era(date)`	Calendar era.
`hour(date)`	Hours.
`isodow(date)`	Numeric ISO weekday (Monday = 1, Sunday = 7).
`isoyear(date)`	ISO Year number (Starts on Monday of week containing Jan 4th).
`microsecond(date)`	Sub-minute microseconds.
`millennium(date)`	Millennium.
`millisecond(date)`	Sub-minute milliseconds.
`minute(date)`	Minutes.
`month(date)`	Month.
`quarter(date)`	Quarter.
`second(date)`	Seconds.
`timezone_hour(date)`	Time zone offset hour portion.
`timezone_minute(date)`	Time zone offset minutes portion.
`timezone(date)`	Time zone offset in minutes.
`week(date)`	ISO Week.
`weekday(date)`	Numeric weekday synonym (Sunday = 0, Saturday = 6).
`weekofyear(date)`	ISO Week (synonym).
`year(date)`	Year.
`yearweek(date)`	`BIGINT` of combined ISO Year number and 2-digit version of ISO Week number.

`century(date)`

Description	Century.
Example	`century(DATE '1992-02-15')`
Result	`20`

`day(date)`

Description	Day.
Example	`day(DATE '1992-02-15')`
Result	`15`

`dayofmonth(date)`

Description	Day (synonym).
Example	`dayofmonth(DATE '1992-02-15')`
Result	`15`

`dayofweek(date)`

Description	Numeric weekday (Sunday = 0, Saturday = 6).
Example	`dayofweek(DATE '1992-02-15')`
Result	`6`

`dayofyear(date)`

Description	Day of the year (starts from 1, i.e., January 1 = 1).
Example	`dayofyear(DATE '1992-02-15')`
Result	`46`

`decade(date)`

Description	Decade (year / 10).
Example	`decade(DATE '1992-02-15')`
Result	`199`

`epoch(date)`

Description	Seconds since 1970-01-01.
Example	`epoch(DATE '1992-02-15')`
Result	`698112000`

era(date)

Description	Calendar era.
Example	`era(DATE '0044-03-15 (BC)')`
Result	`0`

hour(date)

Description	Hours.
Example	`hour(timestamp '2021-08-03 11:59:44.123456')`
Result	`11`

isodow(date)

Description	Numeric ISO weekday (Monday = 1, Sunday = 7).
Example	`isodow(DATE '1992-02-15')`
Result	`6`

isoyear(date)

Description	ISO Year number (Starts on Monday of week containing Jan 4th).
Example	`isoyear(DATE '2022-01-01')`
Result	`2021`

microsecond(date)

Description	Sub-minute microseconds.
Example	`microsecond(timestamp '2021-08-03 11:59:44.123456')`
Result	`44123456`

millennium(date)

Description	Millennium.
Example	`millennium(DATE '1992-02-15')`
Result	`2`

millisecond(date)

Description	Sub-minute milliseconds.
Example	`millisecond(timestamp '2021-08-03 11:59:44.123456')`
Result	`44123`

minute(date)

Description	Minutes.
Example	`minute(timestamp '2021-08-03 11:59:44.123456')`
Result	`59`

month(date)

Description	Month.
Example	`month(DATE '1992-02-15')`
Result	`2`

quarter(date)

Description	Quarter.
Example	`quarter(DATE '1992-02-15')`
Result	`1`

second(date)

Description	Seconds.
Example	`second(timestamp '2021-08-03 11:59:44.123456')`
Result	`44`

timezone_hour(date)

Description	Time zone offset hour portion.
Example	`timezone_hour(DATE '1992-02-15')`
Result	`0`

timezone_minute(date)

Description	Time zone offset minutes portion.
Example	`timezone_minute(DATE '1992-02-15')`
Result	`0`

timezone(date)

Description	Time zone offset in minutes.
Example	`timezone(DATE '1992-02-15')`
Result	`0`

week(date)

Description	ISO Week.
Example	`week(DATE '1992-02-15')`
Result	`7`

weekday(date)

Description	Numeric weekday synonym (Sunday = 0, Saturday = 6).
Example	`weekday(DATE '1992-02-15')`
Result	`6`

weekofyear(date)

Description	ISO Week (synonym).
Example	`weekofyear(DATE '1992-02-15')`
Result	`7`

year(date)

Description	Year.
Example	`year(DATE '1992-02-15')`
Result	`1992`

`yearweek(date)`

Description	BIGINT of combined ISO Year number and 2-digit version of ISO Week number.
Example	`yearweek(DATE '1992-02-15')`
Result	`199207`

Enum Functions

This section describes functions and operators for examining and manipulating ENUM values. The examples assume an enum type created as:

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy', 'anxious');
```

These functions can take `NULL` or a specific value of the type as argument(s). With the exception of `enum_range_boundary`, the result depends only on the type of the argument and not on its value.

Name	Description
`enum_code(enum_value)`	Returns the numeric value backing the given enum value.
`enum_first(enum)`	Returns the first value of the input enum type.
`enum_last(enum)`	Returns the last value of the input enum type.
`enum_range(enum)`	Returns all values of the input enum type as an array.
`enum_range_boundary(enum, enum)`	Returns the range between the two given enum values as an array.

`enum_code(enum_value)`

Description	Returns the numeric value backing the given enum value.
Example	`enum_code('happy'::mood)`
Result	2

`enum_first(enum)`

Description	Returns the first value of the input enum type.
Example	`enum_first(NULL::mood)`
Result	`sad`

`enum_last(enum)`

Description	Returns the last value of the input enum type.
Example	`enum_last(NULL::mood)`
Result	`anxious`

`enum_range(enum)`

Description	Returns all values of the input enum type as an array.
Example	`enum_range(NULL::mood)`
Result	`[sad, ok, happy, anxious]`

`enum_range_boundary(enum, enum)`

Description	Returns the range between the two given enum values as an array. The values must be of the same enum type. When the first parameter is `NULL`, the result starts with the first value of the enum type. When the second parameter is `NULL`, the result ends with the last value of the enum type.
Example	`enum_range_boundary(NULL, 'happy'::mood)`
Result	`[sad, ok, happy]`

Interval Functions

This section describes functions and operators for examining and manipulating `INTERVAL` values.

Interval Operators

The table below shows the available mathematical operators for `INTERVAL` types.

Operator	Description	Example	Result
+	Addition of an `INTERVAL`	`INTERVAL 1 HOUR + INTERVAL 5 HOUR`	`INTERVAL 6 HOUR`
+	Addition to a `DATE`	`DATE '1992-03-22' + INTERVAL 5 DAY`	`1992-03-27`
+	Addition to a `TIMESTAMP`	`TIMESTAMP '1992-03-22 01:02:03' + INTERVAL 5 DAY`	`1992-03-27 01:02:03`
+	Addition to a `TIME`	`TIME '01:02:03' + INTERVAL 5 HOUR`	`06:02:03`
-	Subtraction of an `INTERVAL`	`INTERVAL 5 HOUR - INTERVAL 1 HOUR`	`INTERVAL 4 HOUR`
-	Subtraction from a `DATE`	`DATE '1992-03-27' - INTERVAL 5 DAY`	`1992-03-22`
-	Subtraction from a `TIMESTAMP`	`TIMESTAMP '1992-03-27 01:02:03' - INTERVAL 5 DAY`	`1992-03-22 01:02:03`
-	Subtraction from a `TIME`	`TIME '06:02:03' - INTERVAL 5 HOUR`	`01:02:03`

Interval Functions

The table below shows the available scalar functions for `INTERVAL` types.

Name	Description
`date_part(part, interval)`	Extract datepart component (equivalent to `extract`). See `INTERVAL` for the sometimes surprising rules governing this extraction.
`datepart(part, interval)`	Alias of `date_part`.
`extract(part FROM interval)`	Alias of `date_part`.
`epoch(interval)`	Get total number of seconds, as double precision floating point number, in interval.
`to_centuries(integer)`	Construct a century interval.
`to_days(integer)`	Construct a day interval.
`to_decades(integer)`	Construct a decade interval.
`to_hours(integer)`	Construct an hour interval.
`to_microseconds(integer)`	Construct a microsecond interval.
`to_millennia(integer)`	Construct a millennium interval.
`to_milliseconds(integer)`	Construct a millisecond interval.
`to_minutes(integer)`	Construct a minute interval.
`to_months(integer)`	Construct a month interval.
`to_quarters(integer)`	Construct an interval of `integer` quarters.
`to_seconds(integer)`	Construct a second interval.
`to_weeks(integer)`	Construct a week interval.
`to_years(integer)`	Construct a year interval.

> Only the documented date part components are defined for intervals.

`date_part(part, interval)`

Description	Extract datepart component (equivalent to `extract`). See `INTERVAL` for the sometimes surprising rules governing this extraction.
Example	`date_part('year', INTERVAL '14 months')`
Result	`1`

`datepart(part, interval)`

Description	Alias of `date_part`.
Example	`datepart('year', INTERVAL '14 months')`
Result	`1`

`extract(part FROM interval)`

Description	Alias of `date_part`.

Example	`extract('month' FROM INTERVAL '14 months')`
Result	2

epoch(interval)

Description	Get total number of seconds, as double precision floating point number, in interval.
Example	`epoch(INTERVAL 5 HOUR)`
Result	`18000.0`

to_centuries(integer)

Description	Construct a century interval.
Example	`to_centuries(5)`
Result	`INTERVAL 500 YEAR`

to_days(integer)

Description	Construct a day interval.
Example	`to_days(5)`
Result	`INTERVAL 5 DAY`

to_decades(integer)

Description	Construct a decade interval.
Example	`to_decades(5)`
Result	`INTERVAL 50 YEAR`

to_hours(integer)

Description	Construct an hour interval.
Example	`to_hours(5)`
Result	`INTERVAL 5 HOUR`

to_microseconds(integer)

Description	Construct a microsecond interval.
Example	`to_microseconds(5)`

Result	`INTERVAL 5 MICROSECOND`

`to_millennia(integer)`

Description	Construct a millennium interval.
Example	`to_millennia(5)`
Result	`INTERVAL 5000 YEAR`

`to_milliseconds(integer)`

Description	Construct a millisecond interval.
Example	`to_milliseconds(5)`
Result	`INTERVAL 5 MILLISECOND`

`to_minutes(integer)`

Description	Construct a minute interval.
Example	`to_minutes(5)`
Result	`INTERVAL 5 MINUTE`

`to_months(integer)`

Description	Construct a month interval.
Example	`to_months(5)`
Result	`INTERVAL 5 MONTH`

`to_quarters(integer)`

Description	Construct an interval of `integer` quarters.
Example	`to_quarters(5)`
Result	`INTERVAL 1 YEAR 3 MONTHS`

`to_seconds(integer)`

Description	Construct a second interval.
Example	`to_seconds(5)`
Result	`INTERVAL 5 SECOND`

`to_weeks(integer)`

Description	Construct a week interval.
Example	`to_weeks(5)`
Result	`INTERVAL 35 DAY`

`to_years(integer)`

Description	Construct a year interval.
Example	`to_years(5)`
Result	`INTERVAL 5 YEAR`

Lambda Functions

> **Deprecated.** DuckDB 1.3.0 deprecated the old lambda single arrow syntax (`x -> x + 1`) in favor of the Python-style syntax (`lambda x : x + 1`).
>
> DuckDB 1.3.0 also introduces a new setting to configure the lambda syntax.
>
> ```
> SET lambda_syntax = 'DEFAULT';
> SET lambda_syntax = 'ENABLE_SINGLE_ARROW';
> SET lambda_syntax = 'DISABLE_SINGLE_ARROW';
> ```
>
> Currently, `DEFAULT` enables both syntax styles, i.e., the old single arrow syntax and the Python-style syntax.
>
> DuckDB 1.4.0 will be the last release supporting the single arrow syntax without explicitly enabling it.
>
> DuckDB 1.5.0 disables the single arrow syntax on default.
>
> DuckDB 1.6.0 removes the `lambda_syntax` flag and fully deprecates the single arrow syntax, so the old behavior will no longer be possible.

Lambda functions enable the use of more complex and flexible expressions in queries. DuckDB supports several scalar functions that operate on `LIST`s and accept lambda functions as parameters in the form `lambda <parameter1>, <parameter2>, ... : <expression>`{:.language-sql .highlight}. If the lambda function has only one parameter, then the parentheses can be omitted. The parameters can have any names. For example, the following are all valid lambda functions:

- `lambda param : param > 1`{:.language-sql .highlight}
- `lambda s : contains(concat(s, 'DB'), 'duck')`{:.language-sql .highlight}
- `lambda acc, x : acc + x`{:.language-sql .highlight}

Scalar Functions That Accept Lambda Functions

Name	Description
`list_transform(list, lambda(x))`	Returns a list that is the result of applying the lambda function to each element of the input list. The return type is defined by the return type of the lambda function. See `list_transform` examples.

Name	Description
`list_filter(list, lambda(x))`	Constructs a list from those elements of the input list for which the lambda function returns `true`. DuckDB must be able to cast the lambda function's return type to BOOL. The return type of `list_filter` is the same as the input list's. See `list_filter` examples.
`list_reduce(list, lambda(x, y)[, initial_value]`	Reduces all elements of the input list into a single scalar value by executing the lambda function on a running result and the next list element. The lambda function has an optional `initial_value` argument. See `list_reduce` examples or details.

`list_transform(list, lambda(x))`

Description	Returns a list that is the result of applying the lambda function to each element of the input list. The return type is defined by the return type of the lambda function. See `list_transform` examples.
Example	`list_transform([4, 5, 6], lambda x : x + 1)`{:.language-sql .highlight}
Result	`[5, 6, 7]`
Aliases	`array_transform`, `apply`, `list_apply`, `array_apply`

`list_filter(list, lambda(x))`

Description	Constructs a list from those elements of the input list for which the lambda function returns `true`. DuckDB must be able to cast the lambda function's return type to BOOL. The return type of `list_filter` is the same as the input list's. See `list_filter` examples.
Example	`list_filter([4, 5, 6], lambda x : x > 4)`{:.language-sql .highlight}
Result	`[5, 6]`
Aliases	`array_filter`, `filter`

`list_reduce(list, lambda(x, y)[, initial_value]`

Description	Reduces all elements of the input list into a single scalar value by executing the lambda function on a running result and the next list element. The lambda function has an optional `initial_value` argument. See `list_reduce` examples or details.
Example	`list_reduce([1, 2, 3], lambda x, y : x + y, 100)`{:.language-sql .highlight}
Result	`106`
Aliases	`array_reduce`, `reduce`

Nesting Lambda Functions

All scalar functions can be arbitrarily nested. For example, nested lambda functions to get all squares of even list elements:

```sql
SELECT list_transform(
        list_filter([0, 1, 2, 3, 4, 5], lambda x: x % 2 = 0),
        lambda y: y * y
    );
```

```
[0, 4, 16]
```

Nested lambda function to add each element of the first list to the sum of the second list:

```
SELECT list_transform(
        [1, 2, 3],
        lambda x :
            list_reduce([4, 5, 6], lambda a, b: a + b) + x
    );
```

```
[16, 17, 18]
```

Scoping

Lambda functions confirm to scoping rules in the following order:

- inner lambda parameters
- outer lambda parameters
- column names
- macro parameters

```
CREATE TABLE tbl (x INTEGER);
INSERT INTO tbl VALUES (10);
SELECT list_apply(
            [1, 2],
            lambda x: list_apply([4], lambda x: x + tbl.x)[1] + x
    )
FROM tbl;
```

```
[15, 16]
```

Indexes as Parameters

All lambda functions accept an optional extra parameter that represents the index of the current element. This is always the last parameter of the lambda function (e.g., `i` in `(x, i)`), and is 1-based (i.e., the first element has index 1).

Get all elements that are larger than their index:

```
SELECT list_filter([1, 3, 1, 5], lambda x, i: x > i);
```

```
[3, 5]
```

Examples

`list_transform` Examples

Incrementing each list element by one:

```
SELECT list_transform([1, 2, NULL, 3], lambda x: x + 1);
```

```
[2, 3, NULL, 4]
```

Transforming strings:

```
SELECT list_transform(['Duck', 'Goose', 'Sparrow'], lambda s: concat(s, 'DB'));
```

```
[DuckDB, GooseDB, SparrowDB]
```

Combining lambda functions with other functions:

```
SELECT list_transform([5, NULL, 6], lambda x: coalesce(x, 0) + 1);
```

```
[6, 1, 7]
```

`list_filter` Examples

Filter out negative values:

```
SELECT list_filter([5, -6, NULL, 7], lambda x: x > 0);
```

```
[5, 7]
```

Divisible by 2 and 5:

```
SELECT list_filter(
        list_filter([2, 4, 3, 1, 20, 10, 3, 30], lambda x: x % 2 = 0),
        lambda y: y % 5 = 0
    );
```

```
[20, 10, 30]
```

In combination with `range(...)` to construct lists:

```
SELECT list_filter([1, 2, 3, 4], lambda x: x > #1) FROM range(4);
```

```
[1, 2, 3, 4]
[2, 3, 4]
[3, 4]
[4]
```

`list_reduce` Examples

Sum of all list elements:

```
SELECT list_reduce([1, 2, 3, 4], lambda acc, x: acc + x);
```

```
10
```

Only add up list elements if they are greater than 2:

```
SELECT list_reduce(
        list_filter([1, 2, 3, 4], lambda x: x > 2),
        lambda acc, x: acc + x
    );
```

```
7
```

Concat all list elements:

```
SELECT list_reduce(['DuckDB', 'is', 'awesome'], lambda acc, x: concat(acc, ' ', x));
```

```
DuckDB is awesome
```

Concatenate elements with the index without an initial value:

```
SELECT list_reduce(
        ['a', 'b', 'c', 'd'],
        lambda x, y, i: x || ' - ' || CAST(i AS VARCHAR) || ' - ' || y
    );
```

```
a - 2 - b - 3 - c - 4 - d
```

Concatenate elements with the index with an initial value:

```
SELECT list_reduce(
        ['a', 'b', 'c', 'd'],
        lambda x, y, i: x || ' - ' || CAST(i AS VARCHAR) || ' - ' || y, 'INITIAL'
    );
```

```
INITIAL - 1 - a - 2 - b - 3 - c - 4 - d
```

Limitations

Subqueries in lambda expressions are currently not supported. For example:

```
SELECT list_apply([1, 2, 3], lambda x: (SELECT 42) + x);
```

```
Binder Error:
subqueries in lambda expressions are not supported
```

List Functions

Name	Description
`list[index]`	Bracket notation serves as an alias for `list_extract`.
`list[begin:end]`	Bracket notation with colon is an alias for `list_slice`.
`list[begin:end:step]`	`list_slice` in bracket notation with an added `step` feature.
`array_pop_back(list)`	Returns the list without the last element.
`array_pop_front(list)`	Returns the list without the first element.
`flatten(list_of_lists)`	Concatenate a list of lists into a single list. This only flattens one level of the list (see examples).
`len(list)`	Return the length of the list.
`list_aggregate(list, name)`	Executes the aggregate function `name` on the elements of `list`. See the List Aggregates section for more details.
`list_any_value(list)`	Returns the first non-null value in the list.
`list_append(list, element)`	Appends `element` to `list`.
`list_concat(list1, ..., listn)`	Concatenate lists. `NULL` inputs are skipped. See also `||`
`list_contains(list, element)`	Returns true if the list contains the element.
`list_cosine_similarity(list1, list2)`	Compute the cosine similarity between two lists.
`list_cosine_distance(list1, list2)`	Compute the cosine distance between two lists. Equivalent to `1.0 - list_cosine_similarity`.
`list_distance(list1, list2)`	Calculates the Euclidean distance between two points with coordinates given in two inputs lists of equal length.
`list_distinct(list)`	Removes all duplicates and `NULL` values from a list. Does not preserve the original order.
`list_dot_product(list1, list2)`	Computes the dot product of two same-sized lists of numbers.

Name	Description
`list_negative_dot_product(list1, list2)`	Computes the negative dot product of two same-sized lists of numbers. Equivalent to `-list_dot_product`.
`list_extract(list, index)`	Extract the `index`th (1-based) value from the list.
`list_filter(list, lambda)`	Constructs a list from those elements of the input list for which the lambda function returns true. See the Lambda Functions page for more details.
`list_grade_up(list)`	Works like sort, but the results are the indexes that correspond to the position in the original `list` instead of the actual values.
`list_has_all(list, sub-list)`	Returns true if all elements of sub-list exist in list.
`list_has_any(list1, list2)`	Returns true if any elements exist is both lists.
`list_intersect(list1, list2)`	Returns a list of all the elements that exist in both `l1` and `l2`, without duplicates.
`list_position(list, element)`	Returns the index of the element if the list contains the element. If the element is not found, it returns `NULL`.
`list_prepend(element, list)`	Prepends `element` to `list`.
`list_reduce(list, lambda)`	Returns a single value that is the result of applying the lambda function to each element of the input list. See the Lambda Functions page for more details.
`list_resize(list, size[, value])`	Resizes the list to contain `size` elements. Initializes new elements with `value` or `NULL` if `value` is not set.
`list_reverse_sort(list)`	Sorts the elements of the list in reverse order. See the Sorting Lists section for more details about the `NULL` sorting order.
`list_reverse(list)`	Reverses the list.
`list_select(value_list, index_list)`	Returns a list based on the elements selected by the `index_list`.
`list_slice(list, begin, end, step)`	`list_slice` with added `step` feature.
`list_slice(list, begin, end)`	Extract a sublist using slice conventions. Negative values are accepted. See slicing.
`list_sort(list)`	Sorts the elements of the list. See the Sorting Lists section for more details about the sorting order and the `NULL` sorting order.
`list_transform(list, lambda)`	Returns a list that is the result of applying the lambda function to each element of the input list. See the Lambda Functions page for more details.
`list_unique(list)`	Counts the unique elements of a list.
`list_value(any, ...)`	Create a `LIST` containing the argument values.
`list_where(value_list, mask_list)`	Returns a list with the `BOOLEAN`s in `mask_list` applied as a mask to the `value_list`.
`list_zip(list_1, list_2, ...[, truncate])`	Zips *k* `LIST`s to a new `LIST` whose length will be that of the longest list. Its elements are structs of *k* elements from each list `list_1`, ..., `list_k`, missing elements are replaced with `NULL`. If `truncate` is set, all lists are truncated to the smallest list length.
`repeat(list, count)`	Repeat the `list` `count` times.
`unnest(list)`	Unnests a list by one level. Note that this is a special function that alters the cardinality of the result. See the `unnest` page for more details.

`list[index]`

Description	Bracket notation serves as an alias for `list_extract`.
Example	`[4, 5, 6][3]`
Result	6
Alias	`list_extract`

`list[begin:end]`

Description	Bracket notation with colon is an alias for `list_slice`.
Example	`[4, 5, 6][2:3]`
Result	`[5, 6]`
Alias	`list_slice`

`list[begin:end:step]`

Description	`list_slice` in bracket notation with an added `step` feature.
Example	`[4, 5, 6][:-:2]`
Result	`[4, 6]`
Alias	`list_slice`

`array_pop_back(list)`

Description	Returns the list without the last element.
Example	`array_pop_back([4, 5, 6])`
Result	`[4, 5]`

`array_pop_front(list)`

Description	Returns the list without the first element.
Example	`array_pop_front([4, 5, 6])`
Result	`[5, 6]`

`flatten(list_of_lists)`

Description	Concatenate a list of lists into a single list. This only flattens one level of the list (see examples).
Example	`flatten([[1, 2], [3, 4]])`
Result	`[1, 2, 3, 4]`

`len(list)`

Description	Return the length of the list.
Example	`len([1, 2, 3])`
Result	`3`
Alias	`array_length`

`list_aggregate(list, name)`

Description	Executes the aggregate function `name` on the elements of `list`. See the List Aggregates section for more details.
Example	`list_aggregate([1, 2, NULL], 'min')`
Result	`1`
Aliases	`list_aggr`, `aggregate`, `array_aggregate`, `array_aggr`

`list_any_value(list)`

Description	Returns the first non-null value in the list.
Example	`list_any_value([NULL, -3])`
Result	`-3`

`list_append(list, element)`

Description	Appends `element` to `list`.
Example	`list_append([2, 3], 4)`
Result	`[2, 3, 4]`
Aliases	`array_append`, `array_push_back`

`list_concat(list1, ..., listn)`

Description	Concatenates lists. `NULL` inputs are skipped. See also `||`
Example	`list_concat([2, 3], [4, 5, 6], [7])`
Result	`[2, 3, 4, 5, 6, 7]`
Aliases	`list_cat`, `array_concat`, `array_cat`

`list_contains(list, element)`

Description	Returns true if the list contains the element.

Example	`list_contains([1, 2, NULL], 1)`
Result	`true`
Aliases	`list_has`, `array_contains`, `array_has`

`list_cosine_similarity(list1, list2)`

Description	Compute the cosine similarity between two lists.
Example	`list_cosine_similarity([1, 2, 3], [1, 2, 5])`
Result	`0.9759000729485332`

`list_cosine_distance(list1, list2)`

Description	Compute the cosine distance between two lists. Equivalent to `1.0 - list_cosine_similarity`
Example	`list_cosine_distance([1, 2, 3], [1, 2, 5])`
Result	`0.024099927051466796`

`list_distance(list1, list2)`

Description	Calculates the Euclidean distance between two points with coordinates given in two inputs lists of equal length.
Example	`list_distance([1, 2, 3], [1, 2, 5])`
Result	`2.0`

`list_distinct(list)`

Description	Removes all duplicates and `NULL` values from a list. Does not preserve the original order.
Example	`list_distinct([1, 1, NULL, -3, 1, 5])`
Result	`[1, 5, -3]`
Alias	`array_distinct`

`list_dot_product(list1, list2)`

Description	Computes the dot product of two same-sized lists of numbers.
Example	`list_dot_product([1, 2, 3], [1, 2, 5])`
Result	`20.0`
Alias	`list_inner_product`

`list_negative_dot_product(list1, list2)`

Description	Computes the negative dot product of two same-sized lists of numbers. Equivalent to `-list_dot_product`
Example	`list_negative_dot_product([1, 2, 3], [1, 2, 5])`
Result	`-20.0`
Alias	`list_negative_inner_product`

`list_extract(list, index)`

Description	Extract the `index`th (1-based) value from the list.
Example	`list_extract([4, 5, 6], 3)`
Result	`6`
Aliases	`list_element`, `array_extract`

`list_filter(list, lambda)`

Description	Constructs a list from those elements of the input list for which the lambda function returns true. See the Lambda Functions page for more details.
Example	`list_filter([4, 5, 6], x -> x > 4)`
Result	`[5, 6]`
Aliases	`array_filter`, `filter`

`list_grade_up(list)`

Description	Works like sort, but the results are the indexes that correspond to the position in the original `list` instead of the actual values.
Example	`list_grade_up([30, 10, 40, 20])`
Result	`[2, 4, 1, 3]`
Alias	`array_grade_up`

`list_has_all(list, sub-list)`

Description	Returns true if all elements of sub-list exist in list.
Example	`list_has_all([4, 5, 6], [4, 6])`
Result	`true`
Alias	`array_has_all`

`list_has_any(list1, list2)`

Description	Returns true if any elements exist is both lists.
Example	`list_has_any([1, 2, 3], [2, 3, 4])`
Result	`true`
Alias	`array_has_any`

`list_intersect(list1, list2)`

Description	Returns a list of all the elements that exist in both `l1` and `l2`, without duplicates.
Example	`list_intersect([1, 2, 3], [2, 3, 4])`
Result	`[2, 3]`
Alias	`array_intersect`

`list_position(list, element)`

Description	Returns the index of the element if the list contains the element. If the element is not found, it returns `NULL`.
Example	`list_position([1, 2, NULL], 2)`
Result	2
Aliases	`list_indexof`, `array_position`, `array_indexof`

`list_prepend(element, list)`

Description	Prepends `element` to `list`.
Example	`list_prepend(3, [4, 5, 6])`
Result	`[3, 4, 5, 6]`
Aliases	`array_prepend`, `array_push_front`

`list_reduce(list, lambda)`

Description	Returns a single value that is the result of applying the lambda function to each element of the input list. See the Lambda Functions page for more details.
Example	`list_reduce([4, 5, 6], (acc, x) -> acc + x)`
Result	15
Aliases	`array_reduce`, `reduce`

`list_resize(list, size[, value])`

Description	Resizes the list to contain `size` elements. Initializes new elements with `value` or `NULL` if `value` is not set.
Example	`list_resize([1, 2, 3], 5, 0)`
Result	`[1, 2, 3, 0, 0]`
Alias	`array_resize`

`list_reverse_sort(list)`

Description	Sorts the elements of the list in reverse order. See the Sorting Lists section for more details about the `NULL` sorting order.
Example	`list_reverse_sort([3, 6, 1, 2])`
Result	`[6, 3, 2, 1]`
Alias	`array_reverse_sort`

`list_reverse(list)`

Description	Reverses the list.
Example	`list_reverse([3, 6, 1, 2])`
Result	`[2, 1, 6, 3]`
Alias	`array_reverse`

`list_select(value_list, index_list)`

Description	Returns a list based on the elements selected by the `index_list`.
Example	`list_select([10, 20, 30, 40], [1, 4])`
Result	`[10, 40]`
Alias	`array_select`

`list_slice(list, begin, end, step)`

Description	`list_slice` with added `step` feature.
Example	`list_slice([4, 5, 6], 1, 3, 2)`
Result	`[4, 6]`
Alias	`array_slice`

`list_slice(list, begin, end)`

Description	Extract a sublist using slice conventions. Negative values are accepted. See slicing.
Example	`list_slice([4, 5, 6], 2, 3)`
Result	`[5, 6]`
Alias	`array_slice`

`list_sort(list)`

Description	Sorts the elements of the list. See the Sorting Lists section for more details about the sorting order and the `NULL` sorting order.
Example	`list_sort([3, 6, 1, 2])`
Result	`[1, 2, 3, 6]`
Alias	`array_sort`

`list_transform(list, lambda)`

Description	Returns a list that is the result of applying the lambda function to each element of the input list. See the Lambda Functions page for more details.
Example	`list_transform([4, 5, 6], x -> x + 1)`
Result	`[5, 6, 7]`
Aliases	`array_transform`, `apply`, `list_apply`, `array_apply`

`list_unique(list)`

Description	Counts the unique elements of a list.
Example	`list_unique([1, 1, NULL, -3, 1, 5])`
Result	`3`
Alias	`array_unique`

`list_value(any, ...)`

Description	Create a `LIST` containing the argument values.
Example	`list_value(4, 5, 6)`
Result	`[4, 5, 6]`
Alias	`list_pack`

`list_where(value_list, mask_list)`

Description	Returns a list with the BOOLEANs in `mask_list` applied as a mask to the `value_list`.
Example	`list_where([10, 20, 30, 40], [true, false, false, true])`
Result	`[10, 40]`
Alias	`array_where`

`list_zip(list1, list2, ...)`

Description	Zips *k* `LIST`s to a new `LIST` whose length will be that of the longest list. Its elements are structs of *k* elements from each list `list_1`, ..., `list_k`, missing elements are replaced with `NULL`. If `truncate` is set, all lists are truncated to the smallest list length.
Example	`list_zip([1, 2], [3, 4], [5, 6])`
Result	`[(1, 3, 5), (2, 4, 6)]`
Alias	`array_zip`

`repeat(list, count)`

Description	Repeat the `list` `count` times.
Example	`repeat([1, 2], 5)`
Result	`[1, 2, 1, 2, 1, 2, 1, 2, 1, 2]`

`unnest(list)`

Description	Unnests a list by one level. Note that this is a special function that alters the cardinality of the result. See the `unnest` page for more details.
Example	`unnest([1, 2, 3])`
Result	1, 2, 3

List Operators

The following operators are supported for lists:

Operator	Description	Example	Result
`&&`	Alias for `list_has_any`.	`[1, 2, 3, 4, 5] && [2, 5, 5, 6]`	`true`
`@>`	Alias for `list_has_all`, where the list on the **right** of the operator is the sublist.	`[1, 2, 3, 4] @> [3, 4, 3]`	`true`

Operator	Description	Example	Result
`<@`	Alias for `list_has_all`, where the list on the **left** of the operator is the sublist.	`[1, 4] <@ [1, 2, 3, 4]`	`true`
`||`	Similar to `list_concat`, except any `NULL` input results in `NULL`.	`[1, 2, 3] || [4, 5, 6]`	`[1, 2, 3, 4, 5, 6]`
`<=>`	Alias for `list_cosine_distance`.	`[1, 2, 3] <=> [1, 2, 5]`	`0.007416606`
`<->`	Alias for `list_distance`.	`[1, 2, 3] <-> [1, 2, 5]`	`2.0`

List Comprehension

Python-style list comprehension can be used to compute expressions over elements in a list. For example:

```
SELECT [lower(x) FOR x IN strings] AS strings
FROM (VALUES (['Hello', '', 'World'])) t(strings);
```

strings
[hello, , world]

```
SELECT [upper(x) FOR x IN strings IF len(x) > 0] AS strings
FROM (VALUES (['Hello', '', 'World'])) t(strings);
```

strings
[HELLO, WORLD]

List comprehensions can also use the position of the list elements by adding a second variable. In the following example, we use `x, i`, where `x` is the value and `i` is the position:

```
SELECT [4, 5, 6] AS l, [x FOR x, i IN l IF i != 2] AS filtered;
```

l	filtered
[4, 5, 6]	[4, 6]

Under the hood, `[f(x) FOR x IN y IF g(x)]` is translated to `list_transform(list_filter(y, x -> f(x)), x -> f(x))`.

Range Functions

DuckDB offers two range functions, `range(start, stop, step)` and `generate_series(start, stop, step)`, and their variants with default arguments for `stop` and `step`. The two functions' behavior is different regarding their `stop` argument. This is documented below.

range

The `range` function creates a list of values in the range between `start` and `stop`. The `start` parameter is inclusive, while the `stop` parameter is exclusive. The default value of `start` is 0 and the default value of `step` is 1.

Based on the number of arguments, the following variants of `range` exist.

range(stop)

```
SELECT range(5);
```

```
[0, 1, 2, 3, 4]
```

range(start, stop)

```
SELECT range(2, 5);
```

```
[2, 3, 4]
```

range(start, stop, step)

```
SELECT range(2, 5, 3);
```

```
[2]
```

generate_series

The `generate_series` function creates a list of values in the range between `start` and `stop`. Both the `start` and the `stop` parameters are inclusive. The default value of `start` is 0 and the default value of `step` is 1. Based on the number of arguments, the following variants of `generate_series` exist.

generate_series(stop)

```
SELECT generate_series(5);
```

```
[0, 1, 2, 3, 4, 5]
```

generate_series(start, stop)

```
SELECT generate_series(2, 5);
```

```
[2, 3, 4, 5]
```

generate_series(start, stop, step)

```
SELECT generate_series(2, 5, 3);
```

```
[2, 5]
```

`generate_subscripts(arr, dim)`

The `generate_subscripts(arr, dim)` function generates indexes along the `dim`th dimension of array `arr`.

```
SELECT generate_subscripts([4, 5, 6], 1) AS i;
```

i
1
2
3

Date Ranges

Date ranges are also supported for `TIMESTAMP` and `TIMESTAMP WITH TIME ZONE` values. Note that for these types, the `stop` and `step` arguments have to be specified explicitly (a default value is not provided).

`range` for Date Ranges

```
SELECT *
FROM range(DATE '1992-01-01', DATE '1992-03-01', INTERVAL '1' MONTH);
```

range
1992-01-01 00:00:00
1992-02-01 00:00:00

`generate_series` for Date Ranges

```
SELECT *
FROM generate_series(DATE '1992-01-01', DATE '1992-03-01', INTERVAL '1' MONTH);
```

generate_series
1992-01-01 00:00:00
1992-02-01 00:00:00
1992-03-01 00:00:00

Slicing

The function `list_slice` can be used to extract a sublist from a list. The following variants exist:

- `list_slice(list, begin, end)`
- `list_slice(list, begin, end, step)`
- `array_slice(list, begin, end)`
- `array_slice(list, begin, end, step)`
- `list[begin:end]`
- `list[begin:end:step]`

The arguments are as follows:

- `list`
  - Is the list to be sliced
- `begin`
  - Is the index of the first element to be included in the slice
  - When `begin < 0` the index is counted from the end of the list
  - When `begin < 0` and `-begin > length`, `begin` is clamped to the beginning of the list
  - When `begin > length`, the result is an empty list
  - **Bracket Notation:** When `begin` is omitted, it defaults to the beginning of the list
- `end`
  - Is the index of the last element to be included in the slice
  - When `end < 0` the index is counted from the end of the list
  - When `end > length`, end is clamped to `length`
  - When `end < begin`, the result is an empty list
  - **Bracket Notation:** When `end` is omitted, it defaults to the end of the list. When `end` is omitted and a `step` is provided, `end` must be replaced with a `-`
- `step` *(optional)*
  - Is the step size between elements in the slice
  - When `step < 0` the slice is reversed, and `begin` and `end` are swapped
  - Must be non-zero

Examples:

```
SELECT list_slice([1, 2, 3, 4, 5], 2, 4);
```

```
[2, 3, 4]
```

```
SELECT ([1, 2, 3, 4, 5])[2:4:2];
```

```
[2, 4]
```

```
SELECT([1, 2, 3, 4, 5])[4:2:-2];
```

```
[4, 2]
```

```
SELECT ([1, 2, 3, 4, 5])[:];
```

```
[1, 2, 3, 4, 5]
```

```
SELECT ([1, 2, 3, 4, 5])[:-:2];
```

```
[1, 3, 5]
```

```
SELECT ([1, 2, 3, 4, 5])[:-:-2];
```

```
[5, 3, 1]
```

List Aggregates

The function `list_aggregate` allows the execution of arbitrary existing aggregate functions on the elements of a list. Its first argument is the list (column), its second argument is the aggregate function name, e.g., `min`, `histogram` or `sum`.

`list_aggregate` accepts additional arguments after the aggregate function name. These extra arguments are passed directly to the aggregate function, which serves as the second argument of `list_aggregate`.

```
SELECT list_aggregate([1, 2, -4, NULL], 'min');
```

```
-4
```

```
SELECT list_aggregate([2, 4, 8, 42], 'sum');
```

```
56
```

```
SELECT list_aggregate([[1, 2], [NULL], [2, 10, 3]], 'last');
```

```
[2, 10, 3]
```

```
SELECT list_aggregate([2, 4, 8, 42], 'string_agg', '|');
```

```
2|4|8|42
```

`list_*` Rewrite Functions

The following is a list of existing rewrites. Rewrites simplify the use of the list aggregate function by only taking the list (column) as their argument. `list_avg`, `list_var_samp`, `list_var_pop`, `list_stddev_pop`, `list_stddev_samp`, `list_sem`, `list_approx_count_distinct`, `list_bit_xor`, `list_bit_or`, `list_bit_and`, `list_bool_and`, `list_bool_or`, `list_count`, `list_entropy`, `list_last`, `list_first`, `list_kurtosis`, `list_kurtosis_pop`, `list_min`, `list_max`, `list_product`, `list_skewness`, `list_sum`, `list_string_agg`, `list_mode`, `list_median`, `list_mad` and `list_histogram`.

```
SELECT list_min([1, 2, -4, NULL]);
```

```
-4
```

```
SELECT list_sum([2, 4, 8, 42]);
```

```
56
```

```
SELECT list_last([[1, 2], [NULL], [2, 10, 3]]);
```

```
[2, 10, 3]
```

`array_to_string`

Concatenates list/array elements using an optional delimiter.

```
SELECT array_to_string([1, 2, 3], '-') AS str;
```

```
1-2-3
```

This is equivalent to the following SQL:

```
SELECT list_aggr([1, 2, 3], 'string_agg', '-') AS str;
```

```
1-2-3
```

Sorting Lists

The function `list_sort` sorts the elements of a list either in ascending or descending order. In addition, it allows to provide whether `NULL` values should be moved to the beginning or to the end of the list. It has the same sorting behavior as DuckDB's `ORDER BY` clause. Therefore, (nested) values compare the same in `list_sort` as in `ORDER BY`.

By default, if no modifiers are provided, DuckDB sorts `ASC NULLS FIRST`. I.e., the values are sorted in ascending order and `NULL` values are placed first. This is identical to the default sort order of SQLite. The default sort order can be changed using `PRAGMA` statements..

`list_sort` leaves it open to the user whether they want to use the default sort order or a custom order. `list_sort` takes up to two additional optional parameters. The second parameter provides the sort order and can be either `ASC` or `DESC`. The third parameter provides the `NULL` order and can be either `NULLS FIRST` or `NULLS LAST`.

This query uses the default sort order and the default `NULL` order.

```
SELECT list_sort([1, 3, NULL, 5, NULL, -5]);
```

```
[NULL, NULL, -5, 1, 3, 5]
```

This query provides the sort order. The NULL order uses the configurable default value.

```
SELECT list_sort([1, 3, NULL, 2], 'ASC');
```

```
[NULL, 1, 2, 3]
```

This query provides both the sort order and the NULL order.

```
SELECT list_sort([1, 3, NULL, 2], 'DESC', 'NULLS FIRST');
```

```
[NULL, 3, 2, 1]
```

`list_reverse_sort` has an optional second parameter providing the NULL sort order. It can be either NULLS FIRST or NULLS LAST.

This query uses the default NULL sort order.

```
SELECT list_sort([1, 3, NULL, 5, NULL, -5]);
```

```
[NULL, NULL, -5, 1, 3, 5]
```

This query provides the NULL sort order.

```
SELECT list_reverse_sort([1, 3, NULL, 2], 'NULLS LAST');
```

```
[3, 2, 1, NULL]
```

Flattening

The flatten function is a scalar function that converts a list of lists into a single list by concatenating each sub-list together. Note that this only flattens one level at a time, not all levels of sub-lists.

Convert a list of lists into a single list:

```
SELECT
    flatten([
        [1, 2],
        [3, 4]
    ]);
```

```
[1, 2, 3, 4]
```

If the list has multiple levels of lists, only the first level of sub-lists is concatenated into a single list:

```
SELECT
    flatten([
        [
            [1, 2],
            [3, 4],
        ],
        [
            [5, 6],
            [7, 8],
        ]
    ]);
```

```
[[1, 2], [3, 4], [5, 6], [7, 8]]
```

In general, the input to the flatten function should be a list of lists (not a single level list). However, the behavior of the flatten function has specific behavior when handling empty lists and `NULL` values.

If the input list is empty, return an empty list:

```
SELECT flatten([]);
```

```
[]
```

If the entire input to flatten is `NULL`, return `NULL`:

```
SELECT flatten(NULL);
```

```
NULL
```

If a list whose only entry is `NULL` is flattened, return an empty list:

```
SELECT flatten([NULL]);
```

```
[]
```

If the sub-list in a list of lists only contains `NULL`, do not modify the sub-list:

```
-- (Note the extra set of parentheses vs. the prior example)
SELECT flatten([[NULL]]);
```

```
[NULL]
```

Even if the only contents of each sub-list is `NULL`, still concatenate them together. Note that no de-duplication occurs when flattening. See `list_distinct` function for de-duplication:

```
SELECT flatten([[NULL], [NULL]]);
```

```
[NULL, NULL]
```

Lambda Functions

DuckDB supports lambda functions in the form `(parameter1, parameter2, ...) -> expression`. For details, see the lambda functions page.

Related Functions

- The aggregate functions `list` and `histogram` produce lists and lists of structs.
- The `unnest` function is used to unnest a list by one level.

Map Functions

Name	Description
`cardinality(map)`	Return the size of the map (or the number of entries in the map).
`element_at(map, key)`	Return the value for a given key, or `NULL` if the key is not contained in the map. The type of the key provided in the second parameter must match the type of the map's keys else an error is returned.
`map_concat(maps...)`	Returns a map created from merging the input `maps`. On key collision the value is taken from the last map with that key.
`map_contains(map, key)`	Checks if a map contains a given key.

Name	Description
`map_contains_entry(map, key, value)`	Check if a map contains a given key-value pair.
`map_contains_value(map, value)`	Checks if a map contains a given value.
`map_entries(map)`	Return a list of struct(k, v) for each key-value pair in the map.
`map_extract(map, key)`	Return the value for a given `key` as a list, or `NULL` if the key is not contained in the map. The type of the key provided in the second parameter must match the type of the map's keys else an error is returned.
`map_extract_value(map, key)`	Returns the value for a given `key` or `NULL` if the `key` is not contained in the map. The type of the key provided in the second parameter must match the type of the map's keys else an error is returned.
`map_from_entries(STRUCT(k, v)[])`	Returns a map created from the entries of the array.
`map_keys(map)`	Return a list of all keys in the map.
`map_values(map)`	Return a list of all values in the map.
`map()`	Returns an empty map.
`map[entry]`	Return the value for a given key, or `NULL` if the key is not contained in the map. The type of the key provided in the second parameter must match the type of the map's keys else an error is returned.

`cardinality(map)`

Description	Return the size of the map (or the number of entries in the map).
Example	`cardinality(map([4, 2], ['a', 'b']))`
Result	`2`

`element_at(map, key)`

Description	Return the value for a given key, or `NULL` if the key is not contained in the map. The type of the key provided in the second parameter must match the type of the map's keys else an error is returned.
Example	`element_at(map([100, 5], [42, 43]), 100)`
Result	`[42]`
Aliases	`map_extract(map, key)`,`map[key]`

`map_concat(maps...)`

Description	Returns a map created from merging the input `maps`. On key collision the value is taken from the last map with that key.
Example	`map_concat(MAP {'key1': 10, 'key2': 20}, MAP {'key3': 30}, MAP {'key2': 5})`
Result	`{key1=10, key2=5, key3=30}`

`map_contains(map, key)`

Description	Checks if a map contains a given key.
Example	`map_contains(MAP {'key1': 10, 'key2': 20, 'key3': 30}, 'key2')`
Result	`true`

`map_contains_entry(map, key, value)`

Description	Check if a map contains a given key-value pair.
Example	`map_contains_entry(MAP {'key1': 10, 'key2': 20, 'key3': 30}, 'key2', 20)`
Result	`true`

`map_contains_value(map, value)`

Description	Checks if a map contains a given value.
Example	`map_contains_value(MAP {'key1': 10, 'key2': 20, 'key3': 30}, 20)`
Result	`true`

`map_entries(map)`

Description	Return a list of struct(k, v) for each key-value pair in the map.
Example	`map_entries(map([100, 5], [42, 43]))`
Result	`[{'key': 100, 'value': 42}, {'key': 5, 'value': 43}]`

`map_extract(map, key)`

Description	Return the value for a given `key` as a list, or `NULL` if the key is not contained in the map. The type of the key provided in the second parameter must match the type of the map's keys else an error is returned.
Example	`map_extract(map([100, 5], [42, 43]), 100)`
Result	`[42]`
Aliases	`element_at(map, key)`, `map[key]`

`map_extract_value(map, key)`

Description	Returns the value for a given `key` or `NULL` if the `key` is not contained in the map. The type of the key provided in the second parameter must match the type of the map's keys else an error is returned.
Example	`map_extract_value(map([100, 5], [42, 43]), 100);`

Result	42

`map_from_entries(STRUCT(k, v)[])`

Description	Returns a map created from the entries of the array.
Example	`map_from_entries([{k: 5, v: 'val1'}, {k: 3, v: 'val2'}])`
Result	`{5=val1, 3=val2}`

`map_keys(map)`

Description	Return a list of all keys in the map.
Example	`map_keys(map([100, 5], [42,43]))`
Result	`[100, 5]`

`map_values(map)`

Description	Return a list of all values in the map.
Example	`map_values(map([100, 5], [42, 43]))`
Result	`[42, 43]`

`map()`

Description	Returns an empty map.
Example	`map()`
Result	`{}`

`map[entry]`

Description	Return the value for a given key, or `NULL` if the key is not contained in the map. The type of the key provided in the second parameter must match the type of the map's keys else an error is returned.
Example	`map([100, 5], ['a', 'b'])[100]`
Result	a
Aliases	`element_at(map, key)`,`map_extract(map, key)`

Nested Functions

There are five nested data types:

Name	Type page	Functions page
ARRAY	ARRAY type	ARRAY functions
LIST	LIST type	LIST functions
MAP	MAP type	MAP functions
STRUCT	STRUCT type	STRUCT functions
UNION	UNION type	UNION functions

Numeric Functions

Numeric Operators

The table below shows the available mathematical operators for numeric types.

Operator	Description	Example	Result
+	Addition	2 + 3	5
-	Subtraction	2 - 3	-1
*	Multiplication	2 * 3	6
/	Float division	5 / 2	2.5
//	Division	5 // 2	2
%	Modulo (remainder)	5 % 4	1
**	Exponent	3 ** 4	81
^	Exponent (alias for **)	3 ^ 4	81
&	Bitwise AND	91 & 15	11
|	Bitwise OR	32 | 3	35
<<	Bitwise shift left	1 << 4	16
>>	Bitwise shift right	8 >> 2	2
~	Bitwise negation	~15	-16
!	Factorial of x	4!	24

Division and Modulo Operators

There are two division operators: / and //. They are equivalent when at least one of the operands is a FLOAT or a DOUBLE. When both operands are integers, / performs floating points division (5 / 2 = 2.5) while // performs integer division (5 // 2 = 2).

Supported Types

The modulo, bitwise, and negation and factorial operators work only on integral data types, whereas the others are available for all numeric data types.

Numeric Functions

The table below shows the available mathematical functions.

Name	Description
`@(x)`	Absolute value. Parentheses are optional if `x` is a column name.
`abs(x)`	Absolute value.
`acos(x)`	Computes the inverse cosine of `x`.
`acosh(x)`	Computes the inverse hyperbolic cosine of `x`.
`add(x, y)`	Alias for `x + y`.
`asin(x)`	Computes the inverse sine of `x`.
`asinh(x)`	Computes the inverse hyperbolic sine of `x`.
`atan(x)`	Computes the inverse tangent of `x`.
`atanh(x)`	Computes the inverse hyperbolic tangent of `x`.
`atan2(y, x)`	Computes the inverse tangent of `(y, x)`.
`bit_count(x)`	Returns the number of bits that are set.
`cbrt(x)`	Returns the cube root of the number.
`ceil(x)`	Rounds the number up.
`ceiling(x)`	Rounds the number up. Alias of `ceil`.
`cos(x)`	Computes the cosine of `x`.
`cot(x)`	Computes the cotangent of `x`.
`degrees(x)`	Converts radians to degrees.
`divide(x, y)`	Alias for `x // y`.
`even(x)`	Round to next even number by rounding away from zero.
`exp(x)`	Computes `e ** x`.
`factorial(x)`	See the `!` operator. Computes the product of the current integer and all integers below it.
`fdiv(x, y)`	Performs integer division (`x // y`) but returns a `DOUBLE` value.
`floor(x)`	Rounds the number down.
`fmod(x, y)`	Calculates the modulo value. Always returns a `DOUBLE` value.
`gamma(x)`	Interpolation of the factorial of `x - 1`. Fractional inputs are allowed.
`gcd(x, y)`	Computes the greatest common divisor of `x` and `y`.
`greatest_common_divisor(x, y)`	Computes the greatest common divisor of `x` and `y`.
`greatest(x1, x2, ...)`	Selects the largest value.
`isfinite(x)`	Returns true if the floating point value is finite, false otherwise.
`isinf(x)`	Returns true if the floating point value is infinite, false otherwise.
`isnan(x)`	Returns true if the floating point value is not a number, false otherwise.
`lcm(x, y)`	Computes the least common multiple of `x` and `y`.
`least_common_multiple(x, y)`	Computes the least common multiple of `x` and `y`.
`least(x1, x2, ...)`	Selects the smallest value.
`lgamma(x)`	Computes the log of the `gamma` function.

Name	Description
`ln(x)`	Computes the natural logarithm of x.
`log(x)`	Computes the base-10 logarithm of x.
`log10(x)`	Alias of `log`. Computes the base-10 logarithm of x.
`log2(x)`	Computes the base-2 log of x.
`multiply(x, y)`	Alias for `x * y`.
`nextafter(x, y)`	Return the next floating point value after x in the direction of y.
`pi()`	Returns the value of pi.
`pow(x, y)`	Computes x to the power of y.
`power(x, y)`	Alias of pow. Computes x to the power of y.
`radians(x)`	Converts degrees to radians.
`random()`	Returns a random number x in the range `0.0 <= x < 1.0`.
`round_even(v NUMERIC, s INTEGER)`	Alias of `roundbankers(v, s)`. Round to s decimal places using the *rounding half to even* rule. Values `s < 0` are allowed.
`round(v NUMERIC, s INTEGER)`	Round to s decimal places. Values `s < 0` are allowed.
`setseed(x)`	Sets the seed to be used for the random function.
`sign(x)`	Returns the sign of x as -1, 0 or 1.
`signbit(x)`	Returns whether the signbit is set or not.
`sin(x)`	Computes the sin of x.
`sqrt(x)`	Returns the square root of the number.
`subtract(x, y)`	Alias for `x - y`.
`tan(x)`	Computes the tangent of x.
`trunc(x)`	Truncates the number.
`xor(x, y)`	Bitwise XOR.

`@(x)`

Description	Absolute value. Parentheses are optional if x is a column name.
Example	`@(-17.4)`
Result	`17.4`
Alias	`abs`

`abs(x)`

Description	Absolute value.
Example	`abs(-17.4)`
Result	`17.4`
Alias	`@`

acos(x)

Description	Computes the inverse cosine of x.
Example	`acos(0.5)`
Result	`1.0471975511965976`

acosh(x)

Description	Computes the inverse hyperbolic cosine of x.
Example	`acosh(1.5)`
Result	`0.9624236501192069`

add(x, y)

Description	Alias for `x + y`.
Example	`add(2, 3)`
Result	`5`

asin(x)

Description	Computes the inverse sine of x.
Example	`asin(0.5)`
Result	`0.5235987755982989`

asinh(x)

Description	Computes the inverse hyperbolix sine of x.
Example	`asinh(0.5)`
Result	`0.48121182505960347`

atan(x)

Description	Computes the inverse tangent of x.
Example	`atan(0.5)`
Result	`0.4636476090008061`

atanh(x)

Description	Computes the inverse hyperbolic tangent of x.
Example	`atanh(0.5)`
Result	`0.5493061443340549`

atan2(y, x)

Description	Computes the inverse tangent (y, x).
Example	`atan2(0.5, 0.5)`
Result	`0.7853981633974483`

bit_count(x)

Description	Returns the number of bits that are set.
Example	`bit_count(31)`
Result	5

cbrt(x)

Description	Returns the cube root of the number.
Example	`cbrt(8)`
Result	2

ceil(x)

Description	Rounds the number up.
Example	`ceil(17.4)`
Result	18

ceiling(x)

Description	Rounds the number up. Alias of `ceil`.
Example	`ceiling(17.4)`
Result	18

cos(x)

Description	Computes the cosine of x.
Example	`cos(90)`
Result	`-0.4480736161291701`

cot(x)

Description	Computes the cotangent of x.
Example	`cot(0.5)`
Result	`1.830487721712452`

degrees(x)

Description	Converts radians to degrees.
Example	`degrees(pi())`
Result	`180`

divide(x, y)

Description	Alias for `x // y`.
Example	`divide(5, 2)`
Result	`2`

even(x)

Description	Round to next even number by rounding away from zero.
Example	`even(2.9)`
Result	`4`

exp(x)

Description	Computes `e ** x`.
Example	`exp(0.693)`
Result	`2`

`factorial(x)`

Description	See the `!` operator. Computes the product of the current integer and all integers below it.
Example	`factorial(4)`
Result	`24`

`fdiv(x, y)`

Description	Performs integer division (`x // y`) but returns a DOUBLE value.
Example	`fdiv(5, 2)`
Result	`2.0`

`floor(x)`

Description	Rounds the number down.
Example	`floor(17.4)`
Result	`17`

`fmod(x, y)`

Description	Calculates the modulo value. Always returns a DOUBLE value.
Example	`fmod(5, 2)`
Result	`1.0`

`gamma(x)`

Description	Interpolation of the factorial of `x - 1`. Fractional inputs are allowed.
Example	`gamma(5.5)`
Result	`52.34277778455352`

`gcd(x, y)`

Description	Computes the greatest common divisor of `x` and `y`.
Example	`gcd(42, 57)`
Result	`3`

`greatest_common_divisor(x, y)`

Description	Computes the greatest common divisor of x and y.
Example	`greatest_common_divisor(42, 57)`
Result	3

`greatest(x1, x2, ...)`

Description	Selects the largest value.
Example	`greatest(3, 2, 4, 4)`
Result	4

`isfinite(x)`

Description	Returns true if the floating point value is finite, false otherwise.
Example	`isfinite(5.5)`
Result	`true`

`isinf(x)`

Description	Returns true if the floating point value is infinite, false otherwise.
Example	`isinf('Infinity'::float)`
Result	`true`

`isnan(x)`

Description	Returns true if the floating point value is not a number, false otherwise.
Example	`isnan('NaN'::float)`
Result	`true`

`lcm(x, y)`

Description	Computes the least common multiple of x and y.
Example	`lcm(42, 57)`
Result	798

least_common_multiple(x, y)

Description	Computes the least common multiple of x and y.
Example	`least_common_multiple(42, 57)`
Result	798

least(x1, x2, ...)

Description	Selects the smallest value.
Example	`least(3, 2, 4, 4)`
Result	2

lgamma(x)

Description	Computes the log of the gamma function.
Example	`lgamma(2)`
Result	0

ln(x)

Description	Computes the natural logarithm of x.
Example	`ln(2)`
Result	0.693

log(x)

Description	Computes the base-10 log of x.
Example	`log(100)`
Result	2

log10(x)

Description	Alias of log. Computes the base-10 log of x.
Example	`log10(1000)`
Result	3

log2(x)

Description	Computes the base-2 log of x.
Example	`log2(8)`
Result	3

multiply(x, y)

Description	Alias for `x * y`.
Example	`multiply(2, 3)`
Result	6

nextafter(x, y)

Description	Return the next floating point value after x in the direction of y.
Example	`nextafter(1::float, 2::float)`
Result	`1.0000001`

pi()

Description	Returns the value of pi.
Example	`pi()`
Result	`3.141592653589793`

pow(x, y)

Description	Computes x to the power of y.
Example	`pow(2, 3)`
Result	8

power(x, y)

Description	Alias of pow. Computes x to the power of y.
Example	`power(2, 3)`
Result	8

`radians(x)`

Description	Converts degrees to radians.
Example	`radians(90)`
Result	`1.5707963267948966`

`random()`

Description	Returns a random number x in the range `0.0 <= x < 1.0`.
Example	`random()`
Result	various

`round_even(v NUMERIC, s INTEGER)`

Description	Alias of `roundbankers(v, s)`. Round to s decimal places using the *rounding half to even* rule. Values `s < 0` are allowed.
Example	`round_even(24.5, 0)`
Result	`24.0`

`round(v NUMERIC, s INTEGER)`

Description	Round to s decimal places. Values `s < 0` are allowed.
Example	`round(42.4332, 2)`
Result	`42.43`

`setseed(x)`

Description	Sets the seed to be used for the random function.
Example	`setseed(0.42)`

`sign(x)`

Description	Returns the sign of x as -1, 0 or 1.
Example	`sign(-349)`
Result	`-1`

`signbit(x)`

Description	Returns whether the signbit is set or not.
Example	`signbit(-1.0)`
Result	`true`

`sin(x)`

Description	Computes the sin of `x`.
Example	`sin(90)`
Result	`0.8939966636005579`

`sqrt(x)`

Description	Returns the square root of the number.
Example	`sqrt(9)`
Result	`3`

`subtract(x, y)`

Description	Alias for `x - y`.
Example	`subtract(2, 3)`
Result	`-1`

`tan(x)`

Description	Computes the tangent of `x`.
Example	`tan(90)`
Result	`-1.995200412208242`

`trunc(x)`

Description	Truncates the number.
Example	`trunc(17.4)`
Result	`17`

xor(x, y)

Description	Bitwise XOR.
Example	`xor(17, 5)`
Result	`20`

Pattern Matching

There are four separate approaches to pattern matching provided by DuckDB: the traditional SQL `LIKE` operator, the more recent `SIMILAR TO` operator (added in SQL:1999), a `GLOB` operator, and POSIX-style regular expressions.

`LIKE`

The `LIKE` expression returns `true` if the string matches the supplied pattern. (As expected, the `NOT LIKE` expression returns `false` if `LIKE` returns `true`, and vice versa. An equivalent expression is `NOT (string LIKE pattern)`.)

If pattern does not contain percent signs or underscores, then the pattern only represents the string itself; in that case `LIKE` acts like the equals operator. An underscore (_) in pattern stands for (matches) any single character; a percent sign (%) matches any sequence of zero or more characters.

`LIKE` pattern matching always covers the entire string. Therefore, if it's desired to match a sequence anywhere within a string, the pattern must start and end with a percent sign.

Some examples:

```
SELECT 'abc' LIKE 'abc';    -- true
SELECT 'abc' LIKE 'a%' ;    -- true
SELECT 'abc' LIKE '_b_';    -- true
SELECT 'abc' LIKE 'c';      -- false
SELECT 'abc' LIKE 'c%' ;    -- false
SELECT 'abc' LIKE '%c';     -- true
SELECT 'abc' NOT LIKE '%c'; -- false
```

The keyword `ILIKE` can be used instead of `LIKE` to make the match case-insensitive according to the active locale:

```
SELECT 'abc' ILIKE '%C'; -- true
```

```
SELECT 'abc' NOT ILIKE '%C'; -- false
```

To search within a string for a character that is a wildcard (% or _), the pattern must use an `ESCAPE` clause and an escape character to indicate the wildcard should be treated as a literal character instead of a wildcard. See an example below.

Additionally, the function `like_escape` has the same functionality as a `LIKE` expression with an `ESCAPE` clause, but using function syntax. See the Text Functions Docs for details.

Search for strings with 'a' then a literal percent sign then 'c':

```
SELECT 'a%c' LIKE 'a$%c' ESCAPE '$'; -- true
SELECT 'azc' LIKE 'a$%c' ESCAPE '$'; -- false
```

Case-insensitive ILIKE with ESCAPE:

```
SELECT 'A%c' ILIKE 'a$%c' ESCAPE '$'; -- true
```

There are also alternative characters that can be used as keywords in place of `LIKE` expressions. These enhance PostgreSQL compatibility.

PostgreSQL-style	`LIKE`-style
`~~`	`LIKE`
`!~~`	`NOT LIKE`
`~~*`	`ILIKE`
`!~~*`	`NOT ILIKE`

`SIMILAR TO`

The `SIMILAR TO` operator returns true or false depending on whether its pattern matches the given string. It is similar to `LIKE`, except that it interprets the pattern using a regular expression. Like `LIKE`, the `SIMILAR TO` operator succeeds only if its pattern matches the entire string; this is unlike common regular expression behavior where the pattern can match any part of the string.

A regular expression is a character sequence that is an abbreviated definition of a set of strings (a regular set). A string is said to match a regular expression if it is a member of the regular set described by the regular expression. As with `LIKE`, pattern characters match string characters exactly unless they are special characters in the regular expression language — but regular expressions use different special characters than `LIKE` does.

Some examples:

```
SELECT 'abc' SIMILAR TO 'abc';        -- true
SELECT 'abc' SIMILAR TO 'a';          -- false
SELECT 'abc' SIMILAR TO '.*(b|d).*'; -- true
SELECT 'abc' SIMILAR TO '(b|c).*';    -- false
SELECT 'abc' NOT SIMILAR TO 'abc';    -- false
```

> In PostgreSQL, `~` is equivalent to `SIMILAR TO` and `!~` is equivalent to `NOT SIMILAR TO`. In DuckDB, these equivalences do not hold currently, see the PostgreSQL compatibility page.

Globbing

DuckDB supports file name expansion, also known as globbing, for discovering files. DuckDB's glob syntax uses the question mark (`?`) wildcard to match any single character and the asterisk (`*`) to match zero or more characters. In addition, you can use the bracket syntax (`[...]`) to match any single character contained within the brackets, or within the character range specified by the brackets. An exclamation mark (`!`) may be used inside the first bracket to search for a character that is not contained within the brackets. To learn more, visit the "glob (programming)" Wikipedia page.

`GLOB`

The `GLOB` operator returns `true` or `false` if the string matches the `GLOB` pattern. The `GLOB` operator is most commonly used when searching for filenames that follow a specific pattern (for example a specific file extension).

Some examples:

```
SELECT 'best.txt' GLOB '*.txt';            -- true
SELECT 'best.txt' GLOB '????.txt';         -- true
SELECT 'best.txt' GLOB '?.txt';            -- false
SELECT 'best.txt' GLOB '[abc]est.txt';     -- true
SELECT 'best.txt' GLOB '[a-z]est.txt';     -- true
```

The bracket syntax is case-sensitive:

```
SELECT 'Best.txt' GLOB '[a-z]est.txt';     -- false
SELECT 'Best.txt' GLOB '[a-zA-Z]est.txt';  -- true
```

The `!` applies to all characters within the brackets:

```
SELECT 'Best.txt' GLOB '[!a-zA-Z]est.txt'; -- false
```

To negate a GLOB operator, negate the entire expression:

```
SELECT NOT 'best.txt' GLOB '*.txt';        -- false
```

Three tildes (~~~) may also be used in place of the `GLOB` keyword.

GLOB-style	Symbolic-style
GLOB	~~~

Glob Function to Find Filenames

The glob pattern matching syntax can also be used to search for filenames using the `glob` table function. It accepts one parameter: the path to search (which may include glob patterns).

Search the current directory for all files:

```
SELECT * FROM glob('*');
```

file
duckdb.exe
test.csv
test.json
test.parquet
test2.csv
test2.parquet
todos.json

Globbing Semantics

DuckDB's globbing implementation follows the semantics of Python's `glob` and not the `glob` used in the shell. A notable difference is the behavior of the `**/` construct: `**/<filename>`{:.language-sql .highlight} will not return a file with `<filename>`{:.language-sql .highlight} in top-level directory. For example, with a `README.md` file present in the directory, the following query finds it:

```
SELECT * FROM glob('README.md');
```

file
README.md

However, the following query returns an empty result:

```
SELECT * FROM glob('**/README.md');
```

Meanwhile, the globbing of Bash, Zsh, etc. finds the file using the same syntax:

```
ls **/README.md
```

```
README.md
```

Regular Expressions

DuckDB's regular expression support is documented on the Regular Expressions page. DuckDB supports some PostgreSQL-style operators for regular expression matching:

PostgreSQL-style	Equivalent expression
`~`	`regexp_full_match`
`!~`	`NOT regexp_full_match`
`~*`	(not supported)
`!~*`	(not supported)

Regular Expressions

DuckDB offers pattern matching operators (`LIKE`, `SIMILAR TO`, `GLOB`), as well as support for regular expressions via functions.

Regular Expression Syntax

DuckDB uses the RE2 library as its regular expression engine. For the regular expression syntax, see the RE2 docs.

Functions

All functions accept an optional set of options.

Name	Description
`regexp_extract(string, pattern[, group = 0][, options])`	If `string` contains the regexp `pattern`, returns the capturing group specified by optional parameter `group`; otherwise, returns the empty string. The `group` must be a constant value. If no `group` is given, it defaults to 0. A set of optional `options` can be set.
`regexp_extract(string, pattern, name_list[, options])`	If `string` contains the regexp `pattern`, returns the capturing groups as a struct with corresponding names from `name_list`; otherwise, returns a struct with the same keys and empty strings as values.
`regexp_extract_all(string, regex[, group = 0][, options])`	Finds non-overlapping occurrences of `regex` in `string` and returns the corresponding values of `group`.
`regexp_full_match(string, regex[, options])`	Returns `true` if the entire `string` matches the `regex`.
`regexp_matches(string, pattern[, options])`	Returns `true` if `string` contains the regexp `pattern`, `false` otherwise.

Name	Description
`regexp_replace(string, pattern, replacement[, options])`	If `string` contains the regexp `pattern`, replaces the matching part with `replacement`. By default, only the first occurrence is replaced. A set of optional `options`, including the global flag `g`, can be set.
`regexp_split_to_array(string, regex[, options])`	Alias of `string_split_regex`. Splits the `string` along the `regex`.
`regexp_split_to_table(string, regex[, options])`	Splits the `string` along the `regex` and returns a row for each part.

`regexp_extract(string, pattern[, group = 0][, options])`

Description	If `string` contains the regexp `pattern`, returns the capturing group specified by optional parameter `group`; otherwise, returns the empty string. The `group` must be a constant value. If no `group` is given, it defaults to 0. A set of optional `options` can be set.
Example	`regexp_extract('abc', '([a-z])(b)', 1)`
Result	a

`regexp_extract(string, pattern, name_list[, options])`

Description	If `string` contains the regexp `pattern`, returns the capturing groups as a struct with corresponding names from `name_list`; otherwise, returns a struct with the same keys and empty strings as values. A set of optional `options` can be set.
Example	`regexp_extract('2023-04-15', '(\d+)-(\d+)-(\d+)', ['y', 'm', 'd'])`
Result	`{'y':'2023', 'm':'04', 'd':'15'}`

`regexp_extract_all(string, regex[, group = 0][, options])`

Description	Finds non-overlapping occurrences of `regex` in `string` and returns the corresponding values of `group`. A set of optional `options` can be set.
Example	`regexp_extract_all('Peter: 33, Paul:14', '(\w+):\s*(\d+)', 2)`
Result	`[33, 14]`

`regexp_full_match(string, regex[, options])`

Description	Returns `true` if the entire `string` matches the `regex`. A set of optional `options` can be set.
Example	`regexp_full_match('anabanana', '(an)*')`
Result	`false`

`regexp_matches(string, pattern[, options])`

Description	Returns `true` if `string` contains the regexp `pattern`, `false` otherwise. A set of optional `options` can be set.
Example	`regexp_matches('anabanana', '(an)*')`
Result	`true`

`regexp_replace(string, pattern, replacement[, options])`

Description	If `string` contains the regexp `pattern`, replaces the matching part with `replacement`. By default, only the first occurrence is replaced. A set of optional `options`, including the global flag `g`, can be set.
Example	`regexp_replace('hello', '[lo]', '-')`
Result	`he-lo`

`regexp_split_to_array(string, regex[, options])`

Description	Alias of `string_split_regex`. Splits the `string` along the `regex`. A set of optional `options` can be set.
Example	`regexp_split_to_array('hello world; 42', ';? ')`
Result	`['hello', 'world', '42']`

`regexp_split_to_table(string, regex[, options])`

Description	Splits the `string` along the `regex` and returns a row for each part. A set of optional `options` can be set.
Example	`regexp_split_to_table('hello world; 42', ';? ')`
Result	Three rows: `'hello'`, `'world'`, `'42'`

The `regexp_matches` function is similar to the `SIMILAR TO` operator, however, it does not require the entire string to match. Instead, `regexp_matches` returns `true` if the string merely contains the pattern (unless the special tokens `^` and `$` are used to anchor the regular expression to the start and end of the string). Below are some examples:

```
SELECT regexp_matches('abc', 'abc');       -- true
SELECT regexp_matches('abc', '^abc$');     -- true
SELECT regexp_matches('abc', 'a');         -- true
SELECT regexp_matches('abc', '^a$');       -- false
SELECT regexp_matches('abc', '.*(b|d).*'); -- true
SELECT regexp_matches('abc', '(b|c).*');   -- true
SELECT regexp_matches('abc', '^(b|c).*');  -- false
SELECT regexp_matches('abc', '(?i)A');     -- true
SELECT regexp_matches('abc', 'A', 'i');    -- true
```

Options for Regular Expression Functions

The regex functions support the following `options`.

Option	Description
`'c'`	Case-sensitive matching
`'i'`	Case-insensitive matching
`'l'`	Match literals instead of regular expression tokens
`'m','n','p'`	Newline sensitive matching
`'g'`	Global replace, only available for `regexp_replace`
`'s'`	Non-newline sensitive matching

For example:

```
SELECT regexp_matches('abcd', 'ABC', 'c'); -- false
SELECT regexp_matches('abcd', 'ABC', 'i'); -- true
SELECT regexp_matches('ab^/$cd', '^/$', 'l'); -- true
SELECT regexp_matches(E'hello\nworld', 'hello.world', 'p'); -- false
SELECT regexp_matches(E'hello\nworld', 'hello.world', 's'); -- true
```

Using `regexp_matches`

The `regexp_matches` operator will be optimized to the `LIKE` operator when possible. To achieve best performance, the `'c'` option (case-sensitive matching) should be passed if applicable. Note that by default the RE2 library doesn't match the `.` character to newline.

Original	Optimized equivalent
`regexp_matches('hello world', '^hello', 'c')`	`prefix('hello world', 'hello')`
`regexp_matches('hello world', 'world$', 'c')`	`suffix('hello world', 'world')`
`regexp_matches('hello world', 'hello.world', 'c')`	`LIKE 'hello_world'`
`regexp_matches('hello world', 'he.*rld', 'c')`	`LIKE '%he%rld'`

Using `regexp_replace`

The `regexp_replace` function can be used to replace the part of a string that matches the regexp pattern with a replacement string. The notation `\d` (where d is a number indicating the group) can be used to refer to groups captured in the regular expression in the replacement string. Note that by default, `regexp_replace` only replaces the first occurrence of the regular expression. To replace all occurrences, use the global replace (g) flag.

Some examples for using `regexp_replace`:

```
SELECT regexp_replace('abc', '(b|c)', 'X');         -- aXc
SELECT regexp_replace('abc', '(b|c)', 'X', 'g');    -- aXX
SELECT regexp_replace('abc', '(b|c)', '\1\1\1\1'); -- abbbbc
SELECT regexp_replace('abc', '(.*)c', '\1e');       -- abe
SELECT regexp_replace('abc', '(a)(b)', '\2\1');     -- bac
```

Using `regexp_extract`

The `regexp_extract` function is used to extract a part of a string that matches the regexp pattern. A specific capturing group within the pattern can be extracted using the `group` parameter. If `group` is not specified, it defaults to 0, extracting the first match with the whole pattern.

```
SELECT regexp_extract('abc', '.b.');           -- abc
SELECT regexp_extract('abc', '.b.', 0);        -- abc
SELECT regexp_extract('abc', '.b.', 1);        -- (empty)
SELECT regexp_extract('abc', '([a-z])(b)', 1); -- a
SELECT regexp_extract('abc', '([a-z])(b)', 2); -- b
```

The `regexp_extract` function also supports a `name_list` argument, which is a `LIST` of strings. Using `name_list`, the `regexp_extract` will return the corresponding capture groups as fields of a `STRUCT`:

```
SELECT regexp_extract('2023-04-15', '(\d+)-(\d+)-(\d+)', ['y', 'm', 'd']);
```

```
{'y': 2023, 'm': 04, 'd': 15}
```

```
SELECT regexp_extract('2023-04-15 07:59:56', '^(\d+)-(\d+)-(\d+) (\d+):(\d+):(\d+)', ['y', 'm', 'd']);
```

```
{'y': 2023, 'm': 04, 'd': 15}
```

```
SELECT regexp_extract('duckdb_0_7_1', '^(\w+)_(\d+)_(\d+)', ['tool', 'major', 'minor', 'fix']);
```

```
Binder Error:
Not enough group names in regexp_extract
```

If the number of column names is less than the number of capture groups, then only the first groups are returned. If the number of column names is greater, then an error is generated.

Limitations

Regular expressions only support 9 capture groups: `\1`, `\2`, `\3`, ..., `\9`. Capture groups with two or more digits are not supported.

Struct Functions

Name	Description
`struct.entry`	Dot notation that serves as an alias for `struct_extract` from named STRUCTs.
`struct[entry]`	Bracket notation that serves as an alias for `struct_extract` from named STRUCTs.
`struct[idx]`	Bracket notation that serves as an alias for `struct_extract` from unnamed STRUCTs (tuples), using an index (1-based).
`row(any, ...)`	Create an unnamed STRUCT (tuple) containing the argument values.
`struct_concat(structs...)`	Merge the multiple `structs` into a single STRUCT.
`struct_extract(struct, 'entry')`	Extract the named entry from the STRUCT.
`struct_extract(struct, idx)`	Extract the entry from an unnamed STRUCT (tuple) using an index (1-based).
`struct_extract_at(struct, idx)`	Extract the entry from a STRUCT (tuple) using an index (1-based).
`struct_insert(struct, name := any, ...)`	Add field(s)/value(s) to an existing STRUCT with the argument values. The entry name(s) will be the bound variable name(s).

Name	Description
`struct_pack(name := any, ...)`	Create a STRUCT containing the argument values. The entry name will be the bound variable name.

`struct.entry`

Description	Dot notation that serves as an alias for `struct_extract` from named STRUCTs.
Example	`({'i': 3, 's': 'string'}).i`
Result	3

`struct[entry]`

Description	Bracket notation that serves as an alias for `struct_extract` from named STRUCTs.
Example	`({'i': 3, 's': 'string'})['i']`
Result	3

`struct[idx]`

Description	Bracket notation that serves as an alias for `struct_extract` from unnamed STRUCTs (tuples), using an index (1-based).
Example	`(row(42, 84))[1]`
Result	42

`row(any, ...)`

Description	Create an unnamed STRUCT (tuple) containing the argument values.
Example	`row(i, i % 4, i / 4)`
Result	`(10, 2, 2.5)`

`struct_concat(structs...)`

Description	Merge the multiple `structs` into a single STRUCT.
Example	`struct_concat(struct_pack(i := 4), struct_pack(s := 'string'))`
Result	`{'i': 4, 's': string}`

`struct_extract(struct, 'entry')`

Description	Extract the named entry from the STRUCT.
Example	`struct_extract({'i': 3, 'v2': 3, 'v3': 0}, 'i')`
Result	3

`struct_extract(struct, idx)`

Description	Extract the entry from an unnamed STRUCT (tuple) using an index (1-based).
Example	`struct_extract(row(42, 84), 1)`
Result	42

`struct_extract_at(struct, idx)`

Description	Extract the entry from a STRUCT (tuple) using an index (1-based).
Example	`struct_extract_at({'v1': 10, 'v2': 20, 'v3': 3}, 20)`
Result	20

`struct_insert(struct, name := any, ...)`

Description	Add field(s)/value(s) to an existing STRUCT with the argument values. The entry name(s) will be the bound variable name(s).
Example	`struct_insert({'a': 1}, b := 2)`
Result	`{'a': 1, 'b': 2}`

`struct_pack(name := any, ...)`

Description	Create a STRUCT containing the argument values. The entry name will be the bound variable name.
Example	`struct_pack(i := 4, s := 'string')`
Result	`{'i': 4, 's': string}`

Text Functions

Text Functions and Operators

This section describes functions and operators for examining and manipulating `STRING` values.

Name	Description
`string[index]`	Extracts a single character using a (1-based) index.

Name	Description
`string[begin:end]`	Extracts a string using slice conventions similar to Python. Missing `begin` or `end` arguments are interpreted as the beginning or end of the list respectively. Negative values are accepted.
`string LIKE target`	Returns `true` if the `string` matches the like specifier (see Pattern Matching).
`string SIMILAR TO regex`	Returns `true` if the `string` matches the `regex` (see Pattern Matching).
`string ^@ search_string`	Returns `true` if `string` begins with `search_string`.
`arg1 || arg2`	Concatenates two strings, lists, or blobs. Any `NULL` input results in `NULL`. See also `concat(arg1, arg2, ...)` and `list_concat(list1, list2)`.
`array_extract(string, index)`	Extracts a single character from a `string` using a (1-based) `index`.
`array_slice(list, begin, end)`	Extracts a sublist or substring using slice conventions. Negative values are accepted.
`ascii(string)`	Returns an integer that represents the Unicode code point of the first character of the `string`.
`bar(x, min, max[, width])`	Draws a band whose width is proportional to (`x - min`) and equal to `width` characters when `x` = `max`. `width` defaults to 80.
`base64(blob)`	Converts a `blob` to a base64 encoded string.
`bin(string)`	Converts the `string` to binary representation.
`bit_length(string)`	Number of bits in a `string`.
`char_length(string)`	Number of characters in `string`.
`character_length(string)`	Number of characters in `string`.
`chr(code_point)`	Returns a character which is corresponding the ASCII code value or Unicode code point.
`concat(value, ...)`	Concatenates multiple strings, lists, or blobs. `NULL` inputs are skipped. See also operator `||`.
`concat_ws(separator, string, ...)`	Concatenates many strings, separated by `separator`. `NULL` inputs are skipped.
`contains(string, search_string)`	Returns `true` if `search_string` is found within `string`.
`ends_with(string, search_string)`	Returns `true` if `string` ends with `search_string`.
`format(format, ...)`	Formats a string using the fmt syntax.
`formatReadableDecimalSize(integer)`	Converts `integer` to a human-readable representation using units based on powers of 10 (KB, MB, GB, etc.).
`formatReadableSize(integer)`	Converts `integer` to a human-readable representation using units based on powers of 2 (KiB, MiB, GiB, etc.).
`format_bytes(integer)`	Converts `integer` to a human-readable representation using units based on powers of 2 (KiB, MiB, GiB, etc.).
`from_base64(string)`	Converts a base64 encoded `string` to a character string (`BLOB`).
`from_binary(value)`	Converts a `value` from binary representation to a blob.
`from_hex(value)`	Converts a `value` from hexadecimal representation to a blob.
`greatest(arg1, ...)`	Returns the largest value. For strings lexicographical ordering is used. Note that lowercase characters are considered "larger" than uppercase characters and collations are not supported.
`hash(value, ...)`	Returns a `UBIGINT` with the hash of the `value`. Note that this is not a cryptographic hash.
`hex(string)`	Converts the `string` to hexadecimal representation.

Name	Description
`ilike_escape(string, like_specifier, escape_character)`	Returns `true` if the `string` matches the `like_specifier` (see Pattern Matching) using case-insensitive matching. `escape_character` is used to search for wildcard characters in the `string`.
`instr(string, search_string)`	Returns location of first occurrence of `search_string` in `string`, counting from 1. Returns 0 if no match found.
`lcase(string)`	Converts `string` to lower case.
`least(arg1, ...)`	Returns the smallest value. For strings lexicographical ordering is used. Note that uppercase characters are considered "smaller" than lowercase characters, and collations are not supported.
`left(string, count)`	Extracts the left-most count characters.
`left_grapheme(string, count)`	Extracts the left-most count grapheme clusters.
`len(string)`	Number of characters in `string`.
`length(string)`	Number of characters in `string`.
`length_grapheme(string)`	Number of grapheme clusters in `string`.
`like_escape(string, like_specifier, escape_character)`	Returns `true` if the `string` matches the `like_specifier` (see Pattern Matching) using case-sensitive matching. `escape_character` is used to search for wildcard characters in the `string`.
`lower(string)`	Converts `string` to lower case.
`lpad(string, count, character)`	Pads the `string` with the `character` on the left until it has `count` characters. Truncates the `string` on the right if it has more than `count` characters.
`ltrim(string[, characters])`	Removes any occurrences of any of the `characters` from the left side of the `string`. `characters` defaults to `space`.
`md5(string)`	Returns the MD5 hash of the `string` as a `VARCHAR`.
`md5_number(string)`	Returns the MD5 hash of the `string` as a `HUGEINT`.
`md5_number_lower(string)`	Returns the lower 64-bit segment of the MD5 hash of the `string` as a `UBIGINT`.
`md5_number_upper(string)`	Returns the upper 64-bit segment of the MD5 hash of the `string` as a `UBIGINT`.
`nfc_normalize(string)`	Converts `string` to Unicode NFC normalized string. Useful for comparisons and ordering if text data is mixed between NFC normalized and not.
`not_ilike_escape(string, like_specifier, escape_character)`	Returns `false` if the `string` matches the `like_specifier` (see Pattern Matching) using case-insensitive matching. `escape_character` is used to search for wildcard characters in the `string`.
`not_like_escape(string, like_specifier, escape_character)`	Returns `false` if the `string` matches the `like_specifier` (see Pattern Matching) using case-sensitive matching. `escape_character` is used to search for wildcard characters in the `string`.
`ord(string)`	Returns an `INTEGER` representing the `unicode` codepoint of the first character in the `string`.
`parse_dirname(path[, separator])`	Returns the top-level directory name from the given `path`. `separator` options: `system`, `both_slash` (default), `forward_slash`, `backslash`.
`parse_dirpath(path[, separator])`	Returns the head of the `path` (the pathname until the last slash) similarly to Python's `os.path.dirname`. `separator` options: `system`, `both_slash` (default), `forward_slash`, `backslash`.
`parse_filename(string[, trim_extension][, separator])`	Returns the last component of the `path` similarly to Python's `os.path.basename` function. If `trim_extension` is `true`, the file extension will be removed (defaults to `false`). `separator` options: `system`, `both_slash` (default), `forward_slash`, `backslash`.

Name	Description
`parse_path(path[, separator])`	Returns a list of the components (directories and filename) in the `path` similarly to Python's `pathlib.parts` function. `separator` options: `system`, `both_slash` (default), `forward_slash`, `backslash`.
`position(search_string IN string)`	Return location of first occurrence of `search_string` in `string`, counting from 1. Returns 0 if no match found.
`prefix(string, search_string)`	Returns `true` if `string` starts with `search_string`.
`printf(format, ...)`	Formats a `string` using printf syntax.
`read_text(source)`	Returns the content from `source` (a filename, a list of filenames, or a glob pattern) as a VARCHAR. The file content is first validated to be valid UTF-8. If `read_text` attempts to read a file with invalid UTF-8 an error is thrown suggesting to use `read_blob` instead. See the `read_text` guide for more details.
`regexp_escape(string)`	Escapes special patterns to turn `string` into a regular expression similarly to Python's `re.escape` function.
`regexp_extract(string, regex[, group][, options])`	If `string` contains the `regex` pattern, returns the capturing group specified by optional parameter `group`; otherwise, returns the empty string. The `group` must be a constant value. If no `group` is given, it defaults to 0. A set of optional regex `options` can be set.
`regexp_extract(string, regex, name_list[, options])`	If `string` contains the `regex` pattern, returns the capturing groups as a struct with corresponding names from `name_list`; otherwise, returns a struct with the same keys and empty strings as values. A set of optional regex `options` can be set.
`regexp_extract_all(string, regex[, group][, options])`	Finds non-overlapping occurrences of the `regex` in the `string` and returns the corresponding values of the capturing `group`. A set of optional regex `options` can be set.
`regexp_full_match(string, regex[, col2])`	Returns `true` if the entire `string` matches the `regex`. A set of optional regex `options` can be set.
`regexp_matches(string, regex[, options])`	Returns `true` if `string` contains the `regex`, `false` otherwise. A set of optional regex `options` can be set.
`regexp_replace(string, regex, replacement[, options])`	If `string` contains the `regex`, replaces the matching part with `replacement`. A set of optional regex `options` can be set.
`regexp_split_to_array(string, regex[, options])`	Splits the `string` along the `regex`. A set of optional regex `options` can be set.
`regexp_split_to_table(string, regex)`	Splits the `string` along the `regex` and returns a row for each part.
`repeat(string, count)`	Repeats the `string` `count` number of times.
`replace(string, source, target)`	Replaces any occurrences of the `source` with `target` in `string`.
`reverse(string)`	Reverses the `string`.
`right(string, count)`	Extract the right-most `count` characters.
`right_grapheme(string, count)`	Extracts the right-most `count` grapheme clusters.
`rpad(string, count, character)`	Pads the `string` with the `character` on the right until it has `count` characters. Truncates the `string` on the right if it has more than `count` characters.

Name	Description
`rtrim(string[, characters])`	Removes any occurrences of any of the `characters` from the right side of the `string`. `characters` defaults to `space`.
`sha1(value)`	Returns a VARCHAR with the SHA-1 hash of the `value`.
`sha256(value)`	Returns a VARCHAR with the SHA-256 hash of the `value`
`split(string, separator)`	Splits the `string` along the `separator`.
`split_part(string, separator, index)`	Splits the `string` along the `separator` and returns the data at the (1-based) `index` of the list. If the `index` is outside the bounds of the list, return an empty string (to match PostgreSQL's behavior).
`starts_with(string, search_string)`	Returns `true` if `string` begins with `search_string`.
`str_split(string, separator)`	Splits the `string` along the `separator`.
`str_split_regex(string, regex[, options])`	Splits the `string` along the `regex`. A set of optional regex `options` can be set.
`string_split(string, separator)`	Splits the `string` along the `separator`.
`string_split_regex(string, regex[, options])`	Splits the `string` along the `regex`. A set of optional regex `options` can be set.
`string_to_array(string, separator)`	Splits the `string` along the `separator`.
`strip_accents(string)`	Strips accents from `string`.
`strlen(string)`	Number of bytes in `string`.
`strpos(string, search_string)`	Returns location of first occurrence of `search_string` in `string`, counting from 1. Returns 0 if no match found.
`substr(string, start[, length])`	Extracts substring starting from character `start` up to the end of the string. If optional argument `length` is set, extracts a substring of `length` characters instead. Note that a `start` value of `1` refers to the first character of the `string`.
`substring(string, start[, length])`	Extracts substring starting from character `start` up to the end of the string. If optional argument `length` is set, extracts a substring of `length` characters instead. Note that a `start` value of `1` refers to the first character of the `string`.
`substring_grapheme(string, start[, length])`	Extracts substring starting from grapheme clusters `start` up to the end of the string. If optional argument `length` is set, extracts a substring of `length` grapheme clusters instead. Note that a `start` value of `1` refers to the `first` character of the `string`.
`suffix(string, search_string)`	Returns `true` if `string` ends with `search_string`.
`to_base(number, radix[, min_length])`	Converts `number` to a string in the given base `radix`, optionally padding with leading zeros to `min_length`.
`to_base64(blob)`	Converts a `blob` to a base64 encoded string.
`to_binary(string)`	Converts the `string` to binary representation.
`to_hex(string)`	Converts the `string` to hexadecimal representation.
`translate(string, from, to)`	Replaces each character in `string` that matches a character in the `from` set with the corresponding character in the `to` set. If `from` is longer than `to`, occurrences of the extra characters in `from` are deleted.

Name	Description
`trim(string[, characters])`	Removes any occurrences of any of the `characters` from either side of the `string`. `characters` defaults to `space`.
`ucase(string)`	Converts `string` to upper case.
`unbin(value)`	Converts a `value` from binary representation to a blob.
`unhex(value)`	Converts a `value` from hexadecimal representation to a blob.
`unicode(string)`	Returns an `INTEGER` representing the `unicode` codepoint of the first character in the `string`.
`upper(string)`	Converts `string` to upper case.
`url_decode(string)`	Decodes a URL from a representation using Percent-Encoding.
`url_encode(string)`	Encodes a URL to a representation using Percent-Encoding.

`string[index]`

Description	Extracts a single character using a (1-based) index.
Example	`'DuckDB'[4]`
Result	k
Alias	`array_extract`

`string[begin:end]`

Description	Extracts a string using slice conventions similar to Python. Missing `begin` or `end` arguments are interpreted as the beginning or end of the list respectively. Negative values are accepted.
Example	`'DuckDB'[:4]`
Result	Duck
Alias	`array_slice`

`string LIKE target`

Description	Returns `true` if the `string` matches the like specifier (see Pattern Matching).
Example	`'hello' LIKE '%lo'`
Result	`true`

`string SIMILAR TO regex`

Description	Returns `true` if the `string` matches the `regex` (see Pattern Matching).
Example	`'hello' SIMILAR TO 'l+'`
Result	`false`
Alias	`regexp_full_match`

`string ^@ search_string`

Description	Returns `true` if `string` begins with `search_string`.
Example	`'abc' ^@ 'a'`
Result	`true`
Alias	`starts_with`

`arg1 || arg2`

Description	Concatenates two strings, lists, or blobs. Any `NULL` input results in `NULL`. See also `concat(arg1, arg2, ...)` and `list_concat(list1, list2)`.
Example 1	`'Duck' || 'DB'`
Result	DuckDB
Example 2	`[1, 2, 3] || [4, 5, 6]`
Result	`[1, 2, 3, 4, 5, 6]`
Example 3	`'\xAA'::BLOB || '\xBB'::BLOB`
Result	`\xAA\xBB`

`array_extract(string, index)`

Description	Extracts a single character from a `string` using a (1-based) `index`.
Example	`array_extract('DuckDB', 2)`
Result	`u`

`array_slice(list, begin, end)`

Description	Extracts a sublist or substring using slice conventions. Negative values are accepted.
Example 1	`array_slice('DuckDB', 3, 4)`
Result	`ck`
Example 2	`array_slice('DuckDB', 3, NULL)`
Result	`NULL`
Example 3	`array_slice('DuckDB', 0, -3)`
Result	`Duck`
Alias	`list_slice`

`ascii(string)`

Description	Returns an integer that represents the Unicode code point of the first character of the `string`.
Example	`ascii('Ω')`

Result	937

`bar(x, min, max[, width])`

Description	Draws a band whose width is proportional to (`x - min`) and equal to `width` characters when x = `max`. `width` defaults to 80.
Example	`bar(5, 0, 20, 10)`
Result	

`base64(blob)`

Description	Converts a `blob` to a base64 encoded string.
Example	`base64('A'::BLOB)`
Result	`QQ==`
Alias	`to_base64`

`bin(string)`

Description	Converts the `string` to binary representation.
Example	`bin('Aa')`
Result	`0100000101100001`
Alias	`to_binary`

`bit_length(string)`

Description	Number of bits in a `string`.
Example	`bit_length('abc')`
Result	24

`char_length(string)`

Description	Number of characters in `string`.
Example	`char_length('Hello🦆')`
Result	6
Aliases	`character_length`, `len`, `length`

`character_length(string)`

Description	Number of characters in `string`.
Example	`character_length('Hello🦆')`
Result	6
Aliases	`char_length`, `len`, `length`

`chr(code_point)`

Description	Returns a character which is corresponding the ASCII code value or Unicode code point.
Example	`chr(65)`
Result	A

`concat(value, ...)`

Description	Concatenates multiple strings, lists, or blobs. `NULL` inputs are skipped. See also operator `||`.
Example	`concat('Hello', ' ', 'World')`
Result	`Hello World`

`concat_ws(separator, string, ...)`

Description	Concatenates many strings, separated by `separator`. `NULL` inputs are skipped.
Example	`concat_ws(', ', 'Banana', 'Apple', 'Melon')`
Result	`Banana, Apple, Melon`

`contains(string, search_string)`

Description	Returns `true` if `search_string` is found within `string`.
Example	`contains('abc', 'a')`
Result	`true`

`ends_with(string, search_string)`

Description	Returns `true` if `string` ends with `search_string`.
Example	`ends_with('abc', 'bc')`
Result	`true`
Alias	`suffix`

`format(format, ...)`

Description	Formats a string using the fmt syntax.
Example	`format('Benchmark "{}" took {} seconds', 'CSV', 42)`
Result	`Benchmark "CSV" took 42 seconds`

`formatReadableDecimalSize(integer)`

Description	Converts `integer` to a human-readable representation using units based on powers of 10 (KB, MB, GB, etc.).
Example	`formatReadableDecimalSize(16_000)`
Result	`16.0 kB`

`formatReadableSize(integer)`

Description	Converts `integer` to a human-readable representation using units based on powers of 2 (KiB, MiB, GiB, etc.).
Example	`formatReadableSize(16_000)`
Result	`15.6 KiB`
Alias	`format_bytes`

`format_bytes(integer)`

Description	Converts `integer` to a human-readable representation using units based on powers of 2 (KiB, MiB, GiB, etc.).
Example	`format_bytes(16_000)`
Result	`15.6 KiB`
Alias	`formatReadableSize`

`from_base64(string)`

Description	Converts a base64 encoded `string` to a character string (`BLOB`).
Example	`from_base64('QQ==')`
Result	A

`from_binary(value)`

Description	Converts a `value` from binary representation to a blob.
Example	`from_binary('0110')`

Result	`\x06`
Alias	`unbin`

`from_hex(value)`

Description	Converts a `value` from hexadecimal representation to a blob.
Example	`from_hex('2A')`
Result	`*`
Alias	`unhex`

`greatest(arg1, ...)`

Description	Returns the largest value. For strings lexicographical ordering is used. Note that lowercase characters are considered "larger" than uppercase characters and collations are not supported.
Example 1	`greatest(42, 84)`
Result	`84`
Example 2	`greatest('abc', 'bcd', 'cde', 'EFG')`
Result	`cde`

`hash(value, ...)`

Description	Returns a `UBIGINT` with the hash of the `value`. Note that this is not a cryptographic hash.
Example	`hash('🦆')`
Result	`4164431626903154684`

`hex(string)`

Description	Converts the `string` to hexadecimal representation.
Example	`hex('Hello')`
Result	`48656C6C6F`
Alias	`to_hex`

`ilike_escape(string, like_specifier, escape_character)`

Description	Returns `true` if the `string` matches the `like_specifier` (see Pattern Matching) using case-insensitive matching. `escape_character` is used to search for wildcard characters in the `string`.
Example	`ilike_escape('A%c', 'a$%C', '$')`

Result	`true`

`instr(string, search_string)`

Description	Returns location of first occurrence of `search_string` in `string`, counting from 1. Returns 0 if no match found.
Example	`instr('test test', 'es')`
Result	2
Aliases	`position`, `strpos`

`lcase(string)`

Description	Converts `string` to lower case.
Example	`lcase('Hello')`
Result	`hello`
Alias	`lower`

`least(arg1, ...)`

Description	Returns the smallest value. For strings lexicographical ordering is used. Note that uppercase characters are considered "smaller" than lowercase characters, and collations are not supported.
Example 1	`least(42, 84)`
Result	42
Example 2	`least('abc', 'bcd', 'cde', 'EFG')`
Result	EFG

`left(string, count)`

Description	Extracts the left-most count characters.
Example	`left('Hello🦆', 2)`
Result	He

`left_grapheme(string, count)`

Description	Extracts the left-most count grapheme clusters.
Example	`left_grapheme('👩🏻‍🤝‍👨🏽', 1)`
Result	👩🏻

`len(string)`

Description	Number of characters in `string`.
Example	`length('Hello🦆')`
Result	6
Aliases	`char_length`, `character_length`, `length`

`length(string)`

Description	Number of characters in `string`.
Example	`length('Hello🦆')`
Result	6
Aliases	`char_length`, `character_length`, `len`

`length_grapheme(string)`

Description	Number of grapheme clusters in `string`.
Example	`length_grapheme('👩🏻‍🦲♂👩🏽‍🦲♀')`
Result	2

`like_escape(string, like_specifier, escape_character)`

Description	Returns `true` if the `string` matches the `like_specifier` (see Pattern Matching) using case-sensitive matching. `escape_character` is used to search for wildcard characters in the `string`.
Example	`like_escape('a%c', 'a$%c', '$')`
Result	`true`

`lower(string)`

Description	Converts `string` to lower case.
Example	`lower('Hello')`
Result	`hello`
Alias	`lcase`

`lpad(string, count, character)`

Description	Pads the `string` with the `character` on the left until it has `count` characters. Truncates the `string` on the right if it has more than `count` characters.
Example	`lpad('hello', 8, '>')`
Result	`>>>hello`

`ltrim(string[, characters])`

Description	Removes any occurrences of any of the `characters` from the left side of the `string`. `characters` defaults to `space`.
Example 1	`ltrim('    test  ')`
Result	`test`
Example 2	`ltrim('>>>>test<<', '><')`
Result	`test<<`

`md5(string)`

Description	Returns the MD5 hash of the `string` as a `VARCHAR`.
Example	`md5('abc')`
Result	`900150983cd24fb0d6963f7d28e17f72`

`md5_number(string)`

Description	Returns the MD5 hash of the `string` as a `HUGEINT`.
Example	`md5_number('abc')`
Result	`152195979970564155685860391459828531600`

`md5_number_lower(string)`

Description	Returns the lower 64-bit segment of the MD5 hash of the `string` as a `UBIGINT`.
Example	`md5_number_lower('abc')`
Result	`8250560606382298838`

`md5_number_upper(string)`

Description	Returns the upper 64-bit segment of the MD5 hash of the `string` as a `UBIGINT`.
Example	`md5_number_upper('abc')`
Result	`12704604231530709392`

`nfc_normalize(string)`

Description	Converts `string` to Unicode NFC normalized string. Useful for comparisons and ordering if text data is mixed between NFC normalized and not.
Example	`nfc_normalize('ardèch')`
Result	`ardèch`

`not_ilike_escape(string, like_specifier, escape_character)`

Description	Returns `false` if the `string` matches the `like_specifier` (see Pattern Matching) using case-insensitive matching. `escape_character` is used to search for wildcard characters in the `string`.
Example	`not_ilike_escape('A%c', 'a$%C', '$')`
Result	`false`

`not_like_escape(string, like_specifier, escape_character)`

Description	Returns `false` if the `string` matches the `like_specifier` (see Pattern Matching) using case-sensitive matching. `escape_character` is used to search for wildcard characters in the `string`.
Example	`not_like_escape('a%c', 'a$%c', '$')`
Result	`false`

`ord(string)`

Description	Returns an `INTEGER` representing the `unicode` codepoint of the first character in the `string`.
Example	`[unicode('âbcd'), unicode('â'), unicode(''), unicode(NULL)]`
Result	`[226, 226, -1, NULL]`
Alias	`unicode`

`parse_dirname(path[, separator])`

Description	Returns the top-level directory name from the given `path`. `separator` options: `system`, `both_slash` (default), `forward_slash`, `backslash`.
Example	`parse_dirname('path/to/file.csv', 'system')`
Result	`path`

`parse_dirpath(path[, separator])`

Description	Returns the head of the `path` (the pathname until the last slash) similarly to Python's `os.path.dirname`. `separator` options: `system`, `both_slash` (default), `forward_slash`, `backslash`.
Example	`parse_dirpath('path/to/file.csv', 'forward_slash')`
Result	`path/to`

`parse_filename(string[, trim_extension][, separator])`

Description	Returns the last component of the `path` similarly to Python's `os.path.basename` function. If `trim_extension` is `true`, the file extension will be removed (defaults to `false`). `separator` options: `system`, `both_slash` (default), `forward_slash`, `backslash`.
Example	`parse_filename('path/to/file.csv', true, 'forward_slash')`
Result	`file`

`parse_path(path[, separator])`

Description	Returns a list of the components (directories and filename) in the `path` similarly to Python's `pathlib.parts` function. `separator` options: `system`, `both_slash` (default), `forward_slash`, `backslash`.
Example	`parse_path('path/to/file.csv', 'system')`
Result	`[path, to, file.csv]`

`position(search_string IN string)`

Description	Return location of first occurrence of `search_string` in `string`, counting from 1. Returns 0 if no match found.
Example	`position('b' IN 'abc')`
Result	2
Aliases	`instr`, `strpos`

`prefix(string, search_string)`

Description	Returns `true` if `string` starts with `search_string`.
Example	`prefix('abc', 'ab')`
Result	`true`

`printf(format, ...)`

Description	Formats a `string` using printf syntax.

Example	`printf('Benchmark "%s" took %d seconds', 'CSV', 42)`
Result	`Benchmark "CSV" took 42 seconds`

`read_text(source)`

Description	Returns the content from `source` (a filename, a list of filenames, or a glob pattern) as a VARCHAR. The file content is first validated to be valid UTF-8. If `read_text` attempts to read a file with invalid UTF-8 an error is thrown suggesting to use `read_blob` instead. See the `read_text` guide for more details.
Example	`read_text('hello.txt')`
Result	`hello\n`

`regexp_escape(string)`

Description	Escapes special patterns to turn `string` into a regular expression similarly to Python's `re.escape` function.
Example	`regexp_escape('https://duckdb.org')`
Result	`https\:\/\/duckdb\.org`

`regexp_extract(string, regex[, group][, options])`

Description	If `string` contains the `regex` pattern, returns the capturing group specified by optional parameter `group`; otherwise, returns the empty string. The `group` must be a constant value. If no `group` is given, it defaults to 0. A set of optional regex `options` can be set.
Example	`regexp_extract('ABC', '([a-z])(b)', 1, 'i')`
Result	A

`regexp_extract(string, regex, name_list[, options])`

Description	If `string` contains the `regex` pattern, returns the capturing groups as a struct with corresponding names from `name_list`; otherwise, returns a struct with the same keys and empty strings as values. A set of optional regex `options` can be set.
Example	`regexp_extract('John Doe', '([a-z]+) ([a-z]+)', ['first_name', 'last_name'], 'i')`
Result	`{'first_name': John, 'last_name': Doe}`

`regexp_extract_all(string, regex[, group][, options])`

Description	Finds non-overlapping occurrences of the `regex` in the `string` and returns the corresponding values of the capturing `group`. A set of optional regex `options` can be set.
Example	`regexp_extract_all('Peter: 33, Paul:14', '(\w+):\s*(\d+)', 2)`
Result	`[33, 14]`

`regexp_full_match(string, regex[, col2])`

Description	Returns `true` if the entire `string` matches the `regex`. A set of optional regex `options` can be set.
Example	`regexp_full_match('anabanana', '(an)*')`
Result	`false`

`regexp_matches(string, regex[, options])`

Description	Returns `true` if `string` contains the `regex`, `false` otherwise. A set of optional regex `options` can be set.
Example	`regexp_matches('anabanana', '(an)*')`
Result	`true`

`regexp_replace(string, regex, replacement[, options])`

Description	If `string` contains the `regex`, replaces the matching part with `replacement`. A set of optional regex `options` can be set.
Example	`regexp_replace('hello', '[lo]', '-')`
Result	`he-lo`

`regexp_split_to_array(string, regex[, options])`

Description	Splits the `string` along the `regex`. A set of optional regex `options` can be set.
Example	`regexp_split_to_array('hello world; 42', ';? ')`
Result	`[hello, world, 42]`
Aliases	`str_split_regex`, `string_split_regex`

`regexp_split_to_table(string, regex)`

Description	Splits the `string` along the `regex` and returns a row for each part.
Example	`regexp_split_to_table('hello world; 42', ';? ')`
Result	Multiple rows: `'hello'`, `'world'`, `'42'`

`repeat(string, count)`

Description	Repeats the `string` `count` number of times.
Example	`repeat('A', 5)`
Result	AAAAA

`replace(string, source, target)`

Description	Replaces any occurrences of the `source` with `target` in `string`.
Example	`replace('hello', 'l', '-')`
Result	he--o

`reverse(string)`

Description	Reverses the `string`.
Example	`reverse('hello')`
Result	olleh

`right(string, count)`

Description	Extract the right-most `count` characters.
Example	`right('Hello🍃', 3)`
Result	lo🍃

`right_grapheme(string, count)`

Description	Extracts the right-most `count` grapheme clusters.
Example	`right_grapheme('🤦🏼‍♂🤦🏽‍♀', 1)`
Result	🤦🏽‍♀

`rpad(string, count, character)`

Description	Pads the `string` with the `character` on the right until it has `count` characters. Truncates the `string` on the right if it has more than `count` characters.
Example	`rpad('hello', 10, '<')`
Result	hello<<<<<

rtrim(string[, characters])

Description	Removes any occurrences of any of the `characters` from the right side of the `string`. `characters` defaults to space.
Example 1	`rtrim('    test  ')`
Result	`test`
Example 2	`rtrim('>>>>test<<', '><')`
Result	`>>>>test`

sha1(value)

Description	Returns a VARCHAR with the SHA-1 hash of the `value`.
Example	`sha1('🦆')`
Result	`949bf843dc338be348fb9525d1eb535d31241d76`

sha256(value)

Description	Returns a VARCHAR with the SHA-256 hash of the `value`
Example	`sha256('🦆')`
Result	`d7a5c5e0d1d94c32218539e7e47d4ba9c3c7b77d61332fb60d633dde89e473fb`

split(string, separator)

Description	Splits the `string` along the `separator`.
Example	`split('hello-world', '-')`
Result	`[hello, world]`
Aliases	`str_split`, `string_split`, `string_to_array`

split_part(string, separator, index)

Description	Splits the `string` along the `separator` and returns the data at the (1-based) `index` of the list. If the `index` is outside the bounds of the list, return an empty string (to match PostgreSQL's behavior).
Example	`split_part('a;b;c', ';', 2)`
Result	b

starts_with(string, search_string)

Description	Returns `true` if `string` begins with `search_string`.

Example	`starts_with('abc', 'a')`
Result	`true`
Alias	`^@`

`str_split(string, separator)`

Description	Splits the `string` along the `separator`.
Example	`str_split('hello-world', '-')`
Result	`[hello, world]`
Aliases	`split`, `string_split`, `string_to_array`

`str_split_regex(string, regex[, options])`

Description	Splits the `string` along the `regex`. A set of optional regex `options` can be set.
Example	`str_split_regex('hello world; 42', ';? ')`
Result	`[hello, world, 42]`
Aliases	`regexp_split_to_array`, `string_split_regex`

`string_split(string, separator)`

Description	Splits the `string` along the `separator`.
Example	`string_split('hello-world', '-')`
Result	`[hello, world]`
Aliases	`split`, `str_split`, `string_to_array`

`string_split_regex(string, regex[, options])`

Description	Splits the `string` along the `regex`. A set of optional regex `options` can be set.
Example	`string_split_regex('hello world; 42', ';? ')`
Result	`[hello, world, 42]`
Aliases	`regexp_split_to_array`, `str_split_regex`

`string_to_array(string, separator)`

Description	Splits the `string` along the `separator`.
Example	`string_to_array('hello-world', '-')`
Result	`[hello, world]`

Aliases	`split`, `str_split`, `string_split`

`strip_accents(string)`

Description	Strips accents from `string`.
Example	`strip_accents('mühleisen')`
Result	`muhleisen`

`strlen(string)`

Description	Number of bytes in `string`.
Example	`strlen('🦆')`
Result	4

`strpos(string, search_string)`

Description	Returns location of first occurrence of `search_string` in `string`, counting from 1. Returns 0 if no match found.
Example	`strpos('test test', 'es')`
Result	2
Aliases	`instr`, `position`

`substr(string, start[, length])`

Description	Extracts substring starting from character `start` up to the end of the string. If optional argument `length` is set, extracts a substring of `length` characters instead. Note that a `start` value of `1` refers to the first character of the `string`.
Example 1	`substring('Hello', 2)`
Result	`ello`
Example 2	`substring('Hello', 2, 2)`
Result	`el`
Alias	`substring`

`substring(string, start[, length])`

Description	Extracts substring starting from character `start` up to the end of the string. If optional argument `length` is set, extracts a substring of `length` characters instead. Note that a `start` value of `1` refers to the first character of the `string`.
Example 1	`substring('Hello', 2)`

Result	`ello`
Example 2	`substring('Hello', 2, 2)`
Result	`el`
Alias	`substr`

`substring_grapheme(string, start[, length])`

Description	Extracts substring starting from grapheme clusters `start` up to the end of the string. If optional argument `length` is set, extracts a substring of `length` grapheme clusters instead. Note that a `start` value of `1` refers to the `first` character of the `string`.
Example 1	`substring_grapheme('🦆🤦🏼‍♂️🤦🏽‍♀️🦆', 3)`
Result	🤦🏽‍♀️🦆
Example 2	`substring_grapheme('🦆🤦🏼‍♂️🤦🏽‍♀️🦆', 3, 2)`
Result	🤦🏽‍♀️🦆

`suffix(string, search_string)`

Description	Returns `true` if `string` ends with `search_string`.
Example	`suffix('abc', 'bc')`
Result	`true`
Alias	`ends_with`

`to_base(number, radix[, min_length])`

Description	Converts `number` to a string in the given base `radix`, optionally padding with leading zeros to `min_length`.
Example	`to_base(42, 16, 5)`
Result	`0002A`

`to_base64(blob)`

Description	Converts a `blob` to a base64 encoded string.
Example	`to_base64('A'::BLOB)`
Result	`QQ==`
Alias	`base64`

`to_binary(string)`

Description	Converts the `string` to binary representation.
Example	`to_binary('Aa')`
Result	`0100000101100001`
Alias	`bin`

`to_hex(string)`

Description	Converts the `string` to hexadecimal representation.
Example	`to_hex('Hello')`
Result	`48656C6C6F`
Alias	`hex`

`translate(string, from, to)`

Description	Replaces each character in `string` that matches a character in the `from` set with the corresponding character in the `to` set. If `from` is longer than `to`, occurrences of the extra characters in `from` are deleted.
Example	`translate('12345', '143', 'ax')`
Result	`a2x5`

`trim(string[, characters])`

Description	Removes any occurrences of any of the `characters` from either side of the `string`. `characters` defaults to `space`.
Example 1	`trim('    test  ')`
Result	`test`
Example 2	`trim('>>>>test<<', '><')`
Result	`test`

`ucase(string)`

Description	Converts `string` to upper case.
Example	`ucase('Hello')`
Result	`HELLO`
Alias	`upper`

`unbin(value)`

Description	Converts a `value` from binary representation to a blob.
Example	`unbin('0110')`
Result	`\x06`
Alias	`from_binary`

`unhex(value)`

Description	Converts a `value` from hexadecimal representation to a blob.
Example	`unhex('2A')`
Result	`*`
Alias	`from_hex`

`unicode(string)`

Description	Returns an `INTEGER` representing the `unicode` codepoint of the first character in the `string`.
Example	`[unicode('âbcd'), unicode('â'), unicode(''), unicode(NULL)]`
Result	`[226, 226, -1, NULL]`
Alias	`ord`

`upper(string)`

Description	Converts `string` to upper case.
Example	`upper('Hello')`
Result	`HELLO`
Alias	`ucase`

`url_decode(string)`

Description	Decodes a URL from a representation using Percent-Encoding.
Example	`url_decode('https%3A%2F%2Fduckdb.org%2Fwhy_duckdb%23portable')`
Result	`https://duckdb.org/why_duckdb#portable`

`url_encode(string)`

Description	Encodes a URL to a representation using Percent-Encoding.
Example	`url_encode('this string has/ special+ characters>')`
Result	`this%20string%20has%2F%20special%2B%20characters%3E`

Text Similarity Functions

These functions are used to measure the similarity of two strings using various similarity measures.

Name	Description
`damerau_levenshtein(s1, s2)`	Extension of Levenshtein distance to also include transposition of adjacent characters as an allowed edit operation. In other words, the minimum number of edit operations (insertions, deletions, substitutions or transpositions) required to change one string to another. Characters of different cases (e.g., a and A) are considered different.
`editdist3(s1, s2)`	The minimum number of single-character edits (insertions, deletions or substitutions) required to change one string to the other. Characters of different cases (e.g., a and A) are considered different.
`hamming(s1, s2)`	The Hamming distance between to strings, i.e., the number of positions with different characters for two strings of equal length. Strings must be of equal length. Characters of different cases (e.g., a and A) are considered different.
`jaccard(s1, s2)`	The Jaccard similarity between two strings. Characters of different cases (e.g., a and A) are considered different. Returns a number between 0 and 1.
`jaro_similarity(s1, s2[, score_cutoff])`	The Jaro similarity between two strings. Characters of different cases (e.g., a and A) are considered different. Returns a number between 0 and 1. For similarity < `score_cutoff`, 0 is returned instead. `score_cutoff` defaults to 0.
`jaro_winkler_similarity(s1, s2[, score_cutoff])`	The Jaro-Winkler similarity between two strings. Characters of different cases (e.g., a and A) are considered different. Returns a number between 0 and 1. For similarity < `score_cutoff`, 0 is returned instead. `score_cutoff` defaults to 0.
`levenshtein(s1, s2)`	The minimum number of single-character edits (insertions, deletions or substitutions) required to change one string to the other. Characters of different cases (e.g., a and A) are considered different.
`mismatches(s1, s2)`	The Hamming distance between to strings, i.e., the number of positions with different characters for two strings of equal length. Strings must be of equal length. Characters of different cases (e.g., a and A) are considered different.

`damerau_levenshtein(s1, s2)`

Description	Extension of Levenshtein distance to also include transposition of adjacent characters as an allowed edit operation. In other words, the minimum number of edit operations (insertions, deletions, substitutions or transpositions) required to change one string to another. Characters of different cases (e.g., a and A) are considered different.
Example	`damerau_levenshtein('duckdb', 'udckbd')`
Result	2

`editdist3(s1, s2)`

Description	The minimum number of single-character edits (insertions, deletions or substitutions) required to change one string to the other. Characters of different cases (e.g., a and A) are considered different.
Example	`editdist3('duck', 'db')`
Result	3

Alias	`levenshtein`

`hamming(s1, s2)`

Description	The Hamming distance between to strings, i.e., the number of positions with different characters for two strings of equal length. Strings must be of equal length. Characters of different cases (e.g., a and A) are considered different.
Example	`hamming('duck', 'luck')`
Result	`1`
Alias	`mismatches`

`jaccard(s1, s2)`

Description	The Jaccard similarity between two strings. Characters of different cases (e.g., a and A) are considered different. Returns a number between 0 and 1.
Example	`jaccard('duck', 'luck')`
Result	`0.6`

`jaro_similarity(s1, s2[, score_cutoff])`

Description	The Jaro similarity between two strings. Characters of different cases (e.g., a and A) are considered different. Returns a number between 0 and 1. For similarity < `score_cutoff`, 0 is returned instead. `score_cutoff` defaults to 0.
Example	`jaro_similarity('duck', 'duckdb')`
Result	`0.8888888888888888`

`jaro_winkler_similarity(s1, s2[, score_cutoff])`

Description	The Jaro-Winkler similarity between two strings. Characters of different cases (e.g., a and A) are considered different. Returns a number between 0 and 1. For similarity < `score_cutoff`, 0 is returned instead. `score_cutoff` defaults to 0.
Example	`jaro_winkler_similarity('duck', 'duckdb')`
Result	`0.9333333333333333`

`levenshtein(s1, s2)`

Description	The minimum number of single-character edits (insertions, deletions or substitutions) required to change one string to the other. Characters of different cases (e.g., a and A) are considered different.
Example	`levenshtein('duck', 'db')`

Result	`3`
Alias	`editdist3`

`mismatches(s1, s2)`

Description	The Hamming distance between to strings, i.e., the number of positions with different characters for two strings of equal length. Strings must be of equal length. Characters of different cases (e.g., a and A) are considered different.
Example	`mismatches('duck', 'luck')`
Result	`1`
Alias	`hamming`

Formatters

`fmt` Syntax

The `format(format, parameters...)` function formats strings, loosely following the syntax of the {fmt} open-source formatting library.

Format without additional parameters:

```
SELECT format('Hello world'); -- Hello world
```

Format a string using {}:

```
SELECT format('The answer is {}', 42); -- The answer is 42
```

Format a string using positional arguments:

```
SELECT format('I''d rather be {1} than {0}.', 'right', 'happy'); -- I'd rather be happy than right.
```

Format Specifiers

Specifier	Description	Example
`{:d}`	integer	`654321`
`{:E}`	scientific notation	`3.141593E+00`
`{:f}`	float	`4.560000`
`{:o}`	octal	`2375761`
`{:s}`	string	`asd`
`{:x}`	hexadecimal	`9fbf1`
`{:tX}`	integer, X is the thousand separator	`654 321`

Formatting Types

Integers:

```
SELECT format('{} + {} = {}', 3, 5, 3 + 5); -- 3 + 5 = 8
```

Booleans:

```
SELECT format('{} != {}', true, false); -- true != false
```

Format datetime values:

```
SELECT format('{}', DATE '1992-01-01'); -- 1992-01-01
SELECT format('{}', TIME '12:01:00'); -- 12:01:00
SELECT format('{}', TIMESTAMP '1992-01-01 12:01:00'); -- 1992-01-01 12:01:00
```

Format BLOB:

```
SELECT format('{}', BLOB '\x00hello'); -- \x00hello
```

Pad integers with 0s:

```
SELECT format('{:04d}', 33); -- 0033
```

Create timestamps from integers:

```
SELECT format('{:02d}:{:02d}:{:02d} {}', 12, 3, 16, 'AM'); -- 12:03:16 AM
```

Convert to hexadecimal:

```
SELECT format('{:x}', 123_456_789); -- 75bcd15
```

Convert to binary:

```
SELECT format('{:b}', 123_456_789); -- 111010110111100110100010101
```

Print Numbers with Thousand Separators

Integers:

```
SELECT format('{:,}',  123_456_789); -- 123,456,789
SELECT format('{:t.}', 123_456_789); -- 123.456.789
SELECT format('{:''}', 123_456_789); -- 123'456'789
SELECT format('{:_}',  123_456_789); -- 123_456_789
SELECT format('{:t }', 123_456_789); -- 123 456 789
SELECT format('{:tX}', 123_456_789); -- 123X456X789
```

Float, double and decimal:

```
SELECT format('{:,f}',    123456.789); -- 123,456.78900
SELECT format('{:,.2f}',  123456.789); -- 123,456.79
SELECT format('{:t..2f}', 123456.789); -- 123.456,79
```

`printf` Syntax

The `printf(format, parameters...)` function formats strings using the `printf` syntax.

Format without additional parameters:

```
SELECT printf('Hello world');
```

```
Hello world
```

Format a string using arguments in a given order:

```sql
SELECT printf('The answer to %s is %d', 'life', 42);
```

```
The answer to life is 42
```

Format a string using positional arguments `%position$formatter`, e.g., the second parameter as a string is encoded as `%2$s`:

```sql
SELECT printf('I''d rather be %2$s than %1$s.', 'right', 'happy');
```

```
I'd rather be happy than right.
```

Format Specifiers

Specifier	Description	Example
%c	character code to character	a
%d	integer	654321
%Xd	integer with thousand seperarator X from ,, ., '', _	654_321
%E	scientific notation	3.141593E+00
%f	float	4.560000
%hd	integer	654321
%hhd	integer	654321
%lld	integer	654321
%o	octal	2375761
%s	string	asd
%x	hexadecimal	9fbf1

Formatting Types

Integers:

```sql
SELECT printf('%d + %d = %d', 3, 5, 3 + 5); -- 3 + 5 = 8
```

Booleans:

```sql
SELECT printf('%s != %s', true, false); -- true != false
```

Format datetime values:

```sql
SELECT printf('%s', DATE '1992-01-01'); -- 1992-01-01
SELECT printf('%s', TIME '12:01:00'); -- 12:01:00
SELECT printf('%s', TIMESTAMP '1992-01-01 12:01:00'); -- 1992-01-01 12:01:00
```

Format BLOB:

```sql
SELECT printf('%s', BLOB '\x00hello'); -- \x00hello
```

Pad integers with 0s:

```sql
SELECT printf('%04d', 33); -- 0033
```

Create timestamps from integers:

```sql
SELECT printf('%02d:%02d:%02d %s', 12, 3, 16, 'AM'); -- 12:03:16 AM
```

Convert to hexadecimal:

```sql
SELECT printf('%x', 123_456_789); -- 75bcd15
```

Convert to binary:

```
SELECT printf('%b', 123_456_789); -- 111010110111100110100010101
```

Thousand Separators

Integers:

```
SELECT printf('%,d',  123_456_789); -- 123,456,789
SELECT printf('%.d',  123_456_789); -- 123.456.789
SELECT printf('%''d', 123_456_789); -- 123'456'789
SELECT printf('%_d',  123_456_789); -- 123_456_789
```

Float, double and decimal:

```
SELECT printf('%,f',    123456.789); -- 123,456.789000
SELECT printf('%,.2f', 123456.789); -- 123,456.79
```

Time Functions

This section describes functions and operators for examining and manipulating TIME values.

Time Operators

The table below shows the available mathematical operators for TIME types.

Operator	Description	Example	Result
+	addition of an INTERVAL	TIME '01:02:03' + INTERVAL 5 HOUR	06:02:03
-	subtraction of an INTERVAL	TIME '06:02:03' - INTERVAL 5 HOUR	01:02:03

Time Functions

The table below shows the available scalar functions for TIME types.

Name	Description
current_time	Current time (start of current transaction) in the local time zone. Note that parentheses should be omitted from the function call.
date_diff(part, starttime, endtime)	The number of partition boundaries between the times.
date_part(part, time)	Get subfield (equivalent to extract).
date_sub(part, starttime, endtime)	The number of complete partitions between the times.
datediff(part, starttime, endtime)	Alias of date_diff. The number of partition boundaries between the times.
datepart(part, time)	Alias of date_part. Get subfield (equivalent to extract).

Name	Description
`datesub(part, starttime, endtime)`	Alias of date_sub. The number of complete partitions between the times.
`extract(part FROM time)`	Get subfield from a time.
`get_current_time()`	Current time (start of current transaction) in UTC.
`make_time(bigint, bigint, double)`	The time for the given parts.

The only date parts that are defined for times are `epoch`, `hours`, `minutes`, `seconds`, `milliseconds` and `microseconds`.

`current_time`

Description	Current time (start of current transaction) in the local time zone. Note that parentheses should be omitted from the function call.
Example	`current_time`
Result	`10:31:58.578`
Alias	`get_current_time()`

`date_diff(part, starttime, endtime)`

Description	The number of partition boundaries between the times.
Example	`date_diff('hour', TIME '01:02:03', TIME '06:01:03')`
Result	`5`

`date_part(part, time)`

Description	Get subfield (equivalent to `extract`).
Example	`date_part('minute', TIME '14:21:13')`
Result	`21`

`date_sub(part, starttime, endtime)`

Description	The number of complete partitions between the times.
Example	`date_sub('hour', TIME '01:02:03', TIME '06:01:03')`
Result	`4`

`datediff(part, starttime, endtime)`

Description	Alias of `date_diff`. The number of partition boundaries between the times.
Example	`datediff('hour', TIME '01:02:03', TIME '06:01:03')`
Result	`5`

`datepart(part, time)`

Description	Alias of date_part. Get subfield (equivalent to `extract`).
Example	`datepart('minute', TIME '14:21:13')`
Result	`21`

`datesub(part, starttime, endtime)`

Description	Alias of date_sub. The number of complete partitions between the times.
Example	`datesub('hour', TIME '01:02:03', TIME '06:01:03')`
Result	`4`

`extract(part FROM time)`

Description	Get subfield from a time.
Example	`extract('hour' FROM TIME '14:21:13')`
Result	`14`

`get_current_time()`

Description	Current time (start of current transaction) in UTC.
Example	`get_current_time()`
Result	`10:31:58.578`
Alias	`current_time`

`make_time(bigint, bigint, double)`

Description	The time for the given parts.
Example	`make_time(13, 34, 27.123456)`
Result	`13:34:27.123456`

Timestamp Functions

This section describes functions and operators for examining and manipulating `TIMESTAMP` values. See also the related `TIMESTAMPTZ` functions.

Timestamp Operators

The table below shows the available mathematical operators for `TIMESTAMP` types.

Operator	Description	Example	Result
`+`	addition of an `INTERVAL`	`TIMESTAMP '1992-03-22 01:02:03' + INTERVAL 5 DAY`	`1992-03-27 01:02:03`
`-`	subtraction of `TIMESTAMP`s	`TIMESTAMP '1992-03-27' - TIMESTAMP '1992-03-22'`	`5 days`
`-`	subtraction of an `INTERVAL`	`TIMESTAMP '1992-03-27 01:02:03' - INTERVAL 5 DAY`	`1992-03-22 01:02:03`

Adding to or subtracting from infinite values produces the same infinite value.

Scalar Timestamp Functions

The table below shows the available scalar functions for `TIMESTAMP` values.

Name	Description
`age(timestamp, timestamp)`	Subtract arguments, resulting in the time difference between the two timestamps.
`age(timestamp)`	Subtract from current_date.
`century(timestamp)`	Extracts the century of a timestamp.
`current_localtimestamp()`	Returns the current timestamp (at the start of the transaction).
`date_diff(part, startdate, enddate)`	The number of partition boundaries between the timestamps.
`date_part([part, ...], timestamp)`	Get the listed subfields as a `struct`. The list must be constant.
`date_part(part, timestamp)`	Get subfield (equivalent to `extract`).
`date_sub(part, startdate, enddate)`	The number of complete partitions between the timestamps.
`date_trunc(part, timestamp)`	Truncate to specified precision.
`datediff(part, startdate, enddate)`	Alias of `date_diff`. The number of partition boundaries between the timestamps.
`datepart([part, ...], timestamp)`	Alias of `date_part`. Get the listed subfields as a `struct`. The list must be constant.
`datepart(part, timestamp)`	Alias of `date_part`. Get subfield (equivalent to `extract`).
`datesub(part, startdate, enddate)`	Alias of `date_sub`. The number of complete partitions between the timestamps.
`datetrunc(part, timestamp)`	Alias of `date_trunc`. Truncate to specified precision.
`dayname(timestamp)`	The (English) name of the weekday.
`epoch_ms(ms)`	Converts milliseconds since the epoch to a timestamp.

Name	Description
`epoch_ms(timestamp)`	Returns the total number of milliseconds since the epoch.
`epoch_ns(timestamp)`	Returns the total number of nanoseconds since the epoch.
`epoch_us(timestamp)`	Returns the total number of microseconds since the epoch.
`epoch(timestamp)`	Returns the total number of seconds since the epoch.
`extract(field FROM timestamp)`	Get subfield from a timestamp.
`greatest(timestamp, timestamp)`	The later of two timestamps.
`isfinite(timestamp)`	Returns true if the timestamp is finite, false otherwise.
`isinf(timestamp)`	Returns true if the timestamp is infinite, false otherwise.
`julian(timestamp)`	Extract the Julian Day number from a timestamp.
`last_day(timestamp)`	The last day of the month.
`least(timestamp, timestamp)`	The earlier of two timestamps.
`make_timestamp(bigint, bigint, bigint, bigint, bigint, double)`	The timestamp for the given parts.
`make_timestamp(microseconds)`	Converts microseconds since the epoch to a timestamp.
`make_timestamp_ns(nanoseconds)`	Converts nanoseconds since the epoch to a timestamp.
`monthname(timestamp)`	The (English) name of the month.
`strftime(timestamp, format)`	Converts timestamp to string according to the format string.
`strptime(text, format-list)`	Converts the string `text` to timestamp applying the format strings in the list until one succeeds. Throws an error on failure. To return `NULL` on failure, use `try_strptime`.
`strptime(text, format)`	Converts the string `text` to timestamp according to the format string. Throws an error on failure. To return `NULL` on failure, use `try_strptime`.
`time_bucket(bucket_width, timestamp[, offset])`	Truncate `timestamp` to a grid of width `bucket_width`. The grid is anchored at `2000-01-01 00:00:00[ + offset]` when `bucket_width` is a number of months or coarser units, else `2000-01-03 00:00:00[ + offset]`. Note that `2000-01-03` is a Monday.
`time_bucket(bucket_width, timestamp[, origin])`	Truncate `timestamp` to a grid of width `bucket_width`. The grid is anchored at the `origin` timestamp, which defaults to `2000-01-01 00:00:00` when `bucket_width` is a number of months or coarser units, else `2000-01-03 00:00:00`. Note that `2000-01-03` is a Monday.
`try_strptime(text, format-list)`	Converts the string `text` to timestamp applying the format strings in the list until one succeeds. Returns `NULL` on failure.
`try_strptime(text, format)`	Converts the string `text` to timestamp according to the format string. Returns `NULL` on failure.

There are also dedicated extraction functions to get the subfields.

Functions applied to infinite dates will either return the same infinite dates (e.g., `greatest`) or `NULL` (e.g., `date_part`) depending on what "makes sense". In general, if the function needs to examine the parts of the infinite date, the result will be `NULL`.

age(timestamp, timestamp)

Description	Subtract arguments, resulting in the time difference between the two timestamps.
Example	`age(TIMESTAMP '2001-04-10', TIMESTAMP '1992-09-20')`
Result	`8 years 6 months 20 days`

age(timestamp)

Description	Subtract from current_date.
Example	`age(TIMESTAMP '1992-09-20')`
Result	`29 years 1 month 27 days 12:39:00.844`

century(timestamp)

Description	Extracts the century of a timestamp.
Example	`century(TIMESTAMP '1992-03-22')`
Result	`20`

current_localtimestamp()

Description	Returns the current timestamp with time zone (at the start of the transaction).
Example	`current_localtimestamp()`
Result	`2024-11-30 13:28:48.895`

date_diff(part, startdate, enddate)

Description	The number of partition boundaries between the timestamps.
Example	`date_diff('hour', TIMESTAMP '1992-09-30 23:59:59', TIMESTAMP '1992-10-01 01:58:00')`
Result	`2`

date_part([part, ...], timestamp)

Description	Get the listed subfields as a `struct`. The list must be constant.
Example	`date_part(['year', 'month', 'day'], TIMESTAMP '1992-09-20 20:38:40')`
Result	`{year: 1992, month: 9, day: 20}`

`date_part(part, timestamp)`

Description	Get subfield (equivalent to `extract`).
Example	`date_part('minute', TIMESTAMP '1992-09-20 20:38:40')`
Result	`38`

`date_sub(part, startdate, enddate)`

Description	The number of complete partitions between the timestamps.
Example	`date_sub('hour', TIMESTAMP '1992-09-30 23:59:59', TIMESTAMP '1992-10-01 01:58:00')`
Result	`1`

`date_trunc(part, timestamp)`

Description	Truncate to specified precision.
Example	`date_trunc('hour', TIMESTAMP '1992-09-20 20:38:40')`
Result	`1992-09-20 20:00:00`

`datediff(part, startdate, enddate)`

Description	Alias of `date_diff`. The number of partition boundaries between the timestamps.
Example	`datediff('hour', TIMESTAMP '1992-09-30 23:59:59', TIMESTAMP '1992-10-01 01:58:00')`
Result	`2`

`datepart([part, ...], timestamp)`

Description	Alias of `date_part`. Get the listed subfields as a `struct`. The list must be constant.
Example	`datepart(['year', 'month', 'day'], TIMESTAMP '1992-09-20 20:38:40')`
Result	`{year: 1992, month: 9, day: 20}`

`datepart(part, timestamp)`

Description	Alias of `date_part`. Get subfield (equivalent to `extract`).
Example	`datepart('minute', TIMESTAMP '1992-09-20 20:38:40')`
Result	`38`

`datesub(part, startdate, enddate)`

Description	Alias of `date_sub`. The number of complete partitions between the timestamps.
Example	`datesub('hour', TIMESTAMP '1992-09-30 23:59:59', TIMESTAMP '1992-10-01 01:58:00')`
Result	`1`

`datetrunc(part, timestamp)`

Description	Alias of `date_trunc`. Truncate to specified precision.
Example	`datetrunc('hour', TIMESTAMP '1992-09-20 20:38:40')`
Result	`1992-09-20 20:00:00`

`dayname(timestamp)`

Description	The (English) name of the weekday.
Example	`dayname(TIMESTAMP '1992-03-22')`
Result	Sunday

`epoch_ms(ms)`

Description	Converts milliseconds since the epoch to a timestamp.
Example	`epoch_ms(701222400000)`
Result	`1992-03-22 00:00:00`

`epoch_ms(timestamp)`

Description	Returns the total number of milliseconds since the epoch.
Example	`epoch_ms(TIMESTAMP '2021-08-03 11:59:44.123456')`
Result	`1627991984123`

`epoch_ns(timestamp)`

Description	Return the total number of nanoseconds since the epoch.
Example	`epoch_ns(TIMESTAMP '2021-08-03 11:59:44.123456')`
Result	`1627991984123456000`

`epoch_us(timestamp)`

Description	Returns the total number of microseconds since the epoch.
Example	`epoch_us(TIMESTAMP '2021-08-03 11:59:44.123456')`
Result	`1627991984123456`

`epoch(timestamp)`

Description	Returns the total number of seconds since the epoch.
Example	`epoch('2022-11-07 08:43:04'::TIMESTAMP);`
Result	`1667810584`

`extract(field FROM timestamp)`

Description	Get subfield from a timestamp.
Example	`extract('hour' FROM TIMESTAMP '1992-09-20 20:38:48')`
Result	`20`

`greatest(timestamp, timestamp)`

Description	The later of two timestamps.
Example	`greatest(TIMESTAMP '1992-09-20 20:38:48', TIMESTAMP '1992-03-22 01:02:03.1234')`
Result	`1992-09-20 20:38:48`

`isfinite(timestamp)`

Description	Returns true if the timestamp is finite, false otherwise.
Example	`isfinite(TIMESTAMP '1992-03-07')`
Result	`true`

`isinf(timestamp)`

Description	Returns true if the timestamp is infinite, false otherwise.
Example	`isinf(TIMESTAMP '-infinity')`
Result	`true`

julian(timestamp)

Description	Extract the Julian Day number from a timestamp.
Example	`julian(TIMESTAMP '1992-03-22 01:02:03.1234')`
Result	`2448704.043091706`

last_day(timestamp)

Description	The last day of the month.
Example	`last_day(TIMESTAMP '1992-03-22 01:02:03.1234')`
Result	`1992-03-31`

least(timestamp, timestamp)

Description	The earlier of two timestamps.
Example	`least(TIMESTAMP '1992-09-20 20:38:48', TIMESTAMP '1992-03-22 01:02:03.1234')`
Result	`1992-03-22 01:02:03.1234`

make_timestamp(bigint, bigint, bigint, bigint, bigint, double)

Description	The timestamp for the given parts.
Example	`make_timestamp(1992, 9, 20, 13, 34, 27.123456)`
Result	`1992-09-20 13:34:27.123456`

make_timestamp(microseconds)

Description	Converts microseconds since the epoch to a timestamp.
Example	`make_timestamp(1667810584123456)`
Result	`2022-11-07 08:43:04.123456`

make_timestamp_ns(nanoseconds)

Description	Converts nanoseconds since the epoch to a timestamp.
Example	`make_timestamp_ns(1667810584123456789)`
Result	`2022-11-07 08:43:04.123456789`

`monthname(timestamp)`

Description	The (English) name of the month.
Example	`monthname(TIMESTAMP '1992-09-20')`
Result	`September`

`strftime(timestamp, format)`

Description	Converts timestamp to string according to the format string.
Example	`strftime(timestamp '1992-01-01 20:38:40', '%a, %-d %B %Y - %I:%M:%S %p')`
Result	`Wed, 1 January 1992 - 08:38:40 PM`

`strptime(text, format-list)`

Description	Converts the string `text` to timestamp applying the format strings in the list until one succeeds. Throws an error on failure. To return `NULL` on failure, use `try_strptime`.
Example	`strptime('4/15/2023 10:56:00', ['%d/%m/%Y %H:%M:%S', '%m/%d/%Y %H:%M:%S'])`
Result	`2023-04-15 10:56:00`

`strptime(text, format)`

Description	Converts the string `text` to timestamp according to the format string. Throws an error on failure. To return `NULL` on failure, use `try_strptime`.
Example	`strptime('Wed, 1 January 1992 - 08:38:40 PM', '%a, %-d %B %Y - %I:%M:%S %p')`
Result	`1992-01-01 20:38:40`

`time_bucket(bucket_width, timestamp[, offset])`

Description	Truncate `timestamp` to a grid of width `bucket_width`. The grid includes `2000-01-01 00:00:00[ + offset]` when `bucket_width` is a number of months or coarser units, else `2000-01-03 00:00:00[ + offset]`. Note that `2000-01-03` is a Monday.
Example	`time_bucket(INTERVAL '10 minutes', TIMESTAMP '1992-04-20 15:26:00-07', INTERVAL '5 minutes')`
Result	`1992-04-20 15:25:00`

`time_bucket(bucket_width, timestamp[, origin])`

Description	Truncate `timestamp` to a grid of width `bucket_width`. The grid includes the `origin` timestamp, which defaults to `2000-01-01 00:00:00` when `bucket_width` is a number of months or coarser units, else `2000-01-03 00:00:00`. Note that `2000-01-03` is a Monday.
Example	`time_bucket(INTERVAL '2 weeks', TIMESTAMP '1992-04-20 15:26:00', TIMESTAMP '1992-04-01 00:00:00')`
Result	`1992-04-15 00:00:00`

`try_strptime(text, format-list)`

Description	Converts the string `text` to timestamp applying the format strings in the list until one succeeds. Returns `NULL` on failure.
Example	`try_strptime('4/15/2023 10:56:00', ['%d/%m/%Y %H:%M:%S', '%m/%d/%Y %H:%M:%S'])`
Result	`2023-04-15 10:56:00`

`try_strptime(text, format)`

Description	Converts the string `text` to timestamp according to the format string. Returns `NULL` on failure.
Example	`try_strptime('Wed, 1 January 1992 - 08:38:40 PM', '%a, %-d %B %Y - %I:%M:%S %p')`
Result	`1992-01-01 20:38:40`

Timestamp Table Functions

The table below shows the available table functions for `TIMESTAMP` types.

Name	Description
`generate_series(timestamp, timestamp, interval)`	Generate a table of timestamps in the closed range, stepping by the interval.
`range(timestamp, timestamp, interval)`	Generate a table of timestamps in the half open range, stepping by the interval.

> Infinite values are not allowed as table function bounds.

`generate_series(timestamp, timestamp, interval)`

Description	Generate a table of timestamps in the closed range, stepping by the interval.
Example	`generate_series(TIMESTAMP '2001-04-10', TIMESTAMP '2001-04-11', INTERVAL 30 MINUTE)`

`range(timestamp, timestamp, interval)`

Description	Generate a table of timestamps in the half open range, stepping by the interval.
Example	`range(TIMESTAMP '2001-04-10', TIMESTAMP '2001-04-11', INTERVAL 30 MINUTE)`

Timestamp with Time Zone Functions

This section describes functions and operators for examining and manipulating `TIMESTAMP WITH TIME ZONE` (or `TIMESTAMPTZ`) values. See also the related `TIMESTAMP` functions.

Time zone support is provided by the built-in ICU extension.

In the examples below, the current time zone is presumed to be `America/Los_Angeles` using the Gregorian calendar.

Built-In Timestamp with Time Zone Functions

The table below shows the available scalar functions for `TIMESTAMPTZ` values. Since these functions do not involve binning or display, they are always available.

Name	Description
`current_timestamp`	Current date and time (start of current transaction).
`get_current_timestamp()`	Current date and time (start of current transaction).
`greatest(timestamptz, timestamptz)`	The later of two timestamps.
`isfinite(timestamptz)`	Returns true if the timestamp with time zone is finite, false otherwise.
`isinf(timestamptz)`	Returns true if the timestamp with time zone is infinite, false otherwise.
`least(timestamptz, timestamptz)`	The earlier of two timestamps.
`now()`	Current date and time (start of current transaction).
`timetz_byte_comparable(timetz)`	Converts a `TIME WITH TIME ZONE` to a `UBIGINT` sort key.
`to_timestamp(double)`	Converts seconds since the epoch to a timestamp with time zone.
`transaction_timestamp()`	Current date and time (start of current transaction).

`current_timestamp`

Description	Current date and time (start of current transaction).
Example	`current_timestamp`
Result	`2022-10-08 12:44:46.122-07`

`get_current_timestamp()`

Description	Current date and time (start of current transaction).
Example	`get_current_timestamp()`
Result	`2022-10-08 12:44:46.122-07`

`greatest(timestamptz, timestamptz)`

Description	The later of two timestamps.
Example	`greatest(TIMESTAMPTZ '1992-09-20 20:38:48', TIMESTAMPTZ '1992-03-22 01:02:03.1234')`
Result	`1992-09-20 20:38:48-07`

`isfinite(timestamptz)`

Description	Returns true if the timestamp with time zone is finite, false otherwise.
Example	`isfinite(TIMESTAMPTZ '1992-03-07')`
Result	`true`

`isinf(timestamptz)`

Description	Returns true if the timestamp with time zone is infinite, false otherwise.
Example	`isinf(TIMESTAMPTZ '-infinity')`
Result	`true`

`least(timestamptz, timestamptz)`

Description	The earlier of two timestamps.
Example	`least(TIMESTAMPTZ '1992-09-20 20:38:48', TIMESTAMPTZ '1992-03-22 01:02:03.1234')`
Result	`1992-03-22 01:02:03.1234-08`

`now()`

Description	Current date and time (start of current transaction).
Example	`now()`
Result	`2022-10-08 12:44:46.122-07`

`timetz_byte_comparable(timetz)`

Description	Converts a `TIME WITH TIME ZONE` to a `UBIGINT` sort key.
Example	`timetz_byte_comparable('18:18:16.21-07:00'::TIMETZ)`
Result	`2494691656335442799`

`to_timestamp(double)`

Description	Converts seconds since the epoch to a timestamp with time zone.
Example	`to_timestamp(1284352323.5)`
Result	`2010-09-13 04:32:03.5+00`

`transaction_timestamp()`

Description	Current date and time (start of current transaction).
Example	`transaction_timestamp()`
Result	`2022-10-08 12:44:46.122-07`

Timestamp with Time Zone Strings

With no time zone extension loaded, `TIMESTAMPTZ` values will be cast to and from strings using offset notation. This will let you specify an instant correctly without access to time zone information. For portability, `TIMESTAMPTZ` values will always be displayed using GMT offsets:

```
SELECT '2022-10-08 13:13:34-07'::TIMESTAMPTZ;
```

```
2022-10-08 20:13:34+00
```

If a time zone extension such as ICU is loaded, then a time zone can be parsed from a string and cast to a representation in the local time zone:

```
SELECT '2022-10-08 13:13:34 Europe/Amsterdam'::TIMESTAMPTZ::VARCHAR;
```

```
2022-10-08 04:13:34-07 -- the offset will differ based on your local time zone
```

ICU Timestamp with Time Zone Operators

The table below shows the available mathematical operators for `TIMESTAMP WITH TIME ZONE` values provided by the ICU extension.

Operator	Description	Example	Result
`+`	addition of an `INTERVAL`	`TIMESTAMPTZ '1992-03-22 01:02:03' + INTERVAL 5 DAY`	`1992-03-27 01:02:03`
`-`	subtraction of `TIMESTAMPTZ`s	`TIMESTAMPTZ '1992-03-27' - TIMESTAMPTZ '1992-03-22'`	`5 days`

Operator	Description	Example	Result
-	subtraction of an INTERVAL	TIMESTAMPTZ '1992-03-27 01:02:03' - INTERVAL 5 DAY	1992-03-22 01:02:03

Adding to or subtracting from infinite values produces the same infinite value.

ICU Timestamp with Time Zone Functions

The table below shows the ICU provided scalar functions for TIMESTAMP WITH TIME ZONE values.

Name	Description
age(timestamptz, timestamptz)	Subtract arguments, resulting in the time difference between the two timestamps.
age(timestamptz)	Subtract from current_date.
date_diff(part, startdate, enddate)	The number of partition boundaries between the timestamps.
date_part([part, ...], timestamptz)	Get the listed subfields as a struct. The list must be constant.
date_part(part, timestamptz)	Get subfield (equivalent to *extract*).
date_sub(part, startdate, enddate)	The number of complete partitions between the timestamps.
date_trunc(part, timestamptz)	Truncate to specified precision.
datediff(part, startdate, enddate)	Alias of date_diff. The number of partition boundaries between the timestamps.
datepart([part, ...], timestamptz)	Alias of date_part. Get the listed subfields as a struct. The list must be constant.
datepart(part, timestamptz)	Alias of date_part. Get subfield (equivalent to *extract*).
datesub(part, startdate, enddate)	Alias of date_sub. The number of complete partitions between the timestamps.
datetrunc(part, timestamptz)	Alias of date_trunc. Truncate to specified precision.
epoch_ms(timestamptz)	Converts a timestamptz to milliseconds since the epoch.
epoch_ns(timestamptz)	Converts a timestamptz to nanoseconds since the epoch.
epoch_us(timestamptz)	Converts a timestamptz to microseconds since the epoch.
extract(field FROM timestamptz)	Get subfield from a TIMESTAMP WITH TIME ZONE.
last_day(timestamptz)	The last day of the month.
make_timestamptz(bigint, bigint, bigint, bigint, bigint, double, string)	The TIMESTAMP WITH TIME ZONE for the given parts and time zone.

Name	Description
`make_timestamptz(bigint, bigint, bigint, bigint, bigint, double)`	The `TIMESTAMP WITH TIME ZONE` for the given parts in the current time zone.
`make_timestamptz(microseconds)`	The `TIMESTAMP WITH TIME ZONE` for the given µs since the epoch.
`strftime(timestamptz, format)`	Converts a `TIMESTAMP WITH TIME ZONE` value to string according to the format string.
`strptime(text, format)`	Converts string to `TIMESTAMP WITH TIME ZONE` according to the format string if `%Z` is specified.
`time_bucket(bucket_width, timestamptz[, offset])`	Truncate `timestamptz` to a grid of width `bucket_width`. The grid is anchored at `2000-01-01 00:00:00+00:00[ + offset]` when `bucket_width` is a number of months or coarser units, else `2000-01-03 00:00:00+00:00[ + offset]`. Note that `2000-01-03` is a Monday.
`time_bucket(bucket_width, timestamptz[, origin])`	Truncate `timestamptz` to a grid of width `bucket_width`. The grid is anchored at the `origin` timestamp, which defaults to `2000-01-01 00:00:00+00:00` when `bucket_width` is a number of months or coarser units, else `2000-01-03 00:00:00+00:00`. Note that `2000-01-03` is a Monday.
`time_bucket(bucket_width, timestamptz[, timezone])`	Truncate `timestamptz` to a grid of width `bucket_width`. The grid is anchored at the `origin` timestamp, which defaults to `2000-01-01 00:00:00` in the provided `timezone` when `bucket_width` is a number of months or coarser units, else `2000-01-03 00:00:00` in the provided `timezone`. The default timezone is `'UTC'`. Note that `2000-01-03` is a Monday.

`age(timestamptz, timestamptz)`

Description	Subtract arguments, resulting in the time difference between the two timestamps.
Example	`age(TIMESTAMPTZ '2001-04-10', TIMESTAMPTZ '1992-09-20')`
Result	`8 years 6 months 20 days`

`age(timestamptz)`

Description	Subtract from current_date.
Example	`age(TIMESTAMP '1992-09-20')`
Result	`29 years 1 month 27 days 12:39:00.844`

`date_diff(part, startdate, enddate)`

Description	The number of partition boundaries between the timestamps.
Example	`date_diff('hour', TIMESTAMPTZ '1992-09-30 23:59:59', TIMESTAMPTZ '1992-10-01 01:58:00')`
Result	2

`date_part([part, ...], timestamptz)`

Description	Get the listed subfields as a `struct`. The list must be constant.
Example	`date_part(['year', 'month', 'day'], TIMESTAMPTZ '1992-09-20 20:38:40-07')`
Result	`{year: 1992, month: 9, day: 20}`

`date_part(part, timestamptz)`

Description	Get subfield (equivalent to *extract*).
Example	`date_part('minute', TIMESTAMPTZ '1992-09-20 20:38:40')`
Result	`38`

`date_sub(part, startdate, enddate)`

Description	The number of complete partitions between the timestamps.
Example	`date_sub('hour', TIMESTAMPTZ '1992-09-30 23:59:59', TIMESTAMPTZ '1992-10-01 01:58:00')`
Result	`1`

`date_trunc(part, timestamptz)`

Description	Truncate to specified precision.
Example	`date_trunc('hour', TIMESTAMPTZ '1992-09-20 20:38:40')`
Result	`1992-09-20 20:00:00`

`datediff(part, startdate, enddate)`

Description	Alias of date_diff. The number of partition boundaries between the timestamps.
Example	`datediff('hour', TIMESTAMPTZ '1992-09-30 23:59:59', TIMESTAMPTZ '1992-10-01 01:58:00')`
Result	`2`

`datepart([part, ...], timestamptz)`

Description	Alias of date_part. Get the listed subfields as a `struct`. The list must be constant.
Example	`datepart(['year', 'month', 'day'], TIMESTAMPTZ '1992-09-20 20:38:40-07')`
Result	`{year: 1992, month: 9, day: 20}`

`datepart(part, timestamptz)`

Description	Alias of date_part. Get subfield (equivalent to *extract*).
Example	`datepart('minute', TIMESTAMPTZ '1992-09-20 20:38:40')`
Result	`38`

`datesub(part, startdate, enddate)`

Description	Alias of date_sub. The number of complete partitions between the timestamps.
Example	`datesub('hour', TIMESTAMPTZ '1992-09-30 23:59:59', TIMESTAMPTZ '1992-10-01 01:58:00')`
Result	`1`

`datetrunc(part, timestamptz)`

Description	Alias of date_trunc. Truncate to specified precision.
Example	`datetrunc('hour', TIMESTAMPTZ '1992-09-20 20:38:40')`
Result	`1992-09-20 20:00:00`

`epoch_ms(timestamptz)`

Description	Converts a timestamptz to milliseconds since the epoch.
Example	`epoch_ms('2022-11-07 08:43:04.123456+00'::TIMESTAMPTZ);`
Result	`1667810584123`

`epoch_ns(timestamptz)`

Description	Converts a timestamptz to nanoseconds since the epoch.
Example	`epoch_ns('2022-11-07 08:43:04.123456+00'::TIMESTAMPTZ);`
Result	`1667810584123456000`

`epoch_us(timestamptz)`

Description	Converts a timestamptz to microseconds since the epoch.
Example	`epoch_us('2022-11-07 08:43:04.123456+00'::TIMESTAMPTZ);`
Result	`1667810584123456`

`extract(field FROM timestamptz)`

Description	Get subfield from a `TIMESTAMP WITH TIME ZONE`.
Example	`extract('hour' FROM TIMESTAMPTZ '1992-09-20 20:38:48')`
Result	`20`

`last_day(timestamptz)`

Description	The last day of the month.
Example	`last_day(TIMESTAMPTZ '1992-03-22 01:02:03.1234')`
Result	`1992-03-31`

`make_timestamptz(bigint, bigint, bigint, bigint, bigint, double, string)`

Description	The `TIMESTAMP WITH TIME ZONE` for the given parts and time zone.
Example	`make_timestamptz(1992, 9, 20, 15, 34, 27.123456, 'CET')`
Result	`1992-09-20 06:34:27.123456-07`

`make_timestamptz(bigint, bigint, bigint, bigint, bigint, double)`

Description	The `TIMESTAMP WITH TIME ZONE` for the given parts in the current time zone.
Example	`make_timestamptz(1992, 9, 20, 13, 34, 27.123456)`
Result	`1992-09-20 13:34:27.123456-07`

`make_timestamptz(microseconds)`

Description	The `TIMESTAMP WITH TIME ZONE` for the given µs since the epoch.
Example	`make_timestamptz(1667810584123456)`
Result	`2022-11-07 16:43:04.123456-08`

`strftime(timestamptz, format)`

Description	Converts a `TIMESTAMP WITH TIME ZONE` value to string according to the format string.
Example	`strftime(timestamptz '1992-01-01 20:38:40', '%a, %-d %B %Y - %I:%M:%S %p')`
Result	`Wed, 1 January 1992 - 08:38:40 PM`

`strptime(text, format)`

Description	Converts string to `TIMESTAMP WITH TIME ZONE` according to the format string if `%Z` is specified.
Example	`strptime('Wed, 1 January 1992 - 08:38:40 PST', '%a, %-d %B %Y - %H:%M:%S %Z')`
Result	`1992-01-01 08:38:40-08`

`time_bucket(bucket_width, timestamptz[, offset])`

Description	Truncate `timestamptz` to a grid of width `bucket_width`. The grid is anchored at `2000-01-01 00:00:00+00:00[ + offset]` when `bucket_width` is a number of months or coarser units, else `2000-01-03 00:00:00+00:00[ + offset]`. Note that `2000-01-03` is a Monday.
Example	`time_bucket(INTERVAL '10 minutes', TIMESTAMPTZ '1992-04-20 15:26:00-07', INTERVAL '5 minutes')`
Result	`1992-04-20 15:25:00-07`

`time_bucket(bucket_width, timestamptz[, origin])`

Description	Truncate `timestamptz` to a grid of width `bucket_width`. The grid is anchored at the `origin` timestamp, which defaults to `2000-01-01 00:00:00+00:00` when `bucket_width` is a number of months or coarser units, else `2000-01-03 00:00:00+00:00`. Note that `2000-01-03` is a Monday.
Example	`time_bucket(INTERVAL '2 weeks', TIMESTAMPTZ '1992-04-20 15:26:00-07', TIMESTAMPTZ '1992-04-01 00:00:00-07')`
Result	`1992-04-15 00:00:00-07`

`time_bucket(bucket_width, timestamptz[, timezone])`

Description	Truncate `timestamptz` to a grid of width `bucket_width`. The grid is anchored at the `origin` timestamp, which defaults to `2000-01-01 00:00:00` in the provided `timezone` when `bucket_width` is a number of months or coarser units, else `2000-01-03 00:00:00` in the provided `timezone`. The default timezone is `'UTC'`. Note that `2000-01-03` is a Monday.
Example	`time_bucket(INTERVAL '2 days', TIMESTAMPTZ '1992-04-20 15:26:00-07', 'Europe/Berlin')`
Result	`1992-04-19 15:00:00-07` (=`1992-04-20 00:00:00 Europe/Berlin`)

There are also dedicated extraction functions to get the subfields.

ICU Timestamp Table Functions

The table below shows the available table functions for `TIMESTAMP WITH TIME ZONE` types.

Name	Description
`generate_series(timestamptz, timestamptz, interval)`	Generate a table of timestamps in the closed range (including both the starting timestamp and the ending timestamp), stepping by the interval.
`range(timestamptz, timestamptz, interval)`	Generate a table of timestamps in the half open range (including the starting timestamp, but stopping before the ending timestamp), stepping by the interval.

> Infinite values are not allowed as table function bounds.

`generate_series(timestamptz, timestamptz, interval)`

Description	Generate a table of timestamps in the closed range (including both the starting timestamp and the ending timestamp), stepping by the interval.
Example	`generate_series(TIMESTAMPTZ '2001-04-10', TIMESTAMPTZ '2001-04-11', INTERVAL 30 MINUTE)`

`range(timestamptz, timestamptz, interval)`

Description	Generate a table of timestamps in the half open range (including the starting timestamp, but stopping before the ending timestamp), stepping by the interval.
Example	`range(TIMESTAMPTZ '2001-04-10', TIMESTAMPTZ '2001-04-11', INTERVAL 30 MINUTE)`

ICU Timestamp Without Time Zone Functions

The table below shows the ICU provided scalar functions that operate on plain `TIMESTAMP` values. These functions assume that the `TIMESTAMP` is a "local timestamp".

A local timestamp is effectively a way of encoding the part values from a time zone into a single value. They should be used with caution because the produced values can contain gaps and ambiguities thanks to daylight savings time. Often the same functionality can be implemented more reliably using the `struct` variant of the `date_part` function.

Name	Description
`current_localtime()`	Returns a `TIME` whose GMT bin values correspond to local time in the current time zone.
`current_localtimestamp()`	Returns a `TIMESTAMP` whose GMT bin values correspond to local date and time in the current time zone.
`localtime`	Synonym for the `current_localtime()` function call.
`localtimestamp`	Synonym for the `current_localtimestamp()` function call.
`timezone(text, timestamp)`	Use the date parts of the timestamp in GMT to construct a timestamp in the given time zone. Effectively, the argument is a "local" time.
`timezone(text, timestamptz)`	Use the date parts of the timestamp in the given time zone to construct a timestamp. Effectively, the result is a "local" time.

`current_localtime()`

Description	Returns a `TIME` whose GMT bin values correspond to local time in the current time zone.
Example	`current_localtime()`
Result	`08:47:56.497`

`current_localtimestamp()`

Description	Returns a `TIMESTAMP` whose GMT bin values correspond to local date and time in the current time zone.
Example	`current_localtimestamp()`
Result	`2022-12-17 08:47:56.497`

`localtime`

Description	Synonym for the `current_localtime()` function call.
Example	`localtime`
Result	`08:47:56.497`

`localtimestamp`

Description	Synonym for the `current_localtimestamp()` function call.
Example	`localtimestamp`
Result	`2022-12-17 08:47:56.497`

`timezone(text, timestamp)`

Description	Use the date parts of the timestamp in GMT to construct a timestamp in the given time zone. Effectively, the argument is a "local" time.
Example	`timezone('America/Denver', TIMESTAMP '2001-02-16 20:38:40')`
Result	`2001-02-16 19:38:40-08`

`timezone(text, timestamptz)`

Description	Use the date parts of the timestamp in the given time zone to construct a timestamp. Effectively, the result is a "local" time.
Example	`timezone('America/Denver', TIMESTAMPTZ '2001-02-16 20:38:40-05')`
Result	`2001-02-16 18:38:40`

At Time Zone

The `AT TIME ZONE` syntax is syntactic sugar for the (two argument) `timezone` function listed above:

```
SELECT TIMESTAMP '2001-02-16 20:38:40' AT TIME ZONE 'America/Denver' AS ts;
```

```
2001-02-16 19:38:40-08
```

```
SELECT TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40-05' AT TIME ZONE 'America/Denver' AS ts;
```

```
2001-02-16 18:38:40
```

Note that numeric timezones are not allowed:

```
SELECT TIMESTAMP '2001-02-16 20:38:40-05' AT TIME ZONE '0200' AS ts;
```

```
Not implemented Error: Unknown TimeZone '0200'
```

Infinities

Functions applied to infinite dates will either return the same infinite dates (e.g., `greatest`) or `NULL` (e.g., `date_part`) depending on what "makes sense". In general, if the function needs to examine the parts of the infinite temporal value, the result will be `NULL`.

Calendars

The ICU extension also supports non-Gregorian calendars. If such a calendar is current, then the display and binning operations will use that calendar.

Union Functions

Name	Description
`union.tag`	Dot notation serves as an alias for `union_extract`.
`union_extract(union, 'tag')`	Extract the value with the named tags from the union. `NULL` if the tag is not currently selected.
`union_value(tag := any)`	Create a single member `UNION` containing the argument value. The tag of the value will be the bound variable name.
`union_tag(union)`	Retrieve the currently selected tag of the union as an Enum.

`union.tag`

Description	Dot notation serves as an alias for `union_extract`.
Example	`(union_value(k := 'hello')).k`
Result	`string`

`union_extract(union, 'tag')`

Description	Extract the value with the named tags from the union. `NULL` if the tag is not currently selected.
Example	`union_extract(s, 'k')`
Result	`hello`

`union_value(tag := any)`

Description	Create a single member `UNION` containing the argument value. The tag of the value will be the bound variable name.
Example	`union_value(k := 'hello')`
Result	`'hello'::UNION(k VARCHAR)`

`union_tag(union)`

Description	Retrieve the currently selected tag of the union as an Enum.
Example	`union_tag(union_value(k := 'foo'))`
Result	`'k'`

Utility Functions

Scalar Utility Functions

The functions below are difficult to categorize into specific function types and are broadly useful.

Name	Description
`alias(column)`	Return the name of the column.
`can_cast_implicitly(source_value, target_value)`	Whether or not we can implicitly cast from the types of the source value to the target value.
`checkpoint(database)`	Synchronize WAL with file for (optional) database without interrupting transactions.
`coalesce(expr, ...)`	Return the first expression that evaluates to a non-`NULL` value. Accepts 1 or more parameters. Each expression can be a column, literal value, function result, or many others.
`constant_or_null(arg1, arg2)`	If `arg2` is `NULL`, return `NULL`. Otherwise, return `arg1`.
`count_if(x)`	Aggregate function; rows contribute 1 if `x` is `true` or a non-zero number, else 0.
`create_sort_key(parameters...)`	Constructs a binary-comparable sort key based on a set of input parameters and sort qualifiers.
`current_catalog()`	Return the name of the currently active catalog. Default is memory.
`current_database()`	Return the name of the currently active database.
`current_query()`	Return the current query as a string.
`current_schema()`	Return the name of the currently active schema. Default is main.

Name	Description
`current_schemas(boolean)`	Return list of schemas. Pass a parameter of `true` to include implicit schemas.
`current_setting('setting_name')`	Return the current value of the configuration setting.
`currval('sequence_name')`	Return the current value of the sequence. Note that `nextval` must be called at least once prior to calling `currval`.
`error(message)`	Throws the given error `message`.
`equi_width_bins(min, max, bincount, nice := false)`	Returns the upper boundaries of a partition of the interval `[min, max]` into `bin_count` equal-sized subintervals (for use with, e.g., `histogram`). If `nice = true`, then `min`, `max`, and `bincount` may be adjusted to produce more aesthetically pleasing results.
`force_checkpoint(database)`	Synchronize WAL with file for (optional) database interrupting transactions.
`gen_random_uuid()`	Return a random UUID similar to this: `eeccb8c5-9943-b2bb-bb5e-222f4e14b687`.
`getenv(var)`	Returns the value of the environment variable `var`. Only available in the command line client.
`hash(value)`	Returns a `UBIGINT` with the hash of the `value`.
`icu_sort_key(string, collator)`	Surrogate sort key used to sort special characters according to the specific locale. Collator parameter is optional. Only available when the ICU extension is installed.
`if(a, b, c)`	Ternary conditional operator.
`ifnull(expr, other)`	A two-argument version of coalesce.
`is_histogram_other_bin(arg)`	Returns `true` when `arg` is the ”catch-all element” of its datatype for the purpose of the `histogram_exact` function, which is equal to the ”right-most boundary” of its datatype for the purpose of the `histogram` function.
`md5(string)`	Returns the MD5 hash of the `string` as a `VARCHAR`.
`md5_number(string)`	Returns the MD5 hash of the `string` as a `UHUGEINT`.
`md5_number_lower(string)`	Returns the lower 64-bit segment of the MD5 hash of the `string` as a `UBIGINT`.
`md5_number_upper(string)`	Returns the upper 64-bit segment of the MD5 hash of the `string` as a `UBIGINT`.
`nextval('sequence_name')`	Return the following value of the sequence.
`nullif(a, b)`	Return `NULL` if `a = b`, else return `a`. Equivalent to `CASE WHEN a = b THEN NULL ELSE a END`.
`pg_typeof(expression)`	Returns the lower case name of the data type of the result of the expression. For PostgreSQL compatibility.
`query(query_string_literal)`	Table function that parses and executes the query defined in *`query_string_literal`*. Only literal strings are allowed. Warning: this function allows invoking arbitrary queries, potentially altering the database state.
`query_table(tbl_name)`	Table function that returns the table given in *`tbl_name`*.
`query_table(tbl_names, [by_name])`	Table function that returns the union of tables given in *`tbl_names`*. If the optional *`by_name`* parameter is set to `true`, it uses `UNION ALL BY NAME` semantics.
`read_blob(source)`	Returns the content from `source` (a filename, a list of filenames, or a glob pattern) as a `BLOB`. See the `read_blob` guide for more details.
`read_text(source)`	Returns the content from `source` (a filename, a list of filenames, or a glob pattern) as a `VARCHAR`. The file content is first validated to be valid UTF-8. If `read_text` attempts to read a file with invalid UTF-8 an error is thrown suggesting to use `read_blob` instead. See the `read_text` guide for more details.
`sha1(string)`	Returns a `VARCHAR` with the SHA-1 hash of the `string`.

Name	Description
`sha256(string)`	Returns a VARCHAR with the SHA-256 hash of the `string`.
`stats(expression)`	Returns a string with statistics about the expression. Expression can be a column, constant, or SQL expression.
`txid_current()`	Returns the current transaction's identifier, a `BIGINT` value. It will assign a new one if the current transaction does not have one already.
`typeof(expression)`	Returns the name of the data type of the result of the expression.
`uuid()`	Return a random UUID (UUIDv4) similar to this: `eeccb8c5-9943-b2bb-bb5e-222f4e14b687`.
`uuidv4()`	Return a random UUID (UUIDv4) similar to this: `eeccb8c5-9943-b2bb-bb5e-222f4e14b687`.
`uuidv7()`	Return a random UUIDv7 similar to this: `81964ebe-00b1-7e1d-b0f9-43c29b6fb8f5`.
`uuid_extract_timestamp(uuidv7)`	Extracts timestamp from a UUIDv7 value.
`uuid_extract_version(uuid)`	Extracts UUID version (4 or 7).
`version()`	Return the currently active version of DuckDB in this format.

`alias(column)`

Description	Return the name of the column.
Example	`alias(column1)`
Result	`column1`

`can_cast_implicitly(source_value, target_value)`

Description	Whether or not we can implicitly cast from the types of the source value to the target value.
Example	`can_cast_implicitly(1::BIGINT, 1::SMALLINT)`
Result	`false`

`checkpoint(database)`

Description	Synchronize WAL with file for (optional) database without interrupting transactions.
Example	`checkpoint(my_db)`
Result	success Boolean

`coalesce(expr, ...)`

Description	Return the first expression that evaluates to a non-NULL value. Accepts 1 or more parameters. Each expression can be a column, literal value, function result, or many others.
Example	`coalesce(NULL, NULL, 'default_string')`
Result	`default_string`

`constant_or_null(arg1, arg2)`

Description	If `arg2` is NULL, return NULL. Otherwise, return `arg1`.
Example	`constant_or_null(42, NULL)`
Result	`NULL`

`count_if(x)`

Description	Aggregate function; rows contribute 1 if `x` is `true` or a non-zero number, else 0.
Example	`count_if(42)`
Result	`1`

`create_sort_key(parameters...)`

Description	Constructs a binary-comparable sort key based on a set of input parameters and sort qualifiers.
Example	`create_sort_key('abc', 'ASC NULLS FIRST');`
Result	`\x02bcd\x00`

`current_catalog()`

Description	Return the name of the currently active catalog. Default is memory.
Example	`current_catalog()`
Result	`memory`

`current_database()`

Description	Return the name of the currently active database.
Example	`current_database()`
Result	`memory`

`current_query()`

Description	Return the current query as a string.
Example	`current_query()`
Result	`SELECT current_query();`

`current_schema()`

Description	Return the name of the currently active schema. Default is main.
Example	`current_schema()`
Result	`main`

`current_schemas(boolean)`

Description	Return list of schemas. Pass a parameter of `true` to include implicit schemas.
Example	`current_schemas(true)`
Result	`['temp', 'main', 'pg_catalog']`

`current_setting('setting_name')`

Description	Return the current value of the configuration setting.
Example	`current_setting('access_mode')`
Result	`automatic`

`currval('sequence_name')`

Description	Return the current value of the sequence. Note that `nextval` must be called at least once prior to calling `currval`.
Example	`currval('my_sequence_name')`
Result	`1`

`error(message)`

Description	Throws the given error `message`.
Example	`error('access_mode')`

`equi_width_bins(min, max, bincount, nice := false)`

Description	Returns the upper boundaries of a partition of the interval `[min, max]` into `bin_count` equal-sized subintervals (for use with, e.g., `histogram`). If `nice = true`, then `min`, `max`, and `bincount` may be adjusted to produce more aesthetically pleasing results.
Example	`equi_width_bins(0.1, 2.7, 4, true)`
Result	`[0.5, 1.0, 1.5, 2.0, 2.5, 3.0]`

`force_checkpoint(database)`

Description	Synchronize WAL with file for (optional) database interrupting transactions.
Example	`force_checkpoint(my_db)`
Result	success Boolean

`gen_random_uuid()`

Description	Return a random UUID (UUIDv4) similar to this: `eeccb8c5-9943-b2bb-bb5e-222f4e14b687`.
Example	`gen_random_uuid()`
Result	various

`getenv(var)`

Description	Returns the value of the environment variable `var`. Only available in the command line client.
Example	`getenv('HOME')`
Result	`/path/to/user/home`

`hash(value)`

Description	Returns a `UBIGINT` with the hash of the `value`.
Example	`hash('🦆')`
Result	`2595805878642663834`

`icu_sort_key(string, collator)`

Description	Surrogate sort key used to sort special characters according to the specific locale. Collator parameter is optional. Only available when the ICU extension is installed.
Example	`icu_sort_key('ö', 'DE')`
Result	`460145960106`

`if(a, b, c)`

Description	Ternary conditional operator; returns b if a, else returns c. Equivalent to `CASE WHEN a THEN b ELSE c END`.
Example	`if(2 > 1, 3, 4)`
Result	`3`

`ifnull(expr, other)`

Description	A two-argument version of coalesce.
Example	`ifnull(NULL, 'default_string')`
Result	`default_string`

`is_histogram_other_bin(arg)`

Description	Returns `true` when `arg` is the ”catch-all element” of its datatype for the purpose of the `histogram_exact` function, which is equal to the ”right-most boundary” of its datatype for the purpose of the `histogram` function.
Example	`is_histogram_other_bin('')`
Result	`true`

`md5(string)`

Description	Returns the MD5 hash of the `string` as a `VARCHAR`.
Example	`md5('abc')`
Result	`900150983cd24fb0d6963f7d28e17f72`

`md5_number(string)`

Description	Returns the MD5 hash of the `string` as a `UHUGEINT`.
Example	`md5_number('abc')`
Result	`152195979970564155685860391459828531600`

`md5_number_lower(string)`

Description	Returns the lower 8 bytes of the MD5 hash of `string` as a `UBIGINT`.
Example	`md5_number_lower('abc')`
Result	`8250560606382298838`

`md5_number_upper(string)`

Description	Returns the upper 8 bytes of the MD5 hash of `string` as a `UBIGINT`.
Example	`md5_number_upper('abc')`
Result	`12704604231530709392`

`nextval('sequence_name')`

Description	Return the following value of the sequence.
Example	`nextval('my_sequence_name')`
Result	2

`nullif(a, b)`

Description	Return `NULL` if a = b, else return a. Equivalent to `CASE WHEN a = b THEN NULL ELSE a END`.
Example	`nullif(1+1, 2)`
Result	`NULL`

`pg_typeof(expression)`

Description	Returns the lower case name of the data type of the result of the expression. For PostgreSQL compatibility.
Example	`pg_typeof('abc')`
Result	`varchar`

`query(query_string_literal)`

Description	Table function that parses and executes the query defined in `query_string_literal`. Only literal strings are allowed. Warning: this function allows invoking arbitrary queries, potentially altering the database state.
Example	`query('SELECT 42 AS x')`
Result	`42`

`query_table(tbl_name)`

Description	Table function that returns the table given in `tbl_name`.
Example	`query_table('t1')`
Result	(the rows of `t1`)

`query_table(tbl_names, [by_name])`

Description	Table function that returns the union of tables given in `tbl_names`. If the optional `by_name` parameter is set to `true`, it uses `UNION ALL BY NAME` semantics.
Example	`query_table(['t1', 't2'])`
Result	(the union of the two tables)

`read_blob(source)`

Description	Returns the content from `source` (a filename, a list of filenames, or a glob pattern) as a BLOB. See the `read_blob` guide for more details.
Example	`read_blob('hello.bin')`
Result	`hello\x0A`

`read_text(source)`

Description	Returns the content from `source` (a filename, a list of filenames, or a glob pattern) as a VARCHAR. The file content is first validated to be valid UTF-8. If `read_text` attempts to read a file with invalid UTF-8 an error is thrown suggesting to use `read_blob` instead. See the `read_text` guide for more details.
Example	`read_text('hello.txt')`
Result	`hello\n`

`sha1(string)`

Description	Returns a VARCHAR with the SHA-1 hash of the `string`.
Example	`sha1('🦆')`
Result	`949bf843dc338be348fb9525d1eb535d31241d76`

`sha256(string)`

Description	Returns a VARCHAR with the SHA-256 hash of the `string`.
Example	`sha256('🦆')`
Result	`d7a5c5e0d1d94c32218539e7e47d4ba9c3c7b77d61332fb60d633dde89e473fb`

`stats(expression)`

Description	Returns a string with statistics about the expression. Expression can be a column, constant, or SQL expression.

Example	`stats(5)`
Result	`'[Min: 5, Max: 5][Has Null: false]'`

`txid_current()`

Description	Returns the current transaction's identifier, a `BIGINT` value. It will assign a new one if the current transaction does not have one already.
Example	`txid_current()`
Result	various

`typeof(expression)`

Description	Returns the name of the data type of the result of the expression.
Example	`typeof('abc')`
Result	VARCHAR

`uuid()`

Description	Return a random UUID (UUIDv4) similar to this: `eeccb8c5-9943-b2bb-bb5e-222f4e14b687`.
Example	`uuid()`
Result	various

`uuidv4()`

Description	Return a random UUID (UUIDv4) similar to this: `eeccb8c5-9943-b2bb-bb5e-222f4e14b687`.
Example	`uuidv4()`
Result	various

`uuidv7()`

Description	Return a random UUIDv7 similar to this: `81964ebe-00b1-7e1d-b0f9-43c29b6fb8f5`.
Example	`uuidv7()`
Result	various

`uuid_extract_timestamp(uuidv7)`

Description	Extracts timestamp from a UUIDv7 value.
Example	`uuid_extract_timestamp(uuidv7())`
Result	`2025-04-19 15:51:20.07+00`

`uuid_extract_version(uuid)`

Description	Extracts UUID version (4 or 7).
Example	`uuid_extract_version(uuidv7())`
Result	7

`version()`

Description	Return the currently active version of DuckDB in this format.
Example	`version()`
Result	various

Utility Table Functions

A table function is used in place of a table in a `FROM` clause.

Name	Description
`glob(search_path)`	Return filenames found at the location indicated by the *search_path* in a single column named `file`. The *search_path* may contain glob pattern matching syntax.
`repeat_row(varargs, num_rows)`	Returns a table with `num_rows` rows, each containing the fields defined in `varargs`.

`glob(search_path)`

Description	Return filenames found at the location indicated by the *search_path* in a single column named `file`. The *search_path* may contain glob pattern matching syntax.
Example	`glob('*')`
Result	(table of filenames)

`repeat_row(varargs, num_rows)`

Description	Returns a table with num_rows rows, each containing the fields defined in varargs.
Example	`repeat_row(1, 2, 'foo', num_rows = 3)`
Result	3 rows of `1, 2, 'foo'`

Window Functions

DuckDB supports window functions, which can use multiple rows to calculate a value for each row. Window functions are blocking operators, i.e., they require their entire input to be buffered, making them one of the most memory-intensive operators in SQL.

Window function are available in SQL since SQL:2003 and are supported by major SQL database systems.

Examples

Generate a `row_number` column to enumerate rows:

```
SELECT row_number() OVER ()
FROM sales;
```

> **Tip.** If you only need a number for each row in a table, you can use the `rowid` pseudocolumn.

Generate a `row_number` column to enumerate rows, ordered by `time`:

```
SELECT row_number() OVER (ORDER BY time)
FROM sales;
```

Generate a `row_number` column to enumerate rows, ordered by `time` and partitioned by `region`:

```
SELECT row_number() OVER (PARTITION BY region ORDER BY time)
FROM sales;
```

Compute the difference between the current and the previous-by-`time` `amount`:

```
SELECT amount - lag(amount) OVER (ORDER BY time)
FROM sales;
```

Compute the percentage of the total `amount` of sales per `region` for each row:

```
SELECT amount / sum(amount) OVER (PARTITION BY region)
FROM sales;
```

Syntax

Window functions can only be used in the `SELECT` clause. To share `OVER` specifications between functions, use the statement's `WINDOW` clause and use the `OVER <window_name>{:.language-sql .highlight}` syntax.

General-Purpose Window Functions

The table below shows the available general window functions.

Name	Description
`cume_dist([ORDER BY ordering])`	The cumulative distribution: (number of partition rows preceding or peer with current row) / total partition rows.
`dense_rank()`	The rank of the current row *without gaps;* this function counts peer groups.
`first_value(expr[ ORDER BY ordering][ IGNORE NULLS])`	Returns `expr` evaluated at the row that is the first row (with a non-null value of `expr` if `IGNORE NULLS` is set) of the window frame.
`lag(expr[, offset[, default]][ ORDER BY ordering][ IGNORE NULLS])`	Returns `expr` evaluated at the row that is `offset` rows (among rows with a non-null value of `expr` if `IGNORE NULLS` is set) before the current row within the window frame; if there is no such row, instead return `default` (which must be of the Same type as `expr`). Both `offset` and `default` are evaluated with respect to the current row. If omitted, `offset` defaults to `1` and default to `NULL`.
`last_value(expr[ ORDER BY ordering][ IGNORE NULLS])`	Returns `expr` evaluated at the row that is the last row (among rows with a non-null value of `expr` if `IGNORE NULLS` is set) of the window frame.
`lead(expr[, offset[, default]][ ORDER BY ordering][ IGNORE NULLS])`	Returns `expr` evaluated at the row that is `offset` rows after the current row (among rows with a non-null value of `expr` if `IGNORE NULLS` is set) within the window frame; if there is no such row, instead return `default` (which must be of the Same type as `expr`). Both `offset` and `default` are evaluated with respect to the current row. If omitted, `offset` defaults to `1` and default to `NULL`.
`nth_value(expr, nth[ ORDER BY ordering][ IGNORE NULLS])`	Returns `expr` evaluated at the nth row (among rows with a non-null value of `expr` if `IGNORE NULLS` is set) of the window frame (counting from 1); `NULL` if no such row.
`ntile(num_buckets[ ORDER BY ordering])`	An integer ranging from 1 to `num_buckets`, dividing the partition as equally as possible.
`percent_rank([ORDER BY ordering])`	The relative rank of the current row: `(rank() - 1) / (total partition rows - 1)`.
`rank_dense()`	The rank of the current row *without gaps*.
`rank([ORDER BY ordering])`	The rank of the current row *with gaps;* same as `row_number` of its first peer.
`row_number([ORDER BY ordering])`	The number of the current row within the partition, counting from 1.

`cume_dist([ORDER BY ordering])`

Description	The cumulative distribution: (number of partition rows preceding or peer with current row) / total partition rows. If an `ORDER BY` clause is specified, the distribution is computed within the frame using the provided ordering instead of the frame ordering.
Return type	`DOUBLE`
Example	`cume_dist()`

`dense_rank()`

Description	The rank of the current row *without gaps;* this function counts peer groups.
Return type	`BIGINT`
Example	`dense_rank()`

Aliases	`rank_dense()`

`first_value(expr[ ORDER BY ordering][ IGNORE NULLS])`

Description	Returns `expr` evaluated at the row that is the first row (with a non-null value of `expr` if `IGNORE NULLS` is set) of the window frame. If an `ORDER BY` clause is specified, the first row number is computed within the frame using the provided ordering instead of the frame ordering.
Return type	Same type as `expr`
Example	`first_value(column)`

`lag(expr[, offset[, default]][ ORDER BY ordering][ IGNORE NULLS])`

Description	Returns `expr` evaluated at the row that is `offset` rows (among rows with a non-null value of `expr` if `IGNORE NULLS` is set) before the current row within the window frame; if there is no such row, instead return `default` (which must be of the Same type as `expr`). Both `offset` and `default` are evaluated with respect to the current row. If omitted, `offset` defaults to `1` and default to `NULL`. If an `ORDER BY` clause is specified, the lagged row number is computed within the frame using the provided ordering instead of the frame ordering.
Return type	Same type as `expr`
Aliases	`lag(column, 3, 0)`

`last_value(expr[ ORDER BY ordering][ IGNORE NULLS])`

Description	Returns `expr` evaluated at the row that is the last row (among rows with a non-null value of `expr` if `IGNORE NULLS` is set) of the window frame. If omitted, `offset` defaults to `1` and default to `NULL`. If an `ORDER BY` clause is specified, the last row is determined within the frame using the provided ordering instead of the frame ordering.
Return type	Same type as `expr`
Example	`last_value(column)`

`lead(expr[, offset[, default]][ ORDER BY ordering][ IGNORE NULLS])`

Description	Returns `expr` evaluated at the row that is `offset` rows after the current row (among rows with a non-null value of `expr` if `IGNORE NULLS` is set) within the window frame; if there is no such row, instead return `default` (which must be of the Same type as `expr`). Both `offset` and `default` are evaluated with respect to the current row. If omitted, `offset` defaults to `1` and default to `NULL`. If an `ORDER BY` clause is specified, the leading row number is computed within the frame using the provided ordering instead of the frame ordering.
Return type	Same type as `expr`
Aliases	`lead(column, 3, 0)`

nth_value(expr, nth[ORDER BY ordering][IGNORE NULLS])

Description	Returns `expr` evaluated at the nth row (among rows with a non-null value of `expr` if `IGNORE NULLS` is set) of the window frame (counting from 1); `NULL` if no such row. If an `ORDER BY` clause is specified, the nth row number is computed within the frame using the provided ordering instead of the frame ordering.
Return type	Same type as `expr`
Aliases	`nth_value(column, 2)`

ntile(num_buckets[ORDER BY ordering])

Description	An integer ranging from 1 to `num_buckets`, dividing the partition as equally as possible. If an `ORDER BY` clause is specified, the ntile is computed within the frame using the provided ordering instead of the frame ordering.
Return type	`BIGINT`
Example	`ntile(4)`

percent_rank([ORDER BY ordering])

Description	The relative rank of the current row: `(rank() - 1) / (total partition rows - 1)`. If an `ORDER BY` clause is specified, the relative rank is computed within the frame using the provided ordering instead of the frame ordering.
Return type	`DOUBLE`
Example	`percent_rank()`

rank_dense()

Description	The rank of the current row *without gaps*.
Return type	`BIGINT`
Example	`rank_dense()`
Aliases	`dense_rank()`

rank([ORDER BY ordering])

Description	The rank of the current row *with gaps*; same as `row_number` of its first peer. If an `ORDER BY` clause is specified, the rank is computed within the frame using the provided ordering instead of the frame ordering.
Return type	`BIGINT`
Example	`rank()`

`row_number([ORDER BY ordering])`

Description	The number of the current row within the partition, counting from 1. If an `ORDER BY` clause is specified, the row number is computed within the frame using the provided ordering instead of the frame ordering.
Return type	`BIGINT`
Example	`row_number()`

Aggregate Window Functions

All aggregate functions can be used in a windowing context, including the optional `FILTER` clause. The `first` and `last` aggregate functions are shadowed by the respective general-purpose window functions, with the minor consequence that the `FILTER` clause is not available for these but `IGNORE NULLS` is.

DISTINCT Arguments

All aggregate window functions support using a `DISTINCT` clause for the arguments. When the `DISTINCT` clause is provided, only distinct values are considered in the computation of the aggregate. This is typically used in combination with the `COUNT` aggregate to get the number of distinct elements; but it can be used together with any aggregate function in the system. There are some aggregates that are insensitive to duplicate values (e.g., `min`, `max`) and for them this clause is parsed and ignored.

```
-- Count the number of distinct users at a given point in time
SELECT count(DISTINCT name) OVER (ORDER BY time) FROM sales;
-- Concatenate those distinct users into a list
SELECT list(DISTINCT name) OVER (ORDER BY time) FROM sales;
```

ORDER BY Arguments

All aggregate window functions support using an `ORDER BY` argument clause that is *different* from the window ordering. When the `ORDER BY` argument clause is provided, the values being aggregated are sorted before applying the function. Usually this is not important, but there are some order-sensitive aggregates that can have indeterminate results (e.g., `mode`, `list` and `string_agg`). These can be made deterministic by ordering the arguments. For order-insensitive aggregates, this clause is parsed and ignored.

```
-- Compute the modal value up to each time, breaking ties in favour of the most recent value.
SELECT mode(value ORDER BY time DESC) OVER (ORDER BY time) FROM sales;
```

The SQL standard does not provide for using `ORDER BY` with general-purpose window functions, but we have extended all of these functions (except `dense_rank`) to accept this syntax and use framing to restrict the range that the secondary ordering applies to.

```
-- Compare each athlete's time in an event with the best time to date
SELECT event, date, athlete, time
    first_value(time ORDER BY time DESC) OVER w AS record_time,
    first_value(athlete ORDER BY time DESC) OVER w AS record_athlete,
FROM meet_results
WINDOW w AS (PARTITION BY event ORDER BY datetime)
ORDER BY ALL
```

Note that there is no comma separating the arguments from the `ORDER BY` clause.

Nulls

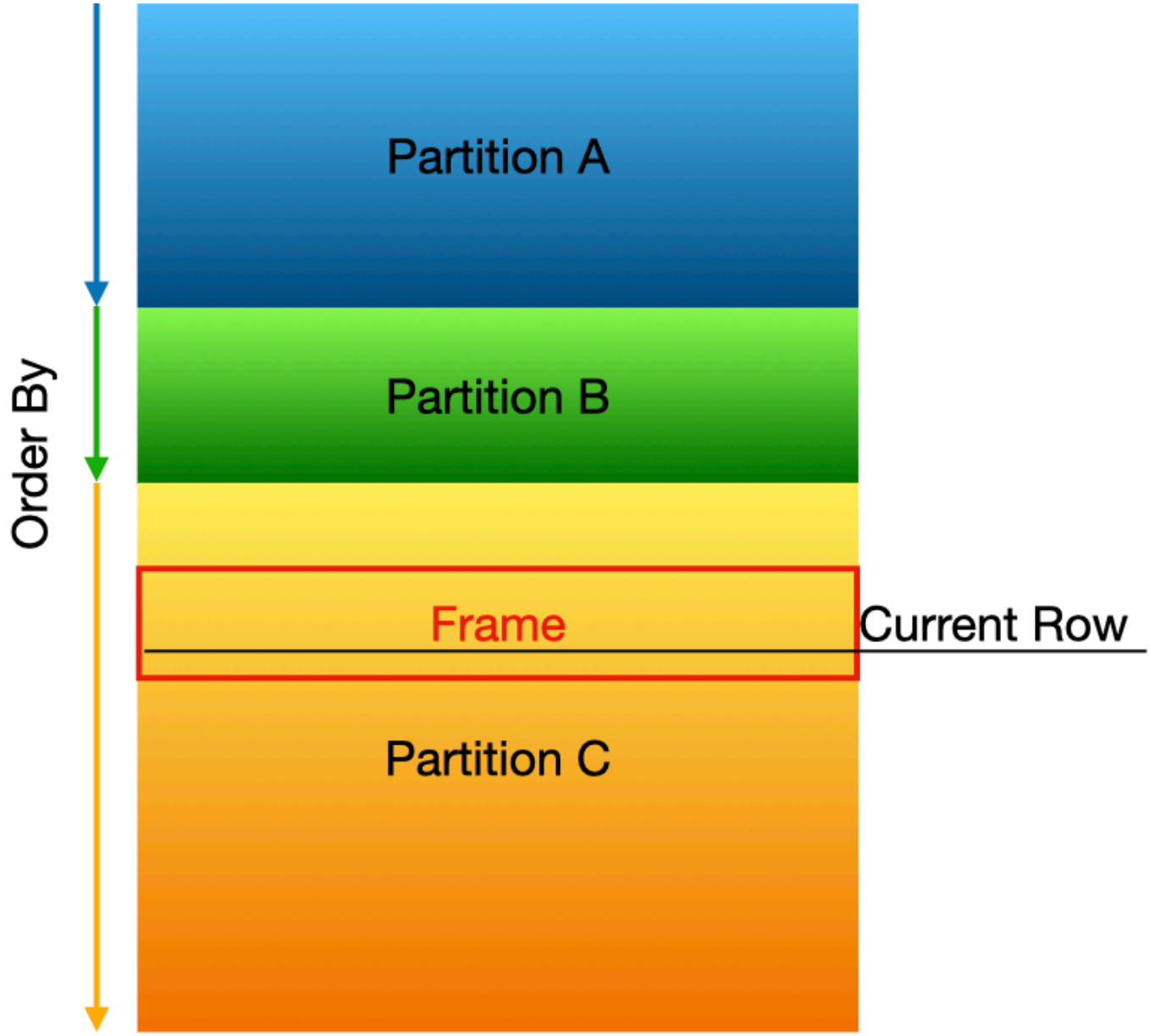
All general-purpose window functions that accept `IGNORE NULLS` respect nulls by default. This default behavior can optionally be made explicit via `RESPECT NULLS`.

In contrast, all aggregate window functions (except for `list` and its aliases, which can be made to ignore nulls via a `FILTER`) ignore nulls and do not accept `RESPECT NULLS`. For example, `sum(column) OVER (ORDER BY time) AS cumulativeColumn` computes a cumulative sum where rows with a `NULL` value of `column` have the same value of `cumulativeColumn` as the row that precedes them.

Evaluation

Windowing works by breaking a relation up into independent *partitions*, *ordering* those partitions, and then computing a new column for each row as a function of the nearby values. Some window functions depend only on the partition boundary and the ordering, but a few (including all the aggregates) also use a *frame*. Frames are specified as a number of rows on either side (*preceding* or *following*) of the *current row*. The distance can be specified as a number of *rows*, as a *range* of values using the partition's ordering value and a distance, or as a number of *groups* (sets of rows with the same sort value).

The full syntax is shown in the diagram at the top of the page, and this diagram visually illustrates computation environment:



Partition and Ordering

Partitioning breaks the relation up into independent, unrelated pieces. Partitioning is optional, and if none is specified then the entire relation is treated as a single partition. Window functions cannot access values outside of the partition containing the row they are being evaluated at.

Ordering is also optional, but without it the results of general-purpose window functions and order-sensitive aggregate functions, and the order of framing are not well-defined. Each partition is ordered using the same ordering clause.

Here is a table of power generation data, available as a CSV file (`power-plant-generation-history.csv`). To load the data, run:

```
CREATE TABLE "Generation History" AS
    FROM 'power-plant-generation-history.csv';
```

After partitioning by plant and ordering by date, it will have this layout:

Plant	Date	MWh
Boston	2019-01-02	564337
Boston	2019-01-03	507405
Boston	2019-01-04	528523
Boston	2019-01-05	469538
Boston	2019-01-06	474163
Boston	2019-01-07	507213
Boston	2019-01-08	613040
Boston	2019-01-09	582588
Boston	2019-01-10	499506
Boston	2019-01-11	482014
Boston	2019-01-12	486134
Boston	2019-01-13	531518
Worcester	2019-01-02	118860
Worcester	2019-01-03	101977
Worcester	2019-01-04	106054
Worcester	2019-01-05	92182
Worcester	2019-01-06	94492
Worcester	2019-01-07	99932
Worcester	2019-01-08	118854
Worcester	2019-01-09	113506
Worcester	2019-01-10	96644
Worcester	2019-01-11	93806
Worcester	2019-01-12	98963
Worcester	2019-01-13	107170

In what follows, we shall use this table (or small sections of it) to illustrate various pieces of window function evaluation.

The simplest window function is `row_number()`. This function just computes the 1-based row number within the partition using the query:

```
SELECT
    "Plant",
    "Date",
    row_number() OVER (PARTITION BY "Plant" ORDER BY "Date") AS "Row"
FROM "Generation History"
ORDER BY 1, 2;
```

The result will be the following:

Plant	Date	Row
Boston	2019-01-02	1
Boston	2019-01-03	2
Boston	2019-01-04	3

Plant	Date	Row
...	...	...
Worcester	2019-01-02	1
Worcester	2019-01-03	2
Worcester	2019-01-04	3
...	...	...

Note that even though the function is computed with an `ORDER BY` clause, the result does not have to be sorted, so the `SELECT` also needs to be explicitly sorted if that is desired.

Framing

Framing specifies a set of rows relative to each row where the function is evaluated. The distance from the current row is given as an expression either `PRECEDING` or `FOLLOWING` the current row in the order specified by the `ORDER BY` clause in the `OVER` specification. This distance can either be specified as an integral number of `ROWS` or `GROUPS`, or as a `RANGE` delta expression. It is invalid for a frame to start after it ends. For a `RANGE` specification, there must be only one ordering expression and it must support subtraction unless only the sentinel boundary values `UNBOUNDED PRECEDING` / `UNBOUNDED FOLLOWING` / `CURRENT ROW` are used. Using the `EXCLUDE` clause, rows comparing equal to the current row in the specified ordering expression (so-called peers) can be excluded from the frame.

The default frame is unbounded (i.e., the entire partition) when no `ORDER BY` clause is present and `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW` when an `ORDER BY` clause is present. By default, the `CURRENT ROW` boundary value (but not the `CURRENT ROW` in the `EXCLUDE` clause) means the current row and all its peers when `RANGE` or `GROUP` framing are used but it means only the current row when `ROWS` framing is used.

ROWS Framing

Here is a simple `ROW` frame query, using an aggregate function:

```
SELECT points,
    sum(points) OVER (
        ROWS BETWEEN 1 PRECEDING
                 AND 1 FOLLOWING) AS we
FROM results;
```

This query computes the `sum` of each point and the points on either side of it:

POINTS	WE
10	18
8	30
12	29
9	30
9	27
9	23
5	21
7	12

POINTS	WE
10	18
8	30
12	29
9	30
9	27
9	23
5	21
7	12

POINTS	WE
10	18
8	30
12	29
9	30
9	27
9	23
5	21
7	12

POINTS	WE
10	18
8	30
12	29
9	30
9	27
9	23
5	21
7	12

Notice that at the edge of the partition, there are only two values added together. This is because frames are cropped to the edge of the partition.

RANGE Framing

Returning to the power data, suppose the data is noisy. We might want to compute a 7 day moving average for each plant to smooth out the noise. To do this, we can use this window query:

```
SELECT "Plant", "Date",
    avg("MWh") OVER (
        PARTITION BY "Plant"
        ORDER BY "Date" ASC
        RANGE BETWEEN INTERVAL 3 DAYS PRECEDING
                  AND INTERVAL 3 DAYS FOLLOWING)
        AS "MWh 7-day Moving Average"
FROM "Generation History"
ORDER BY 1, 2;
```

This query partitions the data by `Plant` (to keep the different power plants' data separate), orders each plant's partition by `Date` (to put the energy measurements next to each other), and uses a RANGE frame of three days on either side of each day for the `avg` (to handle any missing days). This is the result:

Plant	Date	MWh 7-day Moving Average
Boston	2019-01-02	517450.75
Boston	2019-01-03	508793.20
Boston	2019-01-04	508529.83
...	...	...
Boston	2019-01-13	499793.00
Worcester	2019-01-02	104768.25
Worcester	2019-01-03	102713.00
Worcester	2019-01-04	102249.50
...	...	...

GROUPS Framing

The third type of framing counts *groups* of rows relative the current row. A *group* in this framing is a set of values with identical ORDER BY values. If we assume that power is being generated on every day, we can use GROUPS framing to compute the moving average of all power generated in the system without having to resort to date arithmetic:

```
SELECT "Date", "Plant",
    avg("MWh") OVER (
        ORDER BY "Date" ASC
        GROUPS BETWEEN 3 PRECEDING
                   AND 3 FOLLOWING)
        AS "MWh 7-day Moving Average"
FROM "Generation History"
ORDER BY 1, 2;
```

Date	Plant	MWh 7-day Moving Average
2019-01-02	Boston	311109.500

Date	Plant	MWh 7-day Moving Average
2019-01-02	Worcester	311109.500
2019-01-03	Boston	305753.100
2019-01-03	Worcester	305753.100
2019-01-04	Boston	305389.667
2019-01-04	Worcester	305389.667
...	...	...
2019-01-12	Boston	309184.900
2019-01-12	Worcester	309184.900
2019-01-13	Boston	299469.375
2019-01-13	Worcester	299469.375

Notice how the values for each date are the same.

EXCLUDE Clause

EXCLUDE is an optional modifier to the frame clause for excluding rows around the CURRENT ROW. This is useful when you want to compute some aggregate value of nearby rows to see how the current row compares to it.

In the following example, we want to know how an athlete's time in an event compares to the average of all the times recorded for their event within ±10 days:

```
SELECT
    event,
    date,
    athlete,
    avg(time) OVER w AS recent,
FROM results
WINDOW w AS (
    PARTITION BY event
    ORDER BY date
    RANGE BETWEEN INTERVAL 10 DAYS PRECEDING AND INTERVAL 10 DAYS FOLLOWING
        EXCLUDE CURRENT ROW
)
ORDER BY event, date, athlete;
```

There are four options for EXCLUDE that specify how to treat the current row:

- CURRENT ROW – exclude just the current row
- GROUP – exclude the current row and all its "peers" (rows that have the same ORDER BY value)
- TIES – exclude all peer rows, but *not* the current row (this makes a hole on either side)
- NO OTHERS – don't exclude anything (the default)

Exclusion is implemented for both windowed aggregates as well as for the `first`, `last` and `nth_value` functions.

WINDOW Clauses

Multiple different OVER clauses can be specified in the same SELECT, and each will be computed separately. Often, however, we want to use the same layout for multiple window functions. The WINDOW clause can be used to define a *named* window that can be shared between multiple window functions:

```
SELECT "Plant", "Date",
    min("MWh") OVER seven AS "MWh 7-day Moving Minimum",
    avg("MWh") OVER seven AS "MWh 7-day Moving Average",
    max("MWh") OVER seven AS "MWh 7-day Moving Maximum"
FROM "Generation History"
WINDOW seven AS (
    PARTITION BY "Plant"
    ORDER BY "Date" ASC
    RANGE BETWEEN INTERVAL 3 DAYS PRECEDING
              AND INTERVAL 3 DAYS FOLLOWING)
ORDER BY 1, 2;
```

The three window functions will also share the data layout, which will improve performance.

Multiple windows can be defined in the same `WINDOW` clause by comma-separating them:

```
SELECT "Plant", "Date",
    min("MWh") OVER seven AS "MWh 7-day Moving Minimum",
    avg("MWh") OVER seven AS "MWh 7-day Moving Average",
    max("MWh") OVER seven AS "MWh 7-day Moving Maximum",
    min("MWh") OVER three AS "MWh 3-day Moving Minimum",
    avg("MWh") OVER three AS "MWh 3-day Moving Average",
    max("MWh") OVER three AS "MWh 3-day Moving Maximum"
FROM "Generation History"
WINDOW
    seven AS (
        PARTITION BY "Plant"
        ORDER BY "Date" ASC
        RANGE BETWEEN INTERVAL 3 DAYS PRECEDING
                  AND INTERVAL 3 DAYS FOLLOWING),
    three AS (
        PARTITION BY "Plant"
        ORDER BY "Date" ASC
        RANGE BETWEEN INTERVAL 1 DAYS PRECEDING
        AND INTERVAL 1 DAYS FOLLOWING)
ORDER BY 1, 2;
```

The queries above do not use a number of clauses commonly found in select statements, like `WHERE`, `GROUP BY`, etc. For more complex queries you can find where `WINDOW` clauses fall in the canonical order of the `SELECT statement`.

Filtering the Results of Window Functions Using `QUALIFY`

Window functions are executed after the `WHERE` and `HAVING` clauses have been already evaluated, so it's not possible to use these clauses to filter the results of window functions The `QUALIFY` clause avoids the need for a subquery or `WITH` clause to perform this filtering.

Box and Whisker Queries

All aggregates can be used as windowing functions, including the complex statistical functions. These function implementations have been optimized for windowing, and we can use the window syntax to write queries that generate the data for moving box-and-whisker plots:

```
SELECT "Plant", "Date",
    min("MWh") OVER seven AS "MWh 7-day Moving Minimum",
    quantile_cont("MWh", [0.25, 0.5, 0.75]) OVER seven
        AS "MWh 7-day Moving IQR",
    max("MWh") OVER seven AS "MWh 7-day Moving Maximum",
FROM "Generation History"
WINDOW seven AS (
    PARTITION BY "Plant"
```

```
    ORDER BY "Date" ASC
    RANGE BETWEEN INTERVAL 3 DAYS PRECEDING
              AND INTERVAL 3 DAYS FOLLOWING)
ORDER BY 1, 2;
```

Constraints

In SQL, constraints can be specified for tables. Constraints enforce certain properties over data that is inserted into a table. Constraints can be specified along with the schema of the table as part of the CREATE TABLE statement. In certain cases, constraints can also be added to a table using the ALTER TABLE statement, but this is not currently supported for all constraints.

> **Warning.** Constraints have a strong impact on performance: they slow down loading and updates but speed up certain queries. Please consult the Performance Guide for details.

Syntax

Check Constraint

Check constraints allow you to specify an arbitrary Boolean expression. Any columns that *do not* satisfy this expression violate the constraint. For example, we could enforce that the name column does not contain spaces using the following CHECK constraint.

```
CREATE TABLE students (name VARCHAR CHECK (NOT contains(name, ' ')));
INSERT INTO students VALUES ('this name contains spaces');
```

```
Constraint Error:
CHECK constraint failed on table students with expression CHECK((NOT contains("name", ' ')))
```

Not Null Constraint

A not-null constraint specifies that the column cannot contain any NULL values. By default, all columns in tables are nullable. Adding NOT NULL to a column definition enforces that a column cannot contain NULL values.

```
CREATE TABLE students (name VARCHAR NOT NULL);
INSERT INTO students VALUES (NULL);
```

```
Constraint Error:
NOT NULL constraint failed: students.name
```

Primary Key and Unique Constraint

Primary key or unique constraints define a column, or set of columns, that are a unique identifier for a row in the table. The constraint enforces that the specified columns are *unique* within a table, i.e., that at most one row contains the given values for the set of columns.

```
CREATE TABLE students (id INTEGER PRIMARY KEY, name VARCHAR);
INSERT INTO students VALUES (1, 'Student 1');
INSERT INTO students VALUES (1, 'Student 2');
```

```
Constraint Error:
Duplicate key "id: 1" violates primary key constraint
```

```
CREATE TABLE students (id INTEGER, name VARCHAR, PRIMARY KEY (id, name));
INSERT INTO students VALUES (1, 'Student 1');
INSERT INTO students VALUES (1, 'Student 2');
INSERT INTO students VALUES (1, 'Student 1');
```

```
Constraint Error:
Duplicate key "id: 1, name: Student 1" violates primary key constraint
```

In order to enforce this property efficiently, an ART index is automatically created for every primary key or unique constraint that is defined in the table.

Primary key constraints and unique constraints are identical except for two points:

- A table can only have one primary key constraint defined, but many unique constraints
- A primary key constraint also enforces the keys to not be `NULL`.

```
CREATE TABLE students (id INTEGER PRIMARY KEY, name VARCHAR, email VARCHAR UNIQUE);
INSERT INTO students VALUES (1, 'Student 1', 'student1@uni.com');
INSERT INTO students VALUES (2, 'Student 2', 'student1@uni.com');
```

```
Constraint Error:
Duplicate key "email: student1@uni.com" violates unique constraint.
```

```
INSERT INTO students(id, name) VALUES (3, 'Student 3');
INSERT INTO students(name, email) VALUES ('Student 3', 'student3@uni.com');
```

```
Constraint Error:
NOT NULL constraint failed: students.id
```

> **Warning.** Indexes have certain limitations that might result in constraints being evaluated too eagerly, leading to constraint errors such as `violates primary key constraint` and `violates unique constraint`. See the indexes section for more details.

Foreign Keys

Foreign keys define a column, or set of columns, that refer to a primary key or unique constraint from *another* table. The constraint enforces that the key exists in the other table.

```
CREATE TABLE students (id INTEGER PRIMARY KEY, name VARCHAR);
CREATE TABLE subjects (id INTEGER PRIMARY KEY, name VARCHAR);
CREATE TABLE exams (
    exam_id INTEGER PRIMARY KEY,
    subject_id INTEGER REFERENCES subjects(id),
    student_id INTEGER REFERENCES students(id),
    grade INTEGER
);
INSERT INTO students VALUES (1, 'Student 1');
INSERT INTO subjects VALUES (1, 'CS 101');
INSERT INTO exams VALUES (1, 1, 1, 10);
INSERT INTO exams VALUES (2, 1, 2, 10);
```

```
Constraint Error:
Violates foreign key constraint because key "id: 2" does not exist in the referenced table
```

In order to enforce this property efficiently, an ART index is automatically created for every foreign key constraint that is defined in the table.

> **Warning.** Indexes have certain limitations that might result in constraints being evaluated too eagerly, leading to constraint errors such as `violates primary key constraint` and `violates unique constraint`. See the indexes section for more details.

Indexes

Index Types

DuckDB has two built-in index types. Indexes can also be defined via extensions.

Min-Max Index (Zonemap)

A min-max index (also known as zonemap or block range index) is *automatically created* for columns of all general-purpose data types.

Adaptive Radix Tree (ART)

An Adaptive Radix Tree (ART) is mainly used to ensure primary key constraints and to speed up point and very highly selective (i.e., < 0.1%) queries. ART indexes can be created manually using `CREATE INDEX` clause and they are automatically created for columns with a `UNIQUE` or `PRIMARY KEY` constraint.

> **Warning.** ART indexes must currently be able to fit in memory during index creation. Avoid creating ART indexes if the index does not fit in memory during index creation.

Indexes Defined by Extensions

DuckDB supports R-trees for spatial indexing via the `spatial` extension.

Persistence

Both min-max indexes and ART indexes are persisted on disk.

`CREATE INDEX` and `DROP INDEX` Statements

To create an ART index, use the `CREATE INDEX` statement. To drop an ART index, use the `DROP INDEX` statement.

Limitations of ART Indexes

ART indexes create a secondary copy of the data in a second location – this complicates processing, particularly when combined with transactions. Certain limitations apply when it comes to modifying data that is also stored in secondary indexes.

> As expected, indexes have a strong effect on performance, slowing down loading and updates, but speeding up certain queries. Please consult the Performance Guide for details.

Constraint Checking in UPDATE Statements

UPDATE statements on indexed columns and columns that cannot be updated in place are transformed into a DELETE of the original row followed by an INSERT of the updated row. This rewrite has performance implications, particularly for wide tables, as entire rows are rewritten instead of only the affected columns.

Additionally, it causes the following constraint-checking limitation of UPDATE statements. The same limitation exists in other DBMSs, like PostgreSQL.

In the example below, note how the number of rows exceeds DuckDB's standard vector size, which is 2048. The UPDATE statement is rewritten into a DELETE, followed by an INSERT. This rewrite happens per chunk of data (2048 rows) moving through DuckDB's processing pipeline. When updating `i = 2047` to `i = 2048`, we do not yet know that 2048 becomes 2049, and so forth. That is because we have not yet seen that chunk. Thus, we throw a constraint violation.

```
CREATE TABLE my_table (i INTEGER PRIMARY KEY);
INSERT INTO my_table SELECT range FROM range(3_000);
UPDATE my_table SET i = i + 1;
```

```
Constraint Error:
Duplicate key "i: 2048" violates primary key constraint.
```

A workaround is to split the UPDATE into a `DELETE ... RETURNING ...` followed by an INSERT, with some additional logic to (temporarily) store the result of the DELETE. All statements should be run inside a transaction via BEGIN, and eventually COMMIT.

Here's an example of how that could look like in the command line client.

```
CREATE TABLE my_table (i INTEGER PRIMARY KEY);
INSERT INTO my_table SELECT range FROM range(3_000);

BEGIN;
CREATE TEMP TABLE tmp AS SELECT i FROM my_table;
DELETE FROM my_table;
INSERT INTO my_table SELECT i FROM tmp;
DROP TABLE tmp;
COMMIT;
```

In other clients, you might be able to fetch the result of `DELETE ... RETURNING ...`. Then, you can use that result in a subsequent `INSERT ...` statements, or potentially make use of DuckDB's `Appender` (if available in the client).

Over-Eager Constraint Checking in Foreign Keys

This limitation occurs if you meet the following conditions:

- A table has a `FOREIGN KEY` constraint.
- There is an UPDATE on the corresponding `PRIMARY KEY` table, which DuckDB rewrites into a DELETE followed by an INSERT.
- The to-be-deleted row exists in the foreign key table.

If these hold, you'll encounter an unexpected constraint violation:

```
CREATE TABLE pk_table (id INTEGER PRIMARY KEY, payload VARCHAR[]);
INSERT INTO pk_table VALUES (1, ['hello']);
CREATE TABLE fk_table (id INTEGER REFERENCES pk_table(id));
INSERT INTO fk_table VALUES (1);
UPDATE pk_table SET payload = ['world'] WHERE id = 1;
```

```
Constraint Error:
Violates foreign key constraint because key "id: 1" is still referenced by a foreign key in a different
table. If this is an unexpected constraint violation, please refer to our foreign key limitations in the
documentation
```

The reason for this is because DuckDB does not yet support "looking ahead". During the INSERT, it is unaware it will reinsert the foreign key value as part of the UPDATE rewrite.

Meta Queries

Information Schema

The views in the `information_schema` are SQL-standard views that describe the catalog entries of the database. These views can be filtered to obtain information about a specific column or table. DuckDB's implementation is based on PostgreSQL's information schema.

Tables

`character_sets`: Character Sets

Column	Description	Type	Example
`character_set_catalog`	Currently not implemented – always `NULL`.	`VARCHAR`	`NULL`
`character_set_schema`	Currently not implemented – always `NULL`.	`VARCHAR`	`NULL`
`character_set_name`	Name of the character set, currently implemented as showing the name of the database encoding.	`VARCHAR`	`'UTF8'`
`character_repertoire`	Character repertoire, showing UCS if the encoding is `UTF8`, else just the encoding name.	`VARCHAR`	`'UCS'`
`form_of_use`	Character encoding form, same as the database encoding.	`VARCHAR`	`'UTF8'`
`default_collate_catalog`	Name of the database containing the default collation (always the current database).	`VARCHAR`	`'my_db'`
`default_collate_schema`	Name of the schema containing the default collation.	`VARCHAR`	`'pg_catalog'`
`default_collate_name`	Name of the default collation.	`VARCHAR`	`'ucs_basic'`

`columns`: Columns

The view that describes the catalog information for columns is `information_schema.columns`. It lists the column present in the database and has the following layout:

Column	Description	Type	Example
`table_catalog`	Name of the database containing the table (always the current database).	`VARCHAR`	`'my_db'`

Column	Description	Type	Example
`table_schema`	Name of the schema containing the table.	`VARCHAR`	`'main'`
`table_name`	Name of the table.	`VARCHAR`	`'widgets'`
`column_name`	Name of the column.	`VARCHAR`	`'price'`
`ordinal_position`	Ordinal position of the column within the table (count starts at 1).	`INTEGER`	`5`
`column_default`	Default expression of the column.	`VARCHAR`	`1.99`
`is_nullable`	`YES` if the column is possibly nullable, `NO` if it is known not nullable.	`VARCHAR`	`'YES'`
`data_type`	Data type of the column.	`VARCHAR`	`'DECIMAL(18, 2)'`
`character_maximum_length`	If `data_type` identifies a character or bit string type, the declared maximum length; `NULL` for all other data types or if no maximum length was declared.	`INTEGER`	`255`
`character_octet_length`	If `data_type` identifies a character type, the maximum possible length in octets (bytes) of a datum; `NULL` for all other data types. The maximum octet length depends on the declared character maximum length (see above) and the character encoding.	`INTEGER`	`1073741824`
`numeric_precision`	If `data_type` identifies a numeric type, this column contains the (declared or implicit) precision of the type for this column. The precision indicates the number of significant digits. For all other data types, this column is `NULL`.	`INTEGER`	`18`
`numeric_scale`	If `data_type` identifies a numeric type, this column contains the (declared or implicit) scale of the type for this column. The precision indicates the number of significant digits. For all other data types, this column is `NULL`.	`INTEGER`	`2`
`datetime_precision`	If `data_type` identifies a date, time, timestamp, or interval type, this column contains the (declared or implicit) fractional seconds precision of the type for this column, that is, the number of decimal digits maintained following the decimal point in the seconds value. No fractional seconds are currently supported in DuckDB. For all other data types, this column is `NULL`.	`INTEGER`	`0`

`constraint_column_usage`: Constraint Column Usage

This view describes all columns in the current database that are used by some constraint. For a check constraint, this view identifies the columns that are used in the check expression. For a not-null constraint, this view identifies the column that the constraint is defined on. For a foreign key constraint, this view identifies the columns that the foreign key references. For a unique or primary key constraint, this view identifies the constrained columns.

Column	Description	Type	Example
table_catalog	Name of the database that contains the table that contains the column that is used by some constraint (always the current database)	VARCHAR	'my_db'
table_schema	Name of the schema that contains the table that contains the column that is used by some constraint	VARCHAR	'main'
table_name	Name of the table that contains the column that is used by some constraint	VARCHAR	'widgets'
column_name	Name of the column that is used by some constraint	VARCHAR	'price'
constraint_catalog	Name of the database that contains the constraint (always the current database)	VARCHAR	'my_db'
constraint_schema	Name of the schema that contains the constraint	VARCHAR	'main'
constraint_name	Name of the constraint	VARCHAR	'exam_id_students_id_fkey'

`key_column_usage`: Key Column Usage

Column	Description	Type	Example
constraint_catalog	Name of the database that contains the constraint (always the current database).	VARCHAR	'my_db'
constraint_schema	Name of the schema that contains the constraint.	VARCHAR	'main'
constraint_name	Name of the constraint.	VARCHAR	'exams_exam_id_fkey'
table_catalog	Name of the database that contains the table that contains the column that is restricted by this constraint (always the current database).	VARCHAR	'my_db'
table_schema	Name of the schema that contains the table that contains the column that is restricted by this constraint.	VARCHAR	'main'
table_name	Name of the table that contains the column that is restricted by this constraint.	VARCHAR	'exams'
column_name	Name of the column that is restricted by this constraint.	VARCHAR	'exam_id'
ordinal_position	Ordinal position of the column within the constraint key (count starts at 1).	INTEGER	1
position_in_unique_constraint	For a foreign-key constraint, ordinal position of the referenced column within its unique constraint (count starts at 1); otherwise NULL.	INTEGER	1

`referential_constraints`: Referential Constraints

Column	Description	Type	Example
constraint_catalog	Name of the database containing the constraint (always the current database).	VARCHAR	'my_db'
constraint_schema	Name of the schema containing the constraint.	VARCHAR	main
constraint_name	Name of the constraint.	VARCHAR	exam_id_students_id_fkey
unique_constraint_catalog	Name of the database that contains the unique or primary key constraint that the foreign key constraint references.	VARCHAR	'my_db'
unique_constraint_schema	Name of the schema that contains the unique or primary key constraint that the foreign key constraint references.	VARCHAR	'main'
unique_constraint_name	Name of the unique or primary key constraint that the foreign key constraint references.	VARCHAR	'students_id_pkey'
match_option	Match option of the foreign key constraint. Always NONE.	VARCHAR	NONE
update_rule	Update rule of the foreign key constraint. Always NO ACTION.	VARCHAR	NO ACTION
delete_rule	Delete rule of the foreign key constraint. Always NO ACTION.	VARCHAR	NO ACTION

schemata: Database, Catalog and Schema

The top level catalog view is information_schema.schemata. It lists the catalogs and the schemas present in the database and has the following layout:

Column	Description	Type	Example
catalog_name	Name of the database that the schema is contained in.	VARCHAR	'my_db'
schema_name	Name of the schema.	VARCHAR	'main'
schema_owner	Name of the owner of the schema. Not yet implemented.	VARCHAR	'duckdb'
default_character_set_catalog	Applies to a feature not available in DuckDB.	VARCHAR	NULL
default_character_set_schema	Applies to a feature not available in DuckDB.	VARCHAR	NULL
default_character_set_name	Applies to a feature not available in DuckDB.	VARCHAR	NULL
sql_path	Applies to a feature not available in DuckDB.	VARCHAR	NULL

tables: Tables and Views

The view that describes the catalog information for tables and views is information_schema.tables. It lists the tables present in the database and has the following layout:

Column	Description	Type	Example
table_catalog	The catalog the table or view belongs to.	VARCHAR	'my_db'
table_schema	The schema the table or view belongs to.	VARCHAR	'main'
table_name	The name of the table or view.	VARCHAR	'widgets'
table_type	The type of table. One of: BASE TABLE, LOCAL TEMPORARY, VIEW.	VARCHAR	'BASE TABLE'
self_referencing_column_name	Applies to a feature not available in DuckDB.	VARCHAR	NULL
reference_generation	Applies to a feature not available in DuckDB.	VARCHAR	NULL
user_defined_type_catalog	If the table is a typed table, the name of the database that contains the underlying data type (always the current database), else NULL. Currently unimplemented.	VARCHAR	NULL
user_defined_type_schema	If the table is a typed table, the name of the schema that contains the underlying data type, else NULL. Currently unimplemented.	VARCHAR	NULL
user_defined_type_name	If the table is a typed table, the name of the underlying data type, else NULL. Currently unimplemented.	VARCHAR	NULL
is_insertable_into	YES if the table is insertable into, NO if not (Base tables are always insertable into, views not necessarily.)	VARCHAR	'YES'
is_typed	YES if the table is a typed table, NO if not.	VARCHAR	'NO'
commit_action	Not yet implemented.	VARCHAR	'NO'

table_constraints: Table Constraints

Column	Description	Type	Example
constraint_catalog	Name of the database that contains the constraint (always the current database).	VARCHAR	'my_db'
constraint_schema	Name of the schema that contains the constraint.	VARCHAR	'main'
constraint_name	Name of the constraint.	VARCHAR	'exams_exam_id_fkey'
table_catalog	Name of the database that contains the table (always the current database).	VARCHAR	'my_db'
table_schema	Name of the schema that contains the table.	VARCHAR	'main'
table_name	Name of the table.	VARCHAR	'exams'
constraint_type	Type of the constraint: CHECK, FOREIGN KEY, PRIMARY KEY, or UNIQUE.	VARCHAR	'FOREIGN KEY'
is_deferrable	YES if the constraint is deferrable, NO if not.	VARCHAR	'NO'
initially_deferred	YES if the constraint is deferrable and initially deferred, NO if not.	VARCHAR	'NO'
enforced	Always YES.	VARCHAR	'YES'

Column	Description	Type	Example
`nulls_distinct`	If the constraint is a unique constraint, then `YES` if the constraint treats `NULL`s as distinct or `NO` if it treats `NULL`s as not distinct, otherwise `NULL` for other types of constraints.	VARCHAR	`'YES'`

Catalog Functions

Several functions are also provided to see details about the catalogs and schemas that are configured in the database.

Function	Description	Example	Result
`current_catalog()`	Return the name of the currently active catalog. Default is memory.	`current_catalog()`	`'memory'`
`current_schema()`	Return the name of the currently active schema. Default is main.	`current_schema()`	`'main'`
`current_schemas(boolean)`	Return list of schemas. Pass a parameter of `true` to include implicit schemas.	`current_schemas(true)`	`['temp', 'main', 'pg_catalog']`

DuckDB_% Metadata Functions

DuckDB offers a collection of table functions that provide metadata about the current database. These functions reside in the `main` schema and their names are prefixed with `duckdb_`.

The resultset returned by a `duckdb_` table function may be used just like an ordinary table or view. For example, you can use a `duckdb_` function call in the `FROM` clause of a `SELECT` statement, and you may refer to the columns of its returned resultset elsewhere in the statement, for example in the `WHERE` clause.

Table functions are still functions, and you should write parenthesis after the function name to call it to obtain its returned resultset:

```
SELECT * FROM duckdb_settings();
```

Alternatively, you may execute table functions also using the `CALL`-syntax:

```
CALL duckdb_settings();
```

In this case too, the parentheses are mandatory.

> For some of the `duckdb_%` functions, there is also an identically named view available, which also resides in the `main` schema. Typically, these views do a `SELECT` on the `duckdb_` table function with the same name, while filtering out those objects that are marked as internal. We mention it here, because if you accidentally omit the parentheses in your `duckdb_` table function call, you might still get a result, but from the identically named view.

Example:

The `duckdb_views()` *table function* returns all views, including those marked internal:

```
SELECT * FROM duckdb_views();
```

The `duckdb_views` *view* returns views that are not marked as internal:

```
SELECT * FROM duckdb_views;
```

duckdb_columns

The `duckdb_columns()` function provides metadata about the columns available in the DuckDB instance.

Column	Description	Type
`database_name`	The name of the database that contains the column object.	`VARCHAR`
`database_oid`	Internal identifier of the database that contains the column object.	`BIGINT`
`schema_name`	The SQL name of the schema that contains the table object that defines this column.	`VARCHAR`
`schema_oid`	Internal identifier of the schema object that contains the table of the column.	`BIGINT`
`table_name`	The SQL name of the table that defines the column.	`VARCHAR`
`table_oid`	Internal identifier (name) of the table object that defines the column.	`BIGINT`
`column_name`	The SQL name of the column.	`VARCHAR`
`column_index`	The unique position of the column within its table.	`INTEGER`
`comment`	A comment created by the `COMMENT ON` statement.	`VARCHAR`
`internal`	`true` if this column built-in, `false` if it is user-defined.	`BOOLEAN`
`column_default`	The default value of the column (expressed in SQL)	`VARCHAR`
`is_nullable`	`true` if the column can hold `NULL` values; `false` if the column cannot hold `NULL`-values.	`BOOLEAN`
`data_type`	The name of the column datatype.	`VARCHAR`
`data_type_id`	The internal identifier of the column data type.	`BIGINT`
`character_maximum_length`	Always `NULL`. DuckDB text types do not enforce a value length restriction based on a length type parameter.	`INTEGER`
`numeric_precision`	The number of units (in the base indicated by `numeric_precision_radix`) used for storing column values. For integral and approximate numeric types, this is the number of bits. For decimal types, this is the number of digits positions.	`INTEGER`
`numeric_precision_radix`	The number-base of the units in the `numeric_precision` column. For integral and approximate numeric types, this is `2`, indicating the precision is expressed as a number of bits. For the `decimal` type this is `10`, indicating the precision is expressed as a number of decimal positions.	`INTEGER`
`numeric_scale`	Applicable to `decimal` type. Indicates the maximum number of fractional digits (i.e., the number of digits that may appear after the decimal separator).	`INTEGER`

The `information_schema.columns` system view provides a more standardized way to obtain metadata about database columns, but the `duckdb_columns` function also returns metadata about DuckDB internal objects. (In fact, `information_schema.columns` is implemented as a query on top of `duckdb_columns()`)

duckdb_constraints

The `duckdb_constraints()` function provides metadata about the constraints available in the DuckDB instance.

Column	Description	Type
`database_name`	The name of the database that contains the constraint.	`VARCHAR`
`database_oid`	Internal identifier of the database that contains the constraint.	`BIGINT`
`schema_name`	The SQL name of the schema that contains the table on which the constraint is defined.	`VARCHAR`
`schema_oid`	Internal identifier of the schema object that contains the table on which the constraint is defined.	`BIGINT`
`table_name`	The SQL name of the table on which the constraint is defined.	`VARCHAR`
`table_oid`	Internal identifier (name) of the table object on which the constraint is defined.	`BIGINT`
`constraint_index`	Indicates the position of the constraint as it appears in its table definition.	`BIGINT`
`constraint_type`	Indicates the type of constraint. Applicable values are `CHECK`, `FOREIGN KEY`, `PRIMARY KEY`, `NOT NULL`, `UNIQUE`.	`VARCHAR`
`constraint_text`	The definition of the constraint expressed as a SQL-phrase. (Not necessarily a complete or syntactically valid DDL-statement.)	`VARCHAR`
`expression`	If constraint is a check constraint, the definition of the condition being checked, otherwise `NULL`.	`VARCHAR`
`constraint_column_indexes`	An array of table column indexes referring to the columns that appear in the constraint definition.	`BIGINT[]`
`constraint_column_names`	An array of table column names appearing in the constraint definition.	`VARCHAR[]`
`constraint_name`	The naem of constraint.	`VARCHAR`
`referenced_table`	The table referenced by the constraint.	`VARCHAR`
`referenced_column_names`	The column names references the by the constraint.	`VARCHAR[]`

The `information_schema.referential_constraints` and `information_schema.table_constraints` system views provide a more standardized way to obtain metadata about constraints, but the `duckdb_constraints` function also returns metadata about DuckDB internal objects. (In fact, `information_schema.referential_constraints` and `information_schema.table_constraints` are implemented as a query on top of `duckdb_constraints()`)

duckdb_databases

The `duckdb_databases()` function lists the databases that are accessible from within the current DuckDB process. Apart from the database associated at startup, the list also includes databases that were attached later on to the DuckDB process

Column	Description	Type
`database_name`	The name of the database, or the alias if the database was attached using an ALIAS-clause.	`VARCHAR`
`database_oid`	The internal identifier of the database.	`VARCHAR`
`path`	The file path associated with the database.	`VARCHAR`
`comment`	A comment created by the `COMMENT ON` statement.	`VARCHAR`

Column	Description	Type
tags	A map of string key–value pairs.	MAP(VARCHAR, VARCHAR)
internal	true indicates a system or built-in database. false indicates a user-defined database.	BOOLEAN
type	The type indicates the type of RDBMS implemented by the attached database. For DuckDB databases, that value is duckdb.	VARCHAR
readonly	Denotes whether the database is read-only.	BOOLEAN

duckdb_dependencies

The duckdb_dependencies() function provides metadata about the dependencies available in the DuckDB instance.

Column	Description	Type
classid	Always 0	BIGINT
objid	The internal id of the object.	BIGINT
objsubid	Always 0	INTEGER
refclassid	Always 0	BIGINT
refobjid	The internal id of the dependent object.	BIGINT
refobjsubid	Always 0	INTEGER
deptype	The type of dependency. Either regular (n) or automatic (a).	VARCHAR

duckdb_extensions

The duckdb_extensions() function provides metadata about the extensions available in the DuckDB instance.

Column	Description	Type
extension_name	The name of the extension.	VARCHAR
loaded	true if the extension is loaded, false if it's not loaded.	BOOLEAN
installed	true if the extension is installed, false if it's not installed.	BOOLEAN
install_path	(BUILT-IN) if the extension is built-in, otherwise, the filesystem path where binary that implements the extension resides.	VARCHAR
description	Human readable text that describes the extension's functionality.	VARCHAR
aliases	List of alternative names for this extension.	VARCHAR[]
extension_version	The version of the extension (vX.Y.Z for stable versions and 6-character hash for unstable versions).	VARCHAR
install_mode	The installation mode that was used to install the extension: UNKNOWN, REPOSITORY, CUSTOM_PATH, STATICALLY_LINKED, NOT_INSTALLED, NULL.	VARCHAR
installed_from	Name of the repository the extension was installed from, e.g., community or core_nightly. The empty string denotes the core repository.	VARCHAR

duckdb_functions

The `duckdb_functions()` function provides metadata about the functions (including macros) available in the DuckDB instance.

Column	Description	Type
`database_name`	The name of the database that contains this function.	`VARCHAR`
`database_oid`	Internal identifier of the database containing the index.	`BIGINT`
`schema_name`	The SQL name of the schema where the function resides.	`VARCHAR`
`function_name`	The SQL name of the function.	`VARCHAR`
`function_type`	The function kind. Value is one of: `table`,`scalar`,`aggregate`,`pragma`,`macro`	`VARCHAR`
`description`	Description of this function (always `NULL`)	`VARCHAR`
`comment`	A comment created by the `COMMENT ON` statement.	`VARCHAR`
`tags`	A map of string key–value pairs.	`MAP(VARCHAR, VARCHAR)`
`return_type`	The logical data type name of the returned value. Applicable for scalar and aggregate functions.	`VARCHAR`
`parameters`	If the function has parameters, the list of parameter names.	`VARCHAR[]`
`parameter_types`	If the function has parameters, a list of logical data type names corresponding to the parameter list.	`VARCHAR[]`
`varargs`	The name of the data type in case the function has a variable number of arguments, or `NULL` if the function does not have a variable number of arguments.	`VARCHAR`
`macro_definition`	If this is a macro, the SQL expression that defines it.	`VARCHAR`
`has_side_effects`	`false` if this is a pure function. `true` if this function changes the database state (like sequence functions `nextval()` and `curval()`).	`BOOLEAN`
`internal`	`true` if the function is built-in (defined by DuckDB or an extension), `false` if it was defined using the `CREATE MACRO` statement.	`BOOLEAN`
`function_oid`	The internal identifier for this function.	`BIGINT`
`examples`	Examples of using the function. Used to generate the documentation.	`VARCHAR[]`
`stability`	The stability of the function (`CONSISTENT`, `VOLATILE`, `CONSISTENT_WITHIN_QUERY` or `NULL`)	`VARCHAR`

duckdb_indexes

The `duckdb_indexes()` function provides metadata about secondary indexes available in the DuckDB instance.

Column	Description	Type
`database_name`	The name of the database that contains this index.	`VARCHAR`
`database_oid`	Internal identifier of the database containing the index.	`BIGINT`

Column	Description	Type
schema_name	The SQL name of the schema that contains the table with the secondary index.	VARCHAR
schema_oid	Internal identifier of the schema object.	BIGINT
index_name	The SQL name of this secondary index.	VARCHAR
index_oid	The object identifier of this index.	BIGINT
table_name	The name of the table with the index.	VARCHAR
table_oid	Internal identifier (name) of the table object.	BIGINT
comment	A comment created by the COMMENT ON statement.	VARCHAR
tags	A map of string key–value pairs.	MAP(VARCHAR, VARCHAR)
is_unique	true if the index was created with the UNIQUE modifier, false if it was not.	BOOLEAN
is_primary	Always false.	BOOLEAN
expressions	Always NULL.	VARCHAR
sql	The definition of the index, expressed as a CREATE INDEX SQL statement.	VARCHAR

Note that duckdb_indexes only provides metadata about secondary indexes, i.e., those indexes created by explicit CREATE INDEX statements. Primary keys, foreign keys, and UNIQUE constraints are maintained using indexes, but their details are included in the duckdb_constraints() function.

duckdb_keywords

The duckdb_keywords() function provides metadata about DuckDB's keywords and reserved words.

Column	Description	Type
keyword_name	The keyword.	VARCHAR
keyword_category	Indicates the category of the keyword. Values are column_name, reserved, type_function and unreserved.	VARCHAR

duckdb_log_contexts

The duckdb_log_contexts() function provides information on the contexts of DuckDB log entries.

Column	Description	Type
context_id	The identifier of the context. The context_id column in the duckdb_logs table is a foreign key that points to this column.	UBIGINT
scope	The scope of the context (connection, database or file_opener TODO: + more ? https://github.com/duckdb/duckdb/pull/15119).	VARCHAR
connection_id	The identifier of the connection.	UBIGINT

Column	Description	Type
`transaction_id`	The identifier of the transaction.	`UBIGINT`
`query_id`	The identifier of the query.	`UBIGINT`
`thread_id`	The identifier of the thread.	`UBIGINT`

`duckdb_logs`

The `duckdb_logs()` function returns a table of DuckDB log entries.

Column	Description	Type
`context_id`	The identifier of the context of the log entry. Foreign key to the `duckdb_log_contexts` table.	`UBIGINT`
`timestamp`	The timestamp of the log entry.	`TIMESTAMP`
`type`	The type of the log entry. TODO: ??	`VARCHAR`
`log_level`	The level of the log entry (`TRACE`, `DEBUG`, `INFO`, `WARN`, `ERROR` or `FATAL`).	`VARCHAR`
`message`	The message of the log entry.	`VARCHAR`

`duckdb_memory`

The `duckdb_memory()` function provides metadata about DuckDB's buffer manager.

Column	Description	Type
`tag`	The memory tag. It has one of the following values: `BASE_TABLE`, `HASH_TABLE`, `PARQUET_READER`, `CSV_READER`, `ORDER_BY`, `ART_INDEX`, `COLUMN_DATA`, `METADATA`, `OVERFLOW_STRINGS`, `IN_MEMORY_TABLE`, `ALLOCATOR`, `EXTENSION`.	`VARCHAR`
`memory_usage_bytes`	The memory used (in bytes).	`BIGINT`
`temporary_storage_bytes`	The disk storage used (in bytes).	`BIGINT`

`duckdb_optimizers`

The `duckdb_optimizers()` function provides metadata about the optimization rules (e.g., `expression_rewriter`, `filter_pushdown`) available in the DuckDB instance. These can be selectively turned off using `PRAGMA disabled_optimizers`.

Column	Description	Type
`name`	The name of the optimization rule.	`VARCHAR`

duckdb_prepared_statements

The `duckdb_prepared_statements()` function provides metadata about the prepared statements that exist in the current DuckDB session.

Column	Description	Type
`name`	The name of the prepared statement.	`VARCHAR`
`statement`	The SQL statement.	`VARCHAR`
`parameter_types`	The expected parameter types for the statement's parameters. Currently returns `UNKNOWN` for all parameters.	`VARCHAR[]`
`result_types`	The types of the columns in the table returned by the prepared statement.	`VARCHAR[]`

duckdb_schemas

The `duckdb_schemas()` function provides metadata about the schemas available in the DuckDB instance.

Column	Description	Type
`oid`	Internal identifier of the schema object.	`BIGINT`
`database_name`	The name of the database that contains this schema.	`VARCHAR`
`database_oid`	Internal identifier of the database containing the schema.	`BIGINT`
`schema_name`	The SQL name of the schema.	`VARCHAR`
`comment`	A comment created by the `COMMENT ON` statement.	`VARCHAR`
`tags`	A map of string key–value pairs.	`MAP(VARCHAR, VARCHAR)`
`internal`	`true` if this is an internal (built-in) schema, `false` if this is a user-defined schema.	`BOOLEAN`
`sql`	Always `NULL`	`VARCHAR`

The `information_schema.schemata` system view provides a more standardized way to obtain metadata about database schemas.

duckdb_secret_types

The `duckdb_secret_types()` lists secret types that are supported in the current DuckDB session.

Column	Description	Type
`type`	The name of the secret type, e.g., `s3`.	`VARCHAR`
`default_provider`	The default secret provider, e.g., `config`.	`VARCHAR`
`extension`	The extension that registered the secret type, e.g., `aws`.	`VARCHAR`

duckdb_secrets

The `duckdb_secrets()` function provides metadata about the secrets available in the DuckDB instance.

Column	Description	Type
`name`	The name of the secret.	`VARCHAR`
`type`	The type of the secret, e.g., `S3`, `GCS`, `R2`, `AZURE`.	`VARCHAR`
`provider`	The provider of the secret.	`VARCHAR`
`persistent`	Denotes whether the secret is persisent.	`BOOLEAN`
`storage`	The backend for storing the secret.	`VARCHAR`
`scope`	The scope of the secret.	`VARCHAR[]`
`secret_string`	Returns the content of the secret as a string. Sensitive pieces of information, e.g., they access key, are redacted.	`VARCHAR`

duckdb_sequences

The `duckdb_sequences()` function provides metadata about the sequences available in the DuckDB instance.

Column	Description	Type
`database_name`	The name of the database that contains this sequence	`VARCHAR`
`database_oid`	Internal identifier of the database containing the sequence.	`BIGINT`
`schema_name`	The SQL name of the schema that contains the sequence object.	`VARCHAR`
`schema_oid`	Internal identifier of the schema object that contains the sequence object.	`BIGINT`
`sequence_name`	The SQL name that identifies the sequence within the schema.	`VARCHAR`
`sequence_oid`	The internal identifier of this sequence object.	`BIGINT`
`comment`	A comment created by the `COMMENT ON` statement.	`VARCHAR`
`tags`	A map of string key–value pairs.	`MAP(VARCHAR, VARCHAR)`
`temporary`	Whether this sequence is temporary. Temporary sequences are transient and only visible within the current connection.	`BOOLEAN`
`start_value`	The initial value of the sequence. This value will be returned when `nextval()` is called for the very first time on this sequence.	`BIGINT`
`min_value`	The minimum value of the sequence.	`BIGINT`
`max_value`	The maximum value of the sequence.	`BIGINT`
`increment_by`	The value that is added to the current value of the sequence to draw the next value from the sequence.	`BIGINT`
`cycle`	Whether the sequence should start over when drawing the next value would result in a value outside the range.	`BOOLEAN`
`last_value`	`NULL` if no value was ever drawn from the sequence using `nextval(...)`. `1` if a value was drawn.	`BIGINT`
`sql`	The definition of this object, expressed as SQL DDL-statement.	`VARCHAR`

Attributes like `temporary`, `start_value` etc. correspond to the various options available in the `CREATE SEQUENCE` statement and are documented there in full. Note that the attributes will always be filled out in the `duckdb_sequences` resultset, even if they were not explicitly specified in the `CREATE SEQUENCE` statement.

> 1. The column name `last_value` suggests that it contains the last value that was drawn from the sequence, but that is not the case. It's either `NULL` if a value was never drawn from the sequence, or `1` (when there was a value drawn, ever, from the sequence).
> 2. If the sequence cycles, then the sequence will start over from the boundary of its range, not necessarily from the value specified as start value.

`duckdb_settings`

The `duckdb_settings()` function provides metadata about the settings available in the DuckDB instance.

Column	Description	Type
`name`	Name of the setting.	`VARCHAR`
`value`	Current value of the setting.	`VARCHAR`
`description`	A description of the setting.	`VARCHAR`
`input_type`	The logical datatype of the setting's value.	`VARCHAR`
`scope`	The scope of the setting (`LOCAL` or `GLOBAL`).	`VARCHAR`

The various settings are described in the configuration page.

`duckdb_tables`

The `duckdb_tables()` function provides metadata about the base tables available in the DuckDB instance.

Column	Description	Type
`database_name`	The name of the database that contains this table	`VARCHAR`
`database_oid`	Internal identifier of the database containing the table.	`BIGINT`
`schema_name`	The SQL name of the schema that contains the base table.	`VARCHAR`
`schema_oid`	Internal identifier of the schema object that contains the base table.	`BIGINT`
`table_name`	The SQL name of the base table.	`VARCHAR`
`table_oid`	Internal identifier of the base table object.	`BIGINT`
`comment`	A comment created by the `COMMENT ON` statement.	`VARCHAR`
`tags`	A map of string key–value pairs.	`MAP(VARCHAR, VARCHAR)`
`internal`	`false` if this is a user-defined table.	`BOOLEAN`
`temporary`	Whether this is a temporary table. Temporary tables are not persisted and only visible within the current connection.	`BOOLEAN`
`has_primary_key`	`true` if this table object defines a `PRIMARY KEY`.	`BOOLEAN`
`estimated_size`	The estimated number of rows in the table.	`BIGINT`
`column_count`	The number of columns defined by this object.	`BIGINT`

Column	Description	Type
`index_count`	The number of indexes associated with this table. This number includes all secondary indexes, as well as internal indexes generated to maintain `PRIMARY KEY` and/or `UNIQUE` constraints.	`BIGINT`
`check_constraint_count`	The number of check constraints active on columns within the table.	`BIGINT`
`sql`	The definition of this object, expressed as SQL `CREATE TABLE`-statement.	`VARCHAR`

The `information_schema.tables` system view provides a more standardized way to obtain metadata about database tables that also includes views. But the resultset returned by `duckdb_tables` contains a few columns that are not included in `information_schema.tables`.

`duckdb_temporary_files`

The `duckdb_temporary_files()` function provides metadata about the temporary files DuckDB has written to disk, to offload data from memory. This function mostly exists for debugging and testing purposes.

Column	Description	Type
`path`	The name of the temporary file.	`VARCHAR`
`size`	The size in bytes of the temporary file.	`BIGINT`

`duckdb_types`

The `duckdb_types()` function provides metadata about the data types available in the DuckDB instance.

Column	Description	Type
`database_name`	The name of the database that contains this schema.	`VARCHAR`
`database_oid`	Internal identifier of the database that contains the data type.	`BIGINT`
`schema_name`	The SQL name of the schema containing the type definition. Always `main`.	`VARCHAR`
`schema_oid`	Internal identifier of the schema object.	`BIGINT`
`type_name`	The name or alias of this data type.	`VARCHAR`
`type_oid`	The internal identifier of the data type object. If `NULL`, then this is an alias of the type (as identified by the value in the `logical_type` column).	`BIGINT`
`type_size`	The number of bytes required to represent a value of this type in memory.	`BIGINT`
`logical_type`	The 'canonical' name of this data type. The same `logical_type` may be referenced by several types having different `type_names`.	`VARCHAR`

Column	Description	Type
`type_category`	The category to which this type belongs. Data types within the same category generally expose similar behavior when values of this type are used in expression. For example, the `NUMERIC` type_category includes integers, decimals, and floating point numbers.	`VARCHAR`
`comment`	A comment created by the `COMMENT ON` statement.	`VARCHAR`
`tags`	A map of string key–value pairs.	`MAP(VARCHAR, VARCHAR)`
`internal`	Whether this is an internal (built-in) or a user object.	`BOOLEAN`
`labels`	Labels for categorizing types. Used for generating the documentation.	`VARCHAR[]`

`duckdb_variables`

The `duckdb_variables()` function provides metadata about the variables available in the DuckDB instance.

Column	Description	Type
`name`	The name of the variable, e.g., `x`.	`VARCHAR`
`value`	The value of the variable, e.g., `12`.	`VARCHAR`
`type`	The type of the variable, e.g., `INTEGER`.	`VARCHAR`

`duckdb_views`

The `duckdb_views()` function provides metadata about the views available in the DuckDB instance.

Column	Description	Type
`database_name`	The name of the database that contains this view.	`VARCHAR`
`database_oid`	Internal identifier of the database that contains this view.	`BIGINT`
`schema_name`	The SQL name of the schema where the view resides.	`VARCHAR`
`schema_oid`	Internal identifier of the schema object that contains the view.	`BIGINT`
`view_name`	The SQL name of the view object.	`VARCHAR`
`view_oid`	The internal identifier of this view object.	`BIGINT`
`comment`	A comment created by the `COMMENT ON` statement.	`VARCHAR`
`tags`	A map of string key–value pairs.	`MAP(VARCHAR, VARCHAR)`
`internal`	`true` if this is an internal (built-in) view, `false` if this is a user-defined view.	`BOOLEAN`
`temporary`	`true` if this is a temporary view. Temporary views are not persistent and are only visible within the current connection.	`BOOLEAN`
`column_count`	The number of columns defined by this view object.	`BIGINT`
`sql`	The definition of this object, expressed as SQL DDL-statement.	`VARCHAR`

The `information_schema.tables` system view provides a more standardized way to obtain metadata about database views that also includes base tables. But the resultset returned by `duckdb_views` contains also definitions of internal view objects as well as a few columns that are not included in `information_schema.tables`.

DuckDB's SQL Dialect

Overview

DuckDB's SQL dialect is based on PostgreSQL. DuckDB tries to closely match PostgreSQL's semantics, however, some use cases require slightly different behavior. For example, interchangeability with data frame libraries necessitates order preservation of inserts to be supported by default. These differences are documented in the pages below.

Indexing

DuckDB uses 1-based indexing except for JSON objects, which use 0-based indexing.

Examples

The index origin is 1 for strings, lists, etc.

```
SELECT list[1] AS element
FROM (SELECT ['first', 'second', 'third'] AS list);
```

```
┌─────────┐
│ element │
│ varchar │
├─────────┤
│ first   │
└─────────┘
```

The index origin is 0 for JSON objects.

```
SELECT json[1] AS element
FROM (SELECT '["first", "second", "third"]'::JSON AS json);
```

```
┌─────────┐
│ element │
│  json   │
├─────────┤
│ "second"│
└─────────┘
```

Friendly SQL

DuckDB offers several advanced SQL features and syntactic sugar to make SQL queries more concise. We refer to these colloquially as "friendly SQL".

> Several of these features are also supported in other systems while some are (currently) exclusive to DuckDB.

Clauses

- Creating tables and inserting data:
  - `CREATE OR REPLACE TABLE`: avoid `DROP TABLE IF EXISTS` statements in scripts.
  - `CREATE TABLE ... AS SELECT` (CTAS): create a new table from the output of a table without manually defining a schema.
  - `INSERT INTO ... BY NAME`: this variant of the `INSERT` statement allows using column names instead of positions.
  - `INSERT OR IGNORE INTO ...`: insert the rows that do not result in a conflict due to `UNIQUE` or `PRIMARY KEY` constraints.
  - `INSERT OR REPLACE INTO ...`: insert the rows that do not result in a conflict due to `UNIQUE` or `PRIMARY KEY` constraints. For those that result in a conflict, replace the columns of the existing row to the new values of the to-be-inserted row.
- Describing tables and computing statistics:
  - `DESCRIBE`: provides a succinct summary of the schema of a table or query.
  - `SUMMARIZE`: returns summary statistics for a table or query.
- Making SQL clauses more compact and readable:
  - `FROM`-first syntax with an optional `SELECT` clause: DuckDB allows queries in the form of `FROM tbl` which selects all columns (performing a `SELECT *` statement).
  - `GROUP BY ALL`: omit the group-by columns by inferring them from the list of attributes in the `SELECT` clause.
  - `ORDER BY ALL`: shorthand to order on all columns (e.g., to ensure deterministic results).
  - `SELECT * EXCLUDE`: the `EXCLUDE` option allows excluding specific columns from the `*` expression.
  - `SELECT * REPLACE`: the `REPLACE` option allows replacing specific columns with different expressions in a `*` expression.
  - `UNION BY NAME`: perform the `UNION` operation along the names of columns (instead of relying on positions).
  - Prefix aliases in the `SELECT` and `FROM` clauses: write `x: 42` instead of `42 AS x` for improved readability.
- Transforming tables:
  - `PIVOT` to turn long tables to wide tables.
  - `UNPIVOT` to turn wide tables to long tables.
- Defining SQL-level variables:
  - `SET VARIABLE`
  - `RESET VARIABLE`

Query Features

- Column aliases in `WHERE`, `GROUP BY`, and `HAVING`. (Note that column aliases cannot be used in the `ON` clause of `JOIN` clauses.)
- `COLUMNS()` expression can be used to execute the same expression on multiple columns:
  - with regular expressions
  - with `EXCLUDE` and `REPLACE`
  - with lambda functions
- Reusable column aliases (also known as "lateral column aliases"), e.g.: `SELECT i + 1 AS j, j + 2 AS k FROM range(0, 3) t(i)`
- Advanced aggregation features for analytical (OLAP) queries:
  - `FILTER` clause
  - `GROUPING SETS`, `GROUP BY CUBE`, `GROUP BY ROLLUP` clauses
- `count()` shorthand for `count(*)`
- `IN` operator for lists and maps
- Specifying column names for common table expressions (`WITH`)
- Specifying column names in the `JOIN` clause
- Using `VALUES` in the `JOIN` clause
- Using `VALUES` in the anchor part of common table expressions

Literals and Identifiers

- Case-insensitivity while maintaining case of entities in the catalog
- Deduplicating identifiers
- Underscores as digit separators in numeric literals

Data Types

- MAP data type
- UNION data type

Data Import

- Auto-detecting the headers and schema of CSV files
- Directly querying CSV files and Parquet files
- Replacement scans:
  - You can load from files using the syntax `FROM 'my.csv'`, `FROM 'my.csv.gz'`, `FROM 'my.parquet'`, etc.
  - In Python, you can access Pandas data frames using `FROM df`.
- Filename expansion (globbing), e.g.: `FROM 'my-data/part-*.parquet'`

Functions and Expressions

- Dot operator for function chaining: `SELECT ('hello').upper()`
- String formatters: the `format()` function with the `fmt` syntax and the `printf() function`
- List comprehensions
- List slicing and indexing from the back (`[-1]`)
- String slicing
- `STRUCT.*` notation
- Creating `LIST` using square brackets
- Simple `LIST` and `STRUCT` creation
- Updating the schema of `STRUCT`s

Join Types

- `ASOF` joins
- `LATERAL` joins
- `POSITIONAL` joins

Trailing Commas

DuckDB allows trailing commas, both when listing entities (e.g., column and table names) and when constructing `LIST` items. For example, the following query works:

```
SELECT
    42 AS x,
    ['a', 'b', 'c',] AS y,
    'hello world' AS z,
;
```

”Top-N in Group” Queries

Computing the ”top-N rows in a group” ordered by some criteria is a common task in SQL that unfortunately often requires a complex query involving window functions and/or subqueries.

To aid in this, DuckDB provides the aggregate functions `max(arg, n)`, `min(arg, n)`, `arg_max(arg, val, n)`, `arg_min(arg, val, n)`, `max_by(arg, val, n)` and `min_by(arg, val, n)` to efficiently return the ”top” n rows in a group based on a specific column in either ascending or descending order.

For example, let's use the following table:

```
SELECT * FROM t1;
```

grp varchar	val int32
a	2
a	1
b	5
b	4
a	3
b	6

We want to get a list of the top-3 `val` values in each group `grp`. The conventional way to do this is to use a window function in a subquery:

```
SELECT array_agg(rs.val), rs.grp
FROM
    (SELECT val, grp, row_number() OVER (PARTITION BY grp ORDER BY val DESC) AS rid
    FROM t1 ORDER BY val DESC) AS rs
WHERE rid < 4
GROUP BY rs.grp;
```

array_agg(rs.val) int32[]	grp varchar
[3, 2, 1]	a
[6, 5, 4]	b

But in DuckDB, we can do this much more concisely (and efficiently!):

```
SELECT max(val, 3) FROM t1 GROUP BY grp;
```

max(val, 3) int32[]
[3, 2, 1]
[6, 5, 4]

Related Blog Posts

- “Friendlier SQL with DuckDB” blog post
- “Even Friendlier SQL with DuckDB” blog post
- “SQL Gymnastics: Bending SQL into Flexible New Shapes” blog post

Keywords and Identifiers

Identifiers

Similarly to other SQL dialects and programming languages, identifiers in DuckDB's SQL are subject to several rules.

- Unquoted identifiers need to conform to a number of rules:
  - They must not be a reserved keyword (see `duckdb_keywords()`), e.g., `SELECT 123 AS SELECT` will fail.
  - They must not start with a number or special character, e.g., `SELECT 123 AS 1col` is invalid.
  - They cannot contain whitespaces (including tabs and newline characters).
- Identifiers can be quoted using double-quote characters (`"`). Quoted identifiers can use any keyword, whitespace or special character, e.g., `"SELECT"` and `" § 🦆 ¶ "` are valid identifiers.
- Double quotes can be escaped by repeating the quote character, e.g., to create an identifier named `IDENTIFIER "X"`, use `"IDENTIFIER ""X"""`.

Deduplicating Identifiers

In some cases, duplicate identifiers can occur, e.g., column names may conflict when unnesting a nested data structure. In these cases, DuckDB automatically deduplicates column names by renaming them according to the following rules:

- For a column named <name>{:.language-sql .highlight}, the first instance is not renamed.
- Subsequent instances are renamed to <name>_<count>{:.language-sql .highlight}, where <count>{:.language-sql .highlight} starts at 1.

For example:

```
SELECT *
FROM (SELECT UNNEST({'a': 42, 'b': {'a': 88, 'b': 99}}, recursive := true));
```

a	a_1	b
42	88	99

Database Names

Database names are subject to the rules for identifiers.

Additionally, it is best practice to avoid DuckDB's two internal database schema names, `system` and `temp`. By default, persistent databases are named after their filename without the extension. Therefore, the filenames `system.db` and `temp.db` (as well as `system.duckdb` and `temp.duckdb`) result in the database names `system` and `temp`, respectively. If you need to attach to a database that has one of these names, use an alias, e.g.:

```
ATTACH 'temp.db' AS temp2;
USE temp2;
```

Rules for Case-Sensitivity

Keywords and Function Names

SQL keywords and function names are case-insensitive in DuckDB.

For example, the following two queries are equivalent:

```
select COS(Pi()) as CosineOfPi;
SELECT cos(pi()) AS CosineOfPi;
```

CosineOfPi
-1.0

Case-Sensitivity of Identifiers

Identifiers in DuckDB are always case-insensitive, similarly to PostgreSQL. However, unlike PostgreSQL (and some other major SQL implementations), DuckDB also treats quoted identifiers as case-insensitive.

Comparison of identifiers: Case-insensitivity is implemented using an ASCII-based comparison: `col_A` and `col_a` are equal but `col_á` is not equal to them.

```
SELECT col_A FROM (SELECT 'x' AS col_a); -- succeeds
SELECT col_á FROM (SELECT 'x' AS col_a); -- fails
```

Preserving cases: While DuckDB treats identifiers in a case-insensitive manner, it preservers the cases of these identifiers. That is, each character's case (uppercase/lowercase) is maintained as originally specified by the user even if a query uses different cases when referring to the identifier. For example:

```
CREATE TABLE tbl AS SELECT cos(pi()) AS CosineOfPi;
SELECT cosineofpi FROM tbl;
```

CosineOfPi
-1.0

To change this behavior, set the `preserve_identifier_case` configuration option to `false`.

Case-Sensitivity of Keys in Nested Data Structures

The keys of MAPs are case-sensitive:

```
SELECT MAP(['key1'], [1]) = MAP(['KEY1'], [1]) AS equal;
```

```
false
```

The keys of UNIONs and STRUCTs are case-insensitive:

```
SELECT {'key1': 1} = {'KEY1': 1} AS equal;
```

```
true
```

```
SELECT union_value(key1 := 1) = union_value(KEY1 := 1) as equal;
```

```
true
```

Handling Conflicts

In case of a conflict, when the same identifier is spelt with different cases, one will be selected randomly. For example:

```
CREATE TABLE t1 (idfield INTEGER, x INTEGER);
CREATE TABLE t2 (IdField INTEGER, y INTEGER);
INSERT INTO t1 VALUES (1, 123);
INSERT INTO t2 VALUES (1, 456);
SELECT * FROM t1 NATURAL JOIN t2;
```

idfield	x	y
1	123	456

Disabling Preserving Cases

With the `preserve_identifier_case` configuration option set to `false`, all identifiers are turned into lowercase:

```
SET preserve_identifier_case = false;
CREATE TABLE tbl AS SELECT cos(pi()) AS CosineOfPi;
SELECT CosineOfPi FROM tbl;
```

cosineofpi
-1.0

Order Preservation

For many operations, DuckDB preserves the order of rows, similarly to data frame libraries such as Pandas.

Example

Take the following table for example:

```
CREATE TABLE tbl AS
    SELECT *
    FROM (VALUES (1, 'a'), (2, 'b'), (3, 'c')) t(x, y);

SELECT *
FROM tbl;
```

x	y
1	a
2	b
3	c

Let's take the following query that returns the rows where x is an odd number:

```
SELECT *
FROM tbl
WHERE x % 2 == 1;
```

x	y
1	a
3	c

Because the row `(1, 'a')` occurs before `(3, 'c')` in the original table, it is guaranteed to come before that row in this table too.

Clauses

The following clauses guarantee that the original row order is preserved:

- `COPY` (see Insertion Order)
- `FROM` with a single table
- `LIMIT`
- `OFFSET`
- `SELECT`
- `UNION ALL`
- `WHERE`
- Window functions with an empty `OVER` clause
- Common table expressions and table subqueries as long as they only contains the aforementioned components

> **Tip.** `row_number() OVER ()` allows turning the original row order into an explicit column that can be referenced in the operations that don't preserve row order by default. On materialized tables, the `rowid` pseudo-column can be used to the same effect.

The following operations **do not** guarantee that the row order is preserved:

- `FROM` with multiple tables and/or subqueries
- `JOIN`
- `UNION`
- `USING SAMPLE`
- `GROUP BY` (in particular, the output order is undefined and the order in which rows are fed into order-sensitive aggregate functions is undefined unless explicitly specified in the aggregate function)
- `ORDER BY` (specifically, `ORDER BY` may not use a stable algorithm)
- Scalar subqueries

Insertion Order

By default, the following components preserve insertion order:

- CSV reader (`read_csv` function)
- JSON reader (`read_json` function)
- Parquet reader (`read_parquet` function)

Preservation of insertion order is controlled by the `preserve_insertion_order` configuration option. This setting is `true` by default, indicating that the order should be preserved. To change this setting, use:

```
SET preserve_insertion_order = false;
```

PostgreSQL Compatibility

DuckDB's SQL dialect closely follows the conventions of the PostgreSQL dialect. The few exceptions to this are listed on this page.

Floating-Point Arithmetic

DuckDB and PostgreSQL handle floating-point arithmetic differently for division by zero. DuckDB conforms to the IEEE Standard for Floating-Point Arithmetic (IEEE 754) for both division by zero and operations involving infinity values. PostgreSQL returns an error for division by zero but aligns with IEEE 754 for handling infinity values. To show the differences, run the following SQL queries:

```
SELECT 1.0 / 0.0 AS x;
SELECT 0.0 / 0.0 AS x;
SELECT -1.0 / 0.0 AS x;
SELECT 'Infinity'::FLOAT / 'Infinity'::FLOAT AS x;
SELECT 1.0 / 'Infinity'::FLOAT AS x;
SELECT 'Infinity'::FLOAT - 'Infinity'::FLOAT AS x;
SELECT 'Infinity'::FLOAT - 1.0 AS x;
```

Expression	PostgreSQL	DuckDB	IEEE 754
1.0 / 0.0	error	Infinity	Infinity
0.0 / 0.0	error	NaN	NaN
-1.0 / 0.0	error	-Infinity	-Infinity
'Infinity' / 'Infinity'	NaN	NaN	NaN
1.0 / 'Infinity'	0.0	0.0	0.0
'Infinity' - 'Infinity'	NaN	NaN	NaN
'Infinity' - 1.0	Infinity	Infinity	Infinity

Division on Integers

When computing division on integers, PostgreSQL performs integer division, while DuckDB performs float division:

```
SELECT 1 / 2 AS x;
```

PostgreSQL returns `0`, while DuckDB returns `0.5`.

To perform integer division in DuckDB, use the `//` operator:

```
SELECT 1 // 2 AS x;
```

This returns `0`.

`UNION` of Boolean and Integer Values

The following query fails in PostgreSQL but successfully completes in DuckDB:

```
SELECT true AS x
UNION
SELECT 2;
```

PostgreSQL returns an error:

```
ERROR:  UNION types boolean and integer cannot be matched
```

DuckDB performs an enforced cast, therefore, it completes the query and returns the following:

x
1
2

Implicit Casting on Equality Checks

DuckDB performs implicit casting on equality checks, e.g., converting strings to numeric and boolean values. Therefore, there are several instances, where PostgreSQL throws an error while DuckDB successfully computes the result:

Expression	PostgreSQL	DuckDB
'1.1' = 1	error	true
'1.1' = 1.1	true	true
1 = 1.1	false	false
true = 'true'	true	true
true = 1	error	true
'true' = 1	error	error

Case Sensitivity for Quoted Identifiers

PostgreSQL is case-insensitive. The way PostgreSQL achieves case insensitivity is by lowercasing unquoted identifiers within SQL, whereas quoting preserves case, e.g., the following command creates a table named `mytable` but tries to query for `MyTaBLe` because quotes preserve the case.

```
CREATE TABLE MyTaBLe (x INTEGER);
SELECT * FROM "MyTaBLe";
```

```
ERROR:  relation "MyTaBLe" does not exist
```

PostgreSQL does not only treat quoted identifiers as case-sensitive, PostgreSQL treats all identifiers as case-sensitive, e.g., this also does not work:

```
CREATE TABLE "PreservedCase" (x INTEGER);
SELECT * FROM PreservedCase;
```

```
ERROR:  relation "preservedcase" does not exist
```

Therefore, case-insensitivity in PostgreSQL only works if you never use quoted identifiers with different cases.

For DuckDB, this behavior was problematic when interfacing with other tools (e.g., Parquet, Pandas) that are case-sensitive by default – since all identifiers would be lowercased all the time. Therefore, DuckDB achieves case insensitivity by making identifiers fully case insensitive throughout the system but *preserving their case*.

In DuckDB, the scripts above complete successfully:

```
CREATE TABLE MyTaBLe (x INTEGER);
SELECT * FROM "MyTaBLe";
CREATE TABLE "PreservedCase" (x INTEGER);
SELECT * FROM PreservedCase;
SELECT table_name FROM duckdb_tables();
```

table_name
MyTaBLe
PreservedCase

PostgreSQL's behavior of lowercasing identifiers is accessible using the `preserve_identifier_case` option:

```
SET preserve_identifier_case = false;
CREATE TABLE MyTaBLe (x INTEGER);
SELECT table_name FROM duckdb_tables();
```

table_name
mytable

However, the case insensitive matching in the system for identifiers cannot be turned off.

Using Double Equality Sign for Comparison

DuckDB supports both = and == for quality comparison, while PostgreSQL only supports =.

```
SELECT 1 == 1 AS t;
```

DuckDB returns `true`, while PostgreSQL returns:

```
postgres=# SELECT 1 == 1 AS t;
ERROR:  operator does not exist: integer == integer
LINE 1: SELECT 1 == 1 AS t;
```

Note that the use of == is not encouraged due to its limited portability.

Vacuuming Tables

In PostgreSQL, the VACUUM statement garbage collects tables and analyzes tables. In DuckDB, the VACUUM statement is only used to rebuild statistics. For instruction on reclaiming space, refer to the "Reclaiming space" page.

Strings

Since version 1.3.0, DuckDB escapes characters such as `'` in strings serialized in nested data structures. PostgreSQL does not do this.

For an example, run:

```
SELECT ARRAY[''''];
```

PostgreSQL returns:

```
{'}
```

DuckDB returns:

```
['\'']
```

Functions

`regexp_extract` Function

Unlike PostgreSQL's `regexp_substr` function, DuckDB's `regexp_extract` returns empty strings instead of NULLs when there is no match.

`to_date` Function

DuckDB does not support the `to_date` PostgreSQL date formatting function. Instead, please use the `strptime` function.

Resolution of Type Names in the Schema

For `CREATE TABLE` statements, DuckDB attempts to resolve type names in the schema where a table is created. For example:

```
CREATE SCHEMA myschema;
CREATE TYPE myschema.mytype AS ENUM ('as', 'df');
CREATE TABLE myschema.mytable (v mytype);
```

PostgreSQL returns an error on the last statement:

```
ERROR:  type "mytype" does not exist
LINE 1: CREATE TABLE myschema.mytable (v mytype);
```

DuckDB runs the statement and creates the table successfully, confirmed by the following query:

```
DESCRIBE myschema.mytable;
```

column_name	column_type	null	key	default	extra
v	ENUM('as', 'df')	YES	NULL	NULL	NULL

Exploiting Functional Dependencies for `GROUP BY`

PostgreSQL can exploit functional dependencies, such as `i -> j` in the following query:

```
CREATE TABLE tbl (i INTEGER, j INTEGER, PRIMARY KEY (i));
SELECT j
FROM tbl
GROUP BY i;
```

PostgreSQL runs the query.

DuckDB fails:

```
Binder Error:
column "j" must appear in the GROUP BY clause or must be part of an aggregate function.
Either add it to the GROUP BY list, or use "ANY_VALUE(j)" if the exact value of "j" is not important.
```

To work around this, add the other attributes or use the `GROUP BY ALL` clause.

Behavior of Regular Expression Match Operators

PostgreSQL supports the POSIX regular expression matching operators `~` (case-sensitive partial regex matching) and `~*` (case-insensitive partial regex matching) as well as their negated variants, `!~` and `!~*`, respectively.

In DuckDB, `~` is equivalent to `regexp_full_match` and `!~` is equivalent to `NOT regexp_full_match`. The operators `~*` and `!~*` are not supported.

The table below shows that the correspondence between these functions in PostgreSQL and DuckDB is almost non-existent. We recommend avoiding the POSIX regular expression matching operators in DuckDB.

Expression	PostgreSQL	DuckDB
`'aaa' ~ '(a|b)'`	true	false
`'AAA' ~* '(a|b)'`	true	error
`'aaa' !~ '(a|b)'`	false	true
`'AAA' !~* '(a|b)'`	false	error

SQL Quirks

Like all programming languages and libraries, DuckDB has its share of idiosyncrasies and inconsistencies.
Some are vestiges of our feathered friend's evolution; others are inevitable because we strive to adhere to the SQL Standard and specifically to PostgreSQL's dialect (see the "PostgreSQL Compatibility" page for exceptions). The rest may simply come down to different preferences, or we may even agree on what *should* be done but just haven't gotten around to it yet.

Acknowledging these quirks is the best we can do, which is why we have compiled below a list of examples.

Aggregating Empty Groups

On empty groups, the aggregate functions `sum`, `list`, and `string_agg` all return `NULL` instead of `0`, `[]` and `''`, respectively. This is dictated by the SQL Standard and obeyed by all SQL implementations we know. This behavior is inherited by the list aggregate `list_sum`, but not by the DuckDB original `list_dot_product` which returns `0` on empty lists.

0 vs. 1-Based Indexing

To comply with standard SQL, one-based indexing is used almost everywhere, e.g., array and string indexing and slicing, and window functions (`row_number`, `rank`, `dense_rank`). However, similarly to PostgreSQL, JSON features use a zero-based indexing.

Types

`UINT8` vs. `INT8`

`UINT8` and `INT8` are aliases to integer types of different widths:

- `UINT8` corresponds to `UTINYINT` because it's an *8-bit* unsigned integer
- `INT8` corresponds to `BIGINT` because it's an *8-byte* signed integer

Explanation: the `n` in the numeric type `INTn` and `UINTn` denote the width of the number in either bytes or bits. `INT1`, `INT2`, `INT4` correspond to the number of bytes, while `INT16`, `INT32` and `INT64` correpsond to the number of bits. The same applies to `UINT` values. However, the value `n = 8` is a valid choice for both the number of bits and bytes. For unsigned values, `UINT8` corresponds to `UTINYINT` (8 bits). For signed values, `INT8` corresponds to `BIGINT` (8 bytes).

Expressions

Results That May Surprise You

Expression	Result	Note
`-2^2`	`4.0`	PostgreSQL compatibility means the unary minus has higher precedence than the exponentiation operator. Use additional parentheses, e.g., `-(2^2)` or the pow function, e.g., `-pow(2, 2)`, to avoid mistakes.
`'t' = true`	`true`	Compatible with PostgreSQL.
`1 = '1'`	`true`	Compatible with PostgreSQL.
`1 = ' 1'`	`true`	Compatible with PostgreSQL.
`1 = '01'`	`true`	Compatible with PostgreSQL.
`1 = ' 01 '`	`true`	Compatible with PostgreSQL.
`1 = true`	`true`	Not compatible with PostgreSQL.
`1 = '1.1'`	`true`	Not compatible with PostgreSQL.
`1 IN (0, NULL)`	`NULL`	Makes sense if you think of the `NULL`s in the input and output as `UNKNOWN`.
`1 in [0, NULL]`	`false`	
`concat('abc', NULL)`	`abc`	Compatible with PostgreSQL. `list_concat` behaves similarly.
`'abc' || NULL`	`NULL`	

NaN Values

`'NaN'::FLOAT = 'NaN'::FLOAT` and `'NaN'::FLOAT > 3` violate IEEE-754 but mean floating point data types have a total order, like all other data types (beware the consequences for `greatest` / `least`).

age Function

`age(x)` is `current_date - x` instead of `current_timestamp - x`. Another quirk inherited from PostgreSQL.

Extract Functions

`list_extract` / `map_extract` return `NULL` on non-existing keys. `struct_extract` throws an error because keys of structs are like columns.

Clauses

Automatic Column Deduplication in SELECT

Column names are deduplicated with the first occurrence shadowing the others:

```
CREATE TABLE tbl AS SELECT 1 AS a;
SELECT a FROM (SELECT *, 2 AS a FROM tbl);
```

a
1

Case Insensitivity for SELECTing Columns

Due to case-insensitivity, it's not possible to use `SELECT a FROM 'file.parquet'` when a column called `A` appears before the desired column `a` in `file.parquet`.

USING SAMPLE

The `USING SAMPLE` clause is syntactically placed after the `WHERE` and `GROUP BY` clauses (same as the `LIMIT` clause) but is semantically applied before both (unlike the `LIMIT` clause).

Samples

Samples are used to randomly select a subset of a dataset.

Examples

Select a sample of exactly 5 rows from `tbl` using `reservoir` sampling:

```
SELECT *
FROM tbl
USING SAMPLE 5;
```

Select a sample of *approximately* 10% of the table using `system` sampling:

```
SELECT *
FROM tbl
USING SAMPLE 10%;
```

> **Warning.** By default, when you specify a percentage, each *vector* is included in the sample with that probability. If your table contains fewer than ~10k rows, it makes sense to specify the `bernoulli` sampling option instead, which applies the probability to each row independently. Even then, you'll sometimes get more and sometimes less than the specified percentage of the number of rows, but it is much less likely that you get no rows at all. To get exactly 10% of rows (up to rounding), you must use the `reservoir` sampling option.

Select a sample of *approximately* 10% of the table using `bernoulli` sampling:

```
SELECT *
FROM tbl
USING SAMPLE 10 PERCENT (bernoulli);
```

Select a sample of *exactly* 10% (up to rounding) of the table using `reservoir` sampling:

```
SELECT *
FROM tbl
USING SAMPLE 10 PERCENT (reservoir);
```

Select a sample of *exactly* 50 rows of the table using reservoir sampling with a fixed seed (100):

```
SELECT *
FROM tbl
USING SAMPLE reservoir(50 ROWS)
REPEATABLE (100);
```

Select a sample of *approximately* 20% of the table using `system` sampling with a fixed seed (377):

```
SELECT *
FROM tbl
USING SAMPLE 20% (system, 377);
```

Select a sample of *approximately* 20% of `tbl` **before** the join with `tbl2`:

```
SELECT *
FROM tbl TABLESAMPLE reservoir(20%), tbl2
WHERE tbl.i = tbl2.i;
```

Select a sample of *approximately* 20% of `tbl` **after** the join with `tbl2`:

```
SELECT *
FROM tbl, tbl2
WHERE tbl.i = tbl2.i
USING SAMPLE reservoir(20%);
```

Syntax

Samples allow you to randomly extract a subset of a dataset. Samples are useful for exploring a dataset faster, as often you might not be interested in the exact answers to queries, but only in rough indications of what the data looks like and what is in the data. Samples allow you to get approximate answers to queries faster, as they reduce the amount of data that needs to pass through the query engine.

DuckDB supports three different types of sampling methods: `reservoir`, `bernoulli` and `system`. By default, DuckDB uses `reservoir` sampling when an exact number of rows is sampled, and `system` sampling when a percentage is specified. The sampling methods are described in detail below.

Samples require a *sample size*, which is an indication of how many elements will be sampled from the total population. Samples can either be given as a percentage (`10%` or `10 PERCENT`) or as a fixed number of rows (`10` or `10 ROWS`). All three sampling methods support sampling over a percentage, but **only** reservoir sampling supports sampling a fixed number of rows.

Samples are probablistic, that is to say, samples can be different between runs *unless* the seed is specifically specified. Specifying the seed *only* guarantees that the sample is the same if multi-threading is not enabled (i.e., `SET threads = 1`). In the case of multiple threads running over a sample, samples are not necessarily consistent even with a fixed seed.

`reservoir`

Reservoir sampling is a stream sampling technique that selects a random sample by keeping a *reservoir* of size equal to the sample size, and randomly replacing elements as more elements come in. Reservoir sampling allows us to specify *exactly* how many elements we want in the resulting sample (by selecting the size of the reservoir). As a result, reservoir sampling *always* outputs the same amount of elements, unlike system and bernoulli sampling.

Reservoir sampling is only recommended for small sample sizes, and is not recommended for use with percentages. That is because reservoir sampling needs to materialize the entire sample and randomly replace tuples within the materialized sample. The larger the sample size, the higher the performance hit incurred by this process.

Reservoir sampling also incurs an additional performance penalty when multi-processing is used, since the reservoir is to be shared amongst the different threads to ensure unbiased sampling. This is not a big problem when the reservoir is very small, but becomes costly when the sample is large.

> **Best practice.** Avoid using reservoir sampling with large sample sizes if possible. Reservoir sampling requires the entire sample to be materialized in memory.

`bernoulli`

Bernoulli sampling can only be used when a sampling percentage is specified. It is rather straightforward: every row in the underlying table is included with a chance equal to the specified percentage. As a result, bernoulli sampling can return a different number of tuples even if the same percentage is specified. The *expected* number of rows is equal to the specified percentage of the table, but there will be some *variance*.

Because bernoulli sampling is completely independent (there is no shared state), there is no penalty for using bernoulli sampling together with multiple threads.

`system`

System sampling is a variant of bernoulli sampling with one crucial difference: every *vector* is included with a chance equal to the sampling percentage. This is a form of cluster sampling. System sampling is more efficient than bernoulli sampling, as no per-tuple selections have to be performed.

The *expected* number of rows is still equal to the specified percentage of the table, but the *variance* is `vectorSize` times higher. As such, system sampling is not suitable for data sets with fewer than ~10k rows, where it can happen that all rows will be filtered out, or all the data will be included, even when you ask for `50 PERCENT`.

Table Samples

The `TABLESAMPLE` and `USING SAMPLE` clauses are identical in terms of syntax and effect, with one important difference: tablesamples sample directly from the table for which they are specified, whereas the sample clause samples after the entire from clause has been resolved. This is relevant when there are joins present in the query plan.

The `TABLESAMPLE` clause is essentially equivalent to creating a subquery with the `USING SAMPLE` clause, i.e., the following two queries are identical:

Sample 20% of `tbl` **before** the join:

```
SELECT *
FROM
    tbl TABLESAMPLE reservoir(20%),
    tbl2
WHERE tbl.i = tbl2.i;
```

Sample 20% of `tbl` **before** the join:

```
SELECT *
FROM
    (SELECT * FROM tbl USING SAMPLE reservoir(20%)) tbl,
    tbl2
WHERE tbl.i = tbl2.i;
```

Sample 20% **after** the join (i.e., sample 20% of the join result):

```
SELECT *
FROM tbl, tbl2
WHERE tbl.i = tbl2.i
USING SAMPLE reservoir(20%);
```

Configuration

Configuration

DuckDB has a number of configuration options that can be used to change the behavior of the system.

The configuration options can be set using either the SET statement or the PRAGMA statement. They can be reset to their original values using the RESET statement.

The values of configuration options can be queried via the `current_setting()` scalar function or using the `duckdb_settings()` table function. For example:

```
SELECT current_setting('memory_limit') AS memlimit;
```

Or:

```
SELECT value AS memlimit
FROM duckdb_settings()
WHERE name = 'memory_limit';
```

Examples

Set the memory limit of the system to 10 GB.

```
SET memory_limit = '10GB';
```

Configure the system to use 1 thread.

```
SET threads TO 1;
```

Enable printing of a progress bar during long-running queries.

```
SET enable_progress_bar = true;
```

Set the default null order to `NULLS LAST`.

```
SET default_null_order = 'nulls_last';
```

Return the current value of a specific setting.

```
SELECT current_setting('threads') AS threads;
```

threads
10

Query a specific setting.

```
SELECT *
FROM duckdb_settings()
WHERE name = 'threads';
```

name	value	description	input_type	scope
threads	1	The number of total threads used by the system.	BIGINT	GLOBAL

Show a list of all available settings.

```
SELECT *
FROM duckdb_settings();
```

Reset the memory limit of the system back to the default.

```
RESET memory_limit;
```

Secrets Manager

DuckDB has a Secrets manager, which provides a unified user interface for secrets across all backends (e.g., AWS S3) that use them.

Configuration Reference

Configuration options come with different default scopes: `GLOBAL` and `LOCAL`. Below is a list of all available configuration options by scope.

Global Configuration Options

Name	Description	Type	Default value
`access_mode`	Access mode of the database (`AUTOMATIC`, `READ_ONLY` or `READ_WRITE`)	`VARCHAR`	`automatic`
`allocator_background_threads`	Whether to enable the allocator background thread.	`BOOLEAN`	`false`
`allocator_bulk_deallocation_flush_threshold`	If a bulk deallocation larger than this occurs, flush outstanding allocations.	`VARCHAR`	`512.0 MiB`
`allocator_flush_threshold`	Peak allocation threshold at which to flush the allocator after completing a task.	`VARCHAR`	`128.0 MiB`
`allow_community_extensions`	Allow to load community built extensions	`BOOLEAN`	`true`
`allow_extensions_metadata_mismatch`	Allow to load extensions with not compatible metadata	`BOOLEAN`	`false`
`allow_persistent_secrets`	Allow the creation of persistent secrets, that are stored and loaded on restarts	`BOOLEAN`	`true`
`allow_unredacted_secrets`	Allow printing unredacted secrets	`BOOLEAN`	`false`
`allow_unsigned_extensions`	Allow to load extensions with invalid or missing signatures	`BOOLEAN`	`false`
`allowed_directories`	List of directories/prefixes that are ALWAYS allowed to be queried - even when enable_external_access is false	`VARCHAR[]`	`[]`

Name	Description	Type	Default value
allowed_paths	List of files that are ALWAYS allowed to be queried - even when enable_external_access is false	VARCHAR[]	[]
arrow_large_buffer_size	Whether Arrow buffers for strings, blobs, uuids and bits should be exported using large buffers	BOOLEAN	false
arrow_lossless_conversion	Whenever a DuckDB type does not have a clear native or canonical extension match in Arrow, export the types with a duckdb.type_name extension name.	BOOLEAN	false
arrow_output_list_view	Whether export to Arrow format should use ListView as the physical layout for LIST columns	BOOLEAN	false
autoinstall_extension_repository	Overrides the custom endpoint for extension installation on autoloading	VARCHAR	
autoinstall_known_extensions	Whether known extensions are allowed to be automatically installed when a query depends on them	BOOLEAN	false
autoload_known_extensions	Whether known extensions are allowed to be automatically loaded when a query depends on them	BOOLEAN	false
binary_as_string	In Parquet files, interpret binary data as a string.	BOOLEAN	
ca_cert_file	Path to a custom certificate file for self-signed certificates.	VARCHAR	
catalog_error_max_schemas	The maximum number of schemas the system will scan for "did you mean..." style errors in the catalog	UBIGINT	100
checkpoint_threshold, wal_autocheckpoint	The WAL size threshold at which to automatically trigger a checkpoint (e.g., 1GB)	VARCHAR	16.0 MiB
custom_extension_repository	Overrides the custom endpoint for remote extension installation	VARCHAR	
custom_user_agent	Metadata from DuckDB callers	VARCHAR	
default_block_size	The default block size for new duckdb database files (new as-in, they do not yet exist).	UBIGINT	262144
default_collation	The collation setting used when none is specified	VARCHAR	
default_null_order, null_order	NULL ordering used when none is specified (NULLS_FIRST or NULLS_LAST)	VARCHAR	NULLS_LAST
default_order	The order type used when none is specified (ASC or DESC)	VARCHAR	ASC
default_secret_storage	Allows switching the default storage for secrets	VARCHAR	local_file
disable_parquet_prefetching	Disable the prefetching mechanism in Parquet	BOOLEAN	false
disabled_compression_methods	Disable a specific set of compression methods (comma separated)	VARCHAR	
disabled_filesystems	Disable specific file systems preventing access (e.g., LocalFileSystem)	VARCHAR	
disabled_log_types	Sets the list of disabled loggers	VARCHAR	
duckdb_api	DuckDB API surface	VARCHAR	cli
enable_external_access	Allow the database to access external state (through e.g., loading/installing modules, COPY TO/FROM, CSV readers, pandas replacement scans, etc)	BOOLEAN	true

Name	Description	Type	Default value
enable_external_file_cache	Allow the database to cache external files (e.g., Parquet) in memory.	BOOLEAN	true
enable_fsst_vectors	Allow scans on FSST compressed segments to emit compressed vectors to utilize late decompression	BOOLEAN	false
enable_geoparquet_conversion	Attempt to decode/encode geometry data in/as GeoParquet files if the spatial extension is present.	BOOLEAN	true
enable_http_metadata_cache	Whether or not the global http metadata is used to cache HTTP metadata	BOOLEAN	false
enable_logging	Enables the logger	BOOLEAN	0
enable_macro_dependencies	Enable created MACROs to create dependencies on the referenced objects (such as tables)	BOOLEAN	false
enable_object_cache	[PLACEHOLDER] Legacy setting - does nothing	BOOLEAN	NULL
enable_server_cert_verification	Enable server side certificate verification.	BOOLEAN	false
enable_view_dependencies	Enable created VIEWs to create dependencies on the referenced objects (such as tables)	BOOLEAN	false
enabled_log_types	Sets the list of enabled loggers	VARCHAR	
extension_directory	Set the directory to store extensions in	VARCHAR	
external_threads	The number of external threads that work on DuckDB tasks.	UBIGINT	1
force_download	Forces upfront download of file	BOOLEAN	false
http_keep_alive	Keep alive connections. Setting this to false can help when running into connection failures	BOOLEAN	true
http_proxy_password	Password for HTTP proxy	VARCHAR	
http_proxy_username	Username for HTTP proxy	VARCHAR	
http_proxy	HTTP proxy host	VARCHAR	
http_retries	HTTP retries on I/O error	UBIGINT	3
http_retry_backoff	Backoff factor for exponentially increasing retry wait time	FLOAT	4
http_retry_wait_ms	Time between retries	UBIGINT	100
http_timeout	HTTP timeout read/write/connection/retry (in seconds)	UBIGINT	30
immediate_transaction_mode	Whether transactions should be started lazily when needed, or immediately when BEGIN TRANSACTION is called	BOOLEAN	false
index_scan_max_count	The maximum index scan count sets a threshold for index scans. If fewer than MAX(index_scan_max_count, index_scan_percentage * total_row_count) rows match, we perform an index scan instead of a table scan.	UBIGINT	2048
index_scan_percentage	The index scan percentage sets a threshold for index scans. If fewer than MAX(index_scan_max_count, index_scan_percentage * total_row_count) rows match, we perform an index scan instead of a table scan.	DOUBLE	0.001
lock_configuration	Whether or not the configuration can be altered	BOOLEAN	false
logging_level	The log level which will be recorded in the log	VARCHAR	INFO
logging_mode	Enables the logger	VARCHAR	LEVEL_ONLY
logging_storage	Set the logging storage (memory/stdout/file)	VARCHAR	memory

Name	Description	Type	Default value
`max_memory`, `memory_limit`	The maximum memory of the system (e.g., 1GB)	`VARCHAR`	80% of RAM
`max_temp_directory_size`	The maximum amount of data stored inside the 'temp_directory' (when set) (e.g., 1GB)	`VARCHAR`	`90% of available disk space`
`max_vacuum_tasks`	The maximum vacuum tasks to schedule during a checkpoint.	`UBIGINT`	`100`
`old_implicit_casting`	Allow implicit casting to/from VARCHAR	`BOOLEAN`	`false`
`parquet_metadata_cache`	Cache Parquet metadata - useful when reading the same files multiple times	`BOOLEAN`	`false`
`password`	The password to use. Ignored for legacy compatibility.	`VARCHAR`	`NULL`
`prefetch_all_parquet_files`	Use the prefetching mechanism for all types of parquet files	`BOOLEAN`	`false`
`preserve_insertion_order`	Whether or not to preserve insertion order. If set to false the system is allowed to re-order any results that do not contain ORDER BY clauses.	`BOOLEAN`	`true`
`produce_arrow_string_view`	Whether strings should be produced by DuckDB in Utf8View format instead of Utf8	`BOOLEAN`	`false`
`s3_access_key_id`	S3 Access Key ID	`VARCHAR`	
`s3_endpoint`	S3 Endpoint	`VARCHAR`	
`s3_kms_key_id`	S3 KMS Key ID	`VARCHAR`	
`s3_region`	S3 Region	`VARCHAR`	`us-east-1`
`s3_secret_access_key`	S3 Access Key	`VARCHAR`	
`s3_session_token`	S3 Session Token	`VARCHAR`	
`s3_uploader_max_filesize`	S3 Uploader max filesize (between 50GB and 5TB)	`VARCHAR`	`800GB`
`s3_uploader_max_parts_per_file`	S3 Uploader max parts per file (between 1 and 10000)	`UBIGINT`	`10000`
`s3_uploader_thread_limit`	S3 Uploader global thread limit	`UBIGINT`	`50`
`s3_url_compatibility_mode`	Disable Globs and Query Parameters on S3 URLs	`BOOLEAN`	`false`
`s3_url_style`	S3 URL style	`VARCHAR`	`vhost`
`s3_use_ssl`	S3 use SSL	`BOOLEAN`	`true`
`scheduler_process_partial`	Partially process tasks before rescheduling - allows for more scheduler fairness between separate queries	`BOOLEAN`	`false`
`secret_directory`	Set the directory to which persistent secrets are stored	`VARCHAR`	`~/.duckdb/stored_secrets`
`storage_compatibility_version`	Serialize on checkpoint with compatibility for a given duckdb version	`VARCHAR`	`v0.10.2`
`temp_directory`	Set the directory to which to write temp files	`VARCHAR`	`<database_name>.tmp` or `.tmp` (in in-memory mode)

Name	Description	Type	Default value
`threads`, `worker_threads`	The number of total threads used by the system.	`BIGINT`	# CPU cores
`username`, `user`	The username to use. Ignored for legacy compatibility.	`VARCHAR`	`NULL`
`zstd_min_string_length`	The (average) length at which to enable ZSTD compression, defaults to 4096	`UBIGINT`	`4096`

Local Configuration Options

Name	Description	Type	Default value
`asof_loop_join_threshold`	The maximum number of rows we need on the left side of an ASOF join to use a nested loop join	`UBIGINT`	`64`

Name	Description	Type	Default value
custom_profiling_settings	Accepts a JSON enabling custom metrics	VARCHAR	{"ROWS_RETURNED": "true", "LATENCY": "true", "RESULT_SET_SIZE": "true", "OPERATOR_TIMING": "true", "OPERATOR_ROWS_SCANNED": "true", "CUMULATIVE_ROWS_SCANNED": "true", "OPERATOR_CARDINALITY": "true", "OPERATOR_TYPE": "true", "OPERATOR_NAME": "true", "CPU_TIME": "true", "EXTRA_INFO": "true", "SYSTEM_PEAK_BUFFER_MEMORY": "true", "BLOCKED_THREAD_TIME": "true", "CUMULATIVE_CARDINALITY": "true", "SYSTEM_PEAK_TEMP_DIR_SIZE": "true", "QUERY_NAME": "true"}
disable_timestamptz_casts	Disable casting from timestamp to timestamptz	BOOLEAN	false
dynamic_or_filter_threshold	The maximum amount of OR filters we generate dynamically from a hash join	UBIGINT	50
enable_http_logging	Enables HTTP logging	BOOLEAN	true
enable_profiling	Enables profiling, and sets the output format (JSON, QUERY_TREE, QUERY_TREE_OPTIMIZER)	VARCHAR	NULL
enable_progress_bar_print	Controls the printing of the progress bar, when 'enable_progress_bar' is true	BOOLEAN	true

Name	Description	Type	Default value
enable_progress_bar	Enables the progress bar, printing progress to the terminal for long queries	BOOLEAN	true
errors_as_json	Output error messages as structured JSON instead of as a raw string	BOOLEAN	false
explain_output	Output of EXPLAIN statements (ALL, OPTIMIZED_ONLY, PHYSICAL_ONLY)	VARCHAR	physical_only
file_search_path	A comma separated list of directories to search for input files	VARCHAR	
home_directory	Sets the home directory used by the system	VARCHAR	
http_logging_output	The file to which HTTP logging output should be saved, or empty to print to the terminal	VARCHAR	
ieee_floating_point_ops	Use IEE754-compliant floating point operations (returning NAN instead of errors/NULL).	BOOLEAN	true
integer_division	Whether or not the / operator defaults to integer division, or to floating point division	BOOLEAN	false
lambda_syntax	Configures the use of the deprecated single arrow operator (->) for lambda functions.	VARCHAR	DEFAULT
late_materialization_max_rows	The maximum amount of rows in the LIMIT/SAMPLE for which we trigger late materialization	UBIGINT	50
log_query_path	Specifies the path to which queries should be logged (default: NULL, queries are not logged)	VARCHAR	NULL
max_expression_depth	The maximum expression depth limit in the parser. WARNING: increasing this setting and using very deep expressions might lead to stack overflow errors.	UBIGINT	1000
merge_join_threshold	The number of rows we need on either table to choose a merge join	UBIGINT	1000
nested_loop_join_threshold	The number of rows we need on either table to choose a nested loop join	UBIGINT	5
order_by_non_integer_literal	Allow ordering by non-integer literals - ordering by such literals has no effect.	BOOLEAN	false
ordered_aggregate_threshold	The number of rows to accumulate before sorting, used for tuning	UBIGINT	262144
partitioned_write_flush_threshold	The threshold in number of rows after which we flush a thread state when writing using PARTITION_BY	UBIGINT	524288
partitioned_write_max_open_files	The maximum amount of files the system can keep open before flushing to disk when writing using PARTITION_BY	UBIGINT	100
perfect_ht_threshold	Threshold in bytes for when to use a perfect hash table	UBIGINT	12
pivot_filter_threshold	The threshold to switch from using filtered aggregates to LIST with a dedicated pivot operator	UBIGINT	20
pivot_limit	The maximum number of pivot columns in a pivot statement	UBIGINT	100000
prefer_range_joins	Force use of range joins with mixed predicates	BOOLEAN	false
preserve_identifier_case	Whether or not to preserve the identifier case, instead of always lowercasing all non-quoted identifiers	BOOLEAN	true

Name	Description	Type	Default value
`profile_output`, `profiling_output`	The file to which profile output should be saved, or empty to print to the terminal	`VARCHAR`	
`profiling_mode`	The profiling mode (`STANDARD` or `DETAILED`)	`VARCHAR`	`NULL`
`progress_bar_time`	Sets the time (in milliseconds) how long a query needs to take before we start printing a progress bar	`BIGINT`	`2000`
`scalar_subquery_error_on_multiple_rows`	When a scalar subquery returns multiple rows - return a random row instead of returning an error.	`BOOLEAN`	`true`
`schema`	Sets the default search schema. Equivalent to setting search_path to a single value.	`VARCHAR`	`main`
`search_path`	Sets the default catalog search path as a comma-separated list of values	`VARCHAR`	
`streaming_buffer_size`	The maximum memory to buffer between fetching from a streaming result (e.g., 1GB)	`VARCHAR`	`976.5 KiB`

Pragmas

The PRAGMA statement is a SQL extension adopted by DuckDB from SQLite. PRAGMA statements can be issued in a similar manner to regular SQL statements. PRAGMA commands may alter the internal state of the database engine, and can influence the subsequent execution or behavior of the engine.

PRAGMA statements that assign a value to an option can also be issued using the SET statement and the value of an option can be retrieved using `SELECT current_setting(option_name)`.

For DuckDB's built in configuration options, see the Configuration Reference. DuckDB extensions may register additional configuration options. These are documented in the respective extensions' documentation pages.

This page contains the supported PRAGMA settings.

Metadata

Schema Information

List all databases:

```
PRAGMA database_list;
```

List all tables:

```
PRAGMA show_tables;
```

List all tables, with extra information, similarly to `DESCRIBE`:

```
PRAGMA show_tables_expanded;
```

To list all functions:

```
PRAGMA functions;
```

For queries targeting non-existing schemas, DuckDB generates "did you mean..." style error messages. When there are thousands of attached databases, these errors can take a long time to generate. To limit the number of schemas DuckDB looks through, use the `catalog_error_max_schemas` option:

```
SET catalog_error_max_schemas = 10;
```

Table Information

Get info for a specific table:

```
PRAGMA table_info('table_name');
CALL pragma_table_info('table_name');
```

`table_info` returns information about the columns of the table with name `table_name`. The exact format of the table returned is given below:

```
cid INTEGER,        -- cid of the column
name VARCHAR,       -- name of the column
type VARCHAR,       -- type of the column
notnull BOOLEAN,    -- if the column is marked as NOT NULL
dflt_value VARCHAR, -- default value of the column, or NULL if not specified
pk BOOLEAN          -- part of the primary key or not
```

Database Size

Get the file and memory size of each database:

```
PRAGMA database_size;
CALL pragma_database_size();
```

`database_size` returns information about the file and memory size of each database. The column types of the returned results are given below:

```
database_name VARCHAR, -- database name
database_size VARCHAR, -- total block count times the block size
block_size BIGINT,     -- database block size
total_blocks BIGINT,   -- total blocks in the database
used_blocks BIGINT,    -- used blocks in the database
free_blocks BIGINT,    -- free blocks in the database
wal_size VARCHAR,      -- write ahead log size
memory_usage VARCHAR,  -- memory used by the database buffer manager
memory_limit VARCHAR   -- maximum memory allowed for the database
```

Storage Information

To get storage information:

```
PRAGMA storage_info('table_name');
CALL pragma_storage_info('table_name');
```

This call returns the following information for the given table:

Name	Type	Description
`row_group_id`	`BIGINT`	
`column_name`	`VARCHAR`	
`column_id`	`BIGINT`	
`column_path`	`VARCHAR`	
`segment_id`	`BIGINT`	
`segment_type`	`VARCHAR`	
`start`	`BIGINT`	The start row id of this chunk
`count`	`BIGINT`	The amount of entries in this storage chunk
`compression`	`VARCHAR`	Compression type used for this column – see the "Lightweight Compression in DuckDB" blog post
`stats`	`VARCHAR`	
`has_updates`	`BOOLEAN`	
`persistent`	`BOOLEAN`	`false` if temporary table
`block_id`	`BIGINT`	Empty unless persistent
`block_offset`	`BIGINT`	Empty unless persistent

See Storage for more information.

Show Databases

The following statement is equivalent to the SHOW DATABASES statement:

```
PRAGMA show_databases;
```

Resource Management

Memory Limit

Set the memory limit for the buffer manager:

```
SET memory_limit = '1GB';
```

> **Warning.** The specified memory limit is only applied to the buffer manager. For most queries, the buffer manager handles the majority of the data processed. However, certain in-memory data structures such as vectors and query results are allocated outside of the buffer manager. Additionally, aggregate functions with complex state (e.g., `list`, `mode`, `quantile`, `string_agg`, and `approx` functions) use memory outside of the buffer manager. Therefore, the actual memory consumption can be higher than the specified memory limit.

Threads

Set the amount of threads for parallel query execution:

```
SET threads = 4;
```

Collations

List all available collations:

```
PRAGMA collations;
```

Set the default collation to one of the available ones:

```
SET default_collation = 'nocase';
```

Default Ordering for NULLs

Set the default ordering for NULLs to be either `NULLS_FIRST`, `NULLS_LAST`, `NULLS_FIRST_ON_ASC_LAST_ON_DESC` or `NULLS_LAST_ON_ASC_FIRST_ON_DESC`:

```
SET default_null_order = 'NULLS_FIRST';
SET default_null_order = 'NULLS_LAST_ON_ASC_FIRST_ON_DESC';
```

Set the default result set ordering direction to `ASCENDING` or `DESCENDING`:

```
SET default_order = 'ASCENDING';
SET default_order = 'DESCENDING';
```

Ordering by Non-Integer Literals

By default, ordering by non-integer literals is not allowed:

```
SELECT 42 ORDER BY 'hello world';

-- Binder Error: ORDER BY non-integer literal has no effect.
```

To allow this behavior, use the `order_by_non_integer_literal` option:

```
SET order_by_non_integer_literal = true;
```

Implicit Casting to VARCHAR

Prior to version 0.10.0, DuckDB would automatically allow any type to be implicitly cast to VARCHAR during function binding. As a result it was possible to e.g., compute the substring of an integer without using an explicit cast. For version v0.10.0 and later an explicit cast is needed instead. To revert to the old behavior that performs implicit casting, set the `old_implicit_casting` variable to `true`:

```
SET old_implicit_casting = true;
```

Python: Scan All Dataframes

Prior to version 1.1.0, DuckDB's replacement scan mechanism in Python scanned the global Python namespace. To revert to this old behavior, use the following setting:

```
SET python_scan_all_frames = true;
```

Information on DuckDB

Version

Show DuckDB version:

```
PRAGMA version;
CALL pragma_version();
```

Platform

`platform` returns an identifier for the platform the current DuckDB executable has been compiled for, e.g., `osx_arm64`. The format of this identifier matches the platform name as described in the extension loading explainer:

```
PRAGMA platform;
CALL pragma_platform();
```

User Agent

The following statement returns the user agent information, e.g., `duckdb/v0.10.0(osx_arm64)`:

```
PRAGMA user_agent;
```

Metadata Information

The following statement returns information on the metadata store (`block_id`, `total_blocks`, `free_blocks`, and `free_list`):

```
PRAGMA metadata_info;
```

Progress Bar

Show progress bar when running queries:

```
PRAGMA enable_progress_bar;
```

Or:

```
PRAGMA enable_print_progress_bar;
```

Don't show a progress bar for running queries:

```
PRAGMA disable_progress_bar;
```

Or:

```
PRAGMA disable_print_progress_bar;
```

EXPLAIN Output

The output of `EXPLAIN` can be configured to show only the physical plan.

The default configuration of `EXPLAIN`:

```
SET explain_output = 'physical_only';
```

To only show the optimized query plan:

```
SET explain_output = 'optimized_only';
```

To show all query plans:

```
SET explain_output = 'all';
```

Profiling

Enable Profiling

The following query enables profiling with the default format, `query_tree`. Independent of the format, `enable_profiling` is **mandatory** to enable profiling.

```
PRAGMA enable_profiling;
PRAGMA enable_profile;
```

Profiling Format

The format of `enable_profiling` can be specified as `query_tree`, `json`, `query_tree_optimizer`, or `no_output`. Each format prints its output to the configured output, except `no_output`.

The default format is `query_tree`. It prints the physical query plan and the metrics of each operator in the tree.

```
SET enable_profiling = 'query_tree';
```

Alternatively, `json` returns the physical query plan as JSON:

```
SET enable_profiling = 'json';
```

> **Tip.** To visualize query plans, consider using the DuckDB execution plan visualizer developed by the Database Systems Research Group at the University of Tübingen.

To return the physical query plan, including optimizer and planner metrics:

```
SET enable_profiling = 'query_tree_optimizer';
```

Database drivers and other applications can also access profiling information through API calls, in which case users can disable any other output. Even though the parameter reads `no_output`, it is essential to note that this **only** affects printing to the configurable output. When accessing profiling information through API calls, it is still crucial to enable profiling:

```
SET enable_profiling = 'no_output';
```

Profiling Output

By default, DuckDB prints profiling information to the standard output. However, if you prefer to write the profiling information to a file, you can use PRAGMA `profiling_output` to specify a filepath.

> **Warning.** The file contents will be overwritten for every newly issued query. Hence, the file will only contain the profiling information of the last run query:

```
SET profiling_output = '/path/to/file.json';
SET profile_output = '/path/to/file.json';
```

Profiling Mode

By default, a limited amount of profiling information is provided (`standard`).

```
SET profiling_mode = 'standard';
```

For more details, use the detailed profiling mode by setting `profiling_mode` to `detailed`. The output of this mode includes profiling of the planner and optimizer stages.

```
SET profiling_mode = 'detailed';
```

Custom Metrics

By default, profiling enables all metrics except those activated by detailed profiling.

Using the `custom_profiling_settings` PRAGMA, each metric, including those from detailed profiling, can be individually enabled or disabled. This PRAGMA accepts a JSON object with metric names as keys and Boolean values to toggle them on or off. Settings specified by this PRAGMA override the default behavior.

> **Note.** This only affects the metrics when the `enable_profiling` is set to `json` or `no_output`. The `query_tree` and `query_tree_optimizer` always use a default set of metrics.

In the following example, the `CPU_TIME` metric is disabled. The `EXTRA_INFO`, `OPERATOR_CARDINALITY`, and `OPERATOR_TIMING` metrics are enabled.

```
SET custom_profiling_settings = '{"CPU_TIME": "false", "EXTRA_INFO": "true", "OPERATOR_CARDINALITY":
"true", "OPERATOR_TIMING": "true"}';
```

The profiling documentation contains an overview of the available metrics.

Disable Profiling

To disable profiling:

```
PRAGMA disable_profiling;
PRAGMA disable_profile;
```

Query Optimization

Optimizer

To disable the query optimizer:

```
PRAGMA disable_optimizer;
```

To enable the query optimizer:

```
PRAGMA enable_optimizer;
```

Selectively Disabling Optimizers

The `disabled_optimizers` option allows selectively disabling optimization steps. For example, to disable `filter_pushdown` and `statistics_propagation`, run:

```
SET disabled_optimizers = 'filter_pushdown,statistics_propagation';
```

The available optimizations can be queried using the `duckdb_optimizers()` table function.

To re-enable the optimizers, run:

```
SET disabled_optimizers = '';
```

> **Warning.** The `disabled_optimizers` option should only be used for debugging performance issues and should be avoided in production.

Logging

Set a path for query logging:

```
SET log_query_path = '/tmp/duckdb_log/';
```

Disable query logging:

```
SET log_query_path = '';
```

Full-Text Search Indexes

The `create_fts_index` and `drop_fts_index` options are only available when the `fts` extension is loaded. Their usage is documented on the Full-Text Search extension page.

Verification

Verification of External Operators

Enable verification of external operators:

```
PRAGMA verify_external;
```

Disable verification of external operators:

```
PRAGMA disable_verify_external;
```

Verification of Round-Trip Capabilities

Enable verification of round-trip capabilities for supported logical plans:

```
PRAGMA verify_serializer;
```

Disable verification of round-trip capabilities:

```
PRAGMA disable_verify_serializer;
```

Object Cache

Enable caching of objects for e.g., Parquet metadata:

```
PRAGMA enable_object_cache;
```

Disable caching of objects:

```
PRAGMA disable_object_cache;
```

Checkpointing

Compression

During checkpointing, the existing column data + any new changes get compressed. There exist a couple pragmas to influence which compression functions are considered.

Force Compression Prefer using this compression method over any other method if possible:

```
PRAGMA force_compression = 'bitpacking';
```

Disabled Compression Methods Avoid using any of the listed compression methods from the comma separated list:

```
PRAGMA disabled_compression_methods = 'fsst,rle';
```

Force Checkpoint

When `CHECKPOINT` is called when no changes are made, force a checkpoint regardless:

```
PRAGMA force_checkpoint;
```

Checkpoint on Shutdown

Run a CHECKPOINT on successful shutdown and delete the WAL, to leave only a single database file behind:

```
PRAGMA enable_checkpoint_on_shutdown;
```

Don't run a CHECKPOINT on shutdown:

```
PRAGMA disable_checkpoint_on_shutdown;
```

Temp Directory for Spilling Data to Disk

By default, DuckDB uses a temporary directory named `<database_file_name>.tmp{:.language-sql .highlight}` to spill to disk, located in the same directory as the database file. To change this, use:

```
SET temp_directory = '/path/to/temp_dir.tmp/';
```

Returning Errors as JSON

The `errors_as_json` option can be set to obtain error information in raw JSON format. For certain errors, extra information or decomposed information is provided for easier machine processing. For example:

```
SET errors_as_json = true;
```

Then, running a query that results in an error produces a JSON output:

```
SELECT * FROM nonexistent_tbl;
```

```
{
   "exception_type":"Catalog",
   "exception_message":"Table with name nonexistent_tbl does not exist!\nDid you mean
   \"temp.information_schema.tables\"?",
   "name":"nonexistent_tbl",
   "candidates":"temp.information_schema.tables",
   "position":"14",
   "type":"Table",
   "error_subtype":"MISSING_ENTRY"
}
```

IEEE Floating-Point Operation Semantics

DuckDB follows IEEE floating-point operation semantics. If you would like to turn this off, run:

```
SET ieee_floating_point_ops = false;
```

In this case, floating point division by zero (e.g., `1.0 / 0.0`, `0.0 / 0.0` and `-1.0 / 0.0`) will all return NULL.

Query Verification (for Development)

The following PRAGMAs are mostly used for development and internal testing.

Enable query verification:

```
PRAGMA enable_verification;
```

Disable query verification:

```
PRAGMA disable_verification;
```

Enable force parallel query processing:

```
PRAGMA verify_parallelism;
```

Disable force parallel query processing:

```
PRAGMA disable_verify_parallelism;
```

Block Sizes

When persisting a database to disk, DuckDB writes to a dedicated file containing a list of blocks holding the data. In the case of a file that only holds very little data, e.g., a small table, the default block size of 256 kB might not be ideal. Therefore, DuckDB's storage format supports different block sizes.

There are a few constraints on possible block size values.

- Must be a power of two.
- Must be greater or equal to 16384 (16 kB).
- Must be lesser or equal to 262144 (256 kB).

You can set the default block size for all new DuckDB files created by an instance like so:

```
SET default_block_size = '16384';
```

It is also possible to set the block size on a per-file basis, see ATTACH for details.

Secrets Manager

The **Secrets manager** provides a unified user interface for secrets across all backends that use them. Secrets can be scoped, so different storage prefixes can have different secrets, allowing for example to join data across organizations in a single query. Secrets can also be persisted, so that they do not need to be specified every time DuckDB is launched.

> **Warning.** Persistent secrets are stored in unencrypted binary format on the disk.

Types of Secrets

Secrets are typed, their type identifies which service they are for. Most secrets are not included in DuckDB default, instead, they are registered by extensions. Currently, the following secret types are available:

Secret type	Service / protocol	Extension
`azure`	Azure Blob Storage	`azure`
`gcs`	Google Cloud Storage	`httpfs`
`http`	HTTP and HTTPS	`httpfs`
`huggingface`	Hugging Face	`httpfs`
`mysql`	MySQL	`mysql`
`postgres`	PostgreSQL	`postgres`
`r2`	Cloudflare R2	`httpfs`
`s3`	AWS S3	`httpfs`

For each type, there are one or more "secret providers" that specify how the secret is created. Secrets can also have an optional scope, which is a file path prefix that the secret applies to. When fetching a secret for a path, the secret scopes are compared to the path, returning the matching secret for the path. In the case of multiple matching secrets, the longest prefix is chosen.

Creating a Secret

Secrets can be created using the `CREATE SECRET` SQL statement. Secrets can be **temporary** or **persistent**. Temporary secrets are used by default – and are stored in-memory for the life span of the DuckDB instance similar to how settings worked previously. Persistent secrets are stored in **unencrypted binary format** in the `~/.duckdb/stored_secrets` directory. On startup of DuckDB, persistent secrets are read from this directory and automatically loaded.

Secret Providers

To create a secret, a **Secret Provider** needs to be used. A Secret Provider is a mechanism through which a secret is generated. To illustrate this, for the S3, GCS, R2, and AZURE secret types, DuckDB currently supports two providers: `CONFIG` and `credential_chain`. The `CONFIG` provider requires the user to pass all configuration information into the `CREATE SECRET`, whereas the `credential_chain` provider will automatically try to fetch credentials. When no Secret Provider is specified, the `CONFIG` provider is used. For more details on how to create secrets using different providers check out the respective pages on httpfs and azure.

Temporary Secrets

To create a temporary unscoped secret to access S3, we can now use the following:

```
CREATE SECRET my_secret (
    TYPE s3,
    KEY_ID 'my_secret_key',
    SECRET 'my_secret_value',
    REGION 'my_region'
);
```

Note that we implicitly use the default `CONFIG` secret provider here.

Persistent Secrets

In order to persist secrets between DuckDB database instances, we can now use the `CREATE PERSISTENT SECRET` command, e.g.:

```
CREATE PERSISTENT SECRET my_persistent_secret (
    TYPE s3,
    KEY_ID 'my_secret_key',
    SECRET 'my_secret_value'
);
```

By default, this will write the secret (unencrypted) to the `~/.duckdb/stored_secrets` directory. To change the secrets directory, issue:

```
SET secret_directory = 'path/to/my_secrets_dir';
```

Note that setting the value of the `home_directory` configuration option has no effect on the location of the secrets.

Deleting Secrets

Secrets can be deleted using the DROP SECRET statement, e.g.:

```
DROP PERSISTENT SECRET my_persistent_secret;
```

Creating Multiple Secrets for the Same Service Type

If two secrets exist for a service type, the scope can be used to decide which one should be used. For example:

```
CREATE SECRET secret1 (
    TYPE s3,
    KEY_ID 'my_secret_key1',
    SECRET 'my_secret_value1',
    SCOPE 's3://<my-bucket>'
);
```

```
CREATE SECRET secret2 (
    TYPE s3,
    KEY_ID 'my_secret_key2',
    SECRET 'my_secret_value2',
    SCOPE 's3://<my-other-bucket>'
);
```

Now, if the user queries something from `s3://<my-other-bucket>/something`, secret `secret2` will be chosen automatically for that request. To see which secret is being used, the `which_secret` scalar function can be used, which takes a path and a secret type as parameters:

```
FROM which_secret('s3://<my-other-bucket>/file.parquet', 's3');
```

Listing Secrets

Secrets can be listed using the built-in table-producing function, e.g., by using the `duckdb_secrets()` table function:

```
FROM duckdb_secrets();
```

Sensitive information will be redacted.

Extensions

Extensions

DuckDB has a flexible extension mechanism that allows for dynamically loading extensions. Extensions can enhance DuckDB's functionality by providing support for additional file formats, introducing new types, and domain-specific functionality.

> Extensions are loadable on all clients (e.g., Python and R). Extensions distributed via the Core and Community repositories are built and tested on macOS, Windows and Linux. All operating systems are supported for both the AMD64 and the ARM64 architectures.

Listing Extensions

To get a list of extensions, use the `duckdb_extensions` function:

```
SELECT extension_name, installed, description
FROM duckdb_extensions();
```

extension_name	installed	description
arrow	false	A zero-copy data integration between Apache Arrow and DuckDB
autocomplete	false	Adds support for autocomplete in the shell
...	...	...

This list will show which extensions are available, which extensions are installed, at which version, where it is installed, and more. The list includes most, but not all, available core extensions. For the full list, we maintain a list of core extensions.

Built-In Extensions

DuckDB's binary distribution comes standard with a few built-in extensions. They are statically linked into the binary and can be used as is. For example, to use the built-in `json` extension to read a JSON file:

```
SELECT *
FROM 'test.json';
```

To make the DuckDB distribution lightweight, only a few essential extensions are built-in, varying slightly per distribution. Which extension is built-in on which platform is documented in the list of core extensions.

Installing More Extensions

To make an extension that is not built-in available in DuckDB, two steps need to happen:

1. **Extension installation** is the process of downloading the extension binary and verifying its metadata. During installation, DuckDB stores the downloaded extension and some metadata in a local directory. From this directory DuckDB can then load the Extension whenever it needs to. This means that installation needs to happen only once.
2. **Extension loading** is the process of dynamically loading the binary into a DuckDB instance. DuckDB will search the local extension directory for the installed extension, then load it to make its features available. This means that every time DuckDB is restarted, all extensions that are used need to be (re)loaded

> Extension installation and loading are subject to a few limitations.

There are two main methods of making DuckDB perform the **installation** and **loading** steps for an installable extension: **explicitly** and through **autoloading**.

Explicit `INSTALL` and `LOAD`

In DuckDB extensions can also be explicitly installed and loaded. Both non-autoloadable and autoloadable extensions can be installed this way. To explicitly install and load an extension, DuckDB has the dedicated SQL statements `LOAD` and `INSTALL`. For example, to install and load the `spatial` extension, run:

```
INSTALL spatial;
LOAD spatial;
```

With these statements, DuckDB will ensure the spatial extension is installed (ignoring the `INSTALL` statement if it is already installed), then proceed to `LOAD` the spatial extension (again ignoring the statement if it is already loaded).

Extension Repository

Optionally a repository can be provided where the extension should be installed from, by appending `FROM <repository>`{:.language-sql .highlight} to the `INSTALL` / `FORCE INSTALL` command. This repository can either be an alias, such as `community`, or it can be a direct URL, provided as a single-quoted string.

After installing/loading an extension, the `duckdb_extensions` function can be used to get more information.

Autoloading Extensions

For many of DuckDB's core extensions, explicitly loading and installing extensions is not necessary. DuckDB contains an autoloading mechanism which can install and load the core extensions as soon as they are used in a query. For example, when running:

```
SELECT *
FROM 'https://raw.githubusercontent.com/duckdb/duckdb-web/main/data/weather.csv';
```

DuckDB will automatically install and load the `httpfs` extension. No explicit `INSTALL` or `LOAD` statements are required.

Not all extensions can be autoloaded. This can have various reasons: some extensions make several changes to the running DuckDB instance, making autoloading technically not (yet) possible. For others, it is preferred to have users opt-in to the extension explicitly before use due to the way they modify behavior in DuckDB.

To see which extensions can be autoloaded, check the core extensions list.

Community Extensions

DuckDB supports installing third-party community extensions. For example, you can install the `avro` community extension via:

```
INSTALL avro FROM community;
```

Community extensions are contributed by community members but they are built, signed, and distributed in a centralized repository.

Updating Extensions

While built-in extensions are tied to a DuckDB release due to their nature of being built into the DuckDB binary, installable extensions can and do receive updates. To ensure all currently installed extensions are on the most recent version, call:

```
UPDATE EXTENSIONS;
```

For more details on extension version refer to the Extension Versioning page.

Developing Extensions

The same API that the core extensions use is available for developing extensions. This allows users to extend the functionality of DuckDB such that it suits their domain the best. A template for creating extensions is available in the `extension-template` repository. This template also holds some documentation on how to get started building your own extension.

Working with Extensions

See the installation instructions and the advanced installation methods page.

Installing Extensions

To install core DuckDB extensions, use the `INSTALL` command. For example:

```
INSTALL httpfs;
```

This installs the extension from the default repository (`core`).

Extension Repositories

By default, DuckDB extensions are installed from a single repository containing extensions built and signed by the core DuckDB team. This ensures the stability and security of the core set of extensions. These extensions live in the default `core` repository, which points to `http://extensions.duckdb.org`.

Besides the core repository, DuckDB also supports installing extensions from other repositories. For example, the `core_nightly` repository contains nightly builds for core extensions that are built for the latest stable release of DuckDB. This allows users to try out new features in extensions before they are officially published.

Installing Extensions from Different Repositories

To install extensions from the default repository (`core`), run:

```
INSTALL httpfs;
```

To explicitly install an extension from the core repository, run:

```
INSTALL httpfs FROM core;
-- or
INSTALL httpfs FROM 'http://extensions.duckdb.org';
```

To install an extension from the core nightly repository:

```
INSTALL spatial FROM core_nightly;
-- or
INSTALL spatial FROM 'http://nightly-extensions.duckdb.org';
```

To install an extension from a custom repository:

```
INSTALL <custom_extension> FROM 'https://my-custom-extension-repository';
```

Alternatively, the `custom_extension_repository` setting can be used to change the default repository used by DuckDB:

```
SET custom_extension_repository = 'http://nightly-extensions.duckdb.org';
```

DuckDB contains the following predefined repositories:

Alias	URL	Description
`core`	`http://extensions.duckdb.org`	DuckDB core extensions
`core_nightly`	`http://nightly-extensions.duckdb.org`	Nightly builds for `core`

Alias	URL	Description
`community`	`http://community-extensions.duckdb.org`	DuckDB community extensions
`local_build_debug`	`./build/debug/repository`	Repository created when building DuckDB from source in debug mode (for development)
`local_build_release`	`./build/release/repository`	Repository created when building DuckDB from source in release mode (for development)

Working with Multiple Repositories

When working with extensions from different repositories, especially mixing `core` and `core_nightly`, it is important to know the origins and version of the different extensions. For this reason, DuckDB keeps track of this in the extension installation metadata. For example:

```
INSTALL httpfs FROM core;
INSTALL aws FROM core_nightly;
SELECT extension_name, extension_version, installed_from, install_mode
FROM duckdb_extensions();
```

This outputs:

extensions_name	extensions_version	installed_from	install_mode
httpfs	62d61a417f	core	REPOSITORY
aws	42c78d3	core_nightly	REPOSITORY
…	…	…	…

Force Installing to Upgrade Extensions

When DuckDB installs an extension, it is copied to a local directory to be cached and avoid future network traffic. Any subsequent calls to `INSTALL <extension_name>{:.language-sql .highlight}` will use the local version instead of downloading the extension again. To force re-downloading the extension, run:

```
FORCE INSTALL extension_name;
```

Force installing can also be used to overwrite an extension with an extension of the same name from another repository,

For example, first, `spatial` is installed from the core repository:

```
INSTALL spatial;
```

Then, to overwrite this installation with the `spatial` extension from the `core_nightly` repository:

```
FORCE INSTALL spatial FROM core_nightly;
```

Switching between Repositories

To switch repositories for an extension, use the `FORCE INSTALL` command. For example, if you have installed `httpfs` from the `core_nightly` repository but would like to switch back to using `core`, run:

```
FORCE INSTALL httpfs FROM core;
```

Installing Extensions through Client APIs

For many clients, using SQL to load and install extensions is the preferred method. However, some clients have a dedicated API to install and load extensions. For example, the Python client, has dedicated `install_extension(name: str)` and `load_extension(name: str)` methods. For more details on a specific client API, refer to the Client API documentation

Installation Location

By default, extensions are installed under the user's home directory:

`~/.duckdb/extensions/<duckdb_version>/<platform_name>/`

For stable DuckDB releases, the `<duckdb_version>`{:.language-sql .highlight} will be equal to the version tag of that release. For nightly DuckDB builds, it will be equal to the short git hash of the build. So for example, the extensions for DuckDB version v0.10.3 on macOS ARM64 (Apple Silicon) are installed to `~/.duckdb/extensions/v0.10.3/osx_arm64/`. An example installation path for a nightly DuckDB build could be `~/.duckdb/extensions/fc2e4b26a6/linux_amd64`.

To change the default location where DuckDB stores its extensions, use the `extension_directory` configuration option:

```
SET extension_directory = '/path/to/your/extension/directory';
```

Note that setting the value of the `home_directory` configuration option has no effect on the location of the extensions.

Uninstalling Extensions

Currently, DuckDB does not provide a command to uninstall extensions. To uninstall an extension, navigate to the extension's Installation Location and remove its `.duckdb_extension` binary file: For example:

```
rm ~/.duckdb/extensions/v1.2.1/osx_arm64/excel.duckdb_extension
```

Sharing Extensions between Clients

The shared installation location allows extensions to be shared between the client APIs *of the same DuckDB version*, as long as they share the same `platform` or ABI. For example, if an extension is installed with version 1.2.1 of the CLI client on macOS, it is available from the Python, R, etc. client libraries provided that they have access to the user's home directory and use DuckDB version 1.2.1.

Limitations

DuckDB's extension mechanism has the following limitations:

- Extensions cannot be unloaded.
- Extensions cannot be reloaded. If you update extensions, restart the DuckDB process to use newer extensions.

Advanced Installation Methods

Downloading Extensions Directly from S3

Downloading an extension directly can be helpful when building a Lambda service or container that uses DuckDB. DuckDB extensions are stored in public S3 buckets, but the directory structure of those buckets is not searchable. As a result, a direct URL to the file must be used. To download an extension file directly, use the following format:

`http://extensions.duckdb.org/v<duckdb_version>/<platform_name>/<extension_name>.duckdb_extension.gz`

For example:

`http://extensions.duckdb.org/v1.3.1/windows_amd64/json.duckdb_extension.gz`

Installing an Extension from an Explicit Path

The `INSTALL` command can be used with the path to a `.duckdb_extension` file:

```
INSTALL 'path/to/httpfs.duckdb_extension';
```

Note that compressed `.duckdb_extension.gz` files need to be decompressed beforehand. It is also possible to specify remote paths.

Loading an Extension from an Explicit Path

`LOAD` can be used with the path to a `.duckdb_extension`. For example, if the file was available at the (relative) path `path/to/httpfs.duckdb_extension`, you can load it as follows:

```
LOAD 'path/to/httpfs.duckdb_extension';
```

This will skip any currently installed extensions and load the specified extension directly.

Note that using remote paths for compressed files is currently not possible.

Building and Installing Extensions from Source

For building and installing extensions from source, see the Building DuckkDB guide.

Statically Linking Extensions

To statically link extensions, follow the developer documentation's “Using extension config files” section.

Extension Distribution

Platforms

Extension binaries are distributed for several platforms (see below). For platforms where packages for certain extensions are not available, users can build them from source and install the resulting binaries manually.

All official extensions are distributed for the following platforms.

Platform name	Operating system	Architecture	CPU types	Used by
`linux_amd64`	Linux	x86_64 (AMD64)		Node.js packages, etc.
`linux_arm64`	Linux	AArch64 (ARM64)	AWS Graviton, Snapdragon, etc.	All packages
`osx_amd64`	macOS	x86_64 (AMD64)	Intel	All packages
`osx_arm64`	macOS	AArch64 (ARM64)	Apple Silicon M1, M2, etc.	All packages
`windows_amd64`	Windows	x86_64 (AMD64)	Intel, AMD, etc.	All packages

Some extensions are distributed for the following platforms:

- `windows_amd64_mingw`
- `wasm_eh` and `wasm_mvp` (see DuckDB-Wasm's extensions)

For platforms outside the ones listed above, we do not officially distribute extensions (e.g., `linux_arm64_android`).

Extensions Signing

Signed Extensions

Extensions can be signed with a cryptographic key. By default, DuckDB uses its built-in public keys to verify the integrity of extension before loading them. All core and community extensions are signed by the DuckDB team.

Signing the extension simplifies their distribution, this is why they can be distributed over HTTP without the need for HTTPS, which is itself is supported through an extension (`httpfs`).

Unsigned Extensions

> **Warning.** Only load unsigned extensions from sources you trust. Avoid loading unsigned extensions over HTTP. Consult the Securing DuckDB page for guidelines on how set up DuckDB in a secure manner.

If you wish to load your own extensions or extensions from third-parties you will need to enable the `allow_unsigned_extensions` flag. To load unsigned extensions using the CLI client, pass the `-unsigned` flag to it on startup:

```
duckdb -unsigned
```

Now any extension can be loaded, signed or not:

```
LOAD './some/local/ext.duckdb_extension';
```

For client APIs, the `allow_unsigned_extensions` database configuration options needs to be set, see the respective Client API docs. For example, for the Python client, see the Loading and Installing Extensions section in the Python API documentation.

Binary Compatibility

To avoid binary compatibility issues, the binary extensions distributed by DuckDB are tied both to a specific DuckDB version and a platform. This means that DuckDB can automatically detect binary compatibility between it and a loadable extension. When trying to load an extension that was compiled for a different version or platform, DuckDB will throw an error and refuse to load the extension.

Creating a Custom Repository

You can create custom DuckDB extension repository. A DuckDB repository is an HTTP, HTTPS, S3, or local file based directory that serves the extensions files in a specific structure. This structure is described in the "Downloading Extensions Directly from S3" section, and is the same for local paths and remote servers, for example:

```
base_repository_path_or_url
└── v1.0.0
    └── osx_arm64
        ├── autocomplete.duckdb_extension
        ├── httpfs.duckdb_extension
        ├── icu.duckdb_extension
        ├── inet.duckdb_extension
        ├── json.duckdb_extension
        ├── parquet.duckdb_extension
        ├── tpcds.duckdb_extension
        ├── tpcds.duckdb_extension
        └── tpch.duckdb_extension
```

See the `extension-template` repository for all necessary code and scripts to set up a repository.

When installing an extension from a custom repository, DuckDB will search for both a gzipped and non-gzipped version. For example:

```
INSTALL icu FROM '<custom_repository>';
```

The execution of this statement will first look `icu.duckdb_extension.gz`, then `icu.duckdb_extension` in the repository's directory structure.

If the custom repository is served over HTTPS or S3, the `httpfs` extension is required. DuckDB will attempt to autoload the `httpfs` extension when an installation over HTTPS or S3 is attempted.

Versioning of Extensions

Extension Versioning

Most software has some sort of version number. Version numbers serve a few important goals:

- Tie a binary to a specific state of the source code
- Allow determining the expected feature set
- Allow determining the state of the APIs
- Allow efficient processing of bug reports (e.g., bug `#1337` was introduced in version `v3.4.5`)
- Allow determining chronological order of releases (e.g., version `v1.2.3` is older than `v1.2.4`)
- Give an indication of expected stability (e.g., `v0.0.1` is likely not very stable, whereas `v13.11.0` probably is stable)

Just like DuckDB itself, DuckDB extensions have their own version number. To ensure consistent semantics of these version numbers across the various extensions, DuckDB's Core Extensions use a versioning scheme that prescribes how extensions should be versioned. The versioning scheme for Core Extensions is made up of 3 different stability levels: **unstable**, **pre-release**, and **stable**. Let's go over each of the 3 levels and describe their format:

Unstable Extensions

Unstable extensions are extensions that can't (or don't want to) give any guarantees regarding their current stability, or their goals of becoming stable. Unstable extensions are tagged with the **short git hash** of the extension.

For example, at the time of writing this, the version of the `vss` extension is an unstable extension of version `690bfc5`.

What to expect from an extension that has a version number in the **unstable** format?

- The state of the source code of the extension can be found by looking up the hash in the extension repository
- Functionality may change or be removed completely with every release
- This extension's API could change with every release
- This extension may not follow a structured release cycle, new (breaking) versions can be pushed at any time

Pre-Release Extensions

Pre-release extensions are the next step up from Unstable extensions. They are tagged with version in the **SemVer** format, more specifically, those in the `v0.y.z` format. In semantic versioning, versions starting with `v0` have a special meaning: they indicate that the more strict semantics of regular (`>v1.0.0`) versions do not yet apply. It basically means that an extensions is working towards becoming a stable extension, but is not quite there yet.

For example, at the time of writing this, the version of the `delta` extension is a pre-release extension of version `v0.1.0`.

What to expect from an extension that has a version number in the **pre-release** format?

- The extension is compiled from the source code corresponding to the tag.
- Semantic Versioning semantics apply. See the Semantic Versioning specification for details.
- The extension follows a release cycle where new features are tested in nightly builds before being grouped into a release and pushed to the `core` repository.
- Release notes describing what has been added each release should be available to make it easy to understand the difference between versions.

Stable Extensions

Stable extensions are the final step of extension stability. This is denoted by using a **stable SemVer** of format `vx.y.z` where `x>0`.

For example, at the time of writing this, the version of the `parquet` extension is a stable extension of version `v1.0.0`.

What to expect from an extension that has a version number in the **stable** format? Essentially the same as pre-release extensions, but now the more strict SemVer semantics apply: the API of the extension should now be stable and will only change in backwards incompatible ways when the major version is bumped. See the SemVer specification for details

Release Cycle of Pre-Release and Stable Core Extensions

In general for extensions the release cycle depends on their stability level. **unstable** extensions are often in sync with DuckDB's release cycle, but may also be quietly updated between DuckDB releases. **pre-release** and **stable** extensions follow their own release cycle. These may or may not coincide with DuckDB releases. To find out more about the release cycle of a specific extension, refer to the documentation or GitHub page of the respective extension. Generally, **pre-release** and **stable** extensions will document their releases as GitHub releases, an example of which you can see in the `delta` extension.

Finally, there is a small exception: All in-tree extensions simply follow DuckDB's release cycle.

Nightly Builds

Just like DuckDB itself, DuckDB's core extensions have nightly or dev builds that can be used to try out features before they are officially released. This can be useful when your workflow depends on a new feature, or when you need to confirm that your stack is compatible with the upcoming version.

Nightly builds for extensions are slightly complicated due to the fact that currently DuckDB extensions binaries are tightly bound to a single DuckDB version. Because of this tight connection, there is a potential risk for a combinatorial explosion. Therefore, not all combinations of nightly extension build and nightly DuckDB build are available.

In general, there are 2 ways of using nightly builds: using a nightly DuckDB build and using a stable DuckDB build. Let's go over the differences between the two:

From Stable DuckDB

In most cases, user's will be interested in a nightly build of a specific extension, but don't necessarily want to switch to using the nightly build of DuckDB itself. This allows using a specific bleeding-edge feature while limiting the exposure to unstable code.

To achieve this, Core Extensions tend to regularly push builds to the `core_nightly` repository. Let's look at an example:

First we install a **stable DuckDB build**.

Then we can install and load a **nightly** extension like this:

```
INSTALL aws FROM core_nightly;
LOAD aws;
```

In this example we are using the latest **nightly** build of the aws extension with the latest **stable** version of DuckDB.

From Nightly DuckDB

When DuckDB CI produces a nightly binary of DuckDB itself, the binaries are distributed with a set of extensions that are pinned at a specific version. This extension version will be tested for that specific build of DuckDB, but might not be the latest dev build. Let's look at an example:

First, we install a **nightly DuckDB build**. Then, we can install and load the `aws` extension as expected:

```
INSTALL aws;
LOAD aws;
```

Updating Extensions

DuckDB has a dedicated statement that will automatically update all extensions to their latest version. The output will give the user information on which extensions were updated to/from which version. For example:

```
UPDATE EXTENSIONS;
```

extension_name	repository	update_result	previous_version	current_version
httpfs	core	NO_UPDATE_AVAILABLE	70fd6a8a24	70fd6a8a24
delta	core	UPDATED	d9e5cc1	04c61e4
azure	core_nightly	NO_UPDATE_AVAILABLE	49b63dc	49b63dc
aws	core	NO_UPDATE_AVAILABLE	42c78d3	42c78d3

Note that DuckDB will look for updates in the source repository for each extension. So if an extension was installed from `core_nightly`, it will be updated with the latest nightly build.

The update statement can also be provided with a list of specific extensions to update:

```
UPDATE EXTENSIONS (httpfs, aws);
```

extension_name	repository	update_result	previous_version	current_version
httpfs	core	NO_UPDATE_AVAILABLE	7ce5308	7ce5308
aws	core	NO_UPDATE_AVAILABLE	4f318eb	4f318eb

Target DuckDB Version

Currently, when extensions are compiled, they are tied to a specific version of DuckDB. What this means is that, for example, an extension binary compiled for version 0.10.3 does not work for version 1.0.0. In most cases, this will not cause any issues and is fully transparent; DuckDB will automatically ensure it installs the correct binary for its version. For extension developers, this means that they must ensure that new binaries are created whenever a new version of DuckDB is released. However, note that DuckDB provides an extension template that makes this fairly simple.

In-Tree vs. Out-of-Tree

Originally, DuckDB extensions lived exclusively in the DuckDB main repository, `github.com/duckdb/duckdb`. These extensions are called in-tree. Later, the concept of out-of-tree extensions was added, where extensions were separated into their own repository, which we call out-of-tree.

While from a user's perspective, there are generally no noticeable differences, there are some minor differences related to versioning:

- in-tree extensions use the version of DuckDB instead of having their own version
- in-tree extensions do not have dedicated release notes, their changes are reflected in the regular DuckDB release notes
- core out-of tree extensions tend to live in repositories named `github.com/duckdb/duckdb-<extension_name>`{:.language-sql .highlight} but the name may vary. See the full list of core extensions for details.

Troubleshooting of Extensions

This page is a placeholder troubleshooting extension installs. For the time being, please refer to the Installing Extensions page.

Core Extensions

Core Extensions

List of Core Extensions

Name	GitHub	Description	Stage	Aliases
autocomplete		Adds support for autocomplete in the shell	stable	
avro	GitHub	Add support for reading Avro files	stable	
aws	GitHub	Provides features that depend on the AWS SDK	stable	
azure	GitHub	Adds a filesystem abstraction for Azure blob storage to DuckDB	stable	
delta	GitHub	Adds support for Delta Lake	experimental	
ducklake	GitHub	Adds support for DuckLake	experimental	
encodings	GitHub	Adds support for encodings available in the ICU data repository	experimental	
excel	GitHub	Adds support for reading and writing Excel files	experimental	
fts	GitHub	Adds support for full-text search indexes	experimental	
httpfs	GitHub	Adds support for reading and writing files over an HTTP(S) or S3 connection	stable	http, https, s3
iceberg	GitHub	Adds support for Apache Iceberg	experimental	
icu		Adds support for time zones and collations using the ICU library	stable	
inet	GitHub	Adds support for IP-related data types and functions	experimental	
jemalloc		Overwrites system the allocator with jemalloc	stable	
json		Adds support for JSON operations	stable	
mysql	GitHub	Adds support for reading from and writing to a MySQL database	stable	mysql_scanner
parquet		Adds support for reading and writing Parquet files	stable	
postgres	GitHub	Adds support for reading from and writing to a PostgreSQL database	stable	postgres_scanner
spatial	GitHub	Geospatial extension that adds support for working with spatial data and functions	experimental	

Name	GitHub	Description	Stage	Aliases
sqlite	GitHub	Adds support for reading from and writing to SQLite database files	stable	sqlite_scanner, sqlite3
tpcds		Adds TPC-DS data generation and query support	experimental	
tpch		Adds TPC-H data generation and query support	stable	
ui	GitHub	Adds local UI for DuckDB	experimental	
vss	GitHub	Adds support for vector similarity search queries	experimental	

The **Stage** column shows the lifecycle stage of the extension following the convention of the lifecycle stages used in tidyverse.

Default Extensions

Different DuckDB clients ship a different set of extensions. We summarize the main distributions in the table below.

Name	CLI	Python	R	Java	Node.js
autocomplete	yes				
icu	yes	yes		yes	yes
json	yes	yes		yes	yes
parquet	yes	yes	yes	yes	yes

The jemalloc extension's availability is based on the operating system. Please check the jemalloc page for details.

AutoComplete Extension

The `autocomplete` extension adds supports for autocomplete in the CLI client. The extension is shipped by default with the CLI client.

Behavior

For the behavior of the `autocomplete` extension, see the documentation of the CLI client.

Functions

Function	Description
`sql_auto_complete(query_string)`	Attempts autocompletion on the given `query_string`.

Example

```
SELECT *
FROM sql_auto_complete('SEL');
```

Returns:

suggestion	suggestion_start
SELECT	0
DELETE	0
INSERT	0
CALL	0
LOAD	0
CALL	0
ALTER	0
BEGIN	0
EXPORT	0
CREATE	0
PREPARE	0
EXECUTE	0
EXPLAIN	0
ROLLBACK	0
DESCRIBE	0
SUMMARIZE	0

suggestion	suggestion_start
CHECKPOINT	0
DEALLOCATE	0
UPDATE	0
DROP	0

Avro Extension

The `avro` extension enables DuckDB to read Apache Avro files.

> The `avro` extensions was released as a community extension in late 2024 and became a core extension in early 2025.

The `read_avro` Function

The extension adds a single DuckDB function, `read_avro`. This function can be used like so:

```
FROM read_avro('<some_file>.avro');
```

This function will expose the contents of the Avro file as a DuckDB table. You can then use any arbitrary SQL constructs to further transform this table.

File IO

The `read_avro` function is integrated into DuckDB's file system abstraction, meaning you can read Avro files directly from, e.g., HTTP or S3 sources. For example:

```
FROM read_avro('http://blobs.duckdb.org/data/userdata1.avro');
FROM read_avro('s3://<your-bucket>/<some_file>.avro');
```

should ”just” work.

You can also *glob* multiple files in a single read call or pass a list of files to the functions:

```
FROM read_avro('some_file_*.avro');
FROM read_avro(['some_file_1.avro', 'some_file_2.avro']);
```

If the filenames somehow contain valuable information (as is unfortunately all-too-common), you can pass the `filename` argument to `read_avro`:

```
FROM read_avro('some_file_*.avro', filename=true);
```

This will result in an additional column in the result set that contains the actual filename of the Avro file.

Schema Conversion

This extension automatically translates the Avro Schema to the DuckDB schema. *All* Avro types can be translated, except for *recursive type definitions*, which DuckDB does not support.

The type mapping is very straightforward except for Avro's ”unique” way of handling `NULL`. Unlike other systems, Avro does not treat `NULL` as a possible value in a range of e.g. `INTEGER` but instead represents `NULL` as a union of the actual type with a special `NULL` type. This is different to DuckDB, where any value can be `NULL`. Of course DuckDB also supports `UNION` types, but this would be quite cumbersome to work with.

This extension *simplifies* the Avro schema where possible: An Avro union of any type and the special null type is simplified to just the non-null type. For example, an Avro record of the union type `["int","null"]` becomes a DuckDB `INTEGER`, which just happens to be `NULL`

sometimes. Similarly, an Avro union that contains only a single type is converted to the type it contains. For example, an Avro record of the union type `["int"]` also becomes a DuckDB `INTEGER`.

The extension also ”flattens” the Avro schema. Avro defines tables as root-level ”record” fields, which are the same as DuckDB `STRUCT` fields. For more convenient handling, this extension turns the entries of a single top-level record into top-level columns.

Implementation

Internally, this extension uses the ”official” Apache Avro C API, albeit with some minor patching to allow reading of Avro files from memory.

Limitations and Future Plans

- This extension currently does not make use of **parallelism** when reading either a single (large) Avro file or when reading a list of files. Adding support for parallelism in the latter case is on the roadmap.
- There is currently no support for neither projection nor filter **pushdown**, but this is also planned at a later stage.
- There is currently no support for the Wasm or the Windows-MinGW builds of DuckDB due to issues with the Avro library dependency (sigh again). We plan to fix this eventually.
- As mentioned above, DuckDB cannot express recursive type definitions that Avro has, this is unlikely to ever change.
- There is no support to allow users to provide a separate Avro schema file. This is unlikely to change, all Avro files we have seen so far had their schema embedded.
- There is currently no support for the `union_by_name` flag that other readers in DuckDB support. This is planned for the future.

AWS Extension

The aws extension adds functionality (e.g., authentication) on top of the `httpfs` extension's S3 capabilities, using the AWS SDK.

Installing and Loading

The aws extension will be transparently autoloaded on first use from the official extension repository. If you would like to install and load it manually, run:

```
INSTALL aws;
LOAD aws;
```

> In most cases, the aws extension works in conjunction with the [`httpfs` extension](#docs:stable:core_extensions:httpfs:overview.

::).

`config` Provider

The default provider, `config` (i.e., user-configured), allows access to the S3 bucket by manually providing a key. For example:

```
CREATE OR REPLACE SECRET secret (
    TYPE s3,
    PROVIDER config,
    KEY_ID '<AKIAIOSFODNN7EXAMPLE>',
    SECRET '<wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY>',
    REGION '<us-east-1>'
);
```

> **Tip.** If you get an IO Error (`Connection error for HTTP HEAD`), configure the endpoint explicitly via `ENDPOINT 's3.<your_region>.amazonaws.com'`{:.language-sql .highlight}.

Now, to query using the above secret, simply query any `s3://` prefixed file:

```
SELECT *
FROM 's3://<your-bucket>/<your_file>.parquet';
```

`credential_chain` Provider

The `credential_chain` provider allows automatically fetching credentials using mechanisms provided by the AWS SDK. For example, to use the AWS SDK default provider:

```
CREATE OR REPLACE SECRET secret (
    TYPE s3,
    PROVIDER credential_chain
);
```

Again, to query a file using the above secret, simply query any `s3://` prefixed file.

DuckDB also allows specifying a specific chain using the `CHAIN` keyword. This takes a semicolon-separated list (`a;b;c`) of providers that will be tried in order. For example:

```
CREATE OR REPLACE SECRET secret (
    TYPE s3,
    PROVIDER credential_chain,
    CHAIN 'env;config'
);
```

The possible values for CHAIN are the following:

- config
- sts
- sso
- env
- instance
- process

The credential_chain provider also allows overriding the automatically fetched config. For example, to automatically load credentials, and then override the region, run:

```
CREATE OR REPLACE SECRET secret (
    TYPE s3,
    PROVIDER credential_chain,
    CHAIN config,
    REGION '<eu-west-1>'
);
```

Auto-Refresh

Some AWS endpoints require periodic refreshing of the credentials. This can be specified with the REFRESH auto option:

```
CREATE SECRET env_test (
    TYPE s3,
    PROVIDER credential_chain,
    REFRESH auto
);
```

Legacy Features

> **Deprecated.** The load_aws_credentials function is deprecated.

Prior to version 0.10.0, DuckDB did not have a Secrets manager, to load the credentials automatically, the AWS extension provided a special function to load the AWS credentials in the legacy authentication method.

Function	Type	Description
load_aws_credentials	PRAGMA function	Loads the AWS credentials through the AWS Default Credentials Provider Chain

Load AWS Credentials (Legacy)

To load the AWS credentials, run:

```
CALL load_aws_credentials();
```

loaded_access_key_id	loaded_secret_access_key	loaded_session_token	loaded_region
AKIAIOSFODNN7EXAMPLE	<redacted>	NULL	us-east-2

The function takes a string parameter to specify a specific profile:

```
CALL load_aws_credentials('minio-testing-2');
```

loaded_access_key_id	loaded_secret_access_key	loaded_session_token	loaded_region
minio_duckdb_user_2	<redacted>	NULL	NULL

There are several parameters to tweak the behavior of the call:

```
CALL load_aws_credentials('minio-testing-2', set_region = false, redact_secret = false);
```

loaded_access_key_id	loaded_secret_access_key	loaded_session_token	loaded_region
minio_duckdb_user_2	minio_duckdb_user_password_2	NULL	NULL

Azure Extension

The `azure` extension is a loadable extension that adds a filesystem abstraction for the Azure Blob storage to DuckDB.

Installing and Loading

The `azure` extension will be transparently autoloaded on first use from the official extension repository. If you would like to install and load it manually, run:

```
INSTALL azure;
LOAD azure;
```

Usage

Once the authentication is set up, you can query Azure storage as follows:

Azure Blob Storage

Allowed URI schemes: `az` or `azure`

```
SELECT count(*)
FROM 'az://<my_container>/<path>/<my_file>.<parquet_or_csv>';
```

Globs are also supported:

```
SELECT *
FROM 'az://<my_container>/<path>/*.csv';
```

```
SELECT *
FROM 'az://<my_container>/<path>/**';
```

Or with a fully qualified path syntax:

```
SELECT count(*)
FROM 'az://<my_storage_account>.blob.core.windows.net/<my_container>/<path>/<my_file>.<parquet_or_csv>';
```

```
SELECT *
FROM 'az://<my_storage_account>.blob.core.windows.net/<my_container>/<path>/*.csv';
```

Azure Data Lake Storage (ADLS)

Allowed URI schemes: `abfss`

```
SELECT count(*)
FROM 'abfss://<my_filesystem>/<path>/<my_file>.<parquet_or_csv>';
```

Globs are also supported:

```
SELECT *
FROM 'abfss://<my_filesystem>/<path>/*.csv';
```

```
SELECT *
FROM 'abfss://<my_filesystem>/<path>/**';
```

Or with a fully qualified path syntax:

```
SELECT count(*)
FROM 'abfss://<my_storage_account>.dfs.core.windows.net/<my_filesystem>/<path>/<my_file>.<parquet_or_csv>';
```

```
SELECT *
FROM 'abfss://<my_storage_account>.dfs.core.windows.net/<my_filesystem>/<path>/*.csv';
```

Configuration

Use the following configuration options how the extension reads remote files:

Name	Description	Type	Default
`azure_http_stats`	Include HTTP info from Azure Storage in the `EXPLAIN ANALYZE` statement.	`BOOLEAN`	`false`
`azure_read_transfer_concurrency`	Maximum number of threads the Azure client can use for a single parallel read. If `azure_read_transfer_chunk_size` is less than `azure_read_buffer_size` then setting this > 1 will allow the Azure client to do concurrent requests to fill the buffer.	`BIGINT`	`5`
`azure_read_transfer_chunk_size`	Maximum size in bytes that the Azure client will read in a single request. It is recommended that this is a factor of `azure_read_buffer_size`.	`BIGINT`	`1024*1024`
`azure_read_buffer_size`	Size of the read buffer. It is recommended that this is evenly divisible by `azure_read_transfer_chunk_size`.	`UBIGINT`	`1024*1024`
`azure_transport_option_type`	Underlying adapter to use in the Azure SDK. Valid values are: `default` or `curl`.	`VARCHAR`	`default`
`azure_context_caching`	Enable/disable the caching of the underlying Azure SDK HTTP connection in the DuckDB connection context when performing queries. If you suspect that this is causing some side effect, you can try to disable it by setting it to false (not recommended).	`BOOLEAN`	`true`

> Setting `azure_transport_option_type` explicitly to `curl` will have the following effect:
>
> - On Linux, this may solve certificate issue (`Error: Invalid Error: Fail to get a new connection for: https://storage_account_name.blob.core.windows.net/. Problem with the SSL CA cert (path? access rights?)`) because when specifying the extension will try to find the bundle certificate in various paths (that is not done by *curl* by default and might be wrong due to static linking).
> - On Windows, this replaces the default adapter (*WinHTTP*) allowing you to use all *curl* capabilities (for example using a socks proxies).
> - On all operating systems, it will honor the following environment variables:
>   - `CURL_CA_INFO`: Path to a PEM encoded file containing the certificate authorities sent to libcurl. Note that this option is known to only work on Linux and might throw if set on other platforms.
>   - `CURL_CA_PATH`: Path to a directory which holds PEM encoded files, containing the certificate authorities sent to libcurl.

Example:

```
SET azure_http_stats = false;
SET azure_read_transfer_concurrency = 5;
SET azure_read_transfer_chunk_size = 1_048_576;
SET azure_read_buffer_size = 1_048_576;
```

Authentication

The Azure extension has two ways to configure the authentication. The preferred way is to use Secrets.

Authentication with Secret

Multiple Secret Providers are available for the Azure extension:

- If you need to define different secrets for different storage accounts, use the SCOPE configuration. Note that the `SCOPE` requires a trailing slash (`SCOPE 'azure://some_container/'`).
- If you use fully qualified path then the `ACCOUNT_NAME` attribute is optional.

`CONFIG` Provider

The default provider, `CONFIG` (i.e., user-configured), allows access to the storage account using a connection string or anonymously. For example:

```
CREATE SECRET secret1 (
    TYPE azure,
    CONNECTION_STRING '<value>'
);
```

If you do not use authentication, you still need to specify the storage account name. For example:

```
CREATE SECRET secret2 (
    TYPE azure,
    PROVIDER config,
    ACCOUNT_NAME '<storage_account_name>'
);
```

The default `PROVIDER` is `CONFIG`.

`credential_chain` Provider

The `credential_chain` provider allows connecting using credentials automatically fetched by the Azure SDK via the Azure credential chain. By default, the `DefaultAzureCredential` chain used, which tries credentials according to the order specified by the Azure documentation. For example:

```
CREATE SECRET secret3 (
    TYPE azure,
    PROVIDER credential_chain,
    ACCOUNT_NAME '<storage_account_name>'
);
```

DuckDB also allows specifying a specific chain using the `CHAIN` keyword. This takes a semicolon-separated list (`a;b;c`) of providers that will be tried in order. For example:

```
CREATE SECRET secret4 (
    TYPE azure,
    PROVIDER credential_chain,
    CHAIN 'cli;env',
    ACCOUNT_NAME '<storage_account_name>'
);
```

The possible values are the following: `cli`; `managed_identity`; `env`; `default`;

If no explicit `CHAIN` is provided, the default one will be `default`

`SERVICE_PRINCIPAL` Provider

The `SERVICE_PRINCIPAL` provider allows connecting using a Azure Service Principal (SPN).

Either with a secret:

```
CREATE SECRET azure_spn (
    TYPE azure,
    PROVIDER service_principal,
    TENANT_ID '<tenant_id>',
    CLIENT_ID '<client_id>',
    CLIENT_SECRET '<client_secret>',
    ACCOUNT_NAME '<storage_account_name>'
);
```

Or with a certificate:

```
CREATE SECRET azure_spn_cert (
    TYPE azure,
    PROVIDER service_principal,
    TENANT_ID '<tenant_id>',
    CLIENT_ID '<client_id>',
    CLIENT_CERTIFICATE_PATH '<client_cert_path>',
    ACCOUNT_NAME '<storage_account_name>'
);
```

Configuring a Proxy

To configure proxy information when using secrets, you can add `HTTP_PROXY`, `PROXY_USER_NAME`, and `PROXY_PASSWORD` in the secret definition. For example:

```
CREATE SECRET secret5 (
    TYPE azure,
    CONNECTION_STRING '<value>',
```

```
    HTTP_PROXY 'http://localhost:3128',
    PROXY_USER_NAME 'john',
    PROXY_PASSWORD 'doe'
);
```

> - When using secrets, the HTTP_PROXY environment variable will still be honored except if you provide an explicit value for it.
> - When using secrets, the SET variable of the *Authentication with variables* session will be ignored.
> - The Azure `credential_chain` provider, the actual token is fetched at query time, not when the secret is created.

Authentication with Variables (Deprecated)

```
SET variable_name = variable_value;
```

Where `variable_name` can be one of the following:

Name	Description	Type	Default
`azure_storage_connection_string`	Azure connection string, used for authenticating and configuring Azure requests.	`STRING`	-
`azure_account_name`	Azure account name, when set, the extension will attempt to automatically detect credentials (not used if you pass the connection string).	`STRING`	-
`azure_endpoint`	Override the Azure endpoint for when the Azure credential providers are used.	`STRING`	`blob.core.windows.net`
`azure_credential_chain`	Ordered list of Azure credential providers, in string format separated by `;`. For example: `'cli;managed_identity;env'`. See the list of possible values in the `credential_chain` provider section. Not used if you pass the connection string.	`STRING`	-
`azure_http_proxy`	Proxy to use when login & performing request to Azure.	`STRING`	HTTP_PROXY environment variable (if set).
`azure_proxy_user_name`	HTTP proxy username if needed.	`STRING`	-
`azure_proxy_password`	HTTP proxy password if needed.	`STRING`	-

Additional Information

Logging

The Azure extension relies on the Azure SDK to connect to Azure Blob storage and supports printing the SDK logs to the console. To control the log level, set the `AZURE_LOG_LEVEL` environment variable.

For instance, verbose logs can be enabled in Python as follows:

```
import os
import duckdb

os.environ["AZURE_LOG_LEVEL"] = "verbose"

duckdb.sql("CREATE SECRET myaccount (TYPE azure, PROVIDER credential_chain, SCOPE
'az://myaccount.blob.core.windows.net/')")
duckdb.sql("SELECT count(*) FROM 'az://myaccount.blob.core.windows.net/path/to/blob.parquet'")
```

Difference between ADLS and Blob Storage

Even though ADLS implements similar functionality as the Blob storage, there are some important performance benefits to using the ADLS endpoints for globbing, especially when using (complex) glob patterns.

To demonstrate, let's look at an example of how a glob is performed internally using the Blob and ADLS endpoints, respectively.

Using the following filesystem:

```
root
├── l_receipmonth=1997-10
│   ├── l_shipmode=AIR
│   │   └── data_0.csv
│   ├── l_shipmode=SHIP
│   │   └── data_0.csv
│   └── l_shipmode=TRUCK
│       └── data_0.csv
├── l_receipmonth=1997-11
│   ├── l_shipmode=AIR
│   │   └── data_0.csv
│   ├── l_shipmode=SHIP
│   │   └── data_0.csv
│   └── l_shipmode=TRUCK
│       └── data_0.csv
└── l_receipmonth=1997-12
    ├── l_shipmode=AIR
    │   └── data_0.csv
    ├── l_shipmode=SHIP
    │   └── data_0.csv
    └── l_shipmode=TRUCK
        └── data_0.csv
```

The following query is performed through the Blob endpoint:

```
SELECT count(*)
FROM 'az://root/l_receipmonth=1997-*/l_shipmode=SHIP/*.csv';
```

It will perform the following steps:

- List all the files with the prefix `root/l_receipmonth=1997-`

```
- root/l_receipmonth=1997-10/l_shipmode=SHIP/data_0.csv
- root/l_receipmonth=1997-10/l_shipmode=AIR/data_0.csv
- root/l_receipmonth=1997-10/l_shipmode=TRUCK/data_0.csv
- root/l_receipmonth=1997-11/l_shipmode=SHIP/data_0.csv
- root/l_receipmonth=1997-11/l_shipmode=AIR/data_0.csv
- root/l_receipmonth=1997-11/l_shipmode=TRUCK/data_0.csv
- root/l_receipmonth=1997-12/l_shipmode=SHIP/data_0.csv
- root/l_receipmonth=1997-12/l_shipmode=AIR/data_0.csv
```

```
  - root/l_receipmonth=1997-12/l_shipmode=TRUCK/data_0.csv
```

- Filter the result with the requested pattern `root/l_receipmonth=1997-*/l_shipmode=SHIP/*.csv`

```
- root/l_receipmonth=1997-10/l_shipmode=SHIP/data_0.csv
- root/l_receipmonth=1997-11/l_shipmode=SHIP/data_0.csv
- root/l_receipmonth=1997-12/l_shipmode=SHIP/data_0.csv
```

Meanwhile, the same query can be performed through the datalake endpoint as follows:

```
SELECT count(*)
FROM 'abfss://root/l_receipmonth=1997-*/l_shipmode=SHIP/*.csv';
```

This will perform the following steps:

- List all directories in `root/`

```
- root/l_receipmonth=1997-10
- root/l_receipmonth=1997-11
- root/l_receipmonth=1997-12
```

- Filter and list subdirectories: `root/l_receipmonth=1997-10`, `root/l_receipmonth=1997-11`, `root/l_receipmonth=1997-12`

```
- root/l_receipmonth=1997-10/l_shipmode=SHIP
- root/l_receipmonth=1997-10/l_shipmode=AIR
- root/l_receipmonth=1997-10/l_shipmode=TRUCK
- root/l_receipmonth=1997-11/l_shipmode=SHIP
- root/l_receipmonth=1997-11/l_shipmode=AIR
- root/l_receipmonth=1997-11/l_shipmode=TRUCK
- root/l_receipmonth=1997-12/l_shipmode=SHIP
- root/l_receipmonth=1997-12/l_shipmode=AIR
- root/l_receipmonth=1997-12/l_shipmode=TRUCK
```

- Filter and list subdirectories: `root/l_receipmonth=1997-10/l_shipmode=SHIP`, `root/l_receipmonth=1997-11/l_shipmode=SHIP`, `root/l_receipmonth=1997-12/l_shipmode=SHIP`

```
- root/l_receipmonth=1997-10/l_shipmode=SHIP/data_0.csv
- root/l_receipmonth=1997-11/l_shipmode=SHIP/data_0.csv
- root/l_receipmonth=1997-12/l_shipmode=SHIP/data_0.csv
```

As you can see because the Blob endpoint does not support the notion of directories, the filter can only be performed after the listing, whereas the ADLS endpoint will list files recursively. Especially with higher partition/directory counts, the performance difference can be very significant.

Delta Extension

The `delta` extension adds support for the Delta Lake open-source storage format. It is built using the Delta Kernel. The extension offers **read support** for Delta tables, both local and remote.

For implementation details, see the announcement blog post.

> **Warning.** The `delta` extension is currently experimental and is only supported on given platforms.

Installing and Loading

The `delta` extension will be transparently autoloaded on first use from the official extension repository. If you would like to install and load it manually, run:

```
INSTALL delta;
LOAD delta;
```

Usage

To scan a local Delta table, run:

```
SELECT *
FROM delta_scan('file:///some/path/on/local/machine');
```

Reading from an S3 Bucket

To scan a Delta table in an S3 bucket, run:

```
SELECT *
FROM delta_scan('s3://some/delta/table');
```

For authenticating to S3 buckets, DuckDB Secrets are supported:

```
CREATE SECRET (
    TYPE s3,
    PROVIDER credential_chain
);
SELECT *
FROM delta_scan('s3://some/delta/table/with/auth');
```

To scan public buckets on S3, you may need to pass the correct region by creating a secret containing the region of your public S3 bucket:

```
CREATE SECRET (
    TYPE s3,
    REGION 'my-region'
);
SELECT *
FROM delta_scan('s3://some/public/table/in/my-region');
```

Reading from Azure Blob Storage

To scan a Delta table in an Azure Blob Storage bucket, run:

```
SELECT *
FROM delta_scan('az://my-container/my-table');
```

For authenticating to Azure Blob Storage, DuckDB Secrets are supported:

```
CREATE SECRET (
    TYPE azure,
    PROVIDER credential_chain
);
SELECT *
FROM delta_scan('az://my-container/my-table-with-auth');
```

Features

While the `delta` extension is still experimental, many (scanning) features and optimizations are already supported:

- multithreaded scans and Parquet metadata reading
- data skipping/filter pushdown
  - skipping row groups in file (based on Parquet metadata)
  - skipping complete files (based on Delta partition information)
- projection pushdown
- scanning tables with deletion vectors
- all primitive types
- structs
- S3 support with secrets

More optimizations are going to be released in the future.

Supported DuckDB Versions and Platforms

The `delta` extension requires DuckDB version 0.10.3 or newer.

The `delta` extension currently only supports the following platforms:

- Linux AMD64 (x86_64 and ARM64): `linux_amd64` and `linux_arm64`
- macOS Intel and Apple Silicon: `osx_amd64` and `osx_arm64`
- Windows AMD64: `windows_amd64`

Support for the other DuckDB platforms is work-in-progress.

DuckLake

> DuckLake has been released in May 2025. Read the announcement blog post.

The `ducklake` extension add support for attaching to databases stored in the DuckLake format:

Installing and Loading

To install `ducklake`, run:

```
INSTALL ducklake;
```

The `ducklake` extension will be transparently autoloaded on first use in an `ATTACH` clause. If you would like to load it manually, run:

```
LOAD ducklake;
```

Usage

```
ATTACH 'ducklake:metadata.ducklake' AS my_ducklake (DATA_PATH 'data_files');
USE my_ducklake;
```

Tables

In DuckDB, the `ducklake` extension stores the catalog tables for a DuckLake named `my_ducklake` in the `__ducklake_metadata_<my_ducklake>{:.language-sql .highlight}` catalog.

Functions

Note that DuckLake registers several functions. These should be called with the catalog name as the first argument, e.g.:

```
FROM ducklake_snapshots('my_ducklake');
```

snapshot_id int64	snapshot_time timestamp with time zone	schema_version int64	changes map(varchar, varchar[])
0	2025-05-26 11:41:10.838+02	0	{schemas_created=[main]}

`ducklake_snapshots`

Returns the snapshots stored in the DuckLake catalog name `catalog`.

Parameter name	Parameter type	Named parameter	Description
`catalog`	`VARCHAR`	no	

The information is encoded into a table with the following schema:

Column name	Column type
`snapshot_id`	`BIGINT`
`snapshot_time`	`TIMESTAMP WITH TIME ZONE`
`schema_version`	`BIGINT`
`changes`	`MAP(VARCHAR, VARCHAR[])`

`ducklake_table_info`

The `ducklake_table_info` function returns information on the tables stored in the DuckLake catalog named `catalog`.

Parameter name	Parameter type	Named parameter	Description
`catalog`	`VARCHAR`	no	

The information is encoded into a table with the following schema:

Column name	Column type
`table_name`	`VARCHAR`
`schema_id`	`BIGINT`
`table_id`	`BIGINT`
`table_uuid`	`UUID`
`file_count`	`BIGINT`
`file_size_bytes`	`BIGINT`
`delete_file_count`	`BIGINT`
`delete_file_size_bytes`	`BIGINT`

`ducklake_table_insertions`

The `ducklake_table_insertions` function returns the rows inserted in a given table between snapshots of given versions or timestamps. The function has two variants, depending on whether `start_snapshot` and `end_snapshot` have types `BIGINT` or `TIMESTAMP WITH TIME ZONE`.

Parameter name	Parameter type	Named parameter	Description
`catalog`	`VARCHAR`	no	
`schema_name`	`VARCHAR`	no	
`table_name`	`VARCHAR`	no	

Parameter name	Parameter type	Named parameter	Description
`start_snapshot`	`BIGINT` / `TIMESTAMP WITH TIME ZONE`	no	
`end_snapshot`	`BIGINT` / `TIMESTAMP WITH TIME ZONE`	no	

The schema of the table returned by the function is equivalent to that of the table `table_name`.

`ducklake_table_deletions`

The `ducklake_table_deletions` function returns the rows deleted from a given table between snapshots of given versions or timestamps. The function has two variants, depending on whether `start_snapshot` and `end_snapshot` have types `BIGINT` or `TIMESTAMP WITH TIME ZONE`.

Parameter name	Parameter type	Named parameter	Description
`catalog`	`VARCHAR`	no	
`schema_name`	`VARCHAR`	no	
`table_name`	`VARCHAR`	no	
`start_snapshot`	`BIGINT` / `TIMESTAMP WITH TIME ZONE`	no	
`end_snapshot`	`BIGINT` / `TIMESTAMP WITH TIME ZONE`	no	

The schema of the table returned by the function is equivalent to that of the table `table_name`.

`ducklake_table_changes`

The `ducklake_table_changes` function returns the rows changed in a given table between snapshots of given versions or timestamps. The function has two variants, depending on whether `start_snapshot` and `end_snapshot` have types `BIGINT` or `TIMESTAMP WITH TIME ZONE`.

Parameter name	Parameter type	Named parameter	Description
`catalog`	`VARCHAR`	no	
`schema_name`	`VARCHAR`	no	
`table_name`	`VARCHAR`	no	
`start_snapshot`	`BIGINT` / `TIMESTAMP WITH TIME ZONE`	no	
`end_snapshot`	`BIGINT` / `TIMESTAMP WITH TIME ZONE`	no	

The schema of the table returned by the function contains the following three columns plus the schema of the table `table_name`.

Column name	Column type	Description
`snapshot_id`	`BIGINT`	
`rowid`	`BIGINT`	
`change_type`	`VARCHAR`	The type of change: `insert` or `delete`

Commands

`ducklake_cleanup_old_files`

The `ducklake_cleanup_old_files` function cleans up old files in the DuckLake denoted by `catalog`. Upon success, it returns a table with a single column (`Success`) and 0 rows.

Parameter name	Parameter type	Named parameter	Description
`catalog`	`VARCHAR`	no	
`cleanup_all`	`BOOLEAN`	yes	
`dry_run`	`BOOLEAN`	yes	
`older_than`	`TIMESTAMP WITH TIME ZONE`	yes	

`ducklake_expire_snapshots`

The `ducklake_expire_snapshots` function expires snapshots with the versions specified by the `versions` parameter or the ones older than the `older_than` parameter. Upon success, it returns a table with a single column (`Success`) and 0 rows.

Parameter name	Parameter type	Named parameter	Description
`catalog`	`VARCHAR`	no	
`versions`	`UBIGINT[]`	yes	
`older_than`	`TIMESTAMP WITH TIME ZONE`	yes	

`ducklake_merge_adjacent_files`

The `ducklake_merge_adjacent_files` function merges adjacent files in the storage. Upon success, it returns a table with a single column (`Success`) and 0 rows.

Parameter name	Parameter type	Named parameter	Description
`catalog`	`VARCHAR`	no	

Encodings Extension

The `encodings` extension adds supports for reading CSVs using more than 1,000 character encodings available in the ICU data repository.

Installing and Loading

```
INSTALL encodings;
LOAD encodings;
```

Usage

Refer to the encoding while reading from files:

```
FROM read_csv('my_shift_jis.csv', encoding = 'shift_jis');
```

Excel Extension

The `excel` extension provides functions to format numbers per Excel's formatting rules by wrapping the i18npool library, but as of DuckDB 1.2 also provides functionality to read and write Excel (`.xlsx`) files. However, `.xls` files are not supported.

Previously, reading and writing Excel files was handled through the `spatial` extension, which coincidentally included support for XLSX files through one of its dependencies, but this capability may be removed from the spatial extension in the future. Additionally, the `excel` extension is more efficient and provides more control over the import/export process. See the Excel Import and Excel Export pages for instructions.

Installing and Loading

The `excel` extension will be transparently autoloaded on first use from the official extension repository. If you would like to install and load it manually, run:

```
INSTALL excel;
LOAD excel;
```

Excel Scalar Functions

Function	Description
`excel_text(number, format_string)`	Format the given `number` per the rules given in the `format_string`
`text(number, format_string)`	Alias for `excel_text`

Examples

```
SELECT excel_text(1_234_567.897, 'h:mm AM/PM') AS timestamp;
```

timestamp
9:31 PM

```
SELECT excel_text(1_234_567.897, 'h AM/PM') AS timestamp;
```

timestamp
9 PM

Reading XLSX Files

Reading a `.xlsx` file is as simple as just SELECTing from it immediately, e.g.:

```
SELECT *
FROM 'test.xlsx';
```

a	b
1.0	2.0
3.0	4.0

However, if you want to set additional options to control the import process, you can use the `read_xlsx` function instead. The following named parameters are supported.

Option	Type	Default	Description
`header`	BOOLEAN	*Automatically inferred*	Whether to treat the first row as containing the names of the resulting columns.
`sheet`	VARCHAR	*Automatically inferred*	The name of the sheet in the xlsx file to read. Default is the first sheet.
`all_varchar`	BOOLEAN	`false`	Whether to read all cells as containing VARCHARs.
`ignore_errors`	BOOLEAN	`false`	Whether to ignore errors and silently replace cells that cant be cast to the corresponding inferred column type with `NULL`'s.
`range`	VARCHAR	*Automatically inferred*	The range of cells to read, in spreadsheet notation. For example, `A1:B2` reads the cells from A1 to B2. If not specified the resulting range will be inferred as rectangular region of cells between the first row of consecutive non-empty cells and the first empty row spanning the same columns.
`stop_at_empty`	BOOLEAN	*Automatically inferred*	Whether to stop reading the file when an empty row is encountered. If an explicit `range` option is provided, this is `false` by default, otherwise `true`.
`empty_as_varchar`	BOOLEAN	`false`	Whether to treat empty cells as VARCHAR instead of DOUBLE when trying to automatically infer column types.

```
SELECT *
FROM read_xlsx('test.xlsx', header = true);
```

a	b
1.0	2.0
3.0	4.0

Alternatively, the COPY statement with the XLSX format option can be used to import an Excel file into an existing table, in which case the types of the columns in the target table will be used to coerce the types of the cells in the Excel file.

```
CREATE TABLE test (a DOUBLE, b DOUBLE);
COPY test FROM 'test.xlsx' WITH (FORMAT xlsx, HEADER);
SELECT * FROM test;
```

Type and Range Inference

Because Excel itself only really stores numbers or strings in cells, and does not enforce that all cells in a column are of the same type, the `excel` extension has to do some guesswork to "infer" and decide the types of the columns when importing an Excel sheet. While almost all columns are inferred as either DOUBLE or VARCHAR, there are some caveats:

- TIMESTAMP, TIME, DATE and BOOLEAN types are inferred when possible based on the *format* applied to the cell.
- Text cells containing TRUE and FALSE are inferred as BOOLEAN.
- Empty cells are considered to be DOUBLE by default, unless the `empty_as_varchar` option is set to `true`, in which case they are typed as VARCHAR.

If the `all_varchar` option is set to `true`, none of the above applies and all cells are read as VARCHAR.

When no types are specified explicitly, (e.g., when using the `read_xlsx` function instead of `COPY TO ... FROM '<file>.xlsx'`{:.language-sql .highlight}) the types of the resulting columns are inferred based on the first "data" row in the sheet, that is:

- If no explicit range is given
  - The first row after the header if a header is found or forced by the `header` option
  - The first non-empty row in the sheet if no header is found or forced
- If an explicit range is given
  - The second row of the range if a header is found in the first row or forced by the `header` option
  - The first row of the range if no header is found or forced

This can sometimes lead to issues if the first "data row" is not representative of the rest of the sheet (e.g., it contains empty cells) in which case the `ignore_errors` or `empty_as_varchar` options can be used to work around this.

However, when the `COPY TO ... FROM '<file>.xlsx'` syntax is used, no type inference is done and the types of the resulting columns are determined by the types of the columns in the table being copied to. All cells will simply be converted by casting from DOUBLE or VARCHAR to the target column type.

Writing XLSX Files

Writing `.xlsx` files is supported using the COPY statement with XLSX given as the format. The following additional parameters are supported.

Option	Type	Default	Description
`header`	BOOLEAN	`false`	Whether to write the column names as the first row in the sheet
`sheet`	VARCHAR	`Sheet1`	The name of the sheet in the xlsx file to write.
`sheet_row_limit`	INTEGER	`1048576`	The maximum number of rows in a sheet. An error is thrown if this limit is exceeded.

> **Warning.** Many tools only support a maximum of 1,048,576 rows in a sheet, so increasing the `sheet_row_limit` may render the resulting file unreadable by other software.

These are passed as options to the COPY statement after the FORMAT, e.g.:

```
CREATE TABLE test AS
    SELECT *
    FROM (VALUES (1, 2), (3, 4)) AS t(a, b);
COPY test TO 'test.xlsx' WITH (FORMAT xlsx, HEADER true);
```

Type Conversions

Because XLSX files only really support storing numbers or strings – the equivalent of VARCHAR and DOUBLE, the following type conversions are applied when writing XLSX files.

- Numeric types are cast to DOUBLE when writing to an XLSX file.
- Temporal types (TIMESTAMP, DATE, TIME, etc.) are converted to excel "serial" numbers, that is the number of days since 1900-01-01 for dates and the fraction of a day for times. These are then styled with a "number format" so that they appear as dates or times when opened in Excel.
- TIMESTAMP_TZ and TIME_TZ are cast to UTC TIMESTAMP and TIME respectively, with the timezone information being lost.
- BOOLEANs are converted to 1 and 0, with a "number format" applied to make them appear as TRUE and FALSE in Excel.
- All other types are cast to VARCHAR and then written as text cells.

Full-Text Search Extension

Full-Text Search is an extension to DuckDB that allows for search through strings, similar to SQLite's FTS5 extension.

Installing and Loading

The `fts` extension will be transparently autoloaded on first use from the official extension repository. If you would like to install and load it manually, run:

```
INSTALL fts;
LOAD fts;
```

Usage

The extension adds two PRAGMA statements to DuckDB: one to create, and one to drop an index. Additionally, a scalar macro `stem` is added, which is used internally by the extension.

PRAGMA create_fts_index

```
create_fts_index(input_table, input_id, *input_values, stemmer = 'porter',
                 stopwords = 'english', ignore = '(\\.|[^a-z])+',
                 strip_accents = 1, lower = 1, overwrite = 0)
```

PRAGMA that creates a FTS index for the specified table.

Name	Type	Description
`input_table`	VARCHAR	Qualified name of specified table, e.g., `'table_name'` or `'main.table_name'`
`input_id`	VARCHAR	Column name of document identifier, e.g., `'document_identifier'`
`input_values...`	VARCHAR	Column names of the text fields to be indexed (vararg), e.g., `'text_field_1'`, `'text_field_2'`, ..., `'text_field_N'`, or `'\*'` for all columns in input_table of type VARCHAR
`stemmer`	VARCHAR	The type of stemmer to be used. One of `'arabic'`, `'basque'`, `'catalan'`, `'danish'`, `'dutch'`, `'english'`, `'finnish'`, `'french'`, `'german'`, `'greek'`, `'hindi'`, `'hungarian'`, `'indonesian'`, `'irish'`, `'italian'`, `'lithuanian'`, `'nepali'`, `'norwegian'`, `'porter'`, `'portuguese'`, `'romanian'`, `'russian'`, `'serbian'`, `'spanish'`, `'swedish'`, `'tamil'`, `'turkish'`, or `'none'` if no stemming is to be used. Defaults to `'porter'`
`stopwords`	VARCHAR	Qualified name of table containing a single VARCHAR column containing the desired stopwords, or `'none'` if no stopwords are to be used. Defaults to `'english'` for a pre-defined list of 571 English stopwords
`ignore`	VARCHAR	Regular expression of patterns to be ignored. Defaults to `'(\\.|[^a-z])+'`, ignoring all escaped and non-alphabetic lowercase characters

Name	Type	Description
`strip_accents`	BOOLEAN	Whether to remove accents (e.g., convert á to a). Defaults to `1`
`lower`	BOOLEAN	Whether to convert all text to lowercase. Defaults to `1`
`overwrite`	BOOLEAN	Whether to overwrite an existing index on a table. Defaults to `0`

This PRAGMA builds the index under a newly created schema. The schema will be named after the input table: if an index is created on table `'main.table_name'`, then the schema will be named `'fts_main_table_name'`.

`PRAGMA drop_fts_index`

```
drop_fts_index(input_table)
```

Drops a FTS index for the specified table.

Name	Type	Description
`input_table`	VARCHAR	Qualified name of input table, e.g., `'table_name'` or `'main.table_name'`

`match_bm25` Function

```
match_bm25(input_id, query_string, fields := NULL, k := 1.2, b := 0.75, conjunctive := 0)
```

When an index is built, this retrieval macro is created that can be used to search the index.

Name	Type	Description
`input_id`	VARCHAR	Column name of document identifier, e.g., `'document_identifier'`
`query_string`	VARCHAR	The string to search the index for
`fields`	VARCHAR	Comma-separarated list of fields to search in, e.g., `'text_field_2, text_field_N'`. Defaults to `NULL` to search all indexed fields
k	DOUBLE	Parameter *k1* in the Okapi BM25 retrieval model. Defaults to `1.2`
b	DOUBLE	Parameter *b* in the Okapi BM25 retrieval model. Defaults to `0.75`
`conjunctive`	BOOLEAN	Whether to make the query conjunctive i.e., all terms in the query string must be present in order for a document to be retrieved

`stem` Function

```
stem(input_string, stemmer)
```

Reduces words to their base. Used internally by the extension.

Name	Type	Description
`input_string`	VARCHAR	The column or constant to be stemmed.

Name	Type	Description
`stemmer`	VARCHAR	The type of stemmer to be used. One of `'arabic'`, `'basque'`, `'catalan'`, `'danish'`, `'dutch'`, `'english'`, `'finnish'`, `'french'`, `'german'`, `'greek'`, `'hindi'`, `'hungarian'`, `'indonesian'`, `'irish'`, `'italian'`, `'lithuanian'`, `'nepali'`, `'norwegian'`, `'porter'`, `'portuguese'`, `'romanian'`, `'russian'`, `'serbian'`, `'spanish'`, `'swedish'`, `'tamil'`, `'turkish'`, or `'none'` if no stemming is to be used.

Example Usage

Create a table and fill it with text data:

```
CREATE TABLE documents (
    document_identifier VARCHAR,
    text_content VARCHAR,
    author VARCHAR,
    doc_version INTEGER
);
INSERT INTO documents
    VALUES ('doc1',
            'The mallard is a dabbling duck that breeds throughout the temperate.',
            'Hannes Mühleisen',
            3),
           ('doc2',
            'The cat is a domestic species of small carnivorous mammal.',
            'Laurens Kuiper',
            2
           );
```

Build the index, and make both the `text_content` and `author` columns searchable.

```
PRAGMA create_fts_index(
    'documents', 'document_identifier', 'text_content', 'author'
);
```

Search the `author` field index for documents that are authored by `Muhleisen`. This retrieves `doc1`:

```
SELECT document_identifier, text_content, score
FROM (
    SELECT *, fts_main_documents.match_bm25(
        document_identifier,
        'Muhleisen',
        fields := 'author'
    ) AS score
    FROM documents
) sq
WHERE score IS NOT NULL
  AND doc_version > 2
ORDER BY score DESC;
```

document_identifier	text_content	score
doc1	The mallard is a dabbling duck that breeds throughout the temperate.	0.0

Search for documents about `small cats`. This retrieves `doc2`:

```sql
SELECT document_identifier, text_content, score
FROM (
    SELECT *, fts_main_documents.match_bm25(
        document_identifier,
        'small cats'
    ) AS score
    FROM documents
) sq
WHERE score IS NOT NULL
ORDER BY score DESC;
```

document_identifier	text_content	score
doc2	The cat is a domestic species of small carnivorous mammal.	0.0

> **Warning.** The FTS index will not update automatically when input table changes. A workaround of this limitation can be recreating the index to refresh.

httpfs (HTTP and S3)

httpfs Extension for HTTP and S3 Support

The `httpfs` extension is an autoloadable extension implementing a file system that allows reading remote/writing remote files. For plain HTTP(S), only file reading is supported. For object storage using the S3 API, the `httpfs` extension supports reading/writing/globbing files.

Installation and Loading

The `httpfs` extension will be, by default, autoloaded on first use of any functionality exposed by this extension.

To manually install and load the `httpfs` extension, run:

```
INSTALL httpfs;
LOAD httpfs;
```

HTTP(S)

The `httpfs` extension supports connecting to HTTP(S) endpoints.

S3 API

The `httpfs` extension supports connecting to S3 API endpoints.

HTTP(S) Support

With the `httpfs` extension, it is possible to directly query files over the HTTP(S) protocol. This works for all files supported by DuckDB or its various extensions, and provides read-only access.

```
SELECT *
FROM 'https://domain.tld/file.extension';
```

Partial Reading

For CSV files, files will be downloaded entirely in most cases, due to the row-based nature of the format. For Parquet files, DuckDB supports partial reading, i.e., it can use a combination of the Parquet metadata and HTTP range requests to only download the parts of the file that are actually required by the query. For example, the following query will only read the Parquet metadata and the data for the `column_a` column:

```
SELECT column_a
FROM 'https://domain.tld/file.parquet';
```

In some cases, no actual data needs to be read at all as they only require reading the metadata:

```
SELECT count(*)
FROM 'https://domain.tld/file.parquet';
```

Scanning Multiple Files

Scanning multiple files over HTTP(S) is also supported:

```
SELECT *
FROM read_parquet([
    'https://domain.tld/file1.parquet',
    'https://domain.tld/file2.parquet'
]);
```

Authenticating

To authenticate for an HTTP(S) endpoint, create an HTTP secret using the Secrets Manager:

```
CREATE SECRET http_auth (
    TYPE http,
    BEARER_TOKEN '<token>'
);
```

Or:

```
CREATE SECRET http_auth (
    TYPE http,
    EXTRA_HTTP_HEADERS MAP {
        'Authorization': 'Bearer <token>'
    }
);
```

HTTP Proxy

DuckDB supports HTTP proxies.

You can add an HTTP proxy using the Secrets Manager:

```
CREATE SECRET http_proxy (
    TYPE http,
    HTTP_PROXY '<http_proxy_url>',
    HTTP_PROXY_USERNAME '<username>',
    HTTP_PROXY_PASSWORD '<password>'
);
```

Alternatively, you can add it via configuration options:

```
SET http_proxy = '<http_proxy_url>';
SET http_proxy_username = '<username>';
SET http_proxy_password = '<password>';
```

Using a Custom Certificate File

To use the `httpfs` extension with a custom certificate file, set the following configuration options prior to loading the extension:

```
LOAD httpfs;
SET ca_cert_file = '<certificate_file>';
SET enable_server_cert_verification = true;
```

Hugging Face Support

The `httpfs` extension introduces support for the `hf://` protocol to access data sets hosted in Hugging Face repositories. See the announcement blog post for details.

Usage

Hugging Face repositories can be queried using the following URL pattern:

```
hf://datasets/<my_username>/<my_dataset>/<path_to_file>
```

For example, to read a CSV file, you can use the following query:

```
SELECT *
FROM 'hf://datasets/datasets-examples/doc-formats-csv-1/data.csv';
```

Where:

- `datasets-examples` is the name of the user/organization
- `doc-formats-csv-1` is the name of the dataset repository
- `data.csv` is the file path in the repository

The result of the query is:

kind	sound
dog	woof
cat	meow
pokemon	pika
human	hello

To read a JSONL file, you can run:

```
SELECT *
FROM 'hf://datasets/datasets-examples/doc-formats-jsonl-1/data.jsonl';
```

Finally, for reading a Parquet file, use the following query:

```
SELECT *
FROM 'hf://datasets/datasets-examples/doc-formats-parquet-1/data/train-00000-of-00001.parquet';
```

Each of these commands reads the data from the specified file format and displays it in a structured tabular format. Choose the appropriate command based on the file format you are working with.

Creating a Local Table

To avoid accessing the remote endpoint for every query, you can save the data in a DuckDB table by running a `CREATE TABLE ... AS` command. For example:

```
CREATE TABLE data AS
    SELECT *
    FROM 'hf://datasets/datasets-examples/doc-formats-csv-1/data.csv';
```

Then, simply query the `data` table as follows:

```
SELECT *
FROM data;
```

Multiple Files

To query all files under a specific directory, you can use a glob pattern. For example:

```
SELECT count(*) AS count
FROM 'hf://datasets/cais/mmlu/astronomy/*.parquet';
```

count
173

By using glob patterns, you can efficiently handle large datasets and perform comprehensive queries across multiple files, simplifying your data inspections and processing tasks. Here, you can see how you can look for questions that contain the word “planet” in astronomy:

```
SELECT count(*) AS count
FROM 'hf://datasets/cais/mmlu/astronomy/*.parquet'
WHERE question LIKE '%planet%';
```

count
21

Versioning and Revisions

In Hugging Face repositories, dataset versions or revisions are different dataset updates. Each version is a snapshot at a specific time, allowing you to track changes and improvements. In git terms, it can be understood as a branch or specific commit.

You can query different dataset versions/revisions by using the following URL:

```
hf://datasets/<my_username>/<my_dataset>@<my_branch>/<path_to_file>
```

For example:

```
SELECT *
FROM 'hf://datasets/datasets-examples/doc-formats-csv-1@~parquet/**/*.parquet';
```

kind	sound
dog	woof

kind	sound
cat	meow
pokemon	pika
human	hello

The previous query will read all parquet files under the `~parquet` revision. This is a special branch where Hugging Face automatically generates the Parquet files of every dataset to enable efficient scanning.

Authentication

Configure your Hugging Face Token in the DuckDB Secrets Manager to access private or gated datasets. First, visit Hugging Face Settings – Tokens to obtain your access token. Second, set it in your DuckDB session using DuckDB's Secrets Manager. DuckDB supports two providers for managing secrets:

`CONFIG`

The user must pass all configuration information into the `CREATE SECRET` statement. To create a secret using the `CONFIG` provider, use the following command:

```
CREATE SECRET hf_token (
    TYPE huggingface,
    TOKEN 'your_hf_token'
);
```

`credential_chain`

Automatically tries to fetch credentials. For the Hugging Face token, it will try to get it from `~/.cache/huggingface/token`. To create a secret using the `credential_chain` provider, use the following command:

```
CREATE SECRET hf_token (
    TYPE huggingface,
    PROVIDER credential_chain
);
```

S3 API Support

The `httpfs` extension supports reading/writing/globbing files on object storage servers using the S3 API. S3 offers a standard API to read and write to remote files (while regular http servers, predating S3, do not offer a common write API). DuckDB conforms to the S3 API, that is now common among industry storage providers.

Platforms

The `httpfs` filesystem is tested with AWS S3, Minio, Google Cloud, and lakeFS. Other services that implement the S3 API (such as Cloudflare R2) should also work, but not all features may be supported.

The following table shows which parts of the S3 API are required for each `httpfs` feature.

Feature	Required S3 API features
Public file reads	HTTP Range requests
Private file reads	Secret key or session token authentication
File glob	ListObjectsV2
File writes	Multipart upload

Configuration and Authentication

The preferred way to configure and authenticate to S3 endpoints is to use secrets. Multiple secret providers are available.

To migrate from the deprecated S3 API, use a defined secret with a profile. See the "Loading a Secret Based on a Profile" section.

`config` Provider

The default provider, `config` (i.e., user-configured), allows access to the S3 bucket by manually providing a key. For example:

```
CREATE OR REPLACE SECRET secret (
    TYPE s3,
    PROVIDER config,
    KEY_ID '<AKIAIOSFODNN7EXAMPLE>',
    SECRET '<wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY>',
    REGION '<us-east-1>'
);
```

> **Tip.** If you get an IO Error (`Connection error for HTTP HEAD`), configure the endpoint explicitly via `ENDPOINT 's3.<your_region>.amazonaws.com'`{:.language-sql .highlight}.

Now, to query using the above secret, simply query any `s3://` prefixed file:

```
SELECT *
FROM 's3://<your-bucket>/<your_file>.parquet';
```

`credential_chain` Provider

The `credential_chain` provider allows automatically fetching credentials using mechanisms provided by the AWS SDK. For example, to use the AWS SDK default provider:

```
CREATE OR REPLACE SECRET secret (
    TYPE s3,
    PROVIDER credential_chain
);
```

Again, to query a file using the above secret, simply query any `s3://` prefixed file.

DuckDB also allows specifying a specific chain using the `CHAIN` keyword. This takes a semicolon-separated list (`a;b;c`) of providers that will be tried in order. For example:

```
CREATE OR REPLACE SECRET secret (
    TYPE s3,
    PROVIDER credential_chain,
    CHAIN 'env;config'
);
```

The possible values for `CHAIN` are the following:

- `config`
- `sts`
- `sso`
- `env`
- `instance`
- `process`

The `credential_chain` provider also allows overriding the automatically fetched config. For example, to automatically load credentials, and then override the region, run:

```
CREATE OR REPLACE SECRET secret (
    TYPE s3,
    PROVIDER credential_chain,
    CHAIN config,
    REGION '<eu-west-1>'
);
```

Loading a Secret Based on a Profile

To load credentials based on a profile which is not defined as a default from the `AWS_PROFILE` environment variable or as a default profile based on AWS SDK precedence, run:

```
CREATE OR REPLACE SECRET secret (
    TYPE s3,
    PROVIDER credential_chain,
    CHAIN config,
    PROFILE '<my_profile>'
);
```

This approach is equivalent to the deprecated S3 API's's method `load_aws_credentials('<my_profile>')`.

Overview of S3 Secret Parameters

Below is a complete list of the supported parameters that can be used for both the `config` and `credential_chain` providers:

Name	Description	Secret	Type	Default
`ENDPOINT`	Specify a custom S3 endpoint	S3, GCS, R2	`STRING`	`s3.amazonaws.com` for S3,
`KEY_ID`	The ID of the key to use	S3, GCS, R2	`STRING`	-
`REGION`	The region for which to authenticate (should match the region of the bucket to query)	S3, GCS, R2	`STRING`	`us-east-1`
`SECRET`	The secret of the key to use	S3, GCS, R2	`STRING`	-
`SESSION_TOKEN`	Optionally, a session token can be passed to use temporary credentials	S3, GCS, R2	`STRING`	-
`URL_COMPATIBILITY_MODE`	Can help when URLs contain problematic characters	S3, GCS, R2	`BOOLEAN`	`true`
`URL_STYLE`	Either `vhost` or `path`	S3, GCS, R2	`STRING`	`vhost` for S3, `path` for R2 and GCS
`USE_SSL`	Whether to use HTTPS or HTTP	S3, GCS, R2	`BOOLEAN`	`true`

Name	Description	Secret	Type	Default
ACCOUNT_ID	The R2 account ID to use for generating the endpoint URL	R2	STRING	-
KMS_KEY_ID	AWS KMS (Key Management Service) key for Server Side Encryption S3	S3	STRING	-

Platform-Specific Secret Types

S3 Secrets

The httpfs extension supports Server Side Encryption via the AWS Key Management Service (KMS) on S3 using the KMS_KEY_ID option:

```
CREATE OR REPLACE SECRET secret (
    TYPE s3,
    PROVIDER credential_chain,
    CHAIN config,
    REGION '<eu-west-1>',
    KMS_KEY_ID 'arn:aws:kms:<region>:<account_id>:<key>/<key_id>',
    SCOPE 's3://<bucket-sub-path>'
);
```

R2 Secrets

While Cloudflare R2 uses the regular S3 API, DuckDB has a special Secret type, R2, to make configuring it a bit simpler:

```
CREATE OR REPLACE SECRET secret (
    TYPE r2,
    KEY_ID '<AKIAIOSFODNN7EXAMPLE>',
    SECRET '<wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY>',
    ACCOUNT_ID '<my_account_id>'
);
```

Note the addition of the ACCOUNT_ID which is used to generate the correct endpoint URL for you. Also note that R2 Secrets can also use both the CONFIG and credential_chain providers. However, since DuckDB uses an AWS client internally, when using credential_chain, the client will search for AWS credentials in the standard AWS credential locations (environment variables, credential files, etc.). Therefore, your R2 credentials must be made available as AWS environment variables (AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY) for the credential chain to work properly. Finally, R2 secrets are only available when using URLs starting with r2://, for example:

```
SELECT *
FROM read_parquet('r2://<some-file-that-uses-an-r2-secret>.parquet');
```

GCS Secrets

While Google Cloud Storage is accessed by DuckDB using the S3 API, DuckDB has a special Secret type, GCS, to make configuring it a bit simpler:

```
CREATE OR REPLACE SECRET secret (
    TYPE gcs,
    KEY_ID '<my_hmac_access_id>',
    SECRET '<my_hmac_secret_key>'
);
```

Important: The `KEY_ID` and `SECRET` values must be HMAC keys generated specifically for Google Cloud Storage interoperability. These are not the same as regular GCP service account keys or access tokens. You can create HMAC keys by following the Google Cloud documentation for managing HMAC keys.

Note that the above secret will automatically have the correct Google Cloud Storage endpoint configured. Also note that GCS Secrets can also use both the `CONFIG` and `credential_chain` providers. However, since DuckDB uses an AWS client internally, when using `credential_chain`, the client will search for AWS credentials in the standard AWS credential locations (environment variables, credential files, etc.). Therefore, your GCS HMAC keys must be made available as AWS environment variables (`AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`) for the credential chain to work properly. Finally, GCS secrets are only available when using URLs starting with `gcs://` or `gs://`, for example:

```
SELECT *
FROM read_parquet('gcs://<some/file/that/uses/a/gcs/secret>.parquet');
```

Reading

Reading files from S3 is now as simple as:

```
SELECT *
FROM 's3://<your-bucket>/<filename>.<extension>';
```

Partial Reading

The `httpfs` extension supports partial reading from S3 buckets.

Reading Multiple Files

Multiple files are also possible, for example:

```
SELECT *
FROM read_parquet([
    's3://<your-bucket>/<filename-1>.parquet',
    's3://<your-bucket>/<filename-2>.parquet'
]);
```

Globbing

File globbing is implemented using the ListObjectsV2 API call and allows to use filesystem-like glob patterns to match multiple files, for example:

```
SELECT *
FROM read_parquet('s3://<your-bucket>/*.parquet');
```

This query matches all files in the root of the bucket with the Parquet extension.

Several features for matching are supported, such as `*` to match any number of any character, `?` for any single character or `[0-9]` for a single character in a range of characters:

```
SELECT count(*) FROM read_parquet('s3://<your-bucket>/folder*/100?/t[0-9].parquet');
```

A useful feature when using globs is the `filename` option, which adds a column named `filename` that encodes the file that a particular row originated from:

```
SELECT *
FROM read_parquet('s3://<your-bucket>/*.parquet', filename = true);
```

This could for example result in:

column_a	column_b	filename
1	examplevalue1	s3://bucket-name/file1.parquet
2	examplevalue1	s3://bucket-name/file2.parquet

Hive Partitioning

DuckDB also offers support for the Hive partitioning scheme, which is available when using HTTP(S) and S3 endpoints.

Writing

Writing to S3 uses the multipart upload API. This allows DuckDB to robustly upload files at high speed. Writing to S3 works for both CSV and Parquet:

```
COPY table_name TO 's3://<your-bucket>/<filename>.<extension>';
```

Partitioned copy to S3 also works:

```
COPY table TO 's3://<your-bucket>/partitioned' (
    FORMAT parquet,
    PARTITION_BY (<part_col_a>, <part_col_b>)
);
```

An automatic check is performed for existing files/directories, which is currently quite conservative (and on S3 will add a bit of latency). To disable this check and force writing, an `OVERWRITE_OR_IGNORE` flag is added:

```
COPY table TO 's3://<your-bucket>/partitioned' (
    FORMAT parquet,
    PARTITION_BY (<part_col_a>, <part_col_b>),
    OVERWRITE_OR_IGNORE true
);
```

The naming scheme of the written files looks like this:

```
s3://<your-bucket>/partitioned/part_col_a=<val>/part_col_b=<val>/data_<thread_number>.parquet
```

Configuration

Some additional configuration options exist for the S3 upload, though the default values should suffice for most use cases.

Name	Description
`s3_uploader_max_parts_per_file`	Used for part size calculation, see AWS docs
`s3_uploader_max_filesize`	Used for part size calculation, see AWS docs
`s3_uploader_thread_limit`	Maximum number of uploader threads

Legacy Authentication Scheme for S3 API

Prior to version 0.10.0, DuckDB did not have a Secrets manager. Hence, the configuration of and authentication to S3 endpoints was handled via variables. This page documents the legacy authentication scheme for the S3 API.

> **Warning.** This page describes a legacy method to store secrets as DuckDB settings. This increases the risk of accidentally leaking secrets (e.g., by printing their values). Therefore, avoid using these methods for storing secrets. The recommended way to configuration and authentication of S3 endpoints is to use secrets.

Legacy Authentication Scheme

To be able to read or write from S3, the correct region should be set:

```
SET s3_region = 'us-east-1';
```

Optionally, the endpoint can be configured in case a non-AWS object storage server is used:

```
SET s3_endpoint = '<domain>.<tld>:<port>';
```

If the endpoint is not SSL-enabled then run:

```
SET s3_use_ssl = false;
```

Switching between path-style and vhost-style URLs is possible using:

```
SET s3_url_style = 'path';
```

However, note that this may also require updating the endpoint. For example for AWS S3 it is required to change the endpoint to `s3.<region>.amazonaws.com`{:.language-sql .highlight}.

After configuring the correct endpoint and region, public files can be read. To also read private files, authentication credentials can be added:

```
SET s3_access_key_id = '<aws_access_key_id>';
SET s3_secret_access_key = '<aws_secret_access_key>';
```

Alternatively, temporary S3 credentials are also supported. They require setting an additional session token:

```
SET s3_session_token = '<aws_session_token>';
```

The aws extension allows for loading AWS credentials.

Per-Request Configuration

Aside from the global S3 configuration described above, specific configuration values can be used on a per-request basis. This allows for use of multiple sets of credentials, regions, etc. These are used by including them on the S3 URI as query parameters. All the individual configuration values listed above can be set as query parameters. For instance:

```
SELECT *
FROM 's3://bucket/file.parquet?s3_access_key_id=accessKey&s3_secret_access_key=secretKey';
```

Multiple configurations per query are also allowed:

```
SELECT *
FROM 's3://bucket/file.parquet?s3_access_key_id=accessKey1&s3_secret_access_key=secretKey1' t1
INNER JOIN 's3://bucket/file.csv?s3_access_key_id=accessKey2&s3_secret_access_key=secretKey2' t2;
```

Configuration

Some additional configuration options exist for the S3 upload, though the default values should suffice for most use cases.

Additionally, most of the configuration options can be set via environment variables:

DuckDB setting	Environment variable	Note
`s3_region`	`AWS_REGION`	Takes priority over `AWS_DEFAULT_REGION`
`s3_region`	`AWS_DEFAULT_REGION`	
`s3_access_key_id`	`AWS_ACCESS_KEY_ID`	
`s3_secret_access_key`	`AWS_SECRET_ACCESS_KEY`	
`s3_session_token`	`AWS_SESSION_TOKEN`	
`s3_endpoint`	`DUCKDB_S3_ENDPOINT`	
`s3_use_ssl`	`DUCKDB_S3_USE_SSL`	

Iceberg

Iceberg Extension

The `iceberg` extension implements support for the Apache Iceberg open table format and can connect to Iceberg REST Catalogs. For information on how to connect to an Iceberg REST Catalog, please see the Iceberg REST Catalogs page.

Installing and Loading

To install the `iceberg` extension, run:

```
INSTALL iceberg;
```

Note that the `iceberg` extension is not autoloadable. Therefore, you need to load it before using it:

```
LOAD iceberg;
```

Updating the Extension

The `iceberg` extension often receives updates between DuckDB releases. To make sure that you have the latest version, update your extensions:

```
UPDATE EXTENSIONS;
```

Usage

To test the examples, download the `iceberg_data.zip` file and unzip it.

Common Parameters

Parameter	Type	Default	Description
`allow_moved_paths`	`BOOLEAN`	`false`	Allows scanning Iceberg tables that are moved
`metadata_compression_codec`	`VARCHAR`	`''`	Treats metadata files as when set to `'gzip'`
`snapshot_from_id`	`UBIGINT`	`NULL`	Access snapshot with a specific `id`
`snapshot_from_timestamp`	`TIMESTAMP`	`NULL`	Access snapshot with a specific `timestamp`
`version`	`VARCHAR`	`'?'`	Provides an explicit version string, hint file or guessing
`version_name_format`	`VARCHAR`	`'v%s%s.metadata.json,%s%s.metadata.json'`	Controls how versions are converted to metadata file names

Querying Individual Tables

```
SELECT count(*)
FROM iceberg_scan('data/iceberg/lineitem_iceberg', allow_moved_paths = true);
```

count_star()
51793

> The `allow_moved_paths` option ensures that some path resolution is performed, which allows scanning Iceberg tables that are moved.

You can also address specify the current manifest directly in the query, this may be resolved from the catalog prior to the query, in this example the manifest version is a UUID. To do so, navigate to the `data/iceberg` directory and run:

```
SELECT count(*)
FROM iceberg_scan('lineitem_iceberg/metadata/v1.metadata.json');
```

count_star()
60175

The `iceberg` works together with the `httpfs` extension or the `azure` extension to access Iceberg tables in object stores such as S3 or Azure Blob Storage.

```
SELECT count(*)
FROM iceberg_scan('s3://bucketname/lineitem_iceberg/metadata/v1.metadata.json');
```

Access Iceberg Metadata

To access Iceberg Metadata, you can use the `iceberg_metadata` function:

```
SELECT *
FROM iceberg_metadata('data/iceberg/lineitem_iceberg', allow_moved_paths = true);
```

You can also run the `iceberg_metadata` function on Iceberg tables attached via the REST Catalog:

```
SELECT *
FROM iceberg_metadata(iceberg_table);
```

manifest_path	manifest_sequence_number	manifest_content	status	content	file_path	file_format	record_count
lineitem_iceberg/metadata/10eaca8a-1e1c-421e-ad6d-b232e5ee23d3-m1.avro	2	DATA	ADDED	EXISTING	lineitem_iceberg/data/00041-414-f3c73457-bbd6-4b92-9c15-17b241171b16-00001.parquet	PARQUET	51793
lineitem_iceberg/metadata/10eaca8a-1e1c-421e-ad6d-b232e5ee23d3-m0.avro	2	DATA	DELETED	EXISTING	lineitem_iceberg/data/00000-411-0792dcfe-4e25-4ca3-8ada-175286069a47-00001.parquet	PARQUET	60175

Visualizing Snapshots

To visualize the snapshots in an Iceberg table, use the `iceberg_snapshots` function:

```
SELECT *
FROM iceberg_snapshots('data/iceberg/lineitem_iceberg');
```

You can also run the `iceberg_snapshots` function on Iceberg tables attached via the REST Catalog:

```
SELECT *
FROM iceberg_snapshots(iceberg_table);
```

sequence_number	snapshot_id	timestamp_ms	manifest_list
1	3776207205136740581	2023-02-15 15:07:54.504	lineitem_iceberg/metadata/snap-3776207205136740581-1-cf3d0be5-cf70-453d-ad8f-48fdc412e608.avro
2	7635660646343998149	2023-02-15 15:08:14.73	lineitem_iceberg/metadata/snap-7635660646343998149-1-10eaca8a-1e1c-421e-ad6d-b232e5ee23d3.avro

Selecting Metadata Versions

By default, the `iceberg` extension will look for a `version-hint.text` file to identify the proper metadata version to use. This can be overridden by explicitly supplying a version number via the `version` parameter to the functions of the `iceberg` extension:

```
SELECT *
FROM iceberg_snapshots(
    'data/iceberg/lineitem_iceberg',
    version = '1',
    allow_moved_paths = true
);
```

By default, `iceberg` functions will look for both `v{version}.metadata.json` and `{version}.metadata.json` files, or `v{version}.gz.metadata.json` and `{version}.gz.metadata.json` when `metadata_compression_codec = 'gzip'` is specified. Other compression codecs are not supported.

If any text file is provided through the `version` parameter, it is opened and treated as a version hint file:

```
SELECT *
FROM iceberg_snapshots(
    'data/iceberg/lineitem_iceberg',
    version = 'version-hint.txt',
    allow_moved_paths = true
);
```

The `iceberg` extension will open this file and use the **entire content** of the file as a provided version number. Note that the entire content of the `version-hint.txt` file will be treated as a literal version name, with no encoding, escaping or trimming. This includes any whitespace, or unsafe characters which will be explicitly passed formatted into filenames in the logic described below.

Working with Alternative Metadata Naming Conventions

The `iceberg` extension can handle different metadata naming conventions by specifying them as a comma-delimited list of format strings via the `version_name_format` parameter. Each format string must contain two `%s` parameters. The first is the location of the version number in the metadata filename and the second is the location of the filename extension specified by the `metadata_compression_codec`. The behavior described above is provided by the default value of `"v%s%s.metadata.gz,%s%smetadata.gz`. If you had an alternatively named metadata file, e.g., `rev-2.metadata.json.gz`, the table can be read via the follow statement:

```
SELECT *
FROM iceberg_snapshots(
    'data/iceberg/alternative_metadata_gz_naming',
    version = '2',
    version_name_format = 'rev-%s.metadata.json%s',
    metadata_compression_codec = 'gzip',
    allow_moved_paths = true
);
```

"Guessing" Metadata Versions

By default, either a table version number or a `version-hint.text` **must** be provided for the `iceberg` extension to read a table. This is typically provided by an external data catalog. In the event neither is present, the `iceberg` extension can attempt to guess the latest version by passing `?` as the `version` parameter:

```
SELECT count(*)
FROM iceberg_scan(
    'data/iceberg/lineitem_iceberg_no_hint',
    version = '?',
    allow_moved_paths = true
);
```

The "latest" version is assumed to be the filename that is lexicographically largest when sorting the filenames. Collations are not considered. This behavior is not enabled by default as it may potentially violate ACID constraints. It can be enabled by setting `unsafe_enable_version_guessing` to `true`. When this is set, `iceberg` functions will attempt to guess the latest version by default before failing.

```
SET unsafe_enable_version_guessing = true;
SELECT count(*)
FROM iceberg_scan(
    'data/iceberg/lineitem_iceberg_no_hint',
    allow_moved_paths = true
);
```

Limitations

Writing (i.e., exporting to) Iceberg files is currently not supported.

Iceberg REST Catalogs

The `iceberg` extension supports attaching Iceberg REST Catalogs. Before attaching an Iceberg REST Catalog, you must install the `iceberg` extension by following the instructions located in the overview.

If you are attaching to an Iceberg REST Catalog managed by Amazon, please see the instructions for attaching to Amazon S3 tables or Amazon Sagemaker Lakehouse.

For all other Iceberg REST Catalogs, you can follow the instructions below. Please see the Examples section for questions about specific catalogs.

Most Iceberg REST Catalogs authenticate via OAuth2. You can use the existing DuckDB secret workflow to store login credentials for the OAuth2 service.

```
CREATE SECRET iceberg_secret (
    TYPE ICEBERG,
    CLIENT_ID '<admin>',
    CLIENT_SECRET '<password>',
    OAUTH2_SERVER_URI '<http://irc_host_url.com/v1/oauth/tokens>'
);
```

If you already have a Bearer token, you can pass it directly to your `CREATE SECRET` statement

```
CREATE SECRET iceberg_secret (
    TYPE ICEBERG,
    TOKEN '<bearer_token>'
);
```

You can attach the Iceberg catalog with the following `ATTACH` statement.

```
ATTACH '<warehouse>' AS iceberg_catalog (
   TYPE iceberg,
   SECRET iceberg_secret, -- pass a specific secret name to prevent ambiguity
   ENDPOINT <https://rest_endpoint.com>
);
```

To see the available tables run

```
SHOW ALL TABLES;
```

ATTACH OPTIONS

A REST Catalog with OAuth2 authorization can also be attached with just an `ATTACH` statement. See the complete list of `ATTACH` options for a REST catalog below.

Parameter	Type	Default	Description
`ENDPOINT_TYPE`	`VARCHAR`	`NULL`	Used for attaching S3Tables or Glue catalogs. Allowed values are 'GLUE' and 'S3_TABLES'
`ENDPOINT`	`VARCHAR`	`NULL`	URL endpoint to communicate with the REST Catalog. Cannot be used in conjunction with `ENDPOINT_TYPE`
`SECRET`	`VARCHAR`	`NULL`	Name of secret used to communicate with the REST Catalog
`CLIENT_ID`	`VARCHAR`	`NULL`	CLIENT_ID used for Secret
`CLIENT_SECRET`	`VARCHAR`	`NULL`	CLIENT_SECRET needed for Secret
`DEFAULT_REGION`	`VARCHAR`	`NULL`	A Default region to use when communicating with the storage layer
`OAUTH2_SERVER_URI`	`VARCHAR`	`NULL`	OAuth2 server url for getting a Bearer Token
`AUTHORIZATION_TYPE`	`VARCHAR`	`OAUTH2`	Pass `SigV4` for Catalogs the require SigV4 authorization

The following options can only be passed to a `CREATE SECRET` statement, and they require `AUTHORIZATION_TYPE` to be `OAUTH2`

Parameter	Type	Default	Description
`OAUTH2_GRANT_TYPE`	`VARCHAR`	`NULL`	Grant Type when requesting an OAuth Token
`OAUTH2_SCOPE`	`VARCHAR`	`NULL`	Requested scope for the returned OAuth Access Token

Specific Catalog Examples

R2 Catalog

To attach to an R2 cloudflare managed catalog follow the attach steps below.

```
CREATE SECRET r2_secret (
    TYPE ICEBERG,
    TOKEN '<r2_token>'
);
```

You can create a token by following the create an API token steps in getting started.

Then, attach the catalog with the following commands.

```
ATTACH '<warehouse>' AS my_r2_catalog (
    TYPE ICEBERG,
    ENDPOINT '<catalog-uri>'
);
```

The variables for `warehouse` and `catalog-uri` will be available under the settings of the desired R2 Object Storage Catalog (R2 Object Store > Catalog name > Settings).

Polaris

To attach to a Polaris catalog the following commands will work.

```
CREATE SECRET polaris_secret (
    TYPE ICEBERG,
    CLIENT_ID '<admin>',
    CLIENT_SECRET '<password>',
);
```

```
ATTACH 'quickstart_catalog' as polaris_catalog (
    TYPE ICEBERG,
    ENDPOINT '<polaris_rest_catalog_endpoint>'
);
```

Lakekeeper

To attach to a Lakekeeper catalog the following commands will work.

```
CREATE SECRET lakekeeper_secret (
    TYPE ICEBERG,
    CLIENT_ID '<admin>',
    CLIENT_SECRET '<password>',
    OAUTH2_SCOPE '<scope>',
    OAUTH2_SERVER_URI '<lakekeeper_oauth_url>'
);
```

```
ATTACH '<warehouse>' as lakekeeper_catalog (
    TYPE ICEBERG,
    ENDPOINT '<lakekeeper_irc_url>',
    SECRET lakekeeper_secret
);
```

Limitations

Reading from Iceberg REST Catalogs backed by remote storage that is not S3 or S3Tables is not yet supported.

Amazon S3 Tables

> Support for S3 Tables is currently experimental.

The `iceberg` extension supports reading Iceberg tables stored in Amazon S3 Tables.

Requirements

Install the following extensions:

```
INSTALL aws;
INSTALL httpfs;
INSTALL iceberg;
```

Connecting to Amazon S3 Tables

You can let DuckDB detect your AWS credentials and configuration based on the default profile in your `~/.aws` directory by creating the following secret using the Secrets Manager:

```
CREATE SECRET (
    TYPE s3,
    PROVIDER credential_chain
);
```

Alternatively, you can set the values manually:

```
CREATE SECRET (
    TYPE s3,
    KEY_ID '<AKIAIOSFODNN7EXAMPLE>',
    SECRET '<wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY>',
    REGION '<us-east-1>'
);
```

Then, connect to the catalog using you S3 Tables ARN (available in the AWS Management Console) and the `ENDPOINT_TYPE s3_tables` option:

```
ATTACH '<s3_tables_arn>' AS s3_tables (
   TYPE iceberg,
   ENDPOINT_TYPE s3_tables
);
```

To check whether the attachment worked, list all tables:

```
SHOW ALL TABLES;
```

You can query a table as follows:

```
SELECT count(*)
FROM s3_tables.<namespace_name>.<table_name>;
```

Amazon SageMaker Lakehouse (AWS Glue)

> Support for Amazon SageMaker Lakehouse (AWS Glue) is currently experimental.

The `iceberg` extension supports reading Iceberg tables through the Amazon SageMaker Lakehouse (a.k.a. AWS Glue) catalog. Make sure you have the Iceberg extension installed by following the steps at Installing and Loading

Connecting to Amazon SageMaker Lakehouse

Configure your role, region, and credential provider (or explicit credentials) using the Secrets Manager:

```
CREATE SECRET (
    TYPE s3,
    PROVIDER credential_chain,
    CHAIN sts,
    ASSUME_ROLE_ARN 'arn:aws:iam::<account_id>:role/<role>',
    REGION 'us-east-2'
);
```

Then, connect to the catalog using the `ENDPOINT_TYPE glue` option:

```
ATTACH '<account_id>:s3tablescatalog/<namespace_name>' AS glue_catalog (
    TYPE iceberg,
    ENDPOINT_TYPE glue
);
```

To check whether the attachment worked, list all tables:

```
SHOW ALL TABLES;
```

You can query a table as follows:

```
SELECT count(*)
FROM glue_catalog.<namespace_name>.<table_name>;
```

Troubleshooting

Limitations

- Writing Iceberg tables is currently not supported.
- Reading tables with deletes is not yet supported.

Curl Request Fails

Problem

When trying to attach to an Iceberg REST Catalog endpoint, DuckDB returns the following error:

```
IO Error:
Curl Request to '/v1/oauth/tokens' failed with error: 'URL using bad/illegal format or missing URL'
```

Solution

Make sure that you have the latest Iceberg extension installed:

```
duckdb
```

```
FORCE INSTALL iceberg FROM core_nightly;
```

Exit DuckDB and start a new session:

```
duckdb
```

```
LOAD iceberg;
```

HTTP Error 403

Problem

When trying to list the tables in a remote-attached catalog, DuckDB returns the following error:

```
SHOW ALL TABLES;
```

```
Failed to query https://s3tables.us-east-2.amazonaws.com/iceberg/v1/arn:aws:s3tables:... http error 403 thrown.
Message: {"message":"The security token included in the request is invalid."}
```

Solution

Use the `duckdb_secrets()` function to check whether DuckDB loaded the required credentials:

```
.mode line
FROM duckdb_secrets();
```

If you do not see your credentials, set them manually using the following secret:

```
CREATE SECRET (
    TYPE s3,
    KEY_ID '<AKIAIOSFODNN7EXAMPLE>',
    SECRET '<wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY>',
    REGION '<us-east-1>'
);
```

ICU Extension

The `icu` extension contains an easy-to-use version of the collation/timezone part of the ICU library.

Installing and Loading

The `icu` extension will be transparently autoloaded on first use from the official extension repository. If you would like to install and load it manually, run:

```
INSTALL icu;
LOAD icu;
```

Features

The `icu` extension introduces the following features:

- Region-dependent collations
- Time zones, used for timestamp data types and timestamp functions

inet Extension

The `inet` extension defines the `INET` data type for storing IPv4 and IPv6 Internet addresses. It supports the CIDR notation for subnet masks (e.g., `198.51.100.0/22`, `2001:db8:3c4d::/48`).

Installing and Loading

The `inet` extension will be transparently autoloaded on first use from the official extension repository. If you would like to install and load it manually, run:

```
INSTALL inet;
LOAD inet;
```

Examples

```
SELECT '127.0.0.1'::INET AS ipv4, '2001:db8:3c4d::/48'::INET AS ipv6;
```

ipv4	ipv6
127.0.0.1	2001:db8:3c4d::/48

```
CREATE TABLE tbl (id INTEGER, ip INET);
INSERT INTO tbl VALUES
    (1, '192.168.0.0/16'),
    (2, '127.0.0.1'),
    (3, '8.8.8.8'),
    (4, 'fe80::/10'),
    (5, '2001:db8:3c4d:15::1a2f:1a2b');
SELECT * FROM tbl;
```

id	ip
1	192.168.0.0/16
2	127.0.0.1
3	8.8.8.8
4	fe80::/10
5	2001:db8:3c4d:15::1a2f:1a2b

Operations on `INET` Values

`INET` values can be compared naturally, and IPv4 will sort before IPv6. Additionally, IP addresses can be modified by adding or subtracting integers.

```
CREATE TABLE tbl (cidr INET);
INSERT INTO tbl VALUES
    ('127.0.0.1'::INET + 10),
    ('fe80::10'::INET - 9),
    ('127.0.0.1'),
    ('2001:db8:3c4d:15::1a2f:1a2b');
SELECT cidr FROM tbl ORDER BY cidr ASC;
```

cidr
127.0.0.1
127.0.0.11
2001:db8:3c4d:15::1a2f:1a2b
fe80::7

`host` Function

The host component of an `INET` value can be extracted using the `HOST()` function.

```
CREATE TABLE tbl (cidr INET);
INSERT INTO tbl VALUES
    ('192.168.0.0/16'),
    ('127.0.0.1'),
    ('2001:db8:3c4d:15::1a2f:1a2b/96');
SELECT cidr, host(cidr) FROM tbl;
```

cidr	host(cidr)
192.168.0.0/16	192.168.0.0
127.0.0.1	127.0.0.1
2001:db8:3c4d:15::1a2f:1a2b/96	2001:db8:3c4d:15::1a2f:1a2b

`netmask` Function

Computes the network mask for the address's network.

```
CREATE TABLE tbl (cidr INET);
INSERT INTO tbl VALUES
    ('192.168.1.5/24'),
    ('127.0.0.1'),
    ('2001:db8:3c4d:15::1a2f:1a2b/96');
SELECT cidr, netmask(cidr) FROM tbl;
```

cidr	netmask(cidr)
192.168.1.5/24	255.255.255.0/24
127.0.0.1	255.255.255.255
2001:db8:3c4d:15::1a2f:1a2b/96	ffff:ffff:ffff:ffff:ffff:ffff::/96

`network` Function

Returns the network part of the address, zeroing out whatever is to the right of the netmask.

```
CREATE TABLE tbl (cidr INET);
INSERT INTO tbl VALUES
    ('192.168.1.5/24'),
    ('127.0.0.1'),
    ('2001:db8:3c4d:15::1a2f:1a2b/96');
SELECT cidr, network(cidr) FROM tbl;
```

cidr	network(cidr)
192.168.1.5/24	192.168.1.0/24
127.0.0.1	255.255.255.255
2001:db8:3c4d:15::1a2f:1a2b/96	ffff:ffff:ffff:ffff:ffff:ffff::/96

`broadcast` Function

Computes the broadcast address for the address's network.

```
CREATE TABLE tbl (cidr INET);
INSERT INTO tbl VALUES
    ('192.168.1.5/24'),
    ('127.0.0.1'),
    ('2001:db8:3c4d:15::1a2f:1a2b/96');
SELECT cidr, broadcast(cidr) FROM tbl;
```

cidr	broadcast(cidr)
192.168.1.5/24	192.168.1.0/24
127.0.0.1	127.0.0.1
2001:db8:3c4d:15::1a2f:1a2b/96	2001:db8:3c4d:15::/96

<<= Predicate

Is subnet contained by or equal to subnet?

```
CREATE TABLE tbl (cidr INET);
INSERT INTO tbl VALUES
    ('192.168.1.0/24'),
    ('127.0.0.1'),
    ('2001:db8:3c4d:15::1a2f:1a2b/96');
SELECT cidr, INET '192.168.1.5/32' <<= cidr FROM tbl;
```

cidr	(CAST('192.168.1.5/32' AS INET) «= cidr)
192.168.1.5/24	true
127.0.0.1	false
2001:db8:3c4d:15::1a2f:1a2b/96	false

>>= Predicate

Does subnet contain or equal subnet?

```
CREATE TABLE tbl (cidr INET);
INSERT INTO tbl VALUES
    ('192.168.1.0/24'),
    ('127.0.0.1'),
    ('2001:db8:3c4d:15::1a2f:1a2b/96');
SELECT cidr, INET '192.168.0.0/16' >>= cidr FROM tbl;
```

cidr	(CAST('192.168.0.0/16' AS INET) »= cidr)
192.168.1.5/24	true
127.0.0.1	false
2001:db8:3c4d:15::1a2f:1a2b/96	false

HTML Escape and Unescape Functions

```
SELECT html_escape('&');
```

```
┌──────────────────┐
│ html_escape('&') │
│     varchar      │
├──────────────────┤
│ &amp;            │
└──────────────────┘
```

```
SELECT html_unescape('&amp;');
```

```
┌──────────────────────────┐
│ html_unescape('&amp;') │
│         varchar          │
├──────────────────────────┤
│ &                        │
└──────────────────────────┘
```

jemalloc Extension

The `jemalloc` extension replaces the system's memory allocator with jemalloc. Unlike other DuckDB extensions, the `jemalloc` extension is statically linked and cannot be installed or loaded during runtime.

Operating System Support

The availability of the `jemalloc` extension depends on the operating system.

Linux

Linux distributions of DuckDB ships with the `jemalloc` extension. To disable the `jemalloc` extension, build DuckDB from source and set the `SKIP_EXTENSIONS` flag as follows:

```
GEN=ninja SKIP_EXTENSIONS="jemalloc" make
```

macOS

The macOS version of DuckDB does not ship with the `jemalloc` extension but can be built from source to include it:

```
GEN=ninja BUILD_JEMALLOC=1 make
```

Windows

On Windows, this extension is not available.

Configuration

Environment Variables

The jemalloc allocator in DuckDB can be configured via the `MALLOC_CONF` environment variable.

Background Threads

By default, jemalloc's background threads are disabled. To enable them, use the following configuration option:

```
SET allocator_background_threads = true;
```

Background threads asynchronously purge outstanding allocations so that this doesn't have to be done synchronously by the foreground threads. This improves allocation performance, and should be noticeable in allocation-heavy workloads, especially on many-core CPUs.

MySQL Extension

The `mysql` extension allows DuckDB to directly read and write data from/to a running MySQL instance. The data can be queried directly from the underlying MySQL database. Data can be loaded from MySQL tables into DuckDB tables, or vice versa.

Installing and Loading

To install the `mysql` extension, run:

```
INSTALL mysql;
```

The extension is loaded automatically upon first use. If you prefer to load it manually, run:

```
LOAD mysql;
```

Reading Data from MySQL

To make a MySQL database accessible to DuckDB use the `ATTACH` command with the `mysql` or the `mysql_scanner` type:

```
ATTACH 'host=localhost user=root port=0 database=mysql' AS mysqldb (TYPE mysql);
USE mysqldb;
```

Configuration

The connection string determines the parameters for how to connect to MySQL as a set of `key=value` pairs. Any options not provided are replaced by their default values, as per the table below. Connection information can also be specified with environment variables. If no option is provided explicitly, the MySQL extension tries to read it from an environment variable.

Setting	Default	Environment variable
database	NULL	MYSQL_DATABASE
host	localhost	MYSQL_HOST
password		MYSQL_PWD
port	0	MYSQL_TCP_PORT
socket	NULL	MYSQL_UNIX_PORT
user	*current user*	MYSQL_USER
ssl_mode	preferred	
ssl_ca		
ssl_capath		
ssl_cert		
ssl_cipher		
ssl_crl		

Setting	Default	Environment variable
ssl_crlpath		
ssl_key		

Configuring via Secrets

MySQL connection information can also be specified with secrets. The following syntax can be used to create a secret.

```
CREATE SECRET (
    TYPE mysql,
    HOST '127.0.0.1',
    PORT 0,
    DATABASE mysql,
    USER 'mysql',
    PASSWORD ''
);
```

The information from the secret will be used when ATTACH is called. We can leave the connection string empty to use all of the information stored in the secret.

```
ATTACH '' AS mysql_db (TYPE mysql);
```

We can use the connection string to override individual options. For example, to connect to a different database while still using the same credentials, we can override only the database name in the following manner.

```
ATTACH 'database=my_other_db' AS mysql_db (TYPE mysql);
```

By default, created secrets are temporary. Secrets can be persisted using the `CREATE PERSISTENT SECRET` command. Persistent secrets can be used across sessions.

Managing Multiple Secrets

Named secrets can be used to manage connections to multiple MySQL database instances. Secrets can be given a name upon creation.

```
CREATE SECRET mysql_secret_one (
    TYPE mysql,
    HOST '127.0.0.1',
    PORT 0,
    DATABASE mysql,
    USER 'mysql',
    PASSWORD ''
);
```

The secret can then be explicitly referenced using the SECRET parameter in the ATTACH.

```
ATTACH '' AS mysql_db_one (TYPE mysql, SECRET mysql_secret_one);
```

SSL Connections

The `ssl` connection parameters can be used to make SSL connections. Below is a description of the supported parameters.

Setting	Description
ssl_mode	The security state to use for the connection to the server: `disabled, required, verify_ca, verify_identity or preferred` (default: `preferred`)
ssl_ca	The path name of the Certificate Authority (CA) certificate file
ssl_capath	The path name of the directory that contains trusted SSL CA certificate files
ssl_cert	The path name of the client public key certificate file
ssl_cipher	The list of permissible ciphers for SSL encryption
ssl_crl	The path name of the file containing certificate revocation lists
ssl_crlpath	The path name of the directory that contains files containing certificate revocation lists
ssl_key	The path name of the client private key file

Reading MySQL Tables

The tables in the MySQL database can be read as if they were normal DuckDB tables, but the underlying data is read directly from MySQL at query time.

```
SHOW ALL TABLES;
```

name
signed_integers

```
SELECT * FROM signed_integers;
```

t	s	m	i	b
-128	-32768	-8388608	-2147483648	-9223372036854775808
127	32767	8388607	2147483647	9223372036854775807
NULL	NULL	NULL	NULL	NULL

It might be desirable to create a copy of the MySQL databases in DuckDB to prevent the system from re-reading the tables from MySQL continuously, particularly for large tables.

Data can be copied over from MySQL to DuckDB using standard SQL, for example:

```
CREATE TABLE duckdb_table AS FROM mysqlscanner.mysql_table;
```

Writing Data to MySQL

In addition to reading data from MySQL, create tables, ingest data into MySQL and make other modifications to a MySQL database using standard SQL queries.

This allows you to use DuckDB to, for example, export data that is stored in a MySQL database to Parquet, or read data from a Parquet file into MySQL.

Below is a brief example of how to create a new table in MySQL and load data into it.

```
ATTACH 'host=localhost user=root port=0 database=mysqlscanner' AS mysql_db (TYPE mysql);
CREATE TABLE mysql_db.tbl (id INTEGER, name VARCHAR);
INSERT INTO mysql_db.tbl VALUES (42, 'DuckDB');
```

Many operations on MySQL tables are supported. All these operations directly modify the MySQL database, and the result of subsequent operations can then be read using MySQL. Note that if modifications are not desired, ATTACH can be run with the READ_ONLY property which prevents making modifications to the underlying database. For example:

```
ATTACH 'host=localhost user=root port=0 database=mysqlscanner' AS mysql_db (TYPE mysql, READ_ONLY);
```

Supported Operations

Below is a list of supported operations.

CREATE TABLE

```
CREATE TABLE mysql_db.tbl (id INTEGER, name VARCHAR);
```

INSERT INTO

```
INSERT INTO mysql_db.tbl VALUES (42, 'DuckDB');
```

SELECT

```
SELECT * FROM mysql_db.tbl;
```

id	name
42	DuckDB

COPY

```
COPY mysql_db.tbl TO 'data.parquet';
COPY mysql_db.tbl FROM 'data.parquet';
```

You may also create a full copy of the database using the COPY FROM DATABASE statement:

```
COPY FROM DATABASE mysql_db TO my_duckdb_db;
```

UPDATE

```
UPDATE mysql_db.tbl
SET name = 'Woohoo'
WHERE id = 42;
```

DELETE

```
DELETE FROM mysql_db.tbl
WHERE id = 42;
```

ALTER TABLE

```
ALTER TABLE mysql_db.tbl
ADD COLUMN k INTEGER;
```

DROP TABLE

```
DROP TABLE mysql_db.tbl;
```

CREATE VIEW

```
CREATE VIEW mysql_db.v1 AS SELECT 42;
```

CREATE SCHEMA and DROP SCHEMA

```
CREATE SCHEMA mysql_db.s1;
CREATE TABLE mysql_db.s1.integers (i INTEGER);
INSERT INTO mysql_db.s1.integers VALUES (42);
SELECT * FROM mysql_db.s1.integers;
```

i
42

```
DROP SCHEMA mysql_db.s1;
```

Transactions

```
CREATE TABLE mysql_db.tmp (i INTEGER);
BEGIN;
INSERT INTO mysql_db.tmp VALUES (42);
SELECT * FROM mysql_db.tmp;
```

This returns:

i
42

```
ROLLBACK;
SELECT * FROM mysql_db.tmp;
```

This returns an empty table.

> The DDL statements are not transactional in MySQL.

Running SQL Queries in MySQL

The `mysql_query` Table Function

The `mysql_query` table function allows you to run arbitrary read queries within an attached database. `mysql_query` takes the name of the attached MySQL database to execute the query in, as well as the SQL query to execute. The result of the query is returned. Single-quote strings are escaped by repeating the single quote twice.

```
mysql_query(attached_database::VARCHAR, query::VARCHAR)
```

For example:

```
ATTACH 'host=localhost database=mysql' AS mysqldb (TYPE mysql);
SELECT * FROM mysql_query('mysqldb', 'SELECT * FROM cars LIMIT 3');
```

The `mysql_execute` Function

The `mysql_execute` function allows running arbitrary queries within MySQL, including statements that update the schema and content of the database.

```
ATTACH 'host=localhost database=mysql' AS mysqldb (TYPE mysql);
CALL mysql_execute('mysqldb', 'CREATE TABLE my_table (i INTEGER)');
```

Settings

Name	Description	Default
`mysql_bit1_as_boolean`	Whether or not to convert `BIT(1)` columns to `BOOLEAN`	`true`
`mysql_debug_show_queries`	DEBUG SETTING: print all queries sent to MySQL to stdout	`false`
`mysql_experimental_filter_pushdown`	Whether or not to use filter pushdown (currently experimental)	`false`
`mysql_tinyint1_as_boolean`	Whether or not to convert `TINYINT(1)` columns to `BOOLEAN`	`true`

Schema Cache

To avoid having to continuously fetch schema data from MySQL, DuckDB keeps schema information – such as the names of tables, their columns, etc. – cached. If changes are made to the schema through a different connection to the MySQL instance, such as new columns being added to a table, the cached schema information might be outdated. In this case, the function `mysql_clear_cache` can be executed to clear the internal caches.

```
CALL mysql_clear_cache();
```

PostgreSQL Extension

The `postgres` extension allows DuckDB to directly read and write data from a running PostgreSQL database instance. The data can be queried directly from the underlying PostgreSQL database. Data can be loaded from PostgreSQL tables into DuckDB tables, or vice versa. See the official announcement for implementation details and background.

Installing and Loading

The `postgres` extension will be transparently autoloaded on first use from the official extension repository. If you would like to install and load it manually, run:

```
INSTALL postgres;
LOAD postgres;
```

Connecting

To make a PostgreSQL database accessible to DuckDB, use the ATTACH command with the `postgres` or `postgres_scanner` type.

To connect to the `public` schema of the PostgreSQL instance running on localhost in read-write mode, run:

```
ATTACH '' AS postgres_db (TYPE postgres);
```

To connect to the PostgreSQL instance with the given parameters in read-only mode, run:

```
ATTACH 'dbname=postgres user=postgres host=127.0.0.1' AS db (TYPE postgres, READ_ONLY);
```

By default, all schemas are attached. When working with large instances, it can be useful to only attach a specific schema. This can be accomplished using the SCHEMA command.

```
ATTACH 'dbname=postgres user=postgres host=127.0.0.1' AS db (TYPE postgres, SCHEMA 'public');
```

Configuration

The ATTACH command takes as input either a `libpq` connection string or a PostgreSQL URI.

Below are some example connection strings and commonly used parameters. A full list of available parameters can be found in the PostgreSQL documentation.

```
dbname=postgresscanner
host=localhost port=5432 dbname=mydb connect_timeout=10
```

Name	Description	Default
`dbname`	Database name	[user]
`host`	Name of host to connect to	`localhost`
`hostaddr`	Host IP address	`localhost`
`passfile`	Name of file passwords are stored in	`~/.pgpass`

Name	Description	Default
`password`	PostgreSQL password	(empty)
`port`	Port number	`5432`
`user`	PostgreSQL user name	*current user*

An example URI is `postgresql://username@hostname/dbname`.

Configuring via Secrets

PostgreSQL connection information can also be specified with secrets. The following syntax can be used to create a secret.

```
CREATE SECRET (
    TYPE postgres,
    HOST '127.0.0.1',
    PORT 5432,
    DATABASE postgres,
    USER 'postgres',
    PASSWORD ''
);
```

The information from the secret will be used when ATTACH is called. We can leave the PostgreSQL connection string empty to use all of the information stored in the secret.

```
ATTACH '' AS postgres_db (TYPE postgres);
```

We can use the PostgreSQL connection string to override individual options. For example, to connect to a different database while still using the same credentials, we can override only the database name in the following manner.

```
ATTACH 'dbname=my_other_db' AS postgres_db (TYPE postgres);
```

By default, created secrets are temporary. Secrets can be persisted using the `CREATE PERSISTENT SECRET` command. Persistent secrets can be used across sessions.

Managing Multiple Secrets

Named secrets can be used to manage connections to multiple PostgreSQL database instances. Secrets can be given a name upon creation.

```
CREATE SECRET postgres_secret_one (
    TYPE postgres,
    HOST '127.0.0.1',
    PORT 5432,
    DATABASE postgres,
    USER 'postgres',
    PASSWORD ''
);
```

The secret can then be explicitly referenced using the SECRET parameter in the ATTACH.

```
ATTACH '' AS postgres_db_one (TYPE postgres, SECRET postgres_secret_one);
```

Configuring via Environment Variables

PostgreSQL connection information can also be specified with environment variables. This can be useful in a production environment where the connection information is managed externally and passed in to the environment.

```
export PGPASSWORD="secret"
export PGHOST=localhost
export PGUSER=owner
export PGDATABASE=mydatabase
```

Then, to connect, start the duckdb process and run:

```
ATTACH '' AS p (TYPE postgres);
```

Usage

The tables in the PostgreSQL database can be read as if they were normal DuckDB tables, but the underlying data is read directly from PostgreSQL at query time.

```
SHOW ALL TABLES;
```

name
uuids

```
SELECT * FROM uuids;
```

u
6d3d2541-710b-4bde-b3af-4711738636bf
NULL
00000000-0000-0000-0000-000000000001
ffffffff-ffff-ffff-ffff-ffffffffffff

It might be desirable to create a copy of the PostgreSQL databases in DuckDB to prevent the system from re-reading the tables from PostgreSQL continuously, particularly for large tables.

Data can be copied over from PostgreSQL to DuckDB using standard SQL, for example:

```
CREATE TABLE duckdb_table AS FROM postgres_db.postgres_tbl;
```

Writing Data to PostgreSQL

In addition to reading data from PostgreSQL, the extension allows you to create tables, ingest data into PostgreSQL and make other modifications to a PostgreSQL database using standard SQL queries.

This allows you to use DuckDB to, for example, export data that is stored in a PostgreSQL database to Parquet, or read data from a Parquet file into PostgreSQL.

Below is a brief example of how to create a new table in PostgreSQL and load data into it.

```
ATTACH 'dbname=postgresscanner' AS postgres_db (TYPE postgres);
CREATE TABLE postgres_db.tbl (id INTEGER, name VARCHAR);
INSERT INTO postgres_db.tbl VALUES (42, 'DuckDB');
```

Many operations on PostgreSQL tables are supported. All these operations directly modify the PostgreSQL database, and the result of subsequent operations can then be read using PostgreSQL. Note that if modifications are not desired, ATTACH can be run with the READ_ONLY property which prevents making modifications to the underlying database. For example:

```
ATTACH 'dbname=postgresscanner' AS postgres_db (TYPE postgres, READ_ONLY);
```

Below is a list of supported operations.

CREATE TABLE

```
CREATE TABLE postgres_db.tbl (id INTEGER, name VARCHAR);
```

INSERT INTO

```
INSERT INTO postgres_db.tbl VALUES (42, 'DuckDB');
```

SELECT

```
SELECT * FROM postgres_db.tbl;
```

id	name
42	DuckDB

COPY

You can copy tables back and forth between PostgreSQL and DuckDB:

```
COPY postgres_db.tbl TO 'data.parquet';
COPY postgres_db.tbl FROM 'data.parquet';
```

These copies use PostgreSQL binary wire encoding. DuckDB can also write data using this encoding to a file which you can then load into PostgreSQL using a client of your choosing if you would like to do your own connection management:

```
COPY 'data.parquet' TO 'pg.bin' WITH (FORMAT postgres_binary);
```

The file produced will be the equivalent of copying the file to PostgreSQL using DuckDB and then dumping it from PostgreSQL using `psql` or another client:

DuckDB:

```
COPY postgres_db.tbl FROM 'data.parquet';
```

PostgreSQL:

```
\copy tbl TO 'data.bin' WITH (FORMAT BINARY);
```

You may also create a full copy of the database using the `COPY FROM DATABASE` statement:

```
COPY FROM DATABASE postgres_db TO my_duckdb_db;
```

UPDATE

```
UPDATE postgres_db.tbl
SET name = 'Woohoo'
WHERE id = 42;
```

DELETE

```
DELETE FROM postgres_db.tbl
WHERE id = 42;
```

ALTER TABLE

```
ALTER TABLE postgres_db.tbl
ADD COLUMN k INTEGER;
```

DROP TABLE

```
DROP TABLE postgres_db.tbl;
```

CREATE VIEW

```
CREATE VIEW postgres_db.v1 AS SELECT 42;
```

CREATE SCHEMA / DROP SCHEMA

```
CREATE SCHEMA postgres_db.s1;
CREATE TABLE postgres_db.s1.integers (i INTEGER);
INSERT INTO postgres_db.s1.integers VALUES (42);
SELECT * FROM postgres_db.s1.integers;
```

i
42

```
DROP SCHEMA postgres_db.s1;
```

DETACH

```
DETACH postgres_db;
```

Transactions

```
CREATE TABLE postgres_db.tmp (i INTEGER);
BEGIN;
INSERT INTO postgres_db.tmp VALUES (42);
SELECT * FROM postgres_db.tmp;
```

This returns:

i
42

```
ROLLBACK;
SELECT * FROM postgres_db.tmp;
```

This returns an empty table.

Running SQL Queries in PostgreSQL

The `postgres_query` Table Function

The `postgres_query` table function allows you to run arbitrary read queries within an attached database. `postgres_query` takes the name of the attached PostgreSQL database to execute the query in, as well as the SQL query to execute. The result of the query is returned. Single-quote strings are escaped by repeating the single quote twice.

```
postgres_query(attached_database::VARCHAR, query::VARCHAR)
```

For example:

```
ATTACH 'dbname=postgresscanner' AS postgres_db (TYPE postgres);
SELECT * FROM postgres_query('postgres_db', 'SELECT * FROM cars LIMIT 3');
```

brand	model	color
Ferrari	Testarossa	red
Aston Martin	DB2	blue
Bentley	Mulsanne	gray

The `postgres_execute` Function

The `postgres_execute` function allows running arbitrary queries within PostgreSQL, including statements that update the schema and content of the database.

```
ATTACH 'dbname=postgresscanner' AS postgres_db (TYPE postgres);
CALL postgres_execute('postgres_db', 'CREATE TABLE my_table (i INTEGER)');
```

Settings

The extension exposes the following configuration parameters.

Name	Description	Default
`pg_array_as_varchar`	Read PostgreSQL arrays as varchar - enables reading mixed dimensional arrays	`false`
`pg_connection_cache`	Whether or not to use the connection cache	`true`
`pg_connection_limit`	The maximum amount of concurrent PostgreSQL connections	`64`
`pg_debug_show_queries`	DEBUG SETTING: print all queries sent to PostgreSQL to stdout	`false`
`pg_experimental_filter_pushdown`	Whether or not to use filter pushdown (currently experimental)	`true`
`pg_pages_per_task`	The amount of pages per task	`1000`
`pg_use_binary_copy`	Whether or not to use BINARY copy to read data	`true`
`pg_null_byte_replacement`	When writing NULL bytes to Postgres, replace them with the given character	`NULL`
`pg_use_ctid_scan`	Whether or not to parallelize scanning using table ctids	`true`

Schema Cache

To avoid having to continuously fetch schema data from PostgreSQL, DuckDB keeps schema information – such as the names of tables, their columns, etc. – cached. If changes are made to the schema through a different connection to the PostgreSQL instance, such as new columns being added to a table, the cached schema information might be outdated. In this case, the function `pg_clear_cache` can be executed to clear the internal caches.

```
CALL pg_clear_cache();
```

> **Deprecated.** The old `postgres_attach` function is deprecated. It is recommended to switch over to the new ATTACH syntax.

Spatial

Spatial Extension

The `spatial` extension provides support for geospatial data processing in DuckDB. For an overview of the extension, see our blog post.

Installing and Loading

To install the `spatial` extension, run:

```
INSTALL spatial;
```

Note that the `spatial` extension is not autoloadable. Therefore, you need to load it before using it:

```
LOAD spatial;
```

The GEOMETRY Type

The core of the spatial extension is the `GEOMETRY` type. If you're unfamiliar with geospatial data and GIS tooling, this type probably works very different from what you'd expect.

On the surface, the `GEOMETRY` type is a binary representation of "geometry" data made up out of sets of vertices (pairs of X and Y `double` precision floats). But what makes it somewhat special is that its actually used to store one of several different geometry subtypes. These are `POINT`, `LINESTRING`, `POLYGON`, as well as their "collection" equivalents, `MULTIPOINT`, `MULTILINESTRING` and `MULTIPOLYGON`. Lastly there is `GEOMETRYCOLLECTION`, which can contain any of the other subtypes, as well as other `GEOMETRYCOLLECTION`s recursively.

This may seem strange at first, since DuckDB already have types like `LIST`, `STRUCT` and `UNION` which could be used in a similar way, but the design and behavior of the `GEOMETRY` type is actually based on the Simple Features geometry model, which is a standard used by many other databases and GIS software.

The spatial extension also includes a couple of experimental non-standard explicit geometry types, such as `POINT_2D`, `LINESTRING_2D`, `POLYGON_2D` and `BOX_2D` that are based on DuckDBs native nested types, such as `STRUCT` and `LIST`. Since these have a fixed and predictable internal memory layout, it is theoretically possible to optimize a lot of geospatial algorithms to be much faster when operating on these types than on the `GEOMETRY` type. However, only a couple of functions in the spatial extension have been explicitly specialized for these types so far. All of these new types are implicitly castable to `GEOMETRY`, but with a small conversion cost, so the `GEOMETRY` type is still the recommended type to use for now if you are planning to work with a lot of different spatial functions.

`GEOMETRY` is not currently capable of storing additional geometry types such as curved geometries or triangle networks. Additionally, the `GEOMETRY` type does not store SRID information on a per value basis. These limitations may be addressed in the future.

Spatial Functions

Function Index

Scalar Functions

Function	Summary
DuckDB_PROJ_Compiled_Version	Returns a text description of the PROJ library version that that this instance of DuckDB was compiled against.
DuckDB_Proj_Version	Returns a text description of the PROJ library version that is being used by this instance of DuckDB.
ST_Affine	Applies an affine transformation to a geometry.
ST_Area	Compute the area of a geometry.
ST_Area_Spheroid	Returns the area of a geometry in meters, using an ellipsoidal model of the earth
ST_AsGeoJSON	Returns the geometry as a GeoJSON fragment
ST_AsHEXWKB	Returns the geometry as a HEXWKB string
ST_AsSVG	Convert the geometry into a SVG fragment or path
ST_AsText	Returns the geometry as a WKT string
ST_AsWKB	Returns the geometry as a WKB (Well-Known-Binary) blob
ST_Azimuth	Returns the azimuth (a clockwise angle measured from north) of two points in radian.
ST_Boundary	Returns the ”boundary” of a geometry
ST_Buffer	Returns a buffer around the input geometry at the target distance
ST_BuildArea	Creates a polygonal geometry by attemtping to ”fill in” the input geometry.
ST_Centroid	Returns the centroid of a geometry
ST_Collect	Collects a list of geometries into a collection geometry.
ST_CollectionExtract	Extracts geometries from a GeometryCollection into a typed multi geometry.
ST_ConcaveHull	Returns the 'concave' hull of the input geometry, containing all of the source input's points, and which can be used to create polygons from points. The ratio parameter dictates the level of concavity; 1.0 returns the convex hull; and 0 indicates to return the most concave hull possible. Set allowHoles to a non-zero value to allow output containing holes.
ST_Contains	Returns true if the first geometry contains the second geometry
ST_ContainsProperly	Returns true if the first geometry ”properly” contains the second geometry
ST_ConvexHull	Returns the convex hull enclosing the geometry
ST_CoverageInvalidEdges	Returns the invalid edges in a polygonal coverage, which are edges that are not shared by two polygons.
ST_CoverageSimplify	Simplify the edges in a polygonal coverage, preserving the coverange by ensuring that the there are no seams between the resulting simplified polygons.
ST_CoverageUnion	Union all geometries in a polygonal coverage into a single geometry.
ST_CoveredBy	Returns true if geom1 is ”covered by” geom2
ST_Covers	Returns true if the geom1 ”covers” geom2
ST_Crosses	Returns true if geom1 ”crosses” geom2

Function	Summary
ST_DWithin	Returns if two geometries are within a target distance of each-other
ST_DWithin_GEOS	Returns if two geometries are within a target distance of each-other
ST_DWithin_Spheroid	Returns if two POINT_2D's are within a target distance in meters, using an ellipsoidal model of the earths surface
ST_Difference	Returns the ”difference” between two geometries
ST_Dimension	Returns the ”topological dimension” of a geometry.
ST_Disjoint	Returns true if the geometries are disjoint
ST_Distance	Returns the planar distance between two geometries
ST_Distance_GEOS	Returns the planar distance between two geometries
ST_Distance_Sphere	Returns the haversine (great circle) distance between two geometries.
ST_Distance_Spheroid	Returns the distance between two geometries in meters using an ellipsoidal model of the earths surface
ST_Dump	Dumps a geometry into a list of sub-geometries and their ”path” in the original geometry.
ST_EndPoint	Returns the end point of a LINESTRING.
ST_Envelope	Returns the minimum bounding rectangle of a geometry as a polygon geometry
ST_Equals	Returns true if the geometries are ”equal”
ST_Extent	Returns the minimal bounding box enclosing the input geometry
ST_Extent_Approx	Returns the approximate bounding box of a geometry, if available.
ST_ExteriorRing	Returns the exterior ring (shell) of a polygon geometry.
ST_FlipCoordinates	Returns a new geometry with the coordinates of the input geometry ”flipped” so that x = y and y = x
ST_Force2D	Forces the vertices of a geometry to have X and Y components
ST_Force3DM	Forces the vertices of a geometry to have X, Y and M components
ST_Force3DZ	Forces the vertices of a geometry to have X, Y and Z components
ST_Force4D	Forces the vertices of a geometry to have X, Y, Z and M components
ST_GeomFromGeoJSON	Deserializes a GEOMETRY from a GeoJSON fragment.
ST_GeomFromHEXEWKB	Deserialize a GEOMETRY from a HEX(E)WKB encoded string
ST_GeomFromHEXWKB	Deserialize a GEOMETRY from a HEX(E)WKB encoded string
ST_GeomFromText	Deserialize a GEOMETRY from a WKT encoded string
ST_GeomFromWKB	Deserializes a GEOMETRY from a WKB encoded blob
ST_GeometryType	Returns a 'GEOMETRY_TYPE' enum identifying the input geometry type. Possible enum return types are: POINT, LINESTRING, POLYGON, MULTIPOINT, MULTILINESTRING, MULTIPOLYGON, and GEOMETRYCOLLECTION.
ST_HasM	Check if the input geometry has M values.
ST_HasZ	Check if the input geometry has Z values.
ST_Hilbert	Encodes the X and Y values as the hilbert curve index for a curve covering the given bounding box.
ST_Intersection	Returns the intersection of two geometries

Function	Summary
ST_Intersects	Returns true if the geometries intersect
ST_Intersects_Extent	Returns true if the extent of two geometries intersects
ST_IsClosed	Check if a geometry is 'closed'
ST_IsEmpty	Returns true if the geometry is ”empty”.
ST_IsRing	Returns true if the geometry is a ring (both ST_IsClosed and ST_IsSimple).
ST_IsSimple	Returns true if the geometry is simple
ST_IsValid	Returns true if the geometry is valid
ST_Length	Returns the length of the input line geometry
ST_Length_Spheroid	Returns the length of the input geometry in meters, using an ellipsoidal model of the earth
ST_LineInterpolatePoint	Returns a point interpolated along a line at a fraction of total 2D length.
ST_LineInterpolatePoints	Returns a multi-point interpolated along a line at a fraction of total 2D length.
ST_LineMerge	”Merges” the input line geometry, optionally taking direction into account.
ST_LineString2DFromWKB	Deserialize a LINESTRING_2D from a WKB encoded blob
ST_LineSubstring	Returns a substring of a line between two fractions of total 2D length.
ST_M	Returns the M coordinate of a point geometry
ST_MMax	Returns the maximum M coordinate of a geometry
ST_MMin	Returns the minimum M coordinate of a geometry
ST_MakeEnvelope	Create a rectangular polygon from min/max coordinates
ST_MakeLine	Create a LINESTRING from a list of POINT geometries
ST_MakePolygon	Create a POLYGON from a LINESTRING shell
ST_MakeValid	Returns a valid representation of the geometry
ST_MaximumInscribedCircle	Returns the maximum inscribed circle of the input geometry, optionally with a tolerance.
ST_MinimumRotatedRectangle	Returns the minimum rotated rectangle that bounds the input geometry, finding the surrounding box that has the lowest area by using a rotated rectangle, rather than taking the lowest and highest coordinate values as per ST_Envelope().
ST_Multi	Turns a single geometry into a multi geometry.
ST_NGeometries	Returns the number of component geometries in a collection geometry.
ST_NInteriorRings	Returns the number of interior rings of a polygon
ST_NPoints	Returns the number of vertices within a geometry
ST_Node	Returns a ”noded” MultiLinestring, produced by combining a collection of input linestrings and adding additional vertices where they intersect.
ST_Normalize	Returns the ”normalized” representation of the geometry

Function	Summary
ST_NumGeometries	Returns the number of component geometries in a collection geometry.
ST_NumInteriorRings	Returns the number of interior rings of a polygon
ST_NumPoints	Returns the number of vertices within a geometry
ST_Overlaps	Returns true if the geometries overlap
ST_Perimeter	Returns the length of the perimeter of the geometry
ST_Perimeter_Spheroid	Returns the length of the perimeter in meters using an ellipsoidal model of the earths surface
ST_Point	Creates a GEOMETRY point
ST_Point2D	Creates a POINT_2D
ST_Point2DFromWKB	Deserialize a POINT_2D from a WKB encoded blob
ST_Point3D	Creates a POINT_3D
ST_Point4D	Creates a POINT_4D
ST_PointN	Returns the n'th vertex from the input geometry as a point geometry
ST_PointOnSurface	Returns a point guaranteed to lie on the surface of the geometry
ST_Points	Collects all the vertices in the geometry into a MULTIPOINT
ST_Polygon2DFromWKB	Deserialize a POLYGON_2D from a WKB encoded blob
ST_Polygonize	Returns a polygonized representation of the input geometries
ST_QuadKey	Compute the quadkey for a given lon/lat point at a given level.
ST_ReducePrecision	Returns the geometry with all vertices reduced to the given precision
ST_RemoveRepeatedPoints	Remove repeated points from a LINESTRING.
ST_Reverse	Returns the geometry with the order of its vertices reversed
ST_ShortestLine	Returns the shortest line between two geometries
ST_Simplify	Returns a simplified version of the geometry
ST_SimplifyPreserveTopology	Returns a simplified version of the geometry that preserves topology
ST_StartPoint	Returns the start point of a LINESTRING.
ST_TileEnvelope	The ST_TileEnvelope scalar function generates tile envelope rectangular polygons from specified zoom level and tile indices.
ST_Touches	Returns true if the geometries touch
ST_Transform	Transforms a geometry between two coordinate systems
ST_Union	Returns the union of two geometries
ST_VoronoiDiagram	Returns the Voronoi diagram of the supplied MultiPoint geometry
ST_Within	Returns true if the first geometry is within the second
ST_WithinProperly	Returns true if the first geometry ”properly” is contained by the second geometry
ST_X	Returns the X coordinate of a point geometry
ST_XMax	Returns the maximum X coordinate of a geometry
ST_XMin	Returns the minimum X coordinate of a geometry

Function	Summary
ST_Y	Returns the Y coordinate of a point geometry
ST_YMax	Returns the maximum Y coordinate of a geometry
ST_YMin	Returns the minimum Y coordinate of a geometry
ST_Z	Returns the Z coordinate of a point geometry
ST_ZMFlag	Returns a flag indicating the presence of Z and M values in the input geometry.
ST_ZMax	Returns the maximum Z coordinate of a geometry
ST_ZMin	Returns the minimum Z coordinate of a geometry

Aggregate Functions

Function	Summary
ST_CoverageInvalidEdges_Agg	Returns the invalid edges of a coverage geometry
ST_CoverageSimplify_Agg	Simplifies a set of geometries while maintaining coverage
ST_CoverageUnion_Agg	Unions a set of geometries while maintaining coverage
ST_Envelope_Agg	Alias for ST_Extent_Agg.
ST_Extent_Agg	Computes the minimal-bounding-box polygon containing the set of input geometries
ST_Intersection_Agg	Computes the intersection of a set of geometries
ST_MemUnion_Agg	Computes the union of a set of input geometries.
ST_Union_Agg	Computes the union of a set of input geometries

Macro Functions

Function	Summary
ST_Rotate	Alias of ST_RotateZ
ST_RotateX	Rotates a geometry around the X axis. This is a shorthand macro for calling ST_Affine.
ST_RotateY	Rotates a geometry around the Y axis. This is a shorthand macro for calling ST_Affine.
ST_RotateZ	Rotates a geometry around the Z axis. This is a shorthand macro for calling ST_Affine.
ST_Scale	
ST_TransScale	Translates and then scales a geometry in X and Y direction. This is a shorthand macro for calling ST_Affine.
ST_Translate	

Table Functions

Function	Summary
ST_Drivers	Returns the list of supported GDAL drivers and file formats
ST_GeneratePoints	Generates a set of random points within the specified bounding box.
ST_Read	Read and import a variety of geospatial file formats using the GDAL library.
ST_ReadOSM	The ST_ReadOsm() table function enables reading compressed OpenStreetMap data directly from a .osm.pbf file.
ST_ReadSHP	Read a Shapefile without relying on the GDAL library
ST_Read_Meta	Read the metadata from a variety of geospatial file formats using the GDAL library.

Scalar Functions

DuckDB_PROJ_Compiled_Version

Signature

```
VARCHAR DuckDB_PROJ_Compiled_Version ()
```

Description

Returns a text description of the PROJ library version that that this instance of DuckDB was compiled against.

Example

```
SELECT duckdb_proj_compiled_version();
```

duckdb_proj_compiled_version()
varchar
Rel. 9.1.1, December 1st, 2022

DuckDB_Proj_Version

Signature

```
VARCHAR DuckDB_Proj_Version ()
```

Description

Returns a text description of the PROJ library version that is being used by this instance of DuckDB.

Example

```
SELECT duckdb_proj_version();
```

duckdb_proj_version()
varchar
9.1.1

ST_Affine

Signatures

```
GEOMETRY ST_Affine (geom GEOMETRY, a DOUBLE, b DOUBLE, c DOUBLE, d DOUBLE, e DOUBLE, f DOUBLE, g DOUBLE,
h DOUBLE, i DOUBLE, xoff DOUBLE, yoff DOUBLE, zoff DOUBLE)
GEOMETRY ST_Affine (geom GEOMETRY, a DOUBLE, b DOUBLE, d DOUBLE, e DOUBLE, xoff DOUBLE, yoff DOUBLE)
```

Description

Applies an affine transformation to a geometry.

For the 2D variant, the transformation matrix is defined as follows:

```
| a b xoff |
| d e yoff |
| 0 0 1    |
```

For the 3D variant, the transformation matrix is defined as follows:

```
| a b c xoff |
| d e f yoff |
| g h i zoff |
| 0 0 0 1    |
```

The transformation is applied to all vertices of the geometry.

Example

```
-- Translate a point by (2, 3)
SELECT ST_Affine(ST_Point(1, 1),
                 1, 0,   -- a, b
                 0, 1,   -- d, e
                 2, 3);  -- xoff, yoff
----
POINT (3 4)

-- Scale a geometry by factor 2 in X and Y
SELECT ST_Affine(ST_Point(1, 1),
                 2, 0, 0,   -- a, b, c
                 0, 2, 0,   -- d, e, f
                 0, 0, 1,   -- g, h, i
                 0, 0, 0);  -- xoff, yoff, zoff
----
POINT (2 2)
```

ST_Area

Signatures

```
DOUBLE ST_Area (geom GEOMETRY)
DOUBLE ST_Area (polygon POLYGON_2D)
DOUBLE ST_Area (linestring LINESTRING_2D)
DOUBLE ST_Area (point POINT_2D)
DOUBLE ST_Area (box BOX_2D)
```

Description

Compute the area of a geometry.

Returns `0.0` for any geometry that is not a `POLYGON`, `MULTIPOLYGON` or `GEOMETRYCOLLECTION` containing polygon geometries.

The area is in the same units as the spatial reference system of the geometry.

The `POINT_2D` and `LINESTRING_2D` overloads of this function always return `0.0` but are included for completeness.

Example

```
select ST_Area('POLYGON((0 0, 0 1, 1 1, 1 0, 0 0))'::geometry);
-- 1.0
```

ST_Area_Spheroid

Signatures

```
DOUBLE ST_Area_Spheroid (geom GEOMETRY)
DOUBLE ST_Area_Spheroid (poly POLYGON_2D)
```

Description

Returns the area of a geometry in meters, using an ellipsoidal model of the earth

The input geometry is assumed to be in the EPSG:4326 coordinate system (WGS84), with [latitude, longitude] axis order and the area is returned in square meters. This function uses the GeographicLib library, calculating the area using an ellipsoidal model of the earth. This is a highly accurate method for calculating the area of a polygon taking the curvature of the earth into account, but is also the slowest.

Returns `0.0` for any geometry that is not a `POLYGON`, `MULTIPOLYGON` or `GEOMETRYCOLLECTION` containing polygon geometries.

ST_AsGeoJSON

Signature

```
JSON ST_AsGeoJSON (geom GEOMETRY)
```

Description

Returns the geometry as a GeoJSON fragment

This does not return a complete GeoJSON document, only the geometry fragment. To construct a complete GeoJSON document or feature, look into using the DuckDB JSON extension in conjunction with this function. This function supports geometries with Z values, but not M values. M values are ignored.

Example

```
select ST_AsGeoJSON('POLYGON((0 0, 0 1, 1 1, 1 0, 0 0))'::geometry);
----
{"type":"Polygon","coordinates":[[[0.0,0.0],[0.0,1.0],[1.0,1.0],[1.0,0.0],[0.0,0.0]]]}

-- Convert a geometry into a full GeoJSON feature (requires the JSON extension to be loaded)
SELECT CAST({
    type: 'Feature',
    geometry: ST_AsGeoJSON(ST_Point(1,2)),
    properties: {
        name: 'my_point'
    }
} AS JSON);
----
{"type":"Feature","geometry":{"type":"Point","coordinates":[1.0,2.0]},"properties":{"name":"my_point"}}
```

ST_AsHEXWKB

Signature

```
VARCHAR ST_AsHEXWKB (geom GEOMETRY)
```

Description

Returns the geometry as a HEXWKB string

Example

```
SELECT ST_AsHexWKB('POLYGON((0 0, 0 1, 1 1, 1 0, 0 0))'::geometry);
----
01030000000100000005000000000000000000000000000000...
```

ST_AsSVG

Signature

```
VARCHAR ST_AsSVG (geom GEOMETRY, relative BOOLEAN, precision INTEGER)
```

Description

Convert the geometry into a SVG fragment or path

The SVG fragment is returned as a string. The fragment is a path element that can be used in an SVG document. The second boolean argument specifies whether the path should be relative or absolute. The third argument specifies the maximum number of digits to use for the coordinates.

Points are formatted as cx/cy using absolute coordinates or x/y using relative coordinates.

Example

```
SELECT ST_AsSVG('POLYGON((0 0, 0 1, 1 1, 1 0, 0 0))'::GEOMETRY, false, 15);
----
M 0 0 L 0 -1 1 -1 1 0 Z
```

ST_AsText

Signatures

```
VARCHAR ST_AsText (geom GEOMETRY)
VARCHAR ST_AsText (point POINT_2D)
VARCHAR ST_AsText (linestring LINESTRING_2D)
VARCHAR ST_AsText (polygon POLYGON_2D)
VARCHAR ST_AsText (box BOX_2D)
```

Description

Returns the geometry as a WKT string

Example

```
SELECT ST_MakeEnvelope(0,0,1,1);
----
POLYGON ((0 0, 0 1, 1 1, 1 0, 0 0))
```

ST_AsWKB

Signature

```
WKB_BLOB ST_AsWKB (geom GEOMETRY)
```

Description

Returns the geometry as a WKB (Well-Known-Binary) blob

Example

```
SELECT ST_AsWKB('POLYGON((0 0, 0 1, 1 1, 1 0, 0 0))'::GEOMETRY)::BLOB;
----
\x01\x03\x00\x00\x00\x01\x00\x00\x00\x05...
```

ST_Azimuth

Signatures

```
DOUBLE ST_Azimuth (origin GEOMETRY, target GEOMETRY)
DOUBLE ST_Azimuth (origin POINT_2D, target POINT_2D)
```

Description

Returns the azimuth (a clockwise angle measured from north) of two points in radian.

Example

```
SELECT degrees(ST_Azimuth(ST_Point(0, 0), ST_Point(0, 1)));
----
90.0
```

ST_Boundary

Signature

```
GEOMETRY ST_Boundary (geom GEOMETRY)
```

Description

Returns the ”boundary” of a geometry

ST_Buffer

Signatures

```
GEOMETRY ST_Buffer (geom GEOMETRY, distance DOUBLE)
GEOMETRY ST_Buffer (geom GEOMETRY, distance DOUBLE, num_triangles INTEGER)
GEOMETRY ST_Buffer (geom GEOMETRY, distance DOUBLE, num_triangles INTEGER, cap_style VARCHAR, join_style
VARCHAR, mitre_limit DOUBLE)
```

Description

Returns a buffer around the input geometry at the target distance

`geom` is the input geometry.

`distance` is the target distance for the buffer, using the same units as the input geometry.

`num_triangles` represents how many triangles that will be produced to approximate a quarter circle. The larger the number, the smoother the resulting geometry. The default value is 8.

`cap_style` must be one of "CAP_ROUND", "CAP_FLAT", "CAP_SQUARE". This parameter is case-insensitive.

`join_style` must be one of "JOIN_ROUND", "JOIN_MITRE", "JOIN_BEVEL". This parameter is case-insensitive.

`mitre_limit` only applies when `join_style` is "JOIN_MITRE". It is the ratio of the distance from the corner to the mitre point to the corner radius. The default value is 1.0.

This is a planar operation and will not take into account the curvature of the earth.

ST_BuildArea

Signature

```
GEOMETRY ST_BuildArea (geom GEOMETRY)
```

Description

Creates a polygonal geometry by attemtping to "fill in" the input geometry.

Unlike ST_Polygonize, this function does not fill in holes.

ST_Centroid

Signatures

```
GEOMETRY ST_Centroid (geom GEOMETRY)
POINT_2D ST_Centroid (point POINT_2D)
POINT_2D ST_Centroid (linestring LINESTRING_2D)
POINT_2D ST_Centroid (polygon POLYGON_2D)
POINT_2D ST_Centroid (box BOX_2D)
POINT_2D ST_Centroid (box BOX_2DF)
```

Description

Returns the centroid of a geometry

ST_Collect

Signature

```
GEOMETRY ST_Collect (geoms GEOMETRY[])
```

Description

Collects a list of geometries into a collection geometry.

- If all geometries are `POINT`'s, a `MULTIPOINT` is returned.
- If all geometries are `LINESTRING`'s, a `MULTILINESTRING` is returned.
- If all geometries are `POLYGON`'s, a `MULTIPOLYGON` is returned.
- Otherwise if the input collection contains a mix of geometry types, a `GEOMETRYCOLLECTION` is returned.

Empty and `NULL` geometries are ignored. If all geometries are empty or `NULL`, a `GEOMETRYCOLLECTION  EMPTY` is returned.

Example

```
-- With all POINT's, a MULTIPOINT is returned
SELECT ST_Collect([ST_Point(1, 2), ST_Point(3, 4)]);
----
MULTIPOINT (1 2, 3 4)

-- With mixed geometry types, a GEOMETRYCOLLECTION is returned
SELECT ST_Collect([ST_Point(1, 2), ST_GeomFromText('LINESTRING(3 4, 5 6)')]);
----
GEOMETRYCOLLECTION (POINT (1 2), LINESTRING (3 4, 5 6))

-- Note that the empty geometry is ignored, so the result is a MULTIPOINT
SELECT ST_Collect([ST_Point(1, 2), NULL, ST_GeomFromText('GEOMETRYCOLLECTION EMPTY')]);
----
MULTIPOINT (1 2)

-- If all geometries are empty or NULL, a GEOMETRYCOLLECTION EMPTY is returned
SELECT ST_Collect([NULL, ST_GeomFromText('GEOMETRYCOLLECTION EMPTY')]);
----
GEOMETRYCOLLECTION EMPTY

-- Tip: You can use the `ST_Collect` function together with the `list()` aggregate function to collect
multiple rows of geometries into a single geometry collection:

CREATE TABLE points (geom GEOMETRY);

INSERT INTO points VALUES (ST_Point(1, 2)), (ST_Point(3, 4));

SELECT ST_Collect(list(geom)) FROM points;
----
MULTIPOINT (1 2, 3 4)
```

ST_CollectionExtract

Signatures

```
GEOMETRY ST_CollectionExtract (geom GEOMETRY, type INTEGER)
```

```
GEOMETRY ST_CollectionExtract (geom GEOMETRY)
```

Description

Extracts geometries from a GeometryCollection into a typed multi geometry.

If the input geometry is a GeometryCollection, the function will return a multi geometry, determined by the `type` parameter.

- if `type` = 1, returns a MultiPoint containg all the Points in the collection
- if `type` = 2, returns a MultiLineString containg all the LineStrings in the collection
- if `type` = 3, returns a MultiPolygon containg all the Polygons in the collection

If no `type` parameters is provided, the function will return a multi geometry matching the highest "surface dimension" of the contained geometries. E.g. if the collection contains only Points, a MultiPoint will be returned. But if the collection contains both Points and LineStrings, a MultiLineString will be returned. Similarly, if the collection contains Polygons, a MultiPolygon will be returned. Contained geometries of a lower surface dimension will be ignored.

If the input geometry contains nested GeometryCollections, their geometries will be extracted recursively and included into the final multi geometry as well.

If the input geometry is not a GeometryCollection, the function will return the input geometry as is.

Example

```
select st_collectionextract('MULTIPOINT(1 2,3 4)'::geometry, 1);
-- MULTIPOINT (1 2, 3 4)
```

ST_ConcaveHull

Signature

```
GEOMETRY ST_ConcaveHull (geom GEOMETRY, ratio DOUBLE, allowHoles BOOLEAN)
```

Description

Returns the 'concave' hull of the input geometry, containing all of the source input's points, and which can be used to create polygons from points. The ratio parameter dictates the level of concavity; 1.0 returns the convex hull; and 0 indicates to return the most concave hull possible. Set allowHoles to a non-zero value to allow output containing holes.

ST_Contains

Signatures

```
BOOLEAN ST_Contains (geom1 POLYGON_2D, geom2 POINT_2D)
BOOLEAN ST_Contains (geom1 GEOMETRY, geom2 GEOMETRY)
```

Description

Returns true if the first geometry contains the second geometry

In contrast to `ST_ContainsProperly`, this function will also return true if `geom2` is contained strictly on the boundary of `geom1`. A geometry always `ST_Contains` itself, but does not `ST_ContainsProperly` itself.

ST_ContainsProperly

Signature

```
BOOLEAN ST_ContainsProperly (geom1 GEOMETRY, geom2 GEOMETRY)
```

Description

Returns true if the first geometry "properly" contains the second geometry

In contrast to `ST_Contains`, this function does not return true if `geom2` is contained strictly on the boundary of `geom1`. A geometry always `ST_Contains` itself, but does not `ST_ContainsProperly` itself.

ST_ConvexHull

Signature

```
GEOMETRY ST_ConvexHull (geom GEOMETRY)
```

Description

Returns the convex hull enclosing the geometry

ST_CoverageInvalidEdges

Signatures

```
GEOMETRY ST_CoverageInvalidEdges (geoms GEOMETRY[], tolerance DOUBLE)
GEOMETRY ST_CoverageInvalidEdges (geoms GEOMETRY[])
```

Description

Returns the invalid edges in a polygonal coverage, which are edges that are not shared by two polygons. Returns NULL if the input is not a polygonal coverage, or if the input is valid. Tolerance is 0 by default.

ST_CoverageSimplify

Signatures

```
GEOMETRY ST_CoverageSimplify (geoms GEOMETRY[], tolerance DOUBLE, simplify_boundary BOOLEAN)
GEOMETRY ST_CoverageSimplify (geoms GEOMETRY[], tolerance DOUBLE)
```

Description

Simplify the edges in a polygonal coverage, preserving the coverange by ensuring that the there are no seams between the resulting simplified polygons.

By default, the boundary of the coverage is also simplified, but this can be controlled with the optional third 'simplify_boundary' parameter.

ST_CoverageUnion

Signature

```
GEOMETRY ST_CoverageUnion (geoms GEOMETRY[])
```

Description

Union all geometries in a polygonal coverage into a single geometry. This may be faster than using `ST_Union`, but may use more memory.

ST_CoveredBy

Signature

```
BOOLEAN ST_CoveredBy (geom1 GEOMETRY, geom2 GEOMETRY)
```

Description

Returns true if geom1 is ”covered by” geom2

ST_Covers

Signature

```
BOOLEAN ST_Covers (geom1 GEOMETRY, geom2 GEOMETRY)
```

Description

Returns true if the geom1 ”covers” geom2

ST_Crosses

Signature

```
BOOLEAN ST_Crosses (geom1 GEOMETRY, geom2 GEOMETRY)
```

Description

Returns true if geom1 ”crosses” geom2

ST_DWithin

Signature

```
BOOLEAN ST_DWithin (geom1 GEOMETRY, geom2 GEOMETRY, distance DOUBLE)
```

Description

Returns if two geometries are within a target distance of each-other

---

ST_DWithin_GEOS

Signature

```
BOOLEAN ST_DWithin_GEOS (geom1 GEOMETRY, geom2 GEOMETRY, distance DOUBLE)
```

Description

Returns if two geometries are within a target distance of each-other

---

ST_DWithin_Spheroid

Signature

```
BOOLEAN ST_DWithin_Spheroid (p1 POINT_2D, p2 POINT_2D, distance DOUBLE)
```

Description

Returns if two POINT_2D's are within a target distance in meters, using an ellipsoidal model of the earths surface

The input geometry is assumed to be in the EPSG:4326 coordinate system (WGS84), with [latitude, longitude] axis order and the distance is returned in meters. This function uses the GeographicLib library to solve the inverse geodesic problem, calculating the distance between two points using an ellipsoidal model of the earth. This is a highly accurate method for calculating the distance between two arbitrary points taking the curvature of the earths surface into account, but is also the slowest.

ST_Difference

Signature

```
GEOMETRY ST_Difference (geom1 GEOMETRY, geom2 GEOMETRY)
```

Description

Returns the ”difference” between two geometries

ST_Dimension

Signature

```
INTEGER ST_Dimension (geom GEOMETRY)
```

Description

Returns the ”topological dimension” of a geometry.

- For POINT and MULTIPOINT geometries, returns 0
- For LINESTRING and MULTILINESTRING, returns 1
- For POLYGON and MULTIPOLYGON, returns 2
- For GEOMETRYCOLLECTION, returns the maximum dimension of the contained geometries, or 0 if the collection is empty

Example

```
select st_dimension('POLYGON((0 0, 0 1, 1 1, 1 0, 0 0))'::geometry);
----
2
```

ST_Disjoint

Signature

```
BOOLEAN ST_Disjoint (geom1 GEOMETRY, geom2 GEOMETRY)
```

Description

Returns true if the geometries are disjoint

ST_Distance

Signatures

```
DOUBLE ST_Distance (point1 POINT_2D, point2 POINT_2D)
DOUBLE ST_Distance (point POINT_2D, linestring LINESTRING_2D)
DOUBLE ST_Distance (linestring LINESTRING_2D, point POINT_2D)
DOUBLE ST_Distance (geom1 GEOMETRY, geom2 GEOMETRY)
```

Description

Returns the planar distance between two geometries

Example

```
SELECT ST_Distance('POINT (0 0)'::GEOMETRY, 'POINT (3 4)'::GEOMETRY);
----
5.0

-- Z coordinates are ignored
SELECT ST_Distance('POINT Z (0 0 0)'::GEOMETRY, 'POINT Z (3 4 5)'::GEOMETRY);
----
5.0
```

ST_Distance_GEOS

Signature

```
DOUBLE ST_Distance_GEOS (geom1 GEOMETRY, geom2 GEOMETRY)
```

Description

Returns the planar distance between two geometries

ST_Distance_Sphere

Signatures

```
DOUBLE ST_Distance_Sphere (geom1 GEOMETRY, geom2 GEOMETRY)
DOUBLE ST_Distance_Sphere (point1 POINT_2D, point2 POINT_2D)
```

Description

Returns the haversine (great circle) distance between two geometries.

- Only supports POINT geometries.
- Returns the distance in meters.
- The input is expected to be in WGS84 (EPSG:4326) coordinates, using a [latitude, longitude] axis order.

ST_Distance_Spheroid

Signature

```
DOUBLE ST_Distance_Spheroid (p1 POINT_2D, p2 POINT_2D)
```

Description

Returns the distance between two geometries in meters using an ellipsoidal model of the earths surface

The input geometry is assumed to be in the EPSG:4326 coordinate system (WGS84), with [latitude, longitude] axis order and the distance limit is expected to be in meters. This function uses the GeographicLib library to solve the inverse geodesic problem, calculating the distance between two points using an ellipsoidal model of the earth. This is a highly accurate method for calculating the distance between two arbitrary points taking the curvature of the earths surface into account, but is also the slowest.

Example

```
-- Note: the coordinates are in WGS84 and [latitude, longitude] axis order
-- Whats the distance between New York and Amsterdam (JFK and AMS airport)?
SELECT st_distance_spheroid(
st_point(40.6446, -73.7797),
st_point(52.3130, 4.7725)
);
----
5863418.7459356235
-- Roughly 5863km!
```

ST_Dump

Signature

```
STRUCT(geom GEOMETRY, path INTEGER[])[] ST_Dump (geom GEOMETRY)
```

Description

Dumps a geometry into a list of sub-geometries and their ”path” in the original geometry.

You can use the UNNEST(`res, recursive := true`) function to explode resulting list of structs into multiple rows.

Example

```
select st_dump('MULTIPOINT(1 2,3 4)'::geometry);
----
[{'geom': 'POINT(1 2)', 'path': [0]}, {'geom': 'POINT(3 4)', 'path': [1]}]

select unnest(st_dump('MULTIPOINT(1 2,3 4)'::geometry), recursive := true);
--
--   geom      path
--  geometry   int32[]
--
--  POINT (1 2)  [1]
--  POINT (3 4)  [2]
--
```

ST_EndPoint

Signatures

```
GEOMETRY ST_EndPoint (geom GEOMETRY)
POINT_2D ST_EndPoint (line LINESTRING_2D)
```

Description

Returns the end point of a LINESTRING.

ST_Envelope

Signature

```
GEOMETRY ST_Envelope (geom GEOMETRY)
```

Description

Returns the minimum bounding rectangle of a geometry as a polygon geometry

---

ST_Equals

Signature

```
BOOLEAN ST_Equals (geom1 GEOMETRY, geom2 GEOMETRY)
```

Description

Returns true if the geometries are ”equal”

ST_Extent

Signatures

```
BOX_2D ST_Extent (geom GEOMETRY)
BOX_2D ST_Extent (wkb WKB_BLOB)
```

Description

Returns the minimal bounding box enclosing the input geometry

---

ST_Extent_Approx

Signature

```
BOX_2DF ST_Extent_Approx (geom GEOMETRY)
```

Description

Returns the approximate bounding box of a geometry, if available.

This function is only really used internally, and returns the cached bounding box of the geometry if it exists. This function may be removed or renamed in the future.

ST_ExteriorRing

Signatures

```
GEOMETRY ST_ExteriorRing (geom GEOMETRY)
LINESTRING_2D ST_ExteriorRing (polygon POLYGON_2D)
```

Description

Returns the exterior ring (shell) of a polygon geometry.

ST_FlipCoordinates

Signatures

```
GEOMETRY ST_FlipCoordinates (geom GEOMETRY)
POINT_2D ST_FlipCoordinates (point POINT_2D)
LINESTRING_2D ST_FlipCoordinates (linestring LINESTRING_2D)
POLYGON_2D ST_FlipCoordinates (polygon POLYGON_2D)
BOX_2D ST_FlipCoordinates (box BOX_2D)
```

Description

Returns a new geometry with the coordinates of the input geometry ”flipped” so that x = y and y = x

---

ST_Force2D

Signature

```
GEOMETRY ST_Force2D (geom GEOMETRY)
```

Description

Forces the vertices of a geometry to have X and Y components

This function will drop any Z and M values from the input geometry, if present. If the input geometry is already 2D, it will be returned as is.

ST_Force3DM

Signature

```
GEOMETRY ST_Force3DM (geom GEOMETRY, m DOUBLE)
```

Description

Forces the vertices of a geometry to have X, Y and M components

The following cases apply:

- If the input geometry has a Z component but no M component, the Z component will be replaced with the new M value.
- If the input geometry has a M component but no Z component, it will be returned as is.
- If the input geometry has both a Z component and a M component, the Z component will be removed.
- Otherwise, if the input geometry has neither a Z or M component, the new M value will be added to the vertices of the input geometry.

ST_Force3DZ

Signature

```
GEOMETRY ST_Force3DZ (geom GEOMETRY, z DOUBLE)
```

Description

Forces the vertices of a geometry to have X, Y and Z components

The following cases apply:

- If the input geometry has a M component but no Z component, the M component will be replaced with the new Z value.
- If the input geometry has a Z component but no M component, it will be returned as is.
- If the input geometry has both a Z component and a M component, the M component will be removed.
- Otherwise, if the input geometry has neither a Z or M component, the new Z value will be added to the vertices of the input geometry.

ST_Force4D

Signature

```
GEOMETRY ST_Force4D (geom GEOMETRY, z DOUBLE, m DOUBLE)
```

Description

Forces the vertices of a geometry to have X, Y, Z and M components

The following cases apply:

- If the input geometry has a Z component but no M component, the new M value will be added to the vertices of the input geometry.
- If the input geometry has a M component but no Z component, the new Z value will be added to the vertices of the input geometry.
- If the input geometry has both a Z component and a M component, the geometry will be returned as is.
- Otherwise, if the input geometry has neither a Z or M component, the new Z and M values will be added to the vertices of the input geometry.

ST_GeomFromGeoJSON

Signatures

```
GEOMETRY ST_GeomFromGeoJSON (geojson JSON)
GEOMETRY ST_GeomFromGeoJSON (geojson VARCHAR)
```

Description

Deserializes a GEOMETRY from a GeoJSON fragment.

Example

```
SELECT ST_GeomFromGeoJSON('{"type":"Point","coordinates":[1.0,2.0]}');
----
POINT (1 2)
```

ST_GeomFromHEXEWKB

Signature

```
GEOMETRY ST_GeomFromHEXEWKB (hexwkb VARCHAR)
```

Description

Deserialize a GEOMETRY from a HEX(E)WKB encoded string

DuckDB spatial doesnt currently differentiate between WKB and EWKB, so ST_GeomFromHEXWKB and 'ST_GeomFromHEXEWKB" are just aliases of eachother.

ST_GeomFromHEXWKB

Signature

```
GEOMETRY ST_GeomFromHEXWKB (hexwkb VARCHAR)
```

Description

Deserialize a GEOMETRY from a HEX(E)WKB encoded string

DuckDB spatial doesnt currently differentiate between WKB and EWKB, so ST_GeomFromHEXWKB and 'ST_GeomFromHEXEWKB" are just aliases of eachother.

ST_GeomFromText

Signatures

```
GEOMETRY ST_GeomFromText (wkt VARCHAR)
GEOMETRY ST_GeomFromText (wkt VARCHAR, ignore_invalid BOOLEAN)
```

Description

Deserialize a GEOMETRY from a WKT encoded string

ST_GeomFromWKB

Signatures

```
GEOMETRY ST_GeomFromWKB (wkb WKB_BLOB)
GEOMETRY ST_GeomFromWKB (blob BLOB)
```

Description

Deserializes a GEOMETRY from a WKB encoded blob

ST_GeometryType

Signatures

```
ANY ST_GeometryType (geom GEOMETRY)
ANY ST_GeometryType (point POINT_2D)
ANY ST_GeometryType (linestring LINESTRING_2D)
ANY ST_GeometryType (polygon POLYGON_2D)
ANY ST_GeometryType (wkb WKB_BLOB)
```

Description

Returns a 'GEOMETRY_TYPE' enum identifying the input geometry type. Possible enum return types are: `POINT`, `LINESTRING`, `POLYGON`, `MULTIPOINT`, `MULTILINESTRING`, `MULTIPOLYGON`, and `GEOMETRYCOLLECTION`.

Example

```
SELECT DISTINCT ST_GeometryType(ST_GeomFromText('POINT(1 1)'));
----
POINT
```

ST_HasM

Signatures

```
BOOLEAN ST_HasM (geom GEOMETRY)
BOOLEAN ST_HasM (wkb WKB_BLOB)
```

Description

Check if the input geometry has M values.

Example

```
-- HasM for a 2D geometry
SELECT ST_HasM(ST_GeomFromText('POINT(1 1)'));
----
false

-- HasM for a 3DZ geometry
SELECT ST_HasM(ST_GeomFromText('POINT Z(1 1 1)'));
----
false
```

```
-- HasM for a 3DM geometry
SELECT ST_HasM(ST_GeomFromText('POINT M(1 1 1)'));
----
true

-- HasM for a 4D geometry
SELECT ST_HasM(ST_GeomFromText('POINT ZM(1 1 1 1)'));
----
true
```

ST_HasZ

Signatures

```
BOOLEAN ST_HasZ (geom GEOMETRY)
BOOLEAN ST_HasZ (wkb WKB_BLOB)
```

Description

Check if the input geometry has Z values.

Example

```
-- HasZ for a 2D geometry
SELECT ST_HasZ(ST_GeomFromText('POINT(1 1)'));
----
false

-- HasZ for a 3DZ geometry
SELECT ST_HasZ(ST_GeomFromText('POINT Z(1 1 1)'));
----
true

-- HasZ for a 3DM geometry
SELECT ST_HasZ(ST_GeomFromText('POINT M(1 1 1)'));
----
false

-- HasZ for a 4D geometry
SELECT ST_HasZ(ST_GeomFromText('POINT ZM(1 1 1 1)'));
----
true
```

ST_Hilbert

Signatures

```
UINTEGER ST_Hilbert (x DOUBLE, y DOUBLE, bounds BOX_2D)
UINTEGER ST_Hilbert (geom GEOMETRY, bounds BOX_2D)
UINTEGER ST_Hilbert (geom GEOMETRY)
UINTEGER ST_Hilbert (box BOX_2D, bounds BOX_2D)
UINTEGER ST_Hilbert (box BOX_2DF, bounds BOX_2DF)
```

Description

Encodes the X and Y values as the hilbert curve index for a curve covering the given bounding box. If a geometry is provided, the center of the approximate bounding box is used as the point to encode. If no bounding box is provided, the hilbert curve index is mapped to the full range of a single-presicion float. For the BOX_2D and BOX_2DF variants, the center of the box is used as the point to encode.

ST_Intersection

Signature

```
GEOMETRY ST_Intersection (geom1 GEOMETRY, geom2 GEOMETRY)
```

Description

Returns the intersection of two geometries

ST_Intersects

Signatures

```
BOOLEAN ST_Intersects (box1 BOX_2D, box2 BOX_2D)
BOOLEAN ST_Intersects (geom1 GEOMETRY, geom2 GEOMETRY)
```

Description

Returns true if the geometries intersect

ST_Intersects_Extent

Signature

```
BOOLEAN ST_Intersects_Extent (geom1 GEOMETRY, geom2 GEOMETRY)
```

Description

Returns true if the extent of two geometries intersects

ST_IsClosed

Signature

```
BOOLEAN ST_IsClosed (geom GEOMETRY)
```

Description

Check if a geometry is 'closed'

ST_IsEmpty

Signatures

```
BOOLEAN ST_IsEmpty (geom GEOMETRY)
BOOLEAN ST_IsEmpty (linestring LINESTRING_2D)
BOOLEAN ST_IsEmpty (polygon POLYGON_2D)
```

Description

Returns true if the geometry is ”empty”.

ST_IsRing

Signature

```
BOOLEAN ST_IsRing (geom GEOMETRY)
```

Description

Returns true if the geometry is a ring (both ST_IsClosed and ST_IsSimple).

ST_IsSimple

Signature

```
BOOLEAN ST_IsSimple (geom GEOMETRY)
```

Description

Returns true if the geometry is simple

ST_IsValid

Signature

```
BOOLEAN ST_IsValid (geom GEOMETRY)
```

Description

Returns true if the geometry is valid

ST_Length

Signatures

```
DOUBLE ST_Length (geom GEOMETRY)
DOUBLE ST_Length (linestring LINESTRING_2D)
```

Description

Returns the length of the input line geometry

ST_Length_Spheroid

Signatures

```
DOUBLE ST_Length_Spheroid (geom GEOMETRY)
DOUBLE ST_Length_Spheroid (line LINESTRING_2D)
```

Description

Returns the length of the input geometry in meters, using an ellipsoidal model of the earth

The input geometry is assumed to be in the EPSG:4326 coordinate system (WGS84), with [latitude, longitude] axis order and the length is returned in meters. This function uses the GeographicLib library, calculating the length using an ellipsoidal model of the earth. This is a highly accurate method for calculating the length of a line geometry taking the curvature of the earth into account, but is also the slowest.

Returns `0.0` for any geometry that is not a `LINESTRING`, `MULTILINESTRING` or `GEOMETRYCOLLECTION` containing line geometries.

ST_LineInterpolatePoint

Signature

```
GEOMETRY ST_LineInterpolatePoint (line GEOMETRY, fraction DOUBLE)
```

Description

Returns a point interpolated along a line at a fraction of total 2D length.

ST_LineInterpolatePoints

Signature

```
GEOMETRY ST_LineInterpolatePoints (line GEOMETRY, fraction DOUBLE, repeat BOOLEAN)
```

Description

Returns a multi-point interpolated along a line at a fraction of total 2D length.

if repeat is false, the result is a single point, (and equivalent to ST_LineInterpolatePoint), otherwise, the result is a multi-point with points repeated at the fraction interval.

ST_LineMerge

Signatures

```
GEOMETRY ST_LineMerge (geom GEOMETRY)
GEOMETRY ST_LineMerge (geom GEOMETRY, preserve_direction BOOLEAN)
```

Description

”Merges” the input line geometry, optionally taking direction into account.

ST_LineString2DFromWKB

Signature

```
GEOMETRY ST_LineString2DFromWKB (linestring LINESTRING_2D)
```

Description

Deserialize a LINESTRING_2D from a WKB encoded blob

ST_LineSubstring

Signature

```
GEOMETRY ST_LineSubstring (line GEOMETRY, start_fraction DOUBLE, end_fraction DOUBLE)
```

Description

Returns a substring of a line between two fractions of total 2D length.

ST_M

Signature

```
DOUBLE ST_M (geom GEOMETRY)
```

Description

Returns the M coordinate of a point geometry

Example

```
SELECT ST_M(ST_Point(1, 2, 3, 4))
```

ST_MMax

Signature

```
DOUBLE ST_MMax (geom GEOMETRY)
```

Description

Returns the maximum M coordinate of a geometry

Example

```
SELECT ST_MMax(ST_Point(1, 2, 3, 4))
```

ST_MMin

Signature

```
DOUBLE ST_MMin (geom GEOMETRY)
```

Description

Returns the minimum M coordinate of a geometry

Example

```
SELECT ST_MMin(ST_Point(1, 2, 3, 4))
```

ST_MakeEnvelope

Signature

```
GEOMETRY ST_MakeEnvelope (min_x DOUBLE, min_y DOUBLE, max_x DOUBLE, max_y DOUBLE)
```

Description

Create a rectangular polygon from min/max coordinates

ST_MakeLine

Signatures

```
GEOMETRY ST_MakeLine (geoms GEOMETRY[])
GEOMETRY ST_MakeLine (start GEOMETRY, end GEOMETRY)
```

Description

Create a LINESTRING from a list of POINT geometries

Example

```
SELECT ST_MakeLine([ST_Point(0, 0), ST_Point(1, 1)]);
----
LINESTRING(0 0, 1 1)
```

ST_MakePolygon

Signatures

```
GEOMETRY ST_MakePolygon (shell GEOMETRY)
GEOMETRY ST_MakePolygon (shell GEOMETRY, holes GEOMETRY[])
```

Description

Create a POLYGON from a LINESTRING shell

Example

```
SELECT ST_MakePolygon(ST_LineString([ST_Point(0, 0), ST_Point(1, 0), ST_Point(1, 1), ST_Point(0, 0)]));
```

ST_MakeValid

Signature

```
GEOMETRY ST_MakeValid (geom GEOMETRY)
```

Description

Returns a valid representation of the geometry

ST_MaximumInscribedCircle

Signatures

```
STRUCT(center GEOMETRY, nearest GEOMETRY, radius DOUBLE) ST_MaximumInscribedCircle (geom GEOMETRY)
STRUCT(center GEOMETRY, nearest GEOMETRY, radius DOUBLE) ST_MaximumInscribedCircle (geom GEOMETRY,
tolerance DOUBLE)
```

Description

Returns the maximum inscribed circle of the input geometry, optionally with a tolerance.

By default, the tolerance is computed as `max(width, height) / 1000`. The return value is a struct with the center of the circle, the nearest point to the center on the boundary of the geometry, and the radius of the circle.

Example

```
-- Find the maximum inscribed circle of a square
SELECT ST_MaximumInscribedCircle(
    ST_GeomFromText('POLYGON((0 0, 10 0, 10 10, 0 10, 0 0))')
);
----
{'center': POINT (5 5), 'nearest': POINT (5 0), 'radius': 5.0}
```

ST_MinimumRotatedRectangle

Signature

```
GEOMETRY ST_MinimumRotatedRectangle (geom GEOMETRY)
```

Description

Returns the minimum rotated rectangle that bounds the input geometry, finding the surrounding box that has the lowest area by using a rotated rectangle, rather than taking the lowest and highest coordinate values as per ST_Envelope().

ST_Multi

Signature

```
GEOMETRY ST_Multi (geom GEOMETRY)
```

Description

Turns a single geometry into a multi geometry.

If the geometry is already a multi geometry, it is returned as is.

Example

```
SELECT ST_Multi(ST_GeomFromText('POINT(1 2)'));
----
MULTIPOINT (1 2)

SELECT ST_Multi(ST_GeomFromText('LINESTRING(1 1, 2 2)'));
----
MULTILINESTRING ((1 1, 2 2))

SELECT ST_Multi(ST_GeomFromText('POLYGON((0 0, 0 1, 1 1, 1 0, 0 0))'));
----
MULTIPOLYGON (((0 0, 0 1, 1 1, 1 0, 0 0)))
```

ST_NGeometries

Signature

```
INTEGER ST_NGeometries (geom GEOMETRY)
```

Description

Returns the number of component geometries in a collection geometry. If the input geometry is not a collection, this function returns 0 or 1 depending on if the geometry is empty or not.

ST_NInteriorRings

Signatures

```
INTEGER ST_NInteriorRings (geom GEOMETRY)
INTEGER ST_NInteriorRings (polygon POLYGON_2D)
```

Description

Returns the number of interior rings of a polygon

ST_NPoints

Signatures

```
UINTEGER ST_NPoints (geom GEOMETRY)
UBIGINT ST_NPoints (point POINT_2D)
UBIGINT ST_NPoints (linestring LINESTRING_2D)
UBIGINT ST_NPoints (polygon POLYGON_2D)
UBIGINT ST_NPoints (box BOX_2D)
```

Description

Returns the number of vertices within a geometry

ST_Node

Signature

```
GEOMETRY ST_Node (geom GEOMETRY)
```

Description

Returns a ”noded” MultiLinestring, produced by combining a collection of input linestrings and adding additional vertices where they intersect.

Example

```
-- Create a noded multilinestring from two intersecting lines
SELECT ST_Node(
    ST_GeomFromText('MULTILINESTRING((0 0, 2 2), (0 2, 2 0))')
);
----
MULTILINESTRING ((0 0, 1 1), (1 1, 2 2), (0 2, 1 1), (1 1, 2 0))
```

ST_Normalize

Signature

```
GEOMETRY ST_Normalize (geom GEOMETRY)
```

Description

Returns the ”normalized” representation of the geometry

ST_NumGeometries

Signature

```
INTEGER ST_NumGeometries (geom GEOMETRY)
```

Description

Returns the number of component geometries in a collection geometry. If the input geometry is not a collection, this function returns 0 or 1 depending on if the geometry is empty or not.

ST_NumInteriorRings

Signatures

```
INTEGER ST_NumInteriorRings (geom GEOMETRY)
INTEGER ST_NumInteriorRings (polygon POLYGON_2D)
```

Description

Returns the number of interior rings of a polygon

ST_NumPoints

Signatures

```
UINTEGER ST_NumPoints (geom GEOMETRY)
UBIGINT ST_NumPoints (point POINT_2D)
UBIGINT ST_NumPoints (linestring LINESTRING_2D)
UBIGINT ST_NumPoints (polygon POLYGON_2D)
UBIGINT ST_NumPoints (box BOX_2D)
```

Description

Returns the number of vertices within a geometry

ST_Overlaps

Signature

```
BOOLEAN ST_Overlaps (geom1 GEOMETRY, geom2 GEOMETRY)
```

Description

Returns true if the geometries overlap

ST_Perimeter

Signatures

```
DOUBLE ST_Perimeter (geom GEOMETRY)
DOUBLE ST_Perimeter (polygon POLYGON_2D)
DOUBLE ST_Perimeter (box BOX_2D)
```

Description

Returns the length of the perimeter of the geometry

ST_Perimeter_Spheroid

Signatures

```
DOUBLE ST_Perimeter_Spheroid (geom GEOMETRY)
DOUBLE ST_Perimeter_Spheroid (poly POLYGON_2D)
```

Description

Returns the length of the perimeter in meters using an ellipsoidal model of the earths surface

The input geometry is assumed to be in the EPSG:4326 coordinate system (WGS84), with [latitude, longitude] axis order and the length is returned in meters. This function uses the GeographicLib library, calculating the perimeter using an ellipsoidal model of the earth. This is a highly accurate method for calculating the perimeter of a polygon taking the curvature of the earth into account, but is also the slowest.

Returns `0.0` for any geometry that is not a `POLYGON`, `MULTIPOLYGON` or `GEOMETRYCOLLECTION` containing polygon geometries.

ST_Point

Signature

```
GEOMETRY ST_Point (x DOUBLE, y DOUBLE)
```

Description

Creates a GEOMETRY point

ST_Point2D

Signature

```
POINT_2D ST_Point2D (x DOUBLE, y DOUBLE)
```

Description

Creates a POINT_2D

ST_Point2DFromWKB

Signature

```
GEOMETRY ST_Point2DFromWKB (point POINT_2D)
```

Description

Deserialize a POINT_2D from a WKB encoded blob

ST_Point3D

Signature

```
POINT_3D ST_Point3D (x DOUBLE, y DOUBLE, z DOUBLE)
```

Description

Creates a POINT_3D

ST_Point4D

Signature

```
POINT_4D ST_Point4D (x DOUBLE, y DOUBLE, z DOUBLE, m DOUBLE)
```

Description

Creates a POINT_4D

ST_PointN

Signatures

```
GEOMETRY ST_PointN (geom GEOMETRY, index INTEGER)
POINT_2D ST_PointN (linestring LINESTRING_2D, index INTEGER)
```

Description

Returns the n'th vertex from the input geometry as a point geometry

---

ST_PointOnSurface

Signature

```
GEOMETRY ST_PointOnSurface (geom GEOMETRY)
```

Description

Returns a point guaranteed to lie on the surface of the geometry

---

ST_Points

Signature

```
GEOMETRY ST_Points (geom GEOMETRY)
```

Description

Collects all the vertices in the geometry into a MULTIPOINT

Example

```
select st_points('LINESTRING(1 1, 2 2)'::geometry);
----
MULTIPOINT (1 1, 2 2)

select st_points('MULTIPOLYGON Z EMPTY'::geometry);
----
MULTIPOINT Z EMPTY
```

ST_Polygon2DFromWKB

Signature

```
GEOMETRY ST_Polygon2DFromWKB (polygon POLYGON_2D)
```

Description

Deserialize a POLYGON_2D from a WKB encoded blob

ST_Polygonize

Signature

```
GEOMETRY ST_Polygonize (geometries GEOMETRY[])
```

Description

Returns a polygonized representation of the input geometries

Example

```
-- Create a polygon from a closed linestring ring
SELECT ST_Polygonize([
    ST_GeomFromText('LINESTRING(0 0, 0 10, 10 10, 10 0, 0 0)')
]);
```

```
GEOMETRYCOLLECTION (POLYGON ((0 0, 0 10, 10 10, 10 0, 0 0)))
```

ST_QuadKey

Signatures

```
VARCHAR ST_QuadKey (longitude DOUBLE, latitude DOUBLE, level INTEGER)
VARCHAR ST_QuadKey (point GEOMETRY, level INTEGER)
```

Description

Compute the quadkey for a given lon/lat point at a given level. Note that the parameter order is **longitude**, **latitude**.

`level` has to be between 1 and 23, inclusive.

The input coordinates will be clamped to the lon/lat bounds of the earth (longitude between -180 and 180, latitude between -85.05112878 and 85.05112878).

The geometry overload throws an error if the input geometry is not a `POINT`

Example

```
SELECT ST_QuadKey(st_point(11.08, 49.45), 10);
----
1333203202
```

ST_ReducePrecision

Signature

```
GEOMETRY ST_ReducePrecision (geom GEOMETRY, precision DOUBLE)
```

Description

Returns the geometry with all vertices reduced to the given precision

---

ST_RemoveRepeatedPoints

Signatures

```
LINESTRING_2D ST_RemoveRepeatedPoints (line LINESTRING_2D)
LINESTRING_2D ST_RemoveRepeatedPoints (line LINESTRING_2D, tolerance DOUBLE)
GEOMETRY ST_RemoveRepeatedPoints (geom GEOMETRY)
GEOMETRY ST_RemoveRepeatedPoints (geom GEOMETRY, tolerance DOUBLE)
```

Description

Remove repeated points from a LINESTRING.

ST_Reverse

Signature

```
GEOMETRY ST_Reverse (geom GEOMETRY)
```

Description

Returns the geometry with the order of its vertices reversed

ST_ShortestLine

Signature

```
GEOMETRY ST_ShortestLine (geom1 GEOMETRY, geom2 GEOMETRY)
```

Description

Returns the shortest line between two geometries

ST_Simplify

Signature

```
GEOMETRY ST_Simplify (geom GEOMETRY, tolerance DOUBLE)
```

Description

Returns a simplified version of the geometry

ST_SimplifyPreserveTopology

Signature

```
GEOMETRY ST_SimplifyPreserveTopology (geom GEOMETRY, tolerance DOUBLE)
```

Description

Returns a simplified version of the geometry that preserves topology

---

ST_StartPoint

Signatures

```
GEOMETRY ST_StartPoint (geom GEOMETRY)
POINT_2D ST_StartPoint (line LINESTRING_2D)
```

Description

Returns the start point of a LINESTRING.

ST_TileEnvelope

Signature

```
GEOMETRY ST_TileEnvelope (tile_zoom INTEGER, tile_x INTEGER, tile_y INTEGER)
```

Description

The `ST_TileEnvelope` scalar function generates tile envelope rectangular polygons from specified zoom level and tile indices.

This is used in MVT generation to select the features corresponding to the tile extent. The envelope is in the Web Mercator coordinate reference system (EPSG:3857). The tile pyramid starts at zoom level 0, corresponding to a single tile for the world. Each zoom level doubles the number of tiles in each direction, such that zoom level 1 is 2 tiles wide by 2 tiles high, zoom level 2 is 4 tiles wide by 4 tiles high, and so on. Tile indices start at `[x=0, y=0]` at the top left, and increase down and right. For example, at zoom level 2, the top right tile is `[x=3, y=0]`, the bottom left tile is `[x=0, y=3]`, and the bottom right is `[x=3, y=3]`.

```
SELECT ST_TileEnvelope(2, 3, 1);
```

Example

```
SELECT ST_TileEnvelope(2, 3, 1);
```

```
st_tileenvelope(2, 3, 1)
geometry
POLYGON ((1.00188E+07 0, 1.00188E+07 1.00188E+07, 2.00375E+07 1.00188E+07, 2.00375E+07 0, 1.00188E+07 0))
```

ST_Touches

Signature

```
BOOLEAN ST_Touches (geom1 GEOMETRY, geom2 GEOMETRY)
```

Description

Returns true if the geometries touch

ST_Transform

Signatures

```
BOX_2D ST_Transform (box BOX_2D, source_crs VARCHAR, target_crs VARCHAR)
BOX_2D ST_Transform (box BOX_2D, source_crs VARCHAR, target_crs VARCHAR, always_xy BOOLEAN)
POINT_2D ST_Transform (point POINT_2D, source_crs VARCHAR, target_crs VARCHAR)
POINT_2D ST_Transform (point POINT_2D, source_crs VARCHAR, target_crs VARCHAR, always_xy BOOLEAN)
GEOMETRY ST_Transform (geom GEOMETRY, source_crs VARCHAR, target_crs VARCHAR)
GEOMETRY ST_Transform (geom GEOMETRY, source_crs VARCHAR, target_crs VARCHAR, always_xy BOOLEAN)
```

Description

Transforms a geometry between two coordinate systems

The source and target coordinate systems can be specified using any format that the PROJ library supports.

The third optional `always_xy` parameter can be used to force the input and output geometries to be interpreted as having a [easting, northing] coordinate axis order regardless of what the source and target coordinate system definition says. This is particularly useful when transforming to/from the WGS84/EPSG:4326 coordinate system (what most people think of when they hear "longitude"/"latitude" or "GPS coordinates"), which is defined as having a [latitude, longitude] axis order even though [longitude, latitude] is commonly used in practice (e.g. in GeoJSON). More details available in the PROJ documentation.

DuckDB spatial vendors its own static copy of the PROJ database of coordinate systems, so if you have your own installation of PROJ on your system the available coordinate systems may differ to what's available in other GIS software.

Example

```
-- Transform a geometry from EPSG:4326 to EPSG:3857 (WGS84 to WebMercator)
-- Note that since WGS84 is defined as having a [latitude, longitude] axis order
-- we follow the standard and provide the input geometry using that axis order,
-- but the output will be [easting, northing] because that is what's defined by
-- WebMercator.

SELECT
    ST_Transform(
        st_point(52.373123, 4.892360),
        'EPSG:4326',
        'EPSG:3857'
    );
----
POINT (544615.0239773799 6867874.103539125)

-- Alternatively, let's say we got our input point from e.g. a GeoJSON file,
-- which uses WGS84 but with [longitude, latitude] axis order. We can use the
-- `always_xy` parameter to force the input geometry to be interpreted as having
-- a [northing, easting] axis order instead, even though the source coordinate
-- reference system definition (WGS84) says otherwise.

SELECT
    ST_Transform(
        -- note the axis order is reversed here
        st_point(4.892360, 52.373123),
        'EPSG:4326',
        'EPSG:3857',
        always_xy := true
    );
----
POINT (544615.0239773799 6867874.103539125)

-- Transform a geometry from OSG36 British National Grid EPSG:27700 to EPSG:4326 WGS84
-- Standard transform is often fine for the first few decimal places before being wrong
-- which could result in an error starting at about 10m and possibly much more
SELECT ST_Transform(bng, 'EPSG:27700', 'EPSG:4326', xy := true) AS without_grid_file
FROM (SELECT ST_GeomFromText('POINT( 170370.718 11572.405 )') AS bng);
----
POINT (-5.202992651563592 49.96007490162923)

-- By using an official NTv2 grid file, we can reduce the error down around the 9th decimal place
-- which in theory is below a millimetre, and in practise unlikely that your coordinates are that precise
```

```
-- British National Grid "NTv2 format files" download available here:
-- https://www.ordnancesurvey.co.uk/products/os-net/for-developers
SELECT ST_Transform(bng
    , '+proj=tmerc +lat_0=49 +lon_0=-2 +k=0.9996012717 +x_0=400000 +y_0=-100000 +ellps=airy +units=m
+no_defs +nadgrids=/full/path/to/OSTN15-NTv2/OSTN15_NTv2_OSGBtoETRS.gsb +type=crs'
    , 'EPSG:4326', xy := true) AS with_grid_file
FROM (SELECT ST_GeomFromText('POINT( 170370.718 11572.405 )') AS bng) t;
----
POINT (-5.203046090608746 49.96006137018598)
```

ST_Union

Signature

```
GEOMETRY ST_Union (geom1 GEOMETRY, geom2 GEOMETRY)
```

Description

Returns the union of two geometries

ST_VoronoiDiagram

Signature

```
GEOMETRY ST_VoronoiDiagram (geom GEOMETRY)
```

Description

Returns the Voronoi diagram of the supplied MultiPoint geometry

---

ST_Within

Signatures

```
BOOLEAN ST_Within (geom1 POINT_2D, geom2 POLYGON_2D)
BOOLEAN ST_Within (geom1 GEOMETRY, geom2 GEOMETRY)
```

Description

Returns true if the first geometry is within the second

ST_WithinProperly

Signature

```
BOOLEAN ST_WithinProperly (geom1 GEOMETRY, geom2 GEOMETRY)
```

Description

Returns true if the first geometry ”properly” is contained by the second geometry

This function functions the same as `ST_ContainsProperly`, but the arguments are swapped.

ST_X

Signatures

```
DOUBLE ST_X (geom GEOMETRY)
DOUBLE ST_X (point POINT_2D)
```

Description

Returns the X coordinate of a point geometry

Example

```
SELECT ST_X(ST_Point(1, 2))
```

ST_XMax

Signatures

```
DOUBLE ST_XMax (geom GEOMETRY)
DOUBLE ST_XMax (point POINT_2D)
DOUBLE ST_XMax (line LINESTRING_2D)
DOUBLE ST_XMax (polygon POLYGON_2D)
DOUBLE ST_XMax (box BOX_2D)
FLOAT ST_XMax (box BOX_2DF)
```

Description

Returns the maximum X coordinate of a geometry

Example

```
SELECT ST_XMax(ST_Point(1, 2))
```

ST_XMin

Signatures

```
DOUBLE ST_XMin (geom GEOMETRY)
DOUBLE ST_XMin (point POINT_2D)
DOUBLE ST_XMin (line LINESTRING_2D)
DOUBLE ST_XMin (polygon POLYGON_2D)
DOUBLE ST_XMin (box BOX_2D)
FLOAT ST_XMin (box BOX_2DF)
```

Description

Returns the minimum X coordinate of a geometry

Example

```
SELECT ST_XMin(ST_Point(1, 2))
```

ST_Y

Signatures

```
DOUBLE ST_Y (geom GEOMETRY)
DOUBLE ST_Y (point POINT_2D)
```

Description

Returns the Y coordinate of a point geometry

Example

```
SELECT ST_Y(ST_Point(1, 2))
```

ST_YMax

Signatures

```
DOUBLE ST_YMax (geom GEOMETRY)
DOUBLE ST_YMax (point POINT_2D)
DOUBLE ST_YMax (line LINESTRING_2D)
DOUBLE ST_YMax (polygon POLYGON_2D)
DOUBLE ST_YMax (box BOX_2D)
FLOAT ST_YMax (box BOX_2DF)
```

Description

Returns the maximum Y coordinate of a geometry

Example

```
SELECT ST_YMax(ST_Point(1, 2))
```

ST_YMin

Signatures

```
DOUBLE ST_YMin (geom GEOMETRY)
DOUBLE ST_YMin (point POINT_2D)
DOUBLE ST_YMin (line LINESTRING_2D)
DOUBLE ST_YMin (polygon POLYGON_2D)
DOUBLE ST_YMin (box BOX_2D)
FLOAT ST_YMin (box BOX_2DF)
```

Description

Returns the minimum Y coordinate of a geometry

Example

```
SELECT ST_YMin(ST_Point(1, 2))
```

ST_Z

Signature

```
DOUBLE ST_Z (geom GEOMETRY)
```

Description

Returns the Z coordinate of a point geometry

Example

```
SELECT ST_Z(ST_Point(1, 2, 3))
```

ST_ZMFlag

Signatures

```
UTINYINT ST_ZMFlag (geom GEOMETRY)
UTINYINT ST_ZMFlag (wkb WKB_BLOB)
```

Description

Returns a flag indicating the presence of Z and M values in the input geometry. 0 = No Z or M values 1 = M values only 2 = Z values only 3 = Z and M values

Example

```
-- ZMFlag for a 2D geometry
SELECT ST_ZMFlag(ST_GeomFromText('POINT(1 1)'));
----
0

-- ZMFlag for a 3DZ geometry
SELECT ST_ZMFlag(ST_GeomFromText('POINT Z(1 1 1)'));
----
2

-- ZMFlag for a 3DM geometry
SELECT ST_ZMFlag(ST_GeomFromText('POINT M(1 1 1)'));
----
1

-- ZMFlag for a 4D geometry
SELECT ST_ZMFlag(ST_GeomFromText('POINT ZM(1 1 1 1)'));
----
3
```

ST_ZMax

Signature

```
DOUBLE ST_ZMax (geom GEOMETRY)
```

Description

Returns the maximum Z coordinate of a geometry

Example

```
SELECT ST_ZMax(ST_Point(1, 2, 3))
```

ST_ZMin

Signature

```
DOUBLE ST_ZMin (geom GEOMETRY)
```

Description

Returns the minimum Z coordinate of a geometry

Example

```
SELECT ST_ZMin(ST_Point(1, 2, 3))
```

Aggregate Functions

ST_CoverageInvalidEdges_Agg

Signatures

```
GEOMETRY ST_CoverageInvalidEdges_Agg (col0 GEOMETRY)
GEOMETRY ST_CoverageInvalidEdges_Agg (col0 GEOMETRY, col1 DOUBLE)
```

Description

Returns the invalid edges of a coverage geometry

ST_CoverageSimplify_Agg

Signatures

```
GEOMETRY ST_CoverageSimplify_Agg (col0 GEOMETRY, col1 DOUBLE)
GEOMETRY ST_CoverageSimplify_Agg (col0 GEOMETRY, col1 DOUBLE, col2 BOOLEAN)
```

Description

Simplifies a set of geometries while maintaining coverage

ST_CoverageUnion_Agg

Signature

```
GEOMETRY ST_CoverageUnion_Agg (col0 GEOMETRY)
```

Description

Unions a set of geometries while maintaining coverage

ST_Envelope_Agg

Signature

```
GEOMETRY ST_Envelope_Agg (col0 GEOMETRY)
```

Description

Alias for ST_Extent_Agg.

Computes the minimal-bounding-box polygon containing the set of input geometries.

Example

```
SELECT ST_Extent_Agg(geom) FROM UNNEST([ST_Point(1,1), ST_Point(5,5)]) AS _(geom);
-- POLYGON ((1 1, 1 5, 5 5, 5 1, 1 1))
```

ST_Extent_Agg

Signature

```
GEOMETRY ST_Extent_Agg (col0 GEOMETRY)
```

Description

Computes the minimal-bounding-box polygon containing the set of input geometries

Example

```
SELECT ST_Extent_Agg(geom) FROM UNNEST([ST_Point(1,1), ST_Point(5,5)]) AS _(geom);
-- POLYGON ((1 1, 1 5, 5 5, 5 1, 1 1))
```

ST_Intersection_Agg

Signature

```
GEOMETRY ST_Intersection_Agg (col0 GEOMETRY)
```

Description

Computes the intersection of a set of geometries

ST_MemUnion_Agg

Signature

```
GEOMETRY ST_MemUnion_Agg (col0 GEOMETRY)
```

Description

Computes the union of a set of input geometries. ”Slower, but might be more memory efficient than ST_UnionAgg as each geometry is merged into the union individually rather than all at once.

ST_Union_Agg

Signature

```
GEOMETRY ST_Union_Agg (col0 GEOMETRY)
```

Description

Computes the union of a set of input geometries

Macro Functions

ST_Rotate

Signature

```
GEOMETRY ST_Rotate (geom GEOMETRY, radians double)
```

Description

Alias of ST_RotateZ

ST_RotateX

Signature

```
GEOMETRY ST_RotateX (geom GEOMETRY, radians double)
```

Description

Rotates a geometry around the X axis. This is a shorthand macro for calling ST_Affine.

Example

```
-- Rotate a 3D point 90 degrees (π/2 radians) around the X-axis
SELECT ST_RotateX(ST_GeomFromText('POINT Z(0 1 0)'), pi()/2);
----
POINT Z (0 0 1)
```

ST_RotateY

Signature

```
GEOMETRY ST_RotateY (geom GEOMETRY, radians double)
```

Description

Rotates a geometry around the Y axis. This is a shorthand macro for calling ST_Affine.

Example

```
-- Rotate a 3D point 90 degrees (π/2 radians) around the Y-axis
SELECT ST_RotateY(ST_GeomFromText('POINT Z(1 0 0)'), pi()/2);
----
POINT Z (0 0 -1)
```

ST_RotateZ

Signature

```
GEOMETRY ST_RotateZ (geom GEOMETRY, radians double)
```

Description

Rotates a geometry around the Z axis. This is a shorthand macro for calling ST_Affine.

Example

```
-- Rotate a point 90 degrees (π/2 radians) around the Z-axis
SELECT ST_RotateZ(ST_Point(1, 0), pi()/2);
----
POINT (0 1)
```

ST_Scale

Signatures

```
GEOMETRY ST_Scale (geom GEOMETRY, xs double, ys double, zs double)
GEOMETRY ST_Scale (geom GEOMETRY, xs double, ys double)
```

ST_TransScale

Signature

```
GEOMETRY ST_TransScale (geom GEOMETRY, dx double, dy double, xs double, ys double)
```

Description

Translates and then scales a geometry in X and Y direction. This is a shorthand macro for calling ST_Affine.

Example

```
-- Translate by (1, 2) then scale by (2, 3)
SELECT ST_TransScale(ST_Point(1, 1), 1, 2, 2, 3);
----
POINT (4 9)
```

ST_Translate

Signatures

```
GEOMETRY ST_Translate (geom GEOMETRY, dx double, dy double, dz double)
GEOMETRY ST_Translate (geom GEOMETRY, dx double, dy double)
```

Table Functions

ST_Drivers

Signature

```
ST_Drivers ()
```

Description

Returns the list of supported GDAL drivers and file formats

Note that far from all of these drivers have been tested properly. Some may require additional options to be passed to work as expected. If you run into any issues please first consult the consult the GDAL docs.

Example

```
SELECT * FROM ST_Drivers();
```

ST_GeneratePoints

Signature

```
ST_GeneratePoints (col0 BOX_2D, col1 BIGINT)
ST_GeneratePoints (col0 BOX_2D, col1 BIGINT, col2 BIGINT)
```

Description

Generates a set of random points within the specified bounding box.

Takes a bounding box (min_x, min_y, max_x, max_y), a count of points to generate, and optionally a seed for the random number generator.

Example

```
SELECT * FROM ST_GeneratePoints({min_x: 0, min_y:0, max_x:10, max_y:10}::BOX_2D, 5, 42);
```

ST_Read

Signature

```
ST_Read (col0 VARCHAR, keep_wkb BOOLEAN, max_batch_size INTEGER, sequential_layer_scan BOOLEAN, layer
VARCHAR, sibling_files VARCHAR[], spatial_filter WKB_BLOB, spatial_filter_box BOX_2D, allowed_drivers
VARCHAR[], open_options VARCHAR[])
```

Description

Read and import a variety of geospatial file formats using the GDAL library.

The ST_Read table function is based on the GDAL translator library and enables reading spatial data from a variety of geospatial vector file formats as if they were DuckDB tables.

> See ST_Drivers for a list of supported file formats and drivers.

Except for the `path` parameter, all parameters are optional.

Parameter	Type	Description
`path`	VARCHAR	The path to the file to read. Mandatory
`sequential_layer_scan`	BOOLEAN	If set to true, the table function will scan through all layers sequentially and return the first layer that matches the given layer name. This is required for some drivers to work properly, e.g., the OSM driver.

Parameter	Type	Description
`spatial_filter`	WKB_BLOB	If set to a WKB blob, the table function will only return rows that intersect with the given WKB geometry. Some drivers may support efficient spatial filtering natively, in which case it will be pushed down. Otherwise the filtering is done by GDAL which may be much slower.
`open_options`	VARCHAR[]	A list of key-value pairs that are passed to the GDAL driver to control the opening of the file. E.g., the GeoJSON driver supports a FLATTEN_NESTED_ATTRIBUTES=YES option to flatten nested attributes.
`layer`	VARCHAR	The name of the layer to read from the file. If NULL, the first layer is returned. Can also be a layer index (starting at 0).
`allowed_drivers`	VARCHAR[]	A list of GDAL driver names that are allowed to be used to open the file. If empty, all drivers are allowed.
`sibling_files`	VARCHAR[]	A list of sibling files that are required to open the file. E.g., the ESRI Shapefile driver requires a .shx file to be present. Although most of the time these can be discovered automatically.
`spatial_filter_box`	BOX_2D	If set to a BOX_2D, the table function will only return rows that intersect with the given bounding box. Similar to spatial_filter.
`keep_wkb`	BOOLEAN	If set, the table function will return geometries in a wkb_ geometry column with the type WKB_BLOB (which can be cast to BLOB) instead of GEOMETRY. This is useful if you want to use DuckDB with more exotic geometry subtypes that DuckDB spatial doesnt support representing in the GEOMETRY type yet.

Note that GDAL is single-threaded, so this table function will not be able to make full use of parallelism.

By using `ST_Read`, the spatial extension also provides "replacement scans" for common geospatial file formats, allowing you to query files of these formats as if they were tables directly.

```
SELECT * FROM './path/to/some/shapefile/dataset.shp';
```

In practice this is just syntax-sugar for calling ST_Read, so there is no difference in performance. If you want to pass additional options, you should use the ST_Read table function directly.

The following formats are currently recognized by their file extension:

Format	Extension
ESRI ShapeFile	.shp
GeoPackage	.gpkg
FlatGeoBuf	.fgb

Example

```
-- Read a Shapefile
```

```
SELECT * FROM ST_Read('some/file/path/filename.shp');

-- Read a GeoJSON file
CREATE TABLE my_geojson_table AS SELECT * FROM ST_Read('some/file/path/filename.json');
```

ST_ReadOSM

Signature

```
ST_ReadOSM (col0 VARCHAR)
```

Description

The `ST_ReadOsm()` table function enables reading compressed OpenStreetMap data directly from a `.osm.pbf file.`

This function uses multithreading and zero-copy protobuf parsing which makes it a lot faster than using the `ST_Read()` OSM driver, however it only outputs the raw OSM data (Nodes, Ways, Relations), without constructing any geometries. For simple node entities (like PoI's) you can trivially construct POINT geometries, but it is also possible to construct LINESTRING and POLYGON geometries by manually joining refs and nodes together in SQL, although with available memory usually being a limiting factor. The `ST_ReadOSM()` function also provides a "replacement scan" to enable reading from a file directly as if it were a table. This is just syntax sugar for calling `ST_ReadOSM()` though. Example:

```
SELECT * FROM 'tmp/data/germany.osm.pbf' LIMIT 5;
```

Example

```
SELECT *
FROM ST_ReadOSM('tmp/data/germany.osm.pbf')
WHERE tags['highway'] != []
LIMIT 5;
----
┌──────────────────────┬────────┬──────────────────────┬────────┬──────────────────────┬──────────────┬──────────────┬──────────────┐
│         kind         │   id   │         tags         │  refs  │        lat         │    lon     │ ref_
roles │        ref_types        │
│ enum('node', 'way'…  │ int64  │ map(varchar, varch…  │ int64[] │       double       │   double   │
varchar[] │ enum('node', 'way', …  │
├──────────────────────┼────────┼──────────────────────┼────────┼──────────────────────┼──────────────┼──────────────┼──────────────┤
│ node                 │ 122351 │ {bicycle=yes, butt…  │        │         53.5492951 │   9.977553 │
│                      │
│ node                 │ 122397 │ {crossing=no, high…  │        │ 53.520990100000006 │ 10.0156924 │
│                      │
│ node                 │ 122493 │ {TMC:cid_58:tabcd_…  │        │ 53.129614600000004 │  8.1970173 │
│                      │
│ node                 │ 123566 │ {highway=traffic_s…  │        │ 54.617268200000005 │  8.9718171 │
│                      │
│ node                 │ 125801 │ {TMC:cid_58:tabcd_…  │        │ 53.070685000000005 │  8.7819939 │
│                      │
└──────────────────────┴────────┴──────────────────────┴────────┴──────────────────────┴──────────────┴──────────────┴──────────────┘
```

ST_ReadSHP

Signature

```
ST_ReadSHP (col0 VARCHAR, encoding VARCHAR)
```

Description

Read a Shapefile without relying on the GDAL library

ST_Read_Meta

Signature

```
ST_Read_Meta (col0 VARCHAR)
ST_Read_Meta (col0 VARCHAR[])
```

Description

Read the metadata from a variety of geospatial file formats using the GDAL library.

The `ST_Read_Meta` table function accompanies the `ST_Read` table function, but instead of reading the contents of a file, this function scans the metadata instead. Since the data model of the underlying GDAL library is quite flexible, most of the interesting metadata is within the returned `layers` column, which is a somewhat complex nested structure of DuckDB `STRUCT` and `LIST` types.

Example

```
-- Find the coordinate reference system authority name and code for the first layers first geometry
column in the file
SELECT
    layers[1].geometry_fields[1].crs.auth_name as name,
    layers[1].geometry_fields[1].crs.auth_code as code
FROM st_read_meta('../../tmp/data/amsterdam_roads.fgb');
```

R-Tree Indexes

As of DuckDB v1.1.0 the `spatial` extension provides basic support for spatial indexing through the R-tree extension index type.

Why Should I Use an R-Tree Index?

When working with geospatial datasets, it is very common that you want to filter rows based on their spatial relationship with a specific region of interest. Unfortunately, even though DuckDB's vectorized execution engine is pretty fast, this sort of operation does not scale very well to large datasets as it always requires a full table scan to check every row in the table. However, by indexing a table with an R-tree, it is possible to accelerate these types of queries significantly.

How Do R-Tree Indexes Work?

An R-tree is a balanced tree data structure that stores the approximate *minimum bounding rectangle* of each geometry (and the internal ID of the corresponding row) in the leaf nodes, and the bounding rectangle enclosing all of the child nodes in each internal node.

> The *minimum bounding rectangle* (MBR) of a geometry is the smallest rectangle that completely encloses the geometry. Usually when we talk about the bounding rectangle of a geometry (or the bounding "box" in the context of 2D geometry), we mean the minimum bounding rectangle. Additionally, we tend to assume that bounding boxes/rectangles are *axis-aligned,* i.e., the rectangle is **not** rotated – the sides are always parallel to the coordinate axes. The MBR of a point is the point itself.

By traversing the R-tree from top to bottom, it is possible to very quickly search a R-tree-indexed table for only those rows where the indexed geometry column intersect a specific region of interest, as you can skip searching entire sub-trees if the bounding rectangles of their parent nodes don't intersect the query region at all. Once the leaf nodes are reached, only the specific rows whose geometries intersect the query region have to be fetched from disk, and the often much more expensive exact spatial predicate check (and any other filters) only have to be executed for these rows.

What Are the Limitations of R-Tree Indexes in DuckDB?

Before you get started using the R-tree index, there are some limitations to be aware of:

- The R-tree index is only supported for the `GEOMETRY` data type.
- The R-tree index will only be used to perform "index scans" when the table is filtered (using a `WHERE` clause) with one of the following spatial predicate functions (as they all imply intersection): `ST_Equals`, `ST_Intersects`, `ST_Touches`, `ST_Crosses`, `ST_Within`, `ST_Contains`, `ST_Overlaps`, `ST_Covers`, `ST_CoveredBy`, `ST_ContainsProperly`.
- One of the arguments to the spatial predicate function must be a "constant" (i.e., an expression whose result is known at query planning time). This is because the query planner needs to know the bounding box of the query region *before* the query itself is executed in order to use the R-tree index scan.

In the future we want to enable R-tree indexes to be used to accelerate additional predicate functions and more complex queries such a spatial joins.

How to Use R-Tree Indexes in DuckDB

To create an R-tree index, simply use the `CREATE INDEX` statement with the `USING RTREE` clause, passing the geometry column to index within the parentheses. For example:

```
-- Create a table with a geometry column
CREATE TABLE my_table (geom GEOMETRY);

-- Create an R-tree index on the geometry column
CREATE INDEX my_idx ON my_table USING RTREE (geom);
```

You can also pass in additional options when creating an R-tree index using the `WITH` clause to control the behavior of the R-tree index. For example, to specify the maximum number of entries per node in the R-tree, you can use the `max_node_capacity` option:

```
CREATE INDEX my_idx ON my_table USING RTREE (geom) WITH (max_node_capacity = 16);
```

The impact tweaking these options will have on performance is highly dependent on the system setup DuckDB is running on, the spatial distribution of the dataset, and the query patterns of your specific workload. The defaults should be good enough, but you if you want to experiment with different parameters, see the full list of options here.

Example

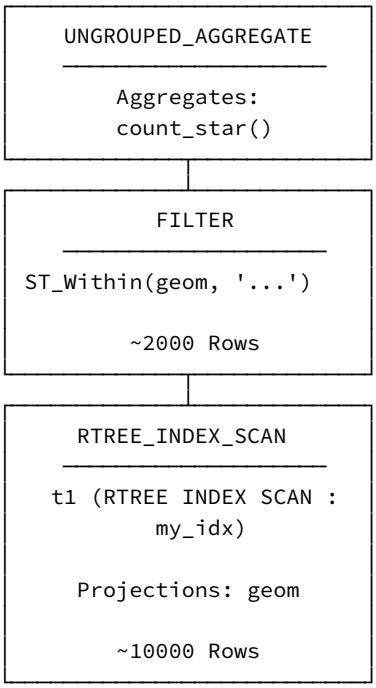
Here is an example that shows how to create an R-tree index on a geometry column and where we can see that the RTREE_INDEX_SCAN operator is used when the table is filtered with a spatial predicate:

```
INSTALL spatial;
LOAD spatial;

-- Create a table with 10_000_000 random points
CREATE TABLE t1 AS SELECT point::GEOMETRY AS geom
FROM st_generatepoints({min_x: 0, min_y: 0, max_x: 100, max_y: 100}::BOX_2D, 10_000, 1337);

-- Create an index on the table.
CREATE INDEX my_idx ON t1 USING RTREE (geom);

-- Perform a query with a "spatial predicate" on the indexed geometry column
-- Note how the second argument in this case, the ST_MakeEnvelope call is a "constant"
SELECT count(*) FROM t1 WHERE ST_Within(geom, ST_MakeEnvelope(45, 45, 65, 65));
```

```
390
```

We can check for ourselves that an R-tree index scan is used by using the EXPLAIN statement:

```
EXPLAIN SELECT count(*) FROM t1 WHERE ST_Within(geom, ST_MakeEnvelope(45, 45, 65, 65));
```



Performance Considerations

Bulk Loading & Maintenance

Creating R-trees on top of an already populated table is much faster than first creating the index and then inserting the data. This is because the R-tree will have to periodically rebalance itself and perform a somewhat costly splitting operation when a node reaches max capacity after an insert, potentially causing additional splits to cascade up the tree. However, when the R-tree index is created on an already populated table, a special bottom up ”bulk loading algorithm” (Sort-Tile-Recursive) is used, which divides all entries into an already balanced tree as the total number of required nodes can be computed from the beginning.

Additionally, using the bulk loading algorithm tends to create a R-tree with a better structure (less overlap between bounding boxes), which usually leads to better query performance. If you find that the performance of querying the R-tree starts to deteriorate after a large number of updates or deletions, dropping and re-creating the index might produce a higher quality R-tree.

Memory Usage

Like DuckDB's built in ART-index, all the associated buffers containing the R-tree will be lazily loaded from disk (when running DuckDB in disk-backed mode), but they are currently never unloaded unless the index is dropped. This means that if you end up scanning the entire index, the entire index will be loaded into memory and stay there for the duration of the database connection. However, all memory used by the R-tree index (even during bulk-loading) is tracked by DuckDB, and will count towards the memory limit set by the `memory_limit` configuration parameter.

Tuning

Depending on you specific workload, you might want to experiment with the `max_node_capacity` and `min_node_capacity` options to change the structure of the R-tree and how it responds to insertions and deletions, see the full list of options here. In general, a tree with a higher total number of nodes (i.e., a lower `max_node_capacity`) *may* result in a more granular structure that enables more aggressive pruning of sub-trees during query execution, but it will also require more memory to store the tree itself and be more punishing when querying larger regions as more internal nodes will have to be traversed.

Options

The following options can be passed to the `WITH` clause when creating an R-tree index: (e.g., `CREATE INDEX my_idx ON my_table USING RTREE (geom) WITH (<option> = <value>);{:.language-sql .highlight}`)

Option	Description	Default
`max_node_capacity`	The maximum number of entries per node in the R-tree	`128`
`min_node_capacity`	The minimum number of entries per node in the R-tree	`0.4 * max_node_capacity`

*Should a node fall under the minimum number of entries after a deletion, the node will be dissolved and all the entries reinserted from the top of the tree. This is a common operation in R-tree implementations to prevent the tree from becoming too unbalanced.

R-Tree Table Functions

The `rtree_index_dump(VARCHAR)` table function can be used to return all the nodes within an R-tree index which might come on handy when debugging, profiling or otherwise just inspecting the structure of the index. The function takes the name of the R-tree index as an argument and returns a table with the following columns:

Column name	Type	Description
`level`	`INTEGER`	The level of the node in the R-tree. The root node has level `0`
`bounds`	`BOX_2DF`	The bounding box of the node
`row_id`	`ROW_TYPE`	If this is a leaf node, the `rowid` of the row in the table, otherwise `NULL`

Example:

```sql
-- Create a table with 64 random points
CREATE TABLE t1 AS SELECT point::GEOMETRY AS geom
FROM st_generatepoints({min_x: 0, min_y: 0, max_x: 100, max_y: 100}::BOX_2D, 64, 1337);

-- Create an R-tree index on the geometry column (with a low max_node_capacity for demonstration
purposes)
CREATE INDEX my_idx ON t1 USING RTREE (geom) WITH (max_node_capacity = 4);

-- Inspect the R-tree index. Notice how the area of the bounding boxes of the branch nodes
-- decreases as we go deeper into the tree.
SELECT
  level,
  bounds::GEOMETRY AS geom,
  CASE WHEN row_id IS NULL THEN st_area(geom) ELSE NULL END AS area,
  row_id,
  CASE WHEN row_id IS NULL THEN 'branch' ELSE 'leaf' END AS kind
FROM rtree_index_dump('my_idx')
ORDER BY area DESC;
```

level int32	geom geometry	area double	row_id int64	kind varchar
0	POLYGON ((2.17285037040710…	3286.396482226409		branch
0	POLYGON ((6.00962591171264…	3193.725100864862		branch
0	POLYGON ((0.74995160102844…	3099.921458393704		branch
0	POLYGON ((14.6168870925903…	2322.2760491675654		branch
1	POLYGON ((2.17285037040710…	604.1520104388514		branch
1	POLYGON ((26.6022186279296…	569.1665467030252		branch
1	POLYGON ((35.7942314147949…	435.24662436250037		branch
1	POLYGON ((62.2643051147460…	396.39027683023596		branch
1	POLYGON ((59.5225715637207…	386.09153403820187		branch
1	POLYGON ((82.3060836791992…	369.15115640929434		branch
·	·	·	·	·
·	·	·	·	·
·	·	·	·	·
2	POLYGON ((20.5411434173584…		35	leaf
2	POLYGON ((14.6168870925903…		36	leaf
2	POLYGON ((43.7271652221679…		39	leaf
2	POLYGON ((53.4629211425781…		44	leaf
2	POLYGON ((26.6022186279296…		62	leaf
2	POLYGON ((53.1732063293457…		63	leaf
2	POLYGON ((78.1427154541015…		10	leaf
2	POLYGON ((75.1728591918945…		15	leaf
2	POLYGON ((62.2643051147460…		42	leaf
2	POLYGON ((80.5032577514648…		49	leaf

84 rows (20 shown) 5 columns

GDAL Integration

The spatial extension integrates the GDAL translator library to read and write spatial data from a variety of geospatial vector file formats. See the documentation for the `st_read` table function for how to make use of this in practice.

In order to spare users from having to setup and install additional dependencies on their system, the spatial extension bundles its own copy of the GDAL library. This also means that spatial's version of GDAL may not be the latest version available or provide support for all of the file formats that a system-wide GDAL installation otherwise would. Refer to the section on the `st_drivers` table function to inspect

which GDAL drivers are currently available.

GDAL Based COPY Function

The spatial extension does not only enable *importing* geospatial file formats (through the `ST_Read` function), it also enables *exporting* DuckDB tables to different geospatial vector formats through a GDAL based `COPY` function.

For example, to export a table to a GeoJSON file, with generated bounding boxes, you can use the following query:

```
COPY <table> TO 'some/file/path/filename.geojson'
WITH (FORMAT gdal, DRIVER 'GeoJSON', LAYER_CREATION_OPTIONS 'WRITE_BBOX=YES');
```

Available options:

- `FORMAT`: is the only required option and must be set to `GDAL` to use the GDAL based copy function.
- `DRIVER`: is the GDAL driver to use for the export. Use `ST_Drivers()` to list the names of all available drivers.
- `LAYER_CREATION_OPTIONS`: list of options to pass to the GDAL driver. See the GDAL docs for the driver you are using for a list of available options.
- SRS: Set a spatial reference system as metadata to use for the export. This can be a WKT string, an EPSG code or a proj-string, basically anything you would normally be able to pass to GDAL. Note that this will **not** perform any reprojection of the input geometry, it just sets the metadata if the target driver supports it.

Limitations

Note that only vector based drivers are supported by the GDAL integration. Reading and writing raster formats are not supported.

SQLite Extension

The SQLite extension allows DuckDB to directly read and write data from a SQLite database file. The data can be queried directly from the underlying SQLite tables. Data can be loaded from SQLite tables into DuckDB tables, or vice versa.

Installing and Loading

The `sqlite` extension will be transparently autoloaded on first use from the official extension repository. If you would like to install and load it manually, run:

```
INSTALL sqlite;
LOAD sqlite;
```

Usage

To make a SQLite file accessible to DuckDB, use the `ATTACH` statement with the `sqlite` or `sqlite_scanner` type. Attached SQLite databases support both read and write operations.

For example, to attach to the `sakila.db` file, run:

```
ATTACH 'sakila.db' (TYPE sqlite);
USE sakila;
```

The tables in the file can be read as if they were normal DuckDB tables, but the underlying data is read directly from the SQLite tables in the file at query time.

```
SHOW TABLES;
```

name
actor
address
category
city
country
customer
customer_list
film
film_actor
film_category
film_list
film_text
inventory

name
language
payment
rental
sales_by_film_category
sales_by_store
staff
staff_list
store

You can query the tables using SQL, e.g., using the example queries from `sakila-examples.sql`:

```
SELECT
    cat.name AS category_name,
    sum(ifnull(pay.amount, 0)) AS revenue
FROM category cat
LEFT JOIN film_category flm_cat
       ON cat.category_id = flm_cat.category_id
LEFT JOIN film fil
       ON flm_cat.film_id = fil.film_id
LEFT JOIN inventory inv
       ON fil.film_id = inv.film_id
LEFT JOIN rental ren
       ON inv.inventory_id = ren.inventory_id
LEFT JOIN payment pay
       ON ren.rental_id = pay.rental_id
GROUP BY cat.name
ORDER BY revenue DESC
LIMIT 5;
```

Data Types

SQLite is a weakly typed database system. As such, when storing data in a SQLite table, types are not enforced. The following is valid SQL in SQLite:

```
CREATE TABLE numbers (i INTEGER);
INSERT INTO numbers VALUES ('hello');
```

DuckDB is a strongly typed database system, as such, it requires all columns to have defined types and the system rigorously checks data for correctness.

When querying SQLite, DuckDB must deduce a specific column type mapping. DuckDB follows SQLite's type affinity rules with a few extensions.

1. If the declared type contains the string `INT` then it is translated into the type `BIGINT`
2. If the declared type of the column contains any of the strings `CHAR`, `CLOB`, or `TEXT` then it is translated into `VARCHAR`.
3. If the declared type for a column contains the string `BLOB` or if no type is specified then it is translated into `BLOB`.
4. If the declared type for a column contains any of the strings `REAL`, `FLOA`, `DOUB`, `DEC` or `NUM` then it is translated into `DOUBLE`.
5. If the declared type is `DATE`, then it is translated into `DATE`.
6. If the declared type contains the string `TIME`, then it is translated into `TIMESTAMP`.
7. If none of the above apply, then it is translated into `VARCHAR`.

As DuckDB enforces the corresponding columns to contain only correctly typed values, we cannot load the string “hello” into a column of type `BIGINT`. As such, an error is thrown when reading from the “numbers” table above:

```
Mismatch Type Error: Invalid type in column "i": column was declared as integer, found "hello" of type
"text" instead.
```

This error can be avoided by setting the `sqlite_all_varchar` option:

```
SET GLOBAL sqlite_all_varchar = true;
```

When set, this option overrides the type conversion rules described above, and instead always converts the SQLite columns into a VARCHAR column. Note that this setting must be set *before* `sqlite_attach` is called.

Opening SQLite Databases Directly

SQLite databases can also be opened directly and can be used transparently instead of a DuckDB database file. In any client, when connecting, a path to a SQLite database file can be provided and the SQLite database will be opened instead.

For example, with the shell, a SQLite database can be opened as follows:

```
duckdb sakila.db
```

```
SELECT first_name
FROM actor
LIMIT 3;
```

first_name
PENELOPE
NICK
ED

Writing Data to SQLite

In addition to reading data from SQLite, the extension also allows you to create new SQLite database files, create tables, ingest data into SQLite and make other modifications to SQLite database files using standard SQL queries.

This allows you to use DuckDB to, for example, export data that is stored in a SQLite database to Parquet, or read data from a Parquet file into SQLite.

Below is a brief example of how to create a new SQLite database and load data into it.

```
ATTACH 'new_sqlite_database.db' AS sqlite_db (TYPE sqlite);
CREATE TABLE sqlite_db.tbl (id INTEGER, name VARCHAR);
INSERT INTO sqlite_db.tbl VALUES (42, 'DuckDB');
```

The resulting SQLite database can then be read into from SQLite.

```
sqlite3 new_sqlite_database.db
```

```
SQLite version 3.39.5 2022-10-14 20:58:05
sqlite> SELECT * FROM tbl;
```

```
id  name
--  ------
42  DuckDB
```

Many operations on SQLite tables are supported. All these operations directly modify the SQLite database, and the result of subsequent operations can then be read using SQLite.

Concurrency

DuckDB can read or modify a SQLite database while DuckDB or SQLite reads or modifies the same database from a different thread or a separate process. More than one thread or process can read the SQLite database at the same time, but only a single thread or process can write to the database at one time. Database locking is handled by the SQLite library, not DuckDB. Within the same process, SQLite uses mutexes. When accessed from different processes, SQLite uses file system locks. The locking mechanisms also depend on SQLite configuration, like WAL mode. Refer to the SQLite documentation on locking for more information.

> **Warning.** Linking multiple copies of the SQLite library into the same application can lead to application errors. See sqlite_scanner Issue #82 for more information.

Settings

The extension exposes the following configuration parameters.

Name	Description	Default
`sqlite_debug_show_queries`	DEBUG SETTING: print all queries sent to SQLite to stdout	`false`

Supported Operations

Below is a list of supported operations.

CREATE TABLE

```
CREATE TABLE sqlite_db.tbl (id INTEGER, name VARCHAR);
```

INSERT INTO

```
INSERT INTO sqlite_db.tbl VALUES (42, 'DuckDB');
```

SELECT

```
SELECT * FROM sqlite_db.tbl;
```

id	name
42	DuckDB

COPY

```
COPY sqlite_db.tbl TO 'data.parquet';
COPY sqlite_db.tbl FROM 'data.parquet';
```

UPDATE

```
UPDATE sqlite_db.tbl SET name = 'Woohoo' WHERE id = 42;
```

DELETE

```
DELETE FROM sqlite_db.tbl WHERE id = 42;
```

ALTER TABLE

```
ALTER TABLE sqlite_db.tbl ADD COLUMN k INTEGER;
```

DROP TABLE

```
DROP TABLE sqlite_db.tbl;
```

CREATE VIEW

```
CREATE VIEW sqlite_db.v1 AS SELECT 42;
```

Transactions

```
CREATE TABLE sqlite_db.tmp (i INTEGER);
```

```
BEGIN;
INSERT INTO sqlite_db.tmp VALUES (42);
SELECT * FROM sqlite_db.tmp;
```

i
42

```
ROLLBACK;
SELECT * FROM sqlite_db.tmp;
```

i

> **Deprecated.** The old `sqlite_attach` function is deprecated. It is recommended to switch over to the new ATTACH syntax.

TPC-DS Extension

The `tpcds` extension implements the data generator and queries for the TPC-DS benchmark.

Installing and Loading

The `tpcds` extension will be transparently autoloaded on first use from the official extension repository. If you would like to install and load it manually, run:

```
INSTALL tpcds;
LOAD tpcds;
```

Usage

To generate data for scale factor 1, use:

```
CALL dsdgen(sf = 1);
```

To run a query, e.g., query 8, use:

```
PRAGMA tpcds(8);
```

s_store_name	sum(ss_net_profit)
able	-10354620.18
ation	-10576395.52
bar	-10625236.01
ese	-10076698.16
ought	-10994052.78

Generating the Schema

It's possible to generate the schema of TPC-DS without any data by setting the scale factor to 0:

```
CALL dsdgen(sf = 0);
```

Limitations

The `tpcds(<query_id>)`{:.language-sql .highlight} function runs a fixed TPC-DS query with pre-defined bind parameters (a.k.a. substitution parameters). It is not possible to change the query parameters using the `tpcds` extension.

TPC-H Extension

The `tpch` extension implements the data generator and queries for the TPC-H benchmark.

Installing and Loading

The `tpch` extension is shipped by default in some DuckDB builds, otherwise it will be transparently autoloaded on first use. If you would like to install and load it manually, run:

```
INSTALL tpch;
LOAD tpch;
```

Usage

Generating Data

To generate data for scale factor 1, use:

```
CALL dbgen(sf = 1);
```

Calling `dbgen` does not clean up existing TPC-H tables. To clean up existing tables, use `DROP TABLE` before running `dbgen`:

```
DROP TABLE IF EXISTS customer;
DROP TABLE IF EXISTS lineitem;
DROP TABLE IF EXISTS nation;
DROP TABLE IF EXISTS orders;
DROP TABLE IF EXISTS part;
DROP TABLE IF EXISTS partsupp;
DROP TABLE IF EXISTS region;
DROP TABLE IF EXISTS supplier;
```

Running a Query

To run a query, e.g., query 4, use:

```
PRAGMA tpch(4);
```

o_orderpriority	order_count
1-URGENT	10594
2-HIGH	10476
3-MEDIUM	10410
4-NOT SPECIFIED	10556
5-LOW	10487

Listing Queries

To list all 22 queries, run:

```
FROM tpch_queries();
```

This function returns a table with columns `query_nr` and `query`.

Listing Expected Answers

To produced the expected results for all queries on scale factors 0.01, 0.1, and 1, run:

```
FROM tpch_answers();
```

This function returns a table with columns `query_nr`, `scale_factor`, and `answer`.

Generating the Schema

It's possible to generate the schema of TPC-H without any data by setting the scale factor to 0:

```
CALL dbgen(sf = 0);
```

Data Generator Parameters

The data generator function `dbgen` has the following parameters:

Name	Type	Description
`catalog`	`VARCHAR`	Target catalog
`children`	`UINTEGER`	Number of partitions
`overwrite`	`BOOLEAN`	(Not used)
`sf`	`DOUBLE`	Scale factor
`step`	`UINTEGER`	Defines the partition to be generated, indexed from 0 to `children` - 1. Must be defined when the `children` arguments is defined
`suffix`	`VARCHAR`	Append the `suffix` to table names

Pre-Generated Data Sets

Pre-generated DuckDB databases for TPC-H are available for download:

- `tpch-sf1.db` (250 MB)
- `tpch-sf3.db` (754 MB)
- `tpch-sf10.db` (2.5 GB)
- `tpch-sf30.db` (7.6 GB)
- `tpch-sf100.db` (26 GB)
- `tpch-sf300.db` (78 GB)
- `tpch-sf1000.db` (265 GB)
- `tpch-sf3000.db` (796 GB)

Resource Usage of the Data Generator

Generating TPC-H data sets for large scale factors takes a significant amount of time. Additionally, *if the generation is performed in a single step,* it requires a large amount of memory. The following table gives an estimate on the resources required to produce DuckDB database files containing the generated TPC-H data set using 128 threads.

Scale factor	Database size	Generation time	Single-step generation's memory usage
100	26 GB	17 minutes	71 GB
300	78 GB	51 minutes	211 GB
1,000	265 GB	2 h 53 minutes	647 GB
3,000	796 GB	8 h 30 minutes	1799 GB

The numbers shown above were achieved by running the `dbgen` function in a single step, for example:

```
CALL dbgen(sf = 300);
```

If you have a limited amount of memory available, you can run the `dbgen` function in steps. For example, you may generate SF300 in 10 steps:

```
CALL dbgen(sf = 300, children = 10, step = 0);
CALL dbgen(sf = 300, children = 10, step = 1);
...
CALL dbgen(sf = 300, children = 10, step = 9);
```

Limitation

The `tpch(<query_id>)`{:.language-sql .highlight} function runs a fixed TPC-H query with pre-defined bind parameters (a.k.a. substitution parameters). It is not possible to change the query parameters using the `tpch` extension. To run the queries with the parameters prescribed by the TPC-H benchmark, use a TPC-H framework implementation.

UI Extension

The `ui` extension adds a user interface for your local DuckDB instance.

The UI is built and maintained by MotherDuck. An overview of its features can be found in the MotherDuck documentation.

Requirements

- An environment with a browser.
- Any DuckDB client except Wasm, v1.2.1 or later.

Usage

To start the UI from the command line:

```
duckdb -ui
```

To start the UI from SQL:

```
CALL start_ui();
```

Running either of these will open the UI in your default browser.

The UI connects to the DuckDB instance it was started from, so any data you've already loaded will be available. Since this instance is a native process (not Wasm), it can leverage all the resources of your local environment: all cores, memory, and files. Closing this instance will cause the UI to stop working.

The UI is served from an HTTP server embedded in DuckDB. To start this server without launching the browser, run:

```
CALL start_ui_server();
```

You can then load the UI in your browser by navigating to `http://localhost:4213`.

To stop the HTTP server, run:

```
CALL stop_ui_server();
```

Local Query Execution

By default, the DuckDB UI runs your queries fully locally: your queries and data never leave your computer. If you would like to use MotherDuck through the UI, you have to opt-in explicitly and sign into MotherDuck.

Configuration

Local Port

The local port of the HTTP server can be configured with a SQL command like:

```
SET ui_local_port = 4213;
```

The environment variable `ui_local_port` can also be used.

The default port is 4213. (Why? 4 = D, 21 = U, 3 = C)

Remote URL

The local HTTP server fetches the files for the UI from a remote HTTP server so they can be kept up-to-date.

The default URL for the remote server is https://ui.duckdb.org.

An alternate remote URL can be configured with a SQL command like:

```
SET ui_remote_url = 'https://ui.duckdb.org';
```

The environment variable `ui_remote_port` can also be used.

This setting is available mainly for testing purposes.

Be sure you trust any URL you configure, as the application can access the data you load into DuckDB.

Because of this risk, the setting is only respected if `allow_unsigned_extensions` is enabled.

Polling Interval

The UI extension polls for some information on a background thread. It watches for changes to the list of attached databases, and it detects when you connect to MotherDuck.

These checks take very little time to complete, so the default polling interval is short (284 milliseconds). You can configure it with a SQL command like:

```
SET ui_polling_interval = 284;
```

The environment variable `ui_polling_interval` can also be used.

Setting the polling interval to 0 will disable polling entirely. This is not recommended, as the list of databases in the UI could get out of date, and some ways of connecting to MotherDuck will not work properly.

Tips

Opening a CSV File with the DuckDB UI

Using the DuckDB CLI client, you can start the UI with a CSV available as a view using the `-cmd` argument:

```
duckdb -cmd "CREATE VIEW <view_name> AS FROM '<filename>.csv';" -ui
```

Running the UI in Read-Only Mode

The DuckDB UI uses DuckDB tables as storage internally (e.g., for saving notebooks). Therefore, running the UI directly on a read-only database is not supported:

```
duckdb -ui -readonly read_only_test.db
```

In the UI, this results in:

```
Catalog Error: SET schema: No catalog + schema named "memory.main" found.
```

To work around this, run the UI on another database file:

```
duckdb -ui ui_catalog.db
```

Then, open a notebook and attach to the database:

```
ATTACH 'test.db' (READ_ONLY) AS my_db;
USE my_db
```

Limitations

- The UI currently does not support the ARM-based Windows platforms (`windows_arm64` and `windows_arm64_mingw`).

Vector Similarity Search Extension

The `vss` extension is an experimental extension for DuckDB that adds indexing support to accelerate vector similarity search queries using DuckDB's new fixed-size ARRAY type.

See the announcement blog post and the "What's New in the Vector Similarity Search Extension?" post.

Usage

To create a new HNSW (Hierarchical Navigable Small Worlds) index on a table with an ARRAY column, use the `CREATE INDEX` statement with the `USING HNSW` clause. For example:

```
INSTALL vss;
LOAD vss;

CREATE TABLE my_vector_table (vec FLOAT[3]);
INSERT INTO my_vector_table
    SELECT array_value(a, b, c)
    FROM range(1, 10) ra(a), range(1, 10) rb(b), range(1, 10) rc(c);
CREATE INDEX my_hnsw_index ON my_vector_table USING HNSW (vec);
```

The index will then be used to accelerate queries that use an `ORDER BY` clause evaluating one of the supported distance metric functions against the indexed columns and a constant vector, followed by a `LIMIT` clause. For example:

```
SELECT *
FROM my_vector_table
ORDER BY array_distance(vec, [1, 2, 3]::FLOAT[3])
LIMIT 3;
```

Additionally, the overloaded `min_by(col, arg, n)` can also be accelerated with the HNSW index if the `arg` argument is a matching distance metric function. This can be used to do quick one-shot nearest neighbor searches. For example, to get the top 3 rows with the closest vectors to `[1, 2, 3]`:

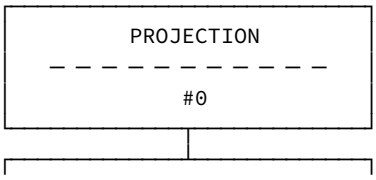
```
SELECT min_by(my_vector_table, array_distance(vec, [1, 2, 3]::FLOAT[3]), 3 ORDER BY vec) AS result
FROM my_vector_table;
```

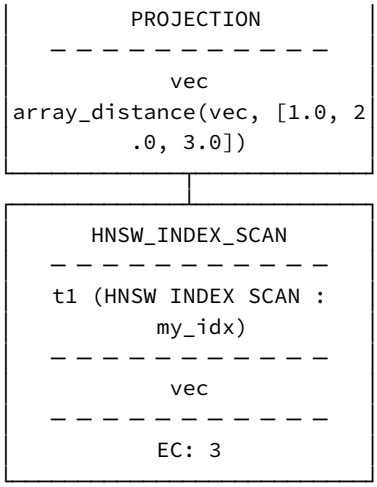
```
[{'vec': [1.0, 2.0, 3.0]}, {'vec': [2.0, 2.0, 3.0]}, {'vec': [1.0, 2.0, 4.0]}]
```

Note how we pass the table name as the first argument to `min_by` to return a struct containing the entire matched row.

We can verify that the index is being used by checking the `EXPLAIN` output and looking for the `HNSW_INDEX_SCAN` node in the plan:

```
EXPLAIN
SELECT *
FROM my_vector_table
ORDER BY array_distance(vec, [1, 2, 3]::FLOAT[3])
LIMIT 3;
```





By default the HNSW index will be created using the euclidean distance `l2sq` (L2-norm squared) metric, matching DuckDBs `array_distance` function, but other distance metrics can be used by specifying the `metric` option during index creation. For example:

```
CREATE INDEX my_hnsw_cosine_index
ON my_vector_table
USING HNSW (vec)
WITH (metric = 'cosine');
```

The following table shows the supported distance metrics and their corresponding DuckDB functions

Metric	Function	Description
`l2sq`	`array_distance`	Euclidean distance
`cosine`	`array_cosine_distance`	Cosine similarity distance
`ip`	`array_negative_inner_product`	Negative inner product

Note that while each HNSW index only applies to a single column you can create multiple HNSW indexes on the same table each individually indexing a different column. Additionally, you can also create multiple HNSW indexes to the same column, each supporting a different distance metric.

Index Options

Besides the `metric` option, the HNSW index creation statement also supports the following options to control the hyperparameters of the index construction and search process:

Option	Default	Description
`ef_construction`	128	The number of candidate vertices to consider during the construction of the index. A higher value will result in a more accurate index, but will also increase the time it takes to build the index.
`ef_search`	64	The number of candidate vertices to consider during the search phase of the index. A higher value will result in a more accurate index, but will also increase the time it takes to perform a search.
`M`	16	The maximum number of neighbors to keep for each vertex in the graph. A higher value will result in a more accurate index, but will also increase the time it takes to build the index.

Option	Default	Description
`M0`	`2 * M`	The base connectivity, or the number of neighbors to keep for each vertex in the zero-th level of the graph. A higher value will result in a more accurate index, but will also increase the time it takes to build the index.

Additionally, you can also override the `ef_search` parameter set at index construction time by setting the `SET hnsw_ef_search = <int>`{:.language-sql .highlight} configuration option at runtime. This can be useful if you want to trade search performance for accuracy or vice-versa on a per-connection basis. You can also unset the override by calling `RESET hnsw_ef_search`{:.language-sql .highlight}.

Persistence

Due to some known issues related to persistence of custom extension indexes, the HNSW index can only be created on tables in in-memory databases by default, unless the `SET hnsw_enable_experimental_persistence = <bool>`{:.language-sql .highlight} configuration option is set to `true`.

The reasoning for locking this feature behind an experimental flag is that "WAL" recovery is not yet properly implemented for custom indexes, meaning that if a crash occurs or the database is shut down unexpectedly while there are uncommitted changes to a HNSW-indexed table, you can end up with **data loss or corruption of the index**.

If you enable this option and experience an unexpected shutdown, you can try to recover the index by first starting DuckDB separately, loading the `vss` extension and then ATTACHing the database file, which ensures that the HNSW index functionality is available during WAL-playback, allowing DuckDB's recovery process to proceed without issues. But we still recommend that you do not use this feature in production environments.

With the `hnsw_enable_experimental_persistence` option enabled, the index will be persisted into the DuckDB database file (if you run DuckDB with a disk-backed database file), which means that after a database restart, the index can be loaded back into memory from disk instead of having to be re-created. With that in mind, there are no incremental updates to persistent index storage, so every time DuckDB performs a checkpoint the entire index will be serialized to disk and overwrite itself. Similarly, after a restart of the database, the index will be deserialized back into main memory in its entirety. Although this will be deferred until you first access the table associated with the index. Depending on how large the index is, the deserialization process may take some time, but it should still be faster than simply dropping and re-creating the index.

Inserts, Updates, Deletes and Re-Compaction

The HNSW index does support inserting, updating and deleting rows from the table after index creation. However, there are two things to keep in mind:

- It's faster to create the index after the table has been populated with data as the initial bulk load can make better use of parallelism on large tables.
- Deletes are not immediately reflected in the index, but are instead "marked" as deleted, which can cause the index to grow stale over time and negatively impact query quality and performance.

To remedy the last point, you can call the `PRAGMA hnsw_compact_index('<index_name>')`{:.language-sql .highlight} pragma function to trigger a re-compaction of the index pruning deleted items, or re-create the index after a significant number of updates.

Bonus: Vector Similarity Search Joins

The `vss` extension also provides a couple of table macros to simplify matching multiple vectors against each other, so called "fuzzy joins". These are:

- `vss_join(left_table, right_table, left_col, right_col, k, metric := 'l2sq')`
- `vss_match(right_table", left_col, right_col, k, metric := 'l2sq')`

> k is the number of records to be selected from the `right_table` for each `left_table` record, ordered by the score.

These **do not** currently make use of the HNSW index but are provided as convenience utility functions for users who are ok with performing brute-force vector similarity searches without having to write out the join logic themselves. In the future these might become targets for index-based optimizations as well.

These functions can be used as follows:

```
CREATE TABLE haystack (id int, vec FLOAT[3]);
CREATE TABLE needle (search_vec FLOAT[3]);

INSERT INTO haystack
    SELECT row_number() OVER (), array_value(a, b, c)
    FROM range(1, 10) ra(a), range(1, 10) rb(b), range(1, 10) rc(c);

INSERT INTO needle
    VALUES ([5, 5, 5]), ([1, 1, 1]);

SELECT *
FROM vss_join(needle, haystack, search_vec, vec, 2) res;
```

score float	left_tbl struct(search_vec float[3])	right_tbl struct(id integer, vec float[3])
0.0	{'search_vec': [5.0, 5.0, 5.0]}	{'id': 365, 'vec': [5.0, 5.0, 5.0]}
1.0	{'search_vec': [5.0, 5.0, 5.0]}	{'id': 356, 'vec': [5.0, 5.0, 4.0]}
0.0	{'search_vec': [1.0, 1.0, 1.0]}	{'id': 1, 'vec': [1.0, 1.0, 1.0]}
1.0	{'search_vec': [1.0, 1.0, 1.0]}	{'id': 2, 'vec': [1.0, 2.0, 1.0]}

Alternatively, we can use the `vss_match` macro as a "lateral join" to get the matches already grouped by the left table. Note that this requires us to specify the left table first, and then the `vss_match` macro which references the search column from the left table (in this case, `search_vec`):

```
SELECT *
FROM needle, vss_match(haystack, search_vec, vec, 2) res;
```

search_vec float[3]	matches struct(score float, "row" struct(id integer, vec float[3]))[]
[5.0, 5.0, 5.0]	[{'score': 0.0, 'row': {'id': 365, 'vec': [5.0, 5.0, 5.0]}}, {'score': 1.0, 'row': {'id': 356, 'vec': [5.0, 5.0, 4.0]}}]
[1.0, 1.0, 1.0]	[{'score': 0.0, 'row': {'id': 1, 'vec': [1.0, 1.0, 1.0]}}, {'score': 1.0, 'row': {'id': 2, 'vec': [1.0, 2.0, 1.0]}}]

Limitations

- Only vectors consisting of `FLOAT`s (32-bit, single precision) are supported at the moment.
- The index itself is not buffer managed and must be able to fit into RAM memory.
- The size of the index in memory does not count towards DuckDB's `memory_limit` configuration parameter.

- HNSW indexes can only be created on tables in in-memory databases, unless the `SET hnsw_enable_experimental_persistence = <bool>` configuration option is set to `true`, see Persistence for more information.
- The vector join table macros (`vss_join` and `vss_match`) do not require or make use of the HNSW index.

Guides

Guides

The guides section contains compact how-to guides that are focused on achieving a single goal. For API references and examples, see the rest of the documentation.

Note that there are many tools using DuckDB, which are not covered in the official guides. To find a list of these tools, check out the Awesome DuckDB repository.

> **Tip.** For a short introductory tutorial, check out the "Analyzing Railway Traffic in the Netherlands" tutorial.

Data Import and Export

- Data import overview
- File access with the `file:` protocol

CSV Files

- How to load a CSV file into a table
- How to export a table to a CSV file

Parquet Files

- How to load a Parquet file into a table
- How to export a table to a Parquet file
- How to run a query directly on a Parquet file

HTTP(S), S3 and GCP

- How to load a Parquet file directly from HTTP(S)
- How to load a Parquet file directly from S3
- How to export a Parquet file to S3
- How to load a Parquet file from S3 Express One
- How to load a Parquet file directly from GCS
- How to load a Parquet file directly from Cloudflare R2
- How to load an Iceberg table directly from S3

JSON Files

- How to load a JSON file into a table
- How to export a table to a JSON file

Excel Files with the Spatial Extension

- How to load an Excel file into a table
- How to export a table to an Excel file

Querying Other Database Systems

- How to directly query a MySQL database
- How to directly query a PostgreSQL database
- How to directly query a SQLite database

Directly Reading Files

- How to directly read a binary file
- How to directly read a text file

Performance

- My workload is slow (troubleshooting guide)
- How to design the schema for optimal performance
- What is the ideal hardware environment for DuckDB
- What performance implications do Parquet files and (compressed) CSV files have
- How to tune workloads
- Benchmarks

Meta Queries

- How to list all tables
- How to view the schema of the result of a query
- How to quickly get a feel for a dataset using summarize
- How to view the query plan of a query
- How to profile a query

ODBC

- How to set up an ODBC application (and more!)

Python Client

- How to install the Python client
- How to execute SQL queries
- How to easily query DuckDB in Jupyter Notebooks
- How to easily query DuckDB in marimo Notebooks
- How to use Multiple Python Threads with DuckDB
- How to use fsspec filesystems with DuckDB

Pandas

- How to execute SQL on a Pandas DataFrame
- How to create a table from a Pandas DataFrame
- How to export data to a Pandas DataFrame

Apache Arrow

- How to execute SQL on Apache Arrow
- How to create a DuckDB table from Apache Arrow
- How to export data to Apache Arrow

Relational API

- How to query Pandas DataFrames with the Relational API

Python Library Integrations

- How to use Ibis to query DuckDB with or without SQL
- How to use DuckDB with Polars DataFrames via Apache Arrow

SQL Features

- Friendly SQL
- As-of join
- Full-text search
- `query` and `query_table` functions

SQL Editors and IDEs

- How to set up the DBeaver SQL IDE

Data Viewers

- How to visualize DuckDB databases with Tableau
- How to draw command-line plots with DuckDB and YouPlot

Data Viewers

Tableau – A Data Visualization Tool

Tableau is a popular commercial data visualization tool. In addition to a large number of built in connectors, it also provides generic database connectivity via ODBC and JDBC connectors.

Tableau has two main versions: Desktop and Online (Server).

- For Desktop, connecting to a DuckDB database is similar to working in an embedded environment like Python.
- For Online, since DuckDB is in-process, the data needs to be either on the server itself

or in a remote data bucket that is accessible from the server.

Database Creation

When using a DuckDB database file the data sets do not actually need to be imported into DuckDB tables; it suffices to create views of the data. For example, this will create a view of the `h2oai` Parquet test file in the current DuckDB code base:

```
CREATE VIEW h2oai AS (
    FROM read_parquet('/Users/username/duckdb/data/parquet-testing/h2oai/h2oai_group_small.parquet')
);
```

Note that you should use full path names to local files so that they can be found from inside Tableau. Also note that you will need to use a version of the driver that is compatible (i.e., from the same release) as the database format used by the DuckDB tool (e.g., Python module, command line) that was used to create the file.

Installing the JDBC Driver

Tableau provides documentation on how to install a JDBC driver for Tableau to use.

> Tableau (both Desktop and Server versions) need to be restarted any time you add or modify drivers.

Driver Links

The link here is for a recent version of the JDBC driver that is compatible with Tableau. If you wish to connect to a database file, you will need to make sure the file was created with a file-compatible version of DuckDB. Also, check that there is only one version of the driver installed as there are multiple filenames in use.

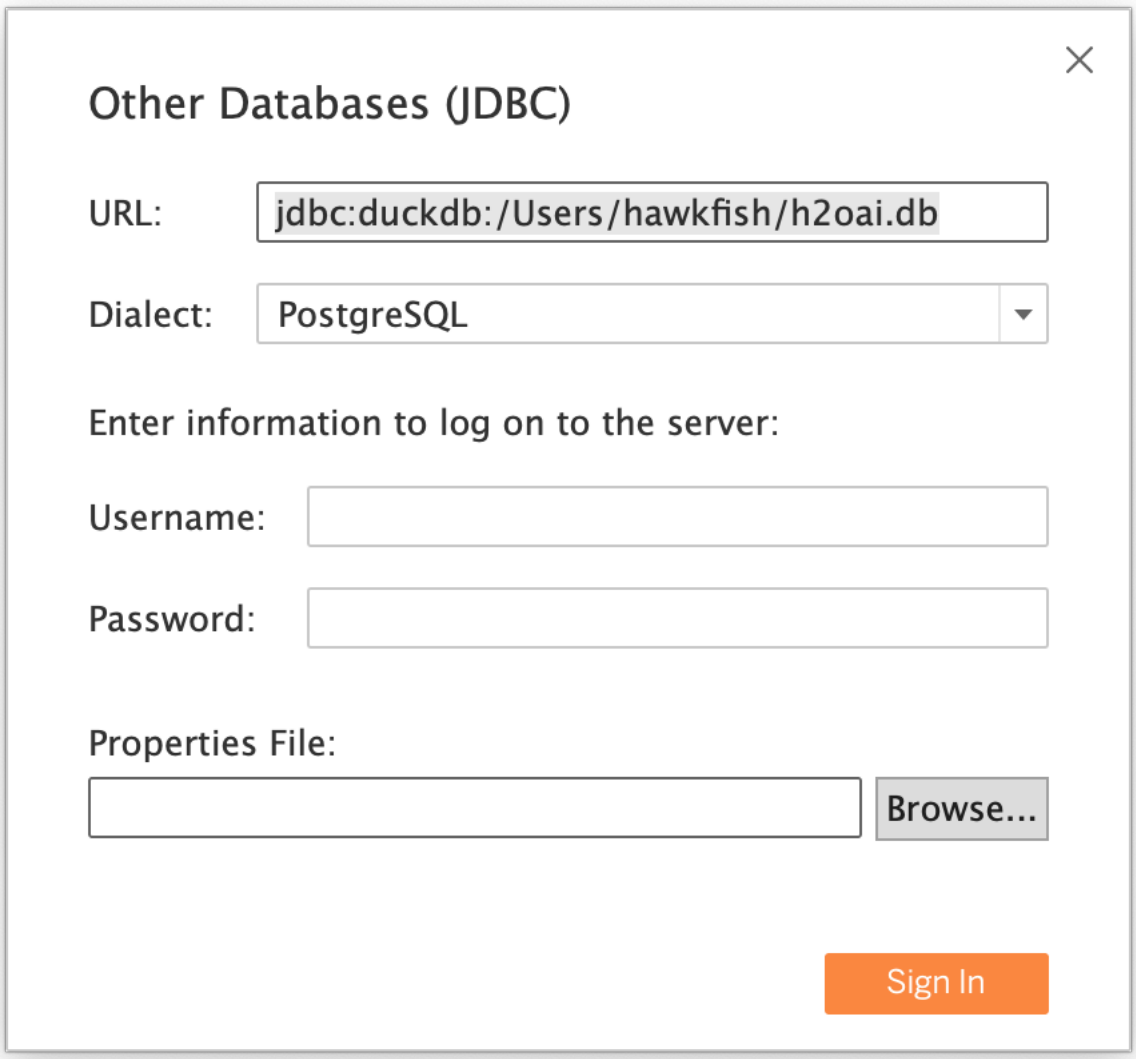
Download the JAR file.

- macOS: Copy it to `~/Library/Tableau/Drivers/`
- Windows: Copy it to `C:\Program Files\Tableau\Drivers`
- Linux: Copy it to `/opt/tableau/tableau_driver/jdbc`.

Using the PostgreSQL Dialect

If you just want to do something simple, you can try connecting directly to the JDBC driver and using Tableau-provided PostgreSQL dialect.

1. Create a DuckDB file containing your views and/or data.
2. Launch Tableau
3. Under Connect > To a Server > More… click on "Other Databases (JDBC)" This will bring up the connection dialogue box. For the URL, enter `jdbc:duckdb:/User/username/path/to/database.db`. For the Dialect, choose PostgreSQL. The rest of the fields can be ignored:



However, functionality will be missing such as `median` and `percentile` aggregate functions. To make the data source connection more compatible with the PostgreSQL dialect, please use the DuckDB taco connector as described below.

Installing the Tableau DuckDB Connector

While it is possible to use the Tableau-provided PostgreSQL dialect to communicate with the DuckDB JDBC driver, we strongly recommend using the DuckDB "taco" connector. This connector has been fully tested against the Tableau dialect generator and is more compatible than the provided PostgreSQL dialect.

The documentation on how to install and use the connector is in its repository, but essentially you will need the `duckdb_jdbc.taco` file. (Despite what the Tableau documentation says, the real security risk is in the JDBC driver code, not the small amount of JavaScript in the Taco.)

Server (Online)

On Linux, copy the Taco file to `/opt/tableau/connectors`. On Windows, copy the Taco file to `C:\Program Files\Tableau\Connectors`. Then issue these commands to disable signature validation:

```
tsm configuration set -k native_api.disable_verify_connector_plugin_signature -v true
```

```
tsm pending-changes apply
```

The last command will restart the server with the new settings.

macOS

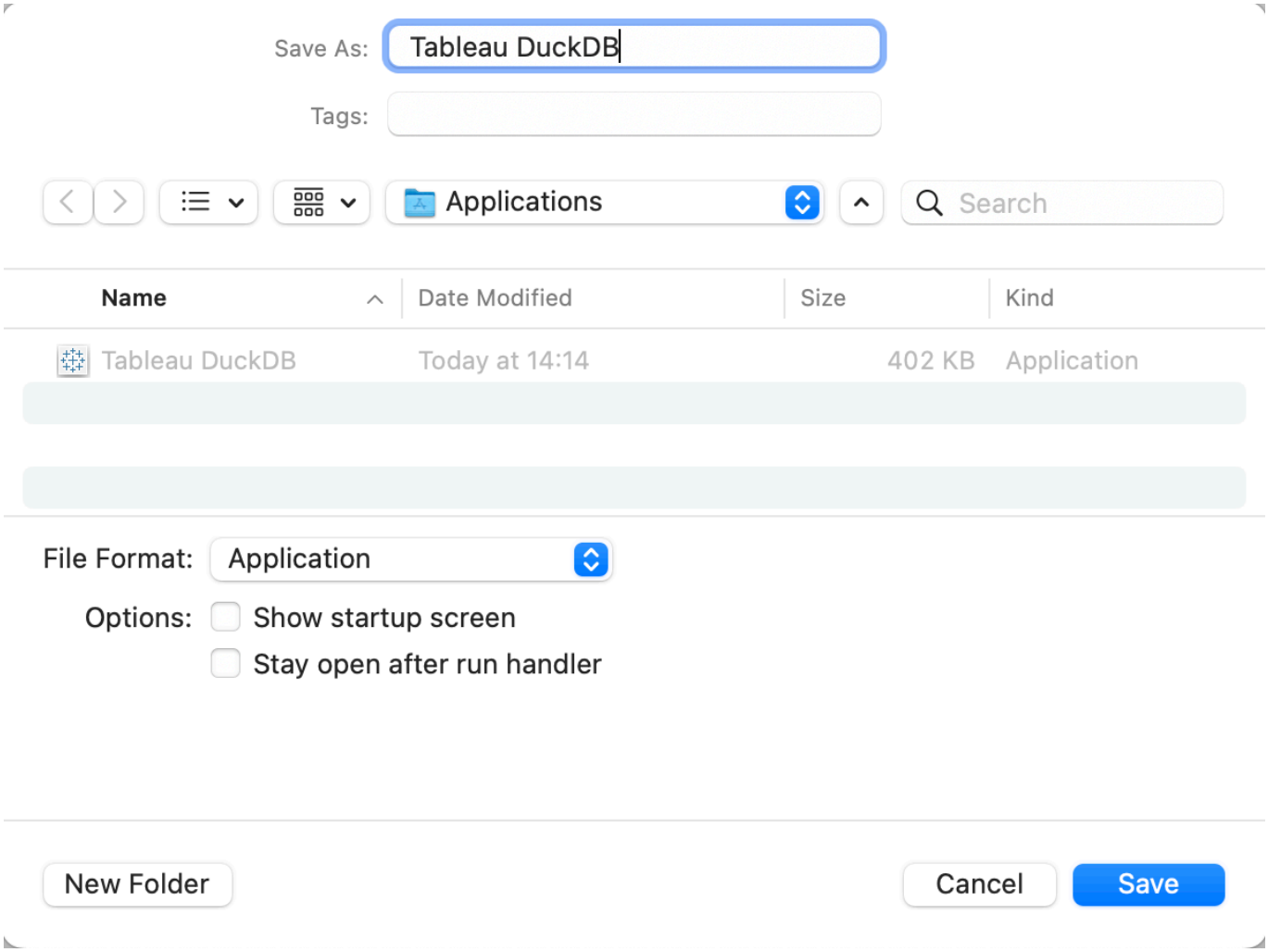
Copy the Taco file to the `/Users/[User]/Documents/My Tableau Repository/Connectors` folder. Then launch Tableau Desktop from the Terminal with the command line argument to disable signature validation:

```
/Applications/Tableau\ Desktop\ <year>.<quarter>.app/Contents/MacOS/Tableau
-DDisableVerifyConnectorPluginSignature=true
```

You can also package this up with AppleScript by using the following script:

```
do shell script "\"/Applications/Tableau Desktop 2023.2.app/Contents/MacOS/Tableau\"
-DDisableVerifyConnectorPluginSignature=true"
quit
```

Create this file with the Script Editor (located in `/Applications/Utilities`) and save it as a packaged application:



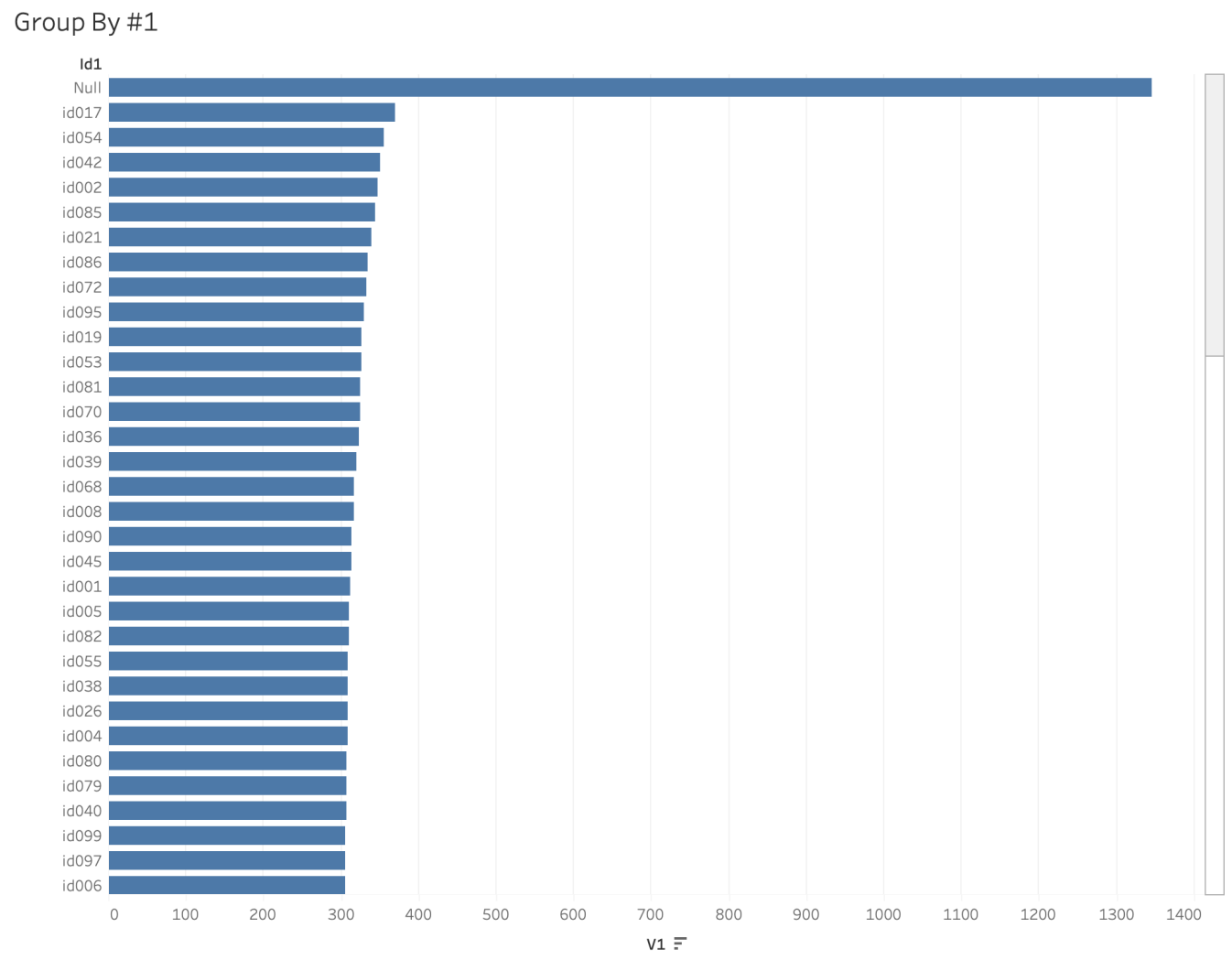
You can then double-click it to launch Tableau. You will need to change the application name in the script when you get upgrades.

Windows Desktop

Copy the Taco file to the `C:\Users\[Windows User]\Documents\My Tableau Repository\Connectors` directory. Then launch Tableau Desktop from a shell with the `-DDisableVerifyConnectorPluginSignature=true` argument to disable signature validation.

Output

Once loaded, you can run queries against your data! Here is the result of the first H2O.ai benchmark query from the Parquet test file:



CLI Charting with YouPlot

DuckDB can be used with CLI graphing tools to quickly pipe input to stdout to graph your data in one line.

YouPlot is a Ruby-based CLI tool for drawing visually pleasing plots on the terminal. It can accept input from other programs by piping data from `stdin`. It takes tab-separated (or delimiter of your choice) data and can easily generate various types of plots including bar, line, histogram and scatter.

With DuckDB, you can write to the console (`stdout`) by using the `TO '/dev/stdout'` command. And you can also write comma-separated values by using `WITH (FORMAT csv, HEADER)`.

Installing YouPlot

Installation instructions for YouPlot can be found on the main YouPlot repository. If you're on a Mac, you can use:

```
brew install youplot
```

Run `uplot --help` to ensure you've installed it successfully!

Piping DuckDB Queries to stdout

By combining the `COPY...TO` function with a CSV output file, data can be read from any format supported by DuckDB and piped to YouPlot. There are three important steps to doing this.

1. As an example, this is how to read all data from `input.json`:

   ```
   duckdb -s "SELECT * FROM read_json_auto('input.json')"
   ```

2. To prepare the data for YouPlot, write a simple aggregate:

   ```
   duckdb -s "SELECT date, sum(purchases) AS total_purchases FROM read_json_auto('input.json') GROUP
   BY 1 ORDER BY 2 DESC LIMIT 10"
   ```

3. Finally, wrap the `SELECT` in the `COPY ... TO` function with an output location of `/dev/stdout`.

   The syntax looks like this:

   ```
   COPY (<query>) TO '/dev/stdout' WITH (FORMAT csv, HEADER);
   ```

   The full DuckDB command below outputs the query in CSV format with a header:

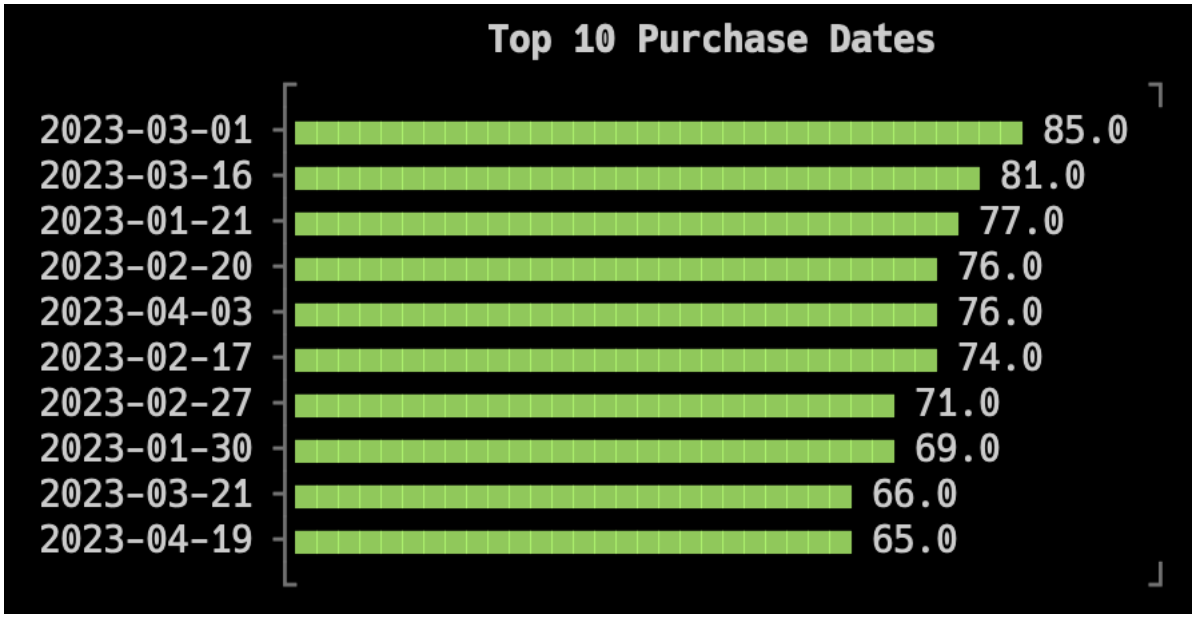
   ```
   duckdb -s "COPY (SELECT date, sum(purchases) AS total_purchases FROM read_json_auto('input.json')
   GROUP BY 1 ORDER BY 2 DESC LIMIT 10) TO '/dev/stdout' WITH (FORMAT csv, HEADER)"
   ```

Connecting DuckDB to YouPlot

Finally, the data can now be piped to YouPlot! Let's assume we have an `input.json` file with dates and number of purchases made by somebody on that date. Using the query above, we'll pipe the data to the `uplot` command to draw a plot of the Top 10 Purchase Dates

```
duckdb -s "COPY (SELECT date, sum(purchases) AS total_purchases FROM read_json_auto('input.json') GROUP
BY 1 ORDER BY 2 DESC LIMIT 10) TO '/dev/stdout' WITH (FORMAT csv, HEADER)" \
     | uplot bar -d, -H -t "Top 10 Purchase Dates"
```

This tells `uplot` to draw a bar plot, use a comma-seperated delimiter (`-d,`), that the data has a header (`-H`), and give the plot a title (`-t`).

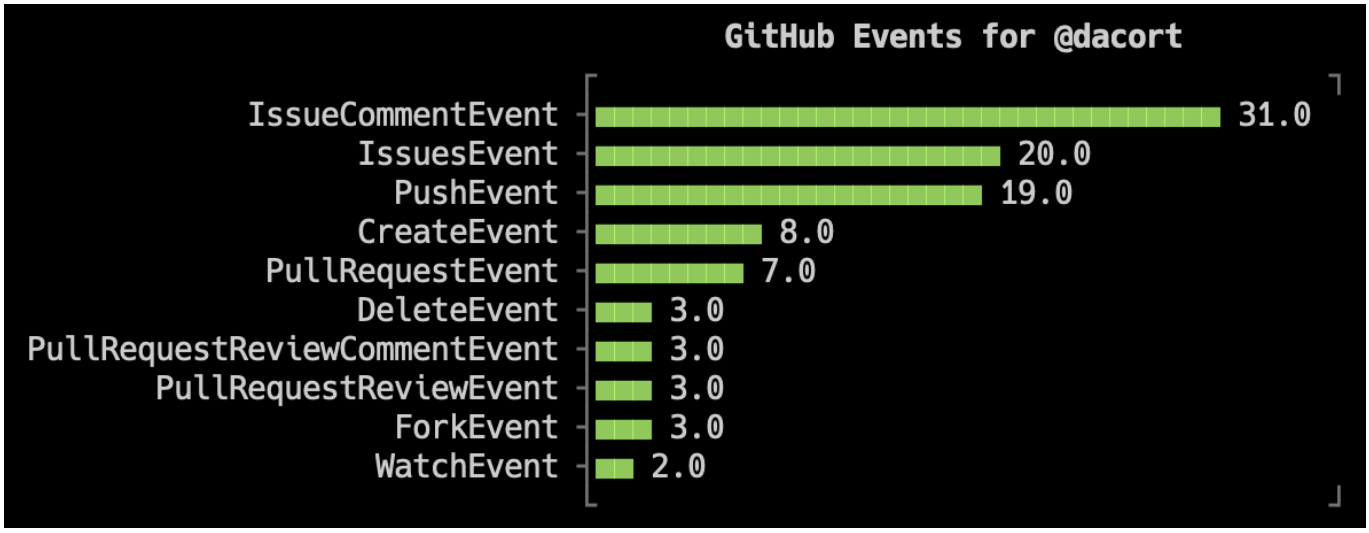


Bonus Round! stdin + stdout

Maybe you're piping some data through `jq`. Maybe you're downloading a JSON file from somewhere. You can also tell DuckDB to read the data from another process by changing the filename to `/dev/stdin`.

Let's combine this with a quick `curl` from GitHub to see what a certain user has been up to lately.

```
curl -sL "https://api.github.com/users/dacort/events?per_page=100" \
    | duckdb -s "COPY (SELECT type, count(*) AS event_count FROM read_json_auto('/dev/stdin') GROUP BY
    1 ORDER BY 2 DESC LIMIT 10) TO '/dev/stdout' WITH (FORMAT csv, HEADER)" \
    | uplot bar -d, -H -t "GitHub Events for @dacort"
```



Database Integration

Database Integration

MySQL Import

To run a query directly on a running MySQL database, the `mysql` extension is required.

Installation and Loading

The extension can be installed using the `INSTALL` SQL command. This only needs to be run once.

```
INSTALL mysql;
```

To load the `mysql` extension for usage, use the `LOAD` SQL command:

```
LOAD mysql;
```

Usage

After the `mysql` extension is installed, you can attach to a MySQL database using the following command:

```
ATTACH 'host=localhost user=root port=0 database=mysqlscanner' AS mysql_db (TYPE mysql, READ_ONLY);
USE mysql_db;
```

The string used by `ATTACH` is a PostgreSQL-style connection string (*not* a MySQL connection string!). It is a list of connection arguments provided in `{key}={value}` format. Below is a list of valid arguments. Any options not provided are replaced by their default values.

Setting	Default
`database`	`NULL`
`host`	`localhost`
`password`	
`port`	`0`
`socket`	`NULL`
`user`	current user

You can directly read and write the MySQL database:

```
CREATE TABLE tbl (id INTEGER, name VARCHAR);
INSERT INTO tbl VALUES (42, 'DuckDB');
```

For a list of supported operations, see the MySQL extension documentation.

PostgreSQL Import

To run a query directly on a running PostgreSQL database, the `postgres` extension is required.

Installation and Loading

The extension can be installed using the `INSTALL` SQL command. This only needs to be run once.

```
INSTALL postgres;
```

To load the `postgres` extension for usage, use the `LOAD` SQL command:

```
LOAD postgres;
```

Usage

After the `postgres` extension is installed, tables can be queried from PostgreSQL using the `postgres_scan` function:

```
-- Scan the table "mytable" from the schema "public" in the database "mydb"
SELECT * FROM postgres_scan('host=localhost port=5432 dbname=mydb', 'public', 'mytable');
```

The first parameter to the `postgres_scan` function is the PostgreSQL connection string, a list of connection arguments provided in `{key}={value}` format. Below is a list of valid arguments.

Name	Description	Default
`host`	Name of host to connect to	`localhost`
`hostaddr`	Host IP address	`localhost`
`port`	Port number	`5432`
`user`	PostgreSQL user name	[OS user name]
`password`	PostgreSQL password	
`dbname`	Database name	[user]
`passfile`	Name of file passwords are stored in	`~/.pgpass`

Alternatively, the entire database can be attached using the `ATTACH` command. This allows you to query all tables stored within the PostgreSQL database as if it was a regular database.

```
-- Attach the PostgreSQL database using the given connection string
ATTACH 'host=localhost port=5432 dbname=mydb' AS test (TYPE postgres);
-- The table "tbl_name" can now be queried as if it is a regular table
SELECT * FROM test.tbl_name;
-- Switch the active database to "test"
USE test;
-- List all tables in the file
SHOW TABLES;
```

For more information see the PostgreSQL extension documentation.

SQLite Import

To run a query directly on a SQLite file, the `sqlite` extension is required.

Installation and Loading

The extension can be installed using the `INSTALL` SQL command. This only needs to be run once.

```
INSTALL sqlite;
```

To load the `sqlite` extension for usage, use the `LOAD` SQL command:

```
LOAD sqlite;
```

Usage

After the SQLite extension is installed, tables can be queried from SQLite using the `sqlite_scan` function:

```
-- Scan the table "tbl_name" from the SQLite file "test.db"
SELECT * FROM sqlite_scan('test.db', 'tbl_name');
```

Alternatively, the entire file can be attached using the `ATTACH` command. This allows you to query all tables stored within a SQLite database file as if they were a regular database.

```
-- Attach the SQLite file "test.db"
ATTACH 'test.db' AS test (TYPE sqlite);
-- The table "tbl_name" can now be queried as if it is a regular table
SELECT * FROM test.tbl_name;
-- Switch the active database to "test"
USE test;
-- List all tables in the file
SHOW TABLES;
```

For more information see the SQLite extension documentation.

File Formats

File Formats

CSV Import

To read data from a CSV file, use the `read_csv` function in the `FROM` clause of a query:

```
SELECT * FROM read_csv('input.csv');
```

Alternatively, you can omit the `read_csv` function and let DuckDB infer it from the extension:

```
SELECT * FROM 'input.csv';
```

To create a new table using the result from a query, use `CREATE TABLE ... AS SELECT` statement:

```
CREATE TABLE new_tbl AS
    SELECT * FROM read_csv('input.csv');
```

We can use DuckDB's optional `FROM`-first syntax to omit `SELECT *`:

```
CREATE TABLE new_tbl AS
    FROM read_csv('input.csv');
```

To load data into an existing table from a query, use `INSERT INTO` from a `SELECT` statement:

```
INSERT INTO tbl
    SELECT * FROM read_csv('input.csv');
```

Alternatively, the `COPY` statement can also be used to load data from a CSV file into an existing table:

```
COPY tbl FROM 'input.csv';
```

For additional options, see the CSV import reference and the `COPY` statement documentation.

CSV Export

To export the data from a table to a CSV file, use the `COPY` statement:

```
COPY tbl TO 'output.csv' (HEADER, DELIMITER ',');
```

The result of queries can also be directly exported to a CSV file:

```
COPY (SELECT * FROM tbl) TO 'output.csv' (HEADER, DELIMITER ',');
```

For additional options, see the `COPY` statement documentation.

Directly Reading Files

DuckDB allows directly reading files via the `read_text` and `read_blob` functions. These functions accept a filename, a list of filenames or a glob pattern, and output the content of each file as a `VARCHAR` or `BLOB`, respectively, as well as additional metadata such as the file size and last modified time.

`read_text`

The `read_text` table function reads from the selected source(s) to a VARCHAR. Each file results in a single row with the `content` field holding the entire content of the respective file.

```
SELECT size, parse_path(filename), content
FROM read_text('test/sql/table_function/files/*.txt');
```

size	parse_path(filename)	content
12	[test, sql, table_function, files, one.txt]	Hello World!
2	[test, sql, table_function, files, three.txt]	42
10	[test, sql, table_function, files, two.txt]	Foo Bar\nFöö Bär

The file content is first validated to be valid UTF-8. If `read_text` attempts to read a file with invalid UTF-8, an error is thrown suggesting to use `read_blob` instead.

`read_blob`

The `read_blob` table function reads from the selected source(s) to a BLOB:

```
SELECT size, content, filename
FROM read_blob('test/sql/table_function/files/*');
```

size	content	filename
178	PK\x03\x04\x0A\x00\x00\x00\x00\x00\xACi=X\x14t\xCE\xC7\x0A…	test/sql/table_function/files/four.blob
12	Hello World!	test/sql/table_function/files/one.txt
2	42	test/sql/table_function/files/three.txt
10	F\xC3\xB6\xC3\xB6 B\xC3\xA4r	test/sql/table_function/files/two.txt

Schema

The schemas of the tables returned by `read_text` and `read_blob` are identical:

```
DESCRIBE FROM read_text('README.md');
```

column_name	column_type	null	key	default	extra
filename	VARCHAR	YES	NULL	NULL	NULL
content	VARCHAR	YES	NULL	NULL	NULL
size	BIGINT	YES	NULL	NULL	NULL
last_modified	TIMESTAMP	YES	NULL	NULL	NULL

Handling Missing Metadata

In cases where the underlying filesystem is unable to provide some of this data due (e.g., because HTTPFS can't always return a valid timestamp), the cell is set to `NULL` instead.

Support for Projection Pushdown

The table functions also utilize projection pushdown to avoid computing properties unnecessarily. So you could e.g., use this to glob a directory full of huge files to get the file size in the size column, as long as you omit the content column the data won't be read into DuckDB.

Excel Import

DuckDB supports reading Excel `.xlsx` files, however, `.xls` files are not supported.

Importing Excel Sheets

Use the `read_xlsx` function in the `FROM` clause of a query:

```
SELECT * FROM read_xlsx('test_excel.xlsx');
```

Alternatively, you can omit the `read_xlsx` function and let DuckDB infer it from the extension:

```
SELECT * FROM 'test_excel.xlsx';
```

However, if you want to be able to pass options to control the import behavior, you should use the `read_xlsx` function.

One such option is the `sheet` parameter, which allows specifying the name of the Excel worksheet:

```
SELECT * FROM read_xlsx('test_excel.xlsx', sheet = 'Sheet1');
```

By default, the first sheet is loaded if no sheet is specified.

Importing a Specific Range

To select a specific range of cells, use the `range` parameter with a string in the format `A1:B2`, where `A1` is the top-left cell and `B2` is the bottom-right cell:

```
SELECT * FROM read_xlsx('test_excel.xlsx', range = 'A1:B2');
```

This can also be used to, e.g., skip the first 5 of rows:

```
SELECT * FROM read_xlsx('test_excel.xlsx', range = 'A5:Z');
```

Or skip the first 5 columns

```
SELECT * FROM read_xlsx('test_excel.xlsx', range = 'E:Z');
```

If no range parameter is provided, the range is automatically inferred as the rectangular region of cells between the first row of consecutive non-empty cells and the first empty row spanning the same columns.

By default, if no range is provided DuckDB will stop reading the Excel file at when encountering an empty row. But when a range is provided, the default is to read until the end of the range. This behavior can be controlled with the `stop_at_empty` parameter:

```
-- Read the first 100 rows, or until the first empty row, whichever comes first
SELECT * FROM read_xlsx('test_excel.xlsx', range = '1:100', stop_at_empty = true);

-- Always read the whole sheet, even if it contains empty rows
SELECT * FROM read_xlsx('test_excel.xlsx', stop_at_empty = false);
```

Creating a New Table

To create a new table using the result from a query, use `CREATE TABLE ... AS` from a `SELECT` statement:

```
CREATE TABLE new_tbl AS
    SELECT * FROM read_xlsx('test_excel.xlsx', sheet = 'Sheet1');
```

Loading to an Existing Table

To load data into an existing table from a query, use `INSERT INTO` from a `SELECT` statement:

```
INSERT INTO tbl
    SELECT * FROM read_xlsx('test_excel.xlsx', sheet = 'Sheet1');
```

Alternatively, you can use the COPY statement with the `XLSX` format option to import an Excel file into an existing table:

```
COPY tbl FROM 'test_excel.xlsx' (FORMAT xlsx, SHEET 'Sheet1');
```

When using the COPY statement to load an Excel file into an existing table, the types of the columns in the target table will be used to coerce the types of the cells in the Excel sheet.

Importing a Sheet with/without a Header

To treat the first row as containing the names of the resulting columns, use the `header` parameter:

```
SELECT * FROM read_xlsx('test_excel.xlsx', header = true);
```

By default, the first row is treated as a header if all the cells in the first row (within the inferred or supplied range) are non-empty strings. To disable this behavior, set `header` to `false`.

Detecting Types

When not importing into an existing table, DuckDB will attempt to infer the types of the columns in the Excel sheet based on their contents and/or "number format".

- `TIMESTAMP`, `TIME`, `DATE` and `BOOLEAN` types are inferred when possible based on the "number format" applied to the cell.
- Text cells containing `TRUE` and `FALSE` are inferred as `BOOLEAN`.
- Empty cells are considered to be of type `DOUBLE` by default.
- Otherwise cells are inferred as `VARCHAR` or `DOUBLE` based on their contents.

This behavior can be adjusted in the following ways.

To treat all empty cells as `VARCHAR` instead of `DOUBLE`, set `empty_as_varchar` to `true`:

```
SELECT * FROM read_xlsx('test_excel.xlsx', empty_as_varchar = true);
```

To disable type inference completely and treat all cells as `VARCHAR`, set `all_varchar` to `true`:

```
SELECT * FROM read_xlsx('test_excel.xlsx', all_varchar = true);
```

Additionally, if the `ignore_errors` parameter is set to `true`, DuckDB will silently replace cells that can't be cast to the corresponding inferred column type with `NULL`'s.

```
SELECT * FROM read_xlsx('test_excel.xlsx', ignore_errors = true);
```

See Also

DuckDB can also export Excel files. For additional details on Excel support, see the excel extension page.

Excel Export

DuckDB supports exporting data to Excel `.xlsx` files. However, `.xls` files are not supported.

Exporting Excel Sheets

To export a table to an Excel file, use the COPY statement with the `FORMAT xlsx` option:

```
COPY tbl TO 'output.xlsx' WITH (FORMAT xlsx);
```

The result of a query can also be directly exported to an Excel file:

```
COPY (SELECT * FROM tbl) TO 'output.xlsx' WITH (FORMAT xlsx);
```

To write the column names as the first row in the Excel file, use the HEADER option:

```
COPY tbl TO 'output.xlsx' WITH (FORMAT xlsx, HEADER true);
```

To name the worksheet in the resulting Excel file, use the SHEET option:

```
COPY tbl TO 'output.xlsx' WITH (FORMAT xlsx, SHEET 'Sheet1');
```

Type Conversions

Because Excel only really supports storing numbers or strings – the equivalent of VARCHAR and DOUBLE, the following type conversions are automatically applied when writing XLSX files:

- Numeric types are cast to DOUBLE.
- Temporal types (TIMESTAMP, DATE, TIME, etc.) are converted to excel "serial" numbers, that is the number of days since 1900-01-01 for dates and the fraction of a day for times. These are then styled with a "number format" so that they appear as dates or times when opened in Excel.
- TIMESTAMP_TZ and TIME_TZ are cast to UTC TIMESTAMP and TIME respectively, with the timezone information being lost.
- BOOLEANs are converted to 1 and 0, with a "number format" applied to make them appear as TRUE and FALSE in Excel.
- All other types are cast to VARCHAR and then written as text cells.

But you can of course also explicitly cast columns to a different type before exporting them to Excel:

```
COPY (SELECT CAST(a AS VARCHAR), b FROM tbl) TO 'output.xlsx' WITH (FORMAT xlsx);
```

See Also

DuckDB can also import Excel files. For additional details on Excel support, see the excel extension page.

JSON Import

To read data from a JSON file, use the `read_json_auto` function in the `FROM` clause of a query:

```
SELECT *
FROM read_json_auto('input.json');
```

To create a new table using the result from a query, use `CREATE TABLE AS` from a `SELECT` statement:

```
CREATE TABLE new_tbl AS
    SELECT *
    FROM read_json_auto('input.json');
```

To load data into an existing table from a query, use `INSERT INTO` from a `SELECT` statement:

```
INSERT INTO tbl
    SELECT *
    FROM read_json_auto('input.json');
```

Alternatively, the `COPY` statement can also be used to load data from a JSON file into an existing table:

```
COPY tbl FROM 'input.json';
```

For additional options, see the JSON Loading reference and the `COPY` statement documentation.

JSON Export

To export the data from a table to a JSON file, use the `COPY` statement:

```
COPY tbl TO 'output.json';
```

The result of queries can also be directly exported to a JSON file:

```
COPY (SELECT * FROM range(3) tbl(n)) TO 'output.json';
```

```
{"n":0}
{"n":1}
{"n":2}
```

The JSON export writes JSON lines by default, standardized as Newline-delimited JSON. The `ARRAY` option can be used to write a single JSON array object instead.

```
COPY (SELECT * FROM range(3) tbl(n)) TO 'output.json' (ARRAY);
```

```
[
        {"n":0},
        {"n":1},
        {"n":2}
]
```

For additional options, see the `COPY` statement documentation.

Parquet Import

To read data from a Parquet file, use the `read_parquet` function in the `FROM` clause of a query:

```
SELECT * FROM read_parquet('input.parquet');
```

Alternatively, you can omit the `read_parquet` function and let DuckDB infer it from the extension:

```
SELECT * FROM 'input.parquet';
```

To create a new table using the result from a query, use `CREATE TABLE ... AS SELECT` statement:

```
CREATE TABLE new_tbl AS
    SELECT * FROM read_parquet('input.parquet');
```

To load data into an existing table from a query, use `INSERT INTO` from a `SELECT` statement:

```
INSERT INTO tbl
    SELECT * FROM read_parquet('input.parquet');
```

Alternatively, the COPY statement can also be used to load data from a Parquet file into an existing table:

```
COPY tbl FROM 'input.parquet' (FORMAT parquet);
```

Adjusting the Schema on the Fly

Yo can load a Parquet file into a slightly different schema (e.g., different number of columns, more relaxed types) using the following trick.

Suppose we have a Parquet file with two columns, `c1` and `c2`:

```
COPY (FROM (VALUES (42, 43)) t(c1, c2))
TO 'f.parquet';
```

If we want to add another column `c3` that is not present in the file, we can run:

```
FROM (VALUES(NULL::VARCHAR, NULL, NULL)) t(c1, c2, c3)
WHERE false
UNION ALL BY NAME
FROM 'f.parquet';
```

The first `FROM` clause generates an empty tables with *three* columns where `c1` is a `VARCHAR`. Then, we use `UNION ALL BY NAME` to union the Parquet file. The result here is:

c1 varchar	c2 int32	c3 int32
42	43	NULL

For additional options, see the Parquet loading reference.

Parquet Export

To export the data from a table to a Parquet file, use the COPY statement:

```
COPY tbl TO 'output.parquet' (FORMAT parquet);
```

The result of queries can also be directly exported to a Parquet file:

```
COPY (SELECT * FROM tbl) TO 'output.parquet' (FORMAT parquet);
```

The flags for setting compression, row group size, etc. are listed in the Reading and Writing Parquet files page.

Querying Parquet Files

To run a query directly on a Parquet file, use the `read_parquet` function in the `FROM` clause of a query.

```
SELECT * FROM read_parquet('input.parquet');
```

The Parquet file will be processed in parallel. Filters will be automatically pushed down into the Parquet scan, and only the relevant columns will be read automatically.

For more information see the blog post "Querying Parquet with Precision using DuckDB".

File Access with the file: Protocol

DuckDB supports using the `file:` protocol. It currently supports the following formats:

- `file:/some/path` (host omitted completely)
- `file:///some/path` (empty host)
- `file://localhost/some/path` (`localhost` as host)

Note that the following formats are *not* supported because they are non-standard:

- `file:some/relative/path` (relative path)
- `file://some/path` (double-slash path)

Additionally, the `file:` protocol currently does not support remote (non-localhost) hosts.

Network and Cloud Storage

Network and Cloud Storage

HTTP Parquet Import

To load a Parquet file over HTTP(S), the `httpfs` extension is required. This can be installed using the `INSTALL` SQL command. This only needs to be run once.

```
INSTALL httpfs;
```

To load the `httpfs` extension for usage, use the `LOAD` SQL command:

```
LOAD httpfs;
```

After the `httpfs` extension is set up, Parquet files can be read over `http(s)`:

```
SELECT * FROM read_parquet('https://<domain>/path/to/file.parquet');
```

For example:

```
SELECT * FROM read_parquet('https://duckdb.org/data/prices.parquet');
```

The function `read_parquet` can be omitted if the URL ends with `.parquet`:

```
SELECT * FROM read_parquet('https://duckdb.org/data/holdings.parquet');
```

Moreover, the `read_parquet` function itself can also be omitted thanks to DuckDB's replacement scan mechanism:

```
SELECT * FROM 'https://duckdb.org/data/holdings.parquet';
```

S3 Parquet Import

Prerequisites

To load a Parquet file from S3, the `httpfs` extension is required. This can be installed using the `INSTALL` SQL command. This only needs to be run once.

```
INSTALL httpfs;
```

To load the `httpfs` extension for usage, use the `LOAD` SQL command:

```
LOAD httpfs;
```

Credentials and Configuration

After loading the `httpfs` extension, set up the credentials and S3 region to read data:

```
CREATE SECRET (
    TYPE s3,
    KEY_ID '<AKIAIOSFODNN7EXAMPLE>',
    SECRET '<wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY>',
    REGION '<us-east-1>'
);
```

> **Tip.** If you get an IO Error (`Connection error for HTTP HEAD`), configure the endpoint explicitly via `ENDPOINT 's3.<your_region>.amazonaws.com'`{:.language-sql .highlight}.

Alternatively, use the aws extension to retrieve the credentials automatically:

```
CREATE SECRET (
    TYPE s3,
    PROVIDER credential_chain
);
```

Querying

After the `httpfs` extension is set up and the S3 configuration is set correctly, Parquet files can be read from S3 using the following command:

```
SELECT * FROM read_parquet('s3://<bucket>/<file>');
```

Google Cloud Storage (GCS) and Cloudflare R2

DuckDB can also handle Google Cloud Storage (GCS) and Cloudflare R2 via the S3 API. See the relevant guides for details.

S3 Parquet Export

To write a Parquet file to S3, the `httpfs` extension is required. This can be installed using the `INSTALL` SQL command. This only needs to be run once.

```
INSTALL httpfs;
```

To load the `httpfs` extension for usage, use the `LOAD` SQL command:

```
LOAD httpfs;
```

After loading the `httpfs` extension, set up the credentials to write data. Note that the `region` parameter should match the region of the bucket you want to access.

```
CREATE SECRET (
    TYPE s3,
    KEY_ID '<AKIAIOSFODNN7EXAMPLE>',
    SECRET '<wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY>',
    REGION '<us-east-1>'
);
```

> **Tip.** If you get an IO Error (`Connection error for HTTP HEAD`), configure the endpoint explicitly via `ENDPOINT 's3.<your_region>.amazonaws.com'`{:.language-sql .highlight}.

Alternatively, use the aws extension to retrieve the credentials automatically:

```
CREATE SECRET (
    TYPE s3,
    PROVIDER credential_chain
);
```

After the `httpfs` extension is set up and the S3 credentials are correctly configured, Parquet files can be written to S3 using the following command:

```
COPY <table_name> TO 's3://<s3-bucket>/<filename>.parquet';
```

Similarly, Google Cloud Storage (GCS) is supported through the Interoperability API. You need to create HMAC keys and provide the credentials as follows:

```
CREATE SECRET (
    TYPE gcs,
    KEY_ID '<AKIAIOSFODNN7EXAMPLE>',
    SECRET '<wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY>'
);
```

After setting up the GCS credentials, you can export using:

```
COPY <table_name> TO 'gs://<gcs_bucket>/<filename>.parquet';
```

S3 Iceberg Import

Prerequisites

To load an Iceberg file from S3, both the `httpfs` and `iceberg` extensions are required. They can be installed using the `INSTALL` SQL command. The extensions only need to be installed once.

```
INSTALL httpfs;
INSTALL iceberg;
```

To load the extensions for usage, use the `LOAD` command:

```
LOAD httpfs;
LOAD iceberg;
```

Credentials

After loading the extensions, set up the credentials and S3 region to read data. You may either use an access key and secret, or a token.

```
CREATE SECRET (
    TYPE s3,
    KEY_ID '<AKIAIOSFODNN7EXAMPLE>',
    SECRET '<wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY>',
    REGION '<us-east-1>'
);
```

Alternatively, use the aws extension to retrieve the credentials automatically:

```
CREATE SECRET (
    TYPE s3,
    PROVIDER credential_chain
);
```

Loading Iceberg Tables from S3

After the extensions are set up and the S3 credentials are correctly configured, Iceberg table can be read from S3 using the following command:

```
SELECT *
FROM iceberg_scan('s3://<bucket>/<iceberg_table_folder>/metadata/<id>.metadata.json');
```

Note that you need to link directly to the manifest file. Otherwise you'll get an error like this:

```
IO Error:
Cannot open file "s3://bucket/iceberg_table_folder/metadata/version-hint.text": No such file or
directory
```

S3 Express One

In late 2023, AWS announced the S3 Express One Zone, a high-speed variant of traditional S3 buckets. DuckDB can read S3 Express One buckets using the `httpfs` extension.

Credentials and Configuration

The configuration of S3 Express One buckets is similar to regular S3 buckets with one exception: we have to specify the endpoint according to the following pattern:

```
s3express-<availability_zone>.<region>.amazonaws.com
```

where the `<availability_zone>`{:.language-sql .highlight} (e.g., `use-az5`) can be obtained from the S3 Express One bucket's configuration page and the `<region>`{:.language-sql .highlight} is the AWS region (e.g., `us-east-1`).

For example, to allow DuckDB to use an S3 Express One bucket, configure the Secrets manager as follows:

```
CREATE SECRET (
    TYPE s3,
    KEY_ID '<AKIAIOSFODNN7EXAMPLE>',
    SECRET '<wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY>',
    REGION '<us-east-1>',
    ENDPOINT 's3express-<use1-az5>.<us-east-1>.amazonaws.com'
);
```

Instance Location

For best performance, make sure that the EC2 instance is in the same availability zone as the S3 Express One bucket you are querying. To determine the mapping between zone names and zone IDs, use the `aws ec2 describe-availability-zones` command.

- Zone name to zone ID mapping:

  ```
  aws ec2 describe-availability-zones --output json \
      | jq -r '.AvailabilityZones[] | select(.ZoneName == "us-east-1f") | .ZoneId'
  ```

  ```
  use1-az5
  ```

- Zone ID to zone name mapping:

  ```
  aws ec2 describe-availability-zones --output json \
      | jq -r '.AvailabilityZones[] | select(.ZoneId == "use1-az5") | .ZoneName'
  ```

  ```
  us-east-1f
  ```

Querying

You can query the S3 Express One bucket as any other S3 bucket:

```
SELECT *
FROM 's3://express-bucket-name--use1-az5--x-s3/my-file.parquet';
```

Performance

We ran two experiments on a `c7gd.12xlarge` instance using the LDBC SF300 Comments `creationDate` Parquet file file (also used in the microbenchmarks of the performance guide).

Experiment	File size	Runtime
Loading only from Parquet	4.1 GB	3.5 s
Creating local table from Parquet	4.1 GB	5.1 s

The "loading only" variant is running the load as part of an `EXPLAIN ANALYZE` statement to measure the runtime without account creating a local table, while the "creating local table" variant uses `CREATE TABLE ... AS SELECT` to create a persistent table on the local disk.

Google Cloud Storage Import

Prerequisites

The Google Cloud Storage (GCS) can be used via the `httpfs` extension. This can be installed with the `INSTALL httpfs` SQL command. This only needs to be run once.

Credentials and Configuration

You need to create HMAC keys and declare them:

```
CREATE SECRET (
    TYPE gcs,
    KEY_ID '<AKIAIOSFODNN7EXAMPLE>',
    SECRET '<wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY>'
);
```

Querying

After setting up the GCS credentials, you can query the GCS data using:

```
SELECT *
FROM read_parquet('gs://<gcs_bucket>/<file.parquet>');
```

Attaching to a Database

You can attach to a database file in read-only mode:

```
LOAD httpfs;
ATTACH 'gs://<gcs_bucket>/<file.duckdb>' AS <duckdb_database> (READ_ONLY);
```

> Databases in Google Cloud Storage can only be attached in read-only mode.

Cloudflare R2 Import

Prerequisites

For Cloudflare R2, the S3 Compatibility API allows you to use DuckDB's S3 support to read and write from R2 buckets.

This requires the `httpfs` extension, which can be installed using the `INSTALL` SQL command. This only needs to be run once.

Credentials and Configuration

You will need to generate an S3 auth token and create an R2 secret in DuckDB:

```
CREATE SECRET (
    TYPE r2,
    KEY_ID '<AKIAIOSFODNN7EXAMPLE>',
    SECRET '<wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY>',
    ACCOUNT_ID '<your-33-character-hexadecimal-account-ID>'
);
```

Querying

After setting up the R2 credentials, you can query the R2 data using DuckDB's built-in methods, such as `read_csv` or `read_parquet`:

```
SELECT * FROM read_parquet('r2://<r2-bucket-name>/<file>');
```

Attach to a DuckDB Database over HTTPS or S3

You can establish a read-only connection to a DuckDB instance via HTTPS or the S3 API.

Prerequisites

This guide requires the `httpfs` extension, which can be installed using the `INSTALL httpfs` SQL command. This only needs to be run once.

Attaching to a Database over HTTPS

To connect to a DuckDB database via HTTPS, use the ATTACH statement as follows:

```
ATTACH 'https://blobs.duckdb.org/databases/stations.duckdb' AS stations_db;
```

> Since DuckDB version 1.1, the ATTACH statement creates a read-only connection to HTTP endpoints. In prior versions, it is necessary to use the READ_ONLY flag.

Then, the database can be queried using:

```
SELECT count(*) AS num_stations
FROM stations_db.stations;
```

num_stations
578

Attaching to a Database over the S3 API

To connect to a DuckDB database via the S3 API, configure the authentication for your bucket (if required). Then, use the ATTACH statement as follows:

```
ATTACH 's3://duckdb-blobs/databases/stations.duckdb' AS stations_db;
```

> Since DuckDB version 1.1, the ATTACH statement creates a read-only connection to HTTP endpoints. In prior versions, it is necessary to use the READ_ONLY flag.

The database can be queried using:

```
SELECT count(*) AS num_stations
FROM stations_db.stations;
```

num_stations
578

> Connecting to S3-compatible APIs such as the Google Cloud Storage (`gs://`) is also supported.

Limitations

- Only read-only connections are allowed, writing the database via the HTTPS protocol or the S3 API is not possible.

Fastly Object Storage Import

Prerequisites

For Fastly Object Storage, the S3 Compatibility API allows you to use DuckDB's S3 support to read and write from Fastly buckets.

This requires the `httpfs` extension, which can be installed using the `INSTALL` SQL command. This only needs to be run once.

Credentials and Configuration

You will need to generate an S3 auth token and create an S3 secret in DuckDB:

```
CREATE SECRET my_secret (
    TYPE s3,
    KEY_ID '<AKIAIOSFODNN7EXAMPLE>',
    SECRET '<wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY>',
    URL_STYLE 'path',
    REGION '<us-east>',
    ENDPOINT '<us-east>.object.fastlystorage.app' -- see note below
);
```

- The `ENDPOINT` needs to point to the Fastly endpoint for the region you want to use (e.g `eu-central.object.fastlystorage.app`).
- `REGION` must use the same region mentioned in `ENDPOINT`.
- `URL_STYLE` needs to use `path`.

Querying

After setting up the Fastly Object Storage credentials, you can query the data there using DuckDB's built-in methods, such as `read_csv` or `read_parquet`:

```
SELECT * FROM 's3://<fastly-bucket-name>/(file).csv';
SELECT * FROM read_parquet('s3://<fastly-bucket-name>/<file>.parquet');
```

Meta Queries

Describe

Describing a Table

In order to view the schema of a table, use the `DESCRIBE` statement (or its aliases `DESC` and `SHOW`) followed by the table name.

```
CREATE TABLE tbl (i INTEGER PRIMARY KEY, j VARCHAR);
DESCRIBE tbl;
SHOW tbl; -- equivalent to DESCRIBE tbl;
```

column_name	column_type	null	key	default	extra
i	INTEGER	NO	PRI	NULL	NULL
j	VARCHAR	YES	NULL	NULL	NULL

Describing a Query

In order to view the schema of the result of a query, prepend `DESCRIBE` to a query.

```
DESCRIBE SELECT * FROM tbl;
```

column_name	column_type	null	key	default	extra
i	INTEGER	YES	NULL	NULL	NULL
j	VARCHAR	YES	NULL	NULL	NULL

Note that there are subtle differences: compared to the result when describing a table, nullability (`null`) and key information (`key`) are lost.

Using `DESCRIBE` in a Subquery

`DESCRIBE` can be used a subquery. This allows creating a table from the description, for example:

```
CREATE TABLE tbl_description AS SELECT * FROM (DESCRIBE tbl);
```

Describing Remote Tables

It is possible to describe remote tables via the `httpfs` extension using the `DESCRIBE TABLE` statement. For example:

```
DESCRIBE TABLE 'https://blobs.duckdb.org/data/Star_Trek-Season_1.csv';
```

column_name	column_type	null	key	default	extra
season_num	BIGINT	YES	NULL	NULL	NULL
episode_num	BIGINT	YES	NULL	NULL	NULL
aired_date	DATE	YES	NULL	NULL	NULL
cnt_kirk_hookups	BIGINT	YES	NULL	NULL	NULL
cnt_downed_redshirts	BIGINT	YES	NULL	NULL	NULL
bool_aliens_almost_took_over_planet	BIGINT	YES	NULL	NULL	NULL
bool_aliens_almost_took_over_enterprise	BIGINT	YES	NULL	NULL	NULL
cnt_vulcan_nerve_pinch	BIGINT	YES	NULL	NULL	NULL
cnt_warp_speed_orders	BIGINT	YES	NULL	NULL	NULL
highest_warp_speed_issued	BIGINT	YES	NULL	NULL	NULL
bool_hand_phasers_fired	BIGINT	YES	NULL	NULL	NULL
bool_ship_phasers_fired	BIGINT	YES	NULL	NULL	NULL
bool_ship_photon_torpedos_fired	BIGINT	YES	NULL	NULL	NULL
cnt_transporter_pax	BIGINT	YES	NULL	NULL	NULL
cnt_damn_it_jim_quote	BIGINT	YES	NULL	NULL	NULL
cnt_im_givin_her_all_shes_got_quote	BIGINT	YES	NULL	NULL	NULL
cnt_highly_illogical_quote	BIGINT	YES	NULL	NULL	NULL
bool_enterprise_saved_the_day	BIGINT	YES	NULL	NULL	NULL

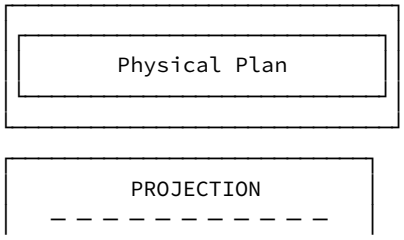
EXPLAIN: Inspect Query Plans

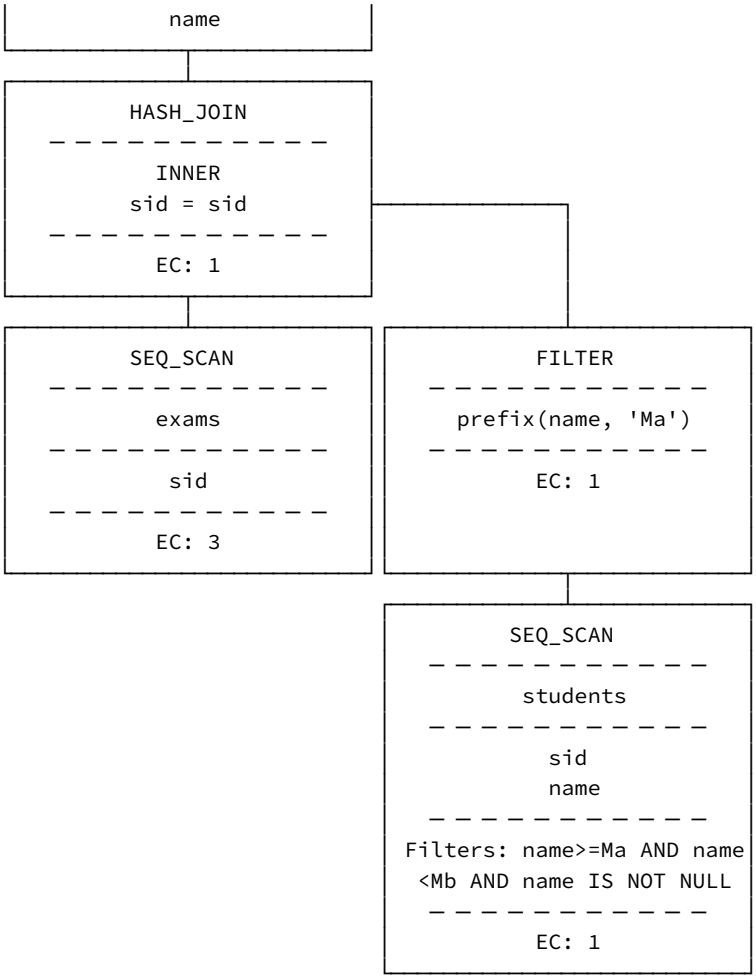
```
EXPLAIN SELECT * FROM tbl;
```

The EXPLAIN statement displays the physical plan, i.e., the query plan that will get executed, and is enabled by prepending the query with EXPLAIN. The physical plan is a tree of operators that are executed in a specific order to produce the result of the query. To generate an efficient physical plan, the query optimizer transforms the existing physical plan into a better physical plan.

To demonstrate, see the below example:

```
CREATE TABLE students (name VARCHAR, sid INTEGER);
CREATE TABLE exams (eid INTEGER, subject VARCHAR, sid INTEGER);
INSERT INTO students VALUES ('Mark', 1), ('Joe', 2), ('Matthew', 3);
INSERT INTO exams VALUES (10, 'Physics', 1), (20, 'Chemistry', 2), (30, 'Literature', 3);

EXPLAIN ANALYZE
    SELECT name
    FROM students
    JOIN exams USING (sid)
    WHERE name LIKE 'Ma%';
```





Note that the query is not actually executed – therefore, we can only see the estimated cardinality (EC) for each operator, which is calculated by using the statistics of the base tables and applying heuristics for each operator.

Additional Explain Settings

The `EXPLAIN` statement supports additional settings that can be used to control the output. The following settings are available:

The default setting. Only shows the physical plan.

```
PRAGMA explain_output = 'physical_only';
```

Shows only the optimized plan.

```
PRAGMA explain_output = 'optimized_only';
```

Shows both the physical and optimized plans.

```
PRAGMA explain_output = 'all';
```

See Also

For more information, see the "Profiling" page.

EXPLAIN ANALYZE: Profile Queries

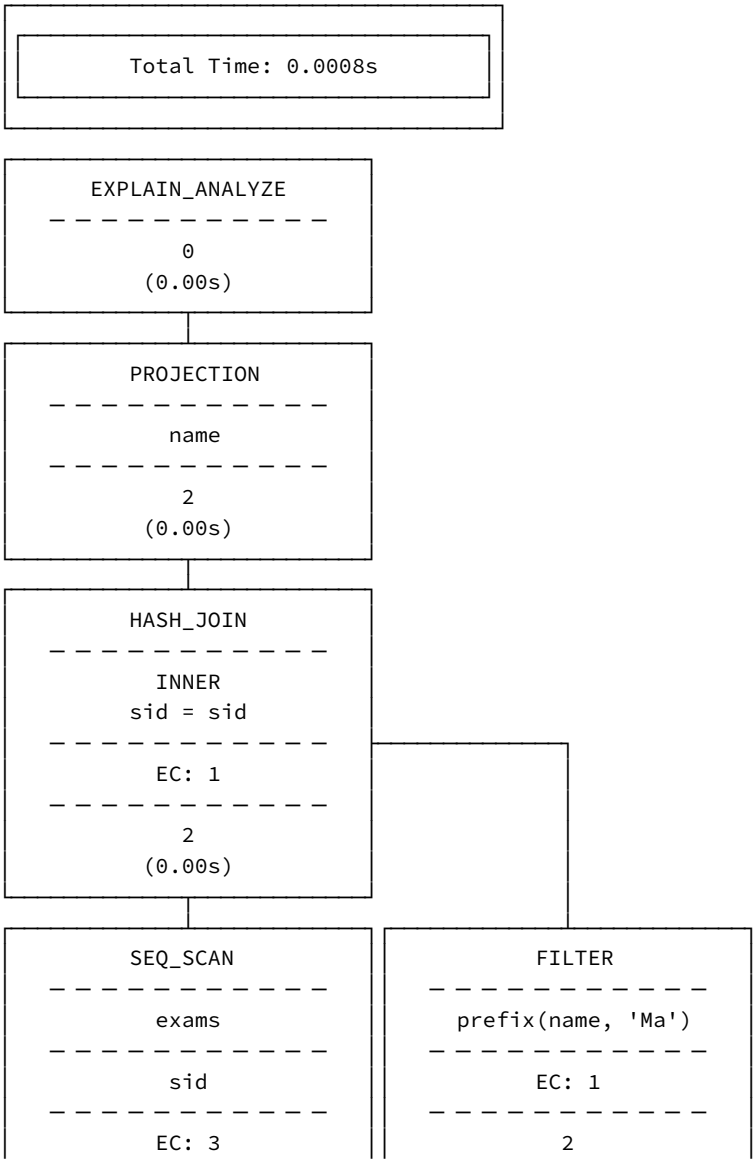
Prepending a query with `EXPLAIN ANALYZE` both pretty-prints the query plan, and executes it, providing run-time performance numbers for every operator, as well as the estimated cardinality (EC) and the actual cardinality.

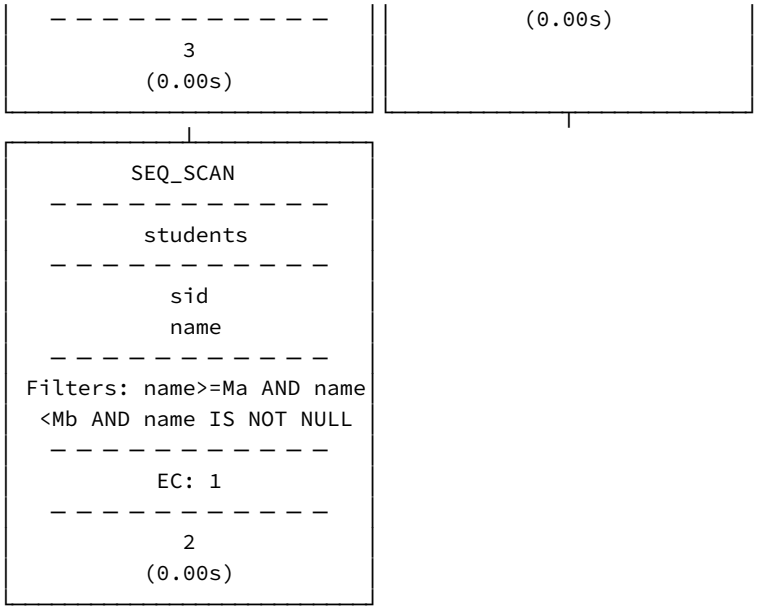
```
EXPLAIN ANALYZE SELECT * FROM tbl;
```

Note that the **cumulative** wall-clock time that is spent on every operator is shown. When multiple threads are processing the query in parallel, the total processing time of the query may be lower than the sum of all the times spent on the individual operators.

Below is an example of running `EXPLAIN ANALYZE` on a query:

```
CREATE TABLE students (name VARCHAR, sid INTEGER);
CREATE TABLE exams (eid INTEGER, subject VARCHAR, sid INTEGER);
INSERT INTO students VALUES ('Mark', 1), ('Joe', 2), ('Matthew', 3);
INSERT INTO exams VALUES (10, 'Physics', 1), (20, 'Chemistry', 2), (30, 'Literature', 3);

EXPLAIN ANALYZE
    SELECT name
    FROM students
    JOIN exams USING (sid)
    WHERE name LIKE 'Ma%';
```





See Also

For more information, see the "Profiling" page.

List Tables

The `SHOW TABLES` command can be used to obtain a list of all tables within the selected schema.

```
CREATE TABLE tbl (i INTEGER);
SHOW TABLES;
```

name
tbl

`SHOW` or `SHOW ALL TABLES` can be used to obtain a list of all tables within **all** attached databases and schemas.

```
CREATE TABLE tbl (i INTEGER);
CREATE SCHEMA s1;
CREATE TABLE s1.tbl (v VARCHAR);
SHOW ALL TABLES;
```

database	schema	table_name	column_names	column_types	temporary
memory	main	tbl	[i]	[INTEGER]	false
memory	s1	tbl	[v]	[VARCHAR]	false

To view the schema of an individual table, use the `DESCRIBE` command.

See Also

The SQL-standard `information_schema` views are also defined. Moreover, DuckDB defines `sqlite_master` and many PostgreSQL system catalog tables for compatibility with SQLite and PostgreSQL respectively.

Summarize

The `SUMMARIZE` command can be used to easily compute a number of aggregates over a table or a query. The `SUMMARIZE` command launches a query that computes a number of aggregates over all columns (`min`, `max`, `approx_unique`, `avg`, `std`, `q25`, `q50`, `q75`, `count`), and return these along the column name, column type, and the percentage of `NULL` values in the column. Note that the quantiles and percentiles are **approximate values**.

Usage

In order to summarize the contents of a table, use `SUMMARIZE` followed by the table name.

```
SUMMARIZE tbl;
```

In order to summarize a query, prepend `SUMMARIZE` to a query.

```
SUMMARIZE SELECT * FROM tbl;
```

Example

Below is an example of `SUMMARIZE` on the `lineitem` table of TPC-H `SF1` table, generated using the `tpch` extension.

```
INSTALL tpch;
LOAD tpch;
CALL dbgen(sf = 1);
```

```
SUMMARIZE lineitem;
```

column_ name	column_ type	min	max	approx_ unique	avg	std	q25	q50	q75	count	null_ percentage
l_ orderkey	INTEGER	1	6000000	1508227	3000279.604204732	1732187.873480351	1519447298986944852	3260012150.0%			
l_partkey	INTEGER	1	200000	202598	100017.98932959	57735.69082650496	49613	99992	150039	60012150.0%	
l_ suppkey	INTEGER	1	10000	10061	5000.602606138	2886.961998730	2514	4999	7500	60012150.0%	
l_ linenumber	INTEGER	1	7	7	3.000575716750	1.6912431403651	2	9328	3	4	60012150.0%
l_ quantity	DECIMAL(15,2)	1.00	50.00	50	25.507967136654	14.426262537016	13	918	26	38	60012150.0%
l_ extendedprice	DECIMAL(15,2)	901.00	104949.50	923139	38255.138484656	23850.43871096	18	2156	36724	55159	60012150.0%
l_ discount	DECIMAL(15,2)	0.00	0.10	11	0.049999430115	0.0316198551080	1	2596	0	0	60012150.0%
l_tax	DECIMAL(15,2)	0.00	0.08	9	0.040013508931	0.0258165517988	0	42728	0	0	60012150.0%

column_name	column_type	min	max	approx_unique	avg	std	q25	q50	q75	count	null_percentage
l_returnflag	VARCHAR	A	R	3	NULL	NULL	NULL	NULL	NULL	600121	50.0%
l_linestatus	VARCHAR	F	O	2	NULL	NULL	NULL	NULL	NULL	600121	50.0%
l_shipdate	DATE	1992-01-02	1998-12-01	2516	NULL	NULL	NULL	NULL	NULL	600121	50.0%
l_commitdate	DATE	1992-01-31	1998-10-31	2460	NULL	NULL	NULL	NULL	NULL	600121	50.0%
l_receiptdate	DATE	1992-01-04	1998-12-31	2549	NULL	NULL	NULL	NULL	NULL	600121	50.0%
l_shipinstruct	VARCHAR	COLLECT COD	TAKE BACK RETURN	4	NULL	NULL	NULL	NULL	NULL	600121	50.0%
l_shipmode	VARCHAR	AIR	TRUCK	7	NULL	NULL	NULL	NULL	NULL	600121	50.0%
l_comment	VARCHAR	Tiresias	zzle? furiously iro	3558599	NULL	NULL	NULL	NULL	NULL	600121	50.0%

Using SUMMARIZE in a Subquery

SUMMARIZE can be used a subquery. This allows creating a table from the summary, for example:

```
CREATE TABLE tbl_summary AS SELECT * FROM (SUMMARIZE tbl);
```

Summarizing Remote Tables

It is possible to summarize remote tables via the httpfs extension using the SUMMARIZE TABLE statement. For example:

```
SUMMARIZE TABLE 'https://blobs.duckdb.org/data/Star_Trek-Season_1.csv';
```

DuckDB Environment

DuckDB provides a number of functions and PRAGMA options to retrieve information on the running DuckDB instance and its environment.

Version

The version() function returns the version number of DuckDB.

```
SELECT version() AS version;
```

version
v1.3.1

Using a PRAGMA:

```
PRAGMA version;
```

library_version	source_id
v1.3.1	2063dda3e6

Platform

The platform information consists of the operating system, system architecture, and, optionally, the compiler. The platform is used when installing extensions. To retrieve the platform, use the following PRAGMA:

```
PRAGMA platform;
```

On macOS, running on Apple Silicon architecture, the result is:

platform
osx_arm64

On Windows, running on an AMD64 architecture, the platform is `windows_amd64`. On Ubuntu Linux, running on the ARM64 architecture, the platform is `linux_arm64`.

Extensions

To get a list of DuckDB extension and their status (e.g., `loaded`, `installed`), use the `duckdb_extensions()` function:

```
SELECT *
FROM duckdb_extensions();
```

Meta Table Functions

DuckDB has the following built-in table functions to obtain metadata about available catalog objects:

- `duckdb_columns()`: columns
- `duckdb_constraints()`: constraints
- `duckdb_databases()`: lists the databases that are accessible from within the current DuckDB process
- `duckdb_dependencies()`: dependencies between objects
- `duckdb_extensions()`: extensions
- `duckdb_functions()`: functions
- `duckdb_indexes()`: secondary indexes
- `duckdb_keywords()`: DuckDB's keywords and reserved words
- `duckdb_optimizers()`: the available optimization rules in the DuckDB instance
- `duckdb_schemas()`: schemas
- `duckdb_sequences()`: sequences
- `duckdb_settings()`: settings
- `duckdb_tables()`: base tables
- `duckdb_temporary_files()`: the temporary files DuckDB has written to disk, to offload data from memory
- `duckdb_types()`: data types
- `duckdb_views()`: views

ODBC

ODBC 101: A Duck Themed Guide to ODBC

What is ODBC?

ODBC which stands for Open Database Connectivity, is a standard that allows different programs to talk to different databases including, of course, DuckDB. This makes it easier to build programs that work with many different databases, which saves time as developers don't have to write custom code to connect to each database. Instead, they can use the standardized ODBC interface, which reduces development time and costs, and programs are easier to maintain. However, ODBC can be slower than other methods of connecting to a database, such as using a native driver, as it adds an extra layer of abstraction between the application and the database. Furthermore, because DuckDB is column-based and ODBC is row-based, there can be some inefficiencies when using ODBC with DuckDB.

> There are links throughout this page to the official Microsoft ODBC documentation, which is a great resource for learning more about ODBC.

General Concepts

- Handles
- Connecting
- Error Handling and Diagnostics
- Buffers and Binding

Handles

A handle is a pointer to a specific ODBC object which is used to interact with the database. There are several different types of handles, each with a different purpose, these are the environment handle, the connection handle, the statement handle, and the descriptor handle. Handles are allocated using the `SQLAllocHandle` which takes as input the type of handle to allocate, and a pointer to the handle, the driver then creates a new handle of the specified type which it returns to the application.

The DuckDB ODBC driver has the following handle types.

Environment

Handle name	Environment
Type name	`SQL_HANDLE_ENV`
:--	:--------
Description	Manages the environment settings for ODBC operations, and provides a global context in which to access data.
Use case	Initializing ODBC, managing driver behavior, resource allocation
Additional information	Must be allocated once per application upon starting, and freed at the end.

Connection

Handle name	Connection
Type name	SQL_HANDLE_DBC
:--	:--------
Description	Represents a connection to a data source. Used to establish, manage, and terminate connections. Defines both the driver and the data source to use within the driver.
Use case	Establishing a connection to a database, managing the connection state
Additional information	Multiple connection handles can be created as needed, allowing simultaneous connections to multiple data sources. *Note:* Allocating a connection handle does not establish a connection, but must be allocated first, and then used once the connection has been established.

Statement

Handle name	Statement
Type name	SQL_HANDLE_STMT
:--	:--------
Description	Handles the execution of SQL statements, as well as the returned result sets.
Use case	Executing SQL queries, fetching result sets, managing statement options.
Additional information	To facilitate the execution of concurrent queries, multiple handles can be allocated per connection.

Descriptor

Handle name	Descriptor
Type name	SQL_HANDLE_DESC
:--	:--------
Description	Describes the attributes of a data structure or parameter, and allows the application to specify the structure of data to be bound/retrieved.
Use case	Describing table structures, result sets, binding columns to application buffers
Additional information	Used in situations where data structures need to be explicitly defined, for example during parameter binding or result set fetching. They are automatically allocated when a statement is allocated, but can also be allocated explicitly.

Connecting

The first step is to connect to the data source so that the application can perform database operations. First the application must allocate an environment handle, and then a connection handle. The connection handle is then used to connect to the data source. There are two functions which can be used to connect to a data source, `SQLDriverConnect` and `SQLConnect`. The former is used to connect to a data source using a connection string, while the latter is used to connect to a data source using a DSN.

Connection String

A connection string is a string which contains the information needed to connect to a data source. It is formatted as a semicolon separated list of key-value pairs, however DuckDB currently only utilizes the DSN and ignores the rest of the parameters.

DSN

A DSN (*Data Source Name*) is a string that identifies a database. It can be a file path, URL, or a database name. For example: `C:\Users\me\duckdb.db` and `DuckDB` are both valid DSNs. More information on DSNs can be found on the "Choosing a Data Source or Driver" page of the SQL Server documentation.

Error Handling and Diagnostics

All functions in ODBC return a code which represents the success or failure of the function. This allows for easy error handling, as the application can simply check the return code of each function call to determine if it was successful. When unsuccessful, the application can then use the `SQLGetDiagRec` function to retrieve the error information. The following table defines the return codes:

Return code	Description
`SQL_SUCCESS`	The function completed successfully
`SQL_SUCCESS_WITH_INFO`	The function completed successfully, but additional information is available, including a warnin
`SQL_ERROR`	The function failed
`SQL_INVALID_HANDLE`	The handle provided was invalid, indicating a programming error, i.e., when a handle is not allocated before it is used, or is the wrong type
`SQL_NO_DATA`	The function completed successfully, but no more data is availabl
`SQL_NEED_DATA`	More data is needed, such as when a parameter data is sent at execution time, or additional connection information is required
`SQL_STILL_EXECUTING`	A function that was asynchronously executed is still executing

Buffers and Binding

A buffer is a block of memory used to store data. Buffers are used to store data retrieved from the database, or to send data to the database. Buffers are allocated by the application, and then bound to a column in a result set, or a parameter in a query, using the `SQLBindCol` and `SQLBindParameter` functions. When the application fetches a row from the result set, or executes a query, the data is stored in the buffer. When the application sends a query to the database, the data in the buffer is sent to the database.

Setting up an Application

The following is a step-by-step guide to setting up an application that uses ODBC to connect to a database, execute a query, and fetch the results in C++.

> To install the driver as well as anything else you will need follow these instructions.

1. Include the SQL Header Files

The first step is to include the SQL header files:

```
#include <sql.h>
#include <sqlext.h>
```

These files contain the definitions of the ODBC functions, as well as the data types used by ODBC. In order to be able to use these header files you have to have the `unixodbc` package installed:

On macOS:

```
brew install unixodbc
```

On Ubuntu and Debian:

```
sudo apt-get install -y unixodbc-dev
```

On Fedora, CentOS, and Red Hat:

```
sudo yum install -y unixODBC-devel
```

Remember to include the header file location in your `CFLAGS`.

For `MAKEFILE`:

```
CFLAGS=-I/usr/local/include
# or
CFLAGS=-/opt/homebrew/Cellar/unixodbc/2.3.11/include
```

For CMAKE:

```
include_directories(/usr/local/include)
# or
include_directories(/opt/homebrew/Cellar/unixodbc/2.3.11/include)
```

You also have to link the library in your CMAKE or `MAKEFILE`. For CMAKE:

```
target_link_libraries(ODBC_application /path/to/duckdb_odbc/libduckdb_odbc.dylib)
```

For `MAKEFILE`:

```
LDLIBS=-L/path/to/duckdb_odbc/libduckdb_odbc.dylib
```

2. Define the ODBC Handles and Connect to the Database

2.a. Connecting with SQLConnect

Then set up the ODBC handles, allocate them, and connect to the database. First the environment handle is allocated, then the environment is set to ODBC version 3, then the connection handle is allocated, and finally the connection is made to the database. The following code snippet shows how to do this:

```
SQLHANDLE env;
SQLHANDLE dbc;

SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &env);

SQLSetEnvAttr(env, SQL_ATTR_ODBC_VERSION, (void*)SQL_OV_ODBC3, 0);

SQLAllocHandle(SQL_HANDLE_DBC, env, &dbc);

std::string dsn = "DSN=duckdbmemory";
SQLConnect(dbc, (SQLCHAR*)dsn.c_str(), SQL_NTS, NULL, 0, NULL, 0);

std::cout << "Connected!" << std::endl;
```

2.b. Connecting with SQLDriverConnect

Alternatively, you can connect to the ODBC driver using `SQLDriverConnect`. `SQLDriverConnect` accepts a connection string in which you can configure the database using any of the available DuckDB configuration options.

```
SQLHANDLE env;
SQLHANDLE dbc;

SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &env);

SQLSetEnvAttr(env, SQL_ATTR_ODBC_VERSION, (void*)SQL_OV_ODBC3, 0);

SQLAllocHandle(SQL_HANDLE_DBC, env, &dbc);

SQLCHAR str[1024];
SQLSMALLINT strl;
std::string dsn = "DSN=DuckDB;access_mode=READ_ONLY"
SQLDriverConnect(dbc, nullptr, (SQLCHAR*)dsn.c_str(), SQL_NTS, str, sizeof(str), &strl, SQL_DRIVER_
COMPLETE)

std::cout << "Connected!" << std::endl;
```

3. Adding a Query

Now that the application is set up, we can add a query to it. First, we need to allocate a statement handle:

```
SQLHANDLE stmt;
SQLAllocHandle(SQL_HANDLE_STMT, dbc, &stmt);
```

Then we can execute a query:

```
SQLExecDirect(stmt, (SQLCHAR*)"SELECT * FROM integers", SQL_NTS);
```

4. Fetching Results

Now that we have executed a query, we can fetch the results. First, we need to bind the columns in the result set to buffers:

```
SQLLEN int_val;
SQLLEN null_val;
SQLBindCol(stmt, 1, SQL_C_SLONG, &int_val, 0, &null_val);
```

Then we can fetch the results:

```
SQLFetch(stmt);
```

5. Go Wild

Now that we have the results, we can do whatever we want with them. For example, we can print them:

```
std::cout << "Value: " << int_val << std::endl;
```

or do any other processing we want. As well as executing more queries and doing any thing else we want to do with the database such as inserting, updating, or deleting data.

6. Free the Handles and Disconnecting

Finally, we need to free the handles and disconnect from the database. First, we need to free the statement handle:

```
SQLFreeHandle(SQL_HANDLE_STMT, stmt);
```

Then we need to disconnect from the database:

```
SQLDisconnect(dbc);
```

And finally, we need to free the connection handle and the environment handle:

```
SQLFreeHandle(SQL_HANDLE_DBC, dbc);
SQLFreeHandle(SQL_HANDLE_ENV, env);
```

Freeing the connection and environment handles can only be done after the connection to the database has been closed. Trying to free them before disconnecting from the database will result in an error.

Sample Application

The following is a sample application that includes a cpp file that connects to the database, executes a query, fetches the results, and prints them. It also disconnects from the database and frees the handles, and includes a function to check the return value of ODBC functions. It also includes a `CMakeLists.txt` file that can be used to build the application.

Sample `.cpp` File

```
#include <iostream>
#include <sql.h>
#include <sqlext.h>

void check_ret(SQLRETURN ret, std::string msg) {
    if (ret != SQL_SUCCESS && ret != SQL_SUCCESS_WITH_INFO) {
        std::cout << ret << ": " << msg << " failed" << std::endl;
        exit(1);
    }
    if (ret == SQL_SUCCESS_WITH_INFO) {
        std::cout << ret << ": " << msg << " succeeded with info" << std::endl;
    }
}

int main() {
    SQLHANDLE env;
    SQLHANDLE dbc;
    SQLRETURN ret;

    ret = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &env);
    check_ret(ret, "SQLAllocHandle(env)");

    ret = SQLSetEnvAttr(env, SQL_ATTR_ODBC_VERSION, (void*)SQL_OV_ODBC3, 0);
    check_ret(ret, "SQLSetEnvAttr");

    ret = SQLAllocHandle(SQL_HANDLE_DBC, env, &dbc);
    check_ret(ret, "SQLAllocHandle(dbc)");

    std::string dsn = "DSN=duckdbmemory";
    ret = SQLConnect(dbc, (SQLCHAR*)dsn.c_str(), SQL_NTS, NULL, 0, NULL, 0);
    check_ret(ret, "SQLConnect");
```

```
    std::cout << "Connected!" << std::endl;

    SQLHANDLE stmt;
    ret = SQLAllocHandle(SQL_HANDLE_STMT, dbc, &stmt);
    check_ret(ret, "SQLAllocHandle(stmt)");

    ret = SQLExecDirect(stmt, (SQLCHAR*)"SELECT * FROM integers", SQL_NTS);
    check_ret(ret, "SQLExecDirect(SELECT * FROM integers)");

    SQLLEN int_val;
    SQLLEN null_val;
    ret = SQLBindCol(stmt, 1, SQL_C_SLONG, &int_val, 0, &null_val);
    check_ret(ret, "SQLBindCol");

    ret = SQLFetch(stmt);
    check_ret(ret, "SQLFetch");

    std::cout << "Value: " << int_val << std::endl;

    ret = SQLFreeHandle(SQL_HANDLE_STMT, stmt);
    check_ret(ret, "SQLFreeHandle(stmt)");

    ret = SQLDisconnect(dbc);
    check_ret(ret, "SQLDisconnect");

    ret = SQLFreeHandle(SQL_HANDLE_DBC, dbc);
    check_ret(ret, "SQLFreeHandle(dbc)");

    ret = SQLFreeHandle(SQL_HANDLE_ENV, env);
    check_ret(ret, "SQLFreeHandle(env)");
}
```

Sample `CMakelists.txt` File

```
cmake_minimum_required(VERSION 3.25)
project(ODBC_Tester_App)

set(CMAKE_CXX_STANDARD 17)
include_directories(/opt/homebrew/Cellar/unixodbc/2.3.11/include)

add_executable(ODBC_Tester_App main.cpp)
target_link_libraries(ODBC_Tester_App /duckdb_odbc/libduckdb_odbc.dylib)
```

Performance

Performance Guide

DuckDB aims to automatically achieve high performance by using well-chosen default configurations and having a forgiving architecture. Of course, there are still opportunities for tuning the system for specific workloads. The Performance Guide's page contain guidelines and tips for achieving good performance when loading and processing data with DuckDB.

The guides include several microbenchmarks. You may find details about these on the Benchmarks page.

Environment

The environment where DuckDB is run has an obvious impact on performance. This page focuses on the effects of the hardware configuration and the operating system used.

Hardware Configuration

CPU

DuckDB works efficiently on both AMD64 (x86_64) and ARM64 (AArch64) CPU architectures.

Memory

> **Best practice.** Aim for 1-4 GB memory per thread.

Minimum Required Memory

As a rule of thumb, DuckDB requires a *minimum* of 125 MB of memory per thread. For example, if you use 8 threads, you need at least 1 GB of memory. If you are working in a memory-constrained environment, consider limiting the number of threads, e.g., by issuing:

```
SET threads = 4;
```

Memory for Ideal Performance

The amount of memory required for ideal performance depends on several factors, including the data set size and the queries to execute. Maybe surprisingly, the *queries* have a larger effect on the memory requirement. Workloads containing large joins over many-to-many tables yield large intermediate datasets and thus require more memory for their evaluation to fully fit into the memory. As an approximation, aggregation-heavy workloads require 1-2 GB memory per thread and join-heavy workloads require 3-4 GB memory per thread.

Larger-than-Memory Workloads

DuckDB can process larger-than-memory workloads by spilling to disk. This is possible thanks to *out-of-core* support for grouping, joining, sorting and windowing operators. Note that larger-than-memory workloads can be processed both in persistent mode and in in-memory mode as DuckDB still spills to disk in both modes.

Local Disk

Disk type. DuckDB's disk-based mode is designed to work best with SSD and NVMe disks. While HDDs are supported, they will result in low performance, especially for write operations.

Disk-based vs. in-memory storage. Counter-intuitively, using a disk-based DuckDB instance can be faster than an in-memory instance due to compression. Read more in the "How to Tune Workloads" page.

File systems. On Linux, DuckDB performs best with the XFS file system but it also performs reasonably well with other file systems such as ext4. On Windows, we recommend using NTFS and avoiding FAT32.

> **Note.** that DuckDB databases have built-in checksums, so integrity checks from the file system are not required to prevent data corruption.

Network-Attached Disks

Cloud disks. DuckDB runs well on network-backed cloud disks such as AWS EBS for both read-only and read-write workloads.

Network-attached storage. Network-attached storage can serve DuckDB for read-only workloads. However, *it is not recommended to run DuckDB in read-write mode on network-attached storage (NAS).* These setups include NFS, network drives such as SMB and Samba. Based on user reports, running read-write workloads on network-attached storage can result in slow and unpredictable performance, as well as spurious errors cased by the underlying file system.

> **Warning.** Avoid running DuckDB in read-write mode on network-attached storage.

> **Best practice.** Fast disks are important if your workload is larger than memory and/or fast data loading is important. Only use network-backed disks if they are reliable (e.g., cloud disks) and guarantee high IO.

Operating System

We recommend using the latest stable version of operating systems: macOS, Windows, and Linux are all well-tested and DuckDB can run on them with high performance.

Linux

DuckDB runs on all mainstream Linux distributions released in the last ≈5 years. If you don't have a particular preference, we recommended using Ubuntu Linux LTS due to its stability and the fact that most of DuckDB's Linux test suite jobs run on Ubuntu workers.

glibc vs. musl libc

DuckDB can be built with both glibc (default) and musl libc (see the build guide). However, note that DuckDB binaries built with musl libc have lower performance. In practice, this can lead to a slowdown of more than 5× on compute-intensive workloads. Therefore, it's recommended to use a Linux distribution with glibc for performance-oriented workloads when running DuckDB.

Memory Allocator

If you have a many-core CPU running on a system where DuckDB ships with `jemalloc` as the default memory allocator, consider enabling the allocator's background threads.

Data Import

Recommended Import Methods

When importing data from other systems to DuckDB, there are several considerations to take into account. We recommend importing using the following order:

1. For systems which are supported by a DuckDB scanner extension, it's preferable to use the scanner. DuckDB currently offers scanners for MySQL, PostgreSQL, and SQLite.
2. If there is a bulk export feature in the data source system, export the data to Parquet or CSV format, then load it using DuckDB's Parquet or CSV loader.
3. If the approaches above are not applicable, consider using the DuckDB appender, currently available in the C, C++, Go, Java, and Rust APIs.

Methods to Avoid

If possible, avoid looping row-by-row (tuple-at-a-time) in favor of bulk operations. Performing row-by-row inserts (even with prepared statements) is detrimental to performance and will result in slow load times.

> **Best practice.** Unless your data is small (<100k rows), avoid using inserts in loops.

Schema

Types

It is important to use the correct type for encoding columns (e.g., `BIGINT`, `DATE`, `DATETIME`). While it is always possible to use string types (VARCHAR, etc.) to encode more specific values, this is not recommended. Strings use more space and are slower to process in operations such as filtering, join, and aggregation.

When loading CSV files, you may leverage the CSV reader's auto-detection mechanism to get the correct types for CSV inputs.

If you run in a memory-constrained environment, using smaller data types (e.g., `TINYINT`) can reduce the amount of memory and disk space required to complete a query. DuckDB's bitpacking compression means small values stored in larger data types will not take up larger sizes on disk, but they will take up more memory during processing.

> **Best practice.** Use the most restrictive types possible when creating columns. Avoid using strings for encoding more specific data items.

Microbenchmark: Using Timestamps

We illustrate the difference in aggregation speed using the `creationDate` column of the LDBC Comment table on scale factor 300. This table has approx. 554 million unordered timestamp values. We run a simple aggregation query that returns the average day-of-the month from the timestamps in two configurations.

First, we use a `DATETIME` to encode the values and run the query using the `extract` datetime function:

```
SELECT avg(extract('day' FROM creationDate)) FROM Comment;
```

Second, we use the VARCHAR type and use string operations:

```
SELECT avg(CAST(creationDate[9:10] AS INTEGER)) FROM Comment;
```

The results of the microbenchmark are as follows:

Column type	Storage size	Query time
DATETIME	3.3 GB	0.9 s
VARCHAR	5.2 GB	3.9 s

The results show that using the DATETIME value yields smaller storage sizes and faster processing.

Microbenchmark: Joining on Strings

We illustrate the difference caused by joining on different types by computing a self-join on the LDBC Comment table at scale factor 100. The table has 64-bit integer identifiers used as the `id` attribute of each row. We perform the following join operation:

```
SELECT count(*) AS count
FROM Comment c1
JOIN Comment c2 ON c1.ParentCommentId = c2.id;
```

In the first experiment, we use the correct (most restrictive) types, i.e., both the `id` and the `ParentCommentId` columns are defined as `BIGINT`. In the second experiment, we define all columns with the `VARCHAR` type. While the results of the queries are the same for all both experiments, their runtime vary significantly. The results below show that joining on `BIGINT` columns is approx. 1.8× faster than performing the same join on VARCHAR-typed columns encoding the same value.

Join column payload type	Join column schema type	Example value	Query time
BIGINT	BIGINT	70368755640078	1.2 s
BIGINT	VARCHAR	'70368755640078'	2.1 s

> **Best practice.** Avoid representing numeric values as strings, especially if you intend to perform e.g., join operations on them.

Constraints

DuckDB allows defining constraints such as `UNIQUE`, `PRIMARY KEY`, and `FOREIGN KEY`. These constraints can be beneficial for ensuring data integrity but they have a negative effect on load performance as they necessitate building indexes and performing checks. Moreover, they *very rarely improve the performance of queries* as DuckDB does not rely on these indexes for join and aggregation operators (see indexing for more details).

> **Best practice.** Do not define constraints unless your goal is to ensure data integrity.

Microbenchmark: The Effect of Primary Keys

We illustrate the effect of using primary keys with the LDBC Comment table at scale factor 300. This table has approx. 554 million entries. In the first experiments, we create the schema *without* a primary key, then load the data. In the second experiment, we create the schema *with* a primary key, then load the data. In the third case, we create the schema *without* a primary key, load the data and then add the primary key constraint. In all cases, we take the data from `.csv.gz` files, and measure the time required to perform the loading.

Operation	Execution time
Load with primary key	461.6 s

Operation	Execution time
Load without primary key	121.0 s
Load without primary key then add primary key	242.0 s

For this data set, primary keys will only have a (small) positive effect on highly selective queries such as when filtering on a single identifier. Defining primary keys (or indexes) will not have an effect on join and aggregation operators.

> **Best practice.** For best bulk load performance, avoid primary key constraints. If they are required, define them after the bulk loading step.

Indexing

DuckDB has two types of indexes: zonemaps and ART indexes.

Zonemaps

DuckDB automatically creates zonemaps (also known as min-max indexes) for the columns of all general-purpose data types. Operations like predicate pushdown into scan operators and computing aggregations use zonemaps. If a filter criterion (like `WHERE column1 = 123`) is in use, DuckDB can skip any row group whose min-max range does not contain that filter value (e.g., it can omit a block with a min-max range of 1000 to 2000 when comparing for `= 123` or `< 400`).

The Effect of Ordering on Zonemaps

The more ordered the data within a column, the more valuable the zonemap indexes will be. For example, a column could contain a random number on every row in the worst case. Then, DuckDB will likely be unable to skip any row groups. If you query specific columns with selective filters, it is best to pre-order data by those columns when inserting it. Even an imperfect ordering will still be helpful. The best case of ordered data commonly arises with `DATETIME` columns.

Microbenchmark: The Effect of Ordering

For an example, let's repeat the microbenchmark for timestamps with an ordered timestamp column using an ascending order vs. an unordered one.

Column type	Ordered	Storage size	Query time
`DATETIME`	yes	1.3 GB	0.6 s
`DATETIME`	no	3.3 GB	0.9 s

The results show that simply keeping the column order allows for improved compression, yielding a 2.5× smaller storage size. It also allows the computation to be 1.5× faster.

Ordered Integers

Another practical way to exploit ordering is to use the `INTEGER` type with automatic increments rather than `UUID` for columns queried using selective filters. In a scenario where a table contains out-of-order `UUID`s, DuckDB has to scan many row groups to find a specific `UUID` value. An ordered `INTEGER` column allows skipping all row groups except those containing the value.

ART Indexes

DuckDB allows defining Adaptive Radix Tree (ART) indexes in two ways. First, such an index is created implicitly for columns with `PRIMARY KEY`, `FOREIGN KEY`, and `UNIQUE` constraints. Second, explicitly running the `CREATE INDEX` statement creates an ART index on the target column(s).

The tradeoffs of having an ART index on a column are as follows:

1. ART indexes enable constraint checking during changes (inserts, updates, and deletes).
2. Changes on indexed tables perform worse than their non-indexed counterparts. That is because of index maintenance for these operations.
3. For some use cases, *single-column ART indexes* improve the performance of highly selective queries using the indexed column.

An ART index does not affect the performance of join, aggregation, and sorting queries.

ART Index Scans

ART index scans probe a single-column ART index for the requested data instead of scanning a table sequentially. Probing can improve the performance of some queries. DuckDB will try to use an index scan for equality and `IN(...)` conditions. It also pushes dynamic filters, e.g., from hash joins, into the scan, allowing dynamic index scans on these filters.

Indexes are only eligible for index scans if they index a single column without expressions. E.g., the following index is eligible for index scans:

```
CREATE INDEX idx ON tbl (col1);
```

E.g., the following two indexes are **NOT** eligible for index scans:

```
CREATE INDEX idx_multi_column ON tbl (col1, col2);
CREATE INDEX idx_expr ON tbl (col1 + 1);
```

The default threshold for index scans is `MAX(2048, 0.001 * table_cardinality)`. You can configure this threshold via `index_scan_percentage` and `index_scan_max_count`, or disable them by setting these values to zero. When in doubt, use `EXPLAIN ANALYZE` to verify that your query plan uses the index scan.

Indexes and Memory

DuckDB registers index memory through its buffer manager. However, these index buffers are not yet buffer-managed. That means DuckDB does not yet destroy any index buffers if it has to evict memory. Thus, indexes can take up a significant portion of DuckDB's available memory, potentially affecting the performance of memory-intensive queries. Re-attaching (`DETACH` + `ATTACH`) the database containing indexes can mitigate this effect, as we deserialize index memory lazily. Disabling index scans and re-attaching after changes can further decrease the impact of indexes on DuckDB's available memory.

Indexes and Opening Databases

Indexes are serialized to disk and deserialized lazily, i.e., when reopening the database. Operations using the index will only load the required parts of the index. Therefore, having an index will not cause any slowdowns when opening an existing database.

> **Best practice.** We recommend following these guidelines:
>
> - Only use primary keys, foreign keys, or unique constraints, if these are necessary for enforcing constraints on your data.
> - Do not define explicit indexes unless you have highly selective queries and enough memory available.
> - If you define an ART index, do so after bulk loading the data to the table. Adding an index prior to loading, either explicitly or via primary/foreign keys, is detrimental to load performance.

Join Operations

How to Force a Join Order

DuckDB has a cost-based query optimizer, which uses statistics in the base tables (stored in a DuckDB database or Parquet files) to estimate the cardinality of operations.

Turn off the Join Order Optimizer

To turn off the join order optimizer, set the following PRAGMAs:

```
SET disabled_optimizers = 'join_order,build_side_probe_side';
```

This disables both the join order optimizer and left/right swapping for joins. This way, DuckDB builds a left-deep join tree following the order of `JOIN` clauses.

```
SELECT ...
FROM ...
JOIN ...  -- this join is performed first
JOIN ...; -- this join is performed second
```

Once the query in question has been executed, turn back the optimizers with the following command:

```
SET disabled_optimizers = '';
```

Create Temporary Tables

To force a particular join order, you can break up the query into multiple queries with each creating a temporary tables:

```
CREATE OR REPLACE TEMPORARY TABLE t1 AS
    ...;

-- join on the result of the first query, t1
CREATE OR REPLACE TEMPORARY TABLE t2 AS
    SELECT * FROM t1 ...;

-- compute the final result using t2
SELECT * FROM t1 ...
```

To clean up, drop the interim tables:

```
DROP TABLE IF EXISTS t1;
DROP TABLE IF EXISTS t2;
```

File Formats

Handling Parquet Files

DuckDB has advanced support for Parquet files, which includes directly querying Parquet files. When deciding on whether to query these files directly or to first load them to the database, you need to consider several factors.

Reasons for Querying Parquet Files

Availability of basic statistics: Parquet files use a columnar storage format and contain basic statistics such as zonemaps. Thanks to these features, DuckDB can leverage optimizations such as projection and filter pushdown on Parquet files. Therefore, workloads that combine projection, filtering, and aggregation tend to perform quite well when run on Parquet files.

Storage considerations: Loading the data from Parquet files will require approximately the same amount of space for the DuckDB database file. Therefore, if the available disk space is constrained, it is worth running the queries directly on Parquet files.

Reasons against Querying Parquet Files

Lack of advanced statistics: The DuckDB database format has the hyperloglog statistics that Parquet files do not have. These improve the accuracy of cardinality estimates, and are especially important if the queries contain a large number of join operators.

Tip. If you find that DuckDB produces a suboptimal join order on Parquet files, try loading the Parquet files to DuckDB tables. The improved statistics likely help obtain a better join order.

Repeated queries: If you plan to run multiple queries on the same data set, it is worth loading the data into DuckDB. The queries will always be somewhat faster, which over time amortizes the initial load time.

High decompression times: Some Parquet files are compressed using heavyweight compression algorithms such as gzip. In these cases, querying the Parquet files will necessitate an expensive decompression time every time the file is accessed. Meanwhile, lightweight compression methods like Snappy, LZ4, and zstd, are faster to decompress. You may use the `parquet_metadata` function to find out the compression algorithm used.

Microbenchmark: Running TPC-H on a DuckDB Database vs. Parquet

The queries on the TPC-H benchmark run approximately 1.1-5.0× slower on Parquet files than on a DuckDB database.

> **Best practice.** If you have the storage space available, and have a join-heavy workload and/or plan to run many queries on the same dataset, load the Parquet files into the database first. The compression algorithm and the row group sizes in the Parquet files have a large effect on performance: study these using the `parquet_metadata` function.

The Effect of Row Group Sizes

DuckDB works best on Parquet files with row groups of 100K-1M rows each. The reason for this is that DuckDB can only parallelize over row groups – so if a Parquet file has a single giant row group it can only be processed by a single thread. You can use the `parquet_metadata` function to figure out how many row groups a Parquet file has. When writing Parquet files, use the `row_group_size` option.

Microbenchmark: Running Aggregation Query at Different Row Group Sizes

We run a simple aggregation query over Parquet files using different row group sizes, selected between 960 and 1,966,080. The results are as follows.

Row group size	Execution time
960	8.77 s
1920	8.95 s
3840	4.33 s
7680	2.35 s
15360	1.58 s
30720	1.17 s

Row group size	Execution time
61440	0.94 s
122880	0.87 s
245760	0.93 s
491520	0.95 s
983040	0.97 s
1966080	0.88 s

The results show that row group sizes <5,000 have a strongly detrimental effect, making runtimes more than 5-10× larger than ideally-sized row groups, while row group sizes between 5,000 and 20,000 are still 1.5-2.5× off from best performance. Above row group size of 100,000, the differences are small: the gap is about 10% between the best and the worst runtime.

Parquet File Sizes

DuckDB can also parallelize across multiple Parquet files. It is advisable to have at least as many total row groups across all files as there are CPU threads. For example, with a machine having 10 threads, both 10 files with 1 row group or 1 file with 10 row groups will achieve full parallelism. It is also beneficial to keep the size of individual Parquet files moderate.

> **Best practice.** The ideal range is between 100 MB and 10 GB per individual Parquet file.

Hive Partitioning for Filter Pushdown

When querying many files with filter conditions, performance can be improved by using a Hive-format folder structure to partition the data along the columns used in the filter condition. DuckDB will only need to read the folders and files that meet the filter criteria. This can be especially helpful when querying remote files.

More Tips on Reading and Writing Parquet Files

For tips on reading and writing Parquet files, see the Parquet Tips page.

Loading CSV Files

CSV files are often distributed in compressed format such as GZIP archives (`.csv.gz`). DuckDB can decompress these files on the fly. In fact, this is typically faster than decompressing the files first and loading them due to reduced IO.

Schema	Load time
Load from GZIP-compressed CSV files (`.csv.gz`)	107.1 s
Decompressing (using parallel `gunzip`) and loading from decompressed CSV files	121.3 s

Loading Many Small CSV Files

The CSV reader runs the CSV sniffer on all files. For many small files, this may cause an unnecessarily high overhead. A potential optimization to speed this up is to turn the sniffer off. Assuming that all files have the same CSV dialect and colum names/types, get the sniffer options as follows:

```
.mode line
SELECT Prompt FROM sniff_csv('part-0001.csv');
```

```
Prompt = FROM read_csv('file_path.csv', auto_detect=false, delim=',', quote='"', escape='"', new_
line='\n', skip=0, header=true, columns={'hello': 'BIGINT', 'world': 'VARCHAR'});
```

Then, you can adjust `read_csv` command, by e.g., applying filename expansion (globbing), and run with the rest of the options detected by the sniffer:

```
FROM read_csv('part-*.csv', auto_detect=false, delim=',', quote='"', escape='"', new_line='\n', skip=0,
header=true, columns={'hello': 'BIGINT', 'world': 'VARCHAR'});
```

Tuning Workloads

The `preserve_insertion_order` Option

When importing or exporting data sets (from/to the Parquet or CSV formats), which are much larger than the available memory, an out of memory error may occur:

```
Out of Memory Error: failed to allocate data of size ... (.../... used)
```

In these cases, consider setting the `preserve_insertion_order` configuration option to `false`:

```
SET preserve_insertion_order = false;
```

This allows the systems to re-order any results that do not contain `ORDER BY` clauses, potentially reducing memory usage.

Parallelism (Multi-Core Processing)

The Effect of Row Groups on Parallelism

DuckDB parallelizes the workload based on *row groups*, i.e., groups of rows that are stored together at the storage level. A row group in DuckDB's database format consists of max. 122,880 rows. Parallelism starts at the level of row groups, therefore, for a query to run on *k* threads, it needs to scan at least *k* * 122,880 rows.

Too Many Threads

Note that in certain cases DuckDB may launch *too many threads* (e.g., due to HyperThreading), which can lead to slowdowns. In these cases, it's worth manually limiting the number of threads using `SET threads = X`.

Larger-than-Memory Workloads (Out-of-Core Processing)

A key strength of DuckDB is support for larger-than-memory workloads, i.e., it is able to process data sets that are larger than the available system memory (also known as *out-of-core processing*). It can also run queries where the intermediate results cannot fit into memory. This section explains the prerequisites, scope, and known limitations of larger-than-memory processing in DuckDB.

Spilling to Disk

Larger-than-memory workloads are supported by spilling to disk. With the default configuration, DuckDB creates the `<database_file_name>.tmp{:.language-sql .highlight}` temporary directory (in persistent mode) or the `.tmp{:.language-sql .highlight}` directory (in in-memory mode). This directory can be changed using the `temp_directory` configuration option, e.g.:

```
SET temp_directory = '/path/to/temp_dir.tmp/';
```

Blocking Operators

Some operators cannot output a single row until the last row of their input has been seen. These are called *blocking operators* as they require their entire input to be buffered, and are the most memory-intensive operators in relational database systems. The main blocking operators are the following:

- *grouping:* `GROUP BY`
- *joining:* `JOIN`
- *sorting:* `ORDER BY`
- *windowing:* `OVER ... (PARTITION BY ... ORDER BY ...)`

DuckDB supports larger-than-memory processing for all of these operators.

Limitations

DuckDB strives to always complete workloads even if they are larger-than-memory. That said, there are some limitations at the moment:

- If multiple blocking operators appear in the same query, DuckDB may still throw an out-of-memory exception due to the complex interplay of these operators.
- Some aggregate functions, such as `list()` and `string_agg()`, do not support offloading to disk.
- Aggregate functions that use sorting are holistic, i.e., they need all inputs before the aggregation can start. As DuckDB cannot yet offload some complex intermediate aggregate states to disk, these functions can cause an out-of-memory exception when run on large data sets.
- The `PIVOT` operation internally uses the `list()` function, therefore it is subject to the same limitation.

Profiling

If your queries are not performing as well as expected, it's worth studying their query plans:

- Use `EXPLAIN` to print the physical query plan without running the query.
- Use `EXPLAIN ANALYZE` to run and profile the query. This will show the CPU time that each step in the query takes. Note that due to multi-threading, adding up the individual times will be larger than the total query processing time.

Query plans can point to the root of performance issues. A few general directions:

- Avoid nested loop joins in favor of hash joins.
- A scan that does not include a filter pushdown for a filter condition that is later applied performs unnecessary IO. Try rewriting the query to apply a pushdown.
- Bad join orders where the cardinality of an operator explodes to billions of tuples should be avoided at all costs.

Prepared Statements

Prepared statements can improve performance when running the same query many times, but with different parameters. When a statement is prepared, it completes several of the initial portions of the query execution process (parsing, planning, etc.) and caches their output. When it is executed, those steps can be skipped, improving performance. This is beneficial mostly for repeatedly running small queries (with a runtime of < 100ms) with different sets of parameters.

Note that it is not a primary design goal for DuckDB to quickly execute many small queries concurrently. Rather, it is optimized for running larger, less frequent queries.

Querying Remote Files

DuckDB uses synchronous IO when reading remote files. This means that each DuckDB thread can make at most one HTTP request at a time. If a query must make many small requests over the network, increasing DuckDB's `threads` setting to larger than the total number of CPU cores (approx. 2-5 times CPU cores) can improve parallelism and performance.

Avoid Reading Unnecessary Data

The main bottleneck in workloads reading remote files is likely to be the IO. This means that minimizing the unnecessarily read data can be highly beneficial.

Some basic SQL tricks can help with this:

- Avoid `SELECT *`. Instead, only select columns that are actually used. DuckDB will try to only download the data it actually needs.
- Apply filters on remote parquet files when possible. DuckDB can use these filters to reduce the amount of data that is scanned.
- Either sort or partition data by columns that are regularly used for filters: this increases the effectiveness of the filters in reducing IO.

To inspect how much remote data is transferred for a query, `EXPLAIN ANALYZE` can be used to print out the total number of requests and total data transferred for queries on remote files.

Avoid Reading Data More Than Once

DuckDB does not cache data from remote files automatically. This means that running a query on a remote file twice will download the required data twice. So if data needs to be accessed multiple times, storing it locally can make sense. To illustrate this, lets look at an example:

Consider the following queries:

```
SELECT col_a + col_b FROM 's3://bucket/file.parquet' WHERE col_a > 10;
SELECT col_a * col_b FROM 's3://bucket/file.parquet' WHERE col_a > 10;
```

These queries download the columns `col_a` and `col_b` from `s3://bucket/file.parquet` twice. Now consider the following queries:

```
CREATE TABLE local_copy_of_file AS
    SELECT col_a, col_b FROM 's3://bucket/file.parquet' WHERE col_a > 10;

SELECT col_a + col_b FROM local_copy_of_file;
SELECT col_a * col_b FROM local_copy_of_file;
```

Here DuckDB will first copy `col_a` and `col_b` from `s3://bucket/file.parquet` into a local table, and then query the local in-memory columns twice. Note also that the filter `WHERE col_a > 10` is also now applied only once.

An important side note needs to be made here though. The first two queries are fully streaming, with only a small memory footprint, whereas the second requires full materialization of columns `col_a` and `col_b`. This means that in some rare cases (e.g., with a high-speed network, but with very limited memory available) it could actually be beneficial to download the data twice.

Best Practices for Using Connections

DuckDB will perform best when reusing the same database connection many times. Disconnecting and reconnecting on every query will incur some overhead, which can reduce performance when running many small queries. DuckDB also caches some data and metadata in memory, and that cache is lost when the last open connection is closed. Frequently, a single connection will work best, but a connection pool may also be used.

Using multiple connections can parallelize some operations, although it is typically not necessary. DuckDB does attempt to parallelize as much as possible within each individual query, but it is not possible to parallelize in all cases. Making multiple connections can process

more operations concurrently. This can be more helpful if DuckDB is not CPU limited, but instead bottlenecked by another resource like network transfer speed.

Persistent vs. In-Memory Tables

DuckDB supports lightweight compression techniques. Currently, these are only applied on persistent (on-disk) databases.

DuckDB does not compress its in-memory tables. The reason for this is that compression is performed during checkpointing, and in-memory tables are not checkpointed.

In some cases, this can result in counter-intuitive performance results where queries are faster on on-disk tables compared to in-memory ones. For example, Q1 of the TPC-H workload is faster when running on-disk compared to the in-memory mode:

```
INSTALL tpch;
LOAD tpch;
CALL dbgen(sf = 30);
.timer on
PRAGMA tpch(1);
```

Database setup	Execution time
In-memory database	4.80 s
Persistent database	0.57 s

My Workload Is Slow

If you find that your workload in DuckDB is slow, we recommend performing the following checks. More detailed instructions are linked for each point.

1. Do you have enough memory? DuckDB works best if you have 1-4 GB memory per thread.
2. Are you using a fast disk? Network-attached disks (such as cloud block storage) cause write-intensive and larger than memory workloads to slow down. For running such workloads in cloud environments, it is recommended to use instance-attached storage (NVMe SSDs).
3. Are you using indexes or constraints (primary key, unique, etc.)? If possible, try disabling them, which boosts load and update performance.
4. Are you using the correct types? For example, use `TIMESTAMP` to encode datetime values.
5. Are you reading from Parquet files? If so, do they have row group sizes between 100k and 1M and file sizes between 100 MB to 10 GB?
6. Does the query plan look right? Study it with `EXPLAIN`.
7. Is the workload running in parallel? Use `htop` or the operating system's task manager to observe this.
8. Is DuckDB using too many threads? Try limiting the amount of threads.

Are you aware of other common issues? If so, please click the *Report content issue* link below and describe them along with their workarounds.

Benchmarks

For several of the recommendations in our performance guide, we use microbenchmarks to back up our claims. For these benchmarks, we use data sets from the TPC-H benchmark and the LDBC Social Network Benchmark's BI workload.

Data Sets

We use the LDBC BI SF300 data set's Comment table (20 GB `.tar.zst` archive, 21 GB when decompressed into `.csv.gz` files), while others use the same table's `creationDate` column (4 GB `.parquet` file).

The TPC data sets used in the benchmark are generated with the DuckDB tpch extension.

A Note on Benchmarks

Running fair benchmarks is difficult, especially when performing system-to-system comparison. When running benchmarks on DuckDB, please make sure you are using the latest version (preferably the preview build). If in doubt about your benchmark results, feel free to contact us at `gabor@duckdb.org`.

Disclaimer on Benchmarks

Note that the benchmark results presented in this guide do not constitute official TPC or LDBC benchmark results. Instead, they merely use the data sets of and some queries provided by the TPC-H and the LDBC BI benchmark frameworks, and omit other parts of the workloads such as updates.

Working with Huge Databases

This page contains information for working with huge DuckDB database files. While most DuckDB databases are well below 1 TB, in our 2024 user survey, 1% of respondents used DuckDB files of 2 TB or more (corresponding to roughly 10 TB of CSV files).

DuckDB's native database format supports huge database files without any practical restrictions, however, there are a few things to keep in mind when working with huge database files.

1. Object storage systems have lower limits on file sizes than block-based storage systems. For example, AWS S3 limits the file size to 5 TB.
2. Checkpointing a DuckDB database can be slow. For example, checkpointing after adding a few rows to a table in the TPC-H SF1000 database takes approximately 5 seconds.
3. On block-based storage, the file has a big effect on performance when working with large files. On Linux, DuckDB performs best with XFS on large files.

For storing large amounts of data, consider using the DuckLake lakehouse format.

Python

Installing the Python Client

Installing via Pip

The latest release of the Python client can be installed using `pip`.

```
pip install duckdb
```

The pre-release Python client can be installed using `--pre`.

```
pip install duckdb --upgrade --pre
```

Installing from Source

The latest Python client can be installed from source from the `tools/pythonpkg` directory in the DuckDB GitHub repository.

```
BUILD_PYTHON=1 GEN=ninja make
cd tools/pythonpkg
python setup.py install
```

For detailed instructions on how to compile DuckDB from source, see the Building guide.

Executing SQL in Python

SQL queries can be executed using the `duckdb.sql` function.

```
import duckdb

duckdb.sql("SELECT 42").show()
```

By default this will create a relation object. The result can be converted to various formats using the result conversion functions. For example, the `fetchall` method can be used to convert the result to Python objects.

```
results = duckdb.sql("SELECT 42").fetchall()
print(results)
```

```
[(42,)]
```

Several other result objects exist. For example, you can use `df` to convert the result to a Pandas DataFrame.

```
results = duckdb.sql("SELECT 42").df()
print(results)
```

```
    42
 0  42
```

By default, a global in-memory connection will be used. Any data stored in files will be lost after shutting down the program. A connection to a persistent database can be created using the `connect` function.

After connecting, SQL queries can be executed using the `sql` command.

```
con = duckdb.connect("file.db")
con.sql("CREATE TABLE integers (i INTEGER)")
con.sql("INSERT INTO integers VALUES (42)")
con.sql("SELECT * FROM integers").show()
```

Jupyter Notebooks

DuckDB's Python client can be used directly in Jupyter notebooks with no additional configuration if desired. However, additional libraries can be used to simplify SQL query development. This guide will describe how to utilize those additional libraries. See other guides in the Python section for how to use DuckDB and Python together.

In this example, we use the JupySQL package. This example workflow is also available as a Google Colab notebook.

Library Installation

Four additional libraries improve the DuckDB experience in Jupyter notebooks.

1. jupysql: Convert a Jupyter code cell into a SQL cell
2. Pandas: Clean table visualizations and compatibility with other analysis
3. matplotlib: Plotting with Python
4. duckdb-engine (DuckDB SQLAlchemy driver): Used by SQLAlchemy to connect to DuckDB (optional)

Run these `pip install` commands from the command line if Jupyter Notebook is not yet installed. Otherwise, see Google Colab link above for an in-notebook example:

```
pip install duckdb
```

Install Jupyter Notebook

```
pip install notebook
```

Or JupyterLab:

```
pip install jupyterlab
```

Install supporting libraries:

```
pip install jupysql pandas matplotlib duckdb-engine
```

Library Import and Configuration

Open a Jupyter Notebook and import the relevant libraries.

Set configurations on jupysql to directly output data to Pandas and to simplify the output that is printed to the notebook.

```
%config SqlMagic.autopandas = True
%config SqlMagic.feedback = False
%config SqlMagic.displaycon = False
```

Connecting to DuckDB Natively

To connect to DuckDB, run:

```
import duckdb
import pandas as pd

%load_ext sql
conn = duckdb.connect()
%sql conn --alias duckdb
```

> **Warning.** Variables are not recognized within a native DuckDB connection.

Connecting to DuckDB via SQLAlchemy

Alternatively, you can connect to DuckDB via SQLAlchemy using `duckdb_engine`. See the performance and feature differences.

```
import duckdb
import pandas as pd
# No need to import duckdb_engine
#  jupysql will auto-detect the driver needed based on the connection string!

# Import jupysql Jupyter extension to create SQL cells
%load_ext sql
```

Either connect to a new in-memory DuckDB, the default connection or a file backed database:

```
%sql duckdb:///:memory:
```

```
%sql duckdb:///:default:
```

```
%sql duckdb:///path/to/file.db
```

> The `%sql` command and `duckdb.sql` share the same default connection if you provide `duckdb:///:default:` as the SQLAlchemy connection string.

Querying DuckDB

Single line SQL queries can be run using `%sql` at the start of a line. Query results will be displayed as a Pandas DataFrame.

```
%sql SELECT 'Off and flying!' AS a_duckdb_column;
```

An entire Jupyter cell can be used as a SQL cell by placing `%%sql` at the start of the cell. Query results will be displayed as a Pandas DataFrame.

```
%%sql
SELECT
    schema_name,
    function_name
FROM duckdb_functions()
ORDER BY ALL DESC
LIMIT 5;
```

To store the query results in a Python variable, use `<<` as an assignment operator. This can be used with both the `%sql` and `%%sql` Jupyter magics.

```
%sql res << SELECT 'Off and flying!' AS a_duckdb_column;
```

If the `%config SqlMagic.autopandas = True` option is set, the variable is a Pandas dataframe, otherwise, it is a `ResultSet` that can be converted to Pandas with the `DataFrame()` function.

Querying Pandas Dataframes

DuckDB is able to find and query any dataframe stored as a variable in the Jupyter notebook.

```
input_df = pd.DataFrame.from_dict({"i": [1, 2, 3],
                                   "j": ["one", "two", "three"]})
```

The dataframe being queried can be specified just like any other table in the FROM clause.

```
%sql output_df << SELECT sum(i) AS total_i FROM input_df;
```

> **Warning.** When using the SQLAlchemy connection, and DuckDB >= 1.1.0, make sure to run `%sql SET python_scan_all_frames=true`, to make Pandas dataframes queryable.

Visualizing DuckDB Data

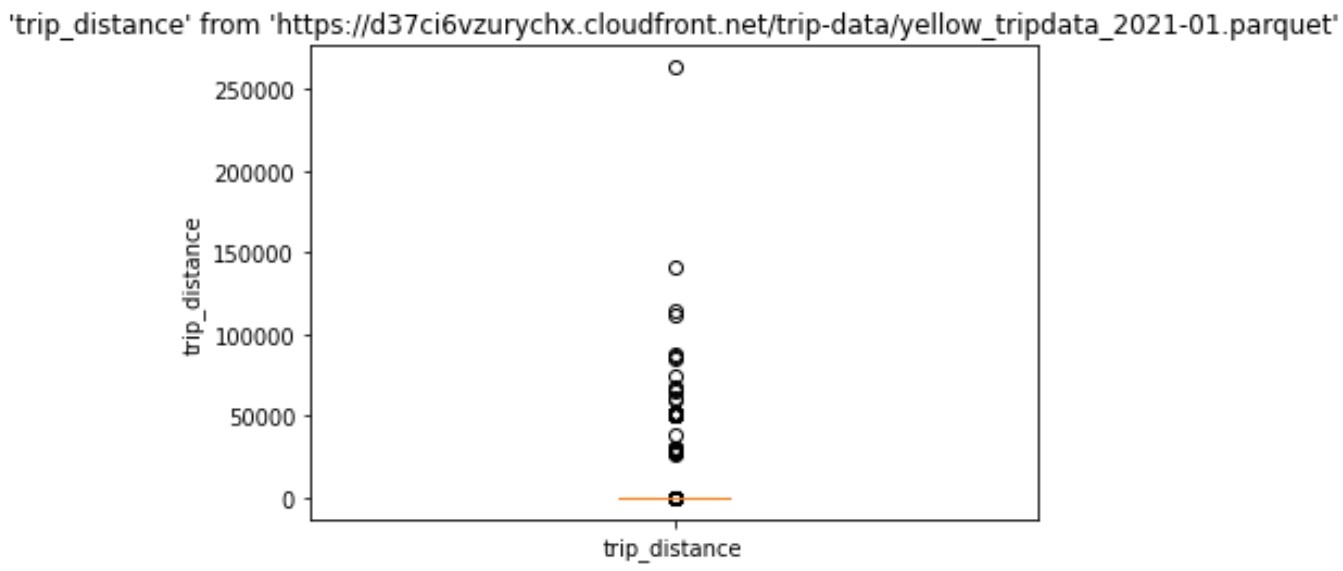
The most common way to plot datasets in Python is to load them using Pandas and then use matplotlib or seaborn for plotting. This approach requires loading all data into memory which is highly inefficient. The plotting module in JupySQL runs computations in the SQL engine. This delegates memory management to the engine and ensures that intermediate computations do not keep eating up memory, efficiently plotting massive datasets.

Boxplot & Histogram

To create a boxplot, call `%sqlplot boxplot`, passing the name of the table and the column to plot. In this case, the name of the table is the path of the locally stored Parquet file.

```
from urllib.request import urlretrieve

_ = urlretrieve(
    "https://d37ci6vzurychx.cloudfront.net/trip-data/yellow_tripdata_2021-01.parquet",
    "yellow_tripdata_2021-01.parquet",
)
```

```
%sqlplot boxplot --table yellow_tripdata_2021-01.parquet --column trip_distance
```



Install and Load DuckDB httpfs Extension

DuckDB's httpfs extension allows Parquet and CSV files to be queried remotely over http. These examples query a Parquet file that contains historical taxi data from NYC. Using the Parquet format allows DuckDB to only pull the rows and columns into memory that are needed rather than downloading the entire file. DuckDB can be used to process local Parquet files as well, which may be desirable if querying the entire Parquet file, or running multiple queries that require large subsets of the file.

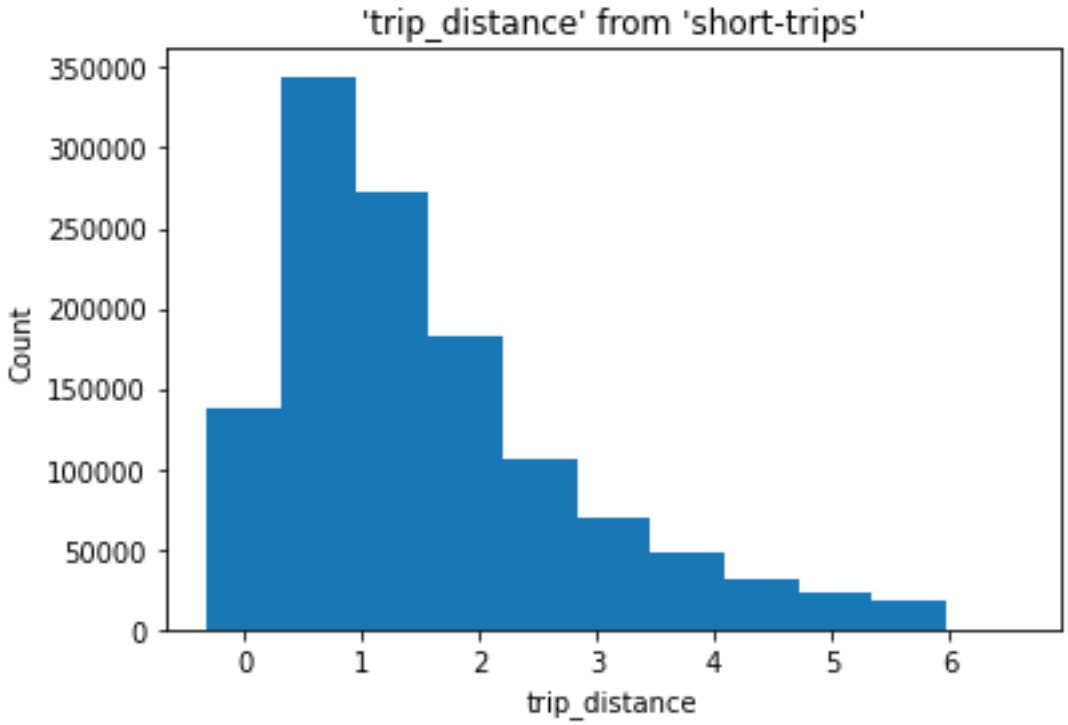
```
%%sql
INSTALL httpfs;
LOAD httpfs;
```

Now, create a query that filters by the 90th percentile. Note the use of the `--save`, and `--no-execute` functions. This tells JupySQL to store the query, but skips execution. It will be referenced in the next plotting call.

```
%%sql --save short_trips --no-execute
SELECT *
FROM 'https://d37ci6vzurychx.cloudfront.net/trip-data/yellow_tripdata_2021-01.parquet'
WHERE trip_distance < 6.3
```

To create a histogram, call `%sqlplot histogram` and pass the name of the table, the column to plot, and the number of bins. This uses `--with short-trips` so JupySQL uses the query defined previously and therefore only plots a subset of the data.

```
%sqlplot histogram --table short_trips --column trip_distance --bins 10 --with short_trips
```



Summary

You now have the ability to alternate between SQL and Pandas in a simple and highly performant way! You can plot massive datasets directly through the engine (avoiding both the download of the entire file and loading all of it into Pandas in memory). Dataframes can be read as tables in SQL, and SQL results can be output into Dataframes. Happy analyzing!

An alternative to `jupysql` is `magic_duckdb`.

marimo Notebooks

marimo is an open-source reactive notebook for Python and SQL that's tightly integrated with DuckDB's Python client, letting you mix and match Python and SQL in a single git-versionable notebook. Unlike traditional notebooks, when you run a cell or interact with a UI element, marimo automatically (or lazily) runs affected cells, keeping code and outputs consistent. Its integration with DuckDB makes it well-suited to interactively working with data, and its representation as a Python file makes it simple to run notebooks as scripts.

Installation

To get started, install marimo and DuckDB from your terminal:

```
pip install "marimo[sql]" # or uv add "marimo[sql]"
```

Install supporting libraries:

```
pip install "polars[pyarrow]" # or uv add "polars[pyarrow]"
```

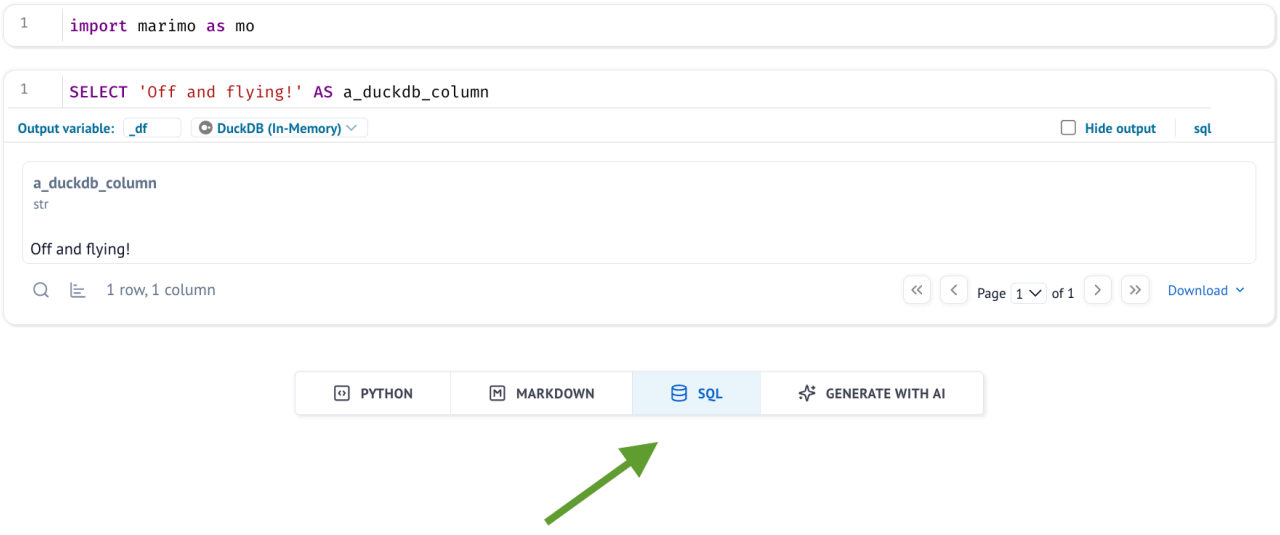
Run a tutorial:

```
marimo tutorial sql
```

SQL in marimo

Create a notebook from your terminal with `marimo edit notebook.py`. Create SQL cells in one of three ways:

1. Right-click the **+** button and pick **SQL cell**
2. Convert any empty cell to SQL via the cell menu
3. Hit the SQL button at the bottom of your notebook



In marimo, SQL cells give the appearance of writing SQL while being serialized as standard Python code using the `mo.sql()` function, which keeps your notebook as pure Python code without requiring special syntax or magic commands.

```
df = mo.sql(f"SELECT 'Off and flying!' AS a_duckdb_column")
```

This is because marimo stores notebooks as pure Python, for many reasons, such as git-friendly diffs and running notebooks as Python scripts.

The SQL statement itself is an f-string, letting you interpolate Python values into the query with `{}` (shown later). In particular, this means your SQL queries can depend on the values of UI elements or other Python values, all part of marimo's dataflow graph.

> **Warning.** Heads up! If you have user-generated content going into the SQL queries, be sure to sanitize your inputs to prevent SQL injection.

Connecting a Custom DuckDB Connection

To connect to a custom DuckDB connection instead of using the default global connection, create a cell and create a DuckDB connection as Python variable:

```
import duckdb

# Create a DuckDB connection
conn = duckdb.connect("path/to/my/duckdb.db")
```

marimo automatically discovers the connection and lets you select it in the SQL cell's connection dropdown.

Database, Schema, and Table Auto-Discovery

marimo introspects connections and display the database, schemas, tables, and columns in the Data Sources panel. This panel lets you quickly navigate your schemas to pull tables and columns into your SQL queries.

Reference a Local Dataframe

Reference a local dataframe in your SQL cell by using the name of the Python variable that holds the dataframe. If you have a database connection with a table of the same name, the database table will be used instead.

```
import polars as pl
df = pl.DataFrame({"column": [1, 2, 3]})
```

```
SELECT * FROM df WHERE column > 2
```

Reference the Output of a SQL Cell

Defining a non-private (non-underscored) output variable in the SQL cell allows you to reference the resulting dataframe in other Python and SQL cells.

Reactive SQL Cells

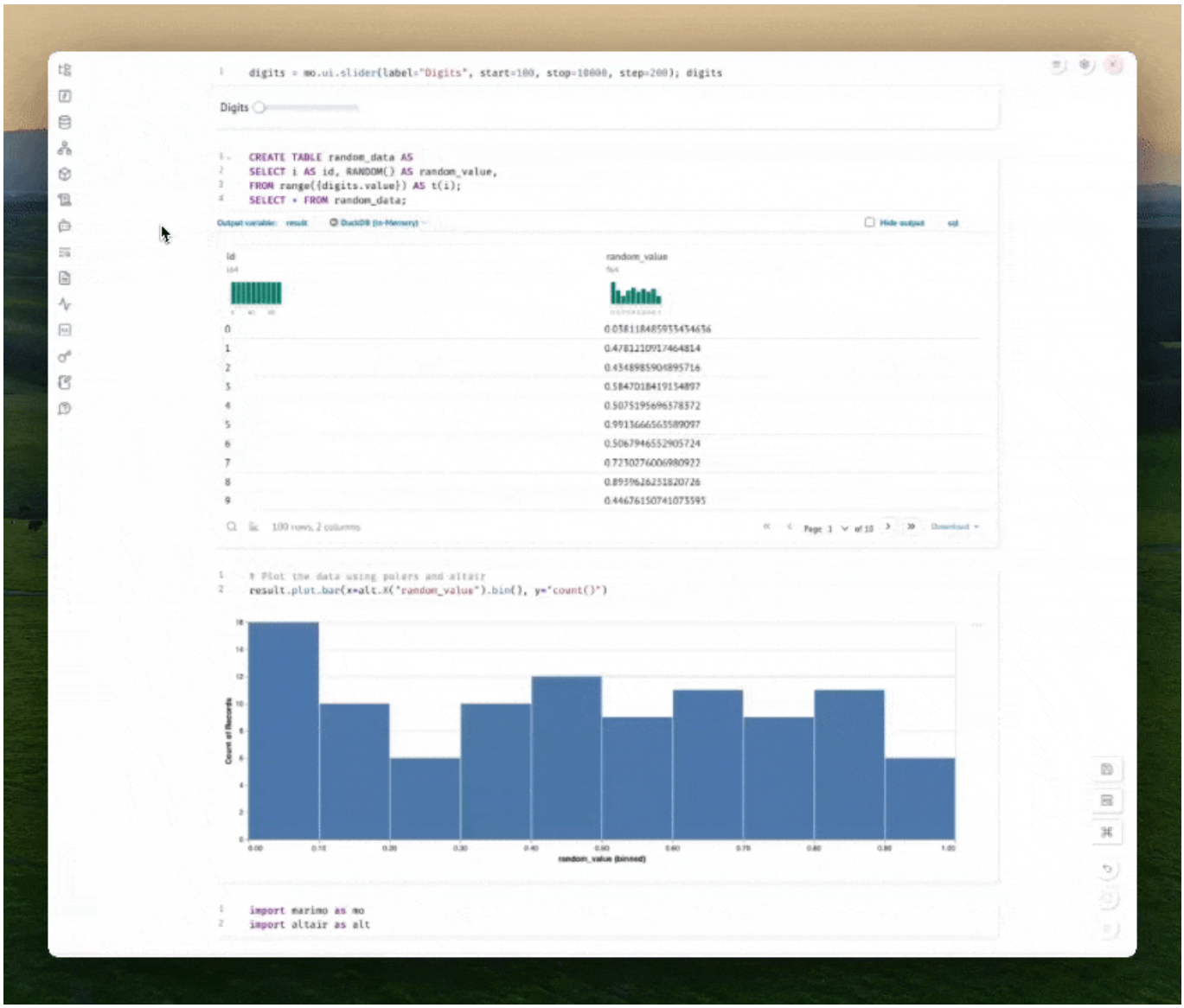
marimo allows you to create reactive SQL cells that automatically update when their dependencies change. **Working with expensive queries or large datasets?** You can configure marimo's runtime to be “lazy”. By doing so, dependent cells are only marked as stale letting the user choose when they should be re-run.

```
digits = mo.ui.slider(label="Digits", start=100, stop=10000, step=200)
digits
```

```
CREATE TABLE random_data AS
    SELECT i AS id, random() AS random_value,
    FROM range({digits.value}) AS t(i);
```

```
SELECT * FROM random_data;
```

Interacting with UI elements, like a slider, makes your data more tangible.



DuckDB-Powered OLAP Analytics in marimo

marimo provides several features that work well with DuckDB for analytical workflows:

- Seamless integration between Python and SQL
- Reactive execution that automatically updates dependent cells when queries change
- Interactive UI elements that can be used to parameterize SQL queries
- Ability to export notebooks as standalone applications or Python scripts, or even run entirely in the browser with WebAssembly.

Next Steps

- Read the marimo docs.
- Try the SQL tutorial: `marimo tutorial sql`.
- The code for this guide is available on GitHub. Run it with `marimo edit <github_url>`.

SQL on Pandas

Pandas DataFrames stored in local variables can be queried as if they are regular tables within DuckDB.

```
import duckdb
import pandas

# Create a Pandas dataframe
my_df = pandas.DataFrame.from_dict({'a': [42]})

# query the Pandas DataFrame "my_df"
# Note: duckdb.sql connects to the default in-memory database connection
results = duckdb.sql("SELECT * FROM my_df").df()
```

The seamless integration of Pandas DataFrames to DuckDB SQL queries is allowed by replacement scans, which replace instances of accessing the `my_df` table (which does not exist in DuckDB) with a table function that reads the `my_df` dataframe.

Import from Pandas

`CREATE TABLE ... AS` and `INSERT INTO` can be used to create a table from any query. We can then create tables or insert into existing tables by referring to the Pandas DataFrame in the query. There is no need to register the DataFrames manually – DuckDB can find them in the Python process by name thanks to replacement scans.

```
import duckdb
import pandas

# Create a Pandas dataframe
my_df = pandas.DataFrame.from_dict({'a': [42]})

# create the table "my_table" from the DataFrame "my_df"
# Note: duckdb.sql connects to the default in-memory database connection
duckdb.sql("CREATE TABLE my_table AS SELECT * FROM my_df")

# insert into the table "my_table" from the DataFrame "my_df"
duckdb.sql("INSERT INTO my_table SELECT * FROM my_df")
```

If the order of columns is different or not all columns are present in the DataFrame, use `INSERT INTO ... BY NAME`:

```
duckdb.sql("INSERT INTO my_table BY NAME SELECT * FROM my_df")
```

See Also

DuckDB also supports exporting to Pandas.

Export to Pandas

The result of a query can be converted to a Pandas DataFrame using the `df()` function.

```
import duckdb

# read the result of an arbitrary SQL query to a Pandas DataFrame
results = duckdb.sql("SELECT 42").df()
results
```

```
   42
0  42
```

See Also

DuckDB also supports importing from Pandas.

Import from Numpy

It is possible to query Numpy arrays from DuckDB. There is no need to register the arrays manually – DuckDB can find them in the Python process by name thanks to replacement scans. For example:

```
import duckdb
import numpy as np

my_arr = np.array([(1, 9.0), (2, 8.0), (3, 7.0)])

duckdb.sql("SELECT * FROM my_arr")
```

column0 double	column1 double	column2 double
1.0	2.0	3.0
9.0	8.0	7.0

See Also

DuckDB also supports exporting to Numpy.

Export to Numpy

The result of a query can be converted to a Numpy array using the `fetchnumpy()` function. For example:

```
import duckdb
import numpy as np

my_arr = duckdb.sql("SELECT unnest([1, 2, 3]) AS x, 5.0 AS y").fetchnumpy()
my_arr
```

```
{'x': array([1, 2, 3], dtype=int32), 'y': masked_array(data=[5.0, 5.0, 5.0],
             mask=[False, False, False],
       fill_value=1e+20)}
```

Then, the array can be processed using Numpy functions, e.g.:

```
np.sum(my_arr['x'])
```

```
6
```

See Also

DuckDB also supports importing from Numpy.

SQL on Apache Arrow

DuckDB can query multiple different types of Apache Arrow objects.

Apache Arrow Tables

Arrow Tables stored in local variables can be queried as if they are regular tables within DuckDB.

```
import duckdb
import pyarrow as pa

# connect to an in-memory database
con = duckdb.connect()

my_arrow_table = pa.Table.from_pydict({'i': [1, 2, 3, 4],
                                       'j': ["one", "two", "three", "four"]})

# query the Apache Arrow Table "my_arrow_table" and return as an Arrow Table
results = con.execute("SELECT * FROM my_arrow_table WHERE i = 2").arrow()
```

Apache Arrow Datasets

Arrow Datasets stored as variables can also be queried as if they were regular tables. Datasets are useful to point towards directories of Parquet files to analyze large datasets. DuckDB will push column selections and row filters down into the dataset scan operation so that only the necessary data is pulled into memory.

```
import duckdb
import pyarrow as pa
import tempfile
import pathlib
import pyarrow.parquet as pq
import pyarrow.dataset as ds

# connect to an in-memory database
con = duckdb.connect()

my_arrow_table = pa.Table.from_pydict({'i': [1, 2, 3, 4],
                                       'j': ["one", "two", "three", "four"]})

# create example Parquet files and save in a folder
base_path = pathlib.Path(tempfile.gettempdir())
(base_path / "parquet_folder").mkdir(exist_ok = True)
pq.write_to_dataset(my_arrow_table, str(base_path / "parquet_folder"))

# link to Parquet files using an Arrow Dataset
my_arrow_dataset = ds.dataset(str(base_path / 'parquet_folder/'))

# query the Apache Arrow Dataset "my_arrow_dataset" and return as an Arrow Table
results = con.execute("SELECT * FROM my_arrow_dataset WHERE i = 2").arrow()
```

Apache Arrow Scanners

Arrow Scanners stored as variables can also be queried as if they were regular tables. Scanners read over a dataset and select specific columns or apply row-wise filtering. This is similar to how DuckDB pushes column selections and filters down into an Arrow Dataset, but

using Arrow compute operations instead. Arrow can use asynchronous IO to quickly access files.

```
import duckdb
import pyarrow as pa
import tempfile
import pathlib
import pyarrow.parquet as pq
import pyarrow.dataset as ds
import pyarrow.compute as pc

# connect to an in-memory database
con = duckdb.connect()

my_arrow_table = pa.Table.from_pydict({'i': [1, 2, 3, 4],
                                       'j': ["one", "two", "three", "four"]})

# create example Parquet files and save in a folder
base_path = pathlib.Path(tempfile.gettempdir())
(base_path / "parquet_folder").mkdir(exist_ok = True)
pq.write_to_dataset(my_arrow_table, str(base_path / "parquet_folder"))

# link to Parquet files using an Arrow Dataset
my_arrow_dataset = ds.dataset(str(base_path / 'parquet_folder/'))

# define the filter to be applied while scanning
# equivalent to "WHERE i = 2"
scanner_filter = (pc.field("i") == pc.scalar(2))

arrow_scanner = ds.Scanner.from_dataset(my_arrow_dataset, filter = scanner_filter)

# query the Apache Arrow scanner "arrow_scanner" and return as an Arrow Table
results = con.execute("SELECT * FROM arrow_scanner").arrow()
```

Apache Arrow RecordBatchReaders

Arrow RecordBatchReaders are a reader for Arrow's streaming binary format and can also be queried directly as if they were tables. This streaming format is useful when sending Arrow data for tasks like interprocess communication or communicating between language runtimes.

```
import duckdb
import pyarrow as pa

# connect to an in-memory database
con = duckdb.connect()

my_recordbatch = pa.RecordBatch.from_pydict({'i': [1, 2, 3, 4],
                                             'j': ["one", "two", "three", "four"]})

my_recordbatchreader = pa.ipc.RecordBatchReader.from_batches(my_recordbatch.schema, [my_recordbatch])

# query the Apache Arrow RecordBatchReader "my_recordbatchreader" and return as an Arrow Table
results = con.execute("SELECT * FROM my_recordbatchreader WHERE i = 2").arrow()
```

Import from Apache Arrow

`CREATE TABLE AS` and `INSERT INTO` can be used to create a table from any query. We can then create tables or insert into existing tables by referring to the Apache Arrow object in the query. This example imports from an Arrow Table, but DuckDB can query different

Apache Arrow formats as seen in the SQL on Arrow guide.

```
import duckdb
import pyarrow as pa

# connect to an in-memory database
my_arrow = pa.Table.from_pydict({'a': [42]})

# create the table "my_table" from the DataFrame "my_arrow"
duckdb.sql("CREATE TABLE my_table AS SELECT * FROM my_arrow")

# insert into the table "my_table" from the DataFrame "my_arrow"
duckdb.sql("INSERT INTO my_table SELECT * FROM my_arrow")
```

Export to Apache Arrow

All results of a query can be exported to an Apache Arrow Table using the `arrow` function. Alternatively, results can be returned as a RecordBatchReader using the `fetch_record_batch` function and results can be read one batch at a time. In addition, relations built using DuckDB's Relational API can also be exported.

Export to an Arrow Table

```
import duckdb
import pyarrow as pa

my_arrow_table = pa.Table.from_pydict({'i': [1, 2, 3, 4],
                                       'j': ["one", "two", "three", "four"]})

# query the Apache Arrow Table "my_arrow_table" and return as an Arrow Table
results = duckdb.sql("SELECT * FROM my_arrow_table").arrow()
```

Export as a RecordBatchReader

```
import duckdb
import pyarrow as pa

my_arrow_table = pa.Table.from_pydict({'i': [1, 2, 3, 4],
                                       'j': ["one", "two", "three", "four"]})

# query the Apache Arrow Table "my_arrow_table" and return as an Arrow RecordBatchReader
chunk_size = 1_000_000
results = duckdb.sql("SELECT * FROM my_arrow_table").fetch_record_batch(chunk_size)

# Loop through the results. A StopIteration exception is thrown when the RecordBatchReader is empty
while True:
    try:
        # Process a single chunk here (just printing as an example)
        print(results.read_next_batch().to_pandas())
    except StopIteration:
        print('Already fetched all batches')
        break
```

Export from Relational API

Arrow objects can also be exported from the Relational API. A relation can be converted to an Arrow table using the `arrow` or `to_arrow_table` functions, or a record batch using `record_batch`. A result can be exported to an Arrow table with `arrow` or the alias `fetch_arrow_table`, or to a RecordBatchReader using `fetch_arrow_reader`.

```
import duckdb

# connect to an in-memory database
con = duckdb.connect()

con.execute('CREATE TABLE integers (i integer)')
con.execute('INSERT INTO integers VALUES (0), (1), (2), (3), (4), (5), (6), (7), (8), (9), (NULL)')

# Create a relation from the table and export the entire relation as Arrow
rel = con.table("integers")
relation_as_arrow = rel.arrow() # or .to_arrow_table()

# Or, calculate a result using that relation and export that result to Arrow
res = rel.aggregate("sum(i)").execute()
result_as_arrow = res.arrow() # or fetch_arrow_table()
```

Relational API on Pandas

DuckDB offers a relational API that can be used to chain together query operations. These are lazily evaluated so that DuckDB can optimize their execution. These operators can act on Pandas DataFrames, DuckDB tables or views (which can point to any underlying storage format that DuckDB can read, such as CSV or Parquet files, etc.). Here we show a simple example of reading from a Pandas DataFrame and returning a DataFrame.

```
import duckdb
import pandas

# connect to an in-memory database
con = duckdb.connect()

input_df = pandas.DataFrame.from_dict({'i': [1, 2, 3, 4],
                                       'j': ["one", "two", "three", "four"]})

# create a DuckDB relation from a dataframe
rel = con.from_df(input_df)

# chain together relational operators (this is a lazy operation, so the operations are not yet executed)
# equivalent to: SELECT i, j, i*2 AS two_i FROM input_df WHERE i >= 2 ORDER BY i DESC LIMIT 2
transformed_rel = rel.filter('i >= 2').project('i, j, i*2 AS two_i').order('i DESC').limit(2)

# trigger execution by requesting .df() of the relation
# .df() could have been added to the end of the chain above - it was separated for clarity
output_df = transformed_rel.df()
```

Relational operators can also be used to group rows, aggregate, find distinct combinations of values, join, union, and more. They are also able to directly insert results into a DuckDB table or write to a CSV.

Please see these additional examples and the available relational methods on the `DuckDBPyRelation` class.

Multiple Python Threads

This page demonstrates how to simultaneously insert into and read from a DuckDB database across multiple Python threads. This could be useful in scenarios where new data is flowing in and an analysis should be periodically re-run. Note that this is all within a single Python process (see the FAQ for details on DuckDB concurrency). Feel free to follow along in this Google Colab notebook.

Setup

First, import DuckDB and several modules from the Python standard library. Note: if using Pandas, add `import pandas` at the top of the script as well (as it must be imported prior to the multi-threading). Then connect to a file-backed DuckDB database and create an example table to store inserted data. This table will track the name of the thread that completed the insert and automatically insert the timestamp when that insert occurred using the DEFAULT expression.

```
import duckdb
from threading import Thread, current_thread
import random

duckdb_con = duckdb.connect('my_peristent_db.duckdb')
# Use connect without parameters for an in-memory database
# duckdb_con = duckdb.connect()
duckdb_con.execute("""
    CREATE OR REPLACE TABLE my_inserts (
        thread_name VARCHAR,
        insert_time TIMESTAMP DEFAULT current_timestamp
    )
""")
```

Reader and Writer Functions

Next, define functions to be executed by the writer and reader threads. Each thread must use the `.cursor()` method to create a thread-local connection to the same DuckDB file based on the original connection. This approach also works with in-memory DuckDB databases.

```
def write_from_thread(duckdb_con):
    # Create a DuckDB connection specifically for this thread
    local_con = duckdb_con.cursor()
    # Insert a row with the name of the thread. insert_time is auto-generated.
    thread_name = str(current_thread().name)
    result = local_con.execute("""
        INSERT INTO my_inserts (thread_name)
        VALUES (?)
    """, (thread_name,)).fetchall()

def read_from_thread(duckdb_con):
    # Create a DuckDB connection specifically for this thread
    local_con = duckdb_con.cursor()
    # Query the current row count
    thread_name = str(current_thread().name)
    results = local_con.execute("""
        SELECT
            ? AS thread_name,
            count(*) AS row_counter,
            current_timestamp
        FROM my_inserts
    """, (thread_name,)).fetchall()
    print(results)
```

Create Threads

We define how many writers and readers to use, and define a list to track all of the threads that will be created. Then, create first writer and then reader threads. Next, shuffle them so that they will be kicked off in a random order to simulate simultaneous writers and readers. Note that the threads have not yet been executed, only defined.

```
write_thread_count = 50
read_thread_count = 5
threads = []

# Create multiple writer and reader threads (in the same process)
# Pass in the same connection as an argument
for i in range(write_thread_count):
    threads.append(Thread(target = write_from_thread,
                          args = (duckdb_con,),
                          name = 'write_thread_' + str(i)))

for j in range(read_thread_count):
    threads.append(Thread(target = read_from_thread,
                          args = (duckdb_con,),
                          name = 'read_thread_' + str(j)))

# Shuffle the threads to simulate a mix of readers and writers
random.seed(6) # Set the seed to ensure consistent results when testing
random.shuffle(threads)
```

Run Threads and Show Results

Now, kick off all threads to run in parallel, then wait for all of them to finish before printing out the results. Note that the timestamps of readers and writers are interspersed as expected due to the randomization.

```
# Kick off all threads in parallel
for thread in threads:
    thread.start()

# Ensure all threads complete before printing final results
for thread in threads:
    thread.join()

print(duckdb_con.execute("""
    SELECT *
    FROM my_inserts
    ORDER BY
        insert_time
""").df())
```

Integration with Ibis

Ibis is a Python dataframe library that supports 20+ backends, with DuckDB as the default. Ibis with DuckDB provides a Pythonic interface for SQL with great performance.

Installation

You can pip install Ibis with the DuckDB backend:

```
pip install 'ibis-framework[duckdb,examples]' # examples is only required to access the sample data Ibis provides
```

or use conda:

```
conda install ibis-framework
```

or use mamba:

```
mamba install ibis-framework
```

Create a Database File

Ibis can work with several file types, but at its core, it connects to existing databases and interacts with the data there. You can get started with your own DuckDB databases or create a new one with example data.

```
import ibis

con = ibis.connect("duckdb://penguins.ddb")
con.create_table(
    "penguins", ibis.examples.penguins.fetch().to_pyarrow(), overwrite = True
)

# Output:
DatabaseTable: penguins
  species           string
  island            string
  bill_length_mm    float64
  bill_depth_mm     float64
  flipper_length_mm int64
  body_mass_g       int64
  sex               string
  year              int64
```

You can now see the example dataset copied over to the database:

```
# reconnect to the persisted database (dropping temp tables)
con = ibis.connect("duckdb://penguins.ddb")
con.list_tables()

# Output:
['penguins']
```

There's one table, called `penguins`. We can ask Ibis to give us an object that we can interact with.

```
penguins = con.table("penguins")
penguins

# Output:
DatabaseTable: penguins
  species           string
  island            string
  bill_length_mm    float64
  bill_depth_mm     float64
  flipper_length_mm int64
  body_mass_g       int64
  sex               string
  year              int64
```

Ibis is lazily evaluated, so instead of seeing the data, we see the schema of the table. To peek at the data, we can call `head` and then `to_pandas` to get the first few rows of the table as a pandas DataFrame.

```
penguins.head().to_pandas()
```

```
  species     island  bill_length_mm  bill_depth_mm  flipper_length_mm  body_mass_g     sex  year
0  Adelie  Torgersen            39.1           18.7              181.0       3750.0    male  2007
1  Adelie  Torgersen            39.5           17.4              186.0       3800.0  female  2007
2  Adelie  Torgersen            40.3           18.0              195.0       3250.0  female  2007
3  Adelie  Torgersen             NaN            NaN                NaN          NaN    None  2007
4  Adelie  Torgersen            36.7           19.3              193.0       3450.0  female  2007
```

`to_pandas` takes the existing lazy table expression and evaluates it. If we leave it off, you'll see the Ibis representation of the table expression that `to_pandas` will evaluate (when you're ready!).

```
penguins.head()
```

```
# Output:
r0 := DatabaseTable: penguins
  species           string
  island            string
  bill_length_mm    float64
  bill_depth_mm     float64
  flipper_length_mm int64
  body_mass_g       int64
  sex               string
  year              int64

Limit[r0, n=5]
```

Ibis returns results as a pandas DataFrame using `to_pandas`, but isn't using pandas to perform any of the computation. The query is executed by DuckDB. Only when `to_pandas` is called does Ibis then pull back the results and convert them into a DataFrame.

Interactive Mode

For the rest of this intro, we'll turn on interactive mode, which partially executes queries to give users a preview of the results. There is a small difference in the way the output is formatted, but otherwise this is the same as calling `to_pandas` on the table expression with a limit of 10 result rows returned.

```
ibis.options.interactive = True
penguins.head()
```

species	island	bill_length_mm	bill_depth_mm	flipper_length_mm	body_mass_g	sex	year
string	string	float64	float64	int64	int64	string	int64
Adelie	Torgersen	39.1	18.7	181	3750	male	2007
Adelie	Torgersen	39.5	17.4	186	3800	female	2007
Adelie	Torgersen	40.3	18.0	195	3250	female	2007
Adelie	Torgersen	nan	nan	NULL	NULL	NULL	2007
Adelie	Torgersen	36.7	19.3	193	3450	female	2007

Common Operations

Ibis has a collection of useful table methods to manipulate and query the data in a table.

filter

`filter` allows you to select rows based on a condition or set of conditions.

We can filter so we only have penguins of the species Adelie:

```
penguins.filter(penguins.species == "Gentoo")
```

species	island	bill_length_mm	bill_depth_mm	flipper_length_mm	body_mass_g	sex	year
string	string	float64	float64	int64	int64	string	int64
Gentoo	Biscoe	46.1	13.2	211	4500	female	2007
Gentoo	Biscoe	50.0	16.3	230	5700	male	2007
Gentoo	Biscoe	48.7	14.1	210	4450	female	2007
Gentoo	Biscoe	50.0	15.2	218	5700	male	2007
Gentoo	Biscoe	47.6	14.5	215	5400	male	2007
Gentoo	Biscoe	46.5	13.5	210	4550	female	2007
Gentoo	Biscoe	45.4	14.6	211	4800	female	2007
Gentoo	Biscoe	46.7	15.3	219	5200	male	2007
Gentoo	Biscoe	43.3	13.4	209	4400	female	2007
Gentoo	Biscoe	46.8	15.4	215	5150	male	2007
…	…	…	…	…	…	…	…

Or filter for Gentoo penguins that have a body mass larger than 6 kg.

```
penguins.filter((penguins.species == "Gentoo") & (penguins.body_mass_g > 6000))
```

species	island	bill_length_mm	bill_depth_mm	flipper_length_mm	body_mass_g	sex	year
string	string	float64	float64	int64	int64	string	int64
Gentoo	Biscoe	49.2	15.2	221	6300	male	2007
Gentoo	Biscoe	59.6	17.0	230	6050	male	2007

You can use any Boolean comparison in a filter (although if you try to do something like use < on a string, Ibis will yell at you).

select

Your data analysis might not require all the columns present in a given table. `select` lets you pick out only those columns that you want to work with.

To select a column you can use the name of the column as a string:

```
penguins.select("species", "island", "year").limit(3)
```

species	island	year
string	string	int64
Adelie	Torgersen	2007
Adelie	Torgersen	2007
Adelie	Torgersen	2007
…	…	…

Or you can use column objects directly (this can be convenient when paired with tab-completion):

```
penguins.select(penguins.species, penguins.island, penguins.year).limit(3)
```

species	island	year
string	string	int64
Adelie	Torgersen	2007
Adelie	Torgersen	2007
Adelie	Torgersen	2007
…	…	…

Or you can mix-and-match:

```
penguins.select("species", "island", penguins.year).limit(3)
```

species	island	year
string	string	int64
Adelie	Torgersen	2007
Adelie	Torgersen	2007
Adelie	Torgersen	2007
…	…	…

mutate

`mutate` lets you add new columns to your table, derived from the values of existing columns.

```
penguins.mutate(bill_length_cm=penguins.bill_length_mm / 10)
```

species	island	bill_length_mm	bill_depth_mm	flipper_length_mm	body_mass_g	sex	year	bill_length_cm
string	string	float64	float64	int64	int64	string	int64	float64
Adelie	Torgersen	39.1	18.7	181	3750	male	2007	3.91
Adelie	Torgersen	39.5	17.4	186	3800	female	2007	3.95
Adelie	Torgersen	40.3	18.0	195	3250	female	2007	4.03
Adelie	Torgersen	nan	nan	NULL	NULL	NULL	2007	nan
Adelie	Torgersen	36.7	19.3	193	3450	female	2007	3.67
Adelie	Torgersen	39.3	20.6	190	3650	male	2007	3.93
Adelie	Torgersen	38.9	17.8	181	3625	female	2007	3.89
Adelie	Torgersen	39.2	19.6	195	4675	male	2007	3.92
Adelie	Torgersen	34.1	18.1	193	3475	NULL	2007	3.41

```
│ Adelie  │ Torgersen │          42.0 │          20.2 │               190 │        4250 │ NULL   │  2007 │
4.20 │
│ …       │ …         │             … │             … │                 … │           … │ …      │     … │
… │
```

Notice that the table is a little too wide to display all the columns now (depending on your screen-size). `bill_length` is now present in millimeters *and* centimeters. Use a `select` to trim down the number of columns we're looking at.

```
penguins.mutate(bill_length_cm=penguins.bill_length_mm / 10).select(
    "species",
    "island",
    "bill_depth_mm",
    "flipper_length_mm",
    "body_mass_g",
    "sex",
    "year",
    "bill_length_cm",
)
```

```
┌─────────┬───────────┬───────────────┬───────────────────┬─────────────┬────────┬───────┬──────────────┐
│ species │ island    │ bill_depth_mm │ flipper_length_mm │ body_mass_g │ sex    │ year  │ bill_length_cm
│
├─────────┼───────────┼───────────────┼───────────────────┼─────────────┼────────┼───────┼──────────────┤
│ string  │ string    │ float64       │ int64             │ int64       │ string │ int64 │ float64      │
├─────────┼───────────┼───────────────┼───────────────────┼─────────────┼────────┼───────┼──────────────┤
│ Adelie  │ Torgersen │          18.7 │               181 │        3750 │ male   │  2007 │         3.91 │
│ Adelie  │ Torgersen │          17.4 │               186 │        3800 │ female │  2007 │         3.95 │
│ Adelie  │ Torgersen │          18.0 │               195 │        3250 │ female │  2007 │         4.03 │
│ Adelie  │ Torgersen │           nan │              NULL │        NULL │ NULL   │  2007 │          nan │
│ Adelie  │ Torgersen │          19.3 │               193 │        3450 │ female │  2007 │         3.67 │
│ Adelie  │ Torgersen │          20.6 │               190 │        3650 │ male   │  2007 │         3.93 │
│ Adelie  │ Torgersen │          17.8 │               181 │        3625 │ female │  2007 │         3.89 │
│ Adelie  │ Torgersen │          19.6 │               195 │        4675 │ male   │  2007 │         3.92 │
│ Adelie  │ Torgersen │          18.1 │               193 │        3475 │ NULL   │  2007 │         3.41 │
│ Adelie  │ Torgersen │          20.2 │               190 │        4250 │ NULL   │  2007 │         4.20 │
│ …       │ …         │             … │                 … │           … │ …      │     … │            … │
└─────────┴───────────┴───────────────┴───────────────────┴─────────────┴────────┴───────┴──────────────┘
```

selectors

Typing out *all* of the column names *except* one is a little annoying. Instead of doing that again, we can use a `selector` to quickly select or deselect groups of columns.

```
import ibis.selectors as s

penguins.mutate(bill_length_cm=penguins.bill_length_mm / 10).select(
    ~s.matches("bill_length_mm")
    # match every column except `bill_length_mm`
)
```

```
┌─────────┬───────────┬───────────────┬───────────────────┬─────────────┬────────┬───────┬──────────────┐
│ species │ island    │ bill_depth_mm │ flipper_length_mm │ body_mass_g │ sex    │ year  │ bill_length_cm
│
├─────────┼───────────┼───────────────┼───────────────────┼─────────────┼────────┼───────┼──────────────┤
│ string  │ string    │ float64       │ int64             │ int64       │ string │ int64 │ float64      │
├─────────┼───────────┼───────────────┼───────────────────┼─────────────┼────────┼───────┼──────────────┤
│ Adelie  │ Torgersen │          18.7 │               181 │        3750 │ male   │  2007 │         3.91 │
│ Adelie  │ Torgersen │          17.4 │               186 │        3800 │ female │  2007 │         3.95 │
│ Adelie  │ Torgersen │          18.0 │               195 │        3250 │ female │  2007 │         4.03 │
└─────────┴───────────┴───────────────┴───────────────────┴─────────────┴────────┴───────┴──────────────┘
```

Adelie	Torgersen	nan	NULL	NULL	NULL	2007	nan
Adelie	Torgersen	19.3	193	3450	female	2007	3.67
Adelie	Torgersen	20.6	190	3650	male	2007	3.93
Adelie	Torgersen	17.8	181	3625	female	2007	3.89
Adelie	Torgersen	19.6	195	4675	male	2007	3.92
Adelie	Torgersen	18.1	193	3475	NULL	2007	3.41
Adelie	Torgersen	20.2	190	4250	NULL	2007	4.20
…	…	…	…	…	…	…	…

You can also use a `selector` alongside a column name.

```
penguins.select("island", s.numeric())
```

island	bill_length_mm	bill_depth_mm	flipper_length_mm	body_mass_g	year
string	float64	float64	int64	int64	int64
Torgersen	39.1	18.7	181	3750	2007
Torgersen	39.5	17.4	186	3800	2007
Torgersen	40.3	18.0	195	3250	2007
Torgersen	nan	nan	NULL	NULL	2007
Torgersen	36.7	19.3	193	3450	2007
Torgersen	39.3	20.6	190	3650	2007
Torgersen	38.9	17.8	181	3625	2007
Torgersen	39.2	19.6	195	4675	2007
Torgersen	34.1	18.1	193	3475	2007
Torgersen	42.0	20.2	190	4250	2007
…	…	…	…	…	…

You can read more about `selectors` in the docs!

order_by

`order_by` arranges the values of one or more columns in ascending or descending order.

By default, `ibis` sorts in ascending order:

```
penguins.order_by(penguins.flipper_length_mm).select(
    "species", "island", "flipper_length_mm"
)
```

species	island	flipper_length_mm
string	string	int64
Adelie	Biscoe	172
Adelie	Biscoe	174
Adelie	Torgersen	176
Adelie	Dream	178
Adelie	Dream	178
Adelie	Dream	178
Chinstrap	Dream	178
Adelie	Dream	179
Adelie	Torgersen	180
Adelie	Biscoe	180
…	…	…

You can sort in descending order using the `desc` method of a column:

```
penguins.order_by(penguins.flipper_length_mm.desc()).select(
    "species", "island", "flipper_length_mm"
)
```

species	island	flipper_length_mm
string	string	int64
Gentoo	Biscoe	231
Gentoo	Biscoe	230
Gentoo	Biscoe	230
Gentoo	Biscoe	230
Gentoo	Biscoe	230
Gentoo	Biscoe	230
Gentoo	Biscoe	230
Gentoo	Biscoe	230
Gentoo	Biscoe	229
Gentoo	Biscoe	229
…	…	…

Or you can use `ibis.desc`

```
penguins.order_by(ibis.desc("flipper_length_mm")).select(
    "species", "island", "flipper_length_mm"
)
```

species	island	flipper_length_mm
string	string	int64
Gentoo	Biscoe	231
Gentoo	Biscoe	230
Gentoo	Biscoe	230
Gentoo	Biscoe	230
Gentoo	Biscoe	230
Gentoo	Biscoe	230
Gentoo	Biscoe	230
Gentoo	Biscoe	230
Gentoo	Biscoe	229
Gentoo	Biscoe	229
…	…	…

aggregate

Ibis has several aggregate functions available to help summarize data.

`mean`, `max`, `min`, `count`, `sum` (the list goes on).

To aggregate an entire column, call the corresponding method on that column.

```
penguins.flipper_length_mm.mean()
```

```
# Output:
200.91520467836258
```

You can compute multiple aggregates at once using the `aggregate` method:

```
penguins.aggregate([penguins.flipper_length_mm.mean(), penguins.bill_depth_mm.max()])
```

Mean(flipper_length_mm)	Max(bill_depth_mm)
float64	float64
200.915205	21.5

But `aggregate` *really* shines when it's paired with `group_by`.

group_by

`group_by` creates groupings of rows that have the same value for one or more columns.

But it doesn't do much on its own -- you can pair it with `aggregate` to get a result.

```
penguins.group_by("species").aggregate()
```

species
string
Adelie Gentoo Chinstrap

We grouped by the `species` column and handed it an "empty" aggregate command. The result of that is a column of the unique values in the `species` column.

If we add a second column to the `group_by`, we'll get each unique pairing of the values in those columns.

```
penguins.group_by(["species", "island"]).aggregate()
```

species	island
string	string
Adelie	Torgersen
Adelie	Biscoe
Adelie	Dream
Gentoo	Biscoe
Chinstrap	Dream

Now, if we add an aggregation function to that, we start to really open things up.

```
penguins.group_by(["species", "island"]).aggregate(penguins.bill_length_mm.mean())
```

species	island	Mean(bill_length_mm)
string	string	float64
Adelie	Torgersen	38.950980
Adelie	Biscoe	38.975000
Adelie	Dream	38.501786
Gentoo	Biscoe	47.504878

Chinstrap	Dream	48.833824

By adding that `mean` to the `aggregate`, we now have a concise way to calculate aggregates over each of the distinct groups in the `group_by`. And we can calculate as many aggregates as we need.

```
penguins.group_by(["species", "island"]).aggregate(
    [penguins.bill_length_mm.mean(), penguins.flipper_length_mm.max()]
)
```

species	island	Mean(bill_length_mm)	Max(flipper_length_mm)
string	string	float64	int64
Adelie	Torgersen	38.950980	210
Adelie	Biscoe	38.975000	203
Adelie	Dream	38.501786	208
Gentoo	Biscoe	47.504878	231
Chinstrap	Dream	48.833824	212

If we need more specific groups, we can add to the `group_by`.

```
penguins.group_by(["species", "island", "sex"]).aggregate(
    [penguins.bill_length_mm.mean(), penguins.flipper_length_mm.max()]
)
```

species	island	sex	Mean(bill_length_mm)	Max(flipper_length_mm)
string	string	string	float64	int64
Adelie	Torgersen	male	40.586957	210
Adelie	Torgersen	female	37.554167	196
Adelie	Torgersen	NULL	37.925000	193
Adelie	Biscoe	female	37.359091	199
Adelie	Biscoe	male	40.590909	203
Adelie	Dream	female	36.911111	202
Adelie	Dream	male	40.071429	208
Adelie	Dream	NULL	37.500000	179
Gentoo	Biscoe	female	45.563793	222
Gentoo	Biscoe	male	49.473770	231
…	…	…	…	…

Chaining It All Together

We've already chained some Ibis calls together. We used `mutate` to create a new column and then `select` to only view a subset of the new table. We were just chaining `group_by` with `aggregate`.

There's nothing stopping us from putting all of these concepts together to ask questions of the data.

How about:

- What was the largest female penguin (by body mass) on each island in the year 2008?

```
penguins.filter((penguins.sex == "female") & (penguins.year == 2008)).group_by(
    ["island"]
).aggregate(penguins.body_mass_g.max())
```

island	Max(body_mass_g)
string	int64
Biscoe	5200
Torgersen	3800
Dream	3900

- What about the largest male penguin (by body mass) on each island for each year of data collection?

```
penguins.filter(penguins.sex == "male").group_by(["island", "year"]).aggregate(
    penguins.body_mass_g.max().name("max_body_mass")
).order_by(["year", "max_body_mass"])
```

island	year	max_body_mass
string	int64	int64
Dream	2007	4650
Torgersen	2007	4675
Biscoe	2007	6300
Torgersen	2008	4700
Dream	2008	4800
Biscoe	2008	6000
Torgersen	2009	4300
Dream	2009	4475
Biscoe	2009	6000

Learn More

That's all for this quick-start guide. If you want to learn more, check out the Ibis documentation.

Integration with Polars

Polars is a DataFrames library built in Rust with bindings for Python and Node.js. It uses Apache Arrow's columnar format as its memory model. DuckDB can read Polars DataFrames and convert query results to Polars DataFrames. It does this internally using the efficient Apache Arrow integration. Note that the `pyarrow` library must be installed for the integration to work.

Installation

```
pip install -U duckdb 'polars[pyarrow]'
```

Polars to DuckDB

DuckDB can natively query Polars DataFrames by referring to the name of Polars DataFrames as they exist in the current scope.

```
import duckdb
import polars as pl
```

```
df = pl.DataFrame(
    {
        "A": [1, 2, 3, 4, 5],
        "fruits": ["banana", "banana", "apple", "apple", "banana"],
        "B": [5, 4, 3, 2, 1],
        "cars": ["beetle", "audi", "beetle", "beetle", "beetle"],
    }
)
duckdb.sql("SELECT * FROM df").show()
```

DuckDB to Polars

DuckDB can output results as Polars DataFrames using the `.pl()` result-conversion method.

```
df = duckdb.sql("""
    SELECT 1 AS id, 'banana' AS fruit
    UNION ALL
    SELECT 2, 'apple'
    UNION ALL
    SELECT 3, 'mango'""""
).pl()
print(df)
```

```
shape: (3, 2)
```

id	fruit
i32	str
1	banana
2	apple
3	mango

To learn more about Polars, feel free to explore their Python API Reference.

Using fsspec Filesystems

DuckDB support for `fsspec` filesystems allows querying data in filesystems that DuckDB's `httpfs` extension does not support. `fsspec` has a large number of inbuilt filesystems, and there are also many external implementations. This capability is only available in DuckDB's Python client because `fsspec` is a Python library, while the `httpfs` extension is available in many DuckDB clients.

Example

The following is an example of using `fsspec` to query a file in Google Cloud Storage (instead of using their S3-compatible API).

Firstly, you must install `duckdb` and `fsspec`, and a filesystem interface of your choice.

```
pip install duckdb fsspec gcsfs
```

Then, you can register whichever filesystem you'd like to query:

```
import duckdb
from fsspec import filesystem

# this line will throw an exception if the appropriate filesystem interface is not installed
```

```
duckdb.register_filesystem(filesystem('gcs'))

duckdb.sql("SELECT * FROM read_csv('gcs:///bucket/file.csv')")
```

> These filesystems are not implemented in C++, hence, their performance may not be comparable to the ones provided by the `httpfs` extension. It is also worth noting that as they are third-party libraries, they may contain bugs that are beyond our control.

SQL Editors

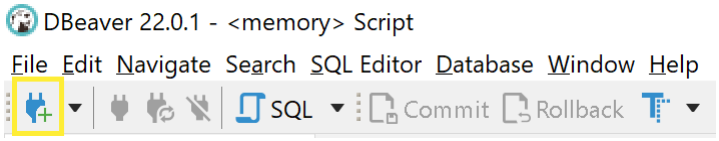
DBeaver SQL IDE

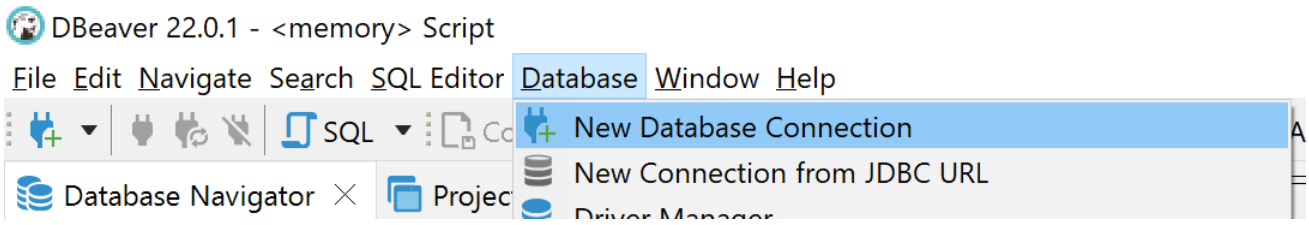
DBeaver is a powerful and popular desktop sql editor and integrated development environment (IDE). It has both an open source and enterprise version. It is useful for visually inspecting the available tables in DuckDB and for quickly building complex queries. DuckDB's JDBC connector allows DBeaver to query DuckDB files, and by extension, any other files that DuckDB can access (like Parquet files).

Installing DBeaver

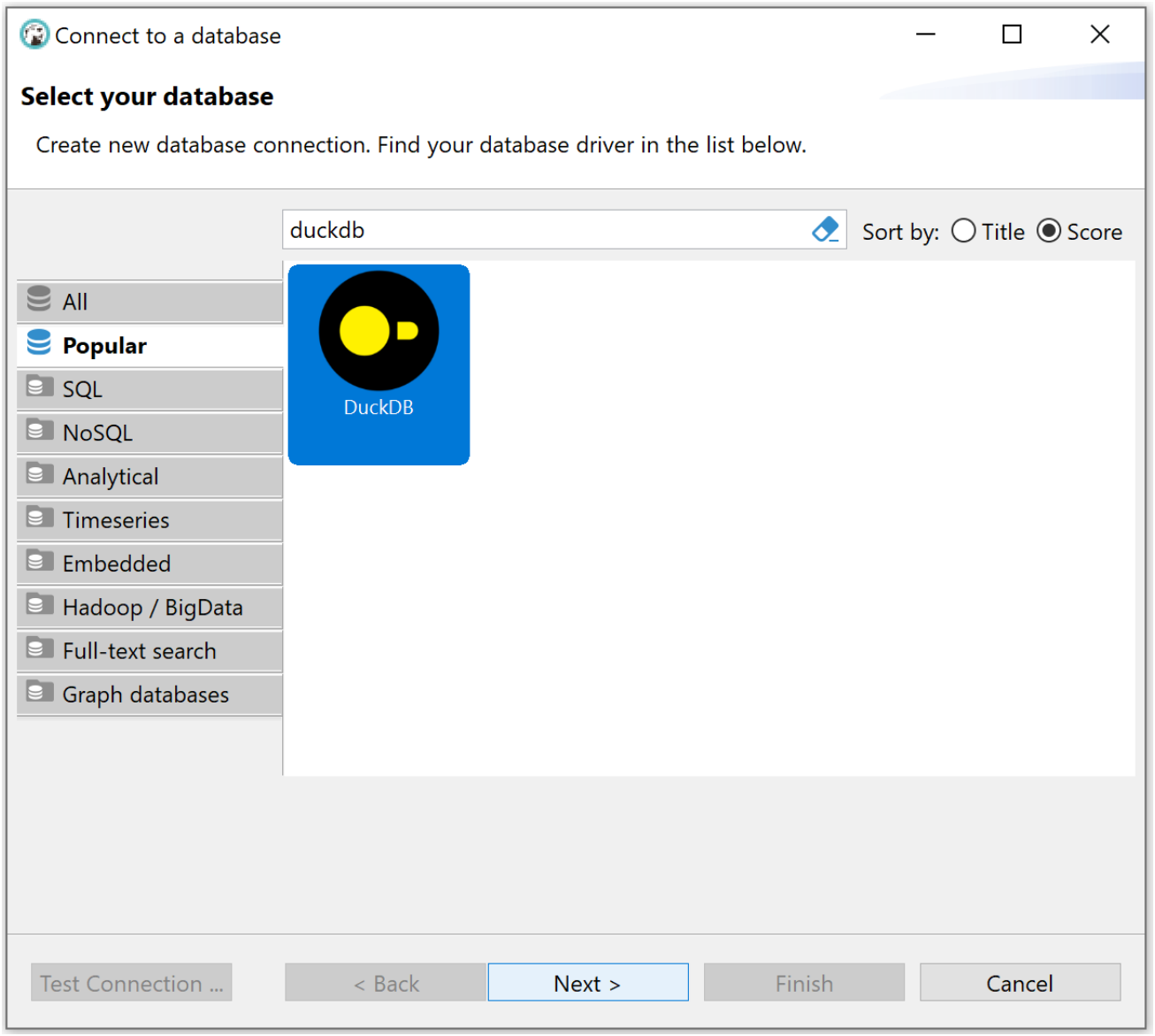
1. Install DBeaver using the download links and instructions found at their download page.

2. Open DBeaver and create a new connection. Either click on the "New Database Connection" button or go to Database > New Database Connection in the menu bar.

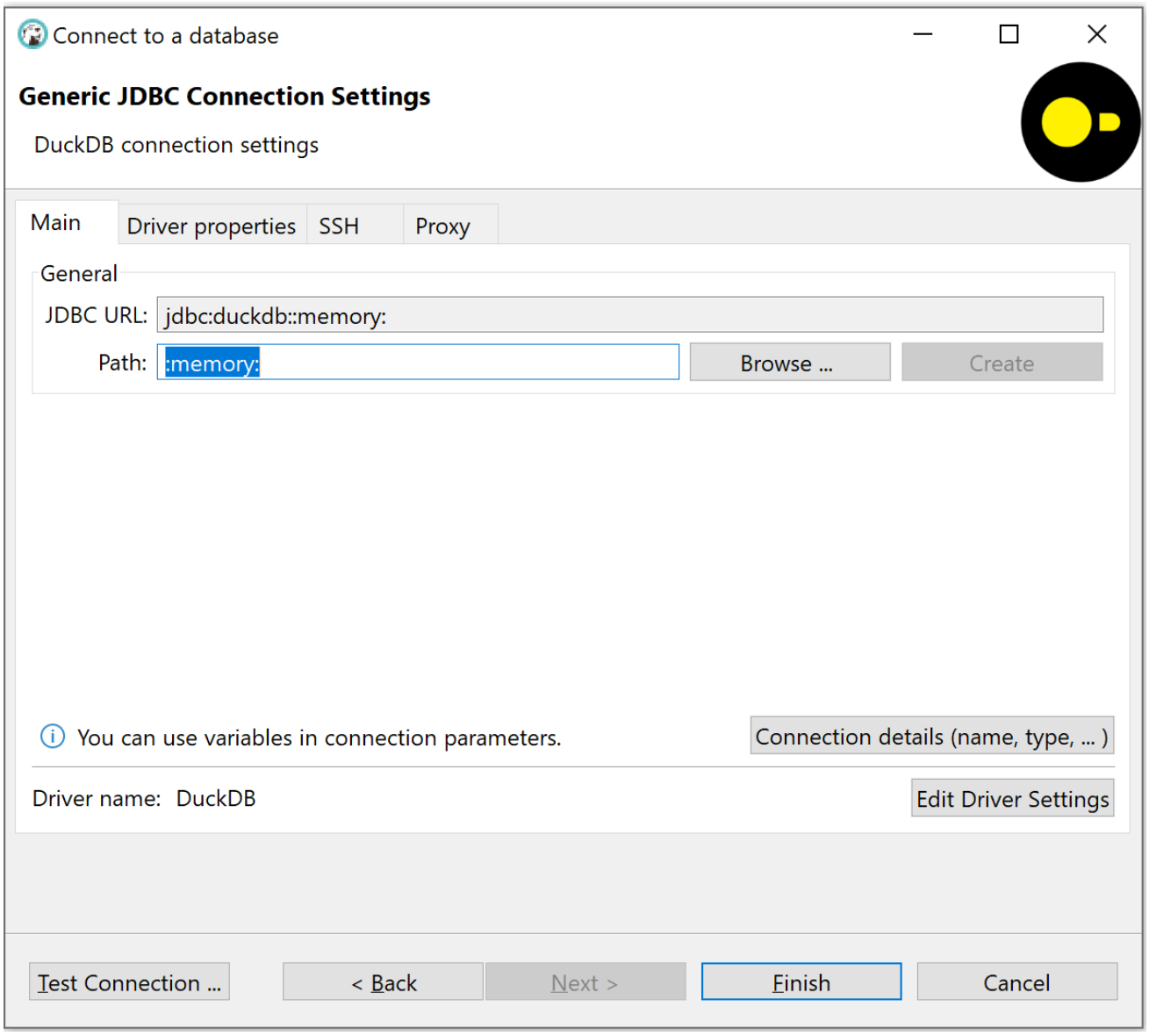




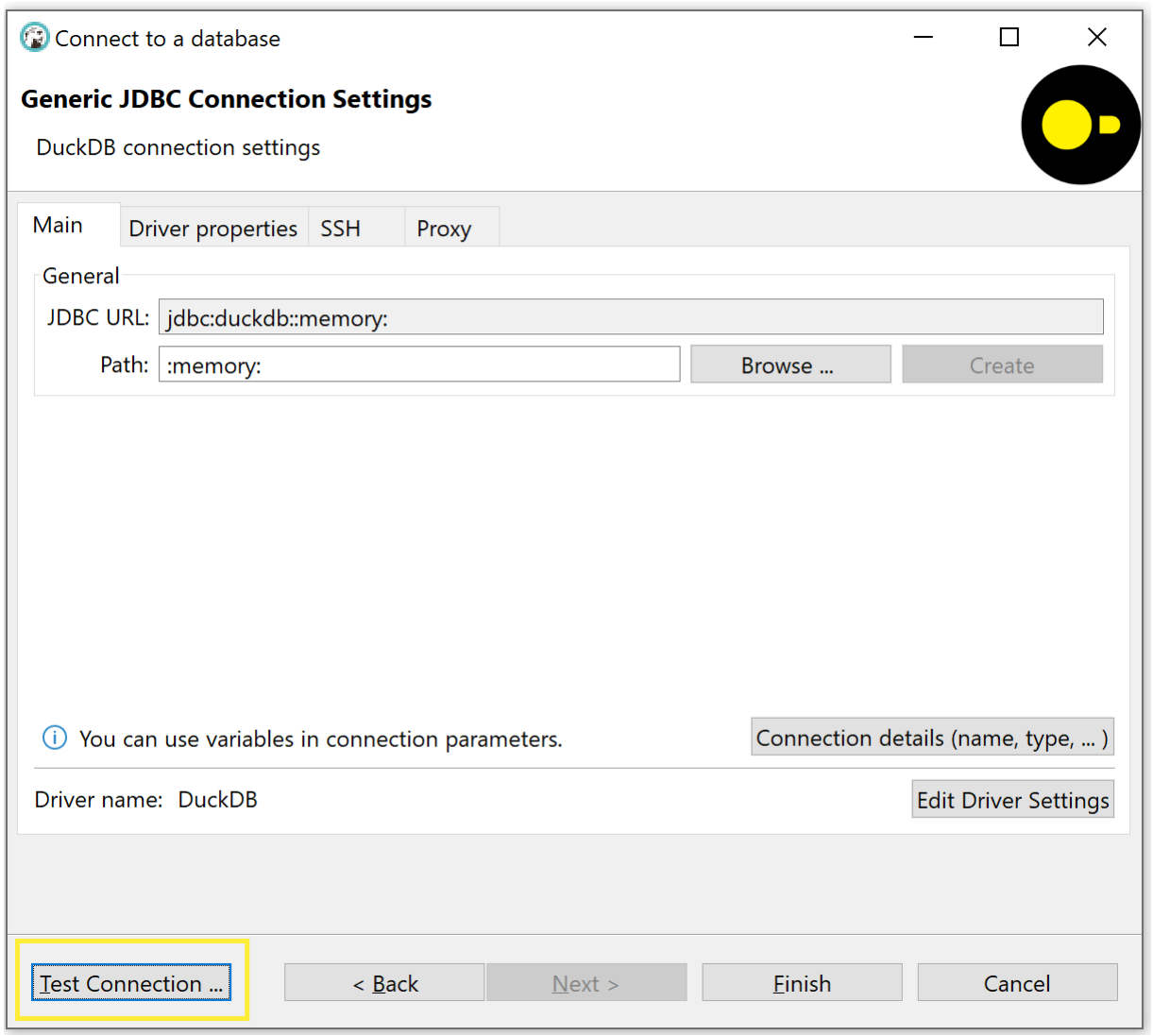
3. Search for DuckDB, select it, and click Next.



4. Enter the path or browse to the DuckDB database file you wish to query. To use an in-memory DuckDB (useful primarily if just interested in querying Parquet files, or for testing) enter `:memory:` as the path.

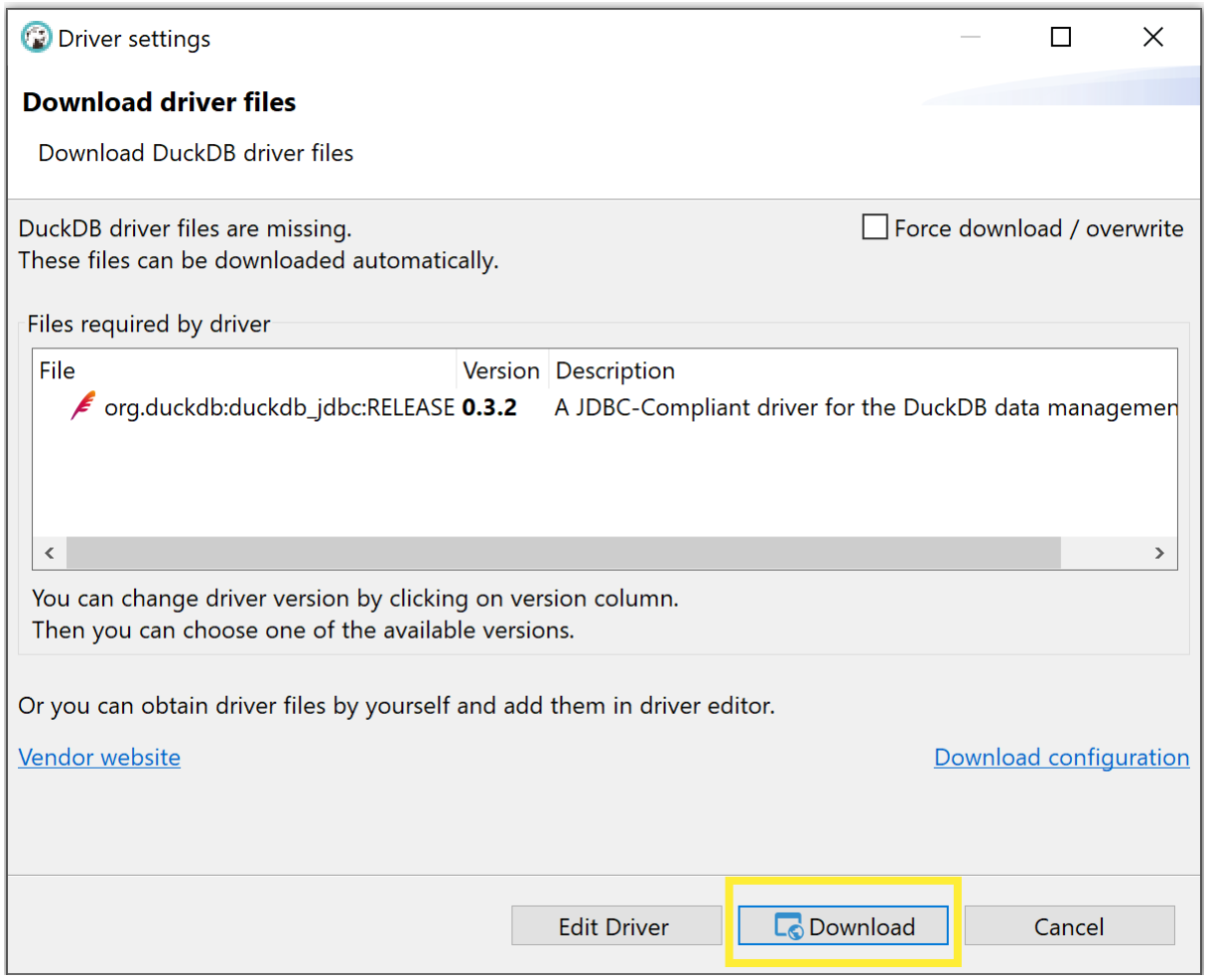


5. Click “Test Connection”. This will then prompt you to install the DuckDB JDBC driver. If you are not prompted, see alternative driver installation instructions below.

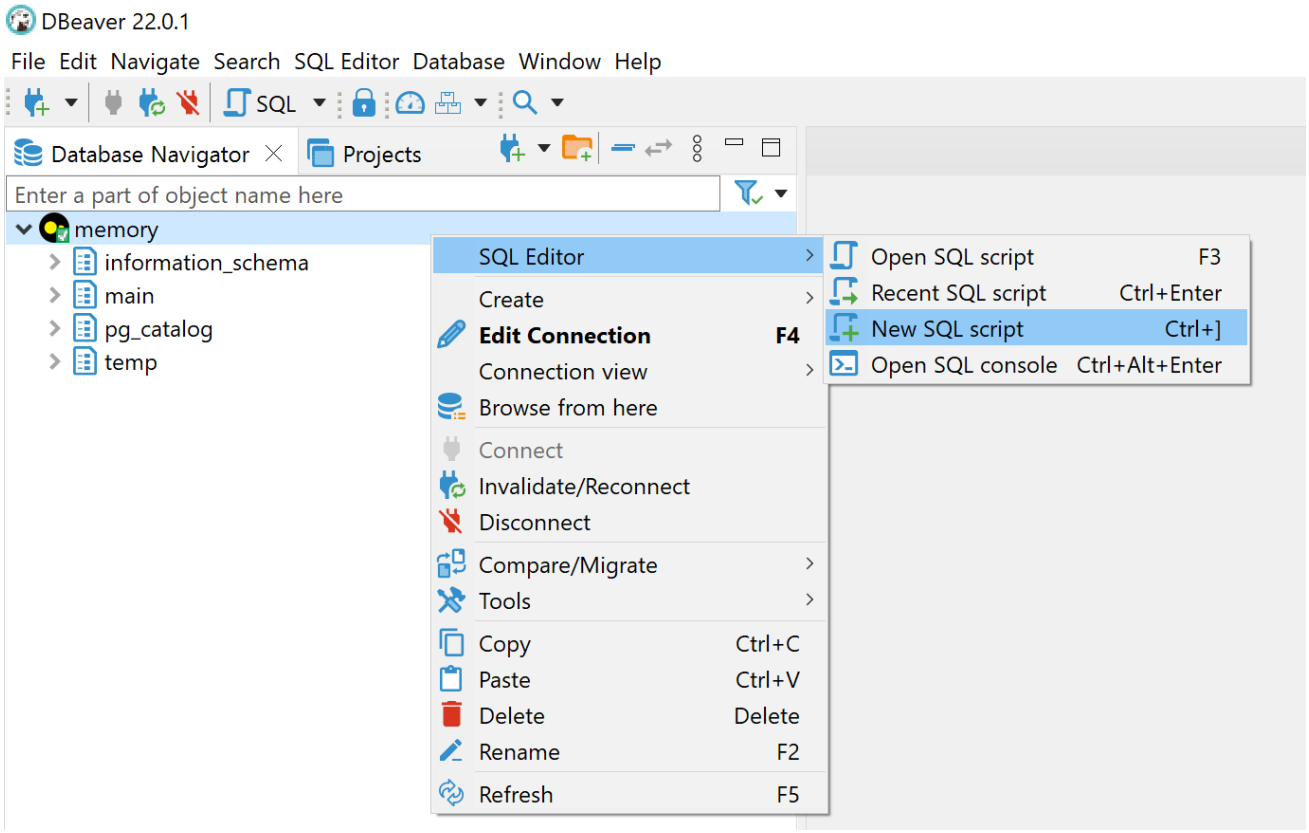


6. Click “Download” to download DuckDB's JDBC driver from Maven. Once download is complete, click “OK”, then click “Finish”.

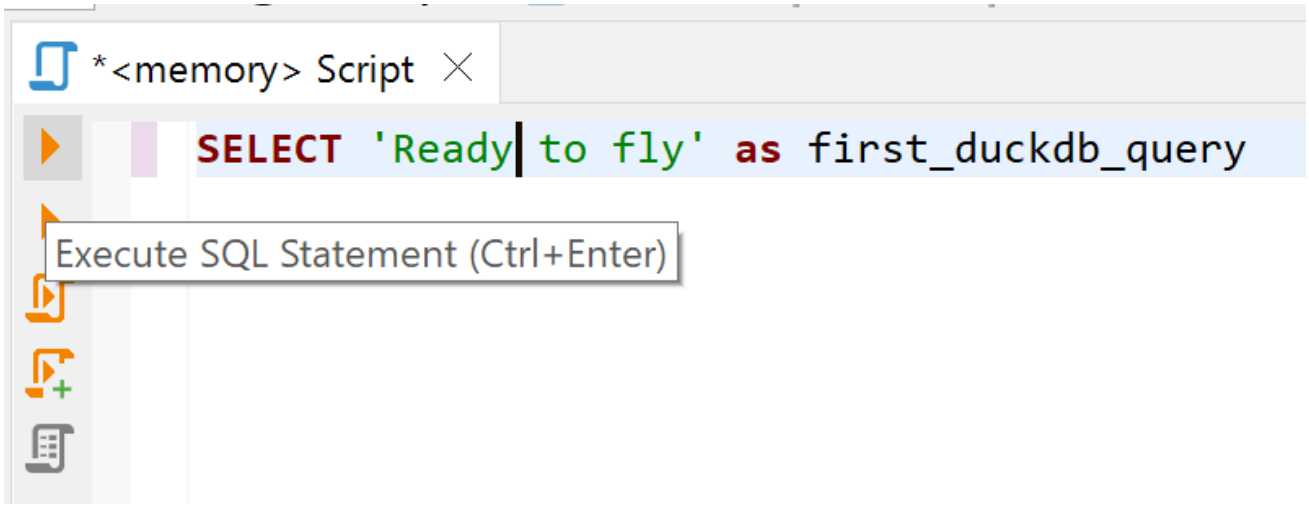
- Note: If you are in a corporate environment or behind a firewall, before clicking download, click the “Download Configuration” link to configure your proxy settings.



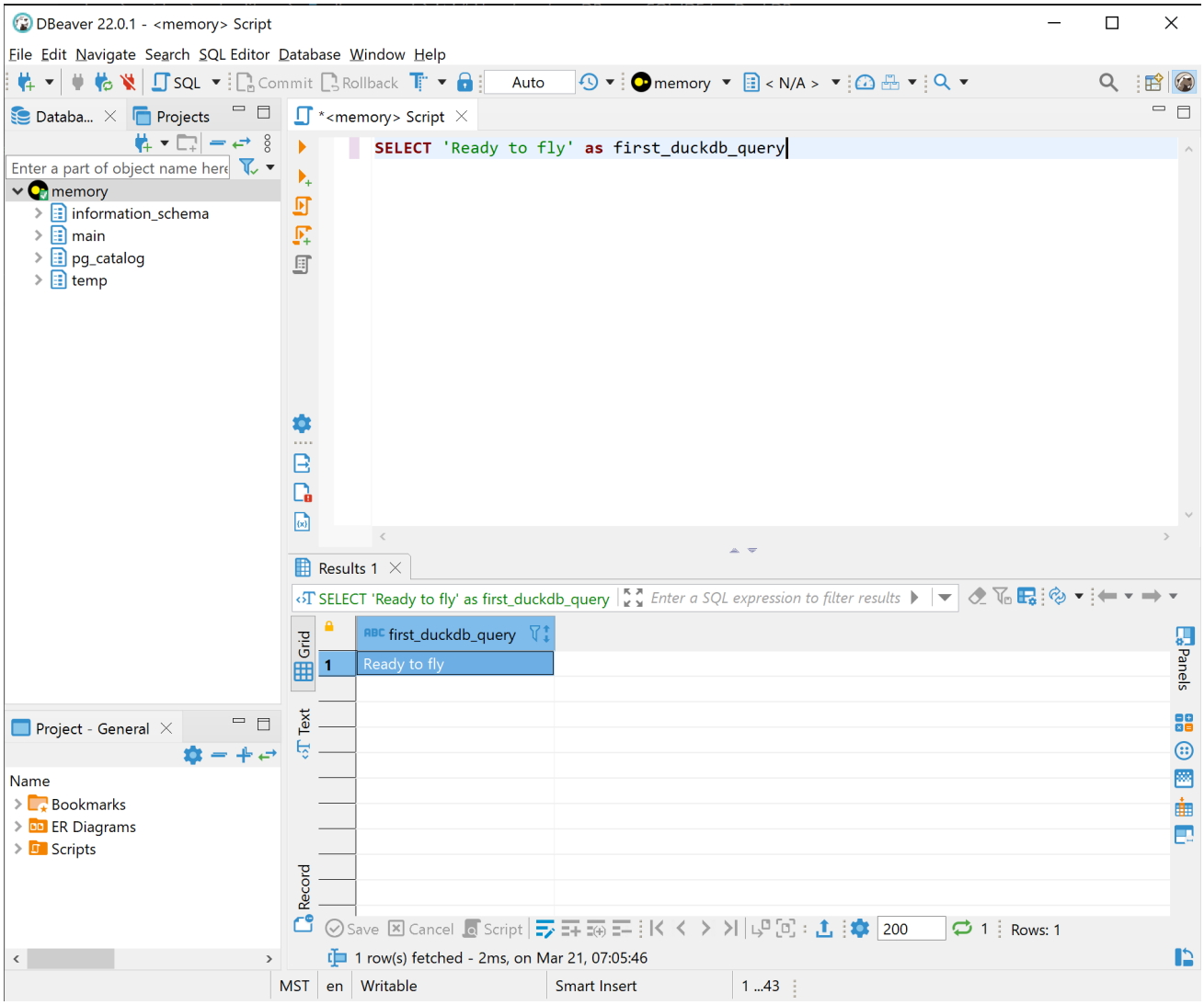
1. You should now see a database connection to your DuckDB database in the left hand “Database Navigator” pane. Expand it to see the tables and views in your database. Right click on that connection and create a new SQL script.



2. Write some SQL and click the “Execute” button.

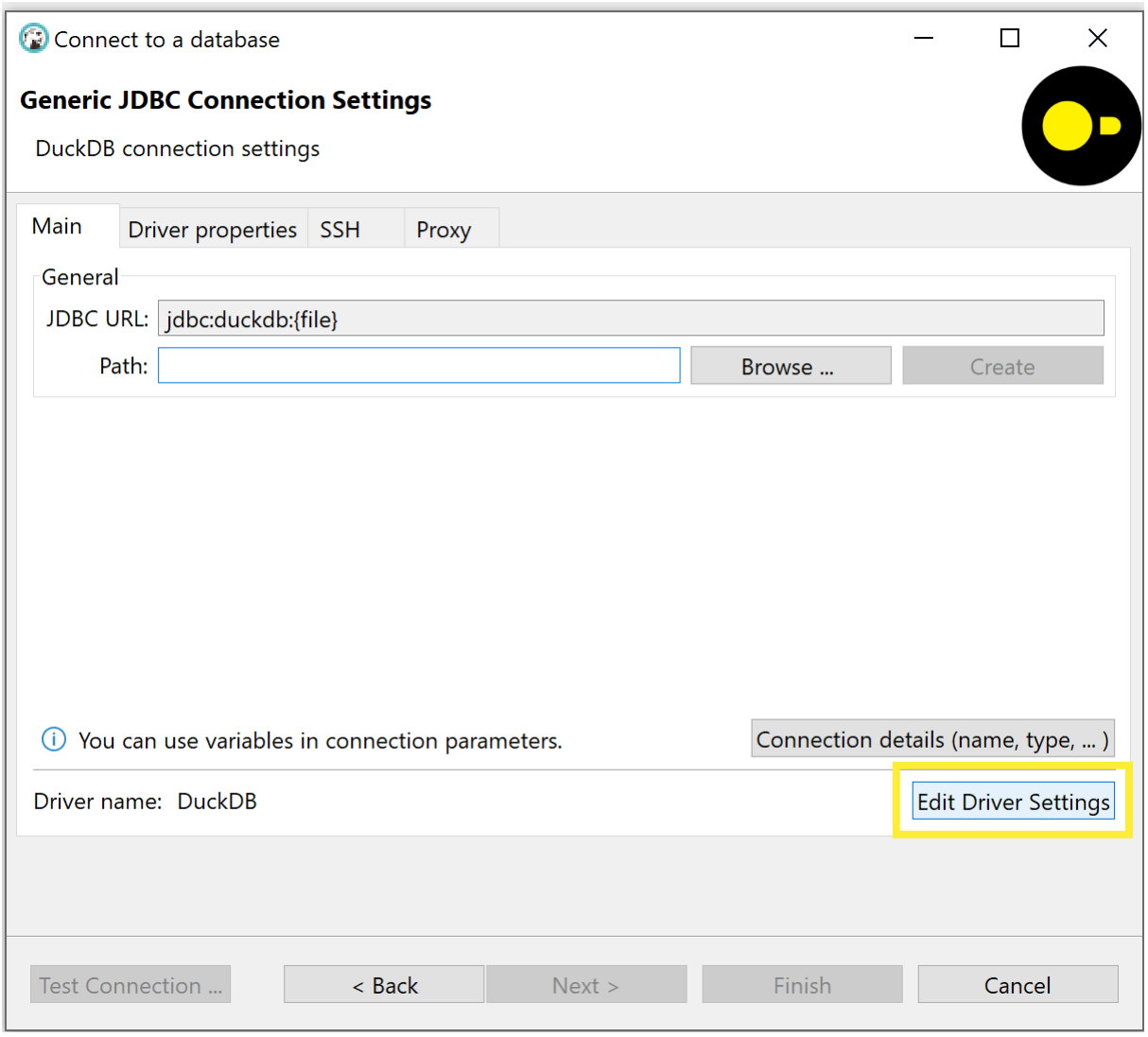


3. Now you're ready to fly with DuckDB and DBeaver!

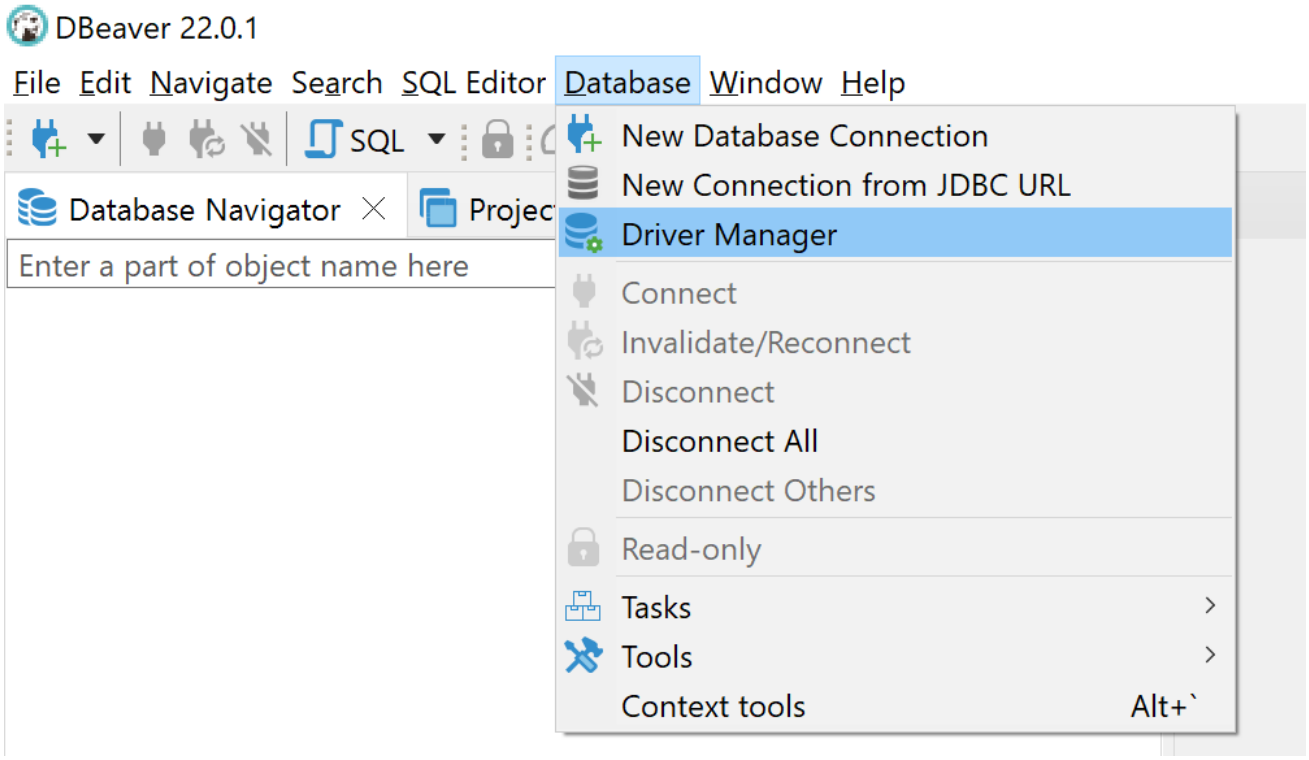


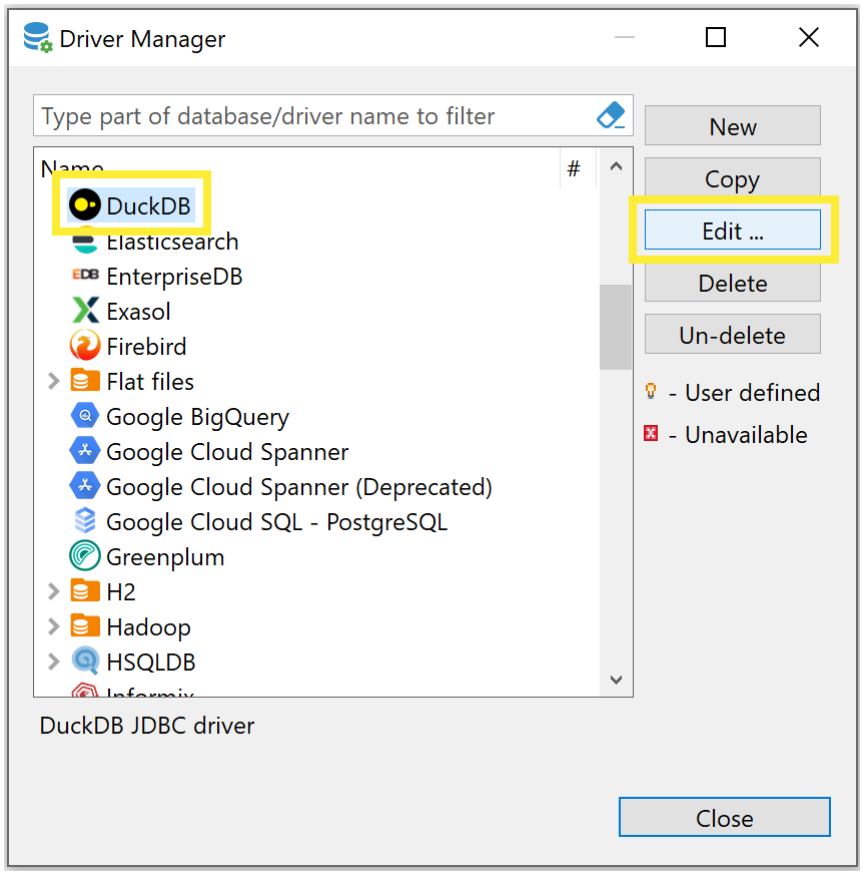
Alternative Driver Installation

1. If not prompted to install the DuckDB driver when testing your connection, return to the “Connect to a database” dialog and click “Edit Driver Settings”.

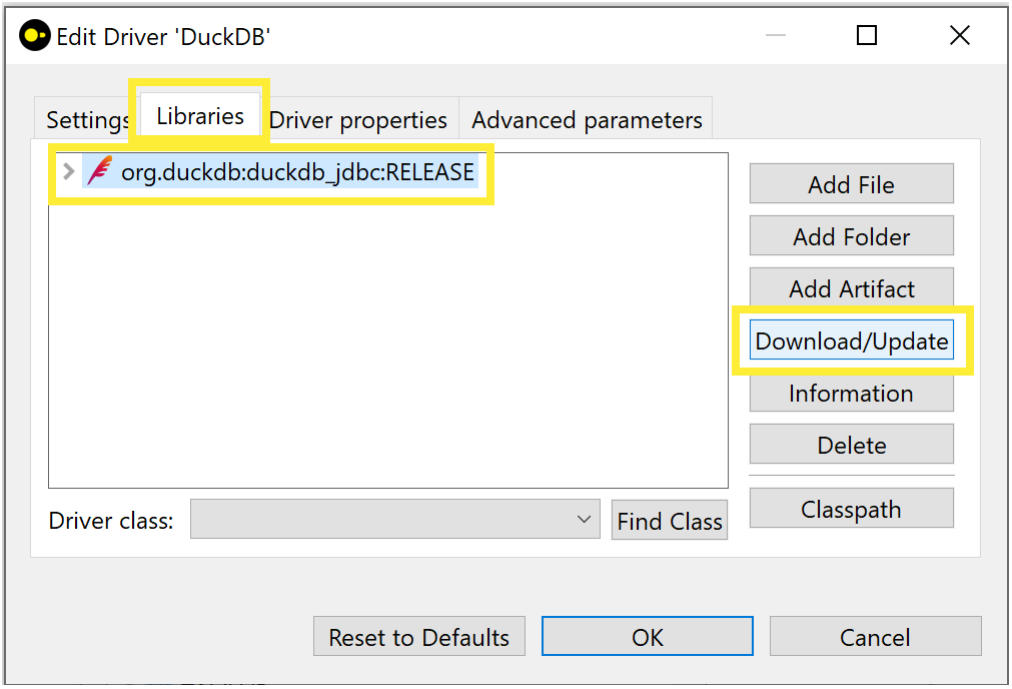


2. (Alternate) You may also access the driver settings menu by returning to the main DBeaver window and clicking Database > Driver Manager in the menu bar. Then select DuckDB, then click Edit.



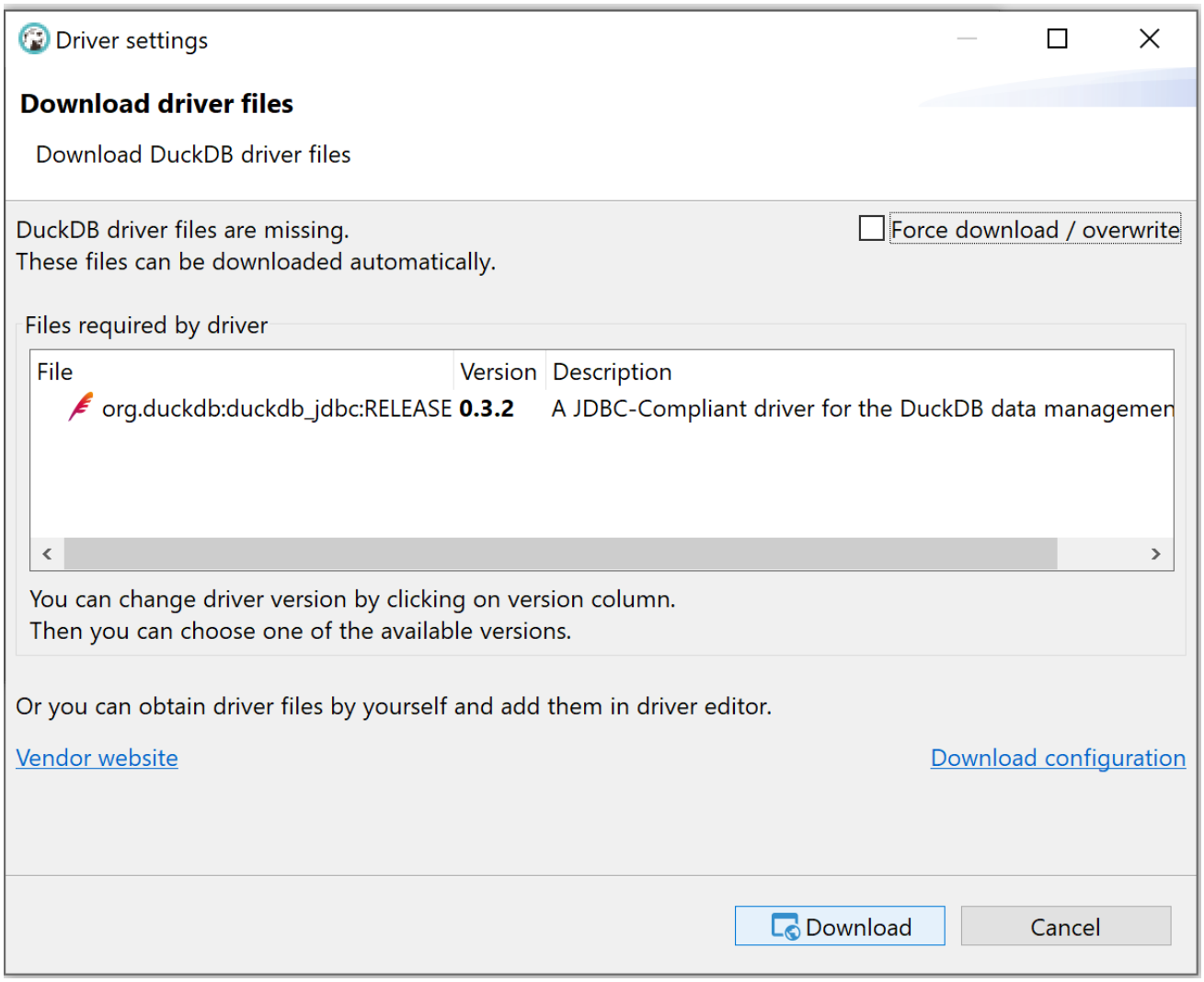


3. Go to the “Libraries” tab, then click on the DuckDB driver and click “Download/Update”. If you do not see the DuckDB driver, first click on “Reset to Defaults”.



4. Click “Download” to download DuckDB's JDBC driver from Maven. Once download is complete, click “OK”, then return to the main DBeaver window and continue with step 7 above.

- Note: If you are in a corporate environment or behind a firewall, before clicking download, click the “Download Configuration” link to configure your proxy settings.


Driver settings
Download driver files
Download DuckDB driver files
DuckDB driver files are missing.
These files can be downloaded automatically.
Force download / overwrite
Files required by driver
File
Version
Description
org.duckdb:duckdb_jdbc:RELEASE
0.3.2
A JDBC-Compliant driver for the DuckDB data managemen
You can change driver version by clicking on version column.
Then you can choose one of the available versions.
Or you can obtain driver files by yourself and add them in driver editor.
Vendor website
Download configuration
Download
Cancel

SQL Features

AsOf Join

What is an AsOf Join?

Time series data is not always perfectly aligned. Clocks may be slightly off, or there may be a delay between cause and effect. This can make connecting two sets of ordered data challenging. AsOf joins are a tool for solving this and other similar problems.

One of the problems that AsOf joins are used to solve is finding the value of a varying property at a specific point in time. This use case is so common that it is where the name came from:

*Give me the value of the property **as of this time***.

More generally, however, AsOf joins embody some common temporal analytic semantics, which can be cumbersome and slow to implement in standard SQL.

Portfolio Example Data Set

Let's start with a concrete example. Suppose we have a table of stock `prices` with timestamps:

ticker	when	price
APPL	2001-01-01 00:00:00	1
APPL	2001-01-01 00:01:00	2
APPL	2001-01-01 00:02:00	3
MSFT	2001-01-01 00:00:00	1
MSFT	2001-01-01 00:01:00	2
MSFT	2001-01-01 00:02:00	3
GOOG	2001-01-01 00:00:00	1
GOOG	2001-01-01 00:01:00	2
GOOG	2001-01-01 00:02:00	3

We have another table containing portfolio `holdings` at various points in time:

ticker	when	shares
APPL	2000-12-31 23:59:30	5.16
APPL	2001-01-01 00:00:30	2.94
APPL	2001-01-01 00:01:30	24.13
GOOG	2000-12-31 23:59:30	9.33
GOOG	2001-01-01 00:00:30	23.45

ticker	when	shares
GOOG	2001-01-01 00:01:30	10.58
DATA	2000-12-31 23:59:30	6.65
DATA	2001-01-01 00:00:30	17.95
DATA	2001-01-01 00:01:30	18.37

To load these tables to DuckDB, run:

```
CREATE TABLE prices AS FROM 'https://duckdb.org/data/prices.csv';
CREATE TABLE holdings AS FROM 'https://duckdb.org/data/holdings.csv';
```

Inner AsOf Joins

We can compute the value of each holding at that point in time by finding the most recent price before the holding's timestamp by using an AsOf Join:

```
SELECT h.ticker, h.when, price * shares AS value
FROM holdings h
ASOF JOIN prices p
       ON h.ticker = p.ticker
      AND h.when >= p.when;
```

This attaches the value of the holding at that time to each row:

ticker	when	value
APPL	2001-01-01 00:00:30	2.94
APPL	2001-01-01 00:01:30	48.26
GOOG	2001-01-01 00:00:30	23.45
GOOG	2001-01-01 00:01:30	21.16

It essentially executes a function defined by looking up nearby values in the `prices` table. Note also that missing `ticker` values do not have a match and don't appear in the output.

Outer AsOf Joins

Because AsOf produces at most one match from the right hand side, the left side table will not grow as a result of the join, but it could shrink if there are missing times on the right. To handle this situation, you can use an *outer* AsOf Join:

```
SELECT h.ticker, h.when, price * shares AS value
FROM holdings h
ASOF LEFT JOIN prices p
           ON h.ticker = p.ticker
          AND h.when >= p.when
ORDER BY ALL;
```

As you might expect, this will produce `NULL` prices and values instead of dropping left side rows when there is no ticker or the time is before the prices begin.

ticker	when	value
APPL	2000-12-31 23:59:30	
APPL	2001-01-01 00:00:30	2.94
APPL	2001-01-01 00:01:30	48.26
GOOG	2000-12-31 23:59:30	
GOOG	2001-01-01 00:00:30	23.45
GOOG	2001-01-01 00:01:30	21.16
DATA	2000-12-31 23:59:30	
DATA	2001-01-01 00:00:30	
DATA	2001-01-01 00:01:30	

AsOf Joins with the USING Keyword

So far we have been explicit about specifying the conditions for AsOf, but SQL also has a simplified join condition syntax for the common case where the column names are the same in both tables. This syntax uses the USING keyword to list the fields that should be compared for equality. AsOf also supports this syntax, but with two restrictions:

- The last field is the inequality
- The inequality is >= (the most common case)

Our first query can then be written as:

```
SELECT ticker, h.when, price * shares AS value
FROM holdings h
ASOF JOIN prices p USING (ticker, "when");
```

Clarification on Column Selection with USING in ASOF Joins

When you use the USING keyword in a join, the columns specified in the USING clause are merged in the result set. This means that if you run:

```
SELECT *
FROM holdings h
ASOF JOIN prices p USING (ticker, "when");
```

You will get back only the columns `h.ticker, h.when, h.shares, p.price`. The columns `ticker` and when will appear only once, with `ticker` and when coming from the left table (holdings).

This behavior is fine for the `ticker` column because the value is the same in both tables. However, for the when column, the values might differ between the two tables due to the >= condition used in the AsOf join. The AsOf join is designed to match each row in the left table (`holdings`) with the nearest preceding row in the right table (`prices`) based on the when column.

If you want to retrieve the when column from both tables to see both timestamps, you need to list the columns explicitly rather than relying on *, like so:

```
SELECT h.ticker, h.when AS holdings_when, p.when AS prices_when, h.shares, p.price
FROM holdings h
ASOF JOIN prices p USING (ticker, "when");
```

This ensures that you get the complete information from both tables, avoiding any potential confusion caused by the default behavior of the USING keyword.

See Also

For implementation details, see the blog post "DuckDB's AsOf joins: Fuzzy Temporal Lookups".

Full-Text Search

DuckDB supports full-text search via the `fts` extension. A full-text index allows for a query to quickly search for all occurrences of individual words within longer text strings.

Example: Shakespeare Corpus

Here's an example of building a full-text index of Shakespeare's plays.

```
CREATE TABLE corpus AS
    SELECT * FROM 'https://blobs.duckdb.org/data/shakespeare.parquet';
```

```
DESCRIBE corpus;
```

column_name	column_type	null	key	default	extra
line_id	VARCHAR	YES	NULL	NULL	NULL
play_name	VARCHAR	YES	NULL	NULL	NULL
line_number	VARCHAR	YES	NULL	NULL	NULL
speaker	VARCHAR	YES	NULL	NULL	NULL
text_entry	VARCHAR	YES	NULL	NULL	NULL

The text of each line is in `text_entry`, and a unique key for each line is in `line_id`.

Creating a Full-Text Search Index

First, we create the index, specifying the table name, the unique id column, and the column(s) to index. We will just index the single column `text_entry`, which contains the text of the lines in the play.

```
PRAGMA create_fts_index('corpus', 'line_id', 'text_entry');
```

The table is now ready to query using the Okapi BM25 ranking function. Rows with no match return a `NULL` score.

What does Shakespeare say about butter?

```
SELECT
    fts_main_corpus.match_bm25(line_id, 'butter') AS score,
    line_id, play_name, speaker, text_entry
FROM corpus
WHERE score IS NOT NULL
ORDER BY score DESC;
```

score	line_id	play_name	speaker	text_entry
4.427313429798464	H4/2.4.494	Henry IV	Carrier	As fat as butter.
3.836270302568675	H4/1.2.21	Henry IV	FALSTAFF	prologue to an egg and butter.

score	line_id	play_name	speaker	text_entry
3.836270302568675	H4/2.1.55	Henry IV	Chamberlain	They are up already, and call for eggs and butter;
3.3844488405497115	H4/4.2.21	Henry IV	FALSTAFF	toasts-and-butter, with hearts in their bellies no
3.3844488405497115	H4/4.2.62	Henry IV	PRINCE HENRY	already made thee butter. But tell me, Jack, whose
3.3844488405497115	AWW/4.1.40	Alls well that ends well	PAROLLES	butter-womans mouth and buy myself another of
3.3844488405497115	AYLI/3.2.93	As you like it	TOUCHSTONE	right butter-womens rank to market.
3.3844488405497115	KL/2.4.132	King Lear	Fool	kindness to his horse, buttered his hay.
3.0278411214953107	AWW/5.2.9	Alls well that ends well	Clown	henceforth eat no fish of fortunes buttering.
3.0278411214953107	MWW/2.2.260	Merry Wives of Windsor	FALSTAFF	Hang him, mechanical salt-butter rogue! I will
3.0278411214953107	MWW/2.2.284	Merry Wives of Windsor	FORD	rather trust a Fleming with my butter, Parson Hugh
3.0278411214953107	MWW/3.5.7	Merry Wives of Windsor	FALSTAFF	Ill have my brains taen out and buttered, and give
3.0278411214953107	MWW/3.5.102	Merry Wives of Windsor	FALSTAFF	to heat as butter; a man of continual dissolution
2.739219044070792	H4/2.4.115	Henry IV	PRINCE HENRY	Didst thou never see Titan kiss a dish of butter?

Unlike standard indexes, full-text indexes don't auto-update as the underlying data is changed, so you need to `PRAGMA drop_fts_index(my_fts_index)` and recreate it when appropriate.

Note on Generating the Corpus Table

For more details, see the "Generating a Shakespeare corpus for full-text searching from JSON" blog post.

- The Columns are: line_id, play_name, line_number, speaker, text_entry.
- We need a unique key for each row in order for full-text searching to work.
- The line_id `KL/2.4.132` means King Lear, Act 2, Scene 4, Line 132.

query and query_table Functions

The `query` and `query_table` functions take a string literal, and convert it into a `SELECT` subquery and a table reference, respectively. Note that these functions only accept literal strings. As such, they are not as powerful (or dangerous) as a generic `eval`.

These functions are conceptually simple, but enable powerful and more dynamic SQL. For example, they allow passing in a table name as a prepared statement parameter:

```
CREATE TABLE my_table (i INTEGER);
INSERT INTO my_table VALUES (42);

PREPARE select_from_table AS SELECT * FROM query_table($1);
EXECUTE select_from_table('my_table');
```

i
42

When combined with the `COLUMNS` expression, we can write very generic SQL-only macros. For example, below is a custom version of `SUMMARIZE` that computes the `min` and `max` of every column in a table:

```
CREATE OR REPLACE MACRO my_summarize(table_name) AS TABLE
SELECT
    unnest([*COLUMNS('alias_.*')]) AS column_name,
    unnest([*COLUMNS('min_.*')]) AS min_value,
    unnest([*COLUMNS('max_.*')]) AS max_value
FROM (
    SELECT
        any_value(alias(COLUMNS(*))) AS "alias_\0",
        min(COLUMNS(*))::VARCHAR AS "min_\0",
        max(COLUMNS(*))::VARCHAR AS "max_\0"
    FROM query_table(table_name::VARCHAR)
);

SELECT *
FROM my_summarize('https://blobs.duckdb.org/data/ontime.parquet')
LIMIT 3;
```

column_name	min_value	max_value
year	2017	2017
quarter	1	3
month	1	9

Snippets

Create Synthetic Data

DuckDB allows you to quickly generate synthetic data sets. To do so, you may use:

- range functions
- hash functions, e.g., `hash`, `md5`, `sha256`
- the Faker Python package via the Python function API
- using cross products (Cartesian products)

For example:

```
import duckdb

from duckdb.typing import *
from faker import Faker

fake = Faker()

def random_date():
    return fake.date_between()

def random_short_text():
    return fake.text(max_nb_chars=20)

def random_long_text():
    return fake.text(max_nb_chars=200)

con = duckdb.connect()
con.create_function("random_date",       random_date,       [], DATE,    type="native", side_effects=True)
con.create_function("random_short_text", random_short_text, [], VARCHAR, type="native", side_
effects=True)
con.create_function("random_long_text",  random_long_text,  [], VARCHAR, type="native", side_
effects=True)

res = con.sql("""
                SELECT
                   hash(i * 10 + j) AS id,
                   random_date() AS creationDate,
                   random_short_text() AS short,
                   random_long_text() AS long,
                   IF (j % 2, true, false) AS bool
                FROM generate_series(1, 5) s(i)
                CROSS JOIN generate_series(1, 2) t(j)
                """)
res.show()
```

This generates the following:

```
┌──────────────────────┬──────────────┬─────────┐
│          id          │ creationDate │  flag   │
│        uint64        │     date     │ boolean │
```

```
┌──────────────────────┬──────────────┬─────────┐
│  6770051751173734325 │ 2019-11-05   │ true    │
│ 16510940941872865459 │ 2002-08-03   │ true    │
│ 13285076694688170502 │ 1998-11-27   │ true    │
│ 11757770452869451863 │ 1998-07-03   │ true    │
│  2064835973596856015 │ 2010-09-06   │ true    │
│ 17776805813723356275 │ 2020-12-26   │ false   │
│ 13540103502347468651 │ 1998-03-21   │ false   │
│  4800297459639118879 │ 2015-06-12   │ false   │
│  7199933130570745587 │ 2005-04-13   │ false   │
│ 18103378254596719331 │ 2014-09-15   │ false   │
├──────────────────────┴──────────────┴─────────┤
│ 10 rows                         3 columns     │
└───────────────────────────────────────────────┘
```

Dutch Railway Datasets

Examples in this documentation often use datasets based on the Dutch Railway datasets. These high-quality datasets are maintained by the team behind the Rijden de Treinen *(Are the trains running?)* application. This page contains download links to our mirrors to the datasets.

Loading the Datasets

You can load the datasets directly as follows:

```
CREATE TABLE services AS
    FROM 'https://blobs.duckdb.org/nl-railway/services-2025-03.csv.gz';

DESCRIBE services;
```

column_name	column_type	null	key	default	extra
Service:RDT-ID	BIGINT	YES	NULL	NULL	NULL
Service:Date	DATE	YES	NULL	NULL	NULL
Service:Type	VARCHAR	YES	NULL	NULL	NULL
Service:Company	VARCHAR	YES	NULL	NULL	NULL
Service:Train number	BIGINT	YES	NULL	NULL	NULL
Service:Completely cancelled	BOOLEAN	YES	NULL	NULL	NULL
Service:Partly cancelled	BOOLEAN	YES	NULL	NULL	NULL
Service:Maximum delay	BIGINT	YES	NULL	NULL	NULL
Stop:RDT-ID	BIGINT	YES	NULL	NULL	NULL
Stop:Station code	VARCHAR	YES	NULL	NULL	NULL
Stop:Station name	VARCHAR	YES	NULL	NULL	NULL
Stop:Arrival time	TIMESTAMP WITH TIME ZONE	YES	NULL	NULL	NULL
Stop:Arrival delay	BIGINT	YES	NULL	NULL	NULL
Stop:Arrival cancelled	BOOLEAN	YES	NULL	NULL	NULL
Stop:Departure time	TIMESTAMP WITH TIME ZONE	YES	NULL	NULL	NULL
Stop:Departure delay	BIGINT	YES	NULL	NULL	NULL
Stop:Departure cancelled	BOOLEAN	YES	NULL	NULL	NULL

Datasets

Yearly Data Sets

- 2019 (347 MB)
- 2020 (355 MB)
- 2021 (350 MB)
- 2022 (355 MB)
- 2023 (346 MB)
- 2024 (357 MB)

Monthly Data Sets

- 2024-01 (29 MB)
- 2024-02 (28 MB)
- 2024-03 (30 MB)
- 2024-04 (28 MB)
- 2024-05 (30 MB)
- 2024-06 (29 MB)
- 2024-07 (30 MB)
- 2024-08 (29 MB)
- 2024-09 (29 MB)
- 2024-10 (30 MB)
- 2024-11 (29 MB)
- 2024-12 (29 MB)
- 2025-01 (30 MB)
- 2025-02 (28 MB)
- 2025-03 (30 MB)

Sharing Macros

DuckDB has a powerful macro mechanism that allows creating shorthands for common tasks.

Sharing a Scalar Macro

First, we defined a macro that pretty-prints a non-negative integer as a short string with thousands, millions, and billions (without rounding) as follows:

```
duckdb pretty_print_integer_macro.duckdb
```

```
CREATE MACRO pretty_print_integer(n) AS
    CASE
        WHEN n >= 1_000_000_000 THEN printf('%dB', n // 1_000_000_000)
        WHEN n >= 1_000_000     THEN printf('%dM', n // 1_000_000)
        WHEN n >= 1_000         THEN printf('%dk', n // 1_000)
        ELSE printf('%d', n)
    END;

SELECT pretty_print_integer(25_500_000) AS x;
```

x
varchar
25M

As one would expect, the macro gets persisted in the database. But this also means that we can host it on an HTTPS endpoint and share it with anyone! We have published this macro on `blobs.duckdb.org`.

You can try it from DuckDB:

```
duckdb
```

Make sure that the `httpfs` extension is installed:

```
INSTALL httpfs;
```

You can now attach to the remote endpoint and use the macro:

```
ATTACH 'https://blobs.duckdb.org/data/pretty_print_integer_macro.duckdb'
    AS pretty_print_macro_db;

SELECT pretty_print_macro_db.pretty_print_integer(42_123) AS x;
```

x
varchar
42k

Sharing a Table Macro

It's also possible to share table macros. For example, we created the `checksum` macro as follows:

```
duckdb compute_table_checksum.duckdb
```

```
CREATE MACRO checksum(table_name) AS TABLE
    SELECT bit_xor(md5_number(COLUMNS(*)::VARCHAR))
    FROM query_table(table_name);
```

To use it, make sure that the `httpfs` extension is installed:

```
INSTALL httpfs;
```

You can attach to the remote endpoint and use the macro:

```
ATTACH 'https://blobs.duckdb.org/data/compute_table_checksum.duckdb'
    AS compute_table_checksum_db;

CREATE TABLE stations AS
    FROM 'https://blobs.duckdb.org/stations.parquet';

.mode line
FROM compute_table_checksum_db.checksum('stations');
```

```
         id = -132780776949939723506211681506129908318
       code = 126327004005066229305810236187733612209
        uic = -145623335062491121476006068124745817380
 name_short = -114540917565721687000878144381189869683
name_medium = -568264780518431562127359918655305384
```

```
 name_long = 126079956280724674884063510870679874110
      slug = -53458800462031706622213217090663245511
   country = 143068442936912051858689770843609587944
      type = 56656623154707854561474006040888797 51
   geo_lat = 160608116135251821259126521573759502306
   geo_lng = -138297281072655463682926723171691547732
```

Analyzing a Git Repository

You can use DuckDB to analyze Git logs using the output of the `git log` command.

Exporting the Git Log

We start by picking a character that doesn't occur in any part of the commit log (author names, messages, etc). Since version v1.2.0, DuckDB's CSV reader supports 4-byte delimiters, making it possible to use emojis! ⊠

Despite being featured in the Emoji Movie (IMDb rating: 3.4), we can assume that the Fish Cake with Swirl emoji (🍥) is not a common occurrence in most Git logs. So, let's clone the `duckdb/duckdb` repository and export its log as follows:

```
git log --date=iso-strict --pretty=format:%ad🍥%h🍥%an🍥%s > git-log.csv
```

The resulting file looks like this:

```
2025-02-25T18:12:54+01:00🍥d608a31e13🍥Mark🍥MAIN_BRANCH_VERSIONING: Adopt also for Python build and
amalgamation (#16400)
2025-02-25T15:05:56+01:00🍥920b39ad96🍥Mark🍥Read support for Parquet Float16 (#16395)
2025-02-25T13:43:52+01:00🍥61f55734b9🍥Carlo Piovesan🍥MAIN_BRANCH_VERSIONING: Adopt also for Python build
and amalgamation
2025-02-25T12:35:28+01:00🍥87eff7ebd3🍥Mark🍥Fix issue #16377 (#16391)
2025-02-25T10:33:49+01:00🍥35af26476e🍥Hannes Mühleisen🍥Read support for Parquet Float16
```

Loading the Git Log into DuckDB

Start DuckDB and read the log as a CSV 🍥SV:

```sql
CREATE TABLE commits AS
    FROM read_csv(
            'git-log.csv',
            delim = '🍥',
            header = false,
            column_names = ['timestamp', 'hash', 'author', 'message']
        );
```

This will result in a nice DuckDB table:

```sql
FROM commits
LIMIT 5;
```

```
│      timestamp       │    hash    │       author       │                                    message
│      timestamp       │  varchar   │      varchar       │                                    varchar
├──────────────────────┼────────────┼────────────────────┼───────────────────────────────────────────────
│ 2025-02-25 17:12:54  │ d608a31e13 │ Mark               │ MAIN_BRANCH_VERSIONING: Adopt also for Python
build and amalgamation (#16400) │
```

```
│ 2025-02-25 14:05:56 │ 920b39ad96 │ Mark             │ Read support for Parquet Float16 (#16395)
│
│ 2025-02-25 12:43:52 │ 61f55734b9 │ Carlo Piovesan   │ MAIN_BRANCH_VERSIONING: Adopt also for Python
build and amalgamation           │
│ 2025-02-25 11:35:28 │ 87eff7ebd3 │ Mark             │ Fix issue #16377 (#16391)
│
│ 2025-02-25 09:33:49 │ 35af26476e │ Hannes Mühleisen │ Read support for Parquet Float16
│
└─────────────────────┴────────────┴──────────────────┴──────────────────────────────────────────────────
```

Analyzing the Log

We can analyze the table as any other in DuckDB.

Common Topics

Let's start with a simple question: which topic was the most commonly mentioned in the commit messages: CI, CLI, or Python?

```
SELECT
    message.lower().regexp_extract('\b(ci|cli|python)\b') AS topic,
    count(*) AS num_commits
FROM commits
WHERE topic <> ''
GROUP BY ALL
ORDER BY num_commits DESC;
```

```
┌─────────┬─────────────┐
│  topic  │ num_commits │
│ varchar │    int64    │
├─────────┼─────────────┤
│ ci      │         828 │
│ python  │         666 │
│ cli     │          49 │
└─────────┴─────────────┘
```

Out of these three topics, commits related to continuous integration dominate the log!

We can also do a more exploratory analysis by looking at all words in the commit messages. To do so, we first tokenize the messages:

```
CREATE TABLE words AS
    SELECT unnest(
        message
            .lower()
            .regexp_replace('\W', ' ')
            .trim(' ')
            .string_split_regex('\W')
        ) AS word
FROM commits;
```

Then, we remove stopwords using a pre-defined list:

```
CREATE TABLE stopwords AS
```

```
    SELECT unnest(['a', 'about', 'above', 'after', 'again', 'against', 'all', 'am', 'an', 'and', 'any',
    'are', 'as', 'at', 'be', 'because', 'been', 'before', 'being', 'below', 'between', 'both', 'but',
    'by', 'can', 'did', 'do', 'does', 'doing', 'don', 'down', 'during', 'each', 'few', 'for', 'from',
    'further', 'had', 'has', 'have', 'having', 'he', 'her', 'here', 'hers', 'herself', 'him', 'himself',
    'his', 'how', 'i', 'if', 'in', 'into', 'is', 'it', 'its', 'itself', 'just', 'me', 'more', 'most',
    'my', 'myself', 'no', 'nor', 'not', 'now', 'of', 'off', 'on', 'once', 'only', 'or', 'other', 'our',
    'ours', 'ourselves', 'out', 'over', 'own', 's', 'same', 'she', 'should', 'so', 'some', 'such', 't',
    'than', 'that', 'the', 'their', 'theirs', 'them', 'themselves', 'then', 'there', 'these', 'they',
    'this', 'those', 'through', 'to', 'too', 'under', 'until', 'up', 'very', 'was', 'we', 'were',
    'what', 'when', 'where', 'which', 'while', 'who', 'whom', 'why', 'will', 'with', 'you', 'your',
    'yours', 'yourself', 'yourselves']) AS word;

CREATE OR REPLACE TABLE words AS
    FROM words
    NATURAL ANTI JOIN stopwords
    WHERE word != '';
```

> We use the `NATURAL ANTI JOIN` clause here, which allows us to elegantly filter out values that occur in the `stopwords` table.

Finally, we select the top-20 most common words.

```
SELECT word, count(*) AS count FROM words
GROUP BY ALL
ORDER BY count DESC
LIMIT 20;
```

w varchar	count int64
merge	12550
fix	6402
branch	6005
pull	5950
request	5945
add	5687
test	3801
master	3289
tests	2339
issue	1971
main	1935
remove	1884
format	1819
duckdb	1710
use	1442
mytherin	1410
fixes	1333
hawkfish	1147
feature	1139
function	1088
20 rows	

As expected, there are many Git terms (`merge`, `branch`, `pull`, etc.), followed by terminology related to development (`fix`, `test`/`tests`, `issue`, `format`). We also see the account names of some developers (`mytherin`, `hawkfish`), which are likely there due to commit message for merging pull requests (e.g., "Merge pull request #13776 from Mytherin/expressiondepth"). Finally, we also see some DuckDB-related terms such as `duckdb` (shocking!) and `function`.

Visualizing the Number of Commits

Let's visualize the number of commits each year:

```
SELECT
    year(timestamp) AS year,
    count(*) AS num_commits,
    num_commits.bar(0, 20_000) AS num_commits_viz
FROM commits
GROUP BY ALL
ORDER BY ALL;
```

year int64	num_commits int64	num_commits_viz varchar
2018	870	████
2019	1621	████████
2020	3484	████████████████
2021	6488	██████████████████████████████
2022	9817	██
2023	14585	██
2024	15949	██
2025	1788	████████

We see a steady growth over the years – especially considering that many of DuckDB's functionalities and clients, which were originally part of the main repository, are now maintained in separate repositories (e.g., Java, R).

Happy hacking!

Importing Duckbox Tables

> The scripts provided in this page work on Linux, macOS, and WSL.

By default, the DuckDB CLI client renders query results in the duckbox format, which uses rich, ASCII-art inspired tables to show data. These tables are often shared verbatim in other documents. For example, take the table used to demonstrate new CSV features in the DuckDB v1.2.0 release blog post:

a varchar	b int64
hello	42
world	84

What if we would like to load this data back to DuckDB? This is not supported by default but it can be achieved by some scripting: we can turn the table into a `│`-separated file and read it with DuckDB's CSV reader. Note that the separator is not the pipe character `|`, instead it is the "Box Drawings Light Vertical" character `│`.

Loading Duckbox Tables to DuckDB

First, we save the table above as `duckbox.csv`. Then, we clean it using `sed`:

```
echo -n > duckbox-cleaned.csv
sed -n "2s/^│ *//;s/ *│$//;s/ *│ */│/p;2q" duckbox.csv >> duckbox-cleaned.csv
sed "1,4d;\$d;s/^│ *//;s/ *│$//;s/ *│ */│/g" duckbox.csv >> duckbox-cleaned.csv
```

The `duckbox-cleaned.csv` file looks as follows:

```
a|b
hello|42
world|84
```

We can then simply load this to DuckDB via:

```
FROM read_csv('duckbox-cleaned.csv', delim = '|');
```

And export it to a CSV:

```
COPY (FROM read_csv('duckbox-cleaned.csv', delim = '|')) TO 'out.csv';
```

```
a,b
hello,42
world,84
```

Using `shellfs`

To parse duckbox tables with a single `read_csv` call – and without creating any temporary files –, we can use the `shellfs` community extension:

```
INSTALL shellfs FROM community;
LOAD shellfs;
FROM read_csv(
        '(sed -n "2s/^| *//;s/ *|$//;s/ *| */|/p;2q" duckbox.csv; ' ||
        'sed "1,4d;\$d;s/^| *//;s/ *|$//;s/ *| */|/g" duckbox.csv) |',
        delim = '|'
    );
```

We can also create a table macro:

```
CREATE MACRO read_duckbox(path) AS TABLE
    FROM read_csv(
            printf(
                '(sed -n "2s/^| *//;s/ *|$//;s/ *| */|/p;2q" %s; ' ||
                'sed "1,4d;\$d;s/^| *//;s/ *|$//;s/ *| */|/g" %s) |',
                path, path
            ),
            delim = '|'
        );
```

Then, reading a duckbox table is as simple as:

```
FROM read_duckbox('duckbox.csv');
```

> `shellfs` is a community extension and it comes without any support or guarantees. Only use it if you can ensure that its inputs are appropriately sanitized. Please consult the Securing DuckDB page for more details.

Limitations

Please consider the following limitations when running this script:

- This approach only works if the table does not have long pipe `|` characters. It also trims spaces from the table cell values. Make sure to factor in these assumptions when running the script.
- The script is compatible with both BSD `sed` (which is the default on macOS) and GNU `sed` (which is the default on Linux and available on macOS as `gsed`).
- Only the data types supported by the CSV sniffer are parsed correctly. Values containing nested data will be parsed as a VARCHAR.

Copying an In-Memory Database to a File

Imagine the following situation – you started DuckDB in in-memory mode but would like to persist the state of your database to disk. To achieve this, **attach to a new disk-based database** and use the `COPY FROM DATABASE ... TO` command:

```
ATTACH 'my_database.db';
COPY FROM DATABASE memory TO my_database;
DETACH my_database;
```

> Ensure that the disk-based database file does not exist before attaching to it.

Troubleshooting

Crashes

DuckDB is thoroughly tested via an extensive test suite. However, bugs can still occur and these can sometimes lead to crashes. This page contains practical information on how to troubleshoot DuckDB crashes.

Types of Crashes

There are a few major types of crashes:

- **Termination signals:** The process stops with a `SIGSEGV` (segmentation fault), `SIGABRT`, etc.: these should never occur. Please submit an issue.
- **Internal errors:** an operation may result in an `Internal Error`, e.g.:

  ```
  INTERNAL Error:
  Attempted to access index 3 within vector of size 3
  ```

  After encountering an internal error, DuckDB enters a restricted mode where any further operations will result in the following error message:

  ```
  FATAL Error:
  Failed: database has been invalidated because of a previous fatal error.
  The database must be restarted prior to being used again.
  ```

- **Out of memory errors:** A DuckDB crash can also be a symptom of the operating system killing the process. For example, many Linux distributions run an OOM reaper or OOM killer process, which kills processes to free up their memory and thus prevents the operating system from running out of memory. If your DuckDB session is killed by the OOM reaper, consult the "OOM errors" page

Recovering Data

If your DuckDB session was writing to a persistent database file prior to crashing, there might be a WAL (write-ahead log) file next to your database named `<database_filename>.wal`{:.language-sql .highlight}. To recover data from the WAL file, simply start a new DuckDB session on the persistent database. DuckDB will then replay the write-ahead log and perform a checkpoint operation, restoring the database to the state before the crash.

Troubleshooting the Crash

Using the Latest Stable and Preview Builds

DuckDB is constantly improving, so there is a chance that the bug you have encountered has already been fixed in the codebase. First, try updating to the **latest stable build**. If this doesn't resolve the problem, try using the **preview build** (also known as the "nightly build").

If you would like use DuckDB with an open pull request applied to the codebase, you can try building it from source.

Search for Existing Issues

There is a chance that someone else already reported the bug that causes the crash. Please search in the GitHub issue tracker for the error message to see potentially related issues. DuckDB has a large community and there may be some suggestions for a workaround.

Disabling the Query Optimizer

Some crashes are caused by DuckDB's query optimizer component. To identify whether the optimizer is causing the crash, try to turn it off and re-run the query:

```
PRAGMA disable_optimizer;
```

If the query finishes successfully, then the crash was caused by one or more optimizer rules. To pinpoint the specific rules that caused the crash, you can try to selectively disable optimizer rules. This way, your query can still benefit from the rest of the optimizer rules.

Try to Isolate the Issue

Some issues are caused by the interplay of different components and extensions, or are specific to certain platforms or client languages. You can often isolate the issue to a smaller problem.

Reproducing in Plain SQL

Issues can also occur due to differences in client libraries. To understand whether this is the case, try reproducing the issue using plain SQL queries with the DuckDB CLI client. If you cannot reproduce the issue in the command line client, it is likely related to the client library.

Different Hardware Setup

According to our experience, several crashes occur due to faulty hardware (overheating hard drives, overclocked CPUs, etc.). Therefore, it's worth trying another computer to run the same workload.

Decomposing the Query

It's a good idea to try to break down the query into multiple smaller queries with each using a separate DuckDB extension and SQL feature.

For example, if you have a query that targets a dataset in an AWS S3 bucket and performs two joins on it, try to rewrite it as a series of smaller steps as follows. Download the dataset's files manually and load them into DuckDB. Then perform the first join and the second join separately. If the multi-step approach still exhibits the crash at some step, then the query that triggers the crash is a good basis for a minimal reproducible example.If the multi-step approach works and the multi-step process no longer crashes, try to reconstruct the original query and observe which step reintroduces the error. In both cases, you will have a better understanding of what is causing the issue and potentially also a workaround that you can use right away. In any case, please consider submitting an issue with your findings.

Submitting an Issue

If you found a crash in DuckDB, please consider submitting an issue in our GitHub issue tracker with a minimal reproducible example.

Out of Memory Errors

DuckDB has a state of the art out-of-core query engine that can spill to disk for larger-than-memory processing. We continuously strive to improve DuckDB to improve its scalability and prevent out of memory errors whenever possible. That said, you may still experience out-of-memory errors if you run queries with multiple blocking operators, certain aggregation functions, `PIVOT` operations, etc., or if you have very little available memory compared to the dataset size.

Types of Out of Memory Errors

Out of memory errors mainly occur in two forms:

`OutOfMemoryException`

Most of the time DuckDB runs out of memory with an `OutOfMemoryException`. For example:

```
duckdb.duckdb.OutOfMemoryException: Out of Memory Error: failed to pin block of size 256.0 KiB (476.7 MiB/476.8 MiB used)
```

OOM Reaper (Linux)

Many Linux distributions have an OOM killer or OOM reaper process whose goal is to prevent memory overcommitment. If the OOM reaper killed your process, you often see the following message where DuckDB was running:

```
Killed
```

To get more detailed information, check the diagnostic messages using the `dmesg` command (you may need `sudo`):

```
sudo dmesg
```

If the process was killed by the OOM killer/reaper, you will find an entry like this:

```
[Fri Apr 18 02:04:10 2025] Out of memory: Killed process 54400 (duckdb) total-vm:1037911068kB, anon-rss:770031964kB, file-rss:0kB, shmem-rss:0kB, UID:1000 pgtables:1814612kB oom_score_adj:0
```

Troubleshooting Out of Memory Errors

To prevent out of memory errors, try to reduce memory usage. To this end, please consult the "How to Tune Workloads" site. In short:

- Reduce the number of threads using the `SET threads = ...` command.
- If your query reads a large mount of data from a file or writes a large amount of data, try setting the `preserve_insertion_order` option to `false`: `SET preserve_insertion_order = false`.
- Reduce the memory limit below the default 80% (see the Limits page). This can be helpful because some DuckDB operations circumvent the buffer manager and can reserve more memory than allowed by the memory limit. If you observe this, set the memory limit to 50-60% of the total system memory by using `SET memory_limit = ...`.
- Break up the query into subqueries. This allows you to see where the intermediate results "blow up", causing the query to run out of memory.

See Also

For more information on DuckDB's memory management, see the "Memory Management in DuckDB" blog post.

Glossary of Terms

This page contains a glossary of a few common terms used in DuckDB.

Terms

In-Process Database Management System

The DBMS runs in the client application's process instead of running as a separate process, which is common in the traditional client–server setup. An alternative term is **embeddable** database management system. In general, the term *"embedded database management system"* should be avoided, as it can be confused with DBMSs targeting *embedded systems* (which run on e.g., microcontrollers).

Replacement Scan

In DuckDB, replacement scans are used when a table name used by a query does not exist in the catalog. These scans can substitute another data source instead of the table. Using replacement scans allows DuckDB to, e.g., seamlessly read Pandas DataFrames or read input data from remote sources without explicitly invoking the functions that perform this (e.g., reading Parquet files from https). For details, see the C API – Replacement Scans page.

Extension

DuckDB has a flexible extension mechanism that allows for dynamically loading extensions. These may extend DuckDB's functionality by providing support for additional file formats, introducing new types, and domain-specific functionality. For details, see the Extensions page.

Platform

The platform is a combination of the operating system (e.g., Linux, macOS, Windows), system architecture (e.g., AMD64, ARM64), and, optionally, the compiler used (e.g., GCC4). Platforms are used to distributed DuckDB binaries and extension packages.

Browsing Offline

You can browse the DuckDB documentation offline in the following formats:

- Single Markdown file (approx. 4 MB)
- PDF file (approx. 15 MB)
- Website packaged in a ZIP file (approx. 50 MB). To browse the website locally, decompress the package, navigate to the `duckdb-docs` directory, and run:

  ```
  python -m http.server
  ```

  Then, connect to http://localhost:8000/.

Operations Manual

Overview

We designed DuckDB to be easy to deploy and operate. We believe that most users do not need to consult the pages of the operations manual. However, there are certain setups – e.g., when DuckDB is running in mission-critical infrastructure – where we would like to offer advice on how to configure DuckDB. The operations manual contains advice for these cases and also offers convenient configuration snippets such as Gitignore files.

For advice on getting the best performance from DuckDB, see also the Performance Guide.

DuckDB's Footprint

Files Created by DuckDB

DuckDB creates several files and directories on disk. This page lists both the global and the local ones.

Global Files and Directories

DuckDB creates the following global files and directories in the user's home directory (denoted with ~):

Location	Description	Shared between versions	Shared between clients
`~/.duckdbrc`	The content of this file is executed when starting the DuckDB CLI client. The commands can be both dot command and SQL statements. The naming of this file follows the `~/.bashrc` and `~/.zshrc` "run commands" files.	Yes	Only used by CLI
`~/.duckdb_history`	History file, similar to `~/.bash_history` and `~/.zsh_history`. Used by the DuckDB CLI client.	Yes	Only used by CLI
`~/.duckdb/extensions`	Binaries of installed extensions.	No	Yes
`~/.duckdb/stored_secrets`	Persistent secrets created by the Secrets manager.	Yes	Yes

Local Files and Directories

DuckDB creates the following files and directories in the working directory (for in-memory connections) or relative to the database file (for persistent connections):

Name	Description	Example
`<database_filename>`{:.language-sql .highlight}	Database file. Only created in on-disk mode. The file can have any extension with typical extensions being `.duckdb`, `.db`, and `.ddb`.	`weather.duckdb`
`.tmp/`	Temporary directory. Only created in in-memory mode.	`.tmp/`
`<database_filename>.tmp/`{:.language-sql .highlight}	Temporary directory. Only created in on-disk mode.	`weather.tmp/`
`<database_filename>.wal`{:.language-sql .highlight}	Write-ahead log file. If DuckDB exits normally, the WAL file is deleted upon exit. If DuckDB crashes, the WAL file is required to recover data.	`weather.wal`

If you are working in a Git repository and would like to disable tracking these files by Git, see the instructions on using `.gitignore` for DuckDB.

Gitignore for DuckDB

If you work in a Git repository, you may want to configure your Gitignore to disable tracking files created by DuckDB. These potentially include the DuckDB database, write ahead log, temporary files.

Sample Gitignore Files

In the following, we present sample Gitignore configuration snippets for DuckDB.

Ignore Temporary Files but Keep Database

This configuration is useful if you would like to keep the database file in the version control system:

```
*.wal
*.tmp/
```

Ignore Database and Temporary Files

If you would like to ignore both the database and the temporary files, extend the Gitignore file to include the database file. The exact Gitignore configuration to achieve this depends on the extension you use for your DuckDB databases (`.duckdb`, `.db`, `.ddb`, etc.). For example, if your DuckDB files use the `.duckdb` extension, add the following lines to your `.gitignore` file:

```
*.duckdb*
*.wal
*.tmp/
```

Reclaiming Space

DuckDB uses a single-file format, which has some inherent limitations w.r.t. reclaiming disk space.

`CHECKPOINT`

To reclaim space after deleting rows, use the `CHECKPOINT` statement.

`VACUUM`

The `VACUUM` statement does *not* trigger vacuuming deletes and hence does not reclaim space.

Compacting a Database by Copying

To compact the database, you can create a fresh copy of the database using the `COPY FROM DATABASE` statement. In the following example, we first connect to the original database `db1`, then the new (empty) database `db2`. Then, we copy the content of `db1` to `db2`.

```
ATTACH 'db1.db' AS db1;
ATTACH 'db2.db' AS db2;
COPY FROM DATABASE db1 TO db2;
```

Logging

Logging

DuckDB contains a logging mechanism to provide additional information to users, such as query execution details, performance metrics, and system events.

Basics

DuckDB's logging mechanism can be enabled or disabled using pragmas. By default, logs are stored in a special table called `duckdb_logs` that can be queried like any other table.

Example:

```
PRAGMA enable_logging;
-- Run some queries..
SELECT * FROM duckdb_logs;
```

Log Level

DuckDB supports different logging levels that control the verbosity of the logs:

- `ERROR`: Only logs error messages
- `WARNING`: Logs warnings and errors
- `INFO`: Logs general information, warnings and errors (default)
- `DEBUG`: Logs detailed debugging information
- `TRACE`: Logs very detailed tracing information

The log level can be set using:

```
PRAGMA enable_logging;
SET logging_level = 'TRACE';
```

Log Types

In DuckDB, log messages can have an associated log type. Log types have 2 main goals. Firstly, they allow using includelists and excludelist to limit which types of log messages are logged. Secondly, log types can have a predetermined message schema which allows DuckDB to automatically parse the messages back into a structured data type.

Logging-Specific Types

To log only messages of a specific type:

```
PRAGMA enable_logging('HTTP');
```

The above pragma will automatically set the correct log level, and will add the `HTTP` type to the `enabled_log_types` settings.

Structured Logging

Some log types like HTTP will have an associated message schema. To make DuckDB automatically parse the message, use the `duckdb_logs_parsed()` macro. For example:

```
SELECT request.headers FROM duckdb_logs_parsed('HTTP');
```

List of Available Log Types

This is a (non-exhaustive) list of the available log types in DuckDB.

Log Type	Description	Structured
`QueryLog`	Logs which queries are executed in DuckDB	No
`FileSystem`	Logs all FileSystem interaction with DuckDB's Filesystem	Yes
HTTP	Logs all HTTP traffic from DuckDB's internal HTTP client	Yes

Log Storage

By default, DuckDB logs to an in-memory log storage. Alternatively, DuckDB can log straight to `stdout` using:

```
PRAGMA enable_logging;
SET logging_storage = 'stdout';
```

Securing DuckDB

Securing DuckDB

DuckDB is quite powerful, which can be problematic, especially if untrusted SQL queries are run, e.g., from public-facing user inputs. This page lists some options to restrict the potential fallout from malicious SQL queries.

The approach to securing DuckDB varies depending on your use case, environment, and potential attack models. Therefore, consider the security-related configuration options carefully, especially when working with confidential data sets.

If you plan to embed DuckDB in your application, please consult the "Embedding DuckDB" page.

Reporting Vulnerabilities

If you discover a potential vulnerability, please report it confidentially via GitHub.

Safe Mode (CLI)

DuckDB's CLI client supports "safe mode", which prevents DuckDB from accessing external files other than the database file. This can be activated via a command line argument or a dot command:

```
duckdb -safe ...
```

```
.safe_mode
```

Restricting File Access

DuckDB can list directories and read arbitrary files via its CSV parser's `read_csv` function or read text via the `read_text` function. This makes it possible to read from the local file system, for example:

```
SELECT *
FROM read_csv('/etc/passwd', sep = ':');
```

Disabling File Access

Files access can be disabled in two ways. First, you can disable individual file systems. For example:

```
SET disabled_filesystems = 'LocalFileSystem';
```

Second, you can also completely disable external access by setting the `enable_external_access` option to `false`.

```
SET enable_external_access = false;
```

This setting implies that:

- ATTACH cannot attach to a database in a file.
- COPY cannot read to or write from files.
- Functions such as `read_csv`, `read_parquet`, `read_json`, etc. cannot read from an external source.

The `allowed_directories` and `allowed_paths` Options

You can restrict DuckDB's access to certain directories or files using the `allowed_directories` and `allowed_paths` options (respectively). These options allows fine-grained access control for the file system. For example, you can set DuckDB to only use the `/tmp` directory.

```
SET allowed_directories = ['/tmp'];
SET enable_external_access = false;
FROM read_csv('test.csv');
```

With the setting applied, DuckDB will refuse to read files in the current working directory:

```
Permission Error:
Cannot access file "test.csv" - file system operations are disabled by configuration
```

Secrets

Secrets are used to manage credentials to log into third party services like AWS or Azure. DuckDB can show a list of secrets using the `duckdb_secrets()` table function. This will redact any sensitive information such as security keys by default. The `allow_unredacted_secrets` option can be set to show all information contained within a security key. It is recommended not to turn on this option if you are running untrusted SQL input.

Queries can access the secrets defined in the Secrets Manager. For example, if there is a secret defined to authenticate with a user, who has write privileges to a given AWS S3 bucket, queries may write to that bucket. This is applicable for both persistent and temporary secrets.

Persistent secrets are stored in unencrypted binary format on the disk. These have the same permissions as SSH keys, `600`, i.e., only user who is running the DuckDB (parent) process can read and write them.

Locking Configurations

Security-related configuration settings generally lock themselves for safety reasons. For example, while we can disable community extensions using the `SET allow_community_extensions = false`, we cannot re-enable them again after the fact without restarting the database. Trying to do so will result in an error:

```
Invalid Input Error: Cannot upgrade allow_community_extensions setting while database is running
```

This prevents untrusted SQL input from re-enabling settings that were explicitly disabled for security reasons.

Nevertheless, many configuration settings do not disable themselves, such as the resource constraints. If you allow users to run SQL statements unrestricted on your own hardware, it is recommended that you lock the configuration after your own configuration has finished using the following command:

```
SET lock_configuration = true;
```

This prevents any configuration settings from being modified from that point onwards.

Prepared Statements to Prevent SQL Injection

Similarly to other SQL databases, it's recommended to use prepared statements in DuckDB to prevent SQL injection.

Therefore, avoid concatenating strings for queries:

```
import duckdb
duckdb.execute("SELECT * FROM (VALUES (32, 'a'), (42, 'b')) t(x) WHERE x = " + str(42)).fetchall()
```

Instead, use prepared statements:

```
import duckdb
duckdb.execute("SELECT * FROM (VALUES (32, 'a'), (42, 'b')) t(x) WHERE x = ?", [42]).fetchall()
```

Constrain Resource Usage

DuckDB can use quite a lot of CPU, RAM, and disk space. To avoid denial of service attacks, these resources can be limited.

The number of CPU threads that DuckDB can use can be set using, for example:

```
SET threads = 4;
```

Where 4 is the number of allowed threads.

The maximum amount of memory (RAM) can also be limited, for example:

```
SET memory_limit = '4GB';
```

The size of the temporary file directory can be limited with:

```
SET max_temp_directory_size = '4GB';
```

Extensions

DuckDB has a powerful extension mechanism, which have the same privileges as the user running DuckDB's (parent) process. This introduces security considerations. Therefore, we recommend reviewing the configuration options for securing extensions.

Privileges

Avoid running DuckDB as a root user (e.g., using `sudo`). There is no good reason to run DuckDB as root.

Generic Solutions

Securing DuckDB can also be supported via proven means, for example:

- Scoping user privileges via `chroot`, relying on the operating system
- Containerization, e.g., Docker and Podman
- Running DuckDB in WebAssembly

Embedding DuckDB

CLI Client

The Command Line Interface (CLI) client is intended for interactive use cases and not for embedding. As a result, it has more features that could be abused by a malicious actor. For example, the CLI client has the `.sh` feature that allows executing arbitrary shell commands. This feature is only present in the CLI client and not in any other DuckDB clients.

```
.sh ls
```

> **Tip.** Calling DuckDB's CLI client via shell commands is **not recommended** for embedding DuckDB. It is recommended to use one of the client libraries, e.g., Python, R, Java, etc.

Securing Extensions

DuckDB has a powerful extension mechanism, which have the same privileges as the user running DuckDB's (parent) process. This introduces security considerations. Therefore, we recommend reviewing the configuration options listed on this page and setting them according to your attack models.

DuckDB Signature Checks

DuckDB extensions are checked on every load using the signature of the binaries. There are currently three categories of extensions:

- Signed with a `core` key. Only extensions vetted by the core DuckDB team are signed with these keys.
- Signed with a `community` key. These are open-source extensions distributed via the DuckDB Community Extensions repository.
- Unsigned.

Overview of Security Levels for Extensions

DuckDB offers the following security levels for extensions.

Usable extensions	Description	Configuration
`core`	Extensions can only be loaded if signed from a `core` key.	`SET allow_community_extensions = false`
`core` and `community`	Extensions can only be loaded if signed from a `core` or `community` key.	This is the default security level.
Any extension including unsigned	Any extensions can be loaded.	`SET allow_unsigned_extensions = true`

Security-related configuration settings lock themselves, i.e., it is only possible to restrict capabilities in the current process.

For example, attempting the following configuration changes will result in an error:

```
SET allow_community_extensions = false;
SET allow_community_extensions = true;
```

```
Invalid Input Error: Cannot upgrade allow_community_extensions setting while database is running
```

Community Extensions

DuckDB has a Community Extensions repository, which allows convenient installation of third-party extensions. Community extension repositories like pip or npm are essentially enabling remote code execution by design. This is less dramatic than it sounds. For better or worse, we are quite used to piping random scripts from the web into our shells, and routinely install a staggering amount of transitive dependencies without thinking twice. Some repositories like CRAN enforce a human inspection at some point, but that's no guarantee for anything either.

We've studied several different approaches to community extension repositories and have picked what we think is a sensible approach: we do not attempt to review the submissions, but require that the *source code of extensions is available*. We do take over the complete build, sign and distribution process. Note that this is a step up from pip and npm that allow uploading arbitrary binaries but a step down from reviewing everything manually. We allow users to report malicious extensions and show adoption statistics like GitHub stars and download count. Because we manage the repository, we can remove problematic extensions from distribution quickly.

Despite this, installing and loading DuckDB extensions from the community extension repository will execute code written by third party developers, and therefore *can* be dangerous. A malicious developer could create and register a harmless-looking DuckDB extension that steals your crypto coins. If you're running a web service that executes untrusted SQL from users with DuckDB, we recommend disabling community extensions. To do so, run:

```
SET allow_community_extensions = false;
```

Disabling Autoinstalling and Autoloading Known Extensions

By default, DuckDB automatically installs and loads known extensions. To disable autoinstalling known extensions, run:

```
SET autoinstall_known_extensions = false;
```

To disable autoloading known extensions, run:

```
SET autoload_known_extensions = false;
```

To lock this configuration, use the `lock_configuration` option:

```
SET lock_configuration = true;
```

Always Require Signed Extensions

By default, DuckDB requires extensions to be either signed as core extensions (created by the DuckDB developers) or community extensions (created by third-party developers but distributed by the DuckDB developers). The `allow_unsigned_extensions` setting can be enabled on start-up to allow loading unsigned extensions. While this setting is useful for extension development, enabling it will allow DuckDB to load *any extensions,* which means more care must be taken to ensure malicious extensions are not loaded.

Non-Deterministic Behavior

Several operators in DuckDB exhibit non-deterministic behavior. Most notably, SQL uses set semantics, which allows results to be returned in a different order. DuckDB exploits this to improve performance, particularly when performing multi-threaded query execution. Other factors, such as using different compilers, operating systems, and hardware architectures, can also cause changes in ordering. This page documents the cases where non-determinism is an *expected behavior*. If you would like to make your queries determinisic, see the "Working Around Non-Determinism" section.

Set Semantics

One of the most common sources of non-determinism is the set semantics used by SQL. E.g., if you run the following query repeatedly, you may get two different results:

```
SELECT *
FROM (
    SELECT 'A' AS x
    UNION
    SELECT 'B' AS x
);
```

Both results A, B and B, A are correct.

Different Results on Different Platforms: `array_distinct`

The `array_distinct` function may return results in a different order on different platforms:

```
SELECT array_distinct(['A', 'A', 'B', NULL, NULL]) AS arr;
```

For this query, both `[A, B]` and `[B, A]` are valid results.

Floating-Point Aggregate Operations with Multi-Threading

Floating-point inaccuracies may produce different results when run in a multi-threaded configurations: For example, `stddev` and `corr` may produce non-deterministic results:

```
CREATE TABLE tbl AS
    SELECT 'ABCDEFG'[floor(random() * 7 + 1)::INT] AS s, 3.7 AS x, i AS y
    FROM range(1, 1_000_000) r(i);

SELECT s, stddev(x) AS standard_deviation, corr(x, y) AS correlation
FROM tbl
GROUP BY s
ORDER BY s;
```

The expected standard deviations and correlations from this query are 0 for all values of `s`. However, when executed on multiple threads, the query may return small numbers (`0 <= z < 10e-16`) due to floating-point inaccuracies.

Working Around Non-Determinism

For the majority of use cases, non-determinism is not causing any issues. However, there are some cases where deterministic results are desirable. In these cases, try the following workarounds:

1. Limit the number of threads to prevent non-determinism introduced by multi-threading.

   ```
   SET threads = 1;
   ```

2. Enforce ordering. For example, you can use the `ORDER BY ALL` clause:

   ```
   SELECT *
   FROM (
       SELECT 'A' AS x
       UNION
       SELECT 'B' AS x
   )
   ORDER BY ALL;
   ```

   You can also sort lists using `list_sort`:

   ```
   SELECT list_sort(array_distinct(['A', 'A', 'B', NULL, NULL])) AS i
   ORDER BY i;
   ```

   It's also possible to introduce a deterministic shuffling.

Limits

This page contains DuckDB's built-in limit values.

Limit Values

Limit	Default value	Configuration option	Comment
Array size	100000	-	
BLOB size	4 GB	-	
Expression depth	1000	`max_expression_depth`	
Memory allocation for a vector	128 GB	-	
Memory use	80% of RAM	`memory_limit`	Note: This limit only applies to the buffer manager.
String size	4 GB	-	
Temporary directory size	unlimited	`max_temp_directory_size`	

Size of Database Files

DuckDB doesn't have a practical limit for the size of a single DuckDB database file. We have database files using 15 TB+ of disk space and they work fine. However, connecting to such a huge database may take a few seconds, and checkpointing can be slower.

Development

DuckDB Repositories

Several components of DuckDB are maintained in separate repositories.

Main Repositories

- `duckdb`: core DuckDB project
- `duckdb-web`: documentation and blog

Clients

- `duckdb-java`: Java (JDBC) client
- `duckdb-node`: Node.js client, first iteration
- `duckdb-node-neo`: Node.js client, second iteration
- `duckdb-odbc`: ODBC client
- `duckdb-pyodide`: Pyodide client
- `duckdb-r`: R client
- `duckdb-rs`: Rust client
- `duckdb-swift`: Swift client
- `duckdb-wasm`: WebAssembly client
- `duckplyr`: a drop-in replacement for dplyr in R
- `go-duckdb`: Go client

Connectors

- `dbt-duckdb`: dbt
- `duckdb-mysql`: MySQL connector
- `duckdb-postgres`: PostgreSQL connector (connect to PostgreSQL from DuckDB)
- `duckdb-sqlite`: SQLite connector
- `pg_duckdb`: official PostgreSQL extension for DuckDB (run DuckDB in PostgreSQL)

Extensions

- Core extension repositories are linked in the Official Extensions page
- Community extensions are served from the Community Extensions repository

Specifications

- DuckLake specification

Profiling

Profiling is essential to help understand why certain queries exhibit specific performance characteristics. DuckDB contains several built-in features to enable query profiling, which this page covers. For a high-level example of using `EXPLAIN`, see the "Inspect Query Plans" page. For an in-depth explanation, see the "Profiling" page in the Developer Documentation.

Statements

The `EXPLAIN` Statement

The first step to profiling a query can include examining the query plan. The `EXPLAIN` statement shows the query plan and describes what is going on under the hood.

The `EXPLAIN ANALYZE` Statement

The query plan helps developers understand the performance characteristics of the query. However, it is often also necessary to examine the performance numbers of individual operators and the cardinalities that pass through them. The `EXPLAIN ANALYZE` statement enables obtaining these, as it pretty-prints the query plan and also executes the query. Thus, it provides the actual run-time performance numbers.

The `FORMAT` Option

The `EXPLAIN [ANALYZE]` statement allows exporting to several formats:

- `text` – default ASCII-art style output
- `graphviz` – produces a DOT output, which can be rendered with Graphviz
- `html` – produces an HTML output, which can rendered with treeflex
- `json` – produces a JSON output

To specify a format, use the `FORMAT` tag:

```
EXPLAIN (FORMAT html) SELECT 42 AS x;
```

Pragmas

DuckDB supports several pragmas for turning profiling on and off and controlling the level of detail in the profiling output.

The following pragmas are available and can be set using either `PRAGMA` or `SET`. They can also be reset using `RESET`, followed by the setting name. For more information, see the "Profiling" section of the pragmas page.

Setting	Description	Default	Options
`enable_profiling`, `enable_profile`	Turn on profiling	`query_tree`	`query_tree`, `json`, `query_tree_optimizer`, `no_output`

Setting	Description	Default	Options
`profiling_output`	Set a profiling output file	Console	A filepath
`profiling_mode`	Toggle additional optimizer and planner metrics	`standard`	`standard`, `detailed`
`custom_profiling_settings`	Enable or disable specific metrics	All metrics except those activated by detailed profiling	A JSON object that matches the following: `{"METRIC_NAME": "boolean", ...}`. See the metrics section below.
`disable_profiling`, `disable_profile`	Turn off profiling		

Metrics

The query tree has two types of nodes: the `QUERY_ROOT` and `OPERATOR` nodes. The `QUERY_ROOT` refers exclusively to the top-level node, and the metrics it contains are measured over the entire query. The `OPERATOR` nodes refer to the individual operators in the query plan. Some metrics are only available for `QUERY_ROOT` nodes, while others are only for `OPERATOR` nodes. The table below describes each metric and which nodes they are available for.

Other than `QUERY_NAME` and `OPERATOR_TYPE`, it is possible to turn all metrics on or off.

Metric	Return type	Unit	Query	Operator	Description
`BLOCKED_THREAD_TIME`	`double`	seconds	✅		The total time threads are blocked
`EXTRA_INFO`	`string`		✅	✅	Unique operator metrics
`LATENCY`	`double`	seconds	✅		The total elapsed query execution time
`OPERATOR_CARDINALITY`	`uint64`	absolute		✅	The cardinality of each operator, i.e., the number of rows it returns to its parent. Operator equivalent of `ROWS_RETURNED`
`OPERATOR_ROWS_SCANNED`	`uint64`	absolute		✅	The total rows scanned by each operator
`OPERATOR_TIMING`	`double`	seconds		✅	The time taken by each operator. Operator equivalent of `LATENCY`
`OPERATOR_TYPE`	`string`			✅	The name of each operator
`QUERY_NAME`	`string`		✅		The query string
`RESULT_SET_SIZE`	`uint64`	bytes	✅	✅	The size of the result
`ROWS_RETURNED`	`uint64`	absolute	✅		The number of rows returned by the query
`SYSTEM_PEAK_BUFFER_MEMORY`	`uint64`	bytes	✅		Peak memory usage of all allocated buffers of the system during the query

Metric	Return type	Unit	Query	Operator	Description
SYSTEM_PEAK_TEMP_DIR_SIZE	uint64	bytes	✅		Peak size of the temp directory of the system during the query

Cumulative Metrics

DuckDB also supports several cumulative metrics that are available in all nodes. In the QUERY_ROOT node, these metrics represent the sum of the corresponding metrics across all operators in the query. The OPERATOR nodes represent the sum of the operator's specific metric and those of all its children recursively.

These cumulative metrics can be enabled independently, even if the underlying specific metrics are disabled. The table below shows the cumulative metrics. It also depicts the metric based on which DuckDB calculates the cumulative metric.

Metric	Unit	Metric calculated cumulatively
CPU_TIME	seconds	OPERATOR_TIMING
CUMULATIVE_CARDINALITY	absolute	OPERATOR_CARDINALITY
CUMULATIVE_ROWS_SCANNED	absolute	OPERATOR_ROWS_SCANNED

CPU_TIME measures the cumulative operator timings. It does not include time spent in other stages, like parsing, query planning, etc. Thus, for some queries, the LATENCY in the QUERY_ROOT can be greater than the CPU_TIME.

Detailed Profiling

When the profiling_mode is set to detailed, an extra set of metrics are enabled, which are only available in the QUERY_ROOT node. These include OPTIMIZER, PLANNER, and PHYSICAL_PLANNER metrics. They are measured in seconds and returned as a double. It is possible to toggle each of these additional metrics individually.

Optimizer Metrics

At the QUERY_ROOT node, there are metrics that measure the time taken by each optimizer. These metrics are only available when the specific optimizer is enabled. The available optimizations can be queried using the duckdb_optimizers(){:.language-sql .highlight} table function.

Each optimizer has a corresponding metric that follows the template: OPTIMIZER_<OPTIMIZER_NAME>{:.language-sql .highlight}. For example, the OPTIMIZER_JOIN_ORDER metric corresponds to the JOIN_ORDER optimizer.

Additionally, the following metrics are available to support the optimizer metrics:

- ALL_OPTIMIZERS: Enables all optimizer metrics and measures the time the optimizer parent node takes.
- CUMULATIVE_OPTIMIZER_TIMING: The cumulative sum of all optimizer metrics. It is usable without turning on all optimizer metrics.

Planner Metrics

The planner is responsible for generating the logical plan. Currently, DuckDB measures two metrics in the planner:

- PLANNER: The time to generate the logical plan from the parsed SQL nodes.
- PLANNER_BINDING: The time taken to bind the logical plan.

Physical Planner Metrics

The physical planner is responsible for generating the physical plan from the logical plan. The following are the metrics supported in the physical planner:

- `PHYSICAL_PLANNER`: The time spent generating the physical plan.
- `PHYSICAL_PLANNER_COLUMN_BINDING`: The time spent binding the columns in the logical plan to physical columns.
- `PHYSICAL_PLANNER_RESOLVE_TYPES`: The time spent resolving the types in the logical plan to physical types.
- `PHYSICAL_PLANNER_CREATE_PLAN`: The time spent creating the physical plan.

Custom Metrics Examples

The following examples demonstrate how to enable custom profiling and set the output format to `json`. In the first example, we enable profiling and set the output to a file. We only enable `EXTRA_INFO`, `OPERATOR_CARDINALITY`, and `OPERATOR_TIMING`.

```
CREATE TABLE students (name VARCHAR, sid INTEGER);
CREATE TABLE exams (eid INTEGER, subject VARCHAR, sid INTEGER);
INSERT INTO students VALUES ('Mark', 1), ('Joe', 2), ('Matthew', 3);
INSERT INTO exams VALUES (10, 'Physics', 1), (20, 'Chemistry', 2), (30, 'Literature', 3);

PRAGMA enable_profiling = 'json';
PRAGMA profiling_output = '/path/to/file.json';

PRAGMA custom_profiling_settings = '{"CPU_TIME": "false", "EXTRA_INFO": "true", "OPERATOR_CARDINALITY":
"true", "OPERATOR_TIMING": "true"}';

SELECT name
FROM students
JOIN exams USING (sid)
WHERE name LIKE 'Ma%';
```

The file's content after executing the query:

```
{
    "extra_info": {},
    "query_name": "SELECT name\nFROM students\nJOIN exams USING (sid)\nWHERE name LIKE 'Ma%';",
    "children": [
        {
            "operator_timing": 0.000001,
            "operator_cardinality": 2,
            "operator_type": "PROJECTION",
            "extra_info": {
                "Projections": "name",
                "Estimated Cardinality": "1"
            },
            "children": [
                {
                    "extra_info": {
                        "Join Type": "INNER",
                        "Conditions": "sid = sid",
                        "Build Min": "1",
                        "Build Max": "3",
                        "Estimated Cardinality": "1"
                    },
                    "operator_cardinality": 2,
                    "operator_type": "HASH_JOIN",
                    "operator_timing": 0.00023899999999999998,
                    "children": [
...
```

The second example adds detailed metrics to the output.

```
PRAGMA profiling_mode = 'detailed';

SELECT name
FROM students
JOIN exams USING (sid)
WHERE name LIKE 'Ma%';
```

The contents of the outputted file:

```
{
  "all_optimizers": 0.001413,
  "cumulative_optimizer_timing": 0.0014120000000000003,
  "planner": 0.000873,
  "planner_binding": 0.000869,
  "physical_planner": 0.000236,
  "physical_planner_column_binding": 0.000005,
  "physical_planner_resolve_types": 0.000001,
  "physical_planner_create_plan": 0.000226,
  "optimizer_expression_rewriter": 0.000029,
  "optimizer_filter_pullup": 0.000002,
  "optimizer_filter_pushdown": 0.000102,
...
  "optimizer_column_lifetime": 0.000009999999999999999,
  "rows_returned": 2,
  "latency": 0.003708,
  "cumulative_rows_scanned": 6,
  "cumulative_cardinality": 11,
  "extra_info": {},
  "cpu_time": 0.000095,
  "optimizer_build_side_probe_side": 0.000017,
  "result_set_size": 32,
  "blocked_thread_time": 0.0,
  "query_name": "SELECT name\nFROM students\nJOIN exams USING (sid)\nWHERE name LIKE 'Ma%';",
  "children": [
    {
      "operator_timing": 0.000001,
      "operator_rows_scanned": 0,
      "cumulative_rows_scanned": 6,
      "operator_cardinality": 2,
      "operator_type": "PROJECTION",
      "cumulative_cardinality": 11,
      "extra_info": {
        "Projections": "name",
        "Estimated Cardinality": "1"
      },
      "result_set_size": 32,
      "cpu_time": 0.000095,
      "children": [
...
```

Query Graphs

It is also possible to render the profiling output as a query graph. The query graph visually represents the query plan, showing the operators and their relationships. The query plan must be output in the `json` format and stored in a file. After writing a profiling output to its designated file, the Python script can render it as a query graph. The script requires the `duckdb` Python module to be installed. It generates an HTML file and opens it in your web browser.

```
python -m duckdb.query_graph /path/to/file.json
```

Notation in Query Plans

In query plans, the hash join operators adhere to the following convention: the *probe side* of the join is the left operand, while the *build side* is the right operand.

Join operators in the query plan show the join type used:

- Inner joins are denoted as `INNER`.
- Left outer joins and right outer joins are denoted as `LEFT` and `RIGHT`, respectively.
- Full outer joins are denoted as `FULL`.

> **Tip.** To visualize query plans, consider using the DuckDB execution plan visualizer developed by the Database Systems Research Group at the University of Tübingen.

Building DuckDB

Building DuckDB from Source

When Should You Build DuckDB?

DuckDB binaries are available for *stable* and *preview* builds on the installation page. In most cases, it's recommended to use these binaries. When you are running on an experimental platform (e.g., Raspberry Pi) or you would like to build the project for an unmerged pull request, you can build DuckDB from source based on the duckdb/duckdb repository hosted on GitHub. This page explains the steps for building DuckDB.

Prerequisites

DuckDB needs CMake and a C++11-compliant compiler (e.g., GCC, Apple-Clang, MSVC). Additionally, we recommend using the Ninja build system, which automatically parallelizes the build process.

Getting Started

A `Makefile` wraps the build process. See Build Configuration for targets and configuration flags.

```
make
make release # same as plain make
make debug
GEN=ninja make # for use with Ninja
BUILD_BENCHMARK=1 make # Build with benchmarks
```

Platforms

Platforms with Full Support

DuckDB fully supports Linux, macOS and Windows. Both x86_64 (amd64) and AArch64 (arm64) builds are available for these platforms, and almost all extensions are distributed for these platforms.

Platform name	Description
`linux_amd64`	Linux x86_64 (amd64) with glibc
`linux_arm64`	Linux AArch64 (arm64) with glibc
`osx_amd64`	macOS 12+ amd64 (Intel CPUs)
`osx_arm64`	macOS 12+ arm64 (Apple Silicon CPUs)
`windows_amd64`	Windows 10+ x86_64 (amd64)
`windows_arm64`	Windows 10+ AArch64 (arm64)

For these platforms, builds are available for both the latest stable version and the preview version (nightly build). In some circumstances, you may still want to build DuckDB from source, e.g., to test an unmerged pull request. For build instructions on these platforms, see:

- Linux
- macOS
- Windows

Platforms with Partial Support

There are several partially supported platforms. For some platforms, DuckDB binaries and extensions (or a subset of extensions) are distributed. For others, building from source is possible.

Platform name	Description
`linux_amd64_musl`	Linux x86_64 (amd64) with musl libc, e.g., Alpine Linux
`linux_arm64_musl`	Linux AArch64 (arm64) with musl libc, e.g., Alpine Linux
`linux_arm64_android`	Android AArch64 (arm64)
`wasm_eh`	WebAssembly Exception Handling

Below, we provide detailed build instructions for some platforms:

- Android
- Raspberry Pi

Platforms with Best Effort Support

Platform name	Description
`freebsd_amd64`	FreeBSD x86_64 (amd64)
`freebsd_arm64`	FreeBSD AArch64 (arm64)
`wasm_mvp`	WebAssembly Minimum Viable Product
`windows_amd64_mingw`	Windows 10+ x86_64 (amd64) with MinGW
`windows_arm64_mingw`	Windows 10+ AArch64 (arm64) with MinGW

> These platforms are not covered by DuckDB's community support. For details on commercial support, see the support policy page.

See also the "Unofficial and Unsupported Platforms" page for details.

Outdated Platforms

Some platforms were supported in older DuckDB versions but are no longer supported.

Platform name	Description
`linux_amd64_gcc4`	Linux AMDM64 (x86_64) with GCC 4, e.g., CentOS 7
`linux_arm64_gcc4`	Linux AArch64 (arm64) with GCC 4, e.g., CentOS 7
`windows_amd64_rtools`	Windows 10+ x86_64 (amd64) for RTools

DuckDB can also be built for end-of-life platforms such as macOS 11 and CentOS 7/8 using the instructions provided for macOS and Linux.

Amalgamation Build

DuckDB can be build as a single pair of C++ header and source code files (`duckdb.hpp` and `duckdb.cpp`) with approximately 0.5M lines of code. To generate this file, run:

```
python scripts/amalgamation.py
```

Note that amalgamation build is provided on a best-effort basis and is not officially supported.

Limitations

Currently, DuckDB has the following limitations:

- The DuckDB codebase is not compatible with C++23. Therefore, trying to compile DuckDB with `-std=c++23` will fail.
- The `-march=native` build flag, i.e., compiling DuckDB with the local machine's native instructions set, is not supported.

Troubleshooting Guides

We provide troubleshooting guides for building DuckDB:

- Generic issues
- Python
- R

Building Configuration

Build Types

DuckDB can be built in many different settings, most of these correspond directly to CMake but not all of them.

`release`

This build has been stripped of all the assertions and debug symbols and code, optimized for performance.

`debug`

This build runs with all the debug information, including symbols, assertions and `#ifdef DEBUG` blocks. Due to these, binaries of this build are expected to be slow. Note: the special debug defines are not automatically set for this build.

`relassert`

This build does not trigger the `#ifdef DEBUG` code blocks but it still has debug symbols that make it possible to step through the execution with line number information and `D_ASSERT` lines are still checked in this build. Binaries of this build mode are significantly faster than those of the `debug` mode.

`reldebug`

This build is similar to `relassert` in many ways, only assertions are also stripped in this build.

`benchmark`

This build is a shorthand for `release` with `BUILD_BENCHMARK=1` set.

`tidy-check`

This creates a build and then runs Clang-Tidy to check for issues or style violations through static analysis. The CI will also run this check, causing it to fail if this check fails.

`format-fix` | `format-changes` | `format-main`

This doesn't actually create a build, but uses the following format checkers to check for style issues:

- clang-format to fix format issues in the code.
- cmake-format to fix format issues in the `CMakeLists.txt` files.

The CI will also run this check, causing it to fail if this check fails.

Extension Selection

Core DuckDB extensions are the ones maintained by the DuckDB team. These are hosted in the `duckdb` GitHub organization and are served by the `core` extension repository.

Core extensions can be built as part of DuckDB via the `CORE_EXTENSIONS` flag, then listing the names of the extensions that are to be built.

```
CORE_EXTENSIONS='tpch;httpfs;fts;json;parquet' make
```

More on this topic at building duckdb extensions.

Package Flags

For every package that is maintained by core DuckDB, there exists a flag in the Makefile to enable building the package. These can be enabled by either setting them in the current `env`, through set up files like `bashrc` or `zshrc`, or by setting them before the call to `make`, for example:

```
BUILD_PYTHON=1 make debug
```

`BUILD_PYTHON`

When this flag is set, the Python package is built.

BUILD_SHELL

When this flag is set, the CLI is built, this is usually enabled by default.

BUILD_BENCHMARK

When this flag is set, DuckDB's in-house benchmark suite is built. More information about this can be found in the README.

BUILD_JDBC

When this flag is set, the Java package is built.

BUILD_ODBC

When this flag is set, the ODBC package is built.

Miscellaneous Flags

DISABLE_UNITY

To improve compilation time, we use Unity Build to combine translation units. This can however hide include bugs, this flag disables using the unity build so these errors can be detected.

DISABLE_SANITIZER

In some situations, running an executable that has been built with sanitizers enabled is not support / can cause problems. Julia is an example of this. With this flag enabled, the sanitizers are disabled for the build.

Overriding Git Hash and Version

It is possible to override the Git hash and version when building from source using the `OVERRIDE_GIT_DESCRIBE` environment variable. This is useful when building from sources that are not part of a complete Git repository (e.g., an archive file with no information on commit hashes and tags). For example:

```
OVERRIDE_GIT_DESCRIBE=v0.10.0-843-g09ea97d0a9 GEN=ninja make
```

Will result in the following output when running `./build/release/duckdb`:

```
v0.10.1-dev843 09ea97d0a9
...
```

Building Extensions

Extensions can be built from source and installed from the resulting local binary.

Building Extensions

To build using extension flags, set the `CORE_EXTENSIONS` flag to the list of extensions that you want to be build. For example:

```
CORE_EXTENSIONS='autocomplete;httpfs;icu;json;tpch' GEN=ninja make
```

This option also accepts out-of-tree extensions such as `delta`:

```
CORE_EXTENSIONS='autocomplete;httpfs;icu;json;tpch;delta' GEN=ninja make
```

In most cases, extension will be directly linked in the resulting DuckDB executable.

Special Extension Flags

`BUILD_JEMALLOC`

When this flag is set, the `jemalloc` extension is built.

`BUILD_TPCE`

When this flag is set, the TPCE libray is built. Unlike TPC-H and TPC-DS this is not a proper extension and it's not distributed as such. Enabling this allows TPC-E enabled queries through our test suite.

Debug Flags

`CRASH_ON_ASSERT`

`D_ASSERT(condition)` is used all throughout the code, these will throw an InternalException in debug builds. With this flag enabled, when the assertion triggers it will instead directly cause a crash.

`DISABLE_STRING_INLINE`

In our execution format `string_t` has the feature to "inline" strings that are under a certain length (12 bytes), this means they don't require a separate allocation. When this flag is set, we disable this and don't inline small strings.

`DISABLE_MEMORY_SAFETY`

Our data structures that are used extensively throughout the non-performance-critical code have extra checks to ensure memory safety, these checks include:

- Making sure `nullptr` is never dereferenced.
- Making sure index out of bounds accesses don't trigger a crash.

With this flag enabled we remove these checks, this is mostly done to check that the performance hit of these checks is negligible.

`DESTROY_UNPINNED_BLOCKS`

When previously pinned blocks in the BufferManager are unpinned, with this flag enabled we destroy them instantly to make sure that there aren't situations where this memory is still being used, despite not being pinned.

DEBUG_STACKTRACE

When a crash or assertion hit occurs in a test, print a stack trace. This is useful when debugging a crash that is hard to pinpoint with a debugger attached.

Using a CMake Configuration File

To build using a CMake configuration file, create an extension configuration file named `extension_config.cmake` with e.g., the following content:

```
duckdb_extension_load(autocomplete)
duckdb_extension_load(fts)
duckdb_extension_load(inet)
duckdb_extension_load(icu)
duckdb_extension_load(json)
duckdb_extension_load(parquet)
```

Build DuckDB as follows:

```
GEN=ninja EXTENSION_CONFIGS="extension_config.cmake" make
```

Then, to install the extensions in one go, run:

```
# for release builds
cd build/release/extension/
# for debug builds
cd build/debug/extension/
# install extensions
for EXTENSION in *; do
    ../duckdb -c "INSTALL '${EXTENSION}/${EXTENSION}.duckdb_extension';"
done
```

Android

DuckDB has experimental support for Android. Please use the latest `main` branch of DuckDB instead of the stable versions.

Building the DuckDB Library Using the Android NDK

We provide build instructions for setups using macOS and Android Studio. For other setups, please adjust the steps accordingly.

1. Open Android Studio. Select the **Tools** menu and pick **SDK Manager**. Select the SDK Tools tab and tick the **NDK (Side by side)** option. Click **OK** to install.
2. Set the Android NDK's location. For example:

   ```
   ANDROID_NDK=~/Library/Android/sdk/ndk/28.0.12433566/
   ```

3. Set the Android ABI. For example:

   ```
   ANDROID_ABI=arm64-v8a
   ```

   Or:

   ```
   ANDROID_ABI=x86_64
   ```

4. If you would like to use the Ninja build system, make sure it is installed and available on the PATH.
5. Set the list of DuckDB extensions to build. These will be statically linked in the binary. For example:

```
DUCKDB_EXTENSIONS="icu;json;parquet"
```

6. Navigate to DuckDB's directory and run the build as follows:

```
PLATFORM_NAME="android_${ANDROID_ABI}"
BUILDDIR=./build/${PLATFORM_NAME}
mkdir -p ${BUILDDIR}
cd ${BUILDDIR}
cmake \
    -G "Ninja" \
    -DEXTENSION_STATIC_BUILD=1 \
    -DDUCKDB_EXTRA_LINK_FLAGS="-llog" \
    -DBUILD_EXTENSIONS=${DUCKDB_EXTENSIONS} \
    -DENABLE_EXTENSION_AUTOLOADING=1 \
    -DENABLE_EXTENSION_AUTOINSTALL=1 \
    -DCMAKE_VERBOSE_MAKEFILE=on \
    -DANDROID_PLATFORM=${ANDROID_PLATFORM} \
    -DLOCAL_EXTENSION_REPO="" \
    -DOVERRIDE_GIT_DESCRIBE="" \
    -DDUCKDB_EXPLICIT_PLATFORM=${PLATFORM_NAME} \
    -DBUILD_UNITTESTS=0 \
    -DBUILD_SHELL=1 \
    -DANDROID_ABI=${ANDROID_ABI} \
    -DCMAKE_TOOLCHAIN_FILE=${ANDROID_NDK}/build/cmake/android.toolchain.cmake \
    -DCMAKE_BUILD_TYPE=Release ../..
cmake \
    --build . \
    --config Release
```

7. For the `arm64-v8a` ABI, the build will produce the `build/android_arm64-v8a/duckdb` and `build/android_arm64-v8a/src/libduckdb.so` binaries.

Building the CLI in Termux

1. To build the command line client in the Termux application, install the following packages:

```
pkg install -y git ninja clang cmake python3
```

2. Set the list of DuckDB extensions to build. These will be statically linked in the binary. For example:

```
DUCKDB_EXTENSIONS="icu;json"
```

3. Build DuckDB as follows:

```
mkdir build
cd build
export LDFLAGS="-llog"
cmake \
   -G "Ninja" \
   -DBUILD_EXTENSIONS="${DUCKDB_EXTENSIONS}" \
   -DDUCKDB_EXPLICIT_PLATFORM=linux_arm64_android \
   -DCMAKE_BUILD_TYPE=Release \
   ..
cmake --build . --config Release
```

Note that you can also use the Python client on Termux:

```
pip install --pre --upgrade duckdb
```

Troubleshooting

Log Library Is Missing

Problem: The build throws the following error:

```
ld.lld: error: undefined symbol: __android_log_write
```

Solution: Make sure the log library is linked:

```
export LDFLAGS="-llog"
```

Linux

Prerequisites

On Linux, install the required packages with the package manager of your distribution.

Ubuntu and Debian

CLI Client

On Ubuntu and Debian (and also MX Linux, Linux Mint, etc.), the requirements for building the DuckDB CLI client are the following:

```
sudo apt-get update
sudo apt-get install -y git g++ cmake ninja-build libssl-dev
git clone https://github.com/duckdb/duckdb
cd duckdb
GEN=ninja make
```

Fedora, CentOS and Red Hat

CLI Client

The requirements for building the DuckDB CLI client on Fedora, CentOS, Red Hat, AlmaLinux, Rocky Linux, etc. are the following:

```
sudo yum install -y git g++ cmake ninja-build openssl-devel
git clone https://github.com/duckdb/duckdb
cd duckdb
GEN=ninja make
```

Alpine Linux

CLI Client

The requirements for building the DuckDB CLI client on Alpine Linux are the following:

```
apk add g++ git make cmake ninja
git clone https://github.com/duckdb/duckdb
cd duckdb
GEN=ninja make
```

Performance with musl libc

Note that Alpine Linux uses musl libc as its C standard library. DuckDB binaries built with musl libc have lower performance compared to the glibc variants: for some workloads, the slowdown can be more than 5×. Therefore, it's recommended to use glibc for performance-oriented workloads.

Distribution for the `linux_*_musl` platforms

Starting with DuckDB v1.2.0, *extensions* are distributed for the `linux_amd64_musl` platform (but not yet for the `linux_amd64_musl` platform). However, there are no official *DuckDB binaries* distributed for musl libc but it can be build with it manually following the instructions on this page.

Python Client on Alpine Linux

Currently, installing the DuckDB Python on Alpine Linux requires compilation from source. To do so, install the required packages before running `pip`:

```
apk add g++ py3-pip python3-dev
pip install duckdb
```

Using the DuckDB CLI Client on Linux

Once the build finishes successfully, you can find the `duckdb` binary in the `build` directory:

```
build/release/duckdb
```

For different build configurations (`debug`, `relassert`, etc.), please consult the "Build Configurations" page.

Building Using Extension Flags

To build using extension flags, set the `CORE_EXTENSIONS` flag to the list of extensions that you want to be build. For example:

```
CORE_EXTENSIONS='autocomplete;httpfs;icu;json;tpch' GEN=ninja make
```

Troubleshooting

R Package on Linux AArch64: `too many GOT entries` Build Error

Problem: Building the R package on Linux running on an ARM64 architecture (AArch64) may result in the following error message:

```
/usr/bin/ld: /usr/include/c++/10/bits/basic_string.tcc:206:
warning: too many GOT entries for -fpic, please recompile with -fPIC
```

Solution: Create or edit the `~/.R/Makevars` file. This example also contains the MAKEFLAGS setting to parallelize the build:

```
ALL_CXXFLAGS = $(PKG_CXXFLAGS) -fPIC $(SHLIB_CXXFLAGS) $(CXXFLAGS)
MAKEFLAGS = -j$(nproc)
```

Building the httpfs Extension Fails

Problem: When building the `httpfs` extension on Linux, the build may fail with the following error.

```
CMake Error at /usr/share/cmake-3.22/Modules/FindPackageHandleStandardArgs.cmake:230 (message):
  Could NOT find OpenSSL, try to set the path to OpenSSL root folder in the
  system variable OPENSSL_ROOT_DIR (missing: OPENSSL_CRYPTO_LIBRARY
  OPENSSL_INCLUDE_DIR)
```

Solution: Install the `libssl-dev` library.

```
sudo apt-get install -y libssl-dev
```

Then, build with:

```
GEN=ninja CORE_EXTENSIONS="httpfs" make
```

macOS

Prerequisites

Install Xcode and Homebrew. Then, install the required packages with:

```
brew install git cmake ninja
```

Building DuckDB

Clone and build DuckDB as follows.

```
git clone https://github.com/duckdb/duckdb
cd duckdb
GEN=ninja make
```

Once the build finishes successfully, you can find the `duckdb` binary in the `build` directory:

```
build/release/duckdb
```

For different build configurations (`debug`, `relassert`, etc.), please consult the Build Configurations page.

Troubleshooting

Build Failure: `'string' file not found`

Problem: The build fails on macOS with following error:

```
FAILED: third_party/libpg_query/CMakeFiles/duckdb_pg_query.dir/src_backend_nodes_list.cpp.o
```

```
/Library/Developer/CommandLineTools/usr/bin/c++ -DDUCKDB_BUILD_LIBRARY -DEXT_VERSION_
PARQUET=\"9cba6a2a03\" -I/Users/builder/external/duckdb/src/include
-I/Users/builder/external/duckdb/third_party/fsst -I/Users/builder/external/duckdb/third_
party/fmt/include -I/Users/builder/external/duckdb/third_party/hyperloglog
-I/Users/builder/external/duckdb/third_party/fastpforlib -I/Users/builder/external/duckdb/third_
party/skiplist -I/Users/builder/external/duckdb/third_party/fast_float
-I/Users/builder/external/duckdb/third_party/re2 -I/Users/builder/external/duckdb/third_party/miniz
-I/Users/builder/external/duckdb/third_party/utf8proc/include -I/Users/builder/external/duckdb/third_
party/concurrentqueue -I/Users/builder/external/duckdb/third_party/pcg
-I/Users/builder/external/duckdb/third_party/tdigest -I/Users/builder/external/duckdb/third_
party/mbedtls/include -I/Users/builder/external/duckdb/third_party/jaro_winkler
-I/Users/builder/external/duckdb/third_party/yyjson/include -I/Users/builder/external/duckdb/third_
party/libpg_query/include -O3 -DNDEBUG -O3 -DNDEBUG   -std=c++11 -arch arm64 -isysroot
/Library/Developer/CommandLineTools/SDKs/MacOSX15.1.sdk -fPIC -fvisibility=hidden -fcolor-diagnostics -w
-MD -MT third_party/libpg_query/CMakeFiles/duckdb_pg_query.dir/src_backend_nodes_list.cpp.o -MF third_
party/libpg_query/CMakeFiles/duckdb_pg_query.dir/src_backend_nodes_list.cpp.o.d -o third_party/libpg_
query/CMakeFiles/duckdb_pg_query.dir/src_backend_nodes_list.cpp.o -c
/Users/builder/external/duckdb/third_party/libpg_query/src_backend_nodes_list.cpp
In file included from /Users/builder/external/duckdb/third_party/libpg_query/src_backend_nodes_
list.cpp:35:
/Users/builder/external/duckdb/third_party/libpg_query/include/pg_functions.hpp:4:10: fatal error:
'string' file not found
    4 | #include <string>
```

Solution: Users report that reinstalling Xcode fixed their problem. See related discussions on the DuckDB GitHub issues and on Stack Overflow.

> **Warning.** Attempting to reinstall your Xcode suite may impact other applications on your system. Proceed with caution.

```
sudo rm -rf /Library/Developer/CommandLineTools
xcode-select --install
```

Debug Build Prints malloc Warning

Problem: The debug build on macOS prints a `malloc` warning, e.g.:

```
duckdb(83082,0x205b30240) malloc: nano zone abandoned due to inability to reserve vm space.
```

Solution: To prevent this, set the `MallocNanoZone` flag to 0:

```
MallocNanoZone=0 make debug
```

To apply this change for your future terminal sessions, you can add the following to your `~/.zshrc` file:

```
export MallocNanoZone=0
```

Raspberry Pi

DuckDB is not officially distributed for the Raspberry Pi OS (previously called Raspbian). You can build it following the instructions on this page.

Raspberry Pi (64-bit)

First, install the required build packages:

```
sudo apt-get update
sudo apt-get install -y git g++ cmake ninja-build
```

Then, clone and build it as follows:

```
git clone https://github.com/duckdb/duckdb
cd duckdb
GEN=ninja CORE_EXTENSIONS="icu;json" make
```

Finally, run it:

```
build/release/duckdb
```

Raspberry Pi (32-bit)

On 32-bit Raspberry Pi boards, you need to add the `-latomic` link flag. As extensions are not distributed for this platform, it's recommended to also include them in the build. For example:

```
mkdir build
cd build
cmake .. \
    -DCORE_EXTENSIONS="httpfs;json;parquet" \
    -DDUCKDB_EXTRA_LINK_FLAGS="-latomic"
make -j4
```

Windows

On Windows, DuckDB requires the Microsoft Visual C++ Redistributable package both as a build-time and runtime dependency. Note that unlike the build process on UNIX-like systems, the Windows builds directly call CMake.

Visual Studio

To build DuckDB on Windows, we recommend using the Visual Studio compiler. To use it, follow the instructions in the CI workflow:

```
python scripts/windows_ci.py
cmake \
    -DCMAKE_BUILD_TYPE=Release \
    -DCMAKE_GENERATOR_PLATFORM=x64 \
    -DENABLE_EXTENSION_AUTOLOADING=1 \
    -DENABLE_EXTENSION_AUTOINSTALL=1 \
    -DDUCKDB_EXTENSION_CONFIGS="${GITHUB_WORKSPACE}/.github/config/bundled_extensions.cmake" \
    -DDISABLE_UNITY=1 \
    -DOVERRIDE_GIT_DESCRIBE="$OVERRIDE_GIT_DESCRIBE"
cmake --build . --config Release --parallel
```

MSYS2 and MinGW64

DuckDB on Windows can also be built with MSYS2 and MinGW64. Note that this build is only supported for compatibility reasons and should only be used if the Visual Studio build is not feasible on a given platform. To build DuckDB with MinGW64, install the required dependencies using Pacman. When prompted with `Enter a selection (default=all)`, select the default option by pressing `Enter`.

```
pacman -Syu git mingw-w64-x86_64-toolchain mingw-w64-x86_64-cmake mingw-w64-x86_64-ninja
git clone https://github.com/duckdb/duckdb
cd duckdb
cmake -G "Ninja" -DCMAKE_BUILD_TYPE=Release -DBUILD_EXTENSIONS="icu;parquet;json"
cmake --build . --config Release
```

Once the build finishes successfully, you can find the `duckdb.exe` binary in the repository's directory:

```
./duckdb.exe
```

Building the Go Client

Building on Windows may result in the following error:

```
go build
```

```
collect2.exe: error: ld returned 5 exit status
```

GitHub user vdmitriyev shared instructions for building the DuckDB Go client on Windows:

1. Get four files (`.dll, .lib, .hpp, .h`) from the `libduckdb-windows-amd64.zip` archive.
2. Place them to, e.g.,: `C:\DuckDB-Go\libs\`.
3. Install the dependencies following the go-duckdb project.
4. Build your project using the following instructions:

   ```
   set PATH=C:\DuckDB-Go\libs\;%PATH%
   set CGO_CFLAGS=-IC:\DuckDB-Go\libs\
   set CGO_LDFLAGS=-LC:\DuckDB-Go\libs\ -lduckdb
   go build
   ```

Python

The DuckDB Python package lives in the main DuckDB source on GitHub under the `/tools/pythonpkg/` folder. It uses pybind11 to create Python bindings with DuckDB.

Prerequisites

For everything described on this page we make the following assumptions:

1. You have a working copy of the DuckDB source (including the git tags) and you run commands from the root of the source.
2. You have a suitable Python installation available in a dedicated virtual environment.

1. DuckDB Repository

Make sure you have checked out the DuckDB source and that you are in its root. E.g.:

```
git clone https://github.com/duckdb/duckdb
...
cd duckdb
```

If you've *forked* DuckDB, you may run into trouble when building the Python package when you haven't pulled in the tags.

```
# Check your remotes
git remote -v

# If you don't see upstream git@github.com:duckdb/duckdb.git, then add it
git remote add upstream git@github.com:duckdb/duckdb.git

# Now you can pull & push the tags
git fetch --tags upstream
git push --tags
```

2. Python Virtual Environment

For everything described here you will need a suitable Python installation. While you technically might be able to use your system Python, we **strongly** recommend you use a Python virtual environment. A virtual environment isolates dependencies and, depending on the tooling you use, gives you control over which Python interpreter you use. This way you don't pollute your system-wide Python with the different packages you need for your projects.

While we use Python's built-in `venv` module in our examples below, and technically this might (or might not!) work for you, we also **strongly** recommend use a tool like astral uv (or Poetry, conda, etc.) that allows you to manage *both* Python interpreter versions and virtual environments.

Create and activate a virtual env as follows:

```
# Create a virtual environment in the .venv folder (in the duckdb source root)
python3 -m venv --prompt duckdb .venv

# Activate the virtual env
source .venv/bin/activate
```

Make sure you have a modern enough version of `pip` available in your virtual env:

```
# Print pip's help
python3 -m pip install --upgrade pip
```

If that fails with `No module named pip` and you use `uv`, then run:

```
# Install pip
uv pip install pip
```

Building from Source

Below are a number of options to build the Python library from source, with or without debug symbols, and with a default or custom set of extensions. Make sure to check out the DuckDB build documentation if you run into trouble building the DuckDB main library.

Default Release, Debug Build or Cloud Storage

The following will build the package with the default set of extensions (json, parquet, icu and core_functions).

Release Build

```
GEN=ninja BUILD_PYTHON=1 make release
```

Debug Build

```
GEN=ninja BUILD_PYTHON=1 make debug
```

Verify

```
python3 -c "import duckdb; print(duckdb.sql('SELECT 42').fetchall())"
```

Adding Extensions

Before thinking about statically linking extensions you should know that the Python package currently doesn't handle linked in extensions very well. If you don't really need to have an extension baked in than the advice is to just stick to installing them at runtime. See `tools/pythonpkg/duckdb_extension_config.cmake` for the default list of extensions that are built with the Python package. Any other extension should be considered problematic.

Having said that, if you do want to give it a try, here's how.

> For more details on building DuckDB extensions look at the documentation.

The DuckDB build process follows the following logic for building extensions:

1. First compose the complete set of extensions that might be included in the build.
2. Then compose the complete set of extensions that should be excluded from the build.
3. Assemble the final set of extensions to be compiled by subtracting the set of excluded extensions from the set of included extensions.

The following mechanisms add to the set of ***included* extensions**:

Mechanism	Syntax / Example
Built-in extensions enabled by default	`extension/extension_config.cmake` (≈30 built-ins)
Python package extensions enabled by default	`tools/pythonpkg/duckdb_extension_config.cmake` (`json;parquet;icu`)
Semicolon-separated include list	`DUCKDB_EXTENSIONS=fts;tpch;json`
Flags	`BUILD_TPCH=1`, `BUILD_JEMALLOC=1`, `BUILD_FTS=1`, …
Presets	`BUILD_ALL_EXT=1` - Build all in-tree extensions`BUILD_ALL_IT_EXT=1` - *Only* build in-tree extensions`BUILD_ALL_OOT_EXT=1` - Build all out-of-tree extensions
Custom config file(s)	`DUCKDB_EXTENSION_CONFIGS=path/to/my.cmake`
Core-only overrides *only relevant with `DISABLE_BUILTIN_EXTENSIONS=1`*	`CORE_EXTENSIONS=httpfs;fts`

The following mechanisms add to the set of ***excluded* extensions**:

Mechanism	Syntax / Example
Semicolon-separated skip list	`SKIP_EXTENSIONS=parquet;jemalloc`
Flags	`DISABLE_PARQUET=1`, `DISABLE_CORE_FUNCTIONS=1`, …
"No built-ins" switch *Throws out* every *statically linked extension* ***except*** `core_functions`. *Use `CORE_EXTENSIONS=…` to whitelist a subset back in.*	`DISABLE_BUILTIN_EXTENSIONS=1`

Show All Installed Extensions

```
python3 -c "import duckdb; print(duckdb.sql('SELECT extension_name, installed, description FROM duckdb_extensions();'))"
```

Development Environment

This section walks you through the following steps:

- Creating a CMake profile for development
- Debugging the Python extension code with lldb

You can do this either on the CLI or from an IDE. The documentation below shows the configuration for CLion, but you should be able to get it to work with other IDEs like VSCode as well.

Debugging from the CLI

Run this to configure the CMake profile needed to debug on the CLI:

```
GEN=ninja BUILD_PYTHON=1 PYTHON_DEV=1 make debug
```

This will take care of the following:

- Builds both the main DuckDB library and the Python library with debug symbols.
- Generates a `compile-commands.json` file that includes CPython and pybind11 headers so that intellisense and clang-tidy checks work in your IDE.
- Installs the required Python dependencies in your virtual env.

Once the build completes, do a sanity check to make sure everything works:

```
python3 -c "import duckdb; print(duckdb.sql('SELECT 42').fetchall())"
```

Debugging

The basic recipe is to start `lldb` with your virtual env's Python interpreter and your script, then set a breakpoint and run your script. For example, given a script `dataframe.df` with the following contents:

```
import duckdb
print(duckdb.sql("select * from range(1000)").df())
```

The following should work:

```
lldb -- .venv/bin/python3 my_script.py
```

```
# Set a breakpoint
(lldb) br s -n duckdb::DuckDBPyRelation::FetchDF
Breakpoint 1: no locations (pending).
WARNING:  Unable to resolve breakpoint to any actual locations.
# The above warning is harmless - the library hasn't been imported yet

# Run the script
(lldb) r
...
    frame #0: 0x000000013025833c
duckdb.cpython-310-darwin.so`duckdb::DuckDBPyRelation::FetchDF(this=0x00006000012f8d20, date_as_
object=false) at pyrelation.cpp:808:7
   805   }
   806
   807   PandasDataFrame DuckDBPyRelation::FetchDF(bool date_as_object) {
-> 808     if (!result) {
   809       if (!rel) {
   810         return py::none();
   811       }
Target 0: (python3) stopped.
```

Debugging in an IDE / CLion

You should be able to get debugging going in an IDE that support `lldb`. Below are the instructions for CLion but you can copy the settings for your favorite IDE.

Configure a CMake Debug Profile

The following CMake profile enables Intellisense and clang-tidy by generating a `compile-commands.json` file so your IDE knows how to inspect the source code, and makes sure that the Python package will be built and installed in your Python virtual env.

Under **Settings** -> **Build, Execution, Deployment** -> **CMake**, add a profile and set the fields as follows:

- **Name**: Debug
- **Build type**: Debug
- **Generator**: Ninja
- **CMake Options** (on a single line):

```
-DCMAKE_PREFIX_PATH=$CMakeProjectDir$/.venv;$CMAKE_PREFIX_PATH
-DPython3_EXECUTABLE=$CMakeProjectDir$/.venv/bin/python3
-DBUILD_PYTHON=1
-DPYTHON_DEV=1
```

Create a Run Config for Debugging

Under **Run** -> **Edit Configurations...** create a new **CMake Application**. Use the following values:

- **Name**: Python Debug
- **Target**: `All targets`
- **Executable**: `[ABS_PATH_TO_YOUR_VENV]/bin/python3` (careful: this is a symlink and sometimes an IDE might try and follow it and fill in the path to the actual executable, but that will not work)
- **Program arguments**: `$FilePath$`
- **Working directory**: `$ProjectFileDir$`
- **Before Launch**: `Build` (this should already be set)

That should be enough: save and close.

Now you can set a breakpoint in a C++ file. You then open your Python script in your editor and use this config and run `Python Debug` in debug mode.

Development and Stubs

`*.pyi` stubs in `duckdb-stubs` are manually maintained. The connection-related stubs are generated using dedicated scripts in `tools/pythonpkg/scripts/`:

- `generate_connection_stubs.py`
- `generate_connection_wrapper_stubs.py`

These stubs are important for autocomplete in many IDEs, as static-analysis based language servers can't introspect `duckdb`'s binary module.

To verify the stubs match the actual implementation:

```
python3 -m pytest tests/stubs
```

If you add new methods to the DuckDB Python API, you'll need to manually add corresponding type hints to the stub files.

What are py::objects and py::handles?

These are classes provided by pybind11, the library we use to manage our interaction with the Python environment. `py::handle` is a direct wrapper around a raw PyObject* and does not manage any references. `py::object` is similar to py::handle but it can handle refcounts.

I say *can* because it doesn't have to, using `py::reinterpret_borrow<py::object>(...)` we can create a non-owning `py::object`, this is essentially just a py::handle but py::handle can't be used if the prototype requires a `py::object`.

`py::reinterpret_steal<py::object>(...)` creates an owning `py::object`, this will increase the refcount of the python object and will decrease the refcount when the `py::object` goes out of scope.

When directly interacting with python functions that return a `PyObject*`, such as `PyDateTime_DATE_GET_TZINFO`, you should generally wrap the call in `py::reinterpret_steal` to take ownership of the returned object.

Troubleshooting

Pip Fails with `No names found, cannot describe anything`

If you've forked DuckDB you may run into trouble when building the Python package when you haven't pulled in the tags.

```
# Check your remotes
git remote -v

# If you don't see upstream git@github.com:duckdb/duckdb.git, then add it
git remote add upstream git@github.com:duckdb/duckdb.git

# Now you can pull & push the tags
git fetch --tags upstream
git push --tags
```

Building with the httpfs Extension Fails

The build fails on OSX when both the `httpfs` extension and the Python package are included:

```
ld: library not found for -lcrypto
clang: error: linker command failed with exit code 1 (use -v to see invocation)
error: command '/usr/bin/clang++' failed with exit code 1
ninja: build stopped: subcommand failed.
make: *** [release] Error 1
```

Linking in the httpfs extension is problematic. Please install it at runtime, if you can.

Importing DuckDB Fails with `symbol not found in flat namespace`

If you seen an error that looks like this:

```
ImportError: dlopen(/usr/bin/python3/site-packages/duckdb/duckdb.cpython-311-darwin.so, 0x0002): symbol
not found in flat namespace '_MD5_Final'
```

... then you've probably tried to link in a problematic extension. As mentioned above: `tools/pythonpkg/duckdb_extension_config.cmake` contains the default list of extensions that are built with the Python package. Any other extension might cause problems.

Python Fails with `No module named 'duckdb.duckdb'`

If you're in `tools/pythonpkg` and try to `import duckdb` you might see:

```
Traceback (most recent call last):
  File "<string>", line 1, in <module>
  File "/duckdb/tools/pythonpkg/duckdb/__init__.py", line 4, in <module>
    import duckdb.functional as functional
  File "/duckdb/tools/pythonpkg/duckdb/functional/__init__.py", line 1, in <module>
    from duckdb.duckdb.functional import (
ModuleNotFoundError: No module named 'duckdb.duckdb'
```

This is because Python imported from the `duckdb` directory (i.e. `tools/pythonpkg/duckdb/`), rather than from the installed package. You should start your interpreter from a different directory instead.

R

This page contains instructions for building the R client library.

The Build Only Uses a Single Thread

Problem: By default, R compiles packages using a single thread, which causes the build to be slow.

Solution: To parallelize the compilation, create or edit the `~/.R/Makevars` file, and add a line like the following:

```
MAKEFLAGS = -j8
```

The above will parallelize the compilation using 8 threads. On Linux/macOS, you can add the following to use all of the machine's threads:

```
MAKEFLAGS = -j$(nproc)
```

However, note that, the more threads that are used, the higher the RAM consumption. If the system runs out of RAM while compiling, then the R session will crash.

Troubleshooting

This page contains solutions to common problems reported by users. If you have platform-specific issues, make sure you also consult the troubleshooting guide for your platform such as the one for Linux builds.

The Build Runs Out of Memory

Problem: Ninja parallelizes the build, which can cause out-of-memory issues on systems with limited resources. These issues have also been reported to occur on Alpine Linux, especially on machines with limited resources.

Solution: Avoid using Ninja by setting the Makefile generator to empty via `GEN=`:

```
GEN= make
```

Unofficial and Unsupported Platforms

> **Warning.** The platforms listed on this page are not officially supported. The build instructions are provided on a best-effort basis. Community contributions are very welcome.

DuckDB is built and distributed for several platform with different levels of support. DuckDB *can be built* for other platforms with varying levels of success. This page provides an overview of these with the intent to clarify which platforms can be expected to work.

32-bit Architectures

32-bit architectures are officially not supported but it is possible to build DuckDB manually for some of these platforms. For example, see the build instructions for 32-bit Raspberry Pi boards.

Note that 32-bit platforms are limited to using 4 GiB RAM due to the amount of addressable memory.

Big-Endian Architectures

Big-endian architectures (such as PowerPC) are not supported by DuckDB. While DuckDB can likely be built on such architectures, the resulting binary may exhibit correctness errors on certain operations. Therefore, it's use is not recommended.

RISC-V Architectures

The user "LivingLinux" on Bluesky managed to build DuckDB for a RISC-V profile and published a video about it. The instruction to build DuckDB, including the `fts` extension, is the following:

```
GEN=ninja \
    CC='gcc-14 -march=rv64gcv_zicsr_zifencei_zihintpause_zvl256b' \
    CXX='g++-14 -march=rv64gcv_zicsr_zifencei_zihintpause_zvl256b' \
    CORE_EXTENSIONS='fts' \
    make
```

For those who do not have a RISC-V chip development environment, you can cross-compile DuckDB using latest g++-riscv64-linux-gnu :

```
GEN=ninja \
    CC='riscv64-linux-gnu-gcc -march=rv64gcv_zicsr_zifencei_zihintpause_zvl256b' \
    CXX='riscv64-linux-gnu-g++ -march=rv64gcv_zicsr_zifencei_zihintpause_zvl256b' \
    make
```

For more reference information on DuckDB RISC-V cross-compiling, see the mocusez/duckdb-riscv-ci and DuckDB Pull Request #16549

Benchmark Suite

DuckDB has an extensive benchmark suite. When making changes that have potential performance implications, it is important to run these benchmarks to detect potential performance regressions.

Getting Started

To build the benchmark suite, run the following command in the DuckDB repository:

```
BUILD_BENCHMARK=1 CORE_EXTENSIONS='tpch' make
```

Listing Benchmarks

To list all available benchmarks, run:

```
build/release/benchmark/benchmark_runner --list
```

Running Benchmarks

Running a Single Benchmark

To run a single benchmark, issue the following command:

```
build/release/benchmark/benchmark_runner benchmark/micro/nulls/no_nulls_addition.benchmark
```

The output will be printed to `stdout` in CSV format, in the following format:

```
name    run timing
benchmark/micro/nulls/no_nulls_addition.benchmark   1   0.121234
benchmark/micro/nulls/no_nulls_addition.benchmark   2   0.121702
benchmark/micro/nulls/no_nulls_addition.benchmark   3   0.122948
benchmark/micro/nulls/no_nulls_addition.benchmark   4   0.122534
benchmark/micro/nulls/no_nulls_addition.benchmark   5   0.124102
```

You can also specify an output file using the `--out` flag. This will write only the timings (delimited by newlines) to that file.

```
build/release/benchmark/benchmark_runner benchmark/micro/nulls/no_nulls_addition.benchmark
--out=timings.out
```

The output will contain the following:

```
0.182472
0.185027
0.184163
0.185281
0.182948
```

Running Multiple Benchmark Using a Regular Expression

You can also use a regular expression to specify which benchmarks to run. Be careful of shell expansion of certain regex characters (e.g., `*` will likely be expanded by your shell, hence this requires proper quoting or escaping).

```
build/release/benchmark/benchmark_runner "benchmark/micro/nulls/.*"
```

Running All Benchmarks

Not specifying any argument will run all benchmarks.

```
build/release/benchmark/benchmark_runner
```

Other Options

The `--info` flag gives you some other information about the benchmark.

```
build/release/benchmark/benchmark_runner benchmark/micro/nulls/no_nulls_addition.benchmark --info
```

```
display_name:NULL Addition (no nulls)
group:micro
subgroup:nulls
```

The `--query` flag will print the query that is run by the benchmark.

```
SELECT min(i + 1) FROM integers;
```

The `--profile` flag will output a query tree.

Testing

Overview

How is DuckDB Tested?

Testing is vital to make sure that DuckDB works properly and keeps working properly. For that reason, we put a large emphasis on thorough and frequent testing:

- We run a batch of small tests on every commit using GitHub Actions, and run a more exhaustive batch of tests on pull requests and commits in the `main` branch.
- We use a fuzzer, which automatically reports of issues found through fuzzing DuckDB.
- We use SQLsmith for generating random queries.

sqllogictest Introduction

For testing plain SQL, we use an extended version of the SQL logic test suite, adopted from SQLite. Every test is a single self-contained file located in the `test/sql` directory. To run tests located outside of the default `test` directory, specify `--test-dir <root_directory>` and make sure provided test file paths are relative to that root directory.

The test describes a series of SQL statements, together with either the expected result, a `statement ok` indicator, or a `statement error` indicator. An example of a test file is shown below:

```
# name: test/sql/projection/test_simple_projection.test
# group [projection]

# enable query verification
statement ok
PRAGMA enable_verification

# create table
statement ok
CREATE TABLE a (i INTEGER, j INTEGER);

# insertion: 1 affected row
statement ok
INSERT INTO a VALUES (42, 84);

query II
SELECT * FROM a;
----
42  84
```

In this example, three statements are executed. The first statements are expected to succeed (prefixed by `statement ok`). The third statement is expected to return a single row with two columns (indicated by `query II`). The values of the row are expected to be 42 and 84 (separated by a tab character). For more information on query result verification, see the result verification section.

The top of every file should contain a comment describing the name and group of the test. The name of the test is always the relative file path of the file. The group is the folder that the file is in. The name and group of the test are relevant because they can be used to execute

only that test in the unittest group. For example, if we wanted to execute *only* the above test, we would run the command `unittest test/sql/projection/test_simple_projection.test`. If we wanted to run all tests in a specific directory, we would run the command `unittest "[projection]"`.

Any tests that are placed in the `test` directory are automatically added to the test suite. Note that the extension of the test is significant. The sqllogictests should either use the `.test` extension, or the `.test_slow` extension. The `.test_slow` extension indicates that the test takes a while to run, and will only be run when all tests are explicitly run using `unittest *`. Tests with the extension `.test` will be included in the fast set of tests.

Query Verification

Many simple tests start by enabling query verification. This can be done through the following PRAGMA statement:

```
statement ok
PRAGMA enable_verification
```

Query verification performs extra validation to ensure that the underlying code runs correctly. The most important part of that is that it verifies that optimizers do not cause bugs in the query. It does this by running both an unoptimized and optimized version of the query, and verifying that the results of these queries are identical.

Query verification is very useful because it not only discovers bugs in optimizers, but also finds bugs in e.g., join implementations. This is because the unoptimized version will typically run using cross products instead. Because of this, query verification can be very slow to do when working with larger data sets. It is therefore recommended to turn on query verification for all unit tests, except those involving larger data sets (more than ~10-100 rows).

Editors & Syntax Highlighting

The sqllogictests are not exactly an industry standard, but several other systems have adopted them as well. Parsing sqllogictests is intentionally simple. All statements have to be separated by empty lines. For that reason, writing a syntax highlighter is not extremely difficult.

A syntax highlighter exists for Visual Studio Code. We have also made a fork that supports the DuckDB dialect of the sqllogictests. You can use the fork by installing the original, then copying the `syntaxes/sqllogictest.tmLanguage.json` into the installed extension (on macOS this is located in `~/.vscode/extensions/benesch.sqllogictest-0.1.1`).

A syntax highlighter is also available for CLion. It can be installed directly on the IDE by searching SQLTest on the marketplace. A GitHub repository is also available, with extensions and bug reports being welcome.

Temporary Files

For some tests (e.g., CSV/Parquet file format tests) it is necessary to create temporary files. Any temporary files should be created in the temporary testing directory. This directory can be used by placing the string `__TEST_DIR__` in a query. This string will be replaced by the path of the temporary testing directory.

```
statement ok
COPY csv_data TO '__TEST_DIR__/output_file.csv.gz' (COMPRESSION gzip);
```

Require & Extensions

To avoid bloating the core system, certain functionality of DuckDB is available only as an extension. Tests can be build for those extensions by adding a `require` field in the test. If the extension is not loaded, any statements that occurs after the require field will be skipped. Examples of this are `require parquet` or `require icu`.

Another usage is to limit a test to a specific vector size. For example, adding `require vector_size 512` to a test will prevent the test from being run unless the vector size greater than or equal to 512. This is useful because certain functionality is not supported for low vector sizes, but we run tests using a vector size of 2 in our CI.

Writing Tests

Development and Testing

It is crucial that any new features that get added have correct tests that not only test the "happy path", but also test edge cases and incorrect usage of the feature. In this section, we describe how DuckDB tests are structured and how to make new tests for DuckDB.

The tests can be run by running the `unittest` program located in the `test` folder. For the default compilations this is located in either `build/release/test/unittest` (release) or `build/debug/test/unittest` (debug).

Philosophy

When testing DuckDB, we aim to route all the tests through SQL. We try to avoid testing components individually because that makes those components more difficult to change later on. As such, almost all of our tests can (and should) be expressed in pure SQL. There are certain exceptions to this, which we will discuss in Catch Tests. However, in most cases you should write your tests in plain SQL.

Frameworks

SQL tests should be written using the sqllogictest framework.

C++ tests can be written using the Catch framework.

Client Connector Tests

DuckDB also has tests for various client connectors. These are generally written in the relevant client language, and can be found in `tools/*/tests`. They also double as documentation of what should be doable from a given client.

Functions for Generating Test Data

DuckDB has built-in functions for generating test data.

`test_all_types` Function

The `test_all_types` table function generates a table whose columns correspond to types (`BOOL`, `TINYINT`, etc.). The table has three rows encoding the minimum value, the maximum value, and the `NULL` value for each type.

```
FROM test_all_types();
```

```
┌──────────┬──────────┬──────────┬──────────────┬──────────────────────┬──────────────────────┬──┬──────────────────────
│  bool    │ tinyint  │ smallint │     int      │        bigint        │       hugeint        │ … │
struct        │   struct_of_arrays   │   array_of_structs   │        map          │        union         │
│ boolean  │  int8    │  int16   │    int32     │        int64         │        int128        │   │ struct(a
integer, …  │ struct(a integer[]…  │ struct(a integer, …  │ map(varchar, varch…  │ union("name" varch…  │
├──────────┼──────────┼──────────┼──────────────┼──────────────────────┼──────────────────────┼──┼──────────────────────
│ false    │    -128  │   -32768 │ -2147483648  │ -9223372036854775808 │  -17014118346046923… │ … │ {'a':
NULL, 'b': N…  │ {'a': NULL, 'b': N…  │ []                   │ {}                   │ Frank                 │
│ true     │     127  │    32767 │  2147483647  │  9223372036854775807 │  170141183460469231… │ … │ {'a':
42, 'b': 🦆…   │ {'a': [42, 999, NU…  │ [{'a': NULL, 'b': …  │ {key1=🦆🦆🦆🦆🦆🦆… │ 5                    │
│ NULL     │    NULL  │     NULL │        NULL  │                 NULL  │                 NULL  │ … │ NULL
│ NULL                  │ NULL                  │ NULL                  │ NULL                  │
```

```
┴─────────┴─────────┴─────────┴─────────┴──────────────────────┴──────────────────────┴───┴──────────────────────┴
  3 rows
44 columns (11 shown)
```

`test_vector_types` Function

The `test_vector_types` table function takes *n* arguments `col1`, ..., `coln` and an optional `BOOLEAN` argument `all_flat`. The function generates a table with *n* columns `test_vector`, `test_vector2`, ..., `test_vectorn`. In each row, each field contains values conforming to the type of their respective column.

```
FROM test_vector_types(NULL::BIGINT);
```

```
┌──────────────────────┐
│     test_vector      │
│        int64         │
├──────────────────────┤
│ -9223372036854775808 │
│  9223372036854775807 │
│                 NULL │
│          ·           │
└──────────────────────┘
```

```
FROM test_vector_types(NULL::ROW(i INTEGER, j VARCHAR, k DOUBLE), NULL::TIMESTAMP);
```

```
┌──────────────────────────────────────────────────────────────────────────────┬──────────────────────────┐
│                                 test_vector                                  │       test_vector2       │
│                 struct(i integer, j varchar, k double)                       │        timestamp         │
├──────────────────────────────────────────────────────────────────────────────┼──────────────────────────┤
│ {'i': -2147483648, 'j': 🦆🦆🦆🦆🦆, 'k': -1.7976931348623157e+308} │ 290309-12-22 (BC) 00:00:00   │
│ {'i': 2147483647, 'j': goo\0se, 'k': 1.7976931348623157e+308}                │ 294247-01-10 04:00:54.775806 │
│ {'i': NULL, 'j': NULL, 'k': NULL}                                            │ NULL                     │
│                                          ·                                   │                          │
└──────────────────────────────────────────────────────────────────────────────┴──────────────────────────┘
```

`test_vector_types` has an optional argument called `all_flat` of type `BOOL`. This only affects the internal representation of the vector.

```
FROM test_vector_types(NULL::ROW(i INTEGER, j VARCHAR, k DOUBLE), NULL::TIMESTAMP, all_flat = true);
-- the output is the same as above but with a different internal representation
```

Debugging

The purpose of the tests is to figure out when things break. Inevitably changes made to the system will cause one of the tests to fail, and when that happens the test needs to be debugged.

First, it is always recommended to run in debug mode. This can be done by compiling the system using the command `make debug`. Second, it is recommended to only run the test that breaks. This can be done by passing the filename of the breaking test to the test suite as a command line parameter (e.g., `build/debug/test/unittest test/sql/projection/test_simple_projection.test`). For more options on running a subset of the tests see the Triggering which tests to run section.

After that, a debugger can be attached to the program and the test can be debugged. In the sqllogictests it is normally difficult to break on a specific query, however, we have expanded the test suite so that a function called `query_break` is called with the line number `line` as parameter for every query that is run. This allows you to put a conditional breakpoint on a specific query. For example, if we want to break on line number 43 of the test file we can create the following break point:

```
gdb: break query_break if line==43
lldb: break s -n query_break -c line==43
```

You can also skip certain queries from executing by placing `mode skip` in the file, followed by an optional `mode unskip`. Any queries between the two statements will not be executed.

Triggering Which Tests to Run

When running the unittest program, by default all the fast tests are run. A specific test can be run by adding the name of the test as an argument. For the sqllogictests, this is the relative path to the test file. To run only a single test:

```
build/debug/test/unittest test/sql/projection/test_simple_projection.test
```

All tests in a given directory can be executed by providing the directory as a parameter with square brackets. To run all tests in the "projection" directory:

```
build/debug/test/unittest "[projection]"
```

All tests, including the slow tests, can be run by running the tests with an asterisk. To run all tests, including the slow tests:

```
build/debug/test/unittest "*"
```

We can run a subset of the tests using the `--start-offset` and `--end-offset` parameters. To run the tests 200..250:

```
build/debug/test/unittest --start-offset=200 --end-offset=250
```

These are also available in percentages. To run tests 10% - 20%:

```
build/debug/test/unittest --start-offset-percentage=10 --end-offset-percentage=20
```

The set of tests to run can also be loaded from a file containing one test name per line, and loaded using the `-f` command.

```
cat test.list
```

```
test/sql/join/full_outer/test_full_outer_join_issue_4252.test
test/sql/join/full_outer/full_outer_join_cache.test
test/sql/join/full_outer/test_full_outer_join.test
```

To run only the tests labeled in the file:

```
build/debug/test/unittest -f test.list
```

Result Verification

The standard way of verifying results of queries is using the `query` statement, followed by the letter `I` times the number of columns that are expected in the result. After the query, four dashes (`----`) are expected followed by the result values separated by tabs. For example,

```
query II
SELECT 42, 84 UNION ALL SELECT 10, 20;
----
42  84
10  20
```

For legacy reasons the letters R and T are also accepted to denote columns.

> **Deprecated.** DuckDB deprecated the usage of types in the sqllogictest. The DuckDB test runner does not use or need them internally – therefore, only `I` should be used to denote columns.

NULL Values and Empty Strings

Empty lines have special significance for the SQLLogic test runner: they signify an end of the current statement or query. For that reason, empty strings and NULL values have special syntax that must be used in result verification. NULL values should use the string `NULL`, and empty strings should use the string `(empty)`, e.g.:

```
query II
SELECT NULL, ''
----
NULL
(empty)
```

Error Verification

In order to signify that an error is expected, the `statement error` indicator can be used. The `statement error` also takes an optional expected result – which is interpreted as the *expected error message*. Similar to `query`, the expected error should be placed after the four dashes (`----`) following the query. The test passes if the error message *contains* the text under `statement error` – the entire error message does not need to be provided. It is recommended that you only use a subset of the error message, so that the test does not break unnecessarily if the formatting of error messages is changed.

```
statement error
SELECT * FROM non_existent_table;
----
Table with name non_existent_table does not exist!
```

Regex

In certain cases result values might be very large or complex, and we might only be interested in whether or not the result *contains* a snippet of text. In that case, we can use the `<REGEX>:` modifier followed by a certain regex. If the result value matches the regex the test is passed. This is primarily used for query plan analysis.

```
query II
EXPLAIN SELECT tbl.a FROM 'data/parquet-testing/arrow/alltypes_plain.parquet' tbl(a) WHERE a = 1 OR a = 2
----
physical_plan    <REGEX>:.*PARQUET_SCAN.*Filters: a=1 OR a=2.*
```

If we instead want the result *not* to contain a snippet of text, we can use the `<!REGEX>:` modifier.

File

As results can grow quite large, and we might want to re-use results over multiple files, it is also possible to read expected results from files using the `<FILE>` command. The expected result is read from the given file. As convention the file path should be provided as relative to the root of the GitHub repository.

```
query I
PRAGMA tpch(1)
----
<FILE>:extension/tpch/dbgen/answers/sf1/q01.csv
```

Row-Wise vs. Value-Wise Result Ordering

The result values of a query can be either supplied in row-wise order, with the individual values separated by tabs, or in value-wise order. In value wise order the individual *values* of the query must appear in row, column order each on an individual line. Consider the following example in both row-wise and value-wise order:

```
# row-wise
query II
SELECT 42, 84 UNION ALL SELECT 10, 20;
----
42  84
10  20

# value-wise
query II
SELECT 42, 84 UNION ALL SELECT 10, 20;
----
42
84
10
20
```

Hashes and Outputting Values

Besides direct result verification, the sqllogic test suite also has the option of using MD5 hashes for value comparisons. A test using hashes for result verification looks like this:

```
query I
SELECT g, string_agg(x,',') FROM strings GROUP BY g
----
200 values hashing to b8126ea73f21372cdb3f2dc483106a12
```

This approach is useful for reducing the size of tests when results have many output rows. However, it should be used sparingly, as hash values make the tests more difficult to debug if they do break.

After it is ensured that the system outputs the correct result, hashes of the queries in a test file can be computed by adding `mode output_hash` to the test file. For example:

```
mode output_hash

query II
SELECT 42, 84 UNION ALL SELECT 10, 20;
----
42  84
10  20
```

The expected output hashes for every query in the test file will then be printed to the terminal, as follows:

```
================================================================================
SQL Query
SELECT 42, 84 UNION ALL SELECT 10, 20;
================================================================================
4 values hashing to 498c69da8f30c24da3bd5b322a2fd455
================================================================================
```

In a similar manner, `mode output_result` can be used in order to force the program to print the result to the terminal for every query run in the test file.

Result Sorting

Queries can have an optional field that indicates that the result should be sorted in a specific manner. This field goes in the same location as the connection label. Because of that, connection labels and result sorting cannot be mixed.

The possible values of this field are `nosort`, `rowsort` and `valuesort`. An example of how this might be used is given below:

```
query I rowsort
SELECT 'world' UNION ALL SELECT 'hello'
----
hello
world
```

In general, we prefer not to use this field and rely on ORDER BY in the query to generate deterministic query answers. However, existing sqllogictests use this field extensively, hence it is important to know of its existence.

Query Labels

Another feature that can be used for result verification are `query labels`. These can be used to verify that different queries provide the same result. This is useful for comparing queries that are logically equivalent, but formulated differently. Query labels are provided after the connection label or sorting specifier.

Queries that have a query label do not need to have a result provided. Instead, the results of each of the queries with the same label are compared to each other. For example, the following script verifies that the queries `SELECT 42+1` and `SELECT 44-1` provide the same result:

```
query I nosort r43
SELECT 42+1;
----

query I nosort r43
SELECT 44-1;
----
```

Persistent Testing

By default, all tests are run in in-memory mode (unless `--force-storage` is enabled). In certain cases, we want to force the usage of a persistent database. We can initiate a persistent database using the `load` command, and trigger a reload of the database using the `restart` command.

```
# load the DB from disk
load __TEST_DIR__/storage_scan.db

statement ok
CREATE TABLE test (a INTEGER);

statement ok
INSERT INTO test VALUES (11), (12), (13), (14), (15), (NULL)

# ...

restart

query I
SELECT * FROM test ORDER BY a
----
```

```
NULL
11
12
13
14
15
```

Note that by default the tests run with `SET wal_autocheckpoint = '0KB'` – meaning a checkpoint is triggered after every statement. WAL tests typically run with the following settings to disable this behavior:

```
statement ok
PRAGMA disable_checkpoint_on_shutdown

statement ok
PRAGMA wal_autocheckpoint = '1TB'
```

Loops

Loops can be used in sqllogictests when it is required to execute the same query many times but with slight modifications in constant values. For example, suppose we want to fire off 100 queries that check for the presence of the values `0..100` in a table:

```
# create the table 'integers' with values 0..100
statement ok
CREATE TABLE integers AS SELECT * FROM range(0, 100, 1) t1(i);

# verify individually that all 100 values are there
loop i 0 100

# execute the query, replacing the value
query I
SELECT count(*) FROM integers WHERE i = ${i};
----
1

# end the loop (note that multiple statements can be part of a loop)
endloop
```

Similarly, `foreach` can be used to iterate over a set of values.

```
foreach partcode millennium century decade year quarter month day hour minute second millisecond
microsecond epoch

query III
SELECT i, date_part('${partcode}', i) AS p, date_part(['${partcode}'], i) AS st
FROM intervals
WHERE p <> st['${partcode}'];
----

endloop
```

`foreach` also has a number of preset combinations that should be used when required. In this manner, when new combinations are added to the preset, old tests will automatically pick up these new combinations.

Preset	Expansion
<compression>	none uncompressed rle bitpacking dictionary fsst chimp patas
<signed>	tinyint smallint integer bigint hugeint

Preset	Expansion
<unsigned>	utinyint usmallint uinteger ubigint uhugeint
<integral>	<signed> <unsigned>
<numeric>	<integral> float double
<alltypes>	<numeric> bool interval varchar json

> Use large loops sparingly. Executing hundreds of thousands of SQL statements will slow down tests unnecessarily. Do not use loops for inserting data.

Data Generation without Loops

Loops should be used sparingly. While it might be tempting to use loops for inserting data using insert statements, this will considerably slow down the test cases. Instead, it is better to generate data using the built-in `range` and `repeat` functions.

To create the table `integers` with the values `[0, 1, .., 98, 99]`, run:

```
CREATE TABLE integers AS SELECT * FROM range(0, 100, 1) t1(i);
```

To create the table `strings` with 100 times the value `hello`, run:

```
CREATE TABLE strings AS SELECT * FROM repeat('hello', 100) t1(s);
```

Using these two functions, together with clever use of cross products and other expressions, many different types of datasets can be efficiently generated. The `random()` function can also be used to generate random data.

An alternative option is to read data from an existing CSV or Parquet file. There are several large CSV files that can be loaded from the directory `test/sql/copy/csv/data/real` using a `COPY INTO` statement or the `read_csv_auto` function.

The TPC-H and TPC-DS extensions can also be used to generate synthetic data, using e.g. `CALL dbgen(sf = 1)` or `CALL dsdgen(sf = 1)`.

Multiple Connections

For tests whose purpose is to verify that the transactional management or versioning of data works correctly, it is generally necessary to use multiple connections. For example, if we want to verify that the creation of tables is correctly transactional, we might want to start a transaction and create a table in `con1`, then fire a query in `con2` that checks that the table is not accessible yet until committed.

We can use multiple connections in the sqllogictests using `connection labels`. The connection label can be optionally appended to any `statement` or `query`. All queries with the same connection label will be executed in the same connection. A test that would verify the above property would look as follows:

```
statement ok con1
BEGIN TRANSACTION

statement ok con1
CREATE TABLE integers (i INTEGER);

statement error con2
SELECT * FROM integers;
```

Concurrent Connections

Using connection modifiers on the statement and queries will result in testing of multiple connections, but all the queries will still be run *sequentially* on a single thread. If we want to run code from multiple connections *concurrently* over multiple threads, we can use the `concurrentloop` construct. The queries in `concurrentloop` will be run concurrently on separate threads at the same time.

```
concurrentloop i 0 10

statement ok
CREATE TEMP TABLE t2 AS (SELECT 1);

statement ok
INSERT INTO t2 VALUES (42);

statement ok
DELETE FROM t2

endloop
```

One caveat with `concurrentloop` is that results are often unpredictable – as multiple clients can hammer the database at the same time we might end up with (expected) transaction conflicts. `statement maybe` can be used to deal with these situations. `statement maybe` essentially accepts both a success, and a failure with a specific error message.

```
concurrentloop i 1 10

statement maybe
CREATE OR REPLACE TABLE t2 AS (SELECT -541240333865773480040026564265311535114 FROM t2 LIMIT 70%);
----
write-write conflict

endloop
```

Catch C/C++ Tests

While we prefer the sqllogic tests for testing most functionality, for certain tests only SQL is not sufficient. This typically happens when you want to test the C++ API. When using pure SQL is really not an option it might be necessary to make a C++ test using Catch.

Catch tests reside in the test directory as well. Here is an example of a catch test that tests the storage of the system:

```
#include "catch.hpp"
#include "test_helpers.hpp"

TEST_CASE("Test simple storage", "[storage]") {
    auto config = GetTestConfig();
    unique_ptr<QueryResult> result;
    auto storage_database = TestCreatePath("storage_test");

    // make sure the database does not exist
    DeleteDatabase(storage_database);
    {
        // create a database and insert values
        DuckDB db(storage_database, config.get());
        Connection con(db);
        REQUIRE_NO_FAIL(con.Query("CREATE TABLE test (a INTEGER, b INTEGER);"));
        REQUIRE_NO_FAIL(con.Query("INSERT INTO test VALUES (11, 22), (13, 22), (12, 21), (NULL, NULL)"));
        REQUIRE_NO_FAIL(con.Query("CREATE TABLE test2 (a INTEGER);"));
        REQUIRE_NO_FAIL(con.Query("INSERT INTO test2 VALUES (13), (12), (11)"));
    }
```

```
    // reload the database from disk a few times
    for (idx_t i = 0; i < 2; i++) {
        DuckDB db(storage_database, config.get());
        Connection con(db);
        result = con.Query("SELECT * FROM test ORDER BY a");
        REQUIRE(CHECK_COLUMN(result, 0, {Value(), 11, 12, 13}));
        REQUIRE(CHECK_COLUMN(result, 1, {Value(), 22, 21, 22}));
        result = con.Query("SELECT * FROM test2 ORDER BY a");
        REQUIRE(CHECK_COLUMN(result, 0, {11, 12, 13}));
    }
    DeleteDatabase(storage_database);
}
```

The test uses the `TEST_CASE` wrapper to create each test. The database is created and queried using the C++ API. Results are checked using either `REQUIRE_FAIL` / `REQUIRE_NO_FAIL` (corresponding to statement ok and statement error) or `REQUIRE(CHECK_COLUMN(...))` (corresponding to query with a result check). Every test that is created in this way needs to be added to the corresponding `CMakeLists.txt`.

Internals

Overview of DuckDB Internals

On this page is a brief description of the internals of the DuckDB engine.

Parser

The parser converts a query string into the following tokens:

- `SQLStatement`
- `QueryNode`
- `TableRef`
- `ParsedExpression`

The parser is not aware of the catalog or any other aspect of the database. It will not throw errors if tables do not exist, and will not resolve **any** types of columns yet. It only transforms a query string into a set of tokens as specified.

ParsedExpression

The ParsedExpression represents an expression within a SQL statement. This can be e.g., a reference to a column, an addition operator or a constant value. The type of the ParsedExpression indicates what it represents, e.g., a comparison is represented as a `ComparisonExpression`.

ParsedExpressions do **not** have types, except for nodes with explicit types such as CAST statements. The types for expressions are resolved in the Binder, not in the Parser.

TableRef

The TableRef represents any table source. This can be a reference to a base table, but it can also be a join, a table-producing function or a subquery.

QueryNode

The QueryNode represents either (1) a `SELECT` statement, or (2) a set operation (i.e., `UNION`, `INTERSECT` or `DIFFERENCE`).

SQL Statement

The SQLStatement represents a complete SQL statement. The type of the SQL Statement represents what kind of statement it is (e.g., `StatementType::SELECT` represents a `SELECT` statement). A single SQL string can be transformed into multiple SQL statements in case the original query string contains multiple queries.

Binder

The binder converts all nodes into their **bound** equivalents. In the binder phase:

- The tables and columns are resolved using the catalog
- Types are resolved
- Aggregate/window functions are extracted

The following conversions happen:

- SQLStatement → `BoundStatement`
- QueryNode → `BoundQueryNode`
- TableRef → `BoundTableRef`
- ParsedExpression → `Expression`

Logical Planner

The logical planner creates `LogicalOperator` nodes from the bound statements. In this phase, the actual logical query tree is created.

Optimizer

After the logical planner has created the logical query tree, the optimizers are run over that query tree to create an optimized query plan. The following query optimizers are run:

- **Expression Rewriter**: Simplifies expressions, performs constant folding
- **Filter Pushdown**: Pushes filters down into the query plan and duplicates filters over equivalency sets. Also prunes subtrees that are guaranteed to be empty (because of filters that statically evaluate to false).
- **Join Order Optimizer**: Reorders joins using dynamic programming. Specifically, the DPccp algorithm from the paper Dynamic Programming Strikes Back is used.
- **Common Sub Expressions**: Extracts common subexpressions from projection and filter nodes to prevent unnecessary duplicate execution.
- **In Clause Rewriter**: Rewrites large static IN clauses to a MARK join or INNER join.

Column Binding Resolver

The column binding resolver converts logical `BoundColumnRefExpresion` nodes that refer to a column of a specific table into `BoundReferenceExpression` nodes that refer to a specific index into the DataChunks that are passed around in the execution engine.

Physical Plan Generator

The physical plan generator converts the resulting logical operator tree into a `PhysicalOperator` tree.

Execution

In the execution phase, the physical operators are executed to produce the query result. DuckDB uses a push-based vectorized model, where `DataChunks` are pushed through the operator tree. For more information, see the talk Push-Based Execution in DuckDB.

Storage Versions and Format

Compatibility

Backward Compatibility

Backward compatibility refers to the ability of a newer DuckDB version to read storage files created by an older DuckDB version. Version 0.10 is the first release of DuckDB that supports backward compatibility in the storage format. DuckDB v0.10 can read and operate on files created by the previous DuckDB version – DuckDB v0.9.

For future DuckDB versions, our goal is to ensure that any DuckDB version released **after** can read files created by previous versions, starting from this release. We want to ensure that the file format is fully backward compatible. This allows you to keep data stored in DuckDB files around and guarantees that you will be able to read the files without having to worry about which version the file was written with or having to convert files between versions.

Forward Compatibility

Forward compatibility refers to the ability of an older DuckDB version to read storage files produced by a newer DuckDB version. DuckDB v0.9 is **partially** forward compatible with DuckDB v0.10. Certain files created by DuckDB v0.10 can be read by DuckDB v0.9.

Forward compatibility is provided on a **best effort** basis. While stability of the storage format is important – there are still many improvements and innovations that we want to make to the storage format in the future. As such, forward compatibility may be (partially) broken on occasion.

How to Move between Storage Formats

When you update DuckDB and open an old database file, you might encounter an error message about incompatible storage formats, pointing to this page. To move your database(s) to newer format you only need the older and the newer DuckDB executable.

Open your database file with the older DuckDB and run the SQL statement `EXPORT DATABASE 'tmp'`. This allows you to save the whole state of the current database in use inside folder `tmp`. The content of the `tmp` folder will be overridden, so choose an empty/non yet existing location. Then, start the newer DuckDB and execute `IMPORT DATABASE 'tmp'` (pointing to the previously populated folder) to load the database, which can be then saved to the file you pointed DuckDB to.

A Bash script to achieve this (to be adapted with the file names and executable locations) is the following

```
/older/duckdb mydata.old.db -c "EXPORT DATABASE 'tmp'"
/newer/duckdb mydata.new.db -c "IMPORT DATABASE 'tmp'"
```

After this, `mydata.old.db` will remain in the old format, `mydata.new.db` will contain the same data but in a format accessible by the more recent DuckDB version, and the folder `tmp` will hold the same data in a universal format as different files.

Check EXPORT documentation for more details on the syntax.

Explicit Storage Versions

DuckDB v1.2.0 introduced the `STORAGE_VERSION` option, which allows explicitly specifying the storage version. Using this, you can opt-in to newer forwards-incompatible features:

```
ATTACH 'file.db' (STORAGE_VERSION 'v1.2.0');
```

This setting specifies the minimum DuckDB version that should be able to read the database file. When database files are written with this option, the resulting files cannot be opened by older DuckDB released versions than the specified version. They can be read by the specified version and all newer versions of DuckDB.

If you attach to DuckDB databases, you can query the storage versions using the following command:

```
SELECT database_name, tags FROM duckdb_databases();
```

This shows the storage versions:

database_name varchar	tags map(varchar, varchar)
file1	{storage_version=v1.2.0}
file2	{storage_version=v1.0.0 - v1.1.3}
...	...

This means that `file2` can be opened by past DuckDB versions while `file1` is compatible only with `v1.2.0` (or future versions).

Converting between Storage Versions

To convert from the new format to the old format for compatibility, use the following sequence in DuckDB v1.2.0+:

```
ATTACH 'file1.db';
ATTACH 'converted_file.db' (STORAGE_VERSION 'v1.0.0');
COPY FROM DATABASE file1 TO converted_file;
```

Storage Header

DuckDB files start with a `uint64_t` which contains a checksum for the main header, followed by four magic bytes (DUCK), followed by the storage version number in a `uint64_t`.

```
hexdump -n 20 -C mydata.db
```

```
00000000  01 d0 e2 63 9c 13 39 3e  44 55 43 4b 2b 00 00 00  |...c..9>DUCK+...|
00000010  00 00 00 00                                       |....|
00000014
```

A simple example of reading the storage version using Python is below.

```
import struct

pattern = struct.Struct('<8x4sQ')

with open('test/sql/storage_version/storage_version.db', 'rb') as fh:
    print(pattern.unpack(fh.read(pattern.size)))
```

Storage Version Table

For changes in each given release, check out the change log on GitHub. To see the commits that changed each storage version, see the commit log.

Storage version	DuckDB version(s)
66	v1.3.x
65	v1.2.x
64	v0.9.x, v0.10.x, v1.0.0, v1.1.x
51	v0.8.x
43	v0.7.x
39	v0.6.x
38	v0.5.x
33	v0.3.3, v0.3.4, v0.4.0
31	v0.3.2
27	v0.3.1
25	v0.3.0
21	v0.2.9
18	v0.2.8
17	v0.2.7
15	v0.2.6
13	v0.2.5
11	v0.2.4
6	v0.2.3
4	v0.2.2
1	v0.2.1 and prior

Compression

DuckDB uses lightweight compression. Note that compression is only applied to persistent databases and is **not applied to in-memory instances**.

Compression Algorithms

The compression algorithms supported by DuckDB include the following:

- Constant Encoding
- Run-Length Encoding (RLE)
- Bit Packing
- Frame of Reference (FOR)
- Dictionary Encoding
- Fast Static Symbol Table (FSST) – VLDB 2020 paper
- Adaptive Lossless Floating-Point Compression (ALP) – SIGMOD 2024 paper
- Chimp – VLDB 2022 paper
- Patas

Disk Usage

The disk usage of DuckDB's format depends on a number of factors, including the data type and the data distribution, the compression methods used, etc. As a rough approximation, loading 100 GB of uncompressed CSV files into a DuckDB database file will require 25 GB of disk space, while loading 100 GB of Parquet files will require 120 GB of disk space.

Row Groups

DuckDB's storage format stores the data in *row groups,* i.e., horizontal partitions of the data. This concept is equivalent to Parquet's row groups. Several features in DuckDB, including parallelism and compression are based on row groups.

Troubleshooting

Error Message When Opening an Incompatible Database File

When opening a database file that has been written by a different DuckDB version from the one you are using, the following error message may occur:

```
Error: unable to open database "...": Serialization Error: Failed to deserialize: ...
```

The message implies that the database file was created with a newer DuckDB version and uses features that are backward incompatible with the DuckDB version used to read the file.

There are two potential workarounds:

1. Update your DuckDB version to the latest stable version.
2. Open the database with the latest version of DuckDB, export it to a standard format (e.g., Parquet), then import it using to any version of DuckDB. See the `EXPORT/IMPORT DATABASE` statements for details.

Execution Format

`Vector` is the container format used to store in-memory data during execution. `DataChunk` is a collection of Vectors, used for instance to represent a column list in a `PhysicalProjection` operator.

Data Flow

DuckDB uses a vectorized query execution model. All operators in DuckDB are optimized to work on Vectors of a fixed size.

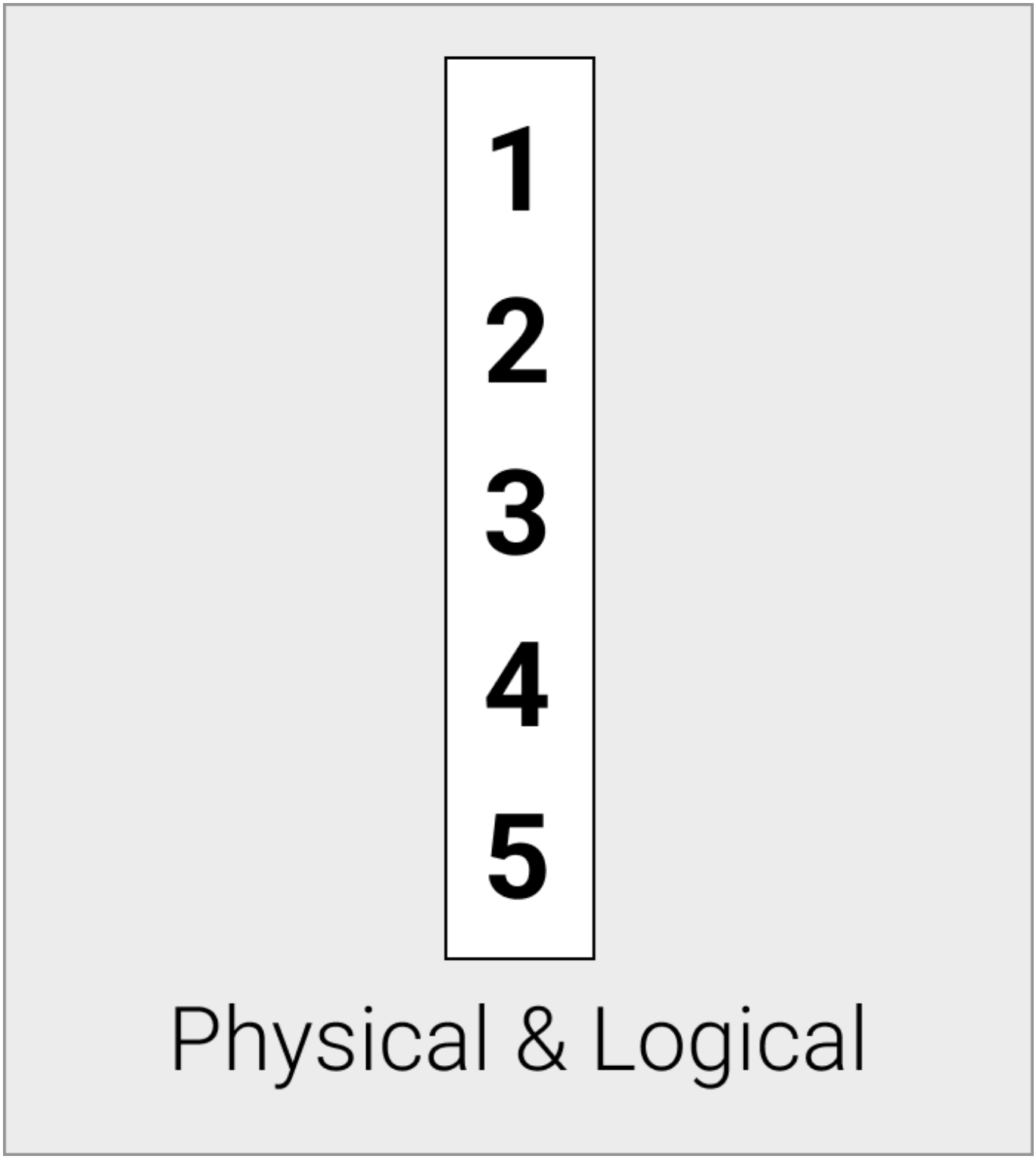
This fixed size is commonly referred to in the code as `STANDARD_VECTOR_SIZE`. The default `STANDARD_VECTOR_SIZE` is 2048 tuples.

Vector Format

Vectors logically represent arrays that contain data of a single type. DuckDB supports different *vector formats*, which allow the system to store the same logical data with a different *physical representation*. This allows for a more compressed representation, and potentially allows for compressed execution throughout the system. Below the list of supported vector formats is shown.

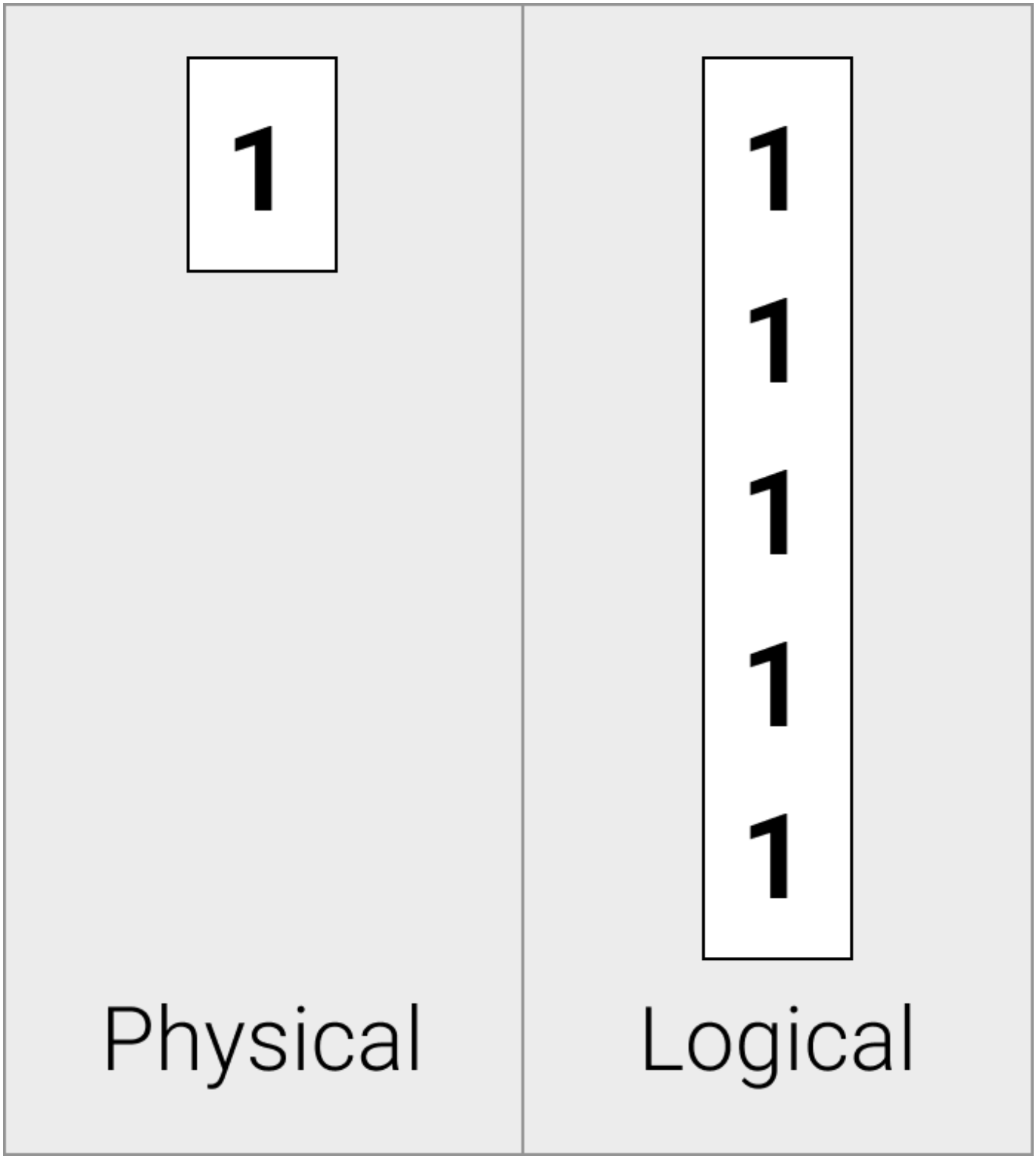
Flat Vectors

Flat vectors are physically stored as a contiguous array, this is the standard uncompressed vector format. For flat vectors the logical and physical representations are identical.



Constant Vectors

Constant vectors are physically stored as a single constant value.



Constant vectors are useful when data elements are repeated – for example, when representing the result of a constant expression in a function call, the constant vector allows us to only store the value once.

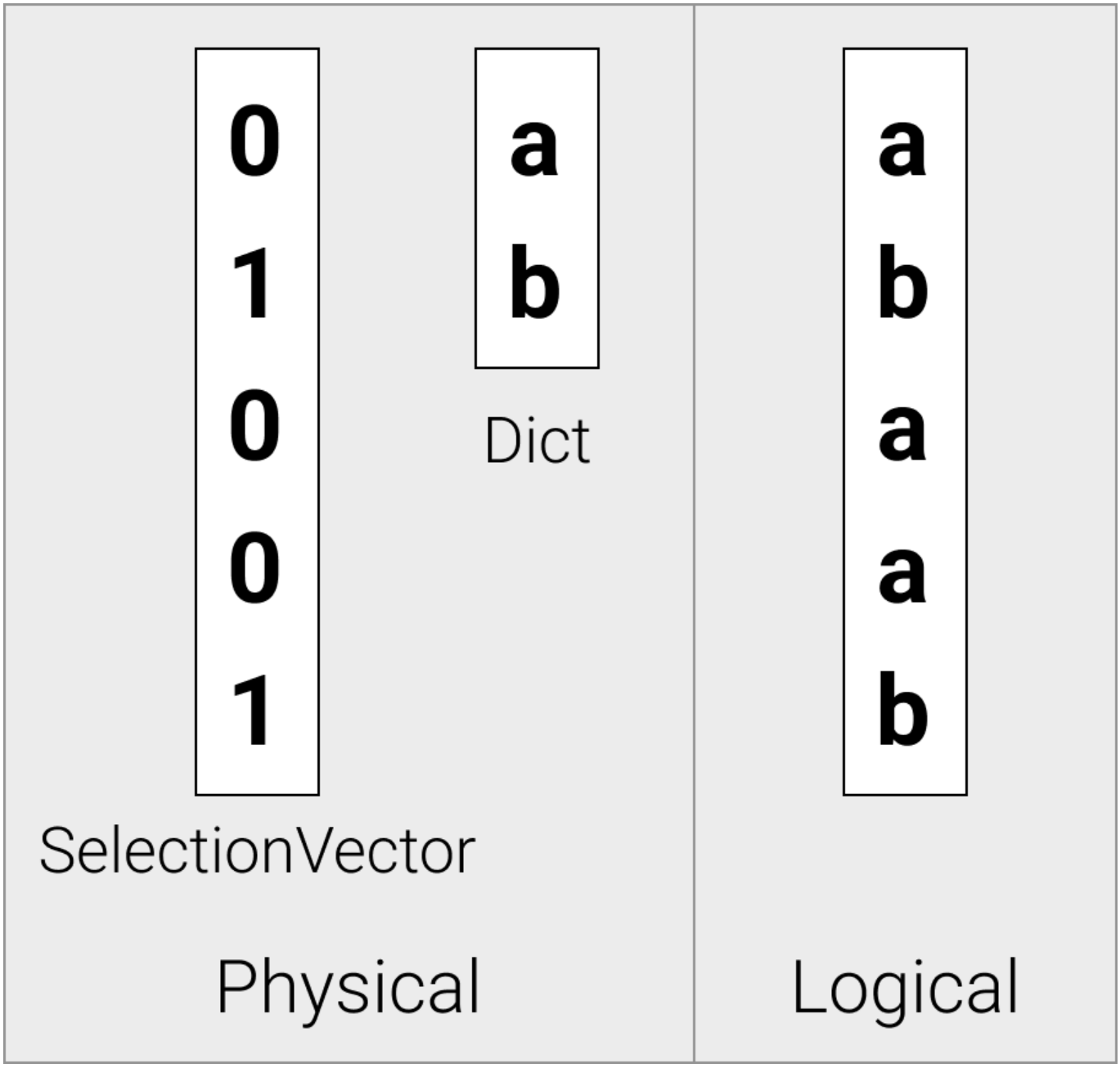
```
SELECT lst || 'duckdb'
FROM range(1000) tbl(lst);
```

Since `duckdb` is a string literal, the value of the literal is the same for every row. In a flat vector, we would have to duplicate the literal 'duckdb' once for every row. The constant vector allows us to only store the literal once.

Constant vectors are also emitted by the storage when decompressing from constant compression.

Dictionary Vectors

Dictionary vectors are physically stored as a child vector, and a selection vector that contains indexes into the child vector.

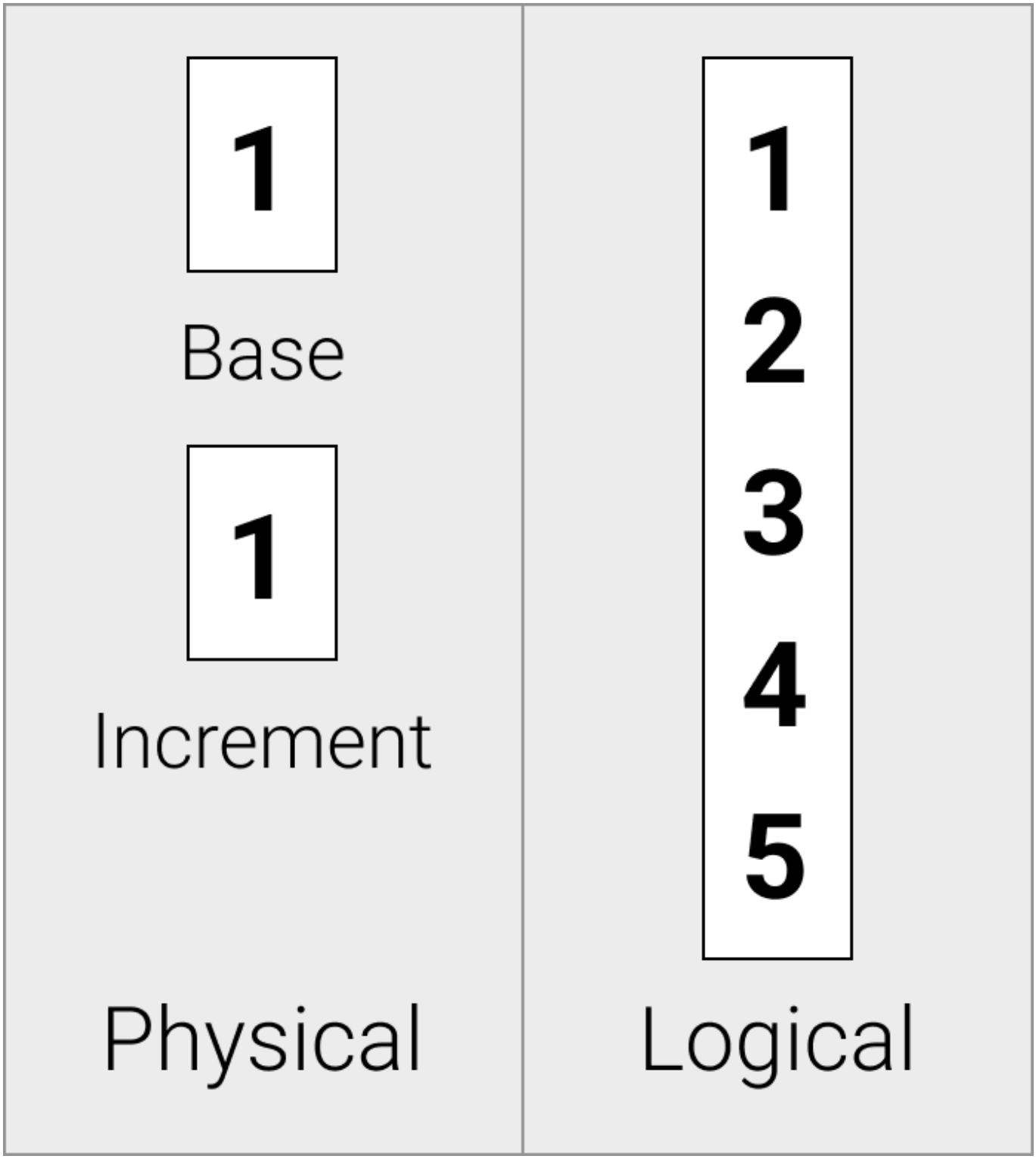


Dictionary vectors are emitted by the storage when decompressing from dictionary

Just like constant vectors, dictionary vectors are also emitted by the storage. When deserializing a dictionary compressed column segment, we store this in a dictionary vector so we can keep the data compressed during query execution.

Sequence Vectors

Sequence vectors are physically stored as an offset and an increment value.



Sequence vectors are useful for efficiently storing incremental sequences. They are generally emitted for row identifiers.

Unified Vector Format

These properties of the different vector formats are great for optimization purposes, for example you can imagine the scenario where all the parameters to a function are constant, we can just compute the result once and emit a constant vector. But writing specialized code for every combination of vector types for every function is unfeasible due to the combinatorial explosion of possibilities.

Instead of doing this, whenever you want to generically use a vector regardless of the type, the UnifiedVectorFormat can be used. This format essentially acts as a generic view over the contents of the Vector. Every type of Vector can convert to this format.

Complex Types

String Vectors

To efficiently store strings, we make use of our `string_t` class.

```
struct string_t {
    union {
        struct {
            uint32_t length;
            char prefix[4];
            char *ptr;
        } pointer;
        struct {
            uint32_t length;
            char inlined[12];
        } inlined;
    } value;
};
```

Short strings (`<= 12 bytes`) are inlined into the structure, while larger strings are stored with a pointer to the data in the auxiliary string buffer. The length is used throughout the functions to avoid having to call `strlen` and having to continuously check for null-pointers. The prefix is used for comparisons as an early out (when the prefix does not match, we know the strings are not equal and don't need to chase any pointers).

List Vectors

List vectors are stored as a series of *list entries* together with a child Vector. The child vector contains the *values* that are present in the list, and the list entries specify how each individual list is constructed.

```
struct list_entry_t {
    idx_t offset;
    idx_t length;
};
```

The offset refers to the start row in the child Vector, the length keeps track of the size of the list of this row.

List vectors can be stored recursively. For nested list vectors, the child of a list vector is again a list vector.

For example, consider this mock representation of a Vector of type `BIGINT[][]`:

```
{
   "type": "list",
   "data": "list_entry_t",
   "child": {
      "type": "list",
      "data": "list_entry_t",
      "child": {
         "type": "bigint",
         "data": "int64_t"
      }
   }
}
```

Struct Vectors

Struct vectors store a list of child vectors. The number and types of the child vectors is defined by the schema of the struct.

Map Vectors

Internally map vectors are stored as a `LIST[STRUCT(key KEY_TYPE, value VALUE_TYPE)]`.

Union Vectors

Internally `UNION` utilizes the same structure as a `STRUCT`. The first "child" is always occupied by the Tag Vector of the `UNION`, which records for each row which of the `UNION`'s types apply to that row.

Pivot Internals

PIVOT

Pivoting is implemented as a combination of SQL query re-writing and a dedicated `PhysicalPivot` operator for higher performance. Each `PIVOT` is implemented as set of aggregations into lists and then the dedicated `PhysicalPivot` operator converts those lists into column names and values. Additional pre-processing steps are required if the columns to be created when pivoting are detected dynamically (which occurs when the `IN` clause is not in use).

DuckDB, like most SQL engines, requires that all column names and types be known at the start of a query. In order to automatically detect the columns that should be created as a result of a `PIVOT` statement, it must be translated into multiple queries. ENUM types are used to find the distinct values that should become columns. Each `ENUM` is then injected into one of the `PIVOT` statement's `IN` clauses.

After the `IN` clauses have been populated with `ENUM`s, the query is re-written again into a set of aggregations into lists.

For example:

```
PIVOT cities
ON year
USING sum(population);
```

is initially translated into:

```
CREATE TEMPORARY TYPE __pivot_enum_0_0 AS ENUM (
    SELECT DISTINCT
        year::VARCHAR
    FROM cities
    ORDER BY
        year
    );
PIVOT cities
ON year IN __pivot_enum_0_0
USING sum(population);
```

and finally translated into:

```
SELECT country, name, list(year), list(population_sum)
FROM (
    SELECT country, name, year, sum(population) AS population_sum
    FROM cities
    GROUP BY ALL
)
GROUP BY ALL;
```

This produces the result:

country	name	list(”year”)	list(population_sum)
NL	Amsterdam	[2000, 2010, 2020]	[1005, 1065, 1158]
US	Seattle	[2000, 2010, 2020]	[564, 608, 738]
US	New York City	[2000, 2010, 2020]	[8015, 8175, 8772]

The `PhysicalPivot` operator converts those lists into column names and values to return this result:

country	name	2000	2010	2020
NL	Amsterdam	1005	1065	1158
US	Seattle	564	608	738
US	New York City	8015	8175	8772

UNPIVOT

Internals

Unpivoting is implemented entirely as rewrites into SQL queries. Each `UNPIVOT` is implemented as set of `unnest` functions, operating on a list of the column names and a list of the column values. If dynamically unpivoting, the `COLUMNS` expression is evaluated first to calculate the column list.

For example:

```
UNPIVOT monthly_sales
ON jan, feb, mar, apr, may, jun
INTO
    NAME month
    VALUE sales;
```

is translated into:

```
SELECT
    empid,
    dept,
    unnest(['jan', 'feb', 'mar', 'apr', 'may', 'jun']) AS month,
    unnest(["jan", "feb", "mar", "apr", "may", "jun"]) AS sales
FROM monthly_sales;
```

Note the single quotes to build a list of text strings to populate `month`, and the double quotes to pull the column values for use in `sales`. This produces the same result as the initial example:

empid	dept	month	sales
1	electronics	jan	1
1	electronics	feb	2
1	electronics	mar	3
1	electronics	apr	4
1	electronics	may	5
1	electronics	jun	6
2	clothes	jan	10
2	clothes	feb	20
2	clothes	mar	30
2	clothes	apr	40
2	clothes	may	50
2	clothes	jun	60

empid	dept	month	sales
3	cars	jan	100
3	cars	feb	200
3	cars	mar	300
3	cars	apr	400
3	cars	may	500
3	cars	jun	600

DuckDB Blog

Testing Out DuckDB's Full Text Search Extension

Publication date: 2021-01-25

Author: Laurens Kuiper

TL;DR: DuckDB now has full-text search functionality, similar to the FTS5 extension in SQLite. The main difference is that our FTS extension is fully formulated in SQL. We tested it out on TREC disks 4 and 5.

Searching through textual data stored in a database can be cumbersome, as SQL does not provide a good way of formulating questions such as "Give me all the documents about **Mallard Ducks**": string patterns with `LIKE` will only get you so far. Despite SQL's shortcomings here, storing textual data in a database is commonplace. Consider the table `products (id INTEGER, name VARCHAR, description VARCHAR)` – it would be useful to search through the `name` and `description` columns for a website that sells these products.

We expect a search engine to return us results within milliseconds. For a long time databases were unsuitable for this task, because they could not search large inverted indexes at this speed: transactional database systems are not made for this use case. However, analytical database systems, can keep up with state-of-the art information retrieval systems. The company Spinque is a good example of this. At Spinque, MonetDB is used as a computation engine for customized search engines.

DuckDB's FTS implementation follows the paper "Old Dogs Are Great at New Tricks". A keen observation there is that advances made to the database system, such as parallelization, will speed up your search engine "for free"!

Alright, enough about the "why", let's get to the "how".

Preparing the Data

The TREC 2004 Robust Retrieval Track has 250 "topics" (search queries) over TREC disks 4 and 5. The data consist of many text files stored in SGML format, along with a corresponding DTD (document type definition) file. This format is rarely used anymore, but it is similar to XML. We will use OpenSP's command line tool `osx` to convert it to XML. Because there are many files, I wrote a Bash script:

```
mkdir -p latimes/xml
for i in $(seq -w 1 9); do
    cat dtds/la.dtd latimes-$i | osx > latimes/xml/latimes-$i.xml
done
```

This sorts the `latimes` files. Repeat for the `fbis`, `cr`, `fr94`, and `ft` files.

To parse the XML I used BeautifulSoup. Each document has a `docno` identifier, and a `text` field. Because the documents do not come from the same source, they differ in what other fields they have. I chose to take all of the fields.

```
import duckdb
import multiprocessing
import pandas as pd
import re
from bs4 import BeautifulSoup as bs
from tqdm import tqdm

# fill variable 'files' with the path to each .xml file that we created here

def process_file(fpath):
    dict_list = []
    with open(fpath, 'r') as f:
        content = f.read()
```

```
        bs_content = bs(content, "html.parser")
        # find all 'doc' nodes
        for doc in bs_content.findChildren('doc', recursive=True):
            row_dict = {}
            for c in doc.findChildren(recursive=True):
                row_dict[c.name] = ''.join(c.findAll(text=True, recursive=False)).trim()
            dict_list.append(row_dict)
    return dict_list

# process documents (in parallel to speed things up)
pool = multiprocessing.Pool(multiprocessing.cpu_count())
list_of_dict_lists = []
for x in tqdm(pool.imap_unordered(process_file, files), total=len(files)):
    list_of_dict_lists.append(x)
pool.close()
pool.join()

# create pandas dataframe from the parsed data
documents_df = pd.DataFrame([x for sublist in list_of_dict_lists for x in sublist])
```

Now that we have a dataframe, we can register it in DuckDB.

```
# create database connection and register the dataframe
con = duckdb.connect(database='db/trec04_05.db', read_only=False)
con.register('documents_df', documents_df)

# create a table from the dataframe so that it persists
con.execute("CREATE TABLE documents AS (SELECT * FROM documents_df)")
con.close()
```

This is the end of my preparation script, so I closed the database connection.

Building the Search Engine

We can now build the inverted index and the retrieval model using a PRAGMA statement. The extension is documented here. We create an index table on table `documents` or `main.documents` that we created with our script. The column that identifies our documents is called `docno`, and we wish to create an inverted index on the fields supplied. I supplied all fields by using the '*' shortcut.

```
con = duckdb.connect(database='db/trec04_05.db', read_only=False)
con.execute("PRAGMA create_fts_index('documents', 'docno', '*', stopwords='english')")
```

Under the hood, a parameterized SQL script is called. The schema `fts_main_documents` is created, along with tables `docs`, `terms`, `dict`, and `stats`, that make up the inverted index. If you're curious what this look like, take a look at our source code under the `extension` folder in DuckDB's source code!

Running the Benchmark

The data is now fully prepared. Now we want to run the queries in the benchmark, one by one. We load the topics file as follows:

```
# the 'topics' file is not structured nicely, therefore we need parse some of it using regex
def after_tag(s, tag):
    m = re.findall(r'<' + tag + r'>([\s\S]*?)<.*>', s)
    return m[0].replace('\n', '').strip()

topic_dict = {}
with open('../../trec/topics', 'r') as f:
    bs_content = bs(f.read(), "lxml")
```

```
    for top in bs_content.findChildren('top'):
        top_content = top.getText()
        # we need the number and title of each topic
        num = after_tag(str(top), 'num').split(' ')[1]
        title = after_tag(str(top), 'title')
        topic_dict[num] = title
```

This gives us a dictionary that has query number as keys, and query strings as values, e.g., `301 -> 'International Organized Crime'`.

We want to store the results in a specific format, so that they can be evaluated by trec eval:

```
# create a prepared statement to make querying our document collection easier
con.execute("""
    PREPARE fts_query AS (
        WITH scored_docs AS (
            SELECT *, fts_main_documents.match_bm25(docno, ?) AS score FROM documents)
        SELECT docno, score
        FROM scored_docs
        WHERE score IS NOT NULL
        ORDER BY score DESC
        LIMIT 1000)
    """)

# enable parallelism
con.execute('PRAGMA threads=32')
results = []
for query in topic_dict:
    q_str = topic_dict[query].replace('\'', ' ')
    con.execute("EXECUTE fts_query('" + q_str + "')")
    for i, row in enumerate(con.fetchall()):
        results.append(query + " Q0 " + row[0].trim() + " " + str(i) + " " + str(row[1]) + " STANDARD")
con.close()

with open('results', 'w+') as f:
    for r in results:
        f.write(r + '\n')
```

Results

Now that we have created our 'results' file, we can compare them to the relevance assessments `qrels` using `trec_eval`.

```
./trec_eval -m P.30 -m map qrels results
```

```
map                     all 0.2324
P_30                    all 0.2948
```

Not bad! While these results are not as high as the reproducible by Anserini, they are definitely acceptable. The difference in performance can be explained by differences in

1. Which stemmer was used (we used 'porter')
2. Which stopwords were used (we used the list of 571 English stopwords used in the SMART system)
3. Pre-processing (removal of accents, punctuation, numbers)
4. BM25 parameters (we used the default k=1.2 and b=0.75, non-conjunctive)
5. Which fields were indexed (we used all columns by supplying '*')

Retrieval time for each query was between 0.5 and 1.3 seconds on our machine, which will be improved with further improvements to DuckDB. I hope you enjoyed reading this blog, and become inspired to test out the extension as well!

Efficient SQL on Pandas with DuckDB

Publication date: 2021-05-14

Authors: Mark Raasveldt and Hannes Mühleisen

TL;DR: DuckDB, a free and open source analytical data management system, can efficiently run SQL queries directly on Pandas DataFrames.

Recently, an article was published advocating for using SQL for Data Analysis. Here at team DuckDB, we are huge fans of SQL. It is a versatile and flexible language that allows the user to efficiently perform a wide variety of data transformations, without having to care about how the data is physically represented or how to do these data transformations in the most optimal way.

While you can very effectively perform aggregations and data transformations in an external database system such as Postgres if your data is stored there, at some point you will need to convert that data back into Pandas and NumPy. These libraries serve as the standard for data exchange between the vast ecosystem of Data Science libraries in Python1 such as scikit-learn or TensorFlow.

1Apache Arrow is gaining significant traction in this domain as well, and DuckDB also quacks Arrow.

If you are reading from a file (e.g., a CSV or Parquet file) often your data will never be loaded into an external database system at all, and will instead be directly loaded into a Pandas DataFrame.

SQL on Pandas

After your data has been converted into a Pandas DataFrame often additional data wrangling and analysis still need to be performed. SQL is a very powerful tool for performing these types of data transformations. Using DuckDB, it is possible to run SQL efficiently right on top of Pandas DataFrames.

As a short teaser, here is a code snippet that allows you to do exactly that: run arbitrary SQL queries directly on Pandas DataFrames using DuckDB.

```
# to install: pip install duckdb
import pandas as pd
import duckdb

mydf = pd.DataFrame({'a' : [1, 2, 3]})
print(duckdb.query("SELECT sum(a) FROM mydf").to_df())
```

In the rest of the article, we will go more in-depth into how this works and how fast it is.

Data Integration & SQL on Pandas

One of the core goals of DuckDB is that accessing data in common formats should be easy. DuckDB is fully capable of running queries in parallel *directly* on top of a Pandas DataFrame (or on a Parquet/CSV file, or on an Arrow table, …). A separate (time-consuming) import step is not necessary.

DuckDB can also write query results directly to any of these formats. You can use DuckDB to process a Pandas DataFrame in parallel using SQL, and convert the result back to a Pandas DataFrame again, so you can then use the result in other Data Science libraries.

When you run a query in SQL, DuckDB will look for Python variables whose name matches the table names in your query and automatically start reading your Pandas DataFrames. Looking back at the previous example we can see this in action:

```
import pandas as pd
import duckdb

mydf = pd.DataFrame({'a' : [1, 2, 3]})
print(duckdb.query("SELECT sum(a) FROM mydf").to_df())
```

The SQL table name `mydf` is interpreted as the local Python variable `mydf` that happens to be a Pandas DataFrame, which DuckDB can read and query directly. The column names and types are also extracted automatically from the DataFrame.

Not only is this process painless, it is highly efficient. For many queries, you can use DuckDB to process data faster than Pandas, and with a much lower total memory usage, *without ever leaving the Pandas DataFrame binary format* ("Pandas-in, Pandas-out"). Unlike when using an external database system such as Postgres, the data transfer time of the input or the output is negligible (see Appendix A for details).

SQL on Pandas Performance

To demonstrate the performance of DuckDB when executing SQL on Pandas DataFrames, we now present a number of benchmarks. The source code for the benchmarks is available for interactive use in Google Colab. In these benchmarks, we operate *purely* on Pandas DataFrames. Both the DuckDB code and the Pandas code operates fully on a `Pandas-in, Pandas-out` basis.

Benchmark Setup and Data Set

We run the benchmark entirely from within the Google Colab environment. For our benchmark dataset, we use the infamous TPC-H data set. Specifically, we focus on the `lineitem` and `orders` tables as these are the largest tables in the benchmark. The total dataset size is around 1 GB in uncompressed CSV format ("scale factor" 1).

As DuckDB is capable of using multiple processors (multi-threading), we include both a single-threaded variant and a variant with two threads. Note that while DuckDB can scale far beyond two threads, Google Colab only supports two.

Setup

First we need to install DuckDB. This is a simple one-liner.

```
pip install duckdb
```

To set up the dataset for processing we download two parquet files using `wget`. After that, we load the data into a Pandas DataFrame using the built-in Parquet reader of DuckDB. The system automatically infers that we are reading a parquet file by looking at the `.parquet` extension of the file.

```
lineitem = duckdb.query(
    "SELECT * FROM 'lineitemsf1.snappy.parquet'"
).to_df()
orders = duckdb.query(
    "SELECT * FROM 'orders.parquet'"
).to_df()
```

Ungrouped Aggregates

For our first query, we will run a set of ungrouped aggregates over the Pandas DataFrame. Here is the SQL query:

```
SELECT
    sum(l_extendedprice),
    min(l_extendedprice),
    max(l_extendedprice),
    avg(l_extendedprice)
FROM lineitem;
```

The Pandas code looks similar:

```
lineitem.agg(
  Sum=('l_extendedprice', 'sum'),
  Min=('l_extendedprice', 'min'),
  Max=('l_extendedprice', 'max'),
  Avg=('l_extendedprice', 'mean')
)
```

Name	Time (s)
DuckDB (1 Thread)	0.079
DuckDB (2 Threads)	0.048
Pandas	0.070

This benchmark involves a very simple query, and Pandas performs very well here. These simple queries are where Pandas excels (ha), as it can directly call into the numpy routines that implement these aggregates, which are highly efficient. Nevertheless, we can see that DuckDB performs similar to Pandas in the single-threaded scenario, and benefits from its multi-threading support when enabled.

Grouped Aggregate

For our second query, we will run the same set of aggregates, but this time include a grouping condition. In SQL, we can do this by adding a GROUP BY clause to the query.

```
SELECT
    l_returnflag,
    l_linestatus,
    sum(l_extendedprice),
    min(l_extendedprice),
    max(l_extendedprice),
    avg(l_extendedprice)
FROM lineitem
GROUP BY
    l_returnflag,
    l_linestatus;
```

In Pandas, we use the groupby function before we perform the aggregation.

```
lineitem.groupby(
  ['l_returnflag', 'l_linestatus']
).agg(
  Sum=('l_extendedprice', 'sum'),
  Min=('l_extendedprice', 'min'),
  Max=('l_extendedprice', 'max'),
  Avg=('l_extendedprice', 'mean')
)
```

Name	Time (s)
DuckDB (1 Thread)	0.43
DuckDB (2 Threads)	0.32
Pandas	0.84

This query is already getting more complex, and while Pandas does a decent job, it is a factor two slower than the single-threaded version of DuckDB. DuckDB has a highly optimized aggregate hash-table implementation that will perform both the grouping and the computation of all the aggregates in a single pass over the data.

Grouped Aggregate with a Filter

Now suppose that we don't want to perform an aggregate over all of the data, but instead only want to select a subset of the data to aggregate. We can do this by adding a filter clause that removes any tuples we are not interested in. In SQL, we can accomplish this through the `WHERE` clause.

```
SELECT
    l_returnflag,
    l_linestatus,
    sum(l_extendedprice),
    min(l_extendedprice),
    max(l_extendedprice),
    avg(l_extendedprice)
FROM lineitem
WHERE
    l_shipdate <= DATE '1998-09-02'
GROUP BY
    l_returnflag,
    l_linestatus;
```

In Pandas, we can create a filtered variant of the DataFrame by using the selection brackets.

```
# filter out the rows
filtered_df = lineitem[
  lineitem['l_shipdate'] < "1998-09-02"]
# perform the aggregate
result = filtered_df.groupby(
  ['l_returnflag', 'l_linestatus']
).agg(
  Sum=('l_extendedprice', 'sum'),
  Min=('l_extendedprice', 'min'),
  Max=('l_extendedprice', 'max'),
  Avg=('l_extendedprice', 'mean')
)
```

In DuckDB, the query optimizer will combine the filter and aggregation into a single pass over the data, only reading relevant columns. In Pandas, however, we have no such luck. The filter as it is executed will actually subset the entire lineitem table, *including any columns we are not using!* As a result of this, the filter operation is much more time-consuming than it needs to be.

We can manually perform this optimization ("projection pushdown" in database literature). To do this, we first need to select only the columns that are relevant to our query and then subset the lineitem dataframe. We will end up with the following code snippet:

```
# projection pushdown
pushed_down_df = lineitem[
  ['l_shipdate',
   'l_returnflag',
   'l_linestatus',
   'l_extendedprice']
]
# perform the filter
filtered_df = pushed_down_df[
  pushed_down_df['l_shipdate'] < "1998-09-02"]
# perform the aggregate
result = filtered_df.groupby(
  ['l_returnflag', 'l_linestatus']
```

```
).agg(
  Sum=('l_extendedprice', 'sum'),
  Min=('l_extendedprice', 'min'),
  Max=('l_extendedprice', 'max'),
  Avg=('l_extendedprice', 'mean')
)
```

Name	Time (s)
DuckDB (1 Thread)	0.60
DuckDB (2 Threads)	0.42
Pandas	3.57
Pandas (manual pushdown)	2.23

While the manual projection pushdown significantly speeds up the query in Pandas, there is still a significant time penalty for the filtered aggregate. To process a filter, Pandas will write a copy of the entire DataFrame (minus the filtered out rows) back into memory. This operation can be time consuming when the filter is not very selective.

Due to its holistic query optimizer and efficient query processor, DuckDB performs significantly better on this query.

Joins

For the final query, we will join (`merge` in Pandas) the lineitem table with the orders table, and apply a filter that only selects orders which have the status we are interested in. This leads us to the following query in SQL:

```
SELECT
    l_returnflag,
    l_linestatus,
    sum(l_extendedprice),
    min(l_extendedprice),
    max(l_extendedprice),
    avg(l_extendedprice)
FROM lineitem
JOIN orders ON (l_orderkey = o_orderkey)
WHERE l_shipdate <= DATE '1998-09-02'
  AND o_orderstatus='O'
GROUP BY
    l_returnflag,
    l_linestatus;
```

For Pandas, we have to add a `merge` step. In a basic approach, we merge lineitem and orders together, then apply the filters, and finally apply the grouping and aggregation. This will give us the following code snippet:

```
# perform the join
merged = lineitem.merge(
  orders,
  left_on='l_orderkey',
  right_on='o_orderkey')
# filter out the rows
filtered_a = merged[
  merged['l_shipdate'] < "1998-09-02"]
filtered_b = filtered_a[
  filtered_a['o_orderstatus'] == "O"]
# perform the aggregate
result = filtered_b.groupby(
  ['l_returnflag', 'l_linestatus']
```

```
).agg(
  Sum=('l_extendedprice', 'sum'),
  Min=('l_extendedprice', 'min'),
  Max=('l_extendedprice', 'max'),
  Avg=('l_extendedprice', 'mean')
)
```

Now we have missed two performance opportunities:

- First, we are merging far too many columns, because we are merging columns that are not required for the remainder of the query (projection pushdown).
- Second, we are merging far too many rows. We can apply the filters prior to the merge to reduce the amount of data that we need to merge (filter pushdown).

Applying these two optimizations manually results in the following code snippet:

```
# projection & filter on lineitem table
lineitem_projected = lineitem[
  ['l_shipdate',
   'l_orderkey',
   'l_linestatus',
   'l_returnflag',
   'l_extendedprice']
]
lineitem_filtered = lineitem_projected[
  lineitem_projected['l_shipdate'] < "1998-09-02"]
# projection and filter on order table
orders_projected = orders[
  ['o_orderkey',
   'o_orderstatus']
]
orders_filtered = orders_projected[
  orders_projected['o_orderstatus'] == 'O']
# perform the join
merged = lineitem_filtered.merge(
  orders_filtered,
  left_on='l_orderkey',
  right_on='o_orderkey')
# perform the aggregate
result = merged.groupby(
  ['l_returnflag', 'l_linestatus']
).agg(
  Sum=('l_extendedprice', 'sum'),
  Min=('l_extendedprice', 'min'),
  Max=('l_extendedprice', 'max'),
  Avg=('l_extendedprice', 'mean')
)
```

Both of these optimizations are automatically applied by DuckDB's query optimizer.

Name	Time (s)
DuckDB (1 Thread)	1.05
DuckDB (2 Threads)	0.53
Pandas	15.2
Pandas (manual pushdown)	3.78

We see that the basic approach is extremely time consuming compared to the optimized version. This demonstrates the usefulness of the

automatic query optimizer. Even after optimizing, the Pandas code is still significantly slower than DuckDB because it stores intermediate results in memory after the individual filters and joins.

Takeaway

Using DuckDB, you can take advantage of the powerful and expressive SQL language without having to worry about moving your data in – and out – of Pandas. DuckDB is extremely simple to install, and offers many advantages such as a query optimizer, automatic multi-threading and larger-than-memory computation. DuckDB uses the Postgres SQL parser, and offers many of the same SQL features as Postgres, including advanced features such as window functions, correlated subqueries, (recursive) common table expressions, nested types and sampling. If you are missing a feature, please open an issue.

Appendix A: There and Back Again: Transferring Data from Pandas to a SQL Engine and Back

Traditional SQL engines use the Client-Server paradigm, which means that a client program connects through a socket to a server. Queries are run on the server, and results are sent back down to the client afterwards. This is the same when using for example Postgres from Python. Unfortunately, this transfer is a serious bottleneck. In-process engines such as SQLite or DuckDB do not run into this problem.

To showcase how costly this data transfer over a socket is, we have run a benchmark involving Postgres, SQLite and DuckDB. The source code for the benchmark can be found on GitHub.

In this benchmark we copy a (fairly small) Pandas data frame consisting of 10M 4-byte integers (40 MB) from Python to the PostgreSQL, SQLite and DuckDB databases. Since the default Pandas `to_sql` was rather slow, we added a separate optimization in which we tell Pandas to write the data frame to a temporary CSV file, and then tell PostgreSQL to directly copy data from that file into a newly created table. This of course will only work if the database server is running on the same machine as Python.

Name	Time (s)
Pandas to Postgres using to_sql	111.25
Pandas to Postgres using temporary CSV file	5.57
Pandas to SQLite using to_sql	6.80
Pandas to DuckDB	0.03

While SQLite performs significantly better than Postgres here, it is still rather slow. That is because the `to_sql` function in Pandas runs a large number of `INSERT INTO` statements, which involves transforming all the individual values of the Pandas DataFrame into a row-wise representation of Python objects which are then passed onto the system. DuckDB on the other hand directly reads the underlying array from Pandas, which makes this operation almost instant.

Transferring query results or tables back from the SQL system into Pandas is another potential bottleneck. Using the built-in `read_sql_query` is extremely slow, but even the more optimized CSV route still takes at least a second for this tiny data set. DuckDB, on the other hand, also performs this transformation almost instantaneously.

Name	Time (s)
PostgreSQL to Pandas using read_sql_query	7.08
PostgreSQL to Pandas using temporary CSV file	1.29
SQLite to Pandas using read_sql_query	5.20
DuckDB to Pandas	0.04

Appendix B: Comparison to PandaSQL

There is a package called PandaSQL that also provides the facilities of running SQL directly on top of Pandas. However, it is built using the to_sql and from_sql infrastructure that we have seen is extremely slow in Appendix A.

Nevertheless, for good measure we have run the first Ungrouped Aggregate query in PandaSQL to time it. When we first tried to run the query on the original dataset, however, we ran into an out-of-memory error that crashed our colab session. For that reason, we have decided to run the benchmark again for PandaSQL using a sample of 10% of the original data set size (600K rows). Here are the results:

Name	Time (s)
DuckDB (1 Thread)	0.023
DuckDB (2 Threads)	0.014
Pandas	0.017
PandaSQL	24.43

We can see that PandaSQL (powered by SQLite) is around 1000X~ slower than either Pandas or DuckDB on this straightforward benchmark. The performance difference was so large we have opted not to run the other benchmarks for PandaSQL.

Appendix C: Query on Parquet Directly

In the benchmarks above, we fully read the parquet files into Pandas. However, DuckDB also has the capability of directly running queries on top of Parquet files (in parallel!). In this appendix, we show the performance of this compared to loading the file into Python first.

For the benchmark, we will run two queries: the simplest query (the ungrouped aggregate) and the most complex query (the final join) and compare the cost of running this query directly on the Parquet file, compared to loading it into Pandas using the `read_parquet` function.

Setup

In DuckDB, we can create a view over the Parquet file using the following query. This allows us to run queries over the Parquet file as if it was a regular table. Note that we do not need to worry about projection pushdown at all: we can just do a `SELECT *` and DuckDB's optimizer will take care of only projecting the required columns at query time.

```
CREATE VIEW lineitem_parquet AS
    SELECT * FROM 'lineitemsf1.snappy.parquet';
CREATE VIEW orders_parquet AS
    SELECT * FROM 'orders.parquet';
```

Ungrouped Aggregate

After we have set up this view, we can run the same queries we ran before, but this time against the `lineitem_parquet` table.

```
SELECT sum(l_extendedprice), min(l_extendedprice), max(l_extendedprice), avg(l_extendedprice) FROM
lineitem_parquet;
```

For Pandas, we will first need to run `read_parquet` to load the data into Pandas. To do this, we use the Parquet reader powered by Apache Arrow. After that, we can run the query as we did before.

```
lineitem_pandas_parquet = pd.read_parquet('lineitemsf1.snappy.parquet')
result = lineitem_pandas_parquet.agg(Sum=('l_extendedprice', 'sum'), Min=('l_extendedprice', 'min'),
Max=('l_extendedprice', 'max'), Avg=('l_extendedprice', 'mean'))
```

However, we now again run into the problem where Pandas will read the Parquet file in its entirety. In order to circumvent this, we will need to perform projection pushdown manually again by providing the `read_parquet` method with the set of columns that we want to read.

The optimizer in DuckDB will figure this out by itself by looking at the query you are executing.

```
lineitem_pandas_parquet = pd.read_parquet('lineitemsf1.snappy.parquet', columns=['l_extendedprice'])
result = lineitem_pandas_parquet.agg(Sum=('l_extendedprice', 'sum'), Min=('l_extendedprice', 'min'),
Max=('l_extendedprice', 'max'), Avg=('l_extendedprice', 'mean'))
```

Name	Time (s)
DuckDB (1 Thread)	0.16
DuckDB (2 Threads)	0.14
Pandas	7.87
Pandas (manual pushdown)	0.17

We can see that the performance difference between doing the pushdown and not doing the pushdown is dramatic. When we perform the pushdown, Pandas has performance in the same ballpark as DuckDB. Without the pushdown, however, it is loading the entire file from disk, including the other 15 columns that are not required to answer the query.

Joins

Now for the final query that we saw in the join section previously. To recap:

```
SELECT
    l_returnflag,
    l_linestatus,
    sum(l_extendedprice),
    min(l_extendedprice),
    max(l_extendedprice),
    avg(l_extendedprice)
FROM lineitem
JOIN orders ON (l_orderkey = o_orderkey)
WHERE l_shipdate <= DATE '1998-09-02'
  AND o_orderstatus='O'
GROUP BY
    l_returnflag,
    l_linestatus;
```

For Pandas we again create two versions. A naive version, and a manually optimized version. The exact code used can be found in Google Colab.

Name	Time (s)
DuckDB (1 Thread)	1.04
DuckDB (2 Threads)	0.89
Pandas	20.4
Pandas (manual pushdown)	3.95

We see that for this more complex query the slight difference in performance between running over a Pandas DataFrame and a Parquet file vanishes, and the DuckDB timings become extremely similar to the timings we saw before. The added Parquet read again increases the necessity of manually performing optimizations on the Pandas code, which is not required at all when running SQL in DuckDB.

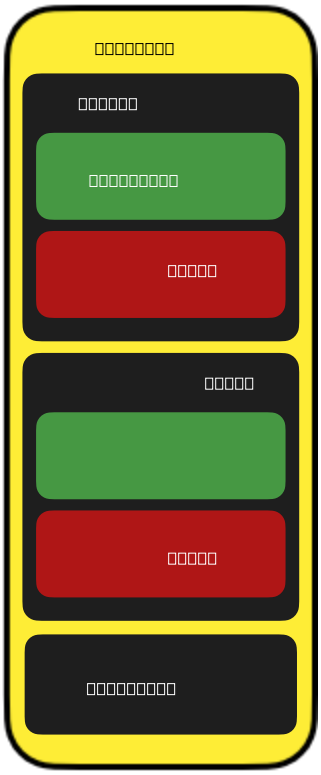
Querying Parquet with Precision Using DuckDB

Publication date: 2021-06-25

Authors: Hannes Mühleisen and Mark Raasveldt

TL;DR: DuckDB, a free and open source analytical data management system, can run SQL queries directly on Parquet files and automatically take advantage of the advanced features of the Parquet format.

Apache Parquet is the most common "Big Data" storage format for analytics. In Parquet files, data is stored in a columnar-compressed binary format. Each Parquet file stores a single table. The table is partitioned into row groups, which each contain a subset of the rows of the table. Within a row group, the table data is stored in a columnar fashion.



The Parquet format has a number of properties that make it suitable for analytical use cases:

1. The columnar representation means that individual columns can be (efficiently) read. No need to always read the entire file!
2. The file contains per-column statistics in every row group (min/max value, and the number of NULL values). These statistics allow the reader to skip row groups if they are not required.
3. The columnar compression significantly reduces the file size of the format, which in turn reduces the storage requirement of data sets. This can often turn Big Data into Medium Data.

DuckDB and Parquet

DuckDB's zero-dependency Parquet reader is able to directly execute SQL queries on Parquet files without any import or analysis step. Because of the natural columnar format of Parquet, this is very fast!

DuckDB will read the Parquet files in a streaming fashion, which means you can perform queries on large Parquet files that do not fit in your main memory.

DuckDB is able to automatically detect which columns and rows are required for any given query. This allows users to analyze much larger and more complex Parquet files without needing to perform manual optimizations or investing in more hardware.

And as an added bonus, DuckDB is able to do all of this using parallel processing and over multiple Parquet files at the same time using the glob syntax.

As a short teaser, here is a code snippet that allows you to directly run a SQL query on top of a Parquet file.

To install the DuckDB package:

```
pip install duckdb
```

To download the Parquet file:

```
wget https://blobs.duckdb.org/data/taxi_2019_04.parquet
```

Then, run the following Python script:

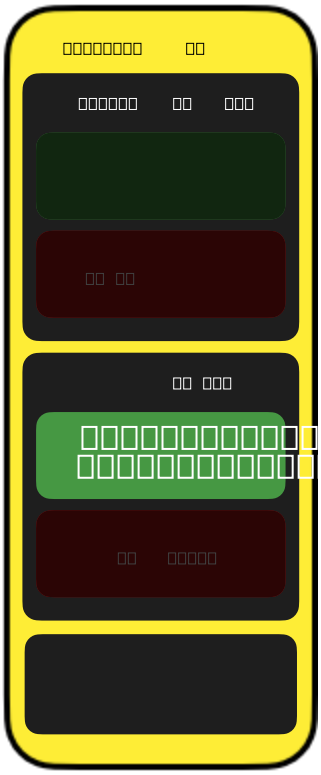
```
import duckdb

print(duckdb.query('''
   SELECT count(*)
   FROM 'taxi_2019_04.parquet'
   WHERE pickup_at BETWEEN '2019-04-15' AND '2019-04-20'
   ''').fetchall())
```

Automatic Filter & Projection Pushdown

Let us dive into the previous query to better understand the power of the Parquet format when combined with DuckDB's query optimizer.

```
SELECT count(*)
FROM 'taxi_2019_04.parquet'
WHERE pickup_at BETWEEN '2019-04-15' AND '2019-04-20';
```

In this query, we read a single column from our Parquet file (`pickup_at`). Any other columns stored in the Parquet file can be entirely skipped, as we do not need them to answer our query.



In addition, only rows that have a `pickup_at` between the 15th and the 20th of April 2019 influence the result of the query. Any rows that do not satisfy this predicate can be skipped.

We can use the statistics inside the Parquet file to great advantage here. Any row groups that have a max value of `pickup_at` lower than `2019-04-15`, or a min value higher than `2019-04-20`, can be skipped. In some cases, that allows us to skip reading entire files.

DuckDB versus Pandas

To illustrate how effective these automatic optimizations are, we will run a number of queries on top of Parquet files using both Pandas and DuckDB.

In these queries, we use a part of the infamous New York Taxi dataset stored as Parquet files, specifically data from April, May and June 2019. These files are ca. 360 MB in size together and contain around 21 million rows of 18 columns each. The three files are placed into the `taxi/` folder.

The examples are available as an interactive notebook over at Google Colab. The timings reported here are from this environment for reproducibility.

Reading Multiple Parquet Files

First we look at some rows in the dataset. There are three Parquet files in the `taxi/` folder. DuckDB supports the globbing syntax, which allows it to query all three files simultaneously.

```
con.execute("""
   SELECT *
   FROM 'taxi/*.parquet'
   LIMIT 5""").df()
```

pickup_at	dropoff_at	passenger_count	trip_distance	rate_code_id
2019-04-01 00:04:09	2019-04-01 00:06:35	1	0.5	1
2019-04-01 00:22:45	2019-04-01 00:25:43	1	0.7	1
2019-04-01 00:39:48	2019-04-01 01:19:39	1	10.9	1
2019-04-01 00:35:32	2019-04-01 00:37:11	1	0.2	1
2019-04-01 00:44:05	2019-04-01 00:57:58	1	4.8	1

Despite the query selecting all columns from three (rather large) Parquet files, the query completes instantly. This is because DuckDB processes the Parquet file in a streaming fashion, and will stop reading the Parquet file after the first few rows are read as that is all required to satisfy the query.

If we try to do the same in Pandas, we realize it is not so straightforward, as Pandas cannot read multiple Parquet files in one call. We will first have to use `pandas.concat` to concatenate the three Parquet files together:

```
import pandas
import glob
df = pandas.concat(
    [pandas.read_parquet(file)
     for file
     in glob.glob('taxi/*.parquet')])
print(df.head(5))
```

Below are the timings for both of these queries.

System	Time (s)
DuckDB	0.015
Pandas	12.300

Pandas takes significantly longer to complete this query. That is because Pandas not only needs to read each of the three Parquet files in their entirety, it has to concatenate these three separate Pandas DataFrames together.

Concatenate into a Single File

We can address the concatenation issue by creating a single big Parquet file from the three smaller parts. We can use the `pyarrow` library for this, which has support for reading multiple Parquet files and streaming them into a single large file. Note that the `pyarrow` parquet reader is the very same parquet reader that is used by Pandas internally.

```
import pyarrow.parquet as pq

# concatenate all three parquet files
pq.write_table(pq.ParquetDataset('taxi/').read(), 'alltaxi.parquet', row_group_size=100000)
```

Note that DuckDB also has support for writing Parquet files using the COPY statement.

Querying the Large File

Now let us repeat the previous experiment, but using the single file instead.

```
# DuckDB
con.execute("""
   SELECT *
   FROM 'alltaxi.parquet'
   LIMIT 5""").df()

# Pandas
pandas.read_parquet('alltaxi.parquet')
      .head(5)
```

System	Time (s)
DuckDB	0.02
Pandas	7.50

We can see that Pandas performs better than before, as the concatenation is avoided. However, the entire file still needs to be read into memory, which takes both a significant amount of time and memory.

For DuckDB it does not really matter how many Parquet files need to be read in a query.

Counting Rows

Now suppose we want to figure out how many rows are in our data set. We can do that using the following code:

```
# DuckDB
con.execute("""
   SELECT count(*)
   FROM 'alltaxi.parquet'
""").df()

# Pandas
len(pandas.read_parquet('alltaxi.parquet'))
```

System	Time (s)
DuckDB	0.015
Pandas	7.500

DuckDB completes the query very quickly, as it automatically recognizes what needs to be read from the Parquet file and minimizes the required reads. Pandas has to read the entire file again, which causes it to take the same amount of time as the previous query.

For this query, we can improve Pandas' time through manual optimization. In order to get a count, we only need a single column from the file. By manually specifying a single column to be read in the `read_parquet` command, we can get the same result but much faster.

```
len(pandas.read_parquet('alltaxi.parquet', columns=['vendor_id']))
```

System	Time (s)
DuckDB	0.015
Pandas	7.500
Pandas (optimized)	1.200

While this is much faster, this still takes more than a second as the entire `vendor_id` column has to be read into memory as a Pandas column only to count the number of rows.

Filtering Rows

It is common to use some sort of filtering predicate to only look at the interesting parts of a data set. For example, imagine we want to know how many taxi rides occur after the 30th of June 2019. We can do that using the following query in DuckDB:

```
con.execute("""
   SELECT count(*)
   FROM 'alltaxi.parquet'
   WHERE pickup_at > '2019-06-30'
""").df()
```

The query completes in 45ms and yields the following result:

count
167022

In Pandas, we can perform the same operation using a naive approach.

```
# pandas naive
len(pandas.read_parquet('alltaxi.parquet')
          .query("pickup_at > '2019-06-30'"))
```

This again reads the entire file into memory, however, causing this query to take 7.5 s. With the manual projection pushdown we can bring this down to 0.9 s. Still significantly higher than DuckDB.

```
# pandas projection pushdown
len(pandas.read_parquet('alltaxi.parquet', columns=['pickup_at'])
          .query("pickup_at > '2019-06-30'"))
```

The pyarrow parquet reader also allows us to perform filter pushdown into the scan, however. Once we add this we end up with a much more competitive 70ms to complete the query.

```
len(pandas.read_parquet('alltaxi.parquet', columns=['pickup_at'], filters=[('pickup_at', '>',
'2019-06-30')]))
```

System	Time (s)
DuckDB	0.05
Pandas	7.50
Pandas (projection pushdown)	0.90
Pandas (projection & filter pushdown)	0.07

This shows that the results here are not due to DuckDB's parquet reader being faster than the pyarrow Parquet reader. The reason that DuckDB performs better on these queries is because its optimizers automatically extract all required columns and filters from the SQL query, which then get automatically utilized in the Parquet reader with no manual effort required.

Interestingly, both the pyarrow Parquet reader and DuckDB are significantly faster than performing this operation natively in Pandas on a materialized DataFrame.

```
# read the entire parquet file into Pandas
df = pandas.read_parquet('alltaxi.parquet')
# run the query natively in Pandas
# note: we only time this part
print(len(df[['pickup_at']].query("pickup_at > '2019-06-30'")))
```

System	Time (s)
DuckDB	0.05
Pandas	7.50
Pandas (projection pushdown)	0.90
Pandas (projection & filter pushdown)	0.07
Pandas (native)	0.26

Aggregates

Finally lets look at a more complex aggregation. Say we want to compute the number of rides per passenger. With DuckDB and SQL, it looks like this:

```
con.execute("""
    SELECT passenger_count, count(*)
    FROM 'alltaxi.parquet'
    GROUP BY passenger_count""").df()
```

The query completes in `220ms` and yields the following result:

passenger_count	count
0	408742
1	15356631
2	3332927
3	944833
4	439066
5	910516
6	546467
7	106
8	72
9	64

For the SQL-averse and as a teaser for a future blog post, DuckDB also has a "Relational API" that allows for a more Python-esque declaration of queries. Here's the equivalent to the above SQL query, that provides the exact same result and performance:

```
con.from_parquet('alltaxi.parquet')
   .aggregate('passenger_count, count(*)')
   .df()
```

Now as a comparison, let's run the same query in Pandas in the same way we did previously.

```
# naive
pandas.read_parquet('alltaxi.parquet')
      .groupby('passenger_count')
      .agg({'passenger_count' : 'count'})

# projection pushdown
pandas.read_parquet('alltaxi.parquet', columns=['passenger_count'])
      .groupby('passenger_count')
```

```
      .agg({'passenger_count' : 'count'})

# native (parquet file pre-loaded into memory)
df.groupby('passenger_count')
  .agg({'passenger_count' : 'count'})
```

System	Time (s)
DuckDB	0.22
Pandas	7.50
Pandas (projection pushdown)	0.58
Pandas (native)	0.51

We can see that DuckDB is faster than Pandas in all three scenarios, without needing to perform any manual optimizations and without needing to load the Parquet file into memory in its entirety.

Conclusion

DuckDB can efficiently run queries directly on top of Parquet files without requiring an initial loading phase. The system will automatically take advantage of all of Parquet's advanced features to speed up query execution.

DuckDB is a free and open source database management system (MIT licensed). It aims to be the SQLite for Analytics, and provides a fast and efficient database system with zero external dependencies. It is available not just for Python, but also for C/C++, R, Java, and more.

Fastest Table Sort in the West – Redesigning DuckDB's Sort

Publication date: 2021-08-27

Author: Laurens Kuiper

TL;DR: DuckDB, a free and open-source analytical data management system, has a new highly efficient parallel sorting implementation that can sort much more data than fits in main memory.

Database systems use sorting for many purposes, the most obvious purpose being when a user adds an ORDER BY clause to their query. Sorting is also used within operators, such as window functions. DuckDB recently improved its sorting implementation, which is now able to sort data in parallel and sort more data than fits in memory. In this post, we will take a look at how DuckDB sorts, and how this compares to other data management systems.

Not interested in the implementation? Jump straight to the experiments!

Sorting Relational Data

Sorting is one of the most well-studied problems in computer science, and it is an important aspect of data management. There are entire communities dedicated to who sorts fastest. Research into sorting algorithms tends to focus on sorting large arrays or key/value pairs. While important, this does not cover how to implement sorting in a database system. There is a lot more to sorting tables than just sorting a large array of integers!

Consider the following example query on a snippet of a TPC-DS table:

```
SELECT c_customer_sk, c_birth_country, c_birth_year
FROM customer
ORDER BY c_birth_country DESC,
         c_birth_year    ASC NULLS LAST;
```

Which yields:

c_customer_sk	c_birth_country	c_birth_year
64760	NETHERLANDS	1991
75011	NETHERLANDS	1992
89949	NETHERLANDS	1992
90766	NETHERLANDS	NULL
42927	GERMANY	1924

In other words: `c_birth_country` is ordered descendingly, and where `c_birth_country` is equal, we sort on `c_birth_year` ascendingly. By specifying NULLS LAST, null values are treated as the lowest value in the `c_birth_year` column. Whole rows are thus reordered, not just the columns in the ORDER BY clause. The columns that are not in the ORDER BY clause we call "payload columns". Therefore, payload column `c_customer_sk` has to be reordered too.

It is easy to implement something that can evaluate the example query using any sorting implementation, for instance, **C++**'s `std::sort`. While `std::sort` is excellent algorithmically, it is still a single-threaded approach that is unable to efficiently sort by multiple columns because function call overhead would quickly dominate sorting time. Below we will discuss why that is.

To achieve good performance when sorting tables, a custom sorting implementation is needed. We are – of course – not the first to implement relational sorting, so we dove into the literature to look for guidance.

In 2006 the famous Goetz Graefe wrote a survey on implementing sorting in database systems. In this survey, he collected many sorting techniques that are known to the community. This is a great guideline if you are about to start implementing sorting for tables.

The cost of sorting is dominated by comparing values and moving data around. Anything that makes these two operations cheaper will have a big impact on the total runtime.

There are two obvious ways to go about implementing a comparator when we have multiple ORDER BY clauses:

1. Loop through the clauses: Compare columns until we find one that is not equal, or until we have compared all columns. This is fairly complex already, as this requires a loop with an if/else inside of it for every single row of data. If we have columnar storage, this comparator has to jump between columns, causing random access in memory.
2. Entirely sort the data by the first clause, then sort by the second clause, but only where the first clause was equal, and so on. This approach is especially inefficient when there are many duplicate values, as it requires multiple passes over the data.

Binary String Comparison

The binary string comparison technique improves sorting performance by simplifying the comparator. It encodes *all* columns in the ORDER BY clause into a single binary sequence that, when compared using `memcmp` will yield the correct overall sorting order. Encoding the data is not free, but since we are using the comparator so much during sorting, it will pay off. Let us take another look at 3 rows of the example:

c_birth_country	c_birth_year
NETHERLANDS	1991
NETHERLANDS	1992
GERMANY	1924

On little-endian hardware, the bytes that represent these values look like this in memory, assuming 32-bit integer representation for the year:

```
c_birth_country
-- NETHERLANDS
01001110 01000101 01010100 01001000 01000101 01010010 01001100 01000001 01001110 01000100 01010011
00000000
-- GERMANY
01000111 01000101 01010010 01001101 01000001 01001110 01011001 00000000

c_birth_year
-- 1991
11000111 00000111 00000000 00000000
-- 1992
11001000 00000111 00000000 00000000
-- 1924
10000100 00000111 00000000 00000000
```

The trick is to convert these to a binary string that encodes the sorting order:

```
-- NETHERLANDS | 1991
10110001 10111010 10101011 10110111 10111010 10101101 10110011 10111110 10110001 10111011 10101100
11111111
10000000 00000000 00000111 11000111
-- NETHERLANDS | 1992
10110001 10111010 10101011 10110111 10111010 10101101 10110011 10111110 10110001 10111011 10101100
11111111
```

```
10000000 00000000 00000111 11001000
-- GERMANY      | 1924
10111000 10111010 10101101 10110010 10111110 10110001 10100110 11111111 11111111 11111111 11111111
11111111
10000000 00000000 00000111 10000100
```

The binary string is fixed-size because this makes it much easier to move it around during sorting.

The string "GERMANY" is shorter than "NETHERLANDS", therefore it is padded with `00000000`'s. All bits in column `c_birth_country` are subsequently inverted because this column is sorted descendingly. If a string is too long we encode its prefix and only look at the whole string if the prefixes are equal.

The bytes in `c_birth_year` are swapped because we need the big-endian representation to encode the sorting order. The first bit is also flipped, to preserve order between positive and negative integers for signed integers. If there are `NULL` values, these must be encoded using an additional byte (not shown in the example).

With this binary string, we can now compare both columns at the same time by comparing only the binary string representation. This can be done with a single `memcmp` in **C++**! The compiler will emit efficient assembly for single function call, even auto-generating SIMD instructions.

This technique solves one of the problems mentioned above, namely the function call overhead when using complex comparators.

Radix Sort

Now that we have a cheap comparator, we have to choose our sorting algorithm. Every computer science student learns about comparison-based sorting algorithms like Quicksort and Merge sort, which have $O(n \log n)$ time complexity, where n is the number of records being sorted.

However, there are also distribution-based sorting algorithms, which typically have a time complexity of $O(n\,k)$, where k is the width of the sorting key. This class of sorting algorithms scales much better with a larger n because k is constant, whereas $\log n$ is not.

One such algorithm is Radix sort. This algorithm sorts the data by computing the data distribution with Counting sort, multiple times until all digits have been counted.

It may sound counter-intuitive to encode the sorting key columns such that we have a cheap comparator, and then choose a sorting algorithm that does not compare records. However, the encoding is necessary for Radix sort: Binary strings that produce a correct order with `memcmp` will produce a correct order if we do a byte-by-byte Radix sort.

Two-Phase Parallel Sorting

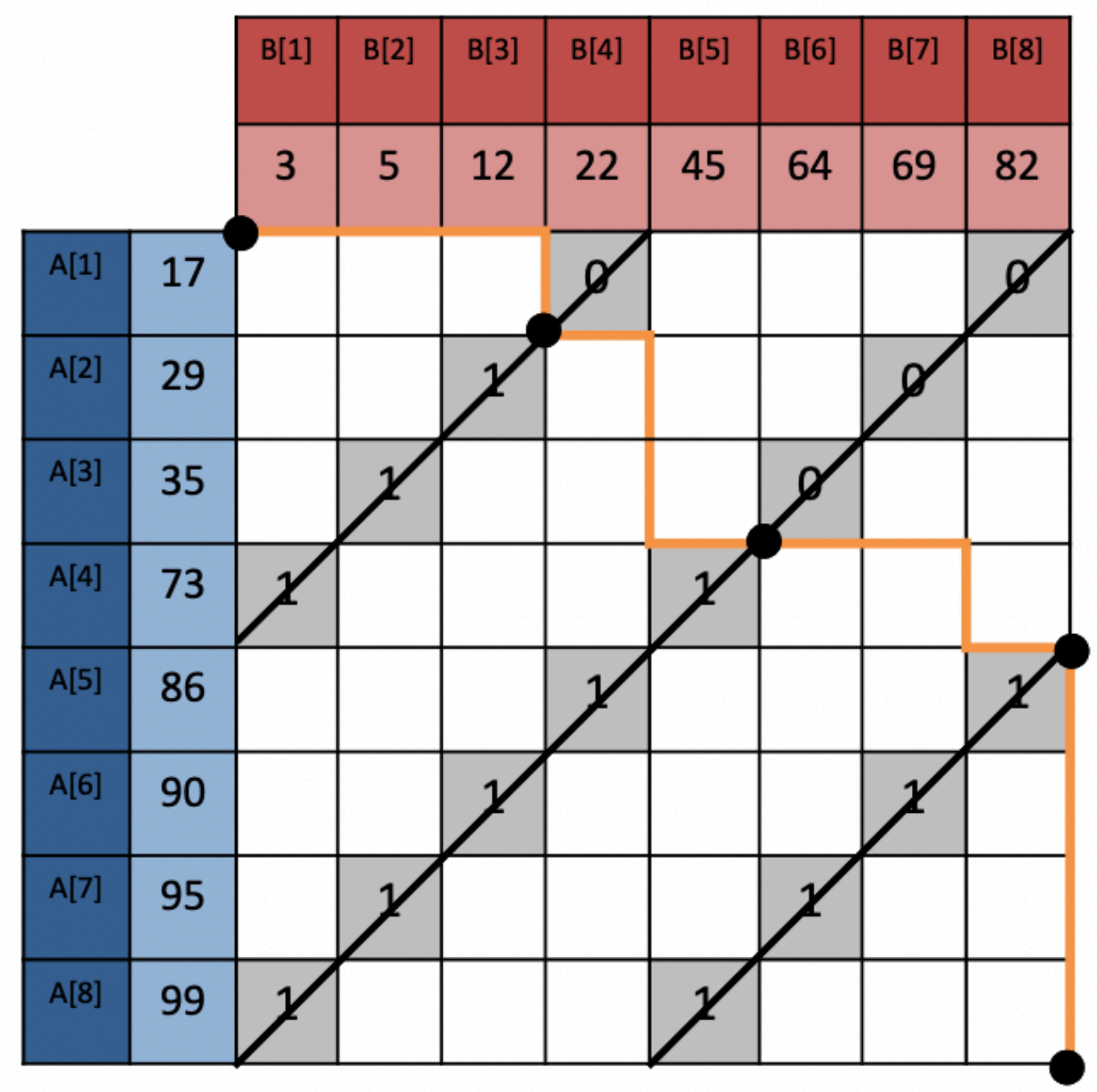
DuckDB uses Morsel-Driven Parallelism, a framework for parallel query execution. For the sorting operator, this means that multiple threads collect roughly an equal amount of data, in parallel, from the table.

We use this parallelism for sorting by first having each thread sort the data it collects using our Radix sort. After this first sorting phase, each thread has one or more sorted blocks of data, which must be combined into the final sorted result. Merge sort is the algorithm of choice for this task. There are two main ways of implementing merge sort: K-way merge and Cascade merge.

K-way merge merges K lists into one sorted list in one pass, and is traditionally used for external sorting (sorting more data than fits in memory) because it minimizes I/O. Cascade merge merges two lists of sorted data at a time until only one sorted list remains, and is used for in-memory sorting because it is more efficient than K-way merge. We aim to have an implementation that has high in-memory performance, which gracefully degrades as we go over the limit of available memory. Therefore, we choose cascade merge.

In a cascade merge sort, we merge two blocks of sorted data at a time until only one sorted block remains. Naturally, we want to use all available threads to compute the merge. If we have many more sorted blocks than threads, we can assign each thread to merge two blocks. However, as the blocks get merged, we will not have enough blocks to keep all threads busy. This is especially slow when the final two blocks are merged: One thread has to process all the data.

To fully parallelize this phase, we have implemented Merge Path by Oded Green et al. Merge Path pre-computes *where* the sorted lists will intersect while merging, shown in the image below (taken from the paper).



The intersections along the merge path can be efficiently computed using Binary Search. If we know where the intersections are, we can merge partitions of the sorted data independently in parallel. This allows us to use all available threads effectively for the entire merge phase. For another trick to improve merge sort, see the appendix.

Columns or Rows?

Besides comparisons, the other big cost of sorting is moving data around. DuckDB has a vectorized execution engine. Data is stored in a columnar layout, which is processed in batches (called chunks) at a time. This layout is great for analytical query processing because the chunks fit in the CPU cache, and it gives a lot of opportunities for the compiler to generate SIMD instructions. However, when the table is sorted, entire rows are shuffled around, rather than columns.

We could stick to the columnar layout while sorting: Sort the key columns, then re-order the payload columns one by one. However, re-ordering will cause a random access pattern in memory for each column. If there are many payload columns, this will be slow. Converting the columns to rows will make re-ordering rows much easier. This conversion is of course not free: Columns need to be copied to rows, and back from rows to columns again after sorting.

Because we want to support external sorting, we have to store data in buffer-managed blocks that can be offloaded to disk. Because we have to copy the input data to these blocks anyway, converting the rows to columns is effectively free.

There are a few operators that are inherently row-based, such as joins and aggregations. DuckDB has a unified internal row layout for these operators, and we decided to use it for the sorting operator as well. This layout has only been used in memory so far. In the next section, we will explain how we got it to work on disk as well. We should note that we will only write sorting data to disk if main memory is not able to hold it.

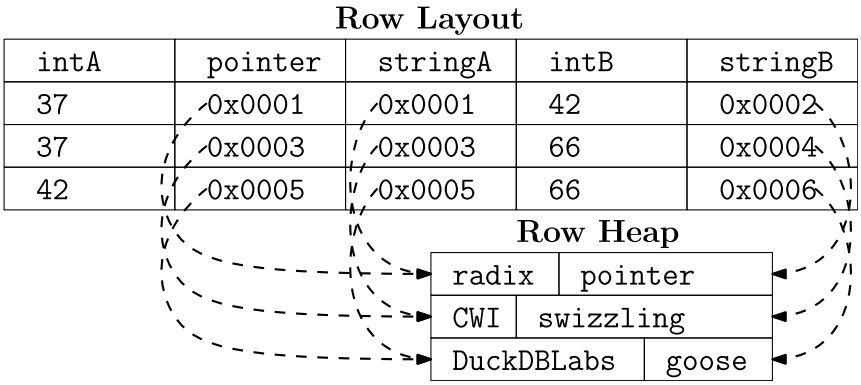
External Sorting

The buffer manager can unload blocks from memory to disk. This is not something we actively do in our sorting implementation, but rather something that the buffer manager decides to do if memory would fill up otherwise. It uses a least-recently-used queue to decide which blocks to write. More on how to properly use this queue in the appendix.

When we need a block, we ”pin” it, which reads it from disk if it is not loaded already. Accessing disk is much slower than accessing memory, therefore it is crucial that we minimize the number of reads and writes.

Unloading data to disk is easy for fixed-size columns like integers, but more difficult for variable-sized columns like strings. Our row layout uses fixed-size rows, which cannot fit strings with arbitrary sizes. Therefore, strings are represented by a pointer, which points into a separate block of memory where the actual string data lives, a so-called ”string heap”.

We have changed our heap to also store strings row-by-row in buffer-managed blocks:



Each row has an additional 8-byte field `pointer` which points to the start of this row in the heap. This is useless in the in-memory representation, but we will see why it is useful for the on-disk representation in just a second.

If the data fits in memory, the heap blocks stay pinned, and only the fixed-size rows are re-ordered while sorting. If the data does not fit in memory, the blocks need to be offloaded to disk, and the heap will also be re-ordered while sorting. When a heap block is offloaded to disk, the pointers pointing into it are invalidated. When we load the block back into memory, the pointers will have changed.

This is where our row-wise layout comes into play. The 8-byte `pointer` field is overwritten with an 8-byte `offset` field, denoting where in the heap block strings of this row can be found. This technique is called ”pointer swizzling”. When we swizzle the pointers, the row layout and heap block look like this:

Row Layout

intA	offset	stringA	intB	stringB
37	0	0	42	5
37	12	0	66	3
42	24	0	66	10

Row Heap

radix	pointer
CWI	swizzling
DuckDBLabs	goose

The pointers to the subsequent string values are also overwritten with an 8-byte relative offset, denoting how far this string is offset from the start of the row in the heap (hence every `stringA` has an offset of `0`: It is the first string in the row). Using relative offsets within rows rather than absolute offsets is very useful during sorting, as these relative offsets stay constant, and do not need to be updated when a row is copied.

When the blocks need to be scanned to read the sorted result, we ”unswizzle” the pointers, making them point to the string again.

With this dual-purpose row-wise representation, we can easily copy around both the fixed-size rows and the variable-sized rows in the heap. Besides having the buffer manager load/unload blocks, the only difference between in-memory and external sorting is that we swizzle/unswizzle pointers to the heap blocks, and copy data from the heap blocks during merge sort.

All this reduces overhead when blocks need to be moved in and out of memory, which will lead to graceful performance degradation as we approach the limit of available memory.

Comparison with Other Systems

Now that we have covered most of the techniques that are used in our sorting implementation, we want to know how we compare to other systems. DuckDB is often used for interactive data analysis, and is therefore often compared to tools like dplyr.

In this setting, people are usually running on laptops or PCs, therefore we will run these experiments on a 2020 MacBook Pro. This laptop has an Apple M1 CPU, which is ARM-based. The M1 processor has 8 cores: 4 high-performance (Firestorm) cores, and 4 energy-efficient (Icestorm) cores. The Firestorm cores have very, very fast single-thread performance, so this should level the playing field between single- and multi-threaded sorting implementations somewhat. The MacBook has 16 GB of memory, and one of the fastest SSDs found in a laptop.

We will be comparing against the following systems:

1. ClickHouse, version 21.7.5
2. HyPer, version 2021.2.1.12564
3. Pandas, version 1.3.2
4. SQLite, version 3.36.0

ClickHouse and HyPer are included in our comparison because they are analytical SQL engines with an emphasis on performance. Pandas and SQLite are included in our comparison because they can be used to perform relational operations within Python, like DuckDB. Pandas operates fully in memory, whereas SQLite is a more traditional disk-based system. This list of systems should give us a good mix of single-/multi-threaded, and in-memory/external sorting.

ClickHouse was built for M1 using this guide. We have set the memory limit to 12 GB, and `max_bytes_before_external_sort` to 10 GB, following this suggestion.

HyPer is Tableau's data engine, created by the database group at the University of Munich. It does not run natively (yet) on ARM-based processors like the M1. We will use Rosetta 2, macOS's x86 emulator to run it. Emulation causes some overhead, so we have included an experiment on an x86 machine in the appendix.

Benchmarking sorting in database systems is not straightforward. Ideally, we would like to measure only the time it takes to sort the data, not the time it takes to read the input data and show the output. Not every system has a profiler to measure the time of the sorting operator exactly, so this is not an option.

To approach a fair comparison, we will measure the end-to-end time of queries that sort the data and write the result to a temporary table, i.e.:

```
CREATE TEMPORARY TABLE output AS
SELECT ...
FROM ...
ORDER BY ...;
```

There is no perfect solution to this problem, but this should give us a good comparison because the end-to-end time of this query should be dominated by sorting. For Pandas we will use `sort_values` with `inplace=False` to mimic this query.

In ClickHouse, temporary tables can only exist in memory, which is problematic for our out-of-core experiments. Therefore we will use a regular `TABLE`, but then we also need to choose a table engine. Most of the table engines apply compression or create an index, which we do not want to measure. Therefore we have chosen the simplest on-disk engine, which is File, with format Native.

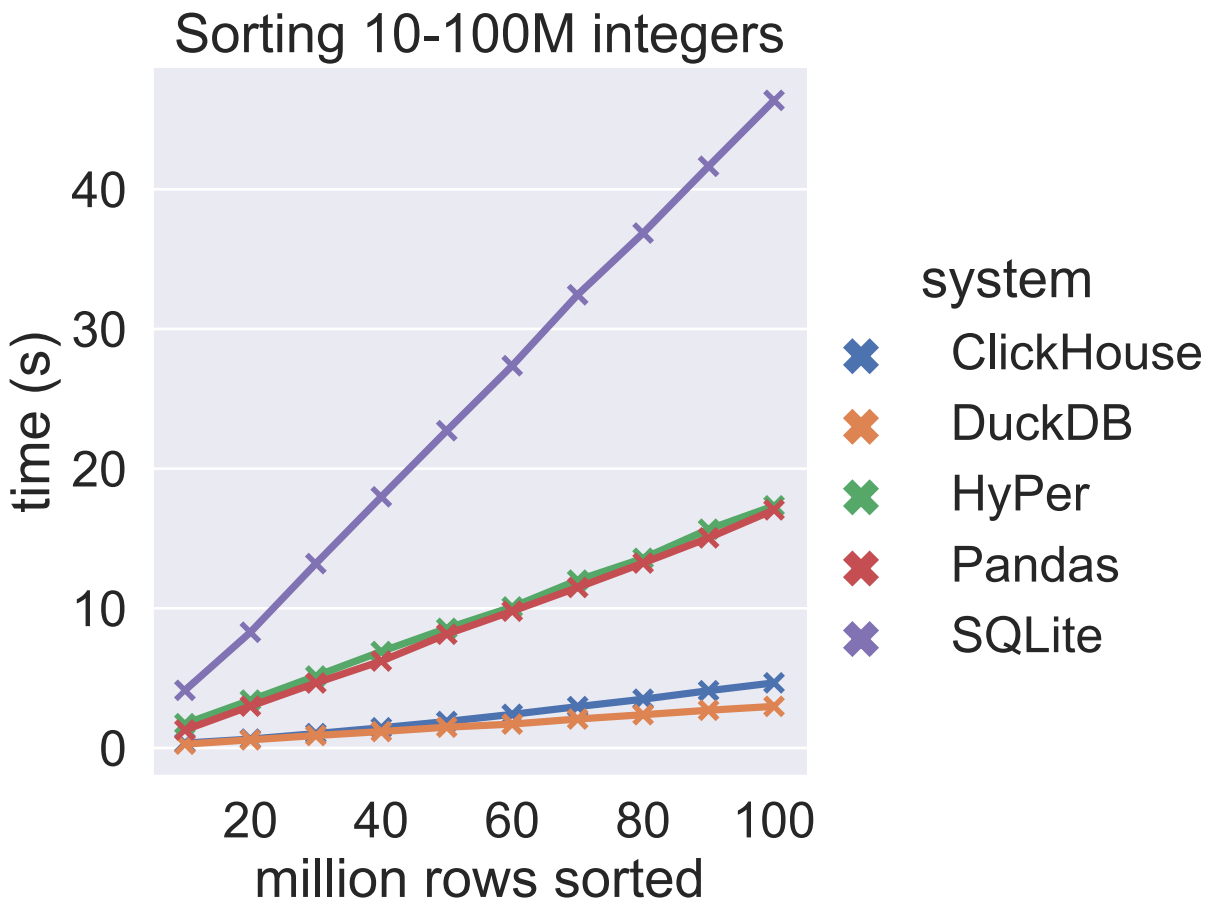
The table engine we chose for the input tables for ClickHouse is MergeTree with `ORDER BY tuple()`. We chose this because we encountered strange behavior with `File(Native)` input tables, where there was no difference in runtime between the queries `SELECT * FROM ... ORDER BY` and `SELECT col1 FROM ... ORDER BY`. Presumably, because all columns in the table were sorted regardless of how many there were selected.

To measure stable end-to-end query time, we run each query 5 times and report the median run time. There are some differences in reading/writing tables between the systems. For instance, Pandas cannot read/write from/to disk, so both the input and output data frame will be in memory. DuckDB will not write the output table to disk unless there is not enough room to keep it in memory, and therefore also may have an advantage. However, sorting dominates the total runtime, so these differences are not that impactful.

Random Integers

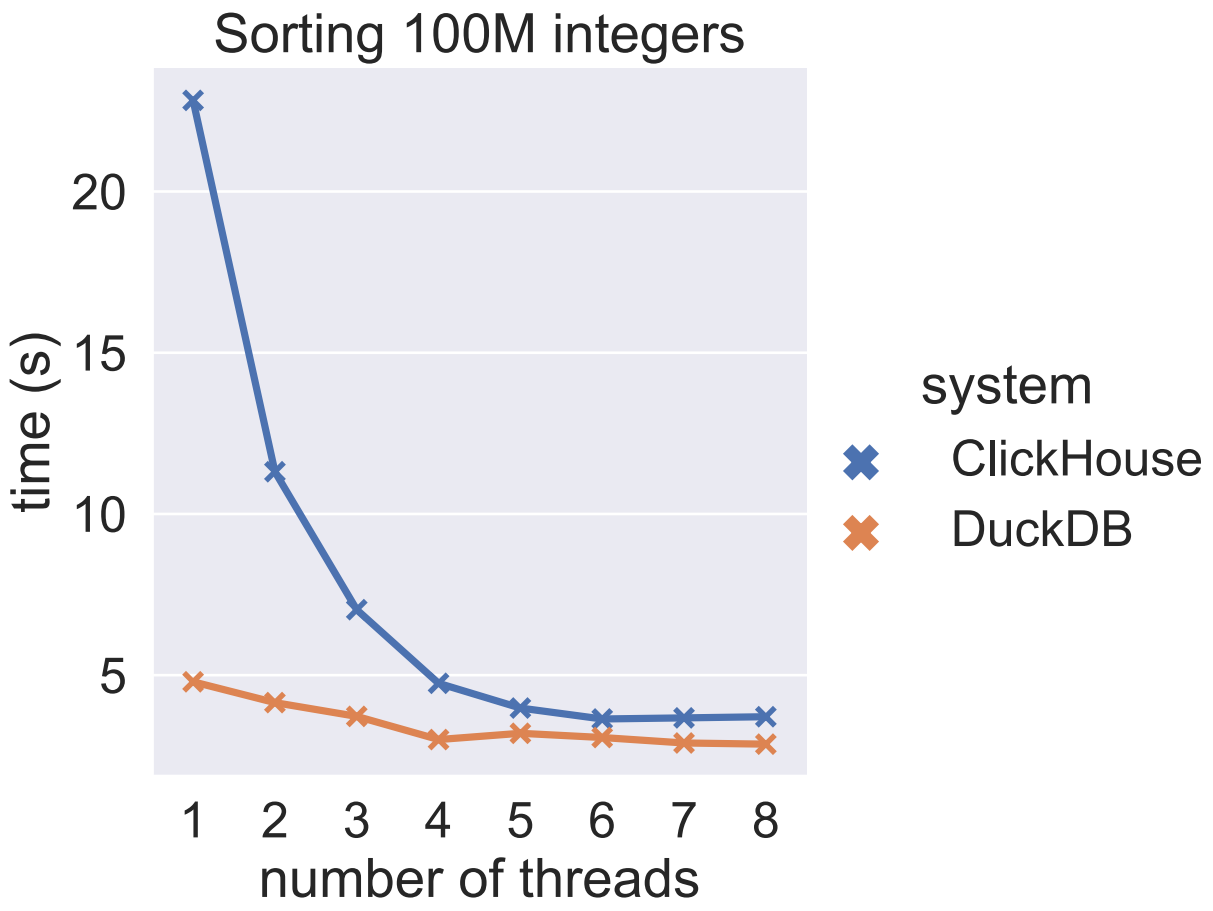
We will start with a simple example. We have generated the first 100 million integers and shuffled them, and we want to know how well the systems can sort them. This experiment is more of a micro-benchmark than anything else and is of little real-world significance.

For our first experiment, we will look at how the systems scale with the number of rows. From the initial table with integers, we have made 9 more tables, with 10M, 20M, ..., 90M integers each.



Being a traditional disk-based database system, SQLite always opts for an external sorting strategy. It writes intermediate sorted blocks to disk even if they fit in main-memory, therefore it is much slower. The performance of the other systems is in the same ballpark, with DuckDB and ClickHouse going toe-to-toe with ~3 and ~4 seconds for 100M integers. Because SQLite is so much slower, we will not include it in our next set of experiments (TPC-DS).

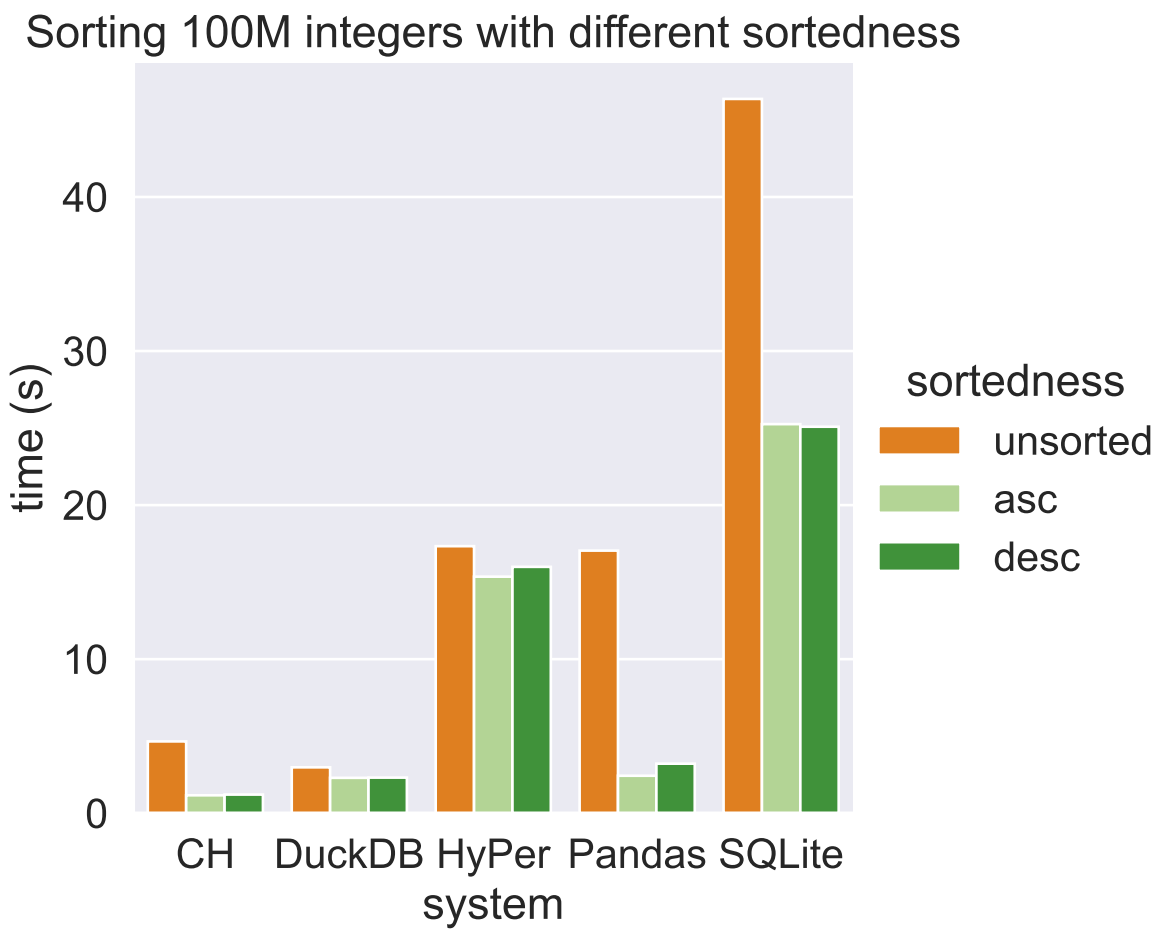
DuckDB and ClickHouse both make very good use out of all available threads, with a single-threaded sort in parallel, followed by a parallel merge sort. We are not sure what strategy HyPer uses. For our next experiment, we will zoom in on multi-threading, and see how well ClickHouse and DuckDB scale with the number of threads (we were not able to set the number of threads for HyPer).



This plot demonstrates that Radix sort is very fast. DuckDB sorts 100M integers in just under 5 seconds using a single thread, which is much faster than ClickHouse. Adding threads does not improve performance as much for DuckDB, because Radix Sort is so much faster than Merge Sort. Both systems end up at about the same performance at 4 threads.

Beyond 4 threads we do not see performance improve much more, due to the CPU architecture. For all of the of other the experiments, we have set both DuckDB and ClickHouse to use 4 threads.

For our last experiment with random integers, we will see how the sortedness of the input may impact performance. This is especially important to do in systems that use Quicksort because Quicksort performs much worse on inversely sorted data than on random data.



Not surprisingly, all systems perform better on sorted data, sometimes by a large margin. ClickHouse, Pandas, and SQLite likely have some optimization here: e.g., keeping track of sortedness in the catalog, or checking sortedness while scanning the input. DuckDB and HyPer have only a very small difference in performance when the input data is sorted, and do not have such an optimization. For DuckDB the slightly improved performance can be explained due to a better memory access pattern during sorting: When the data is already sorted the access pattern is mostly sequential.

Another interesting result is that DuckDB sorts data faster than some of the other systems can read already sorted data.

TPC-DS

For the next comparison, we have improvised a relational sorting benchmark on two tables from the standard TPC Decision Support benchmark (TPC-DS). TPC-DS is challenging for sorting implementations because it has wide tables (with many columns, unlike the tables in TPC-H), and a mix of fixed- and variable-sized types. The number of rows increases with the scale factor. The tables used here are `catalog_sales` and `customer`.

`catalog_sales` has 34 columns, all fixed-size types (integer and double), and grows to have many rows as the scale factor increases. `customer` has 18 columns (10 integers, and 8 strings), and a decent amount of rows as the scale factor increases. The row counts of both tables at each scale factor are shown in the table below.

SF	customer	catalog_sales
1	100,000	1,441,548
10	500,000	14,401,261
100	2,000,000	143,997,065

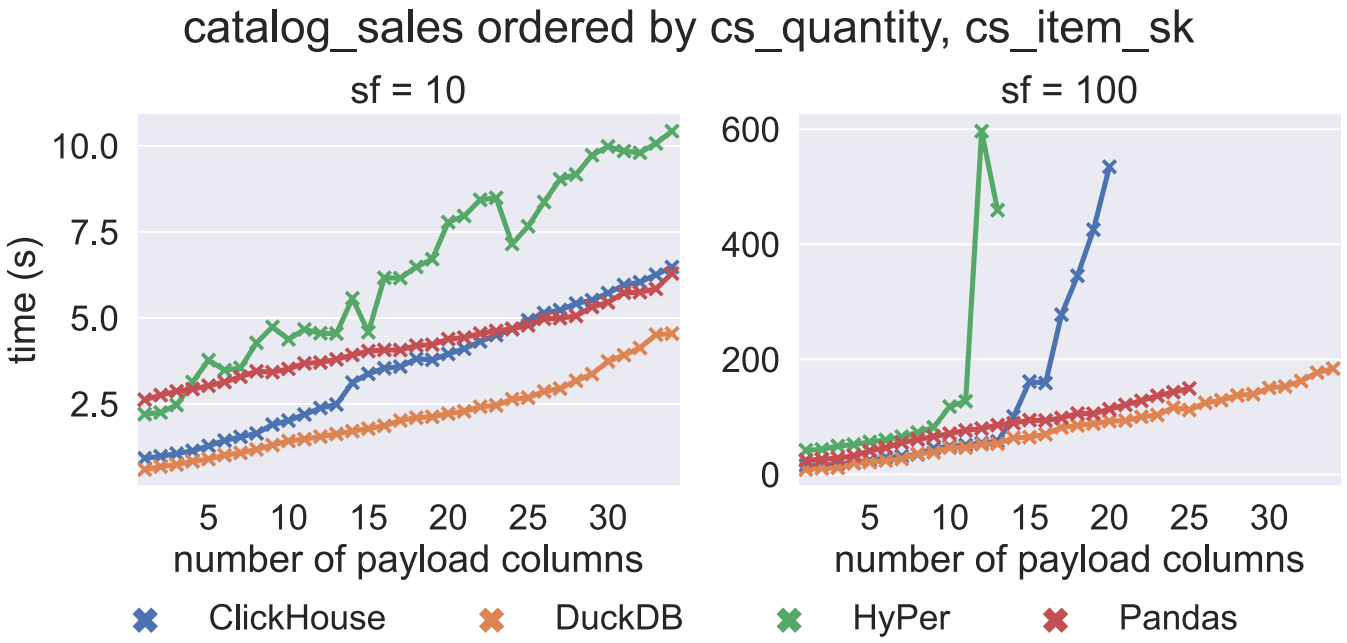
SF	customer	catalog_sales
300	5,000,000	260,014,080

We will use `customer` at SF100 and SF300, which fits in memory at every scale factor. We will use `catalog_sales` table at SF10 and SF100, which does not fit in memory anymore at SF100.

The data was generated using DuckDB's TPC-DS extension, then exported to CSV in a random order to undo any ordering patterns that could have been in the generated data.

Catalog Sales (Numeric Types)

Our first experiment on the `catalog_sales` table is selecting 1 column, then 2 columns, ..., up to all 34, always ordering by `cs_quantity` and `cs_item_sk`. This experiment will tell us how well the different systems can re-order payload columns.



We see similar trends at SF10 and SF100, but for SF100, at around 12 payload columns or so, the data does not fit in memory anymore, and ClickHouse and HyPer show a big drop in performance. ClickHouse switches to an external sorting strategy, which is much slower than its in-memory strategy. Therefore, adding a few payload columns results in a runtime that is orders of magnitude higher. At 20 payload columns ClickHouse runs into the following error:

```
DB::Exception: Memory limit (for query) exceeded: would use 11.18 GiB (attempt to allocate chunk of
4204712 bytes), maximum: 11.18 GiB: (while reading column cs_list_price): (while reading from part
./store/523/5230c288-7ed5-45fa-9230-c2887ed595fa/all_73_108_2/ from mark 4778 with max_rows_to_read =
8192): While executing MergeTreeThread.
```

HyPer also drops in performance before erroring out with the following message:

```
ERROR:  Cannot allocate 333982248 bytes of memory: The `global memory limit` limit of 12884901888 bytes
was exceeded.
```

As far as we are aware, HyPer uses `mmap`, which creates a mapping between memory and a file. This allows the operating system to move data between memory and disk. While useful, it is no substitute for a proper external sort, as it creates random access to disk, which is very slow.

Pandas performs surprisingly well on SF100, despite the data not fitting in memory. Pandas can only do this because macOS dynamically increases swap size. Most operating systems do not do this and would fail to load the data at all. Using swap usually slows down processing significantly, but the SSD is so fast that there is no visible performance drop!

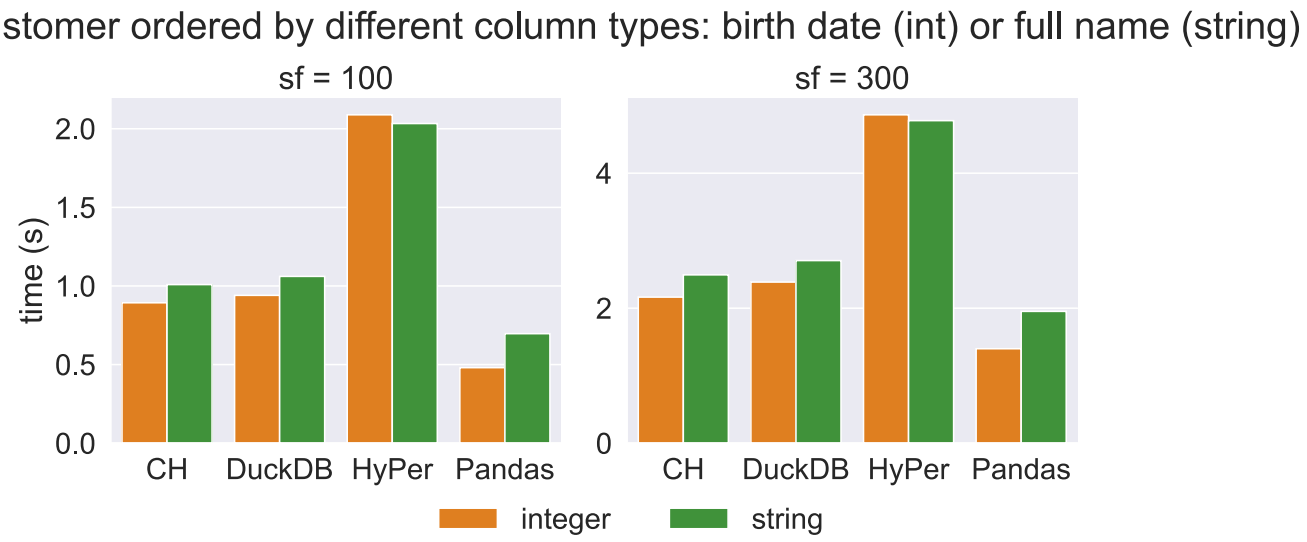
While Pandas loads the data, swap size grows to an impressive ~40 GB: Both the file and the data frame are fully in memory/swap at the same time, rather than streamed into memory. This goes down to ~20 GB of memory/swap when the file is done being read. Pandas is able to get quite far into the experiment until it crashes with the following error:

```
UserWarning: resource_tracker: There appear to be 1 leaked semaphore objects to clean up at shutdown
```

DuckDB performs well both in-memory and external, and there is no clear visible point at which data no longer fits in memory: Runtime is fast and reliable.

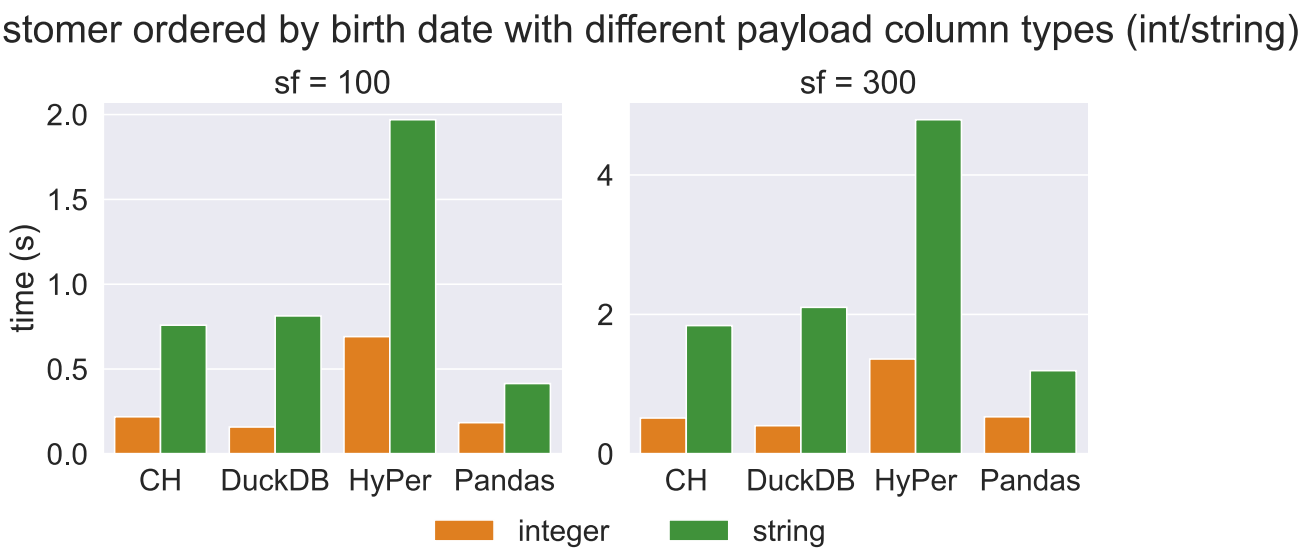
Customer (Strings & Integers)

Now that we have seen how the systems handle large amounts of fixed-size types, it is time to see some variable-size types! For our first experiment on the `customer` table, we will select all columns, and order them by either 3 integer columns (`c_birth_year`, `c_birth_month`, `c_birth_day`), or by 2 string columns (`c_first_name`, `c_last_name`). Comparing strings is much, much more difficult than comparing integers, because strings can have variable sizes, and need to be compare byte-by-byte, whereas integers always have the same comparison.



As expected, ordering by strings is more expensive than ordering by integers, except for HyPer, which is impressive. Pandas has only a slightly bigger difference between ordering by integers and ordering by strings than ClickHouse and DuckDB. This difference is explained by an expensive comparator between strings. Pandas uses NumPy's sort, which is efficiently implemented in **C**. However, when this sorts strings, it has to use virtual function calls to compare a Python string object, which is slower than a simple "<" between integers in **C**. Nevertheless, Pandas performs well on the `customer` table.

In our next experiment, we will see how the payload type affects performance. `customer` has 10 integer columns and 8 string columns. We will either select all integer columns or all string columns and order by (`c_birth_year`, `c_birth_month`, `c_birth_day`) every time.



As expected, re-ordering strings takes much more time than re-ordering integers. Pandas has an advantage here because it already has the strings in memory, and most likely only needs to re-order pointers to these strings. The database systems need to copy strings twice: Once when reading the input table, and again when creating the output table. Profiling in DuckDB reveals that the actual sorting takes less than a second at SF300, and most time is spent on (de)serializing strings.

Conclusion

DuckDB's new parallel sorting implementation can efficiently sort more data than fits in memory, making use of the speed of modern SSDs. Where other systems crash because they run out of memory, or switch to an external sorting strategy that is much slower, DuckDB's performance gracefully degrades as it goes over the memory limit.

The code that was used to run the experiments can be found on GitHub. If we made any mistakes, please let us know!

DuckDB is a free and open-source database management system (MIT licensed). It aims to be the SQLite for Analytics, and provides a fast and efficient database system with zero external dependencies. It is available not just for Python, but also for C/C++, R, Java, and more.

Discuss this post on Hacker News

Read our paper on sorting at ICDE '23

Listen to Laurens' appearance on the Disseminate podcast:

- Spotify
- Google
- Apple

Appendix A: Predication

Another technique we have used to speed up merge sort is *predication*. With this technique, we turn code with *if/else* branches into code without branches. Modern CPUs try to predict whether the *if*, or the *else* branch will be predicted. If this is hard to predict, it can slow down the code. Take a look at the example of pseudo-code with branches below.

```
// continue until merged
while (l_ptr && r_ptr) {
  // check which side is smaller
  if (memcmp(l_ptr, r_ptr, entry) < 0) {
    // copy from left side and advance
    memcpy(result_ptr, l_ptr, entry);
```

```
    l_ptr += entry;
  } else {
    // copy from right side and advance
    memcpy(result_ptr, r_ptr, entry);
    r_ptr += entry;
  }
  // advance result
  result_ptr += entry;
}
```

We are merging the data from the left and right blocks into a result block, one entry at a time, by advancing pointers. This code can be made *branchless* by using the comparison boolean as a 0 or 1, shown in the pseudo-code below.

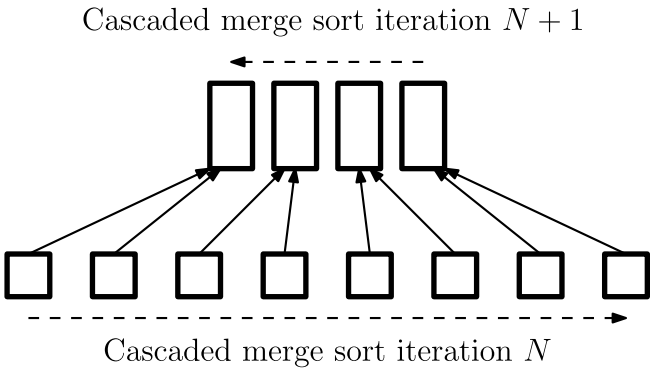
```
// continue until merged
while (l_ptr && r_ptr) {
  // store comparison result in a bool
  bool left_less = memcmp(l_ptr, r_ptr, entry) < 0;
  bool right_less = 1 - left_less;
  // copy from either side
  memcpy(result_ptr, l_ptr, left_less * entry);
  memcpy(result_ptr, r_ptr, right_less * entry);
  // advance either one
  l_ptr += left_less * entry;
  l_ptr += right_less * entry;
  // advance result
  result_ptr += entry;
}
```

When `left_less` is true, it is equal to 1. This means `right_less` is false, and therefore equal to 0. We use this to copy `entry` bytes from the left side, and 0 bytes from the right side, and incrementing the left and right pointers accordingly.

With predicated code, the CPU does not have to predict which instructions to execute, which means there will be fewer instruction cache misses!

Appendix B: Zig-Zagging

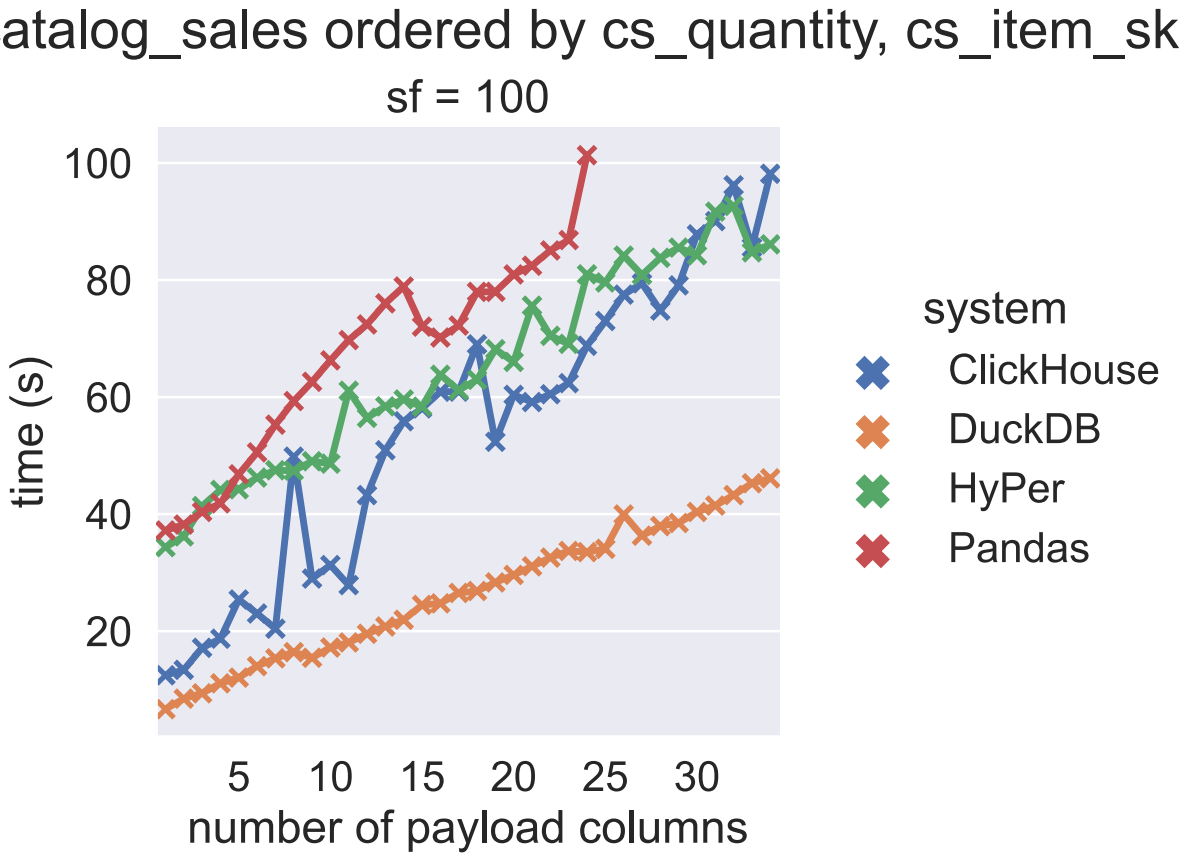
A simple trick to reduce I/O is zig-zagging through the pairs of blocks to merge in the cascaded merge sort. This is illustrated in the image below (dashes arrows indicate in which order the blocks are merged).



By zig-zagging through the blocks, we start an iteration by merging the last blocks that were merged in the previous iteration. Those blocks are likely still in memory, saving us some precious read/write operations.

Appendix C: x86 Experiment

We also ran the `catalog_sales` SF100 experiment on a machine with x86 CPU architecture, to get a more fair comparison with HyPer (without Rosetta 2 emulation). The machine has an Intel(R) Xeon(R) W-2145 CPU @ 3.70 GHz, which has 8 cores (up to 16 virtual threads), and 128 GB of RAM, so this time the data fits fully in memory. We have set the number of threads that DuckDB and ClickHouse use to 8 because we saw no visibly improved performance past 8.



Pandas performs comparatively worse than on the MacBook, because it has a single-threaded implementation, and this CPU has a lower single-thread performance. Again, Pandas crashes with an error (this machine does not dynamically increase swap):

```
numpy.core._exceptions.MemoryError: Unable to allocate 6.32 GiB for an array with shape (6, 141430723) and data type float64
```

DuckDB, HyPer, and ClickHouse all make good use out of more available threads, being significantly faster than on the MacBook.

An interesting pattern in this plot is that DuckDB and HyPer scale very similarly with additional payload columns. Although DuckDB is faster at sorting, re-ordering the payload seems to cost about the same for both systems. Therefore it is likely that HyPer also uses a row layout.

ClickHouse scales worse with additional payload columns. ClickHouse does not use a row layout, and therefore has to pay the cost of random access as each column is re-ordered after sorting.

Windowing in DuckDB

Publication date: 2021-10-13

Author: Richard Wesley

TL;DR: DuckDB, a free and open-source analytical data management system, has a state-of-the-art windowing engine that can compute complex moving aggregates like inter-quartile ranges as well as simpler moving averages.

Window functions (those using the OVER clause) are important tools for analyzing data series, but they can be slow if not implemented carefully. In this post, we will take a look at how DuckDB implements windowing. We will also see how DuckDB can leverage its aggregate function architecture to compute useful moving aggregates such as moving inter-quartile ranges (IQRs).

Beyond Sets

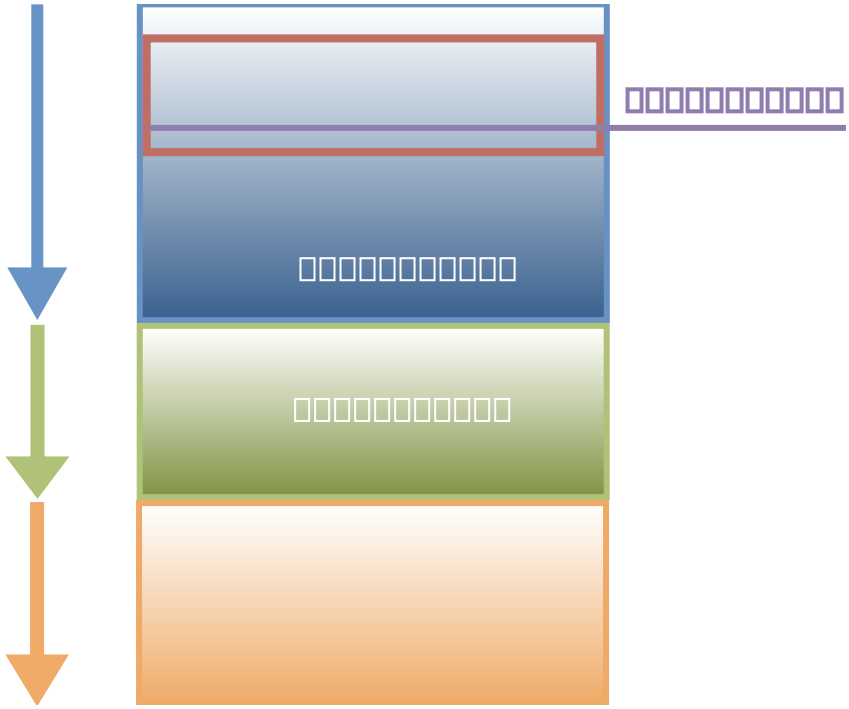
The original relational model as developed by Codd in the 1970s treated relations as *unordered sets* of tuples. While this was nice for theoretical computer science work, it ignored the way humans think using physical analogies (the "embodied brain" model from neuroscience). In particular, humans naturally order data to help them understand it and engage with it. To help with this, SQL uses the `SELECT` clause for horizontal layout and the `ORDER BY` clause for vertical layout.

Still, the orderings that humans put on data are often more than neurological crutches. For example, time places a natural ordering on measurements, and wide swings in those measurements can themselves be important data, or they may indicate that the data needs to be cleaned by smoothing. Trends may be present or relative changes may be more important for analysis than raw values. To help answer such questions, SQL introduced *analytic* (or *window*) functions in 2003.

Window Functions

Windowing works by breaking a relation up into independent *partitions*, *ordering* those partitions, and then defining various functions that can be computed for each row using the nearby values. These functions include all the aggregate functions (such as `sum` and `avg`) as well as some window-specific functions (such as `rank()` and `nth_value(<expression>, <N>)`).

Some window functions depend only on the partition boundary and the ordering, but a few (including all the aggregates) also use a *frame*. Frames are specified as a number of rows on either side (*preceding* or *following*) of the *current row*. The distance can either be specified as a number of *rows* or a *range* of values using the partition's ordering value and a distance.



Framing is the most confusing part of the windowing environment, so let's look at a very simple example and ignore the partitioning and ordering for a moment.

```
SELECT points,
    sum(points) OVER (
        ROWS BETWEEN 1 PRECEDING
                 AND 1 FOLLOWING) AS we
FROM results;
```

This query computes the sum of each point and the points on either side of it:

POINTS	WE
10	18
8	30
12	29
9	30
9	27
9	23
5	21
7	12

POINTS	WE
10	18
8	30
12	29
9	30
9	27
9	23
5	21
7	12

POINTS	WE
10	18
8	30
12	29
9	30
9	27
9	23
5	21
7	12

POINTS	WE
10	18
8	30
12	29
9	30
9	27
9	23
5	21
7	12

Notice that at the edge of the partition, there are only two values added together.

Power Generation Example

Now let's look at a concrete example of a window function query. Suppose we have some power plant generation data:

Plant	Date	MWh
Boston	2019-01-02	564337
Boston	2019-01-03	507405
Boston	2019-01-04	528523
Boston	2019-01-05	469538
Boston	2019-01-06	474163
Boston	2019-01-07	507213
Boston	2019-01-08	613040
Boston	2019-01-09	582588
Boston	2019-01-10	499506
Boston	2019-01-11	482014
Boston	2019-01-12	486134
Boston	2019-01-13	531518
Worcester	2019-01-02	118860
Worcester	2019-01-03	101977
Worcester	2019-01-04	106054
Worcester	2019-01-05	92182
Worcester	2019-01-06	94492
Worcester	2019-01-07	99932
Worcester	2019-01-08	118854
Worcester	2019-01-09	113506
Worcester	2019-01-10	96644
Worcester	2019-01-11	93806
Worcester	2019-01-12	98963
Worcester	2019-01-13	107170

The data is noisy, so we want to compute a 7 day moving average for each plant. To do this, we can use this window query:

```
SELECT "Plant", "Date",
    avg("MWh") OVER (
        PARTITION BY "Plant"
        ORDER BY "Date" ASC
        RANGE BETWEEN INTERVAL 3 DAYS PRECEDING
                  AND INTERVAL 3 DAYS FOLLOWING)
        AS "MWh 7-day Moving Average"
FROM "Generation History"
ORDER BY 1, 2;
```

This query computes the seven day moving average of the power generated by each power plant on each day. The `OVER` clause is the way that SQL specifies that a function is to be computed in a window. It partitions the data by `Plant` (to keep the different power plants' data separate), orders each plant's partition by `Date` (to put the energy measurements next to each other), and uses a `RANGE` frame of three days on either side of each day for the `avg` (to handle any missing days). Here is the result:

Plant	Date	MWh 7-dayMoving Average
Boston	2019-01-02	517450.75

Plant	Date	MWh 7-dayMoving Average
Boston	2019-01-03	508793.20
Boston	2019-01-04	508529.83
Boston	2019-01-05	523459.85
Boston	2019-01-06	526067.14
Boston	2019-01-07	524938.71
Boston	2019-01-08	518294.57
Boston	2019-01-09	520665.42
Boston	2019-01-10	528859.00
Boston	2019-01-11	532466.66
Boston	2019-01-12	516352.00
Boston	2019-01-13	499793.00
Worcester	2019-01-02	104768.25
Worcester	2019-01-03	102713.00
Worcester	2019-01-04	102249.50
Worcester	2019-01-05	104621.57
Worcester	2019-01-06	103856.71
Worcester	2019-01-07	103094.85
Worcester	2019-01-08	101345.14
Worcester	2019-01-09	102313.85
Worcester	2019-01-10	104125.00
Worcester	2019-01-11	104823.83
Worcester	2019-01-12	102017.80
Worcester	2019-01-13	99145.75

You can request multiple different `OVER` clauses in the same `SELECT`, and each will be computed separately. Often, however, you want to use the same window for multiple functions, and you can do this by using a `WINDOW` clause to define a *named* window:

```
SELECT "Plant", "Date",
    avg("MWh") OVER seven AS "MWh 7-day Moving Average"
FROM "Generation History"
WINDOW seven AS (
    PARTITION BY "Plant"
    ORDER BY "Date" ASC
    RANGE BETWEEN INTERVAL 3 DAYS PRECEDING
              AND INTERVAL 3 DAYS FOLLOWING)
ORDER BY 1, 2;
```

This would be useful, for example, if one also wanted the 7-day moving `min` and `max` to show the bounds of the data.

Under the Feathers

That is a long list of complicated functionality! Making it all work relatively quickly has many pieces, so lets have a look at how they all get implemented in DuckDB.

Pipeline Breaking

The first thing to notice is that windowing is a ”pipeline breaker”. That is, the `Window` operator has to read all of its inputs before it can start computing a function. This means that if there is some other way to compute something, it may well be faster to use a different technique.

One common analytic task is to find the last value in some group. For example, suppose we want the last recorded power output for each plant. It is tempting to use the `rank()` window function with a reverse sort for this task:

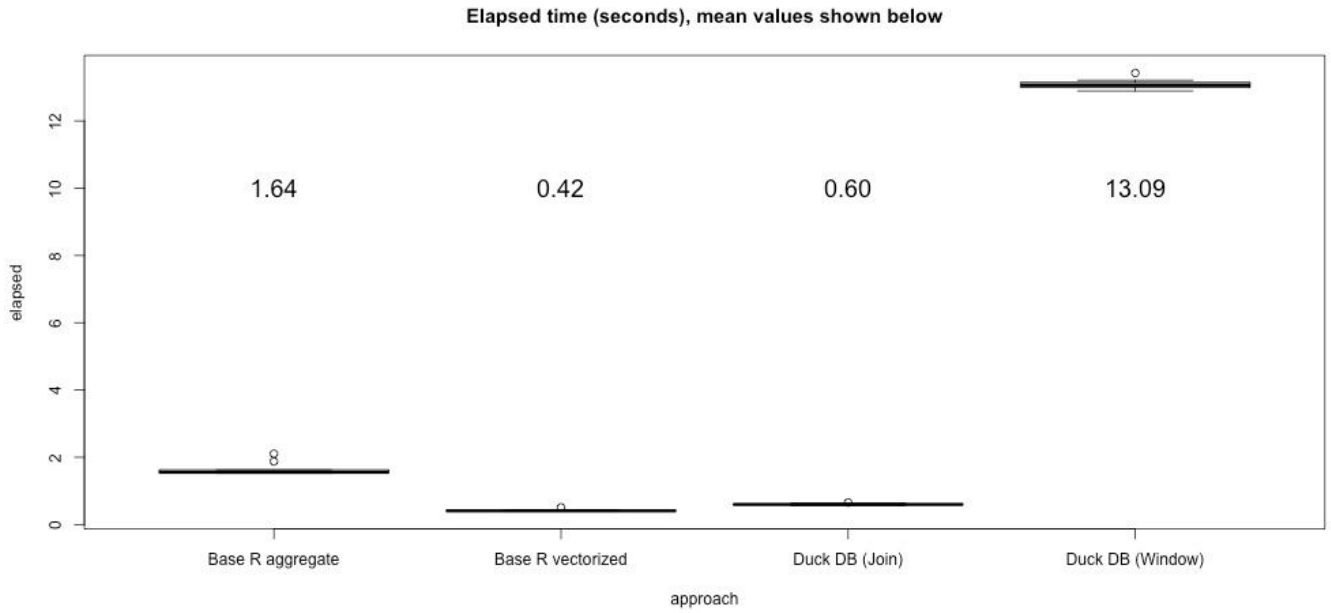
```
SELECT "Plant", "MWh"
FROM (
    SELECT "Plant", "MWh",
        rank() OVER (
            PARTITION BY "Plant"
            ORDER BY "Date" DESC) AS r
    FROM table) t
WHERE r = 1;
```

but this requires materialising the entire table, partitioning it, sorting the partitions, and then pulling out a single row from those partitions. A much faster way to do this is to use a self join to filter the table to contain only the last (`max`) value of the DATE field:

```
SELECT table."Plant", "MWh"
FROM table,
    (SELECT "Plant", max("Date") AS "Date"
     FROM table GROUP BY 1) lasts
WHERE table."Plant" = lasts."Plant"
  AND table."Date" = lasts."Date";
```

This join query requires two scans of the table, but the only materialised data is the filtering table (which is probably much smaller than the original table), and there is no sorting at all.

This type of query showed up in a user's blog and we found that the join query was over 20 times faster on their data set:



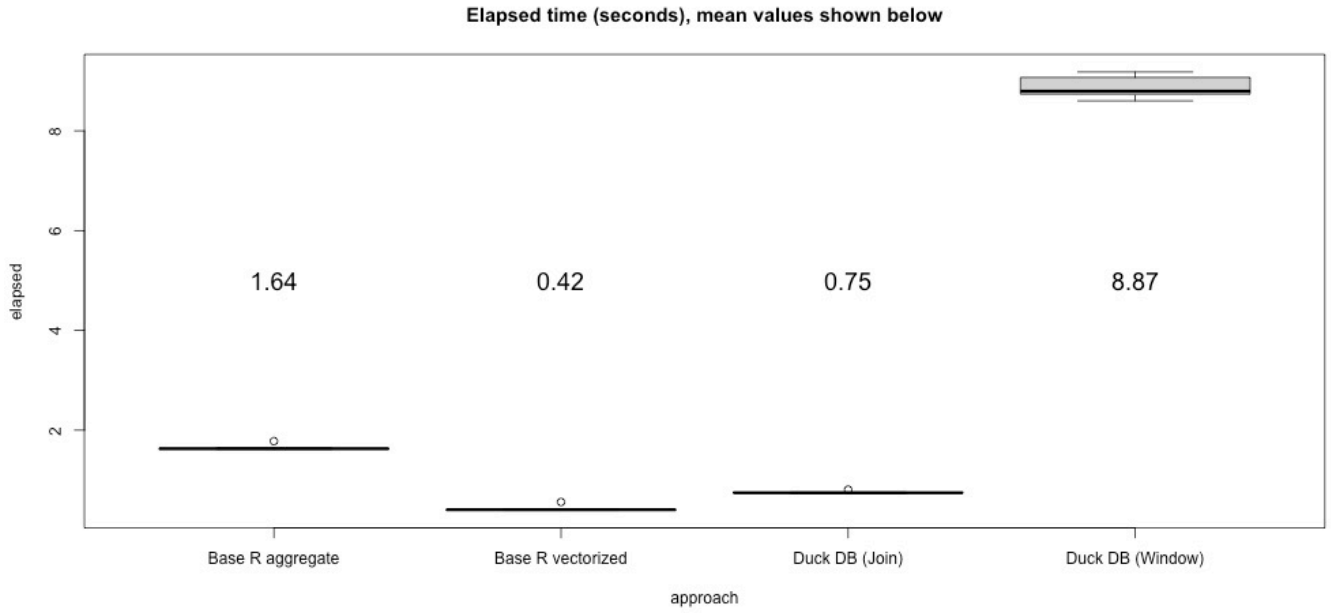
Of course most analytic tasks that use windowing *do* require using the `Window` operator, and DuckDB uses a collection of techniques to make the performance as fast as possible.

Partitioning and Sorting

At one time, windowing was implemented by sorting on both the partition and the ordering fields and then finding the partition boundaries. This is resource intensive, both because the entire relation must be sorted, and because sorting is `O(N log N)` in the size of the relation. Fortunately, there are faster ways to implement this step.

To reduce resource consumption, DuckDB uses the partitioning scheme from Leis et al.'s *Efficient Processing of Window Functions in Analytical SQL Queries* and breaks the partitions up into 1024 chunks using `O(N)` hashing. The chunks still need to be sorted on all the fields because there may be hash collisions, but each partition can now be 1024 times smaller, which reduces the runtime significantly. Moreover, the partitions can easily be extracted and processed in parallel.

Sorting in DuckDB recently got a big performance boost, along with the ability to work on partitions that were larger than memory. This functionality has been also added to the `Window` operator, resulting in a 33% improvement in the last-in-group example:



As a final optimization, even though you can request multiple window functions, DuckDB will collect functions that use the same partitioning and ordering, and share the data layout between those functions.

Aggregation

Most of the general-purpose window functions are straightforward to compute, but windowed aggregate functions can be expensive because they need to look at multiple values for each row. They often need to look at the same value multiple times, or repeatedly look at a large number of values, so over the years several approaches have been taken to improve performance.

Naïve Windowed Aggregation

Before explaining how DuckDB implements windowed aggregation, we need to take a short detour through how ordinary aggregates are implemented. Aggregate "functions" are implemented using three required operations and one optional operation:

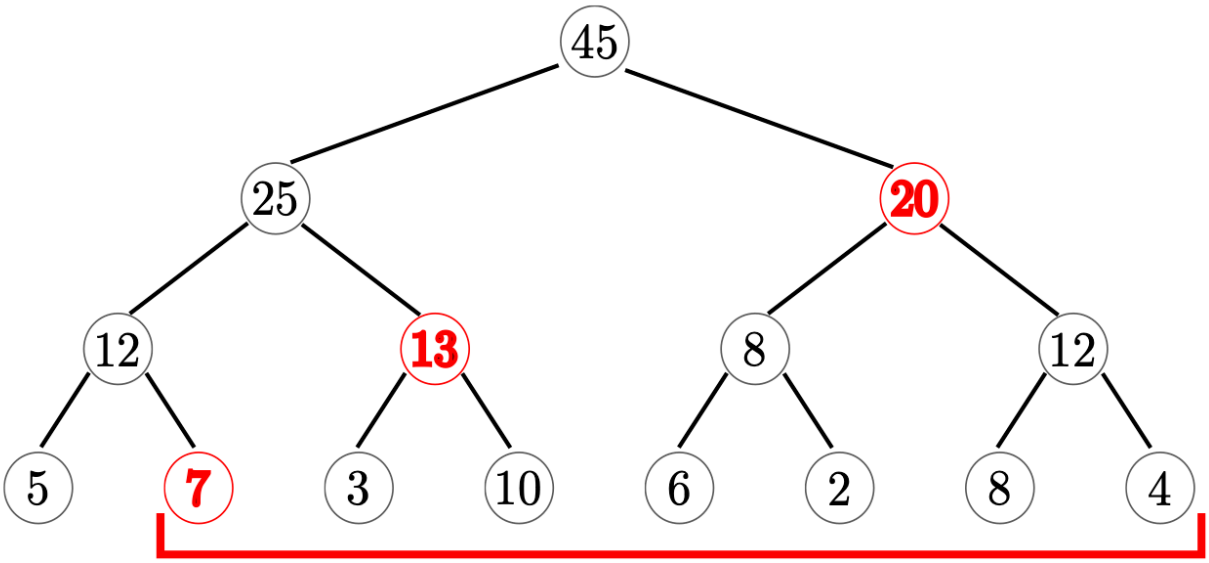
- *Initialize* – Creates a state that will be updated. For `sum`, this is the running total, starting at `NULL` (because a `sum` of zero items is `NULL`, not zero.)
- *Update* – Updates the state with a new value. For `sum`, this adds the value to the state.
- *Finalize* – Produces the final aggregate value from the state. For `sum`, this just copies the running total.
- *Combine* – Combines two states into a single state. Combine is optional, but when present it allows the aggregate to be computed in parallel. For `sum`, this produces a new state with the sum of the two input values.

The simplest way to compute an individual windowed aggregate value is to *initialize* a state, *update* the state with all the values in the window frame, and then use *finalize* to produce the value of the windowed aggregate. This naïve algorithm will always work, but it is quite inefficient. For example, a running total will re-add all the values from the start of the partition for each running total, and this has a run time of `O(N^2)`.

To improve on this, some databases add additional "moving state" operations that can add or remove individual values incrementally. This reduces computation in some common cases, but it can only be used for certain aggregates. For example, it doesn't work for `min`) because you don't know if there are multiple duplicate minima. Moreover, if the frame boundaries move around a lot, it can still degenerate to an `O(N^2)` run time.

Segment Tree Aggregation

Instead of adding more functions, DuckDB uses the *segment tree* approach from Leis et al. above. This works by building a tree on top of the entire partition with the aggregated values at the bottom. Values are combined into states at nodes above them in the tree until there is a single root:



To compute a value, the algorithm generates states for the ragged ends of the frame, *combines* states in the tree above the values in the frame, and *finalizes* the result from the last remaining state. So in the example above (Figure 5 from Leis et al.) only three values need to be added instead of 7. This technique can be used for all *combinable* aggregates.

General Windowed Aggregation

The biggest drawback of segment trees is the need to manage a potentially large number of intermediate states. For the simple states used for standard distributive aggregates like `sum`, this is not a problem because the states are small, the tree keeps the number of states logarithmically low, and the state used to compute each value is also cheap.

For some aggregates, however, the state is not small. Typically these are so-called *holistic* aggregates, where the value depends on all the values of the frame. Examples of such aggregates are `mode` and `quantile`, where each state may have to contain a copy of *all* the values seen so far. While segment trees *can* be used to implement moving versions of any combinable aggregate, this can be quite expensive for large, complex states – and this was not the original goal of the algorithm.

To solve this problem, we use the approach from Wesley and Xu's *Incremental Computation of Common Windowed Holistic Aggregates*, which generalises segment trees to aggregate-specific data structures. The aggregate can define a fifth optional *window* operation, which will be passed the bottom of the tree and the bounds of the current and previous frame. The aggregate can then create an appropriate data structure for its implementation.

For example, the `mode` function maintains a hash table of counts that it can update efficiently, and the `quantile` function maintains a partially sorted list of frame indexes. Moreover, the `quantile` functions can take an array of quantile values, which further increases performance by sharing the partially ordered results among the different quantile values.

Because these aggregates can be used in a windowing context, the moving average example above can be easily modified to produce a moving inter-quartile range:

```
SELECT "Plant", "Date",
    quantile_cont("MWh", [0.25, 0.5, 0.75]) OVER seven
        AS "MWh 7-day Moving IQR"
FROM "Generation History"
WINDOW seven AS (
    PARTITION BY "Plant"
    ORDER BY "Date" ASC
    RANGE BETWEEN INTERVAL 3 DAYS PRECEDING
              AND INTERVAL 3 DAYS FOLLOWING)
ORDER BY 1, 2;
```

Moving quantiles like this are more robust to anomalies, which makes them a valuable tool for data series analysis, but they are not generally implemented in most database systems. There are some approaches that can be used in some query engines, but the lack of a general moving aggregation architecture means that these solutions can be unnatural or complex. DuckDB's implementation uses the standard window notation, which means you don't have to learn new syntax or pull the data out into another tool.

Ordered Set Aggregates

Window functions are often closely associated with some special "ordered set aggregates" defined by the SQL standard. Some databases implement these functions using the `Window` operator, but this is rather inefficient because sorting the data (an `O(N log N)` operation) is not required – it suffices to use Hoare's `O(N)` `FIND` algorithm as used in the STL's `std::nth_element`. DuckDB translates these ordered set aggregates to use the faster `quantile_cont`, `quantile_disc`, and `mode` regular aggregate functions, thereby avoiding using windowing entirely.

Extensions

This architecture also means that any new aggregates we add can benefit from the existing windowing infrastructure. DuckDB is an open source project, and we welcome submissions of useful aggregate functions – or you can create your own domain-specific ones in your own fork. At some point we hope to have a UDF architecture that will allow plug-in aggregates, and the simplicity and power of the interface will let these plugins leverage the notational simplicity and run time performance that the internal functions enjoy.

Conclusion

DuckDB's windowing implementation uses a variety of techniques to speed up what can be the slowest part of an analytic query. It is well integrated with the sorting subsystem and the aggregate function architecture, which makes expressing advanced moving aggregates both natural and efficient.

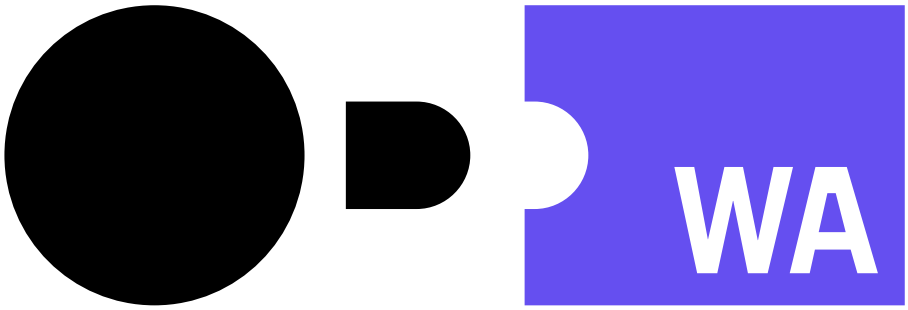
DuckDB is a free and open-source database management system (MIT licensed). It aims to be the SQLite for Analytics, and provides a fast and efficient database system with zero external dependencies. It is available not just for Python, but also for C/C++, R, Java, and more.

DuckDB-Wasm: Efficient Analytical SQL in the Browser

Publication date: 2021-10-29

Authors: André Kohn and Dominik Moritz

TL;DR: DuckDB-Wasm is an in-process analytical SQL database for the browser. It is powered by WebAssembly, speaks Arrow fluently, reads Parquet, CSV and JSON files backed by Filesystem APIs or HTTP requests and has been tested with Chrome, Firefox, Safari and Node.js. You can try it in your browser at shell.duckdb.org or on Observable.



DuckDB-Wasm is fast! If you're here for performance numbers, head over to our benchmarks at shell.duckdb.org/versus.

Efficient Analytics in the Browser

The web browser has evolved to a universal computation platform that even runs in your car. Its rise has been accompanied by increasing requirements for the browser programming language JavaScript. JavaScript was, first and foremost, designed to be very flexible which comes at the cost of a reduced processing efficiency compared to native languages like C++. This becomes particularly apparent when considering the execution times of more complex data analysis tasks that often fall behind the native execution by orders of magnitude. In the past, such analysis tasks have therefore been pushed to servers that tie any client-side processing to additional round-trips over the internet and introduce their own set of scalability problems.

The processing capabilities of browsers were boosted tremendously 4 years ago with the introduction of WebAssembly:

> WebAssembly (abbreviated Wasm) is a binary instruction format for a stack-based virtual machine. Wasm is designed as a portable compilation target for programming languages, enabling deployment on the web for client and server applications.
>
> The Wasm stack machine is designed to be encoded in a size- and load-time efficient binary format. WebAssembly aims to execute at native speed by taking advantage of common hardware capabilities available on a wide range of platforms.
>
> (ref: https://webassembly.org/)

Four years later, the WebAssembly revolution is in full progress with first implementations being shipped in four major browsers. It has already brought us game engines, entire IDEs and even a browser version of Photoshop. Today, we join the ranks with a first release of the npm library @duckdb/duckdb-wasm.

As an in-process analytical database, DuckDB has the rare opportunity to siginificantly speed up OLAP workloads in the browser. We believe that there is a need for a comprehensive and self-contained data analysis library. DuckDB-wasm automatically offloads your queries to dedicated worker threads and reads Parquet, CSV and JSON files from either your local filesystem or HTTP servers driven by plain SQL input. In this blog post, we want to introduce the library and present challenges on our journey towards a browser-native OLAP database.

DuckDB-Wasm is not yet stable. You will find rough edges and bugs in this release. Please share your thoughts with us on GitHub.

How to Get Data In?

Let's dive into examples. DuckDB-Wasm provides a variety of ways to load your data. First, raw SQL value clauses like `INSERT INTO sometable VALUES (1, 'foo'), (2, 'bar')` are easy to formulate and only depend on plain SQL text. Alternatively, SQL statements like `CREATE TABLE foo AS SELECT * FROM 'somefile.parquet'` consult our integrated web filesystem to resolve `somefile.parquet` locally, remotely or from a buffer. The methods `insertCSVFromPath` and `insertJSONFromPath` further provide convenient ways to import CSV and JSON files using additional typed settings like column types. And finally, the method `insertArrowFromIPCStream` (optionally through `insertArrowTable`, `insertArrowBatches` or `insertArrowVectors`) copies raw IPC stream bytes directly into a WebAssembly stream decoder.

The following example presents different options how data can be imported into DuckDB-Wasm:

```
// Data can be inserted from an existing arrow.Table
await c.insertArrowTable(existingTable, { name: "arrow_table" });
// ..., from Arrow vectors
await c.insertArrowVectors({
    col1: arrow.Int32Vector.from([1, 2]),
    col2: arrow.Utf8Vector.from(["foo", "bar"]),
}, {
    name: "arrow_vectors"
});
// ..., from a raw Arrow IPC stream
const c = await db.connect();
const streamResponse = await fetch(` someapi`);
const streamReader = streamResponse.body.getReader();
const streamInserts = [];
while (true) {
    const { value, done } = await streamReader.read();
    if (done) break;
    streamInserts.push(c.insertArrowFromIPCStream(value, { name: "streamed" }));
}
await Promise.all(streamInserts);

// ..., from CSV files
// (interchangeable: registerFile{Text,Buffer,URL,Handle})
await db.registerFileText(` data.csv`, "1|foo\n2|bar\n");
// ... with typed insert options
await db.importCSVFromPath('data.csv', {
    schema: 'main',
    name: 'foo',
    detect: false,
    header: false,
    delimiter: '|',
    columns: {
        col1: new arrow.Int32(),
        col2: new arrow.Utf8(),
    }
});

// ..., from JSON documents in row-major format
await db.registerFileText("rows.json", `[
    { "col1": 1, "col2": "foo" },
    { "col1": 2, "col2": "bar" },
]`);
// ... or column-major format
await db.registerFileText("columns.json", `{
    "col1": [1, 2],
    "col2": ["foo", "bar"]
}`);
```

```
// ... with typed insert options
await db.importJSONFromPath('rows.json', { name: 'rows' });
await db.importJSONFromPath('columns.json', { name: 'columns' });

// ..., from Parquet files
const pickedFile: File = letUserPickFile();
await db.registerFileHandle("local.parquet", pickedFile);
await db.registerFileURL("remote.parquet", "https://origin/remote.parquet");

// ..., by specifying URLs in the SQL text
await c.query(`
    CREATE TABLE direct AS
        SELECT * FROM 'https://origin/remote.parquet'
`);
// ..., or by executing raw insert statements
await c.query(` INSERT INTO existing_table
    VALUES (1, "foo"), (2, "bar")`);
```

How to Get Data Out?

Now that we have the data loaded, DuckDB-Wasm can run queries on two different ways that differ in the result materialization. First, the method `query` runs a query to completion and returns the results as single `arrow.Table`. Second, the method `send` fetches query results lazily through an `arrow.RecordBatchStreamReader`. Both methods are generic and allow for typed results in Typescript:

```
// Either materialize the query result
await conn.query<{ v: arrow.Int32 }>(`
    SELECT * FROM generate_series(1, 100) t(v)
`);
// ..., or fetch the result chunks lazily
for await (const batch of await conn.send<{ v: arrow.Int32 }>(`
    SELECT * FROM generate_series(1, 100) t(v)
`)) {
    // ...
}
```

Alternatively, you can prepare statements for parameterized queries using:

```
// Prepare query
const stmt = await conn.prepare<{ v: arrow.Int32 }>(
    `SELECT (v + ?) AS v FROM generate_series(0, 10000) t(v);`
);
// ... and run the query with materialized results
await stmt.query(234);
// ... or result chunks
for await (const batch of await stmt.send(234)) {
    // ...
}
```

Looks like Arrow to Me

DuckDB-Wasm uses Arrow as data protocol for the data import and all query results. Arrow is a database-friendly columnar format that is organized in chunks of column vectors, called record batches and that support zero-copy reads with only a small overhead. The npm library `apache-arrow` implements the Arrow format in the browser and is already used by other data processing frameworks, like Arquero. Arrow therefore not only spares us the implementation of the SQL type logic in JavaScript, it also makes us compatible to existing tools.

Why not use plain Javascript objects?

WebAssembly is isolated and memory-safe. This isolation is part of it's DNA and drives fundamental design decisions in DuckDB-Wasm. For example, WebAssembly introduces a barrier towards the traditional JavaScript heap. Crossing this barrier is difficult as JavaScript has to deal with native function calls, memory ownership and serialization performance. Languages like C++ make this worse as they rely on smart pointers that are not available through the FFI. They leave us with the choice to either pass memory ownership to static singletons within the WebAssembly instance or maintain the memory through C-style APIs in JavaScript, a language that is too dynamic for sound implementations of the RAII idiom. The memory-isolation forces us to serialize data before we can pass it to the WebAssembly instance. Browsers can serialize JavaScript objects natively to and from JSON using the functions `JSON.stringify` and `JSON.parse` but this is slower compared to, for example, copying raw native arrays.

Web Filesystem

DuckDB-Wasm integrates a dedicated filesystem for WebAssembly. DuckDB itself is built on top of a virtual filesystem that decouples higher level tasks, such as reading a Parquet file, from low-level filesystem APIs that are specific to the operating system. We leverage this abstraction in DuckDB-Wasm to tailor filesystem implementations to the different WebAssembly environments.

The following figure shows our current web filesystem in action. The sequence diagram presents a user running a SQL query that scans a single Parquet file. The query is first offloaded to a dedicated web worker through a JavaScript API. There, it is passed to the WebAssembly module that processes the query until the execution hits the `parquet_scan` table function. This table function then reads the file using a buffered filesystem which, in turn, issues paged reads on the web filesystem. This web filesystem then uses an environment-specific runtime to read the file from several possible locations.

Depending on the context, the Parquet file may either reside on the local device, on a remote server or in a buffer that was registered by the user upfront. We deliberately treat all three cases equally to unify the retrieval and processing of external data. This does not only simplify the analysis, it also enables more advanced features like partially consuming structured file formats. Parquet files, for example, consist of multiple row groups that store data in a column-major fashion. As a result, we may not need to download the entire file for a query but only required bytes.

A query like `SELECT count(*) FROM parquet_scan(...)`, for example, can be evaluated on the file metadata alone and will finish in milliseconds even on remote files that are several terabytes large. Another more general example are paging scans with `LIMIT` and `OFFSET` qualifiers such as `SELECT * FROM parquet_scan(...) LIMIT 20 OFFSET 40`, or queries with selective filter predicates where entire row groups can be skipped based on metadata statistics. These partial file reads are no groundbreaking novelty and could be implemented in JavaScript today, but with DuckDB-Wasm, these optimizations are now driven by the semantics of SQL queries instead of fine-tuned application logic.

Note: The common denominator among the available File APIs is unfortunately not large. This limits the features that we can provide in the browser. For example, local persistency of DuckDB databases would be a feature with significant impact but requires a way to either read and write synchronously into user-provided files or IndexedDB. We might be able to bypass these limitations in the future but this is subject of ongoing research.

Advanced Features

WebAssembly 1.0 has landed in all major browsers. The WebAssembly Community Group fixed the design of this first version back in November 2017, which is now referred to as WebAssembly MVP. Since then, the development has been ongoing with eight additional features that have been added to the standard and at least five proposals that are currently in progress.

The rapid pace of this development presents challenges and opportunities for library authors. On the one hand, the different features find their way into the browsers at different speeds which leads to a fractured space of post-MVP functionality. On the other hand, features can bring flat performance improvements and are therefore indispensable when aiming for a maximum performance.

The most promising feature for DuckDB-Wasm is exception handling which is already enabled by default in Chrome 95. DuckDB and DuckDB-Wasm are written in C++ and use exceptions for faulty situations. DuckDB does not use exceptions for general control flow but to automatically propagate errors upwards to the top-level plan driver. In native environments, these exceptions are implemented as ”zero-cost exceptions” as they induce no overhead until they are thrown. With the WebAssembly MVP, however, that is no longer possible as the compiler toolchain Emscripten has to emulate exceptions through JavaScript. Without WebAssembly exceptions, DuckDB-Wasm calls throwing

functions through a JavaScript hook that can catch exceptions emulated through JavaScript `aborts`. An example for these hook calls is shown in the following figure. Both stack traces originate from a single paged read of a Parquet file in DuckDB-Wasm. The left side shows a stack trace with the WebAssembly MVP and requires multiple calls through the functions `wasm-to-js-i*`. The right stack trace uses WebAssembly exceptions without any hook calls.

This fractured feature space is a temporary challenge that will be resolved once high-impact features like exception handling, SIMD and bulk-memory operations are available everywhere. In the meantime, we will ship multiple WebAssembly modules that are compiled for different feature sets and adaptively pick the best bundle for you using dynamic browser checks.

The following example shows how the asynchronous version of DuckDB-Wasm can be instantiated using either manual or JsDelivr bundles:

```
// Import the ESM bundle (supports tree-shaking)
import * as duckdb from '@duckdb/duckdb-wasm/dist/duckdb-esm.js';

// Either bundle them manually, for example as Webpack assets
import duckdb_wasm from '@duckdb/duckdb-wasm/dist/duckdb.wasm';
import duckdb_wasm_next from '@duckdb/duckdb-wasm/dist/duckdb-next.wasm';
import duckdb_wasm_next_coi from '@duckdb/duckdb-wasm/dist/duckdb-next-coi.wasm';
const WEBPACK_BUNDLES: duckdb.DuckDBBundles = {
    asyncDefault: {
        mainModule: duckdb_wasm,
        mainWorker: new URL('@duckdb/duckdb-wasm/dist/duckdb-browser-async.worker.js',
import.meta.url).toString(),
    },
    asyncNext: {
        mainModule: duckdb_wasm_next,
        mainWorker: new URL('@duckdb/duckdb-wasm/dist/duckdb-browser-async-next.worker.js',
import.meta.url).toString(),
    },
    asyncNextCOI: {
        mainModule: duckdb_wasm_next_coi,
        mainWorker: new URL(
            '@duckdb/duckdb-wasm/dist/duckdb-browser-async-next-coi.worker.js',
            import.meta.url,
        ).toString(),
        pthreadWorker: new URL(
            '@duckdb/duckdb-wasm/dist/duckdb-browser-async-next-coi.pthread.worker.js',
            import.meta.url,
        ).toString(),
    },
};
// ..., or load the bundles from jsdelivr
const JSDELIVR_BUNDLES = duckdb.getJsDelivrBundles();

// Select a bundle based on browser checks
const bundle = await duckdb.selectBundle(JSDELIVR_BUNDLES);
// Instantiate the asynchronus version of DuckDB-Wasm
const worker = new Worker(bundle.mainWorker!);
const logger = new duckdb.ConsoleLogger();
const db = new duckdb.AsyncDuckDB(logger, worker);
await db.instantiate(bundle.mainModule, bundle.pthreadWorker);
```

You can also test the features and selected bundle in your browser using the web shell command `.features`.

Multithreading

In 2018, the Spectre and Meltdown vulnerabilities sent crippling shockwaves through the internet. Today, we are facing the repercussions of these events, in particular in software that runs arbitrary user code – such as web browsers. Shortly after the publications, all major browser

vendors restricted the use of `SharedArrayBuffers` to prevent dangerous timing attacks. `SharedArrayBuffers` are raw buffers that can be shared among web workers for global state and an alternative to the browser-specific message passing. These restrictions had detrimental effects on WebAssembly modules since `SharedArrayBuffers` are neccessary for the implementation of POSIX threads in WebAssembly.

Without `SharedArrayBuffers`, WebAssembly modules can run in a dedicated web worker to unblock the main event loop but won't be able to spawn additional workers for parallel computations within the same instance. By default, we therefore cannot unleash the parallel query execution of DuckDB in the web. However, browser vendors have recently started to reenable `SharedArrayBuffers` for websites that are cross-origin-isolated. A website is cross-origin-isolated if it ships the main document with the following HTTP headers:

```
Cross-Origin-Embedder-Policy: require-corp
Cross-Origin-Opener-Policy: same-origin
```

These headers will instruct browsers to A) isolate the top-level document from other top-level documents outside its own origin and B) prevent the document from making arbitrary cross-origin requests unless the requested resource explicitly opts in. Both restrictions have far reaching implications for a website since many third-party data sources won't yet provide the headers today and the top-level isolation currently hinders the communication with, for example, OAuth pop up's (there are plans to lift that).

We therefore assume that DuckDB-Wasm will find the majority of users on non-isolated websites. We are, however, experimenting with dedicated bundles for isolated sites using the suffix `-next-coi`) and will closely monitor the future need of our users.

Web Shell

We further host a web shell powered by DuckDB-Wasm alongside the library release at shell.duckdb.org. Use the following shell commands to query remote TPC-H files at scale factor 0.01. When querying your own, make sure to properly set CORS headers since your browser will otherwise block these requests. You can alternatively use the `.files` command to register files from the local filesystem.

```
.timer on

SELECT count(*)
FROM 'https://blobs.duckdb.org/data/tpch-sf0.01-parquet/lineitem.parquet';

SELECT count(*)
FROM 'https://blobs.duckdb.org/data/tpch-sf0.01-parquet/customer.parquet';

SELECT avg(c_acctbal)
FROM 'https://blobs.duckdb.org/data/tpch-sf0.01-parquet/customer.parquet';

SELECT *
FROM 'https://blobs.duckdb.org/data/tpch-sf0.01-parquet/orders.parquet'
LIMIT 10;

SELECT n_name, avg(c_acctbal)
FROM
    'https://blobs.duckdb.org/data/tpch-sf0.01-parquet/customer.parquet',
    'https://blobs.duckdb.org/data/tpch-sf0.01-parquet/nation.parquet'
WHERE c_nationkey = n_nationkey
GROUP BY n_name;

SELECT *
FROM
    'https://blobs.duckdb.org/data/tpch-sf0.01-parquet/region.parquet',
    'https://blobs.duckdb.org/data/tpch-sf0.01-parquet/nation.parquet'
WHERE r_regionkey = n_regionkey;
```

Evaluation

The following table teases the execution times of some TPC-H queries at scale factor 0.5 using the libraries DuckDB-Wasm, sql.js, Arquero and Lovefield. You can find a more in-depth discussion with all TPC-H queries, additional scale factors and microbenchmarks on the "DuckDB-Wasm versus X" page.

Query	DuckDB-Wasm	sql.js	Arquero	Lovefield
1	**0.855 s**	8.441 s	24.031 s	12.666 s
3	**0.179 s**	1.758 s	16.848 s	3.587 s
4	**0.151 s**	0.384 s	6.519 s	3.779 s
5	**0.197 s**	1.965 s	18.286 s	13.117 s
6	**0.086 s**	1.294 s	1.379 s	5.253 s
7	**0.319 s**	2.677 s	6.013 s	74.926 s
8	**0.236 s**	4.126 s	2.589 s	18.983 s
10	**0.351 s**	1.238 s	23.096 s	18.229 s
12	**0.276 s**	1.080 s	11.932 s	10.372 s
13	**0.194 s**	5.887 s	16.387 s	9.795 s
14	**0.086 s**	1.194 s	6.332 s	6.449 s
16	**0.137 s**	0.453 s	0.294 s	5.590 s
19	**0.377 s**	1.272 s	65.403 s	9.977 s

Future Research

We believe that WebAssembly unveils hitherto dormant potential for shared query processing between clients and servers. Pushing computation closer to the client can eliminate costly round-trips to the server and thus increase interactivity and scalability of in-browser analytics. We further believe that the release of DuckDB-Wasm could be the first step towards a more universal data plane spanning across multiple layers including traditional database servers, clients, CDN workers and computational storage. As an in-process analytical database, DuckDB might be the ideal driver for distributed query plans that increase the scalability and interactivity of SQL databases at low costs.

Fast Moving Holistic Aggregates

Publication date: 2021-11-12

Author: Richard Wesley

TL;DR: DuckDB, a free and open-source analytical data management system, has a windowing API that can compute complex moving aggregates like interquartile ranges and median absolute deviation much faster than the conventional approaches.

In a previous post, we described the DuckDB windowing architecture and mentioned the support for some advanced moving aggregates. In this post, we will compare the performance various possible moving implementations of these functions and explain how DuckDB's performant implementations work.

What Is an Aggregate Function?

When people think of aggregate functions, they typically have something simple in mind such as `SUM` or `AVG`. But more generally, what an aggregate function does is *summarise* a set of values into a single value. Such summaries can be arbitrarily complex, and involve any data type. For example, DuckDB provides aggregates for concatenating strings (`STRING_AGG`) and constructing lists (`LIST`). In SQL, aggregated sets come from either a `GROUP BY` clause or an `OVER` windowing specification.

Holistic Aggregates

All of the basic SQL aggregate functions like `SUM` and `MAX` can be computed by reading values one at a time and throwing them away. But there are some functions that potentially need to keep track of all the values before they can produce a result. These are called *holistic* aggregates, and they require more care when implementing.

For some aggregates (like `STRING_AGG`) the order of the values can change the result. This is not a problem for windowing because `OVER` clauses can specify an ordering, but in a `GROUP BY` clause, the values are unordered. To handle this, order sensitive aggregates can include a `WITHIN GROUP(ORDER BY <expr>)` clause to specify the order of the values. Because the values must all be collected and sorted, aggregates that use the `WITHIN GROUP` clause are holistic.

Statistical Holistic Aggregates

Because sorting the arguments to a windowed aggregate can be specified with the `OVER` clause, you might wonder if there are any other kinds of holistic aggregates that do not use sorting, or which use an ordering different from the one in the `OVER` clause. It turns out that there are a number of important statistical functions that turn into holistic aggregates in SQL. In particular, here are the statistical holistic aggregates that DuckDB currently supports:

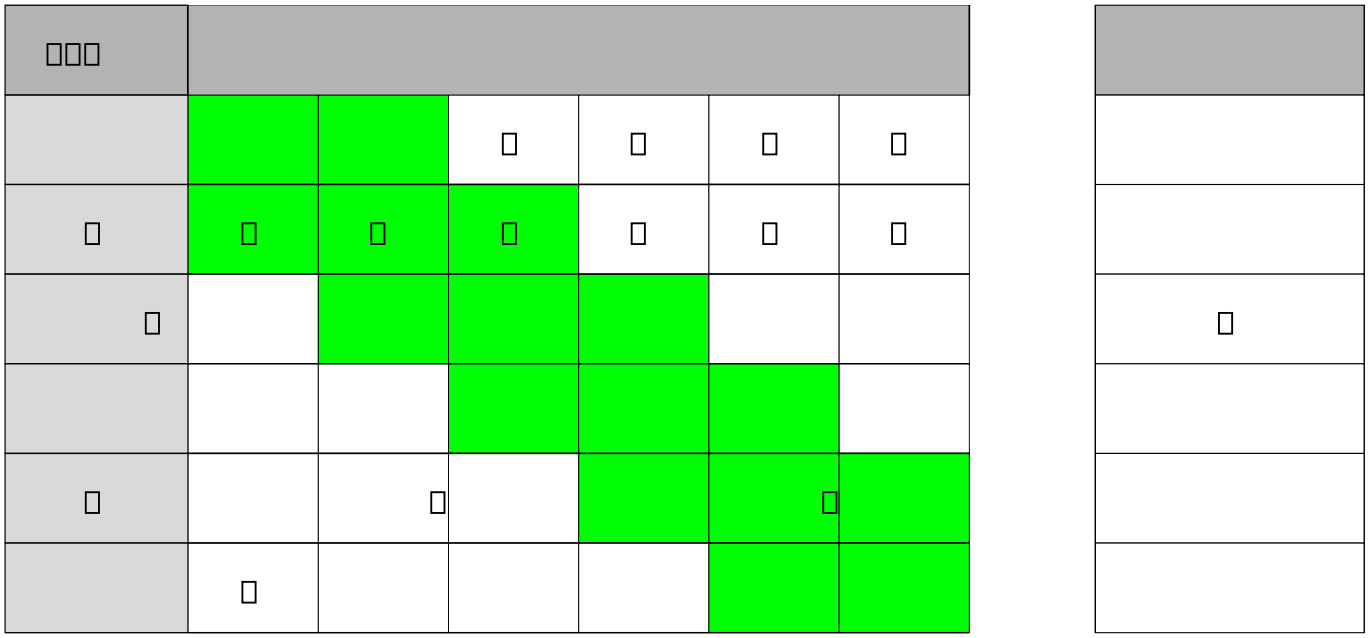
Function	Description
`mode(x)`	The most common value in a set
`median(x)`	The middle value of a set
`quantile_disc(x, <frac>)`	The exact value corresponding to a fractional position.
`quantile_cont(x, <frac>)`	The interpolated value corresponding to a fractional position.

Function	Description
`quantile_disc(x, [<frac>...])`	A list of the exact values corresponding to a list of fractional positions.
`quantile_cont(x, [<frac>...])`	A list of the interpolated value corresponding to a list of fractional positions.
`mad(x)`	The median of the absolute values of the differences of each value from the median.

Where things get really interesting is when we try to compute moving versions of these aggregates. For example, computing a moving AVG is fairly straightforward: You can subtract values that have left the frame and add in the new ones, or use the segment tree approach from the previous post on windowing.

Python Example

Computing a moving median is not as easy. Let's look at a simple example of how we might implement moving `median` in Python for the following string data, using a frame that includes one element from each side:



For this example we are using strings so we don't have to worry about interpolating values.

```
data = ('a', 'b', 'c', 'd', 'c', 'b',)
w = len(data)
for row in range(w):
    l = max(row - 1, 0)       # First index of the frame
    r = min(row + 1, w-1)     # Last index of the frame
    frame = list(data[l:r+1]) # Copy the frame values
    frame.sort()              # Sort the frame values
    n = (r - l) // 2          # Middle index of the frame
    median = frame[n]         # The median is the middle value
    print(row, data[row], median)
```

Each frame has a different set of values to aggregate and we can't change the order in the table, so we have to copy them each time before we sort. Sorting is slow, and there is a lot of repetition.

All of these holistic aggregates have similar problems if we just reuse the simple implementations for moving versions. Fortunately, there are much faster approaches for all of them.

Moving Holistic Aggregation

In the previous post on windowing, we explained the component operations used to implement a generic aggregate function (initialize, update, finalize, combine and window). In the rest of this post, we will dig into how they can be implemented for these complex aggregates.

Quantile

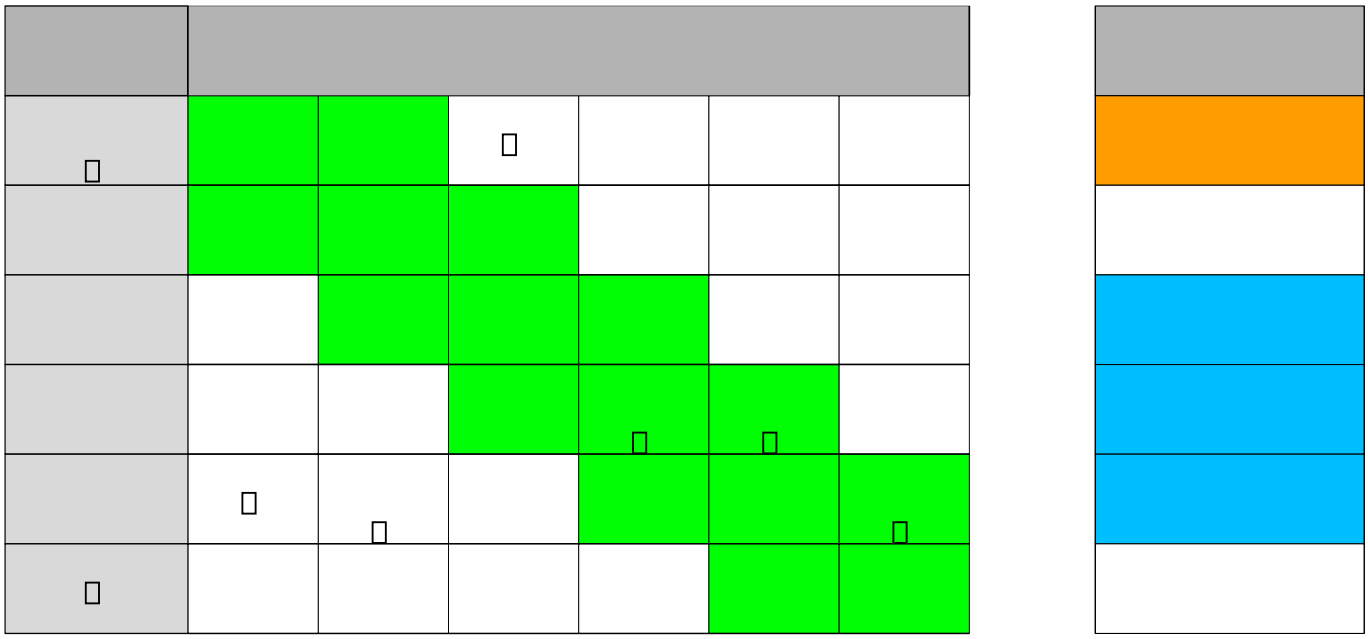
The `quantile` aggregate variants all extract the value(s) at a given fraction (or fractions) of the way through the ordered list of values in the set. The simplest variant is the `median` function, which we met in the introduction, which uses a fraction of `0.5`. There are other variants depending on whether the values are quantitative (i.e., they have a distance and the values can be interpolated) or merely ordinal (i.e., they can be ordered, but ties have to be broken.) Still other variants depend on whether the fraction is a single value or a list of values, but they can all be implemented in similar ways.

A common way to implement `quantile` that we saw in the Python example is to collect all the values into the state, sort them, and then read out the values at the requested positions. (This is probably why the SQL standard refers to it as an "ordered-set aggregate".) States can be combined by concatenation, which lets us group in parallel and build segment trees for windowing.

This approach is very time-consuming because sorting is `O(N log N)`, but happily for `quantile` we can use a related algorithm called `QuickSelect`, which can find a positional value in only `O(N)` time by *partially sorting* the array. You may have run into this algorithm if you have ever used the `std::nth_element` algorithm in the C++ standard library. This works well for grouped quantiles, but for moving quantiles the segment tree approach ends up being about 5% slower than just starting from scratch for each value.

To really improve the performance of moving quantiles, we note that the partial order probably does not change much between frames. If we maintain a list of indirect indicies into the window and call `nth_element` on the indicies, we can reorder the partially ordered indicies instead of the values themselves. In the common case where the frame has the same size, we can even check to see whether the new value disrupts the partial ordering at all, and skip the reordering! With this approach, we can obtain a significant performance boost of 1.5-10 times.

In this example, we have a 3-element frame (green) that moves one space to the right for each value:



The median values in orange must be computed from scratch. Notice that in the example, this only happens at the start of the window. The median values in white are computed using the existing partial ordering. In the example, this happens when the frame changes size. Finally, the median values in blue do not require reordering because the new value is the same as the old value. With this algorithm, we can create a faster implementation of single-fraction `quantile` without sorting.

InterQuartile Ranges (IQR)

We can extend this implementation to *lists* of fractions by leveraging the fact that each call to `nth_element` partially orders the values, which further improves performance. The ”reuse” trick can be generalised to distinguish between fractions that are undisturbed and ones that need to be recomputed.

A common application of multiple fractions is computing interquartile ranges by using the fraction list `[0.25, 0.5, 0.75]`. This is the fraction list we use for the multiple fraction benchmarks. Combined with moving `MIN` and `MAX`, this moving aggregate can be used to generate the data for a moving box-and-whisker plot.

Median Absolute Deviation (MAD)

Maintaining the partial ordering can also be used to boost the performance of the median absolute deviation (or `mad`) aggregate. Unfortunately, the second partial ordering can't use the single value trick because the ”function” being used to partially order the values will have changed if the data median changes. Still, the values are still probably not far off, which again improves the performance of `nth_element`.

Mode

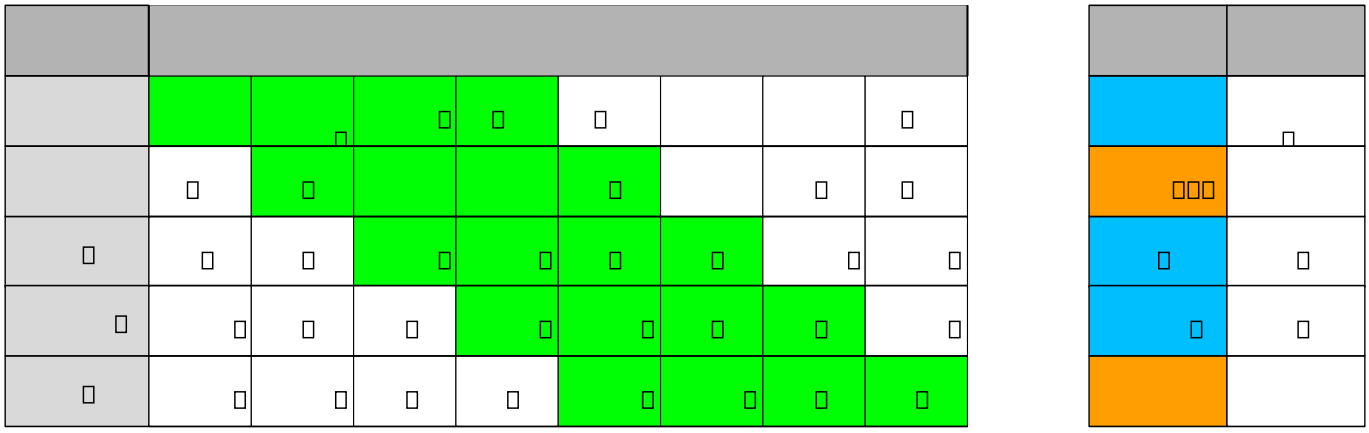
The `mode` aggregate returns the most common value in a set. One common way to implement it is to accumulate all the values in the state, sort them and then scan for the longest run. These states can be combined by merging, which lets us compute the mode in parallel and build segment trees for windowing.

Once again, this approach is very time-consuming because sorting is `O(N log N)`. It may also use more memory than necessary because it keeps *all* the values instead of keeping only the unique values. If there are heavy-hitters in the list, (which is typically what `mode` is being used to find) this can be significant.

Another way to implement `mode` is to use a hash map for the state that maps values to counts. Hash tables are typically `O(N)` for accumulation, which is an improvement on sorting, and they only need to store unique values. If the state also tracks the largest value and count seen so far, we can just return that value when we finalize the aggregate. States can be combined by merging, which lets us group in parallel and build segment trees for windowing.

Unfortunately, as the benchmarks below demonstrate, this segment tree approach for windowing is quite slow! The overhead of merging the hash tables for the segment trees turns out to be about 5% slower than just building a new hash table for each row in the window. But for a moving `mode` computation, we can instead make a single hash table and update it every time the frame moves, removing the old values, adding the new values, and updating the value/count pair. At times the current mode value may have its count decremented, but when that happens we can rescan the table to find the new mode.

In this example, the 4-element frame (green) moves one space to the right for each value:



When the mode is unchanged (blue) it can be used directly. When the mode becomes ambiguous (orange), we must recan the table. This approach is much faster, and in the benchmarks it comes in between 15 and 55 times faster than the other two.

Microbenchmarks

To benchmark the various implementations, we run moving window queries against a 10M table of integers:

```
CREATE TABLE rank100 AS
    SELECT b % 100 AS a, b FROM range(10000000) tbl(b);
```

The results are then re-aggregated down to one row to remove the impact of streaming the results. The frames are 100 elements wide, and the test is repeated with a fixed trailing frame:

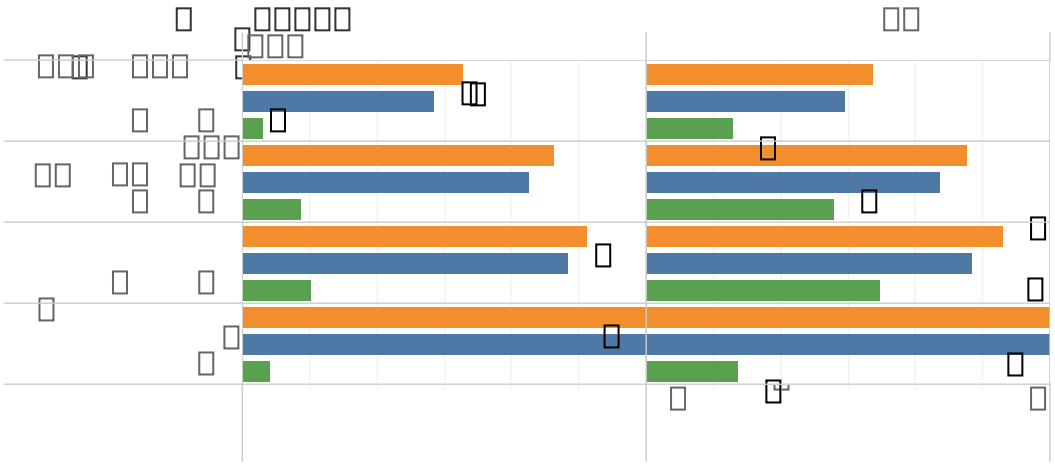
```
SELECT quantile_cont(a, [0.25, 0.5, 0.75]) OVER (
    ORDER BY b ASC
    ROWS BETWEEN 100 PRECEDING AND CURRENT ROW) AS iqr
FROM rank100;
```

and a variable frame that moves pseudo-randomly around the current value:

```
SELECT quantile_cont(a, [0.25, 0.5, 0.75]) OVER (
    ORDER BY b ASC
    ROWS BETWEEN mod(b * 47, 521) PRECEDING AND 100 - mod(b * 47, 521) FOLLOWING) AS iqr
FROM rank100;
```

The two examples here are the interquartile range queries; the other queries use the single argument aggregates `median`, `mad` and `mode`.

As a final step, we ran the same query with `count(*)`, which has the same overhead as the other benchmarks, but is trivial to compute (it just returns the frame size). That overhead was subtracted from the run times to give the algorithm timings:



As can be seen, there is a substantial benefit from implementing the window operation for all of these aggregates, often on the order of a factor of ten.

An unexpected finding was that the segment tree approach for these complex states is always slower (by about 5%) than simply creating the state for each output row. This suggests that when writing combinable complex aggregates, it is well worth benchmarking the aggregate and then considering providing a window operation instead of deferring to the segment tree machinery.

Conclusion

DuckDB's aggregate API enables aggregate functions to define a windowing operation that can significantly improve the performance of moving window computations for complex aggregates. This functionality has been used to significantly speed up windowing for several statistical aggregates, such as mode, interquartile ranges and median absolute deviation.

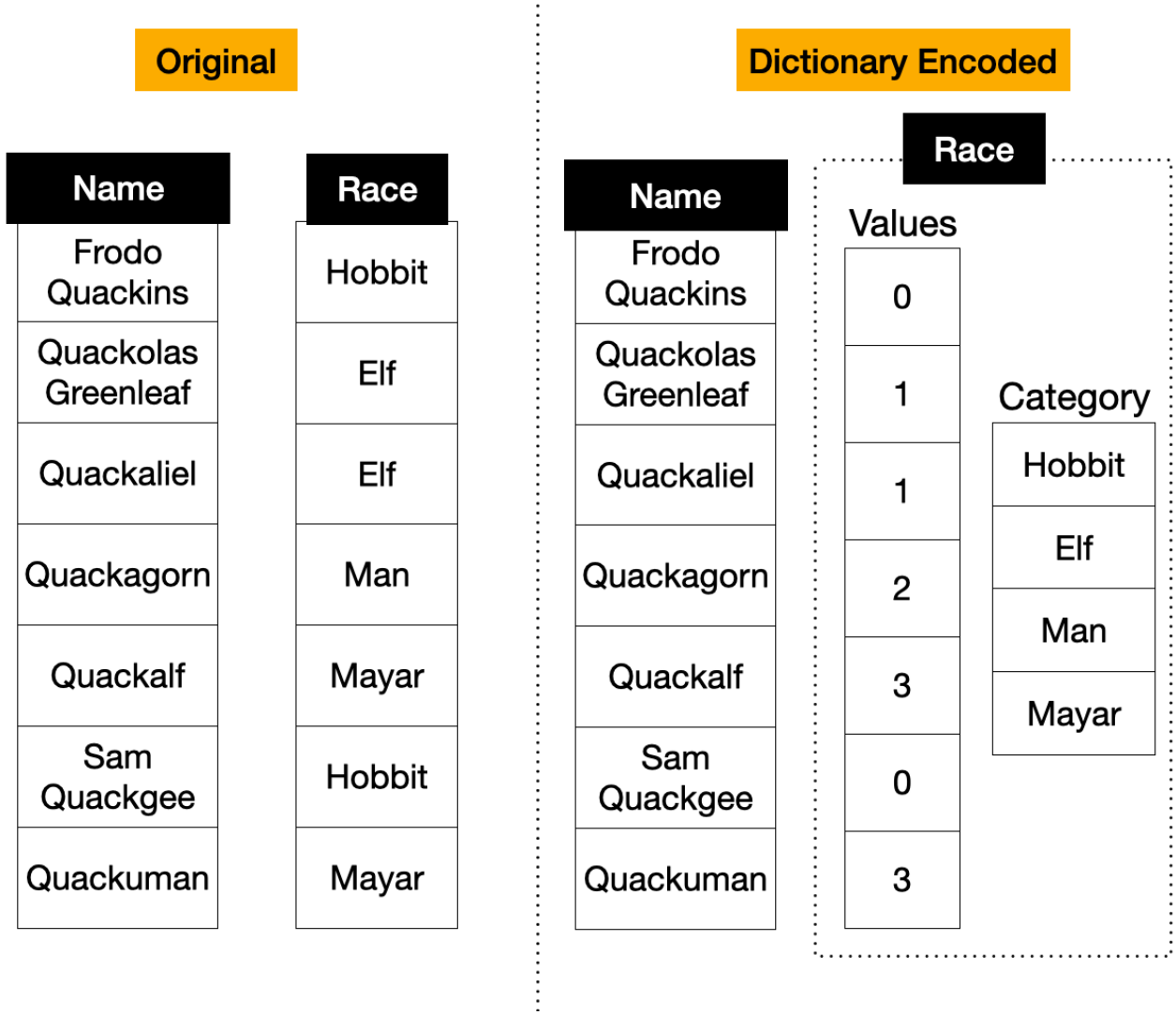
DuckDB is a free and open-source database management system (MIT licensed). It aims to be the SQLite for Analytics, and provides a fast and efficient database system with zero external dependencies. It is available not just for Python, but also for C/C++, R, Java, and more.

DuckDB – Lord of the Enums: The Fellowship of the Categorical and Factors

Publication date: 2021-11-26

Author: Pedro Holanda

String types are one of the most commonly used types. However, often string columns have a limited number of distinct values. For example, a country column will never have more than a few hundred unique entries. Storing a data type as a plain string causes a waste of storage and compromises query performance. A better solution is to dictionary encode these columns. In dictionary encoding, the data is split into two parts: the category and the values. The category stores the actual strings, and the values stores a reference to the strings. This encoding is depicted below.



In the old times, users would manually perform dictionary encoding by creating lookup tables and translating their ids back with join operations. Environments like Pandas and R support these types more elegantly. Pandas Categorical and R Factors are types that allow for columns of strings with many duplicate entries to be efficiently stored through dictionary encoding.

Dictionary encoding not only allows immense storage savings but also allows systems to operate on numbers instead of on strings, drastically boosting query performance. By lowering RAM usage, ENUMs also allow DuckDB to scale to significantly larger datasets.

To allow DuckDB to fully integrate with these encoded structures, we implemented Enum Types. This blog post will show code snippets of how to use ENUM types from both SQL API and Python/R clients, and will demonstrate the performance benefits of the enum types over using regular strings. To the best of our knowledge, DuckDB is the first RDBMS that natively integrates with Pandas categorical columns and R factors.

SQL

Our Enum SQL syntax is heavily inspired by Postgres. Below, we depict how to create and use the ENUM type.

```
CREATE TYPE lotr_race AS ENUM ('Mayar', 'Hobbit', 'Orc');

CREATE TABLE character (
    name text,
    race lotr_race
);
```

```
INSERT INTO character VALUES ('Frodo Quackins','Hobbit'), ('Quackalf ', 'Mayar');

-- We can perform a normal string comparison
-- Note that 'Hobbit' will be cast to a lotr_race
-- hence this comparison is actually a fast integer comparison
SELECT name FROM character WHERE race = 'Hobbit';
----
Frodo Quackins
```

ENUM columns behave exactly the same as normal VARCHAR columns. They can be used in string functions (such as `LIKE` or `substring`), they can be compared, ordered, etc. The only exception is that ENUM columns can only hold the values that are specified in the enum definition. Inserting a value that is not part of the enum definition will result in an error.

DuckDB ENUMs are currently static (i.e., values can not be added or removed after the ENUM definition). However, ENUM updates are on the roadmap for the next version.

See the documentation for more information.

Python

Setup

First we need to install DuckDB and Pandas. The installation process of both libraries in Python is straightforward:

```
# Python Install
pip install duckdb
pip install pandas
```

Usage

Pandas columns from the categorical type are directly converted to DuckDB's ENUM types:

```
import pandas as pd
import duckdb

# Our unencoded data.
data = ['Hobbit', 'Elf', 'Elf', 'Man', 'Mayar', 'Hobbit', 'Mayar']

# 'pd.Categorical' automatically encodes the data as a categorical column
df_in = pd.DataFrame({'races': pd.Categorical(data),})

# We can query this dataframe as we would any other
# The conversion from categorical columns to enums happens automatically
df_out = duckdb.execute("SELECT * FROM df_in").df()
```

R

Setup

We only need to install DuckDB in our R client, and we are ready to go.

```
# R Install
install.packages("duckdb")
```

Usage

Similar to our previous example with Pandas, R Factor columns are also automatically converted to DuckDB's ENUM types.

```r
library ("duckdb")

con <- dbConnect(duckdb::duckdb())
on.exit(dbDisconnect(con, shutdown = TRUE))

# Our unencoded data.
data <- c('Hobbit', 'Elf', 'Elf', 'Man', 'Mayar', 'Hobbit', 'Mayar')

# Our R dataframe holding an encoded version of our data column
# 'as.factor' automatically encodes it.
df_in <- data.frame(races=as.factor(data))


duckdb::duckdb_register(con, "characters", df_in)
df_out <- dbReadTable(con, "characters")
```

Benchmark Comparison

To demonstrate the performance of DuckDB when running operations on categorical columns of Pandas DataFrames, we present a number of benchmarks. The source code for the benchmarks is available on GitHub. In our benchmarks we always consume and produce Pandas DataFrames.

Dataset

Our dataset is composed of one dataframe with 4 columns and 10 million rows. The first two columns are named `race` and `subrace` representing races. They are both categorical, with the same categories but different values. The other two columns `race_string` and `subrace_string` are the string representations of `race` and ‘subrace“‘.

```python
def generate_df(size):
  race_categories = ['Hobbit', 'Elf', 'Man', 'Mayar']
  race = np.random.choice(race_categories, size)
  subrace = np.random.choice(race_categories, size)
  return pd.DataFrame({'race': pd.Categorical(race),
                       'subrace': pd.Categorical(subrace),
                       'race_string': race,
                       'subrace_string': subrace,})

size = pow(10,7) #10,000,000 rows
df = generate_df(size)
```

Grouped Aggregation

In our grouped aggregation benchmark, we do a count of how many characters for each race we have in the `race` or `race_string` column of our table.

```python
def duck_categorical(df):
  return con.execute("SELECT race, count(*) FROM df GROUP BY race").df()

def duck_string(df):
  return con.execute("SELECT race_string, count(*) FROM df GROUP BY race_string").df()
```

```
def pandas(df):
  return df.groupby(['race']).agg({'race': 'count'})

def pandas_string(df):
  return df.groupby(['race_string']).agg({'race_string': 'count'})
```

The table below depicts the timings of this operation. We can see the benefits of performing grouping on encoded values over strings, with DuckDB being 4× faster when grouping small unsigned values.

Name	Time (s)
DuckDB (Categorical)	0.01
DuckDB (String)	0.04
Pandas (Categorical)	0.06
Pandas (String)	0.40

Filter

In our filter benchmark, we do a count of how many Hobbit characters we have in the `race` or `race_string` column of our table.

```
def duck_categorical(df):
  return con.execute("SELECT count(*) FROM df WHERE race = 'Hobbit'").df()

def duck_string(df):
  return con.execute("SELECT count(*) FROM df WHERE race_string = 'Hobbit'").df()

def pandas(df):
  filtered_df = df[df.race == "Hobbit"]
  return filtered_df.agg({'race': 'count'})

def pandas_string(df):
  filtered_df = df[df.race_string == "Hobbit"]
  return filtered_df.agg({'race_string': 'count'})
```

For the DuckDB enum type, DuckDB converts the string `Hobbit` to a value in the `ENUM`, which returns an unsigned integer. We can then do fast numeric comparisons, instead of expensive string comparisons, which results in greatly improved performance.

Name	Time (s)
DuckDB (Categorical)	0.003
DuckDB (String)	0.023
Pandas (Categorical)	0.158
Pandas (String)	0.440

Enum – Enum Comparison

In this benchmark, we perform an equality comparison of our two breed columns. `race` and `subrace` or `race_string` and ‘subrace_string“‘

```
def duck_categorical(df):
  return con.execute("SELECT count(*) FROM df WHERE race = subrace").df()

def duck_string(df):
```

```
  return con.execute("SELECT count(*) FROM df WHERE race_string = subrace_string").df()

def pandas(df):
  filtered_df = df[df.race == df.subrace]
  return filtered_df.agg({'race': 'count'})

def pandas_string(df):
  filtered_df = df[df.race_string == df.subrace_string]
  return filtered_df.agg({'race_string': 'count'})
```

DuckDB ENUMs can be compared directly on their encoded values. This results in a time difference similar to the previous case, again because we are able to compare numeric values instead of strings.

Name	Time (s)
DuckDB (Categorical)	0.005
DuckDB (String)	0.040
Pandas (Categorical)	0.130
Pandas (String)	0.550

Storage

In this benchmark, we compare the storage savings of storing ENUM Types vs Strings.

```
race_categories = ['Hobbit', 'Elf', 'Man','Mayar']
race = np.random.choice(race_categories, size)
categorical_race = pd.DataFrame({'race': pd.Categorical(race),})
string_race = pd.DataFrame({'race': race,})
con = duckdb.connect('duck_cat.db')
con.execute("CREATE TABLE character AS SELECT * FROM categorical_race")
con = duckdb.connect('duck_str.db')
con.execute("CREATE TABLE character AS SELECT * FROM string_race")
```

The table below depicts the DuckDB file size differences when storing the same column as either an Enum or a plain string. Since the dictionary-encoding does not repeat the string values, we can see a reduction of one order of magnitude in size.

Name	Size (MB)
DuckDB (Categorical)	11
DuckDB (String)	102

What about the Sequels?

There are three main directions we will pursue in the following versions of DuckDB related to ENUMs.

1. Automatic Storage Encoding: As described in the introduction, users frequently define database columns as Strings when in reality they are ENUMs. Our idea is to automatically detect and dictionary-encode these columns, without any input of the user and in a way that is completely invisible to them.
2. ENUM Updates: As said in the introduction, our ENUMs are currently static. We will allow the insertion and removal of ENUM categories.
3. Integration with other Data Formats: We want to expand our integration with data formats that implement ENUM-like structures.

Feedback

As usual, let us know what you think about our ENUM integration, which data formats you would like us to integrate with and any ideas you would like us to pursue on this topic! Feel free to send me an email. If you encounter any problems when using our ENUMs, please open an issue in our issue tracker!

DuckDB Quacks Arrow: A Zero-Copy Data Integration between Apache Arrow and DuckDB

Publication date: 2021-12-03

Authors: Pedro Holanda and Jonathan Keane

TL;DR: The zero-copy integration between DuckDB and Apache Arrow allows for rapid analysis of larger than memory datasets in Python and R using either SQL or relational APIs.

This post is a collaboration with and cross-posted on the Arrow blog.

Part of Apache Arrow is an in-memory data format optimized for analytical libraries. Like Pandas and R Dataframes, it uses a columnar data model. But the Arrow project contains more than just the format: The Arrow C++ library, which is accessible in Python, R, and Ruby via bindings, has additional features that allow you to compute efficiently on datasets. These additional features are on top of the implementation of the in-memory format described above. The datasets may span multiple files in Parquet, CSV, or other formats, and files may even be on remote or cloud storage like HDFS or Amazon S3. The Arrow C++ query engine supports the streaming of query results, has an efficient implementation of complex data types (e.g., Lists, Structs, Maps), and can perform important scan optimizations like Projection and Filter Pushdown.

DuckDB is a new analytical data management system that is designed to run complex SQL queries within other processes. DuckDB has bindings for R and Python, among others. DuckDB can query Arrow datasets directly and stream query results back to Arrow. This integration allows users to query Arrow data using DuckDB's SQL Interface and API, while taking advantage of DuckDB's parallel vectorized execution engine, without requiring any extra data copying. Additionally, this integration takes full advantage of Arrow's predicate and filter pushdown while scanning datasets.

This integration is unique because it uses zero-copy streaming of data between DuckDB and Arrow and vice versa so that you can compose a query using both together. This results in three main benefits:

1. **Larger Than Memory Analysis:** Since both libraries support streaming query results, we are capable of executing on data without fully loading it from disk. Instead, we can execute one batch at a time. This allows us to execute queries on data that is bigger than memory.
2. **Complex Data Types:** DuckDB can efficiently process complex data types that can be stored in Arrow vectors, including arbitrarily nested structs, lists, and maps.
3. **Advanced Optimizer:** DuckDB's state-of-the-art optimizer can push down filters and projections directly into Arrow scans. As a result, only relevant columns and partitions will be read, allowing the system to e.g., take advantage of partition elimination in Parquet files. This significantly accelerates query execution.

For those that are just interested in benchmarks, you can jump ahead benchmark section below.

Quick Tour

Before diving into the details of the integration, in this section we provide a quick motivating example of how powerful and simple to use is the DuckDB-Arrow integration. With a few lines of code, you can already start querying Arrow datasets. Say you want to analyze the infamous NYC Taxi Dataset and figure out if groups tip more or less than single riders.

R

Both Arrow and DuckDB support dplyr pipelines for people more comfortable with using dplyr for their data analysis. The Arrow package includes two helper functions that allow us to pass data back and forth between Arrow and DuckDB (`to_duckdb()` and `to_arrow()`).

This is especially useful in cases where something is supported in one of Arrow or DuckDB but not the other. For example, if you find a complex dplyr pipeline where the SQL translation doesn't work with DuckDB, use `to_arrow()` before the pipeline to use the Arrow engine. Or, if you have a function (e.g., windowed aggregates) that aren't yet implemented in Arrow, use `to_duckdb()` to use the DuckDB engine. All while not paying any cost to (re)serialize the data when you pass it back and forth!

```
library(duckdb)
library(arrow)
library(dplyr)

# Open dataset using year, month folder partition
ds <- arrow::open_dataset("nyc-taxi", partitioning = c("year", "month"))

ds %>%
  # Look only at 2015 on, where the number of passenger is positive, the trip distance is
  # greater than a quarter mile, and where the fare amount is positive
  filter(year > 2014 & passenger_count > 0 & trip_distance > 0.25 & fare_amount > 0) %>%
  # Pass off to DuckDB
  to_duckdb() %>%
  group_by(passenger_count) %>%
  mutate(tip_pct = tip_amount / fare_amount) %>%
  summarise(
    fare_amount = mean(fare_amount, na.rm = TRUE),
    tip_amount = mean(tip_amount, na.rm = TRUE),
    tip_pct = mean(tip_pct, na.rm = TRUE)
  ) %>%
  arrange(passenger_count) %>%
  collect()
```

Python

The workflow in Python is as simple as it is in R. In this example we use DuckDB's Relational API.

```
import duckdb
import pyarrow as pa
import pyarrow.dataset as ds

# Open dataset using year, month folder partition
nyc = ds.dataset('nyc-taxi/', partitioning=["year", "month"])

# We transform the nyc dataset into a DuckDB relation
nyc = duckdb.arrow(nyc)

# Run same query again
nyc.filter("year > 2014 & passenger_count > 0 & trip_distance > 0.25 & fare_amount > 0")
    .aggregate("SELECT avg(fare_amount), avg(tip_amount), avg(tip_amount / fare_amount) AS tip_pct",
"passenger_count").arrow()
```

DuckDB and Arrow: The Basics

In this section, we will look at some basic examples of the code needed to read and output Arrow tables in both Python and R.

Setup

First we need to install DuckDB and Arrow. The installation process for both libraries is shown below.

Python:

```
pip install duckdb
pip install pyarrow
```

R:

```
install.packages("duckdb")
install.packages("arrow")
```

To execute the sample examples in this section, we need to download the following custom Parquet files:

- integers.parquet
- lineitemsf1.snappy.parquet

Python

There are two ways in Python of querying data from Arrow.

1. Through the Relational API:

```
# Reads Parquet File to an Arrow Table
arrow_table = pq.read_table('integers.parquet')

# Transforms Arrow Table -> DuckDB Relation
rel_from_arrow = duckdb.arrow(arrow_table)

# we can run a SQL query on this and print the result
print(rel_from_arrow.query('arrow_table', 'SELECT sum(data) FROM arrow_table WHERE data >
50').fetchone())

# Transforms DuckDB Relation -> Arrow Table
arrow_table_from_duckdb = rel_from_arrow.arrow()
```

2. By using replacement scans and querying the object directly with SQL:

```
# Reads Parquet File to an Arrow Table
arrow_table = pq.read_table('integers.parquet')

# Gets Database Connection
con = duckdb.connect()

# we can run a SQL query on this and print the result
print(con.execute('SELECT sum(data) FROM arrow_table WHERE data > 50').fetchone())

# Transforms Query Result from DuckDB to Arrow Table
# We can directly read the arrow object through DuckDB's replacement scans.
con.execute("SELECT * FROM arrow_table").fetch_arrow_table()
```

It is possible to transform both DuckDB Relations and Query Results back to Arrow.

R

In R, you can interact with Arrow data in DuckDB by registering the table as a view (an alternative is to use dplyr as shown above).

```
library(duckdb)
library(arrow)
library(dplyr)

# Reads Parquet File to an Arrow Table
arrow_table <- arrow::read_parquet("integers.parquet", as_data_frame = FALSE)
```

```r
# Gets Database Connection
con <- dbConnect(duckdb::duckdb())

# Registers arrow table as a DuckDB view
arrow::to_duckdb(arrow_table, table_name = "arrow_table", con = con)

# we can run a SQL query on this and print the result
print(dbGetQuery(con, "SELECT sum(data) FROM arrow_table WHERE data > 50"))

# Transforms Query Result from DuckDB to Arrow Table
result <- dbSendQuery(con, "SELECT * FROM arrow_table")
```

Streaming Data from/to Arrow

In the previous section, we depicted how to interact with Arrow tables. However, Arrow also allows users to interact with the data in a streaming fashion. Either consuming it (e.g., from an Arrow Dataset) or producing it (e.g., returning a RecordBatchReader). And of course, DuckDB is able to consume Datasets and produce RecordBatchReaders. This example uses the NYC Taxi Dataset, stored in Parquet files partitioned by year and month, which we can download through the Arrow R package:

```r
arrow::copy_files("s3://ursa-labs-taxi-data", "nyc-taxi")
```

Python

```python
# Reads dataset partitioning it in year/month folder
nyc_dataset = ds.dataset('nyc-taxi/', partitioning=["year", "month"])

# Gets Database Connection
con = duckdb.connect()

query = con.execute("SELECT * FROM nyc_dataset")
# DuckDB's queries can now produce a Record Batch Reader
record_batch_reader = query.fetch_record_batch()
# Which means we can stream the whole query per batch.
# This retrieves the first batch
chunk = record_batch_reader.read_next_batch()
```

R

```r
# Reads dataset partitioning it in year/month folder
nyc_dataset = open_dataset("nyc-taxi/", partitioning = c("year", "month"))

# Gets Database Connection
con <- dbConnect(duckdb::duckdb())

# We can use the same function as before to register our arrow dataset
duckdb::duckdb_register_arrow(con, "nyc", nyc_dataset)

res <- dbSendQuery(con, "SELECT * FROM nyc", arrow = TRUE)
# DuckDB's queries can now produce a Record Batch Reader
record_batch_reader <- duckdb::duckdb_fetch_record_batch(res)

# Which means we can stream the whole query per batch.
# This retrieves the first batch
cur_batch <- record_batch_reader$read_next_batch()
```

The preceding R code shows in low-level detail how the data is streaming. We provide the helper `to_arrow()` in the Arrow package which is a wrapper around this that makes it easy to incorporate this streaming into a dplyr pipeline.

> In Arrow 6.0.0, `to_arrow()` currently returns the full table, but will allow full streaming in our upcoming 7.0.0 release.

Benchmark Comparison

Here we demonstrate in a simple benchmark the performance difference between querying Arrow datasets with DuckDB and querying Arrow datasets with Pandas. For both the Projection and Filter pushdown comparison, we will use Arrow tables. That is due to Pandas not being capable of consuming Arrow stream objects.

For the NYC Taxi benchmarks, we used a server in the SciLens cluster and for the TPC-H benchmarks, we used a MacBook Pro with an M1 CPU. In both cases, parallelism in DuckDB was used (which is now on by default).

For the comparison with Pandas, note that DuckDB runs in parallel, while pandas only support single-threaded execution. Besides that, one should note that we are comparing automatic optimizations. DuckDB's query optimizer can automatically push down filters and projections. This automatic optimization is not supported in pandas, but it is possible for users to manually perform some of these predicate and filter pushdowns by manually specifying them in the `read_parquet()` call.

Projection Pushdown

In this example we run a simple aggregation on two columns of our lineitem table.

```
# DuckDB
lineitem = pq.read_table('lineitemsf1.snappy.parquet')
con = duckdb.connect()

# Transforms Query Result from DuckDB to Arrow Table
con.execute("""SELECT sum(l_extendedprice * l_discount) AS revenue
               FROM
               lineitem;""").fetch_arrow_table()

# Pandas
arrow_table = pq.read_table('lineitemsf1.snappy.parquet')

# Converts an Arrow table to a Dataframe
df = arrow_table.to_pandas()

# Runs aggregation
res =  pd.DataFrame({'sum': [(df.l_extendedprice * df.l_discount).sum()]})

# Creates an Arrow Table from a Dataframe
new_table = pa.Table.from_pandas(res)
```

Name	Time (s)
DuckDB	0.19
Pandas	2.13

The lineitem table is composed of 16 columns, however, to execute this query only two columns `l_extendedprice` and `l_discount` are necessary. Since DuckDB can push down the projection of these columns, it is capable of executing this query about one order of magnitude faster than Pandas.

Filter Pushdown

For our filter pushdown we repeat the same aggregation used in the previous section, but add filters on 4 more columns.

```
# DuckDB
lineitem = pq.read_table('lineitemsf1.snappy.parquet')

# Get database connection
con = duckdb.connect()

# Transforms Query Result from DuckDB to Arrow Table
con.execute("""SELECT sum(l_extendedprice * l_discount) AS revenue
        FROM
            lineitem
        WHERE
            l_shipdate >= CAST('1994-01-01' AS date)
            AND l_shipdate < CAST('1995-01-01' AS date)
            AND l_discount BETWEEN 0.05
            AND 0.07
            AND l_quantity < 24; """).fetch_arrow_table()

# Pandas
arrow_table = pq.read_table('lineitemsf1.snappy.parquet')

df = arrow_table.to_pandas()
filtered_df = lineitem[
        (lineitem.l_shipdate >= "1994-01-01") &
        (lineitem.l_shipdate < "1995-01-01") &
        (lineitem.l_discount >= 0.05) &
        (lineitem.l_discount <= 0.07) &
        (lineitem.l_quantity < 24)]

res =  pd.DataFrame({'sum': [(filtered_df.l_extendedprice * filtered_df.l_discount).sum()]})
new_table = pa.Table.from_pandas(res)
```

Name	Time (s)
DuckDB	0.04
Pandas	2.29

The difference now between DuckDB and Pandas is more drastic, being two orders of magnitude faster than Pandas. Again, since both the filter and projection are pushed down to Arrow, DuckDB reads less data than Pandas, which can't automatically perform this optimization.

Streaming

As demonstrated before, DuckDB is capable of consuming and producing Arrow data in a streaming fashion. In this section we run a simple benchmark, to showcase the benefits in speed and memory usage when comparing it to full materialization and Pandas. This example uses the full NYC taxi dataset which you can download

```
# DuckDB
# Open dataset using year, month folder partition
nyc = ds.dataset('nyc-taxi/', partitioning=["year", "month"])

# Get database connection
con = duckdb.connect()

# Run query that selects part of the data
query = con.execute("SELECT total_amount, passenger_count, year FROM nyc where total_amount > 100 and
year > 2014")
```

```python
# Create Record Batch Reader from Query Result.
# "fetch_record_batch()" also accepts an extra parameter related to the desired produced chunk size.
record_batch_reader = query.fetch_record_batch()

# Retrieve all batch chunks
chunk = record_batch_reader.read_next_batch()
while len(chunk) > 0:
    chunk = record_batch_reader.read_next_batch()

# Pandas
# We must exclude one of the columns of the NYC dataset due to an unimplemented cast in Arrow.
working_columns = ["vendor_id","pickup_at","dropoff_at","passenger_count","trip_distance","pickup_
longitude",
    "pickup_latitude","store_and_fwd_flag","dropoff_longitude","dropoff_latitude","payment_type",
    "fare_amount","extra","mta_tax","tip_amount","tolls_amount","total_amount","year", "month"]

# Open dataset using year, month folder partition
nyc_dataset = ds.dataset(dir, partitioning=["year", "month"])
# Generate a scanner to skip problematic column
dataset_scanner = nyc_dataset.scanner(columns=working_columns)

# Materialize dataset to an Arrow Table
nyc_table = dataset_scanner.to_table()

# Generate Dataframe from Arow Table
nyc_df = nyc_table.to_pandas()

# Apply Filter
filtered_df = nyc_df[
    (nyc_df.total_amount > 100) &
    (nyc_df.year >2014)]

# Apply Projection
res = filtered_df[["total_amount", "passenger_count","year"]]

# Transform Result back to an Arrow Table
new_table = pa.Table.from_pandas(res)
```

Name	Time (s)	Peak memory usage (GBs)
DuckDB	0.05	0.3
Pandas	146.91	248

The difference in times between DuckDB and Pandas is a combination of all the integration benefits we explored in this article. In DuckDB the filter pushdown is applied to perform partition elimination (i.e., we skip reading the Parquet files where the year is <= 2014). The filter pushdown is also used to eliminate unrelated row_groups (i.e., row groups where the total amount is always <= 100). Due to our projection pushdown, Arrow only has to read the columns of interest from the Parquet files, which allows it to read only 4 out of 20 columns. On the other hand, Pandas is not capable of automatically pushing down any of these optimizations, which means that the full dataset must be read. **This results in the 4 orders of magnitude difference in query execution time.**

In the table above, we also depict the comparison of peak memory usage between DuckDB (Streaming) and Pandas (Fully-Materializing). In DuckDB, we only need to load the row group of interest into memory. Hence our memory usage is low. We also have constant memory usage since we only have to keep one of these row groups in-memory at a time. Pandas, on the other hand, has to fully materialize all Parquet files when executing the query. Because of this, we see a constant steep increase in its memory consumption. **The total difference in memory consumption of the two solutions is around 3 orders of magnitude.**

Conclusion and Feedback

In this blog post, we mainly showcased how to execute queries on Arrow datasets with DuckDB. There are additional libraries that can also consume the Arrow format but they have different purposes and capabilities. As always, we are happy to hear if you want to see benchmarks with different tools for a post in the future! Feel free to drop us an email or share your thoughts directly in the Hacker News post.

Last but not least, if you encounter any problems when using our integration, please open an issue in either DuckDB's issue tracker or Arrow's issue tracker, depending on which library has a problem.

DuckDB Time Zones: Supporting Calendar Extensions

Publication date: 2022-01-06

Author: Richard Wesley

TL;DR: The DuckDB ICU extension now provides time zone support.

Time zone support is a common request for temporal analytics, but the rules are complex and somewhat arbitrary. The most well supported library for locale-specific operations is the International Components for Unicode (ICU). DuckDB already provided collated string comparisons using ICU via an extension (to avoid dependencies), and we have now connected the existing ICU calendar and time zone functions to the main code via the new `TIMESTAMP WITH TIME ZONE` (or `TIMESTAMPTZ` for short) data type. The ICU extension is pre-bundled in DuckDB's Python client and can be optionally installed in the remaining clients.

In this post, we will describe how time works in DuckDB and what time zone functionality has been added.

What is Time?

> People assume that time is a strict progression of cause to effect, but actually from a non-linear, non-subjective viewpoint it's more like a big ball of wibbly wobbly timey wimey stuff.
> -- Doctor Who: Blink

Time in databases can be very confusing because the way we talk about time is itself confusing. Local time, GMT, UTC, time zones, leap years, proleptic Gregorian calendars – it all looks like a big mess. But if you step back, modeling time is actually fairly simple, and can be reduced to two pieces: instants and binning.

Instants

You will often hear people (and documentation) say that database time is stored in UTC. This is sort of right, but it is more accurate to say that databases store *instants*. An instant is a point in universal time, and they are usually given as a count of some time increment from a fixed point in time (called the *epoch*). In DuckDB, the fixed point is the Unix epoch `1970-01-01 00:00:00 +00:00`, and the increment is microseconds (µs). (Note that to avoid confusion we will be using ISO-8601 y-m-d notation in this post to denote instants.) In other words, a `TIMESTAMP` column contains instants.

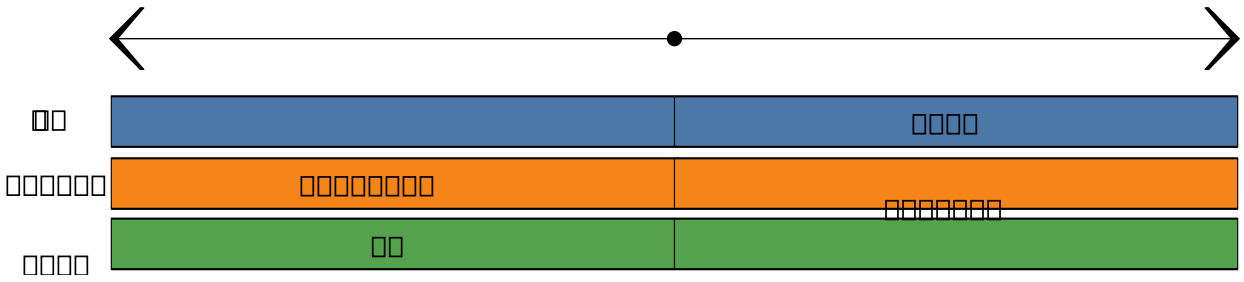
There are three other temporal types in SQL:

- `DATE` – an integral count of days from a fixed date. In DuckDB, the fixed date is `1970-01-01`, again in UTC.
- `TIME` – a (positive) count of microseconds up to a single day
- `INTERVAL` – a set of fields for counting time differences. In DuckDB, intervals count months, days and microseconds. (Months are not completely well-defined, but when present, they represent 30 days.)

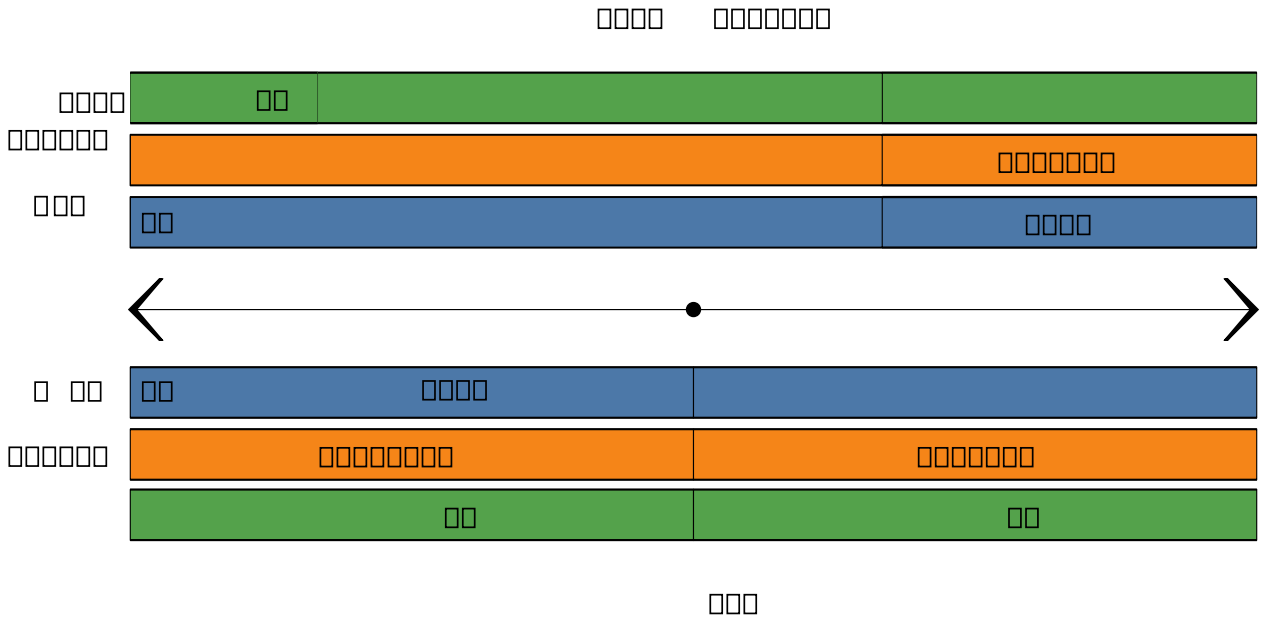
None of these other temporal types except `TIME` can have a `WITH TIME ZONE` modifier (and shorter `TZ` suffix), but to understand what that modifier means, we first need to talk about *temporal binning*.

Temporal Binning

Instants are pretty straightforward – they are just a number – but binning is the part that trips people up. Binning is probably a familiar idea if you have worked with continuous data: You break up a set of values into ranges and map each value to the range (or *bin*) that it falls into. Temporal binning is just doing this to instants:

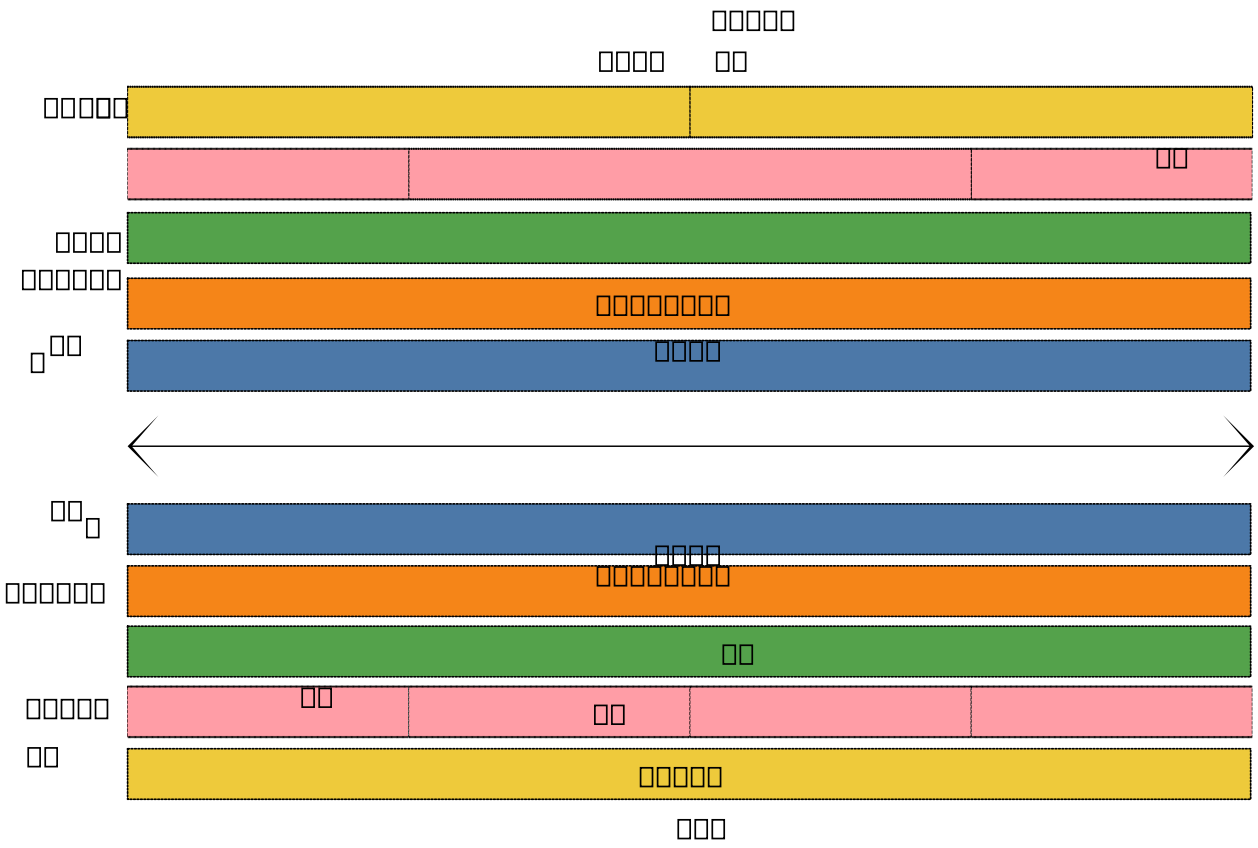


Temporal binning systems are often called *calendars*, but we are going to avoid that term for now because calendars are usually associated with dates, and temporal binning also includes rules for time. These time rules are called *time zones*, and they also impact where the day boundaries used by the calendar fall. For example, here is what the binning for a second time zone looks like at the epoch:



The most confusing thing about temporal binning is that there is more than one way to bin time, and it is not always obvious what binning should be used. For example, what I mean by ”today” is a bin of instants often determined by where I live. Every instant that is part of my ”today” goes in that bin. But notice that I qualified ”today” with ”where I live”, and that qualification determines what binning system is being used. But ”today” could also be determined by ”where the events happened”, which would require a different binning to be applied.

The biggest temporal binning problem most people run into occurs when daylight savings time changes. This example contains a daylight savings time change where the ”hour” bin is two hours long! To distinguish the two hours, we needed to include another bin containing the offset from UTC:



As this example shows, in order to bin the instants correctly, we need to know the binning rules that apply. It also shows that we can't just use the built in binning operations, because they don't understand daylight savings time.

Naïve Timestamps

Instants are sometimes created from a string format using a local binning system instead of an instant. This results in the instants being offset from UTC, which can cause problems with daylight savings time. These are called *naïve* timestamps, and they may constitute a data cleaning problem.

Cleaning naïve timestamps requires determining the offset for each timestamp and then updating the value to be an instant. For most values, this can be done with an inequality join against a table containing the correct offsets, but the ambiguous values may need to be fixed by hand. It may also be possible to correct the ambiguous values by assuming that they were inserted in order and looking for ”backwards jumps” using window functions.

A simple way to avoid this situation going forward is to add the UTC offset to non-UTC strings: `2021-07-31 07:20:15 -07:00`. The DuckDB VARCHAR cast operation parses these offsets correctly and will generate the corresponding instant.

Time Zone Data Types

The SQL standard defines temporal data types qualified by `WITH TIME ZONE`. This terminology is confusing because it seems to imply that the time zone will be stored with the value, but what it really means is ”bin this value using the session's `TimeZone` setting”. Thus a `TIMESTAMPTZ` column also stores instants, but expresses a ”hint” that it should use a specific binning system.

There are a number of operations that can be performed on instants without a binning system:

- Comparing;

- Sorting;
- Increment (µs) difference;
- Casting to and from regular `TIMESTAMP`s.

These common operations have been implemented in the main DuckDB code base, while the binning operations have been delegated to extensions such as ICU.

One small difference between the display of the new `WITH TIME ZONE` types and the older types is that the new types will be displayed with a `+00` UTC offset. This is simply to make the type differences visible in command line interfaces and for testing. Properly formatting a `TIMESTAMPTZ` for display in a locale requires using a binning system.

ICU Temporal Binning

DuckDB already uses an ICU extension for collating strings for a particular locale, so it was natural to extend it to expose the ICU calendar and time zone functionality.

ICU Time Zones

The first step for supporting time zones is to add the `TimeZone` setting that should be applied. DuckDB extensions can define and validate their own settings, and the ICU extension now does this:

```
-- Load the extension
-- This is not needed in Python or R, as the extension is already installed
LOAD icu;

-- Show the current time zone. The default is set to ICU's current time zone.
SELECT * FROM duckdb_settings() WHERE name = 'TimeZone';
```

```
TimeZone    Europe/Amsterdam    The current time zone    VARCHAR
```

```
-- Choose a time zone.
SET TimeZone = 'America/Los_Angeles';

-- Emulate Postgres' time zone table
SELECT name, abbrev, utc_offset
FROM pg_timezone_names()
ORDER BY 1
LIMIT 5;
```

```
ACT ACT 09:30:00
AET AET 10:00:00
AGT AGT -03:00:00
ART ART 02:00:00
AST AST -09:00:00
```

ICU Temporal Binning Functions

Databases like DuckDB and Postgres usually provide some temporal binning functions such as `YEAR` or `DATE_PART`. These functions are part of a single binning system for the conventional (proleptic Gregorian) calendar and the UTC time zone. Note that casting to a string is a binning operation because the text produced contains bin values.

Because timestamps that require custom binning have a different data type, the ICU extension can define additional functions with bindings to `TIMESTAMPTZ`:

- `+` – Add an `INTERVAL` to a timestamp
- `-` – Subtract an `INTERVAL` from a timestamp

- AGE – Compute an INTERVAL describing the months/days/microseconds between two timestamps (or one timestamp and the current instant).
- DATE_DIFF – Count part boundary crossings between two timestamp
- DATE_PART – Extract a named timestamp part. This includes the part alias functions such as YEAR.
- DATE_SUB – Count the number of complete parts between two timestamp
- DATE_TRUNC – Truncate a timestamp to the given precision
- LAST_DAY – Returns the last day of the month
- MAKE_TIMESTAMPTZ – Constructs a TIMESTAMPTZ from parts, including an optional final time zone specifier.

We have not implemented these functions for TIMETZ because this type has limited utility, but it would not be difficult to add in the future. We have also not implemented string formatting/casting to VARCHAR because the type casting system is not yet extensible, and the current ICU build we are using does not embed this data.

ICU Calendar Support

ICU can also perform binning operations for some non-Gregorian calendars. We have added support for these calendars via a Calendar setting and the icu_calendar_names table function:

```
LOAD icu;

-- Show the current calendar. The default is set to ICU's current locale.
SELECT * FROM duckdb_settings() WHERE name = 'Calendar';

Calendar    gregorian   The current calendar    VARCHAR

-- List the available calendars
SELECT DISTINCT name FROM icu_calendar_names()
ORDER BY 1 DESC LIMIT 5;

roc
persian
japanese
iso8601
islamic-umalqura

-- Choose a calendar
SET Calendar = 'japanese';

-- Extract the current Japanese era number using Tokyo time
SET TimeZone = 'Asia/Tokyo';

SELECT
     era('2019-05-01 00:00:00+10'::TIMESTAMPTZ),
     era('2019-05-01 00:00:00+09'::TIMESTAMPTZ);

235  236
```

Caveats

ICU has some differences in behavior and representation from the DuckDB implementation. These are hopefully minor issues that should only be of concern to serious time nerds.

- ICU represents instants as millisecond counts using a DOUBLE. This makes it lose accuracy far from the epoch (e.g., around the first millennium)
- ICU uses the Julian calendar for dates before the Gregorian change on 1582-10-15 instead of the proleptic Gregorian calendar. This means that dates prior to the changeover will differ, although ICU will give the date as actually written at the time.
- ICU computes ages by using part increments instead of using the length of the earlier month like DuckDB and Postgres.

Future Work

Temporal analysis is a large area, and while the ICU time zone support is a big step forward, there is still much that could be done. Some of these items are core DuckDB improvements that could benefit all temporal binning systems and some expose more ICU functionality. There is also the prospect for writing other custom binning systems via extensions.

DuckDB Features

Here are some general projects that all binning systems could benefit from:

- Add a `DATE_ROLL` function that emulates the ICU calendar `roll` operation for "rotating" around a containing bin;
- Making casting operations extensible so extensions can add their own support;

ICU Functionality

ICU is a very rich library with a long pedigree, and there is much that could be done with the existing library:

- Create a more general `MAKE_TIMESTAPTZ` variant that takes a `STRUCT` with the parts. This could be useful for some non-Gregorian calendars.
- Extend the embedded data to contain locale temporal information (such as month names) and support formatting (`to_char`) and parsing (`to_timestamp`) of local dates. One issue here is that the ICU date formatting language is more sophisticated than the Postgres language, so multiple functions might be required (e.g., `icu_to_char`);
- Extend the binning functions to take per-row calendar and time zone specifications to support row-level temporal analytics such as "what time of day did this happen"?

Separation of Concerns

Because the time zone data type is defined in the main code base, but the calendar operations are provided by an extension, it is now possible to write application-specific extensions with custom calendar and time zone support such as:

- Financial 4-4-5 calendars;
- ISO week-based years;
- Table-driven calendars;
- Astronomical calendars with leap seconds;
- Fun calendars, such as Shire Reckoning and French Republican!

Conclusion and Feedback

In this blog post, we described the new DuckDB time zone functionality as implemented via the ICU extension. We hope that the functionality provided can enable temporal analytic applications involving time zones. We also look forward to seeing any custom calendar extensions that our users dream up!

Last but not least, if you encounter any problems when using our integration, please open an issue in DuckDB's issue tracker!

Parallel Grouped Aggregation in DuckDB

Publication date: 2022-03-07

Authors: Hannes Mühleisen and Mark Raasveldt

TL;DR: DuckDB has a fully parallelized aggregate hash table that can efficiently aggregate over millions of groups.

Grouped aggregations are a core data analysis command. It is particularly important for large-scale data analysis ("OLAP") because it is useful for computing statistical summaries of huge tables. DuckDB contains a highly optimized parallel aggregation capability for fast and scalable summarization.

Jump straight to the benchmarks?

Introduction

GROUP BY changes the result set cardinality – instead of returning the same number of rows of the input (like a normal SELECT), GROUP BY returns as many rows as there are groups in the data. Consider this (weirdly familiar) example query:

```
SELECT
    l_returnflag,
    l_linestatus,
    sum(l_extendedprice),
    avg(l_quantity)
FROM
    lineitem
GROUP BY
    l_returnflag,
    l_linestatus;
```

GROUP BY is followed by two column names, `l_returnflag` and `l_linestatus`. Those are the columns to compute the groups on, and the resulting table will contain all combinations of the same column that occur in the data. We refer to the columns in the GROUP BY clause as the "grouping columns" and all occurring combinations of values therein as "groups". The SELECT clause contains four (not five) expressions: References to the grouping columns, and two aggregates: the `sum` over `l_extendedprice` and the `avg` over `l_quantity`. We refer to those as the "aggregates". If executed, the result of this query looks something like this:

l_returnflag	l_linestatus	sum(l_extendedprice)	avg(l_quantity)
N	O	114935210409.19	25.5
R	F	56568041380.9	25.51
A	F	56586554400.73	25.52
N	F	1487504710.38	25.52

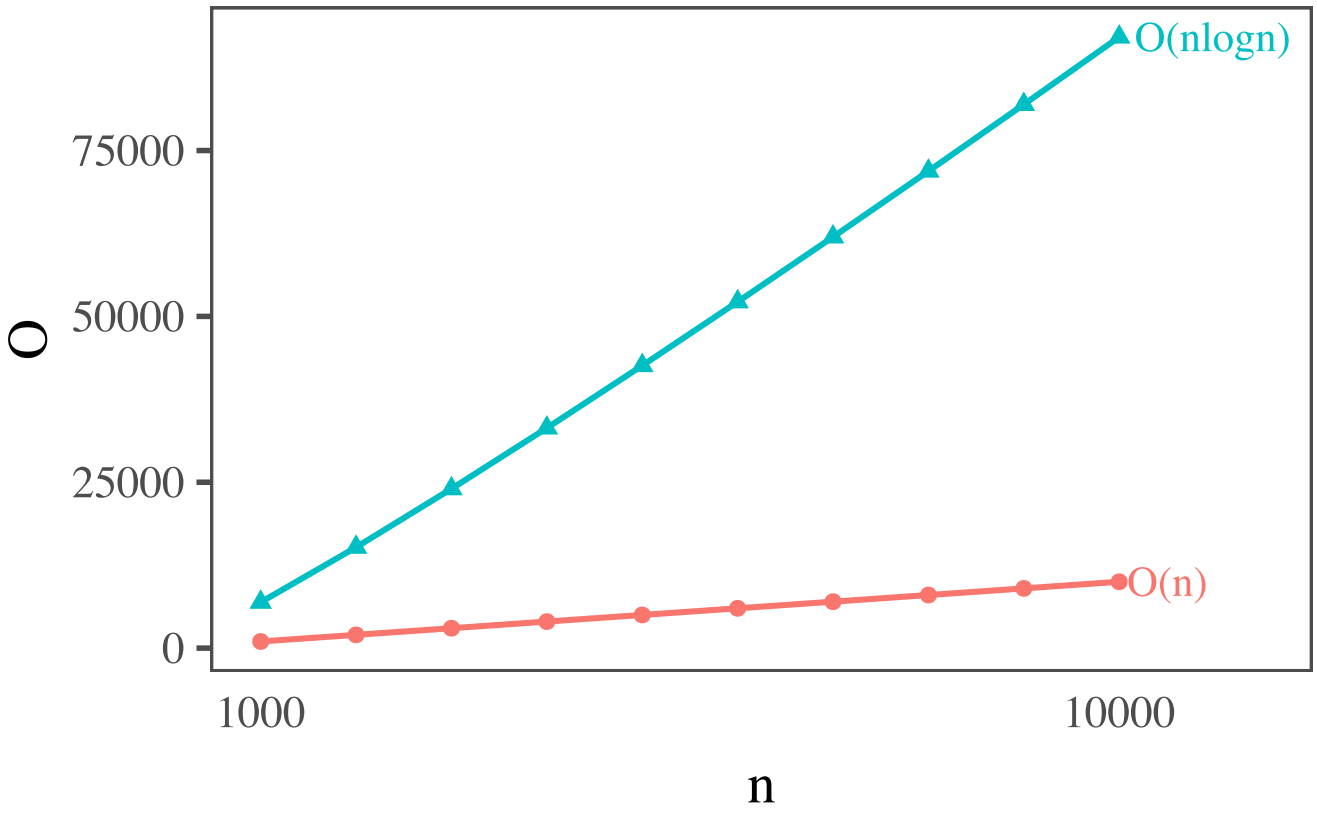
In general, SQL allows only columns that are mentioned in the GROUP BY clause to be part of the SELECT expressions directly, all other columns need to be subject to one of the aggregate functions like `sum`, `avg` etc. There are many more aggregate functions depending on which SQL system you use.

How should a query processing engine compute such an aggregation? There are many design decisions involved, and we will discuss those below and in particular the decisions made by DuckDB. The main issue when computing grouping results is that the groups can occur in

the input table in any order. Were the input already sorted on the grouping columns, computing the aggregation would be trivial, as we could just compare the current values for the grouping columns with the previous ones. If a change occurs, the next group begins and a new aggregation result needs to be computed. Since the sorted case is easy, one straightforward way of computing grouped aggregates is to sort the input table on the grouping columns first, and then use the trivial approach. But sorting the input is unfortunately still a computationally expensive operation despite our best efforts. In general, sorting has a computational complexity of `O(nlogn)` with n being the number of rows sorted.

Hash Tables for Aggregation

A better way is to use a hash table. Hash tables are a foundational data structure in computing that allow us to find entries with a computational complexity of `O(1)`. A full discussion on how hash tables work is far beyond the scope of this post. Below we try to focus on a very basic description and considerations related to aggregate computation.



To add n rows to a hash table we are looking at a complexity of `O(n)`, much, much better than `O(nlogn)` for sorting, especially when n goes into the billions. The figure above illustrates how the complexity develops as the table size increases. Another big advantage is that we do not have to make a sorted copy of the input first, which is going to be just as large as the input. Instead, the hash table will have at most as many entries as there are groups, which can be (and usually are) dramatically fewer than input rows. The overall process is thus this: Scan the input table, and for each row, update the hash table accordingly. Once the input is exhausted, we scan the hash table to provide rows to upstream operators or the query result directly.

Collision Handling

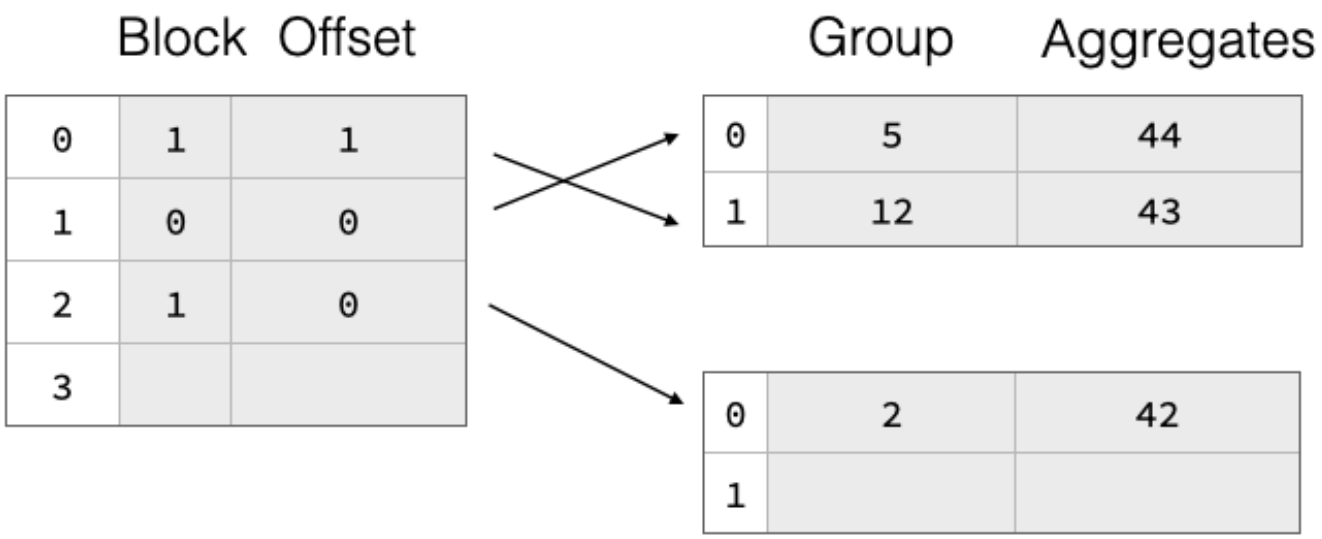
So, hash table it is then! We build a hash table on the input with the groups as keys and the aggregates as the entries. Then, for every input row, we compute a hash of the group values, find the entry in the hash table, and either create or update the aggregate states with the values from the row? Its unfortunately not that simple: Two rows with *different* values for the grouping columns may result in a hash that points to the *same* hash table entry, which would lead to incorrect results.

There are two main approaches to work around this problem: "Chaining" or "linear probing". With chaining, we do not keep the aggregate values in the hash table directly, but rather keep a list of group values and aggregates. If grouping values points to a hash table entry with an empty list, the new group and the aggregates are simply added. If grouping values point to an existing list, we check for every list entry whether the grouping values match. If so, we update the aggregates for that group. If not, we create a new list entry. In linear probing there are no such lists, but on finding an existing entry, we will compare the grouping values, and if they match we will update the entry. If they do not match, we move one entry down in the hash table and try again. This process finishes when either a matching group entry has been found or an empty hash table entry is found. While theoretically equivalent, computer hardware architecture will favor linear probing because of cache locality. Because linear probing walks the hash table entries *linearly*, the next entry will very likely be in the CPU cache and hence access is faster. Chaining will generally lead to random access and much worse performance on modern hardware architectures. We have therefore adopted linear probing for our aggregate hash table.

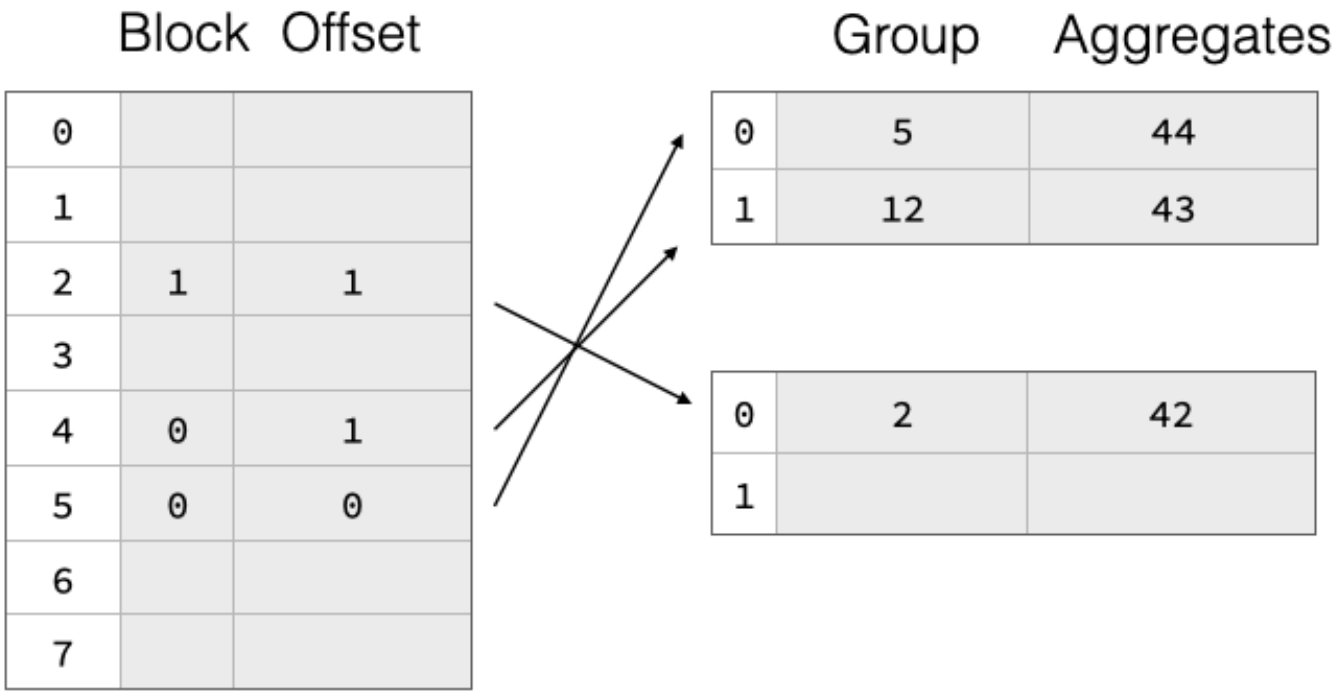
Both chaining and linear probing will degrade in theoretical lookup performance from O(1) to O(n) wrt hash table size if there are too many collisions, i.e., too many groups hashing to the same hash table entry. A common solution to this problem is to resize the hash table once the "fill ratio" exceeds some threshold, e.g., 75% is the default for Java's `HashMap`. This is particularly important as we do not know the amount of groups in the result before starting the aggregation. Neither do we assume to know the amount of rows in the input table. We thus start with a fairly small hash table and resize it once the fill ratio exceeds a threshold. The basic hash table structure is shown in the figure below, the table has four slots 0-4. There are already three groups in the table, with group keys 12, 5 and 2. Each group has aggregate values (e.g., from a `SUM`) of 43 etc.

	Group	Aggregates
0	12	43
1	5	44
2	2	42
3		

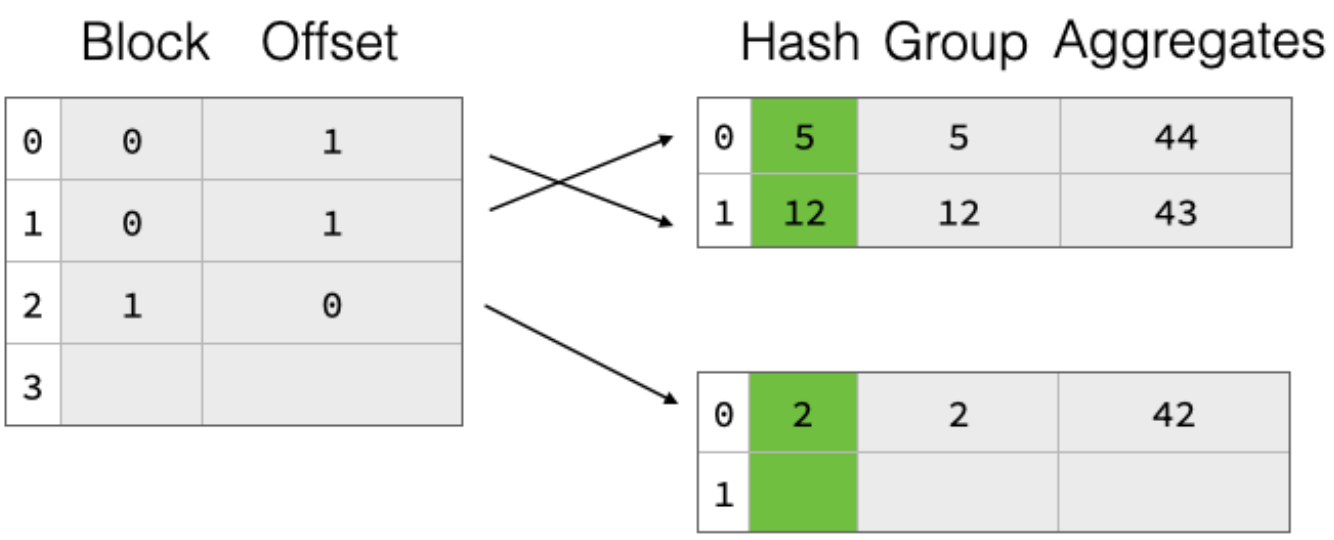
A big challenge with the resize of a partially filled hash table after the resize, all the groups are in the wrong place and we would have to move everything, which will be very expensive.



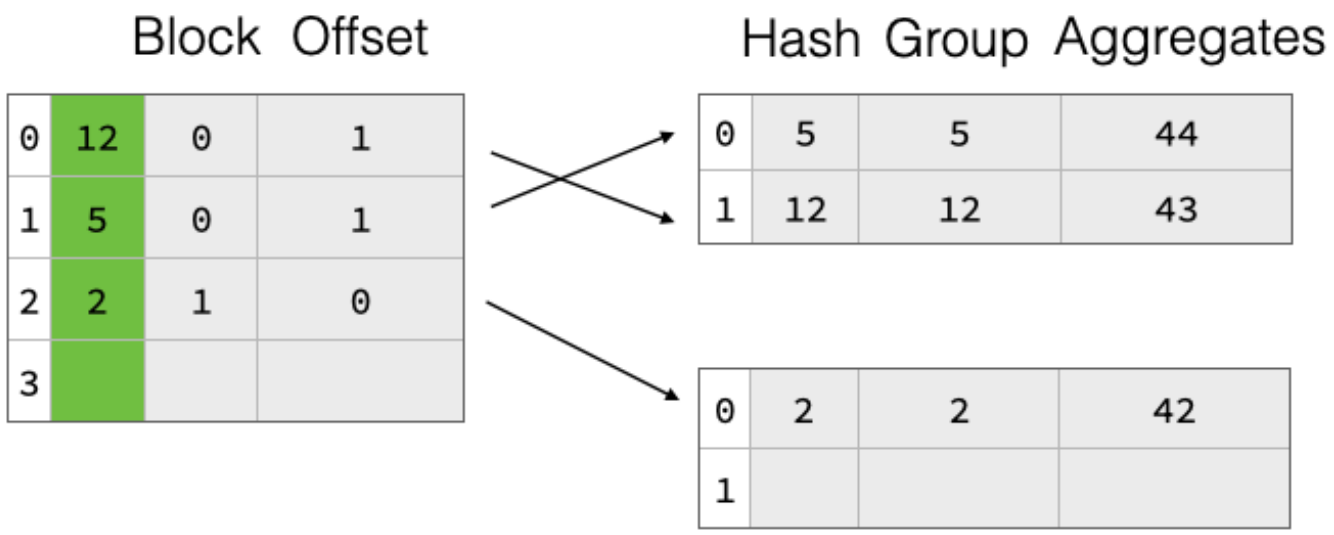
To support resize efficiently, we have implemented a two-part aggregate hash table consisting of a separately-allocated pointer array which points into payload blocks that contain grouping values and aggregate states for each group. The pointers are not actual pointers but symbolic, they refer to a block ID and a row offset within said block. This is shown in the figure above, the hash table entries are split over two payload blocks. On resize, we throw away the pointer array and allocate a bigger one. Then, we read all payload blocks again, hash the group values, and re-insert pointers to them into the new pointer array. The group data thus remains unchanged, which greatly reduces the cost of resizing the hash table. This can be seen in the figure below, where we double the pointer array size but the payload blocks remain unchanged.



The naive two-part hash table design would require a re-hashing of *all* group values on resize, which can be quite expensive especially for string values. To speed this up, we also write the raw hash of the group values to the payload blocks for every group. Then, during resize, we don't have to re-hash the groups but can just read them from the payload blocks, compute the new offset into the pointer array, and insert there.



The two-part hash table has a big drawback when looking up entries: There is no ordering between the pointer array and the group entries in the payload blocks. Hence, following the pointer creates random access in the memory hierarchy. This will lead to unnecessary stalls in the computation. To mitigate this issue, we extend the memory layout of the pointer array to include some (1 or 2) bytes from the group hash in addition to the pointer to the payload value. This way, linear probing can first compare the hash bits in the pointer array with the current group hash and decide whether it's worth following the payload pointer or not. This can potentially continue for every group in the pointer chain. Only when the hash bits match we have to actually follow the pointer and compare the actual groups. This optimization greatly reduces the amount of times the pointer to the payload blocks has to be followed and thereby reduces the amount of random accesses into memory which are directly related to overall performance. It has the nice side-effect of also greatly reducing full group comparisons which can also be expensive, e.g., when aggregating on groups that contain strings.



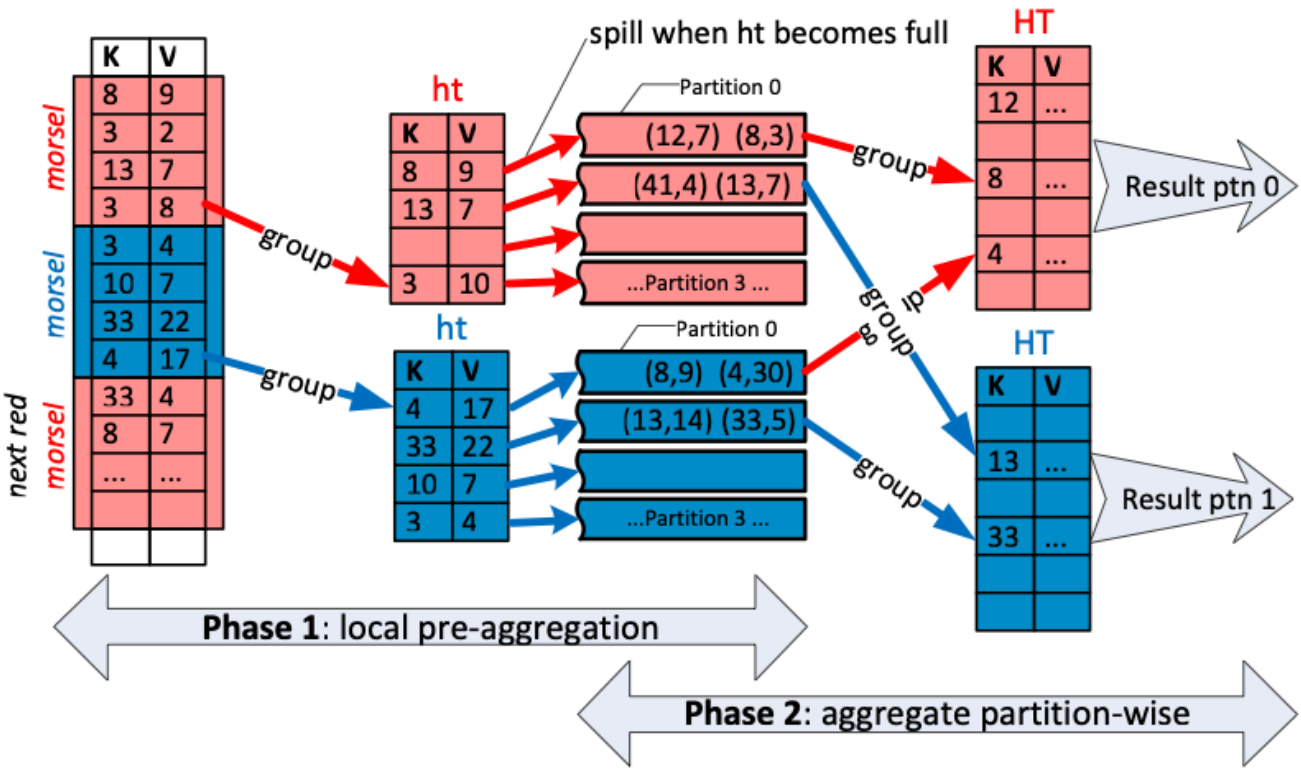
Another (smaller) optimization here concerns the width of the pointer array entries. For small hash tables with few entries, we do not need many bits to encode the payload block offset pointers. DuckDB supports both 4 byte and 8 byte pointer array entries.

For most aggregate queries, the vast majority of query processing time is spent looking up hash table entries, which is why it's worth spending time on optimizing them. If you're curious, the code for all this is in the DuckDB repo, `aggregate_hashtable.cpp`. There is another optimization for when we know that there are only a few distinct groups from column statistics, the perfect hash aggregate, but that's for another post. But we're not done here just yet.

Parallel Aggregation

While we now have an aggregate hash table design that should do fairly well for grouped aggregations, we still have not considered the fact that DuckDB automatically parallelizes all queries to use multiple hardware threads ("CPUs"). How does parallelism work together with hash tables? In general, the answer is unfortunately: "Badly". Hash tables are delicate structures that don't handle parallel modifications well. For example, imagine one thread would want to resize the hash table while another wants to add some new group data to it. Or how should we handle multiple threads inserting new groups at the same time for the same entry? One could use locks to make sure that only one thread at a time is using the table, but this would mostly defeat parallelizing the query. There has been plenty of research into concurrency-friendly hash tables but the short summary is that it's still an open issue.

It is possible to let each thread read data from downstream operators and build individual, local hash tables and merge those together later from a single thread. This works quite nicely if there are few groups like in the example at the top of this post. If there are few groups, a single thread can merge many thread-local hash tables without creating a bottleneck. However, it's entirely possible there are as many groups as there are input rows, for this tends to happen a lot when someone groups on a column that would be a candidate for a primary key, e.g., `observation_number`, `timestamp` etc. What is thus needed is a parallel merge of the parallel hash tables. We adopt a method from Leis et al.: Each thread builds not one, but multiple *partitioned* hash tables based on a radix-partitioning on the group hash.



The key observation here is that if two groups have a different hash value, they cannot possibly be the same. Because of this property, it is possible to use the hash values to create fully independent partitions of the groups without requiring any communication between threads as long as all the threads use the same partitioning scheme (see Phase 1 in the above diagram).

After all the local hash tables have been constructed, we assign individual partitions to each worker thread and merge the hash tables within that partition together (Phase 2). Because the partitions were created using the radix partitioning scheme on the hash, all worker threads can independently merge the hash tables within their respective partitions. The result is correct because each group goes into a single partition and that partition only.

One interesting detail is that we never need to build a final (possibly giant) hash table that holds all the groups because the radix group partitioning ensures that each group is localized to a partition.

There are two additional optimizations for the parallel partitioned hash table strategy: 1) We only start partitioning once a single thread's aggregate hash table exceeds a fixed limit of entries, currently set to 10 000 rows. This is because using a partitioned hash table is not free. For every row added, we have to figure out which partition it should go into, and we have to merge everything back together at the end. For

this reason, we will not start partitioning until the parallelization benefit outweighs the cost. Since the partitioning decision is individual to each thread, it may well be possible only some threads start partitioning. If that is the case, we will need to partition the hash tables of the threads that have not done so before starting merging them. This is a fully thread-local operation however and does not interfere with parallelism. 2) We will stop adding values to a hash table once its pointer array exceeds a certain threshold. Every thread then builds multiple sets of potentially partitioned hash tables. This is because we do not want the pointer array to become arbitrarily large. While this potentially creates duplicate entries for the same group in multiple hash tables, this is not problematic because we merge them all later anyway. This optimization works particularly well on data sets that have many distinct groups, but have group values that are clustered in the input in some manner. For example, when grouping by day in a data set that is ordered on date.

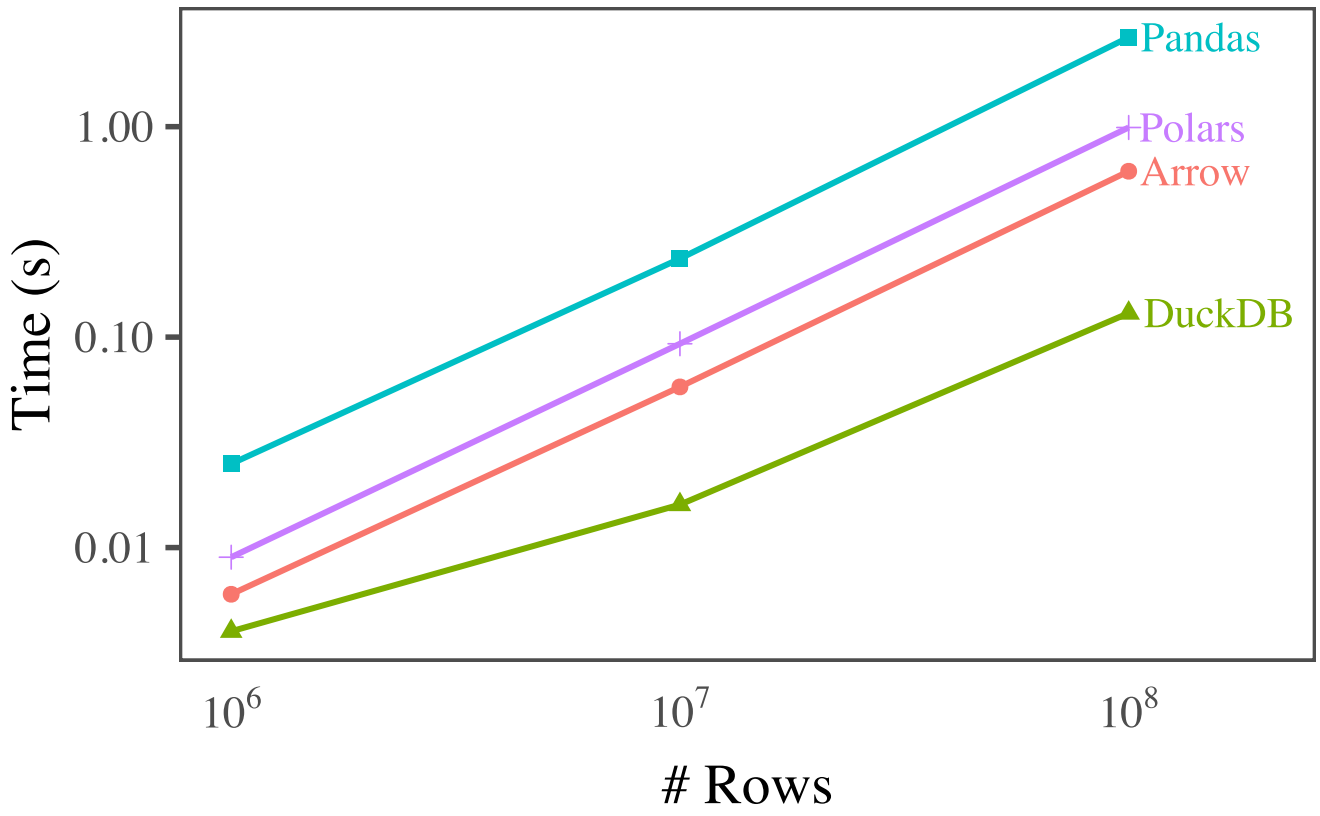
There are some kinds of aggregates which cannot use the parallel and partitioned hash table approach. While it is trivial to parallelize a sum, because the sum of the overall result is just the sum of the individual results, this is fairly impossible for computations like `median`, which DuckDB also supports. Also for this reason, DuckDB also supports `approx_quantile`, which *is* parallelizable.

Experiments

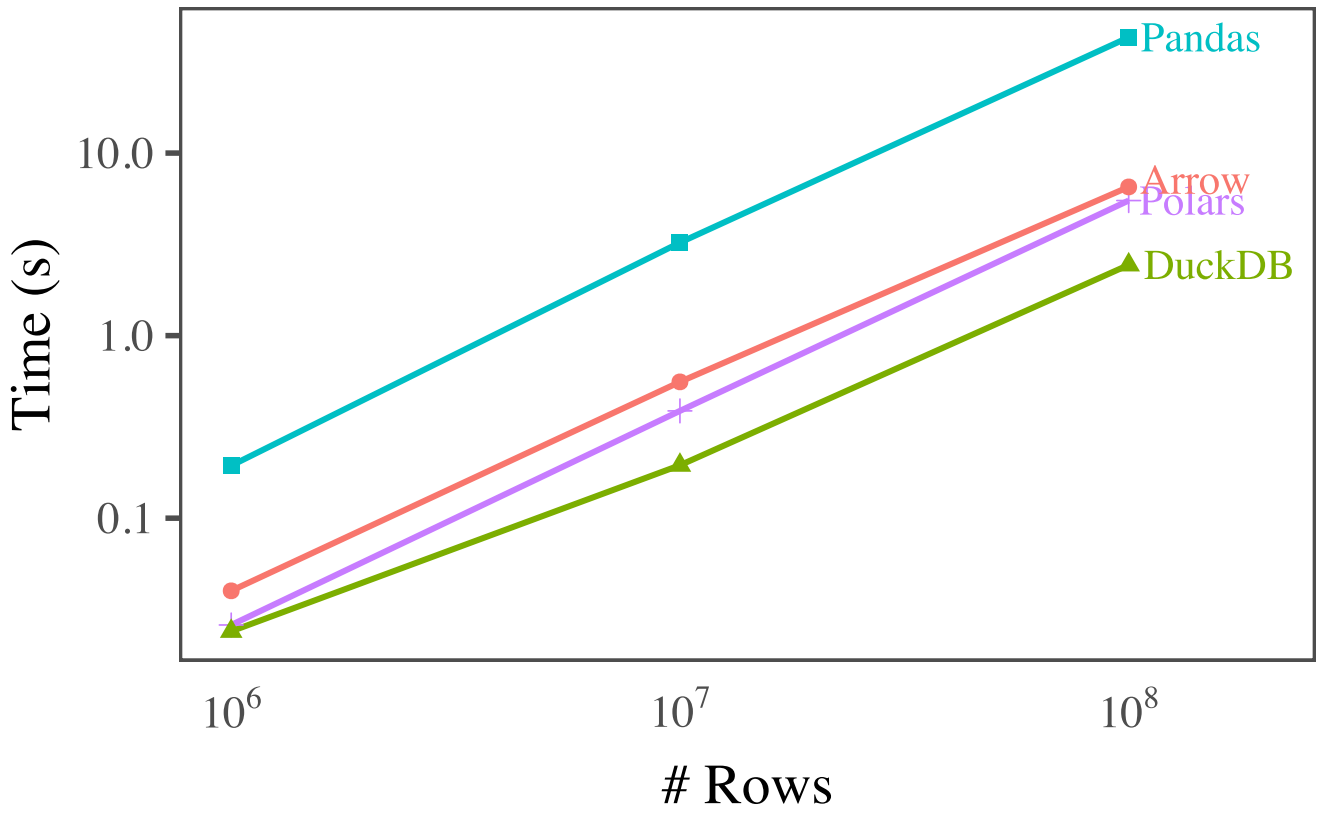
Putting all this together, it's now time for some performance experiments. We will compare DuckDB's aggregation operator as described above with the same operator in various Python data wrangling libraries. The other contenders are Pandas, Polars and Arrow. Those are chosen since they can all execute an aggregation operator on Pandas DataFrames without converting into some other storage format first, just like DuckDB.

For our benchmarks, we generate a synthetic dataset with a pre-defined number of groups over two integer columns and some random integer data to aggregate. The entire dataset is shuffled before the experiments to prevent taking advantage of the clustered nature of the synthetically generated data. For each group, we compute two aggregates, sum of the data column and a simple count. The SQL version of this aggregation would be `SELECT g1, g2, sum(d), count(*) FROM dft GROUP BY g1, g2 LIMIT 1;`. In the experiments below, we vary the dataset size and the amount of groups in them. This should nicely show the scaling behavior of the aggregation.

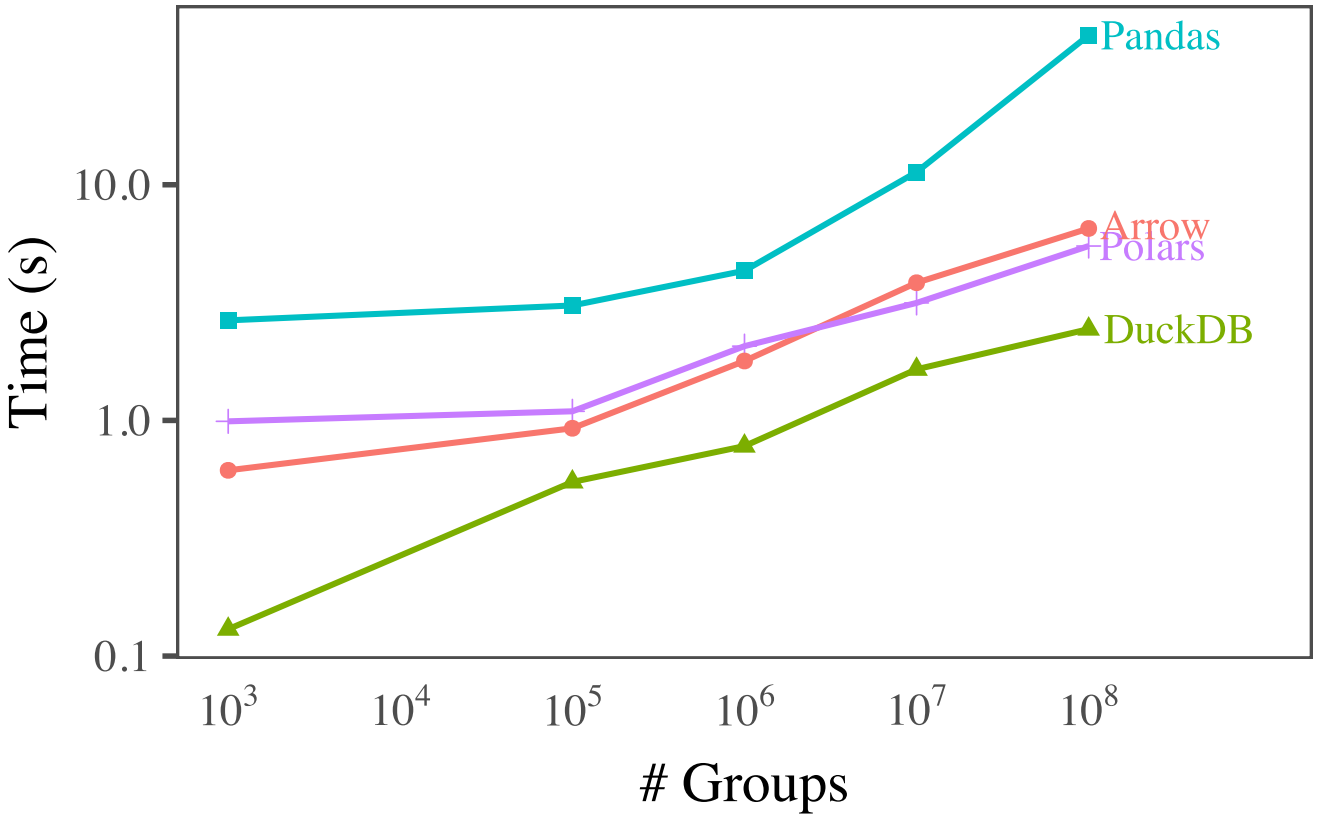
Because we are not interested in measuring the result set materialization time which would be significant for millions of groups, we follow the aggregation with an operator that only retrieves the first row. This does not change the complexity of the aggregation at all, since it needs to collect all data before producing even the first result row, since there might be data in the very last input data row that changes results for the first result. Of course this would be fairly unrealistic in practice, but it should nicely isolate the behavior of the aggregation operator only, since a `head(1)` operation on three columns should be fairly cheap and constant in execution time.



We measure the elapsed wall clock time required to complete each aggregation. To account for minor variation, we repeat each measurement three times and report the median time required. All experiments were run on a 2021 MacBook Pro with a ten-core M1 Max processor and 64 GB of RAM. Our data generation benchmark script is available online and we invite interested readers to re-run the experiment on their machines.



Now let's discuss some results. We start with varying the amount of rows in the table between one million and 100 millions. We repeat the experiment for both a fixed (small) group count of 1000 and when the amount of groups is equal to the amount of rows. Results are plotted as a *log-log plot*, we can see how DuckDB consistently outperforms the other systems, with the single-threaded Pandas being slowest, Polars and Arrow being generally similar.



For the next experiment, we fix the amount of rows at 100M (the largest size we experimented with) and show the full behavior when increasing the group size. We can see again how DuckDB consistently exhibits good scaling behavior when increasing group size, because it can effectively parallelize all phases of aggregation as outlined above. If you are interested in how we generated those plots, the plotting script is available, too.

Conclusion

Data analysis pipelines using mostly aggregation spend the vast majority of their execution time in the aggregate hash table, which is why it is worth spending an ungodly amount of human time optimizing them. We have some ideas for future work on this, for example we would like to extend our work when comparing sorting keys to comparing groups in the aggregate hash table. We also would like to add capabilities of dynamically choosing the amount of partitions a thread uses based on dynamic observation of the created hash table, e.g., if partitions are imbalanced we could use more bits to do so. Another large area of future work is to make our aggregate hash table work with out-of-core operations, where an individual hash table no longer fits in memory, this is particularly problematic when merging. And of course there are always opportunities to fine-tune an aggregation operator, and we are continuously improving DuckDBs aggregation operator.

If you want to work on cutting edge data engineering like this that will be used by thousands of people, consider contributing to DuckDB or join us at DuckDB Labs in Amsterdam!

Friendlier SQL with DuckDB

Publication date: 2022-05-04

Author: Alex Monahan

TL;DR: DuckDB offers several extensions to the SQL syntax. For a full list of these features, see the Friendly SQL documentation page.

An elegant user experience is a key design goal of DuckDB. This goal guides much of DuckDB's architecture: it is simple to install, seam-

less to integrate with other data structures like Pandas, Arrow, and R Dataframes, and requires no dependencies. Parallelization occurs automatically, and if a computation exceeds available memory, data is gracefully buffered out to disk. And of course, DuckDB's processing speed makes it easier to get more work accomplished.

However, SQL is not famous for being user-friendly. DuckDB aims to change that! DuckDB includes both a Relational API for dataframe-style computation, and a highly Postgres-compatible version of SQL. If you prefer dataframe-style computation, we would love your feedback on our roadmap. If you are a SQL fan, read on to see how DuckDB is bringing together both innovation and pragmatism to make it easier to write SQL in DuckDB than anywhere else. Please reach out on GitHub or Discord and let us know what other features would simplify your SQL workflows. Join us as we teach an old dog new tricks!

SELECT * EXCLUDE

A traditional SQL `SELECT` query requires that requested columns be explicitly specified, with one notable exception: the `*` wildcard. `SELECT *` allows SQL to return all relevant columns. This adds tremendous flexibility, especially when building queries on top of one another. However, we are often interested in *almost* all columns. In DuckDB, simply specify which columns to `EXCLUDE`:

```
SELECT * EXCLUDE (jar_jar_binks, midichlorians) FROM star_wars;
```

Now we can save time repeatedly typing all columns, improve code readability, and retain flexibility as additional columns are added to underlying tables.

DuckDB's implementation of this concept can even handle exclusions from multiple tables within a single statement:

```
SELECT
    sw.* EXCLUDE (jar_jar_binks, midichlorians),
    ff.* EXCLUDE cancellation
FROM star_wars sw, firefly ff;
```

SELECT * REPLACE

Similarly, we often wish to use all of the columns in a table, aside from a few small adjustments. This would also prevent the use of `*` and require a list of all columns, including those that remain unedited. In DuckDB, easily apply changes to a small number of columns with `REPLACE`:

```
SELECT
    * REPLACE (movie_count+3 AS movie_count, show_count*1000 AS show_count)
FROM star_wars_owned_by_disney;
```

This allows views, CTE's, or sub-queries to be built on one another in a highly concise way, while remaining adaptable to new underlying columns.

GROUP BY ALL

A common cause of repetitive and verbose SQL code is the need to specify columns in both the `SELECT` clause and the `GROUP BY` clause. In theory this adds flexibility to SQL, but in practice it rarely adds value. DuckDB now offers the `GROUP BY` we all expected when we first learned SQL – just `GROUP BY ALL` columns in the `SELECT` clause that aren't wrapped in an aggregate function!

```
SELECT
    systems,
    planets,
    cities,
    cantinas,
    sum(scum + villainy) AS total_scum_and_villainy
FROM star_wars_locations
GROUP BY ALL;
-- GROUP BY systems, planets, cities, cantinas
```

Now changes to a query can be made in only one place instead of two! Plus this prevents many mistakes where columns are removed from a SELECT list, but not from the GROUP BY, causing duplication.

Not only does this dramatically simplify many queries, it also makes the above EXCLUDE and REPLACE clauses useful in far more situations. Imagine if we wanted to adjust the above query by no longer considering the level of scum and villainy in each specific cantina:

```
SELECT
    * EXCLUDE (cantinas, booths, scum, villainy),
    sum(scum + villainy) AS total_scum_and_villainy
FROM star_wars_locations
GROUP BY ALL;
-- GROUP BY systems, planets, cities
```

Now that is some concise and flexible SQL! How many of your GROUP BY clauses could be re-written this way?

ORDER BY ALL

Another common cause for repetition in SQL is the ORDER BY clause. DuckDB and other RDBMSs have previously tackled this issue by allowing queries to specify the numbers of columns to ORDER BY (For example, ORDER BY 1, 2, 3). However, frequently the goal is to order by all columns in the query from left to right, and maintaining that numeric list when adding or subtracting columns can be error prone. In DuckDB, simply ORDER BY ALL:

```
SELECT
    age,
    sum(civility) AS total_civility
FROM star_wars_universe
GROUP BY ALL
ORDER BY ALL;
-- ORDER BY age, total_civility
```

This is particularly useful when building summaries, as many other client tools automatically sort results in this manner. DuckDB also supports ORDER BY ALL DESC to sort each column in reverse order, and options to specify NULLS FIRST or NULLS LAST.

Column Aliases in WHERE / GROUP BY / HAVING

In many SQL dialects, it is not possible to use an alias defined in a SELECT clause anywhere but in the ORDER BY clause of that statement. This commonly leads to verbose CTE's or subqueries in order to utilize those aliases. In DuckDB, a non-aggregate alias in the SELECT clause can be immediately used in the WHERE and GROUP BY clauses, and aggregate aliases can be used in the HAVING clause, even at the same query depth. No subquery needed!

```
SELECT
    only_imperial_storm_troopers_are_so_precise AS nope,
    turns_out_a_parsec_is_a_distance AS very_speedy,
    sum(mistakes) AS total_oops
FROM oops
WHERE
    nope = 1
GROUP BY
    nope,
    very_speedy
HAVING
    total_oops > 0;
```

Case Insensitivity While Maintaining Case

DuckDB allows queries to be case insensitive, while maintaining the specified case as data flows into and out of the system. This simplifies queries within DuckDB while ensuring compatibility with external libraries.

```
CREATE TABLE mandalorian AS SELECT 1 AS "THIS_IS_THE_WAY";
SELECT this_is_the_way FROM mandalorian;
```

THIS_IS_THE_WAY
1

Friendly Error Messages

Regardless of expertise, and despite DuckDB's best efforts to understand our intentions, we all make mistakes in our SQL queries. Many RDBMSs leave you trying to use the force to detect an error. In DuckDB, if you make a typo on a column or table name, you will receive a helpful suggestion about the most similar name. Not only that, you will receive an arrow that points directly to the offending location within your query.

```
SELECT * FROM star_trek;
```

```
Error: Catalog Error: Table with name star_trek does not exist!
Did you mean "star_wars"?
LINE 1: SELECT * FROM star_trek;
                      ^
```

(Don't worry, ducks and duck-themed databases still love some Trek as well).

DuckDB's suggestions are even context specific. Here, we receive a suggestion to use the most similar column from the table we are querying.

```
SELECT long_ago FROM star_wars;
```

```
Error: Binder Error: Referenced column "long_ago" not found in FROM clause!
Candidate bindings: "star_wars.long_long_ago"
LINE 1: SELECT long_ago FROM star_wars;
               ^
```

String Slicing

Even as SQL fans, we know that SQL can learn a thing or two from newer languages. Instead of using bulky `SUBSTRING` functions, you can slice strings in DuckDB using bracket syntax. As a note, SQL is required to be 1-indexed, so that is a slight difference from other languages (although it keeps DuckDB internally consistent and similar to other DBs).

```
SELECT 'I love you! I know'[:-3] AS nearly_soloed;
```

nearly_soloed
I love you! I k

Simple List and Struct Creation

DuckDB provides nested types to allow more flexible data structures than the purely relational model would allow, while retaining high performance. To make them as easy as possible to use, creating a `LIST` (array) or a `STRUCT` (object) uses simpler syntax than other SQL systems. Data types are automatically inferred.

```
SELECT
    ['A-Wing', 'B-Wing', 'X-Wing', 'Y-Wing'] AS starfighter_list,
    {name: 'Star Destroyer', common_misconceptions: 'Can''t in fact destroy a star'} AS star_destroyer_
facts;
```

List Slicing

Bracket syntax may also be used to slice a `LIST`. Again, note that this is 1-indexed for SQL compatibility.

```
SELECT
    starfighter_list[2:2] AS dont_forget_the_b_wing
FROM (SELECT ['A-Wing', 'B-Wing', 'X-Wing', 'Y-Wing'] AS starfighter_list);
```

dont_forget_the_b_wing
[B-Wing]

Struct Dot Notation

Use convenient dot notation to access the value of a specific key in a DuckDB STRUCT column. If keys contain spaces, double quotes can be used.

```
SELECT
    planet.name,
    planet."Amount of sand"
FROM (SELECT {name: 'Tatooine', 'Amount of sand': 'High'} AS planet);
```

Trailing Commas

Have you ever removed your final column from a SQL `SELECT` and been met with an error, only to find you needed to remove the trailing comma as well!? Never? Ok, Jedi... On a more serious note, this feature is an example of DuckDB's responsiveness to the community. In under 2 days from seeing this issue in a tweet (not even about DuckDB!), this feature was already built, tested, and merged into the primary branch. You can include trailing commas in many places in your query, and we hope this saves you from the most boring but frustrating of errors!

```
SELECT
    x_wing,
    proton_torpedoes,
    --targeting_computer
FROM luke_whats_wrong
GROUP BY
    x_wing,
    proton_torpedoes,
;
```

Function Aliases from Other Databases

For many functions, DuckDB supports multiple names in order to align with other database systems. After all, ducks are pretty versatile – they can fly, swim, and walk! Most commonly, DuckDB supports PostgreSQL function names, but many SQLite names are supported, as well as some from other systems. If you are migrating your workloads to DuckDB and a different function name would be helpful, please reach out – they are very easy to add as long as the behavior is the same! See our functions documentation for details.

```
SELECT
    'Use the Force, Luke'[:13] AS sliced_quote_1,
    substr('I am your father', 1, 4) AS sliced_quote_2,
    substring('Obi-Wan Kenobi, you''re my only hope', 17, 100) AS sliced_quote_3;
```

Auto-Increment Duplicate Column Names

As you are building a query that joins similar tables, you'll often encounter duplicate column names. If the query is the final result, DuckDB will simply return the duplicated column names without modifications. However, if the query is used to create a table, or nested in a subquery or Common Table Expression (where duplicate columns are forbidden by other databases!), DuckDB will automatically assign new names to the repeated columns to make query prototyping easier.

```
SELECT
    *
FROM (
    SELECT
        s1.tie_fighter,
        s2.tie_fighter
    FROM squadron_one s1
    CROSS JOIN squadron_two s2
    ) theyre_coming_in_too_fast;
```

tie_fighter	tie_fighter:1
green_one	green_two

Implicit Type Casts

DuckDB believes in using specific data types for performance, but attempts to automatically cast between types whenever necessary. For example, when joining between an integer and a varchar, DuckDB will automatically cast them to be the same type and complete the join successfully. A `List` or `IN` expression may also be created with a mixture of types, and they will be automatically cast as well. Also, `INTEGER` and `BIGINT` are interchangeable, and thanks to DuckDB's new storage compression, a `BIGINT` usually doesn't even take up any extra space! Now you can store your data as the optimal data type, but use it easily for the best of both!

```
CREATE TABLE sith_count_int AS SELECT 2::INTEGER AS sith_count;
CREATE TABLE sith_count_varchar AS SELECT 2::VARCHAR AS sith_count;

SELECT
    *
FROM sith_count_int s_int
JOIN sith_count_varchar s_char
  ON s_int.sith_count = s_char.sith_count;
```

sith_count	sith_count
2	2

Other Friendly Features

There are many other features of DuckDB that make it easier to analyze data with SQL!

DuckDB makes working with time easier in many ways, including by accepting multiple different syntaxes (from other databases) for the `INTERVAL` data type used to specify a length of time.

DuckDB also implements multiple SQL clauses outside of the traditional core clauses including the `SAMPLE` clause for quickly selecting a random subset of your data and the `QUALIFY` clause that allows filtering of the results of window functions (much like a `HAVING` clause does for aggregates).

The `DISTINCT ON` clause allows DuckDB to select unique combinations of a subset of the columns in a `SELECT` clause, while returning the first row of data for columns not checked for uniqueness.

Ideas for the Future

In addition to what has already been implemented, several other improvements have been suggested. Let us know if one would be particularly useful – we are flexible with our roadmap! If you would like to contribute, we are very open to PRs and you are welcome to reach out on GitHub or Discord ahead of time to talk through a new feature's design.

- Choose columns via regex
  - Decide which columns to select with a pattern rather than specifying columns explicitly
  - ClickHouse supports this with the `COLUMNS` expression
- Incremental column aliases
  - Refer to previously defined aliases in subsequent calculated columns rather than re-specifying the calculations
- Dot operators for JSON types
  - The JSON extension is brand new (see our documentation!) and already implements friendly `->` and `->>` syntax

Thanks for checking out DuckDB! May the Force be with you...

Range Joins in DuckDB

Publication date: 2022-05-27

Author: Richard Wesley

TL;DR: DuckDB has fully parallelized range joins that can efficiently join millions of range predicates.

Range intersection joins are an important operation in areas such as temporal analytics, and occur when two inequality conditions are present in a join predicate. Database implementations often rely on slow `O(N^2)` algorithms that compare every pair of rows for these operations. Instead, DuckDB leverages its fast sorting logic to implement two highly optimized parallel join operators for these kinds of range predicates, resulting in 20-30× faster queries. With these operators, DuckDB can be used effectively in more time-series-oriented use cases.

Introduction

Joining tables row-wise is one of the fundamental and distinguishing operations of the relational model. A join connects two tables horizontally using some Boolean condition called a *predicate*. This sounds straightforward, but how fast the join can be performed depends on the expressions in the predicate. This has lead to the creation of different join algorithms that are optimized for different predicate types.

In this post, we will explain several join algorithms and their capabilities. In particular, we will describe a newly added ”range join” algorithm that makes connecting tables on overlapping time intervals or multiple ordering conditions much faster.

Flight Data

No, this part isn't about ducks, but about air group flight statistics from the Battlestar Galactica reboot. We have a couple of tables we will be using: `Pilots`, `Crafts`, `Missions` and `Battles`. Some data was lost when the fleet dispersed, but hopefully this is enough to provide some ”real life” examples!

The `Pilots` table contains the pilots and their data that does not change (name, call sign, serial number):

id	callsign	name	serial
1	Apollo	Lee Adama	234567
2	Starbuck	Kara Thrace	462753
3	Boomer	Sharon Valeri	312743
4	Kat	Louanne Katraine	244977
5	Hotdog	Brendan Costanza	304871
6	Husker	William Adama	204971
...	...	...	...

The `Crafts` table contains all the various fighting craft (ignoring the ”Ship Of Theseus” problem of recycled parts!):

id	type	tailno
1	Viper	N7242C
2	Viper	2794NC
3	Raptor	312
4	Blackbird	N9999C
...	...	...

The `Missions` table contains all the missions flown by pilots. Missions have a `begin` and end time logged with the flight deck. We will use some common pairings (and an unusual mission at the end where Commander Adama flew his old Viper):

pid	cid	begin	end
2	2	3004-05-04 13:22:12	3004-05-04 15:05:49
1	2	3004-05-04 10:00:00	3004-05-04 18:19:12
3	3	3004-05-04 13:33:52	3004-05-05 19:12:21
6	1	3008-03-20 08:14:37	3008-03-20 10:21:15
...	...	...	...

The `Battles` table contains the time window of each battle with the Cylons.

battle	begin	end
Fall of the Colonies	3004-05-04 13:21:45	3004-05-05 02:47:16
Red Moon	3004-05-28 07:55:27	3004-05-28 08:12:19
Tylium Asteroid	3004-06-09 09:00:00	3004-06-09 11:14:29
Resurrection Ship	3004-10-28 22:00:00	3004-10-28 23:47:05
...	...	...

These last two tables (`Missions` and `Battles`) are examples of *state tables*. An object in a state table has a state that runs between two time points. For the battles, the state is just yes/no. For the missions, the state is a pilot/craft combination.

Equality Predicates

The most common type of join involves comparing one or more pairs of expressions for equality, often a primary key and a foreign key. For example, if we want a list of the craft flown by the pilots, we can join the `Pilots` table to the `Craft` table through the `Missions` table:

```
SELECT callsign, count(*), tailno
FROM Pilots p, Missions m, Crafts c
WHERE p.id = m.pid
  AND c.id = m.cid
GROUP BY ALL
ORDER BY 2 DESC;
```

This will give us a table like:

callsign	count(*)	tailno
Starbuck	127	2794NC
Boomer	55	R1234V
Apollo	3	N7242C
Husker	1	N7242C
...	...	...

Range Predicates

The thing to notice in this example is that the conditions joining the tables are equalities connected with ANDs. But relational joins can be defined using *any* Boolean predicate – even ones without equality or AND.

One common operation in temporal databases is intersecting two state tables. Suppose we want to find the time intervals when each pilot was engaged in combat so we can compute combat hours for seniority? Vipers are launched quickly, but not before the battle has started, and there can be malfunctions or pilots may be delayed getting to the flight deck.

```
SELECT callsign, battle,
    greatest(m.begin, b.begin) AS begin,
    least(m.end, b.end) AS end
FROM Pilots p, Missions m, Crafts c, Battles b
WHERE m.begin < b.end
  AND b.begin < m.end
  AND p.id = m.pid
  AND c.id = m.cid;
```

This join creates a set of records containing the call sign and period in combat for each pilot. It handles the case where a pilot returns for a new craft, excludes patrol flights, and even handles the situation when a patrol flight turns into combat! This is because intersecting state tables this way produces a *joint state table* – an important temporal database operation. Here are a few rows from the result:

callsign	battle	begin	end
Starbuck	Fall of the Colonies	3004-05-04 13:22:12	3004-05-04 15:05:49
Apollo	Fall of the Colonies	3004-05-04 13:21:45	3004-05-04 18:19:12
Boomer	Fall of the Colonies	3004-05-04 13:33:52	3004-05-05 02:47:16
...	...	...	...

Apollo was already in flight when the first Cylon attack came, so the query puts his `begin` time for the battle at the start of the battle, not when he launched for the decomissioning flyby. Starbuck and Boomer were scrambled after the battle started, but Boomer did not return until after the battle was effectively over, so her `end` time is moved back to the official end of the battle.

What is important here is that the join condition between the pilot/mission/craft relation and the battle table has no equalities in it. This kind of join is traditionally very expensive to compute, but as we will see, there are ways of speeding it up.

Infinite Time

One common problem with populating state tables is how to represent the open edges. For example, the begin time for the first state might not be known, or the current state may not have ended yet.

Often such values are represented by `NULL`s, but this complicates the intersection query because comparing with `NULL` yields `NULL`. This issue can be worked around by using `coalese(end, <large timestamp>)`, but that adds a computation to every row, most of

which don't need it. Another approach is to just use `<large timestamp>` directly instead of the `NULL`, which solves the expression computation problem but introduces an arbitrary time value. This value may give strange results when used in computations.

DuckDB provides a third alterantive from Postgres that can be used for these situations: infinite time values. Infinite time values will compare as expected, but arithmetic with them will produce `NULL`s or infinities, indicating that the computation is not well defined.

Common Join Algorithms

To see why these joins can be expensive, let's start by looking at the two most common join algorithms.

Hash Joins

Joins with at least one equality condition ANDed to the rest of the conditions are called *equi-joins*. They are usually implemented using a hash table like this:

```
hashes = {}
for b in build:
    hashes[b.pk] = b

result = []
for p in probe:
    result.append((p, hashes[p.fk], ))
```

The expressions from one side (the *build* side) are computed and hashed, then the corresponding expressions from the other side (the *probe* side) are looked up in the hash table and checked for a match.

We can modify this a bit when only *some* of the ANDed conditions are equalities by checking the other conditions once we find the equalities in the hash table. The important point is that we can use a hash table to make the join run time `O(N)`. This modification is a general technique that can be used with any join algorithm which reduces the possible matches.

Nested Loop Joins

Since relational joins can be defined using *any* Boolean predicate – even one without equality or AND, hash joins do not always work. The join algorithm of last resort in these situations is called a *Nested Loop Join* (or NLJ for short), and consists of just comparing every row from the probe side with every row from the build side:

```
result = []
for p in probe:
    for b in build
        if compare(p, b):
            result.append((p, b, ))
```

This is `O(M x N)` in the number of rows, which can be very slow if the tables are large. Even worse, most practical analytic queries (such as the combat hours example above) will not return anything like this many results, so a lot of effort may be wasted. But without an algorithm that is tuned for a kind of predicate, this is what we would have to use.

Range Joins

When we have a range comparison (one of <, <= >, >=) as one of the join conditions, we can take advantage of the ordering it implies by sorting the input relations on some of the join conditions. Sorting is `O(N log N)`, which suggests that this could be faster than an NLJ, and indeed this turns out to be the case.

Piecewise Merge Join

Before the advent of hash joins, databases would often sort the join inputs to find matches. For equi-joins, a repeated binary search would then find the matching values on the build side in `O(M log N)` time. This is called a *Merge Join*, and it runs faster than `O(M x N)`, but not as fast as the `O(N)` time of a hash join. Still, in the case where we have a single range comparison, the binary search lets us find the first match for a probe value. We can then find all the remaining matches by looking after the first one.

If we also sort the probe side, we can even know where to start the search for the next probe value because it will be after where we found the previous value. This is how *Piecewise Merge Join* (PWMJ) works: We sort the build side so that the values are ordered by the predicate (either ASC or DESC), then sort each probe chunk the same way so we can quickly scan through sets of values to find possible matches. This can be significantly faster than NLJ for these types of queries. If there are more join conditions, we can then check the generated matches to make sure all conditions are met because once again the sorting has significantly reduced the number of checks that have to be made.

Inequality Join (IEJoin)

For two range conditions (like the combat pay query), there are even faster algorithms available. We have recently added a new join called IEJoin, which sorts on two predicates to really speed things up.

The way that IEJoin works is to first sort both tables on the values for the first condition and merge the two sort keys into a combined table that tracks the two input tables' row numbers. Next, it sorts the positions in the combined table on the second range condition. It can then quickly scan for matches that pass both conditions. And just like for hash joins, we can check any remaining conditions because we have hopefully significantly reduced the number pairs we have to test.

Walk Through

Because the algorithm is a bit tricky, let's step through a small example. (If you are reading the paper, this is a simplified version of the ”Union Arrays” optimization from §4.3, but I find this version of the algorithm is much easier to understand than the version in §3.1.) We are going to look at Qp from the paper, which is a self join on the table ”West”:

West	t_id	time	cost	cores
s1	404	100	6	4
s2	498	140	11	2
s3	676	80	10	1
s4	742	90	5	4

We are looking for pairs of billing ids where the second id had a shorter time than the first, but a higher cost:

```
SELECT s1.t_id, s2.t_id AS t_id2
FROM west s1, west s2
WHERE s1.time > s2.time
  AND s1.cost < s2.cost;
```

There are two pairs that meet this criteria:

t_id	t_id2
404	676
742	676

(This is an example of another kind of double range query where we are looking for anomalies.)

First, we sort both input tables on the first condition key (`time`). (We sort DESC because we want the values to satisfy the join condition (>) from left to right.)

Because they are sorted the same way, we can merge the condition keys from the sorted tables into a new table called `L1` after marking each row with the table it came from (using negative row numbers to indicate the right table):

L1	s2	s2	s1	s1	s4	s4	s3	s3
time	140	140	100	100	90	90	80	80
cost	11	11	6	6	5	5	10	10
rid	1	-1	2	-2	3	-3	4	-4

The `rid` column lets us map rows in `L1` back to the original table.

Next, we build a second table `L2` with the second condition key (`cost`) and the row positions (P) of `L1` (not the row numbers from the original tables!) We sort `L2` on `cost` (DESC again this time because now we want the join condition to hold from right to left):

L2	s2	s2	s3	s3	s1	s1	s4	s4
cost	11	11	10	10	6	6	5	5
P	0	1	6	7	2	3	4	5

The sorted column of `L1` row positions is called the *permutation array*, and we can use it to find the corresponding position of the `time` value for a given `cost`.

At this point we have two tables (`L1` and `L2`), each sorted on one of the join conditions and pointing back to the tables it was derived from. Moreover, the sort orders have been chosen so that the condition holds from left to right (resp. right to left). Since the conditions are transitive, this means that whenever we have a value that satisfies a condition at a point in the table, it also satisfies it for everything to the right (resp. left)!

With this setup, we can scan `L2` from left to right looking for rows that match both conditions using two indexes:

- `i` iterates across `L2` from left to right;
- `off2` tracks `i` and is used to identify `costs` that satisfy the join condition compared to `i`. (Note that for loose inequalities, this could be to the right of `i`);

We use a bitmap B to track which rows in `L1` that the `L2` scan has already identified as satisfying the `cost` condition compared to the `L2` scan position `i`.

Because we only want matches between one left and one right row, we can skip matches where the `rid`s have different signs. To leverage this observation, we only process values of `i` that are in the left hand table (`rid[P[i]]` is positive), and we only mark bits for rows in the right hand table (`rid[P[i]]` is negative). In this example, the right side rows are the odd numbered values in P (which are conveniently also the odd values of `i`), which makes them easy to track in the example.

For the other rows, here is what happens:

i	off2	cost[i]	cost[off2]	P[i]	rid[P[i]]	B	Result
0	0	11	11	0	1	`00000000`	[]
2	0..2	10	11..10	6	4	`01000000`	[]
4	2..4	6	10..6	2	2	`01000001`	[{s4, s3}]
6	4..6	5	6..5	4	3	`01010001`	[{s1, s3}]

Whenever we find `costs` that satisfy the condition to the left of the scan location (between `off2` and `i`), we use `P[off2]` to mark the bits in B corresponding to those positions in `L1` that reference right side rows. This records that the `cost` condition is satisfied for those rows. Then whenever we have a position `P[i]` in `L1`, we can scan B to the right to find values that also satisfy the `cost` condition. This works because everything to the right of `P[i]` in `L1` satisfies the `price` condition thanks the sort order of `L1` and the transitivity of the comparison operations.

In more detail:

1. When `i` and `off2` are `0`, the `cost` condition < is not satisfied, so nothing happens;
2. When `i` is `1`, we are looking at a row from the right side of the join, so we skip it and move on;
3. When `i` is `2`, we are now looking at a row from the left side, so we bring `off2` forward until the `cost` condition fails, marking B where it succeeds at `P[1] = [1]`;
4. We then scan the `time` values in `L1` right from position `P[i=2] = 6` and find no matches in B;
5. When `i` is `4`, we bring `off2` forward again, marking B at `P[3] = [7]`;
6. We then scan `time` from position 2 and find matches at `[6,7]`, one of which (6) is from the right side table;
7. When `i` is `6`, we bring `off2` forward again, marking B at `P[5] = [3]`;
8. We then scan `time` from position 4 and again find matches at `[6,7]`;
9. Finally, when `i` runs off the end, we have no new `cost` values, so nothing happens;

What makes this fast is that we only have to check a few bits to find the matches. When we do need to perform comparisons, we can use the fast radix comparison code from our sorting code, which doesn't require special templated versions for every data type. This not only reduces the code size and complexity, it ”future-proofs” it against new data types.

Further Details

That walk through is a slightly simplified, single threaded version of the actual algorithm. There are a few more details that may be of interest:

- Scanning large, mostly empty bit maps can be slow, so we use the Bloom filter optimization from §4.2.
- The published algorithm assumes that there are no duplicate `L1` values in either table. To handle the general case, we use an exponential search to find the first `L1` value that satisfies the predicate with respect to the current position and scan right from that point;
- We also adapted the distributed Algorithm 3 from §5 by joining pairs of the sorted blocks generated by the sort code on separate threads. This allows us to fully parallelize the operator by first using parallel sorting and then by breaking up the join into independent pieces;
- Breaking up the pieces for parallel execution also allows us to spool join blocks that are not being processed to disk, making the join scalable.

Special Joins

One of the nice things about IEJoin is that it is very general and implements a number of more specialized join types reasonably efficiently. For example, the state intersection query above is an example of an *interval join* where we are looking to join on the intersection of two intervals.

Another specialized join that can be accelerated with `IEJoin` is a *band join*. This can be used to join values that are ”close” to each other

```
SELECT r.id, s.id
FROM r, s
WHERE r.value - s.value BETWEEN a AND b;
```

This translates into a double inequality join condition:

```
SELECT r.id, s.id
FROM r, s
WHERE s.value + a <= r.value AND r.value <= s.value + b;
```

which is exactly the type of join expression that IEJoin handles.

Performance

So how fast is the IEJoin? It is so fast that it is difficult to compare it to the previous range join algorithms because the improvements are so large that the other algorithms do not complete in a reasonable amount of time!

Simple Measurements

To give an example, here are the run times for a 100K self join of some employee tax and salary data, where the goal is to find the 1001 pairs of employees where one has a higher salary but the other has a higher tax rate:

```
SELECT
    r.id,
    s.id
FROM Employees r
JOIN Employees s
    ON r.salary < s.salary
    AND r.tax > s.tax;
```

Algorithm	Time (s)
NLJ	21.440
PWMJ	38.698
IEJoin	0.280

Another example is a self join to find 3772 overlapping events in a 30K event table:

```
SELECT
    r.id,
    s.id
FROM events r
JOIN events s
    ON r.start <= s.end
    AND r.end >= s.start
    AND r.id <> s.id;
```

Algorithm	Time (s)
NLJ	6.985
PWMJ	4.780
IEJoin	0.226

In both cases we see performance improvements of 20-100x, which is very helpful when you run a lot of queries like these!

Optimization Measurements

A third example demonstrates the importance of the join pair filtering and exponential search optimizations. The data is a state table of library circulation data from another interval join paper, and the query is a point-in-period temporal query used to generate Figure 4d:

```
SELECT x, count(*) AS y
FROM books,
    (SELECT x FROM range('2013-01-01'::TIMESTAMP, '2014-01-01'::TIMESTAMP, INTERVAL 1 DAY) tbl(x)) dates
WHERE checkout <= x AND x <= return
```

```
GROUP BY ALL
ORDER BY 1;
```

The result is a count of the number of books checked out at midnight on each day. These are the runtimes on an 18 core iMac Pro:

Improvement	Time	CPU
Unoptimized	> 30 m	~100%
Filtering	119.76 s	269%
Exponential	11.21 s	571%

The query joins a 35M row table with a 365 row table, so most of the data comes from the left hand side. By avoiding setting bits for the matching rows in the left table, we eliminate almost all `L1` checks. This dramatically reduces the runtime and improved the CPU utilization.

The data also has a large number of rows corresponding to books that were checked out at the start of the year, which all have the same `checkout` date. Searching left linearly in the first block to find the first match for the scan resulted in repeated runs of ~120K comparisons. This caused the runtime to be completely dominated by processing the first block. By reducing the number of comparisons for these rows from an average of ~60K to 16, the runtime dropped by a factor of 10 and the CPU utilization doubled.

Conclusion and Feedback

In this blog post, we explained the new DuckDB range join improvements provided by the new IEJoin operator. This should greatly improve the response time of state table joins and anomaly detection joins. We hope this makes your DuckDB experience even better – and please let us know if you run into any problems! Feel free to reach out on our GitHub page, or our Discord server.

Persistent Storage of Adaptive Radix Trees in DuckDB

Publication date: 2022-07-27

Author: Pedro Holanda

TL;DR: DuckDB uses Adaptive Radix Tree (ART) Indexes to enforce constraints and to speed up query filters. Up to this point, indexes were not persisted, causing issues like loss of indexing information and high reload times for tables with data constraints. We now persist ART Indexes to disk, drastically diminishing database loading times (up to orders of magnitude), and we no longer lose track of existing indexes. This blog post contains a deep dive into the implementation of ART storage, benchmarks, and future work. Finally, to better understand how our indexes are used, I'm asking you to answer the following survey. It will guide us when defining our future roadmap.

DuckDB uses ART Indexes to keep primary key (PK), foreign key (FK), and unique constraints. They also speed up point-queries, range queries (with high selectivity), and joins. Before the bleeding edge version (or V0.4.1, depending on when you are reading this post), DuckDB did not persist ART indexes on disk. When storing a database file, only the information about existing PKs and FKs would be stored, with all other indexes being transient and non-existing when restarting the database. For PKs and FKs, they would be fully reconstructed when reloading the database, creating the inconvenience of high-loading times.

A lot of scientific work has been published regarding ART Indexes, most notably on synchronization, cache-efficiency, and evaluation. However, up to this point, no public work exists on serializing and buffer managing an ART Tree. Some say that Hyper, the database in Tableau, persists ART indexes, but again, there is no public information on how that is done.

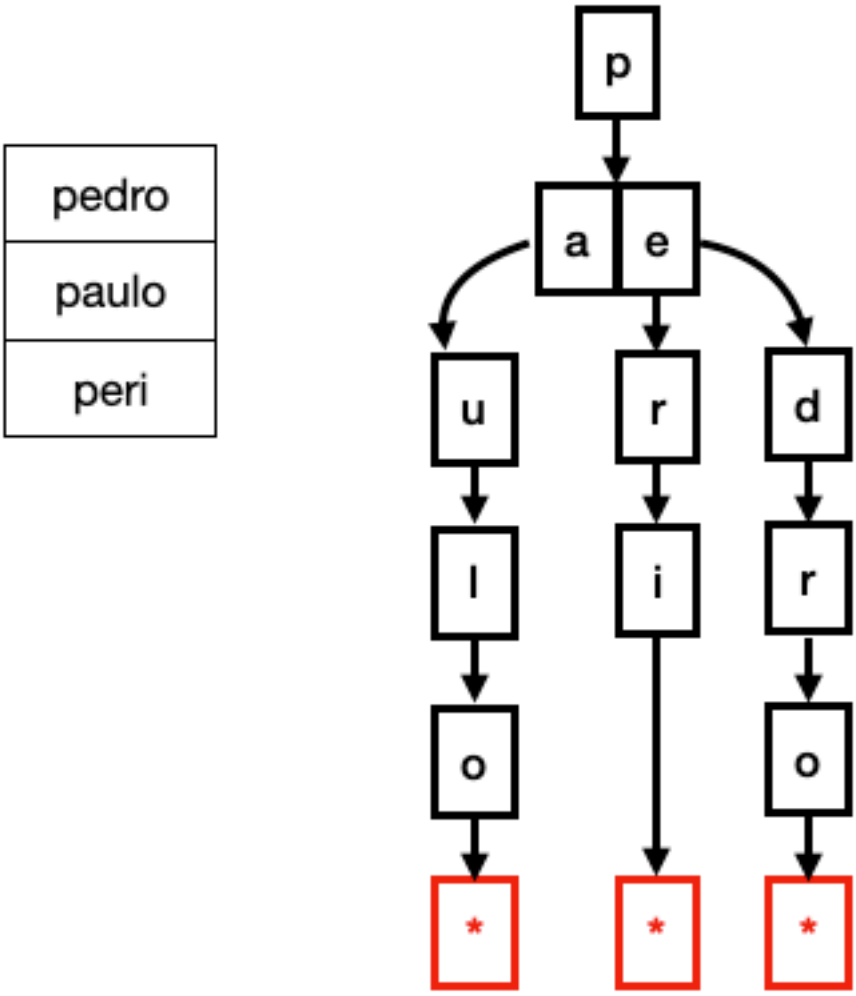
This blog post will describe how DuckDB stores and loads ART indexes. In particular, how the index is lazily loaded (i.e., an ART node is only loaded into memory when necessary). In the ART Index Section, we go through what an ART Index is, how it works, and some examples. In the ART in DuckDB Section, we explain why we decided to use an ART index in DuckDB where it is used and discuss the problems of not persisting ART indexes. In the ART Storage Section, we explain how we serialize and buffer manage ART Indexes in DuckDB. In the Benchmarks Section, we compare DuckDB v0.4.0 (before ART Storage) with the bleeding edge version of DuckDB. We demonstrate the difference in the loading costs of PKs and FKs in both versions and the differences between lazily loading an ART index and accessing a fully loaded ART Index. Finally, in the Road Map section, we discuss the drawbacks of our current implementations and the plans on the list of ART index goodies for the future.

ART Index

Adaptive Radix Trees are, in essence, Tries that apply vertical and horizontal compression to create compact index structures.

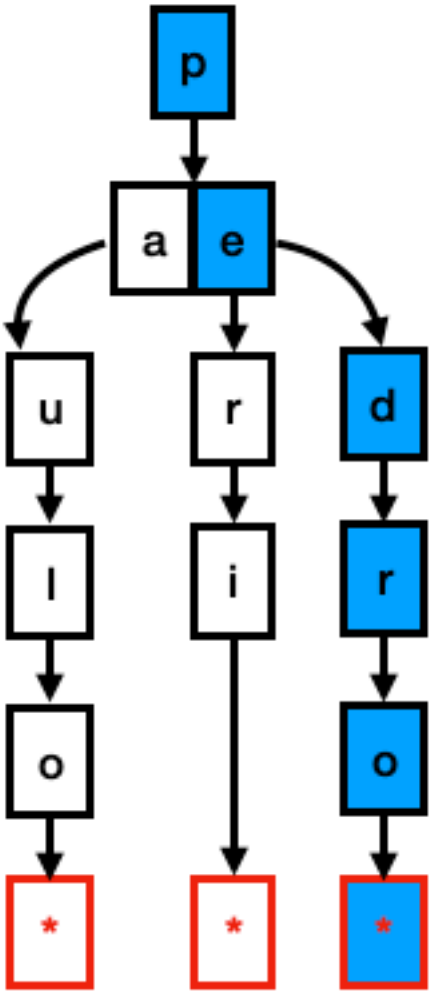
Trie

Tries are tree data structures, where each tree level holds information on part of the dataset. They are commonly exemplified with strings. In the figure below, you can see a Trie representation of a table containing the strings ”pedro”, ”paulo” and ”peri” The root node represents the first character ”p” with children ”a” (from paulo) and ”e” (from pedro and peri), and so on.



To perform lookups on a Trie, you must match each character of the key to the current level of the Trie. For example, if you search for pedro,

you must check the root contains the letter p. If it does, you check if any of its children contains the letter e, up to the point you reach a leaf node containing the pointer to the tuple that holds this string. (See figure below).



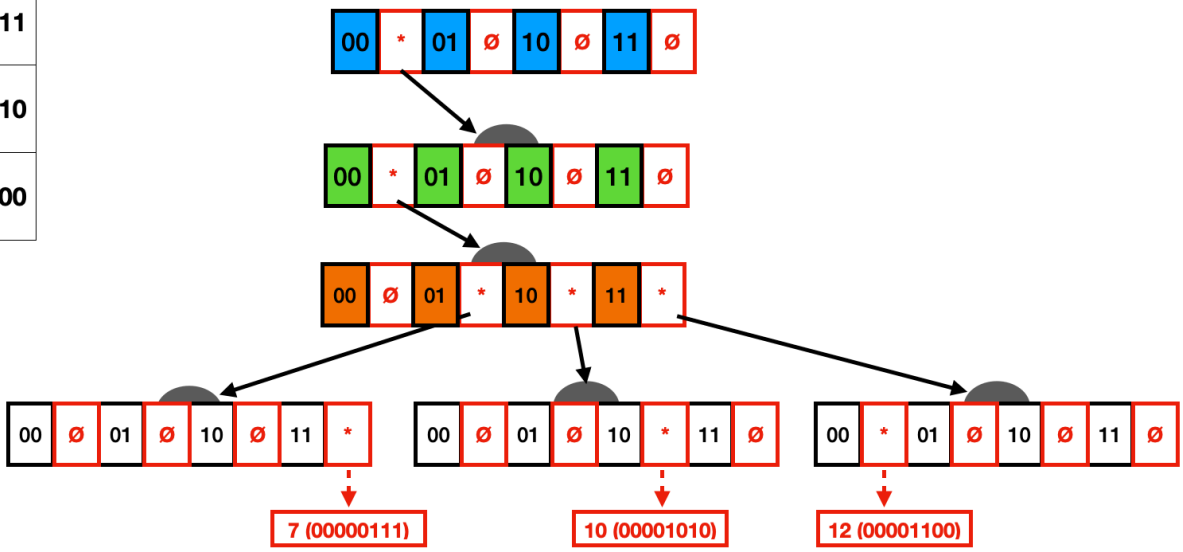
The main advantage of Tries is that they have O(k) lookups, meaning that in the worst case, the lookup cost will equal the length of the strings.

In reality, Tries can also be used for numeric data types. However, storing them character by character-like strings would be wasteful. Take, for example, the `UBIGINT` data type. In reality, `UBIGINT` is a `uint64_t` which takes 64 bits (i.e., 8 bytes) of space. The maximum value of a `uint64_t` is `18,446,744,073,709,551,615`. Hence if we represented it, like in the example above, we would need 17 levels on the Trie. In practice, Tries are created on a bit fan-out, which tells how many bits are represented per level of the Trie. A `uint64_t` Trie with 8-bit fan-out would have a maximum of 8 levels, each representing a byte.

To have more realistic examples, from this point onwards, all depictions in this post will be with bit representations. In DuckDB, the fan-out is always 8 bits. However, for simplicity, the following examples in this blog post will have a fan-out of 2 bits.

In the example below, we have a Trie that indexes the values 7, 10, and 12. You can also see the binary representation of each value on the table next to them. Each node consists of the bits 0 and 1, with a pointer next to them. This pointer can either be set (represented by `*`) or null (represented by ∅). Similar to the string Trie we had before, each level of the Trie will represent two bits, with the pointer next to these bits pointing to their children. Finally, the leaves point to the actual data.

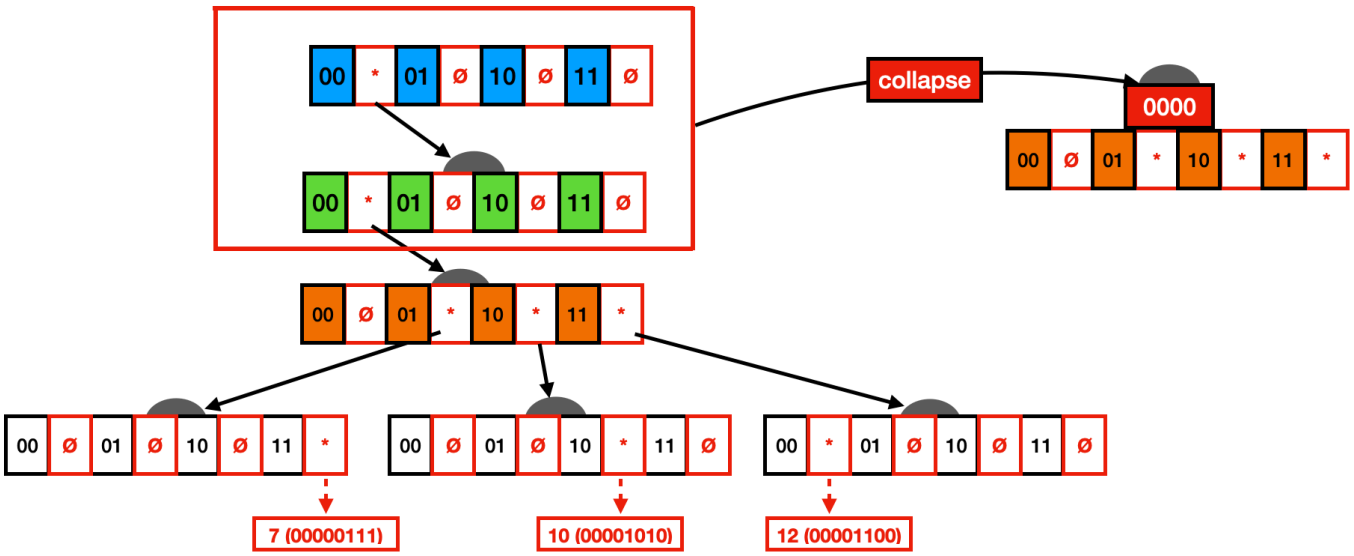
7	00000111
10	00001010
12	00001100



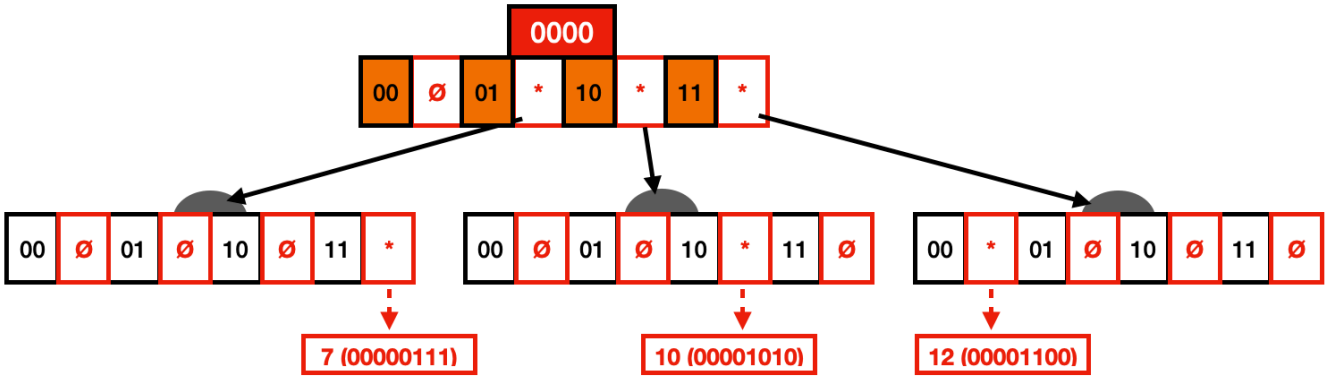
One can quickly notice that this Trie representation is wasteful on two different fronts. First, many nodes only have one child (i.e., one path), which could be collapsed by vertical compression (i.e., Radix Tree). Second, many nodes have null pointers, storing space without any information in them, which could be resolved with horizontal compression.

Vertical Compression (i.e., Radix Trees)

The basic idea of vertical compression is that we collapse paths with nodes that only have one child. To support this, nodes store a prefix variable containing the collapsed path to that node. You can see a representation of this in the figure below. For example, one can see that the first four nodes have only one child. These nodes can be collapsed to the third node (i.e., the first one that bifurcates) as a prefix path. When performing lookups, the key must match all values included in the prefix path.



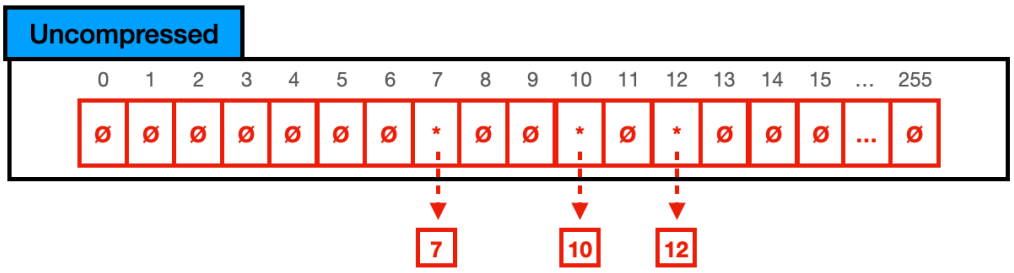
Below you can see the resulting Trie after vertical compression. This Trie variant is commonly known as a Radix Tree. Although a lot of wasted space has already been saved with this Trie variant, we still have many nodes with unset pointers.



Horizontal Compression (i.e., ART)

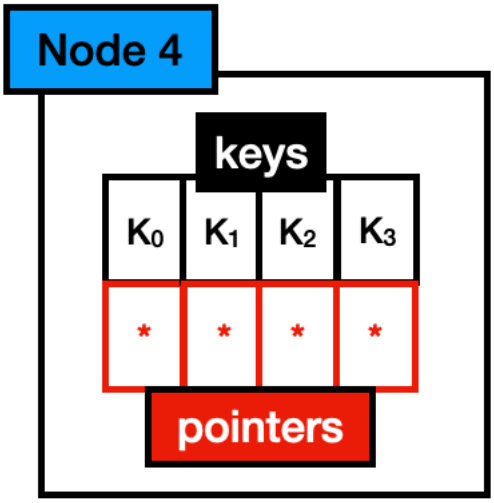
To fully understand the design decisions behind ART indexes, we must first extend the 2-bit fan-out to 8-bits, the commonly found fan-out for database systems.

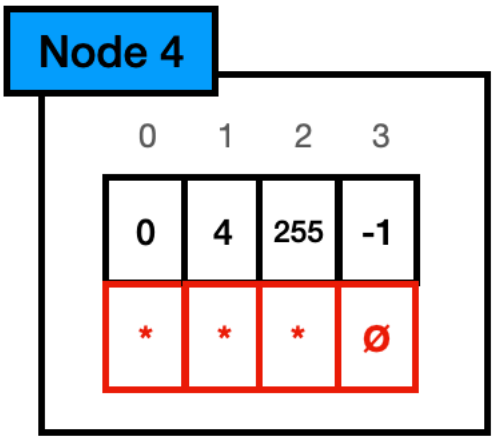
7	00000111
10	00001010
12	00001100



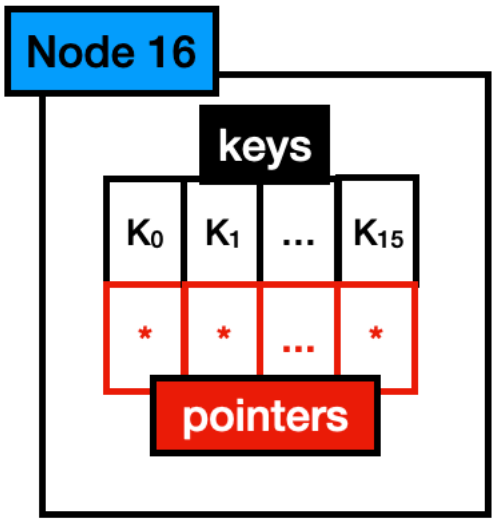
Below you can see the same nodes as before in a TRIE node of 8 bits. In reality, these nodes will store (2^8) 256 pointers, with the key being the array position of the pointer. In the case depicted by this example, we have a node with (256 pointers * 8 bytes) 2048 byte size while only actually utilizing 24 bytes (3 pointers * 8 bytes), which means that 2016 bytes are entirely wasted. To avoid this situation. ART indexes are composed of 4 different node types that depend on how full the current node is. Below I quickly describe each node with a graphical representation of them. In the graphical representation, I present a conceptual visualization of the node and an example with keys 0,4 and 255.

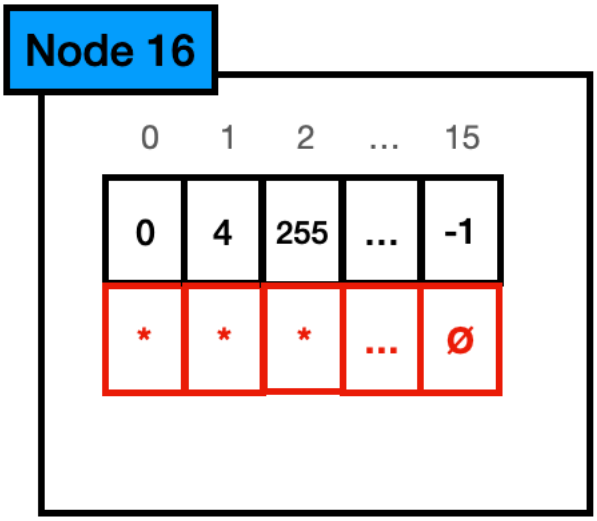
Node 4: Node 4 holds up to 4 different keys. Each key is stored in a one-byte array, with one pointer per key. With its total size being 40 bytes (4*1 + 4*8). Note that the pointer array is aligned with the key array (e.g., key 0 is in position 0 of the keys array, hence its pointer is in position 0 of the pointers array)



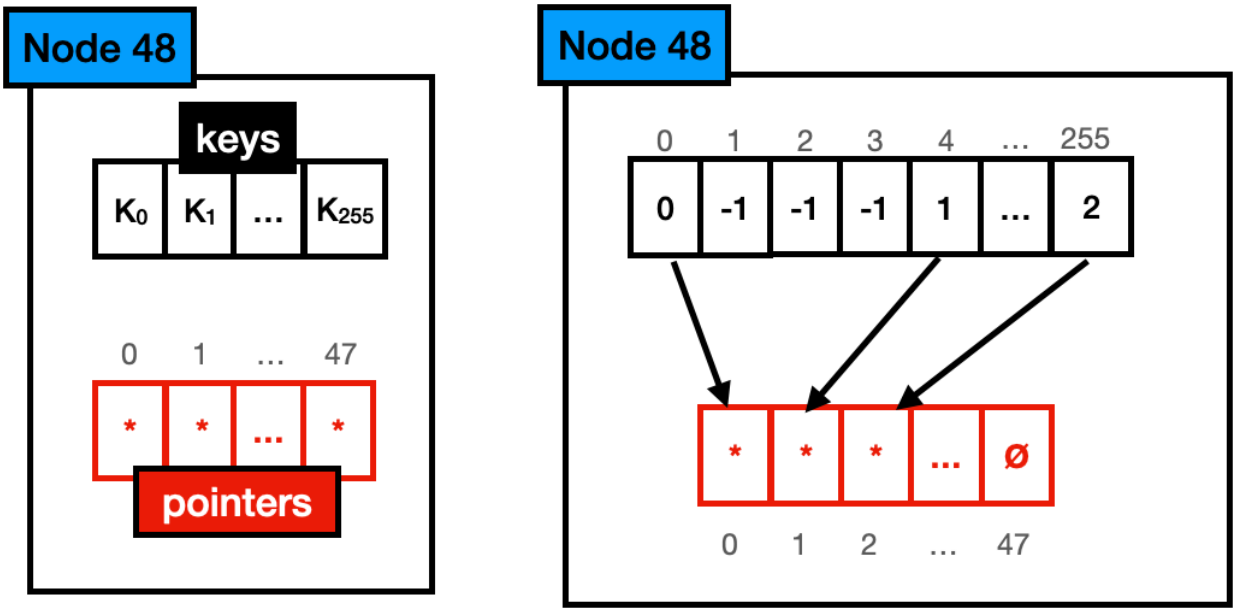


Node 16 : Node 16 holds up to 16 different keys. Like node 4, each key is stored in a one-byte array, with one pointer per key. With its total size being 144 bytes (16*1 + 16*8). Like Node 4, the pointer array is aligned with the key array.

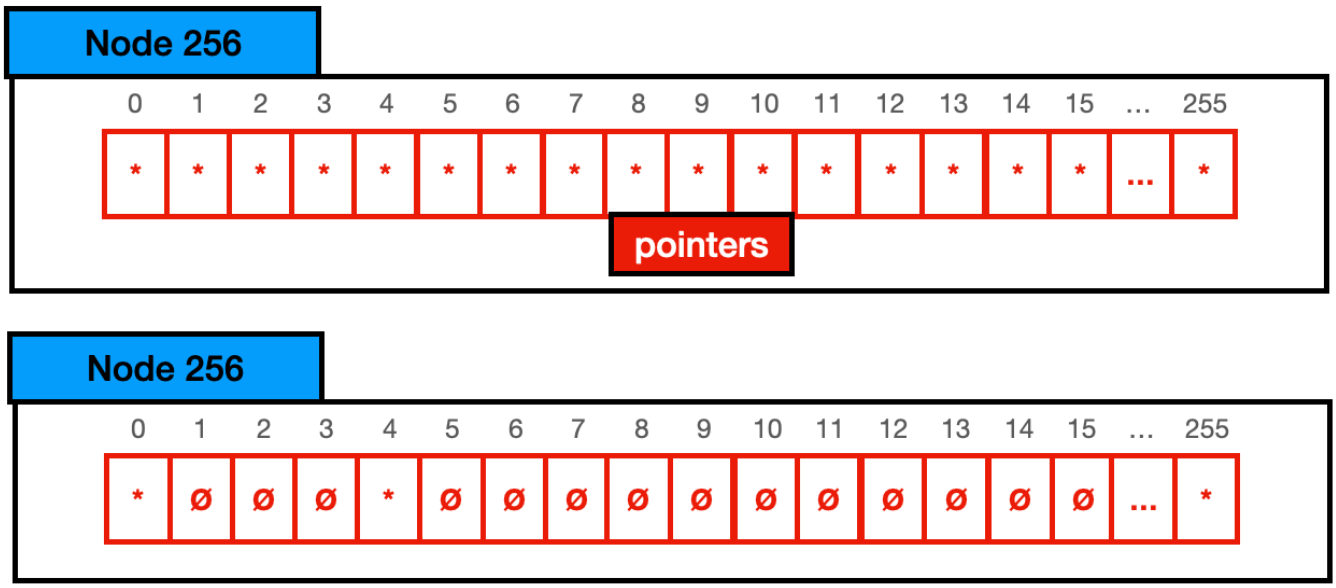




Node 48 : Node 48 holds up to 48 different keys. When a key is present in this node, the one-byte array position representing that key will hold an index into the pointer array that points to the child of that key. Its total size is 640 bytes (256*1 + 48*8). Note that the pointer array and the key array are not aligned anymore. The key array points to the position in the pointer array where the pointer of that key is stored (e.g., the key 255 in the key array is set to 2 because the position 2 of the pointer array points to the child pertinent to that key).

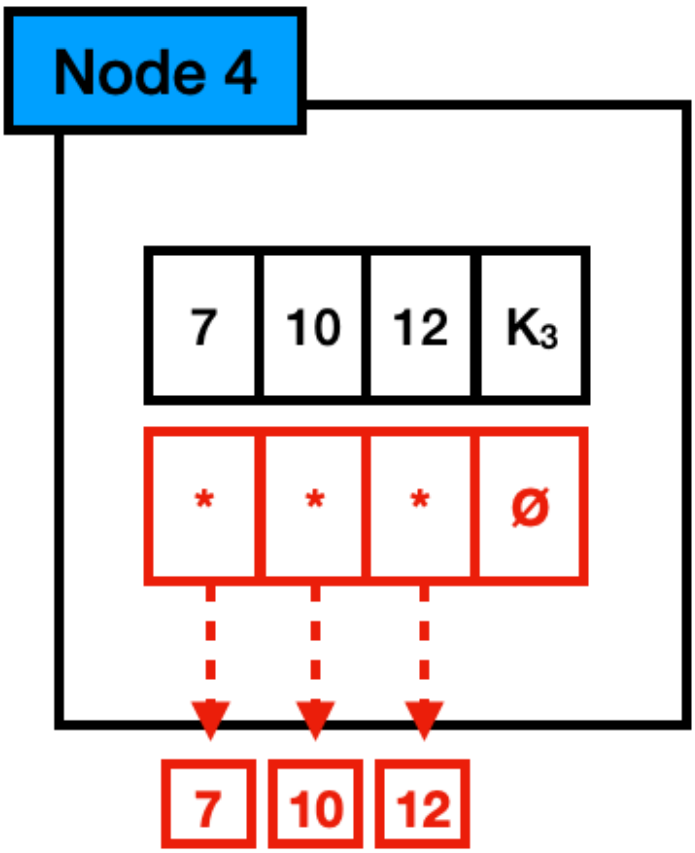


Node 256: Node 256 holds up to 256 different keys, hence all possible values in the distribution. It only has a pointer vector, if the pointer is set, the key exists, and it points to its child. Its total size is 2048 bytes (256 pointers * 8 bytes).



For the example in the previous section, we could use a Node 4 instead of a Node 256 to store the keys, since we only have 3 keys present. Hence it would look like the following:

7	00000111
10	00001010
12	00001100



ART in DuckDB

When considering which index structure to implement in DuckDB, we wanted a structure that could be used to keep PK/FK/Unique constraints while also being able to speed up range queries and Joins. Database systems commonly implement Hash-Tables for constraint checks and BP-Trees for range queries. However, we saw in ART Indexes an opportunity to diminish the code complexity by having one data structures for two use cases. The main characteristics that ART Index provides that we take advantage of are:

1. Compact Structure. Since the ART internal nodes are rather small, they can fit in CPU caches, being a more cache-conscious structure than BP-Trees.
2. Fast Point Queries. The worst case for an ART point query is O(k), which is sufficiently fast for constraint checking.
3. No dramatic regression on insertions. Many Hash-Table variants must be rebuilt when they reach a certain size. In practice, one insert might cause a significant regression in time, with a query suddenly taking orders of magnitude more time to complete, with no apparent reason for the user. In the ART, inserts might cause node growths (e.g., a Node 4 might grow to a Node 16), but these are inexpensive.
4. Ability to run range queries. Although the ART does not run range queries as fast as BP-Trees since it must perform tree traversals, where the BP-Tree can scan leaf nodes sequentially, it still presents an advantage over hash tables since these types of queries can be done (Some might argue that you can use Hash Tables for range queries, but meh). This allows us to efficiently use ART for highly selective range queries and index joins.
5. Maintainability. Using one structure for both constraint checks and range queries instead of two is more code efficient and maintainable.

What Is It Used For?

As said previously, ART indexes are mainly used in DuckDB on three fronts.

1. Data Constraints. Primary key, Foreign Keys, and Unique constraints are all maintained by an ART Index. When inserting data in a tuple with a constraint, this will effectively try to perform an insertion in the ART index and fail if the tuple already exists.

```
CREATE TABLE integers (i INTEGER PRIMARY KEY);
-- Insert unique values into ART
INSERT INTO integers VALUES (3), (2);
-- Insert conflicting value in ART will fail
INSERT INTO integers VALUES (3);

CREATE TABLE fk_integers (j INTEGER, FOREIGN KEY (j) REFERENCES integers(i));
-- This insert works normally
INSERT INTO fk_integers VALUES (2), (3);
-- This fails after checking the ART in integers
INSERT INTO fk_integers VALUES (4);
```

2. Range Queries. Highly selective range queries on indexed columns will also use the ART index underneath.

```
CREATE TABLE integers (i INTEGER PRIMARY KEY);
-- Insert unique values into ART
INSERT INTO integers VALUES (3), (2), (1), (8) , (10);
-- Range queries (if highly selective) will also use the ART index
SELECT * FROM integers WHERE i >= 8;
```

3. Joins. Joins with a small number of matches will also utilize existing ART indexes.

```
-- Optionally you can always force index joins with the following pragma
PRAGMA force_index_join;

CREATE TABLE t1 (i INTEGER PRIMARY KEY);
CREATE TABLE t2 (i INTEGER PRIMARY KEY);
-- Insert unique values into ART
INSERT INTO t1 VALUES (3), (2), (1), (8), (10);
INSERT INTO t2 VALUES (3), (2), (1), (8), (10);
-- Joins will also use the ART index
SELECT * FROM t1 INNER JOIN t2 ON (t1.i = t2.i);
```

4. Indexes over expressions. ART indexes can also be used to quickly look up expressions.

```
CREATE TABLE integers (i INTEGER, j INTEGER);

INSERT INTO integers VALUES (1, 1), (2, 2), (3, 3);

-- Creates index over the i + j expression
CREATE INDEX i_index ON integers USING ART((i + j));

-- Uses ART index point query
SELECT i FROM integers WHERE i + j = 2;
```

ART Storage

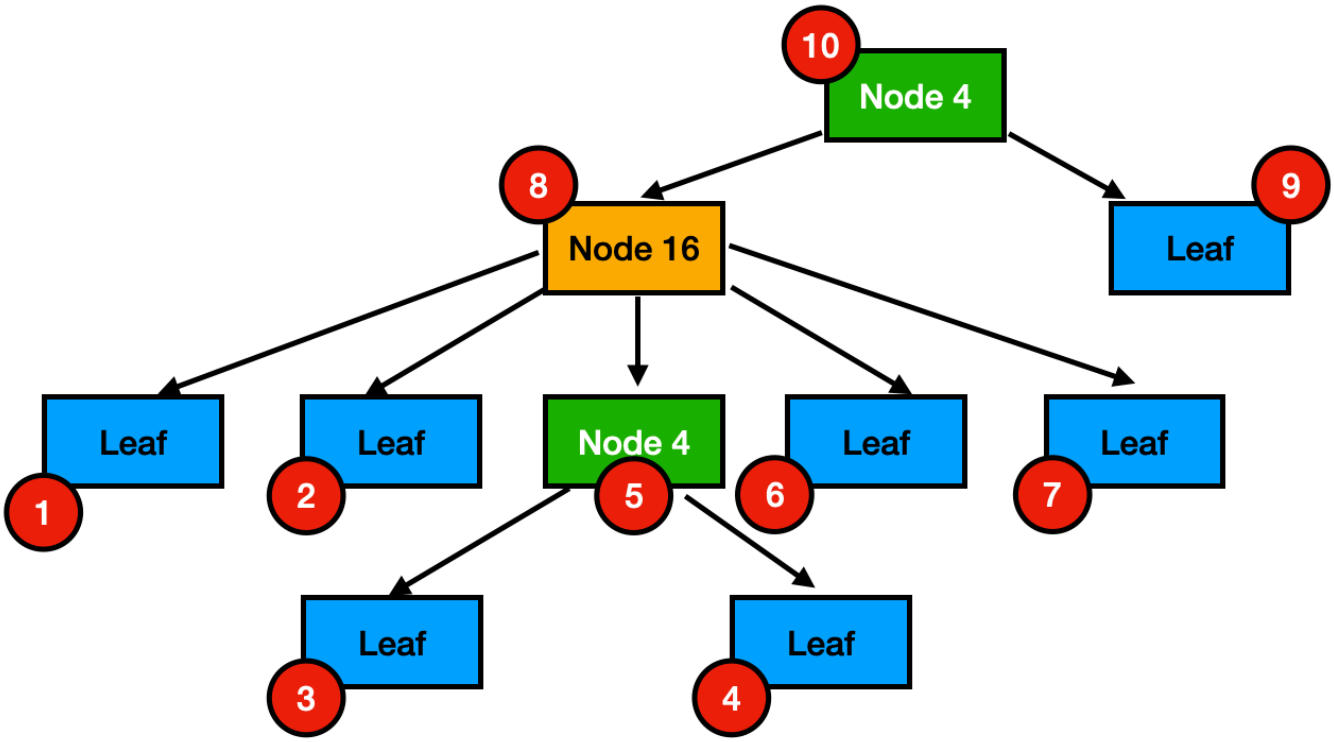
There are two main constraints when storing ART indexes:

1. The index must be stored in an order that allows for lazy-loading. Otherwise, we would have to fully load the index, including nodes that might be unnecessary to queries that would be executed in that session.
2. It must not increase the node size. Otherwise, we diminish the cache-conscious effectiveness of the ART index.

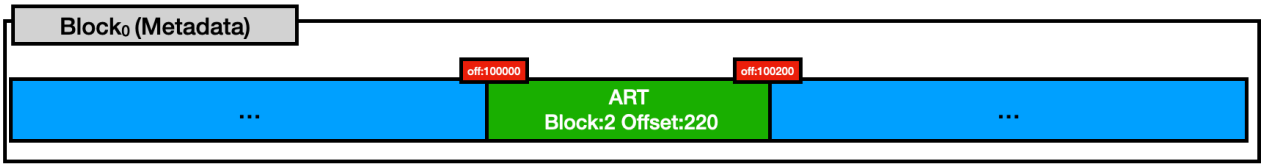
Post-Order Traversal

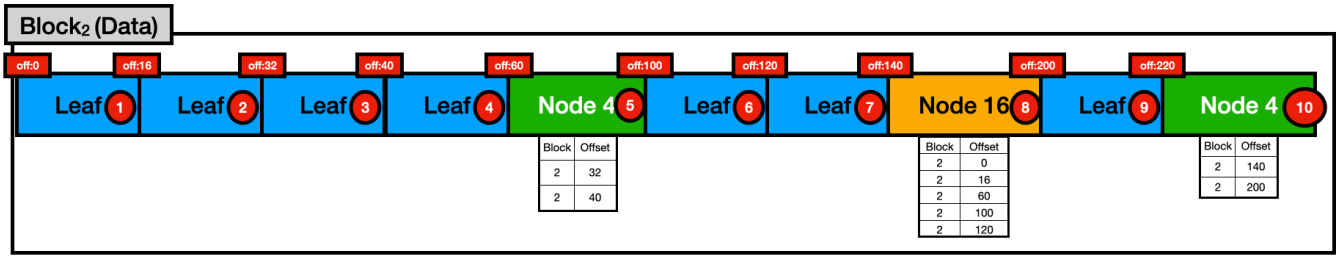
To allow for lazy loading, we must store all children of a node, collect the information of where each child is stored, and then, when storing the actual node, we store the disk information of each of its children. To perform this type of operation, we do a post-order traversal.

The post-order traversal is shown in the figure below. The circles in red represent the numeric order where the nodes will be stored. If we start from the root node (i.e., Node 4 with storage order 10), we must first store both children (i.e., Node 16 with storage order 8 and the Leaf with storage order 9). This goes on recursively for each of its children.



The figure below shows an actual representation of what this would look like in DuckDB's block format. In DuckDB, data is stored in 256 kB contiguous blocks, with some blocks reserved for metadata and some for actual data. Each block is represented by an `id`. To allow for navigation within a block, they are partitioned by byte offsets hence each block contains 256,000 different offsets





In this example, we have `Block 0` that stored some of our database metadata. In particular, between offsets 100,000 and 100,200 we store information pertinent to one ART index. This will store information on the index (e.g., name, constraints, expression) and the `<Block,Offset>` position of its root node.

For example, let's assume we are doing a lookup of the key with `row_ids` stored in the Leaf with storage order 1. We would start by loading the Art Root node on `<Block:2, Offset:220>`, by checking the keys stored in that Node, we would then see we must load the Node 16 at `<Block:2, Offset:140>`, and then finally our Leaf at `<Block:0, Offset:0>`. That means that for this lookup, only these

3 nodes were loaded into memory. Subsequent access to these nodes would only require memory access, while access to different nodes (e.g., Leaf storage order 2) would still result in disk access.

One major problem with implementing this (de)serialization process is that now we not only have to keep information about the memory address of pointers but also if they are already in memory and if not, what's the `<Block,Offset>` position they are stored.

If we stored the Block Id and Offset in new variables, it would dramatically increase the ART node sizes, diminishing its effectiveness as a cache-conscious data structure.

Take Node 256 as an example. The cost of holding 256 pointers is 2048 bytes (256 pointers * 8 bytes). Let's say we decide to store the Block Information on a new array like the following:

```
struct BlockPointer {
    uint32_t block_id;
    uint32_t offset;
}

class Node256 : public Node  {
    // Pointers to the child nodes
    Node* children[256];
    BlockPointer block_info[256];
}
```

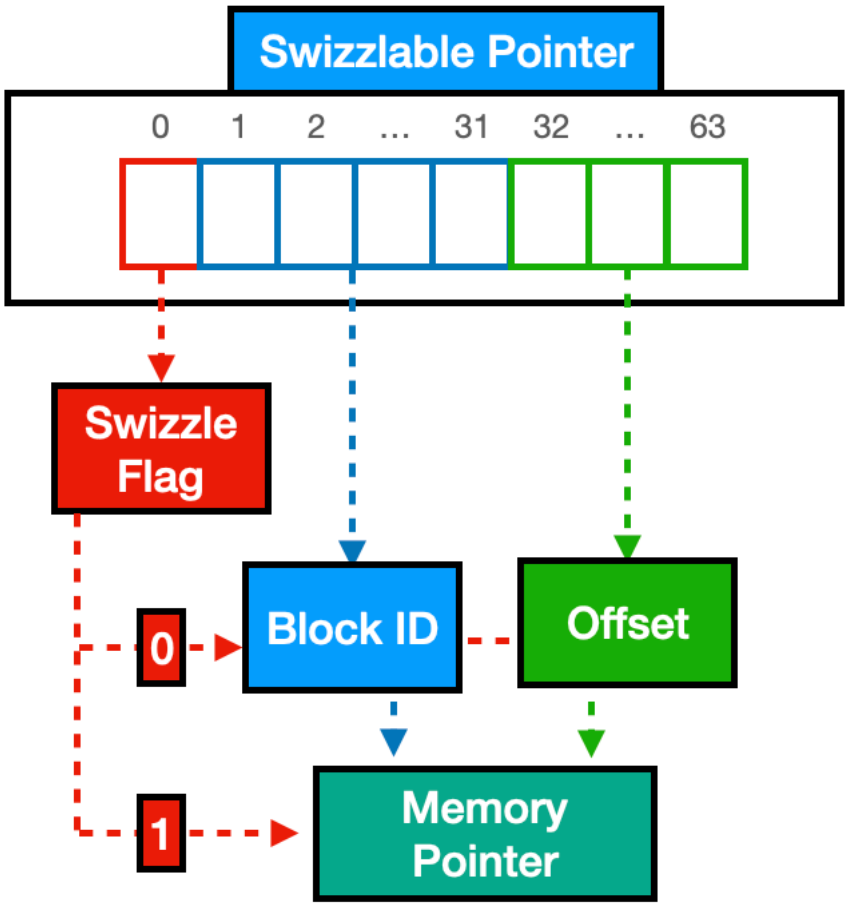
Node 256 would increase 2048 bytes (256 * (4+4)), causing it to double its current size to 4096 bytes.

Pointer Swizzling

To avoid the increase in sizes of ART nodes, we decided to implement Swizzlable Pointers and use them instead of regular pointers.

The idea is that we don't need all 64 bits (i.e., 48 bits give you an address space of 256 terabyte, supporting any of the current architectures, see the related discussion on Stack Overflow and the "64-bit computing" Wikipedia page) in a pointer to point to a memory address. Hence we can use the most significant bit as a flag (i.e., the Swizzle Flag). If the swizzle flag is set, the value in our Swizzlable Pointer is a memory address for the Node. Otherwise, the variable stores the Block Information of where the Node is stored. In the latter case, we use the following 31 bits to store the Block ID and the remaining 32 bits to store the offset.

In the following figure, you can see a visual representation of DuckDB's Swizzlable Pointer.



Benchmarks

To evaluate the benefits and disadvantages of our current storage implementation, we run a benchmark (Available at this Colab Link), where we create a table containing 50,000,000 integral elements with a primary key constraint on top of them.

```
con = duckdb.connect("vault.db")
con.execute("CREATE TABLE integers (x INTEGER PRIMARY KEY)")
con.execute("INSERT INTO integers SELECT * FROM range(50000000)")
```

We run this benchmark on two different versions of DuckDB, one where the index is not stored (i.e., v0.4.0), which means it is always in memory and fully reconstructed at a database restart, and another one where the index is stored (i.e., bleeding-edge version), using the lazy-loading technique described previously.

Storing Time

We first measure the additional cost of serializing our index.

```
cur_time = time.time()
con.close()
print("Storage time: " + str(time.time() - cur_time))
```

Storage Time

Name	Time (s)
Reconstruction	8.99
Storage	18.97

We can see storing the index is 2x more expensive than not storing the index. The reason is that our table consists of one column with 50,000,000 `int32_t` values. However, when storing the ART, we also store 50,000,000 `int64_t` values for their respective `row_ids` in the leaves. This increase in the elements is the main reason for the additional storage cost.

Load Time

We now measure the loading time of restarting our database.

```
cur_time = time.time()
con = duckdb.connect("vault.db")
print("Load time: " + str(time.time() - cur_time))
```

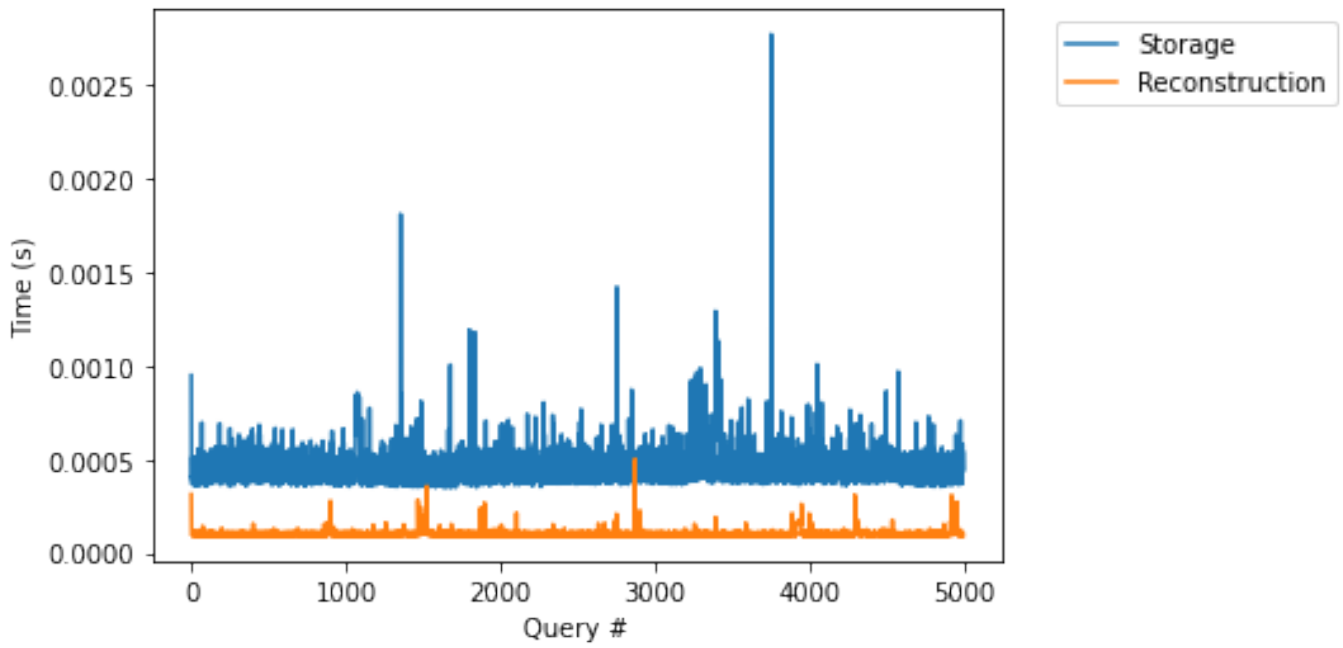
Name	Time (s)
Reconstruction	7.75
Storage	0.06

Here we can see a two-order magnitude difference in the times of loading the database. This difference is basically due to the complete reconstruction of the ART index during loading. In contrast, in the `Storage` version, only the metadata information about the ART index is loaded at this point.

Query Time (Cold)

We now measure the cold query time (i.e., the Database has just been restarted, which means that in the `Storage` version, the index does not exist in memory yet.) of running point queries on our index. We run 5000 point queries, equally spaced on 10000 elements in our distribution. We use this value to always force the point queries to load a large number of unused nodes.

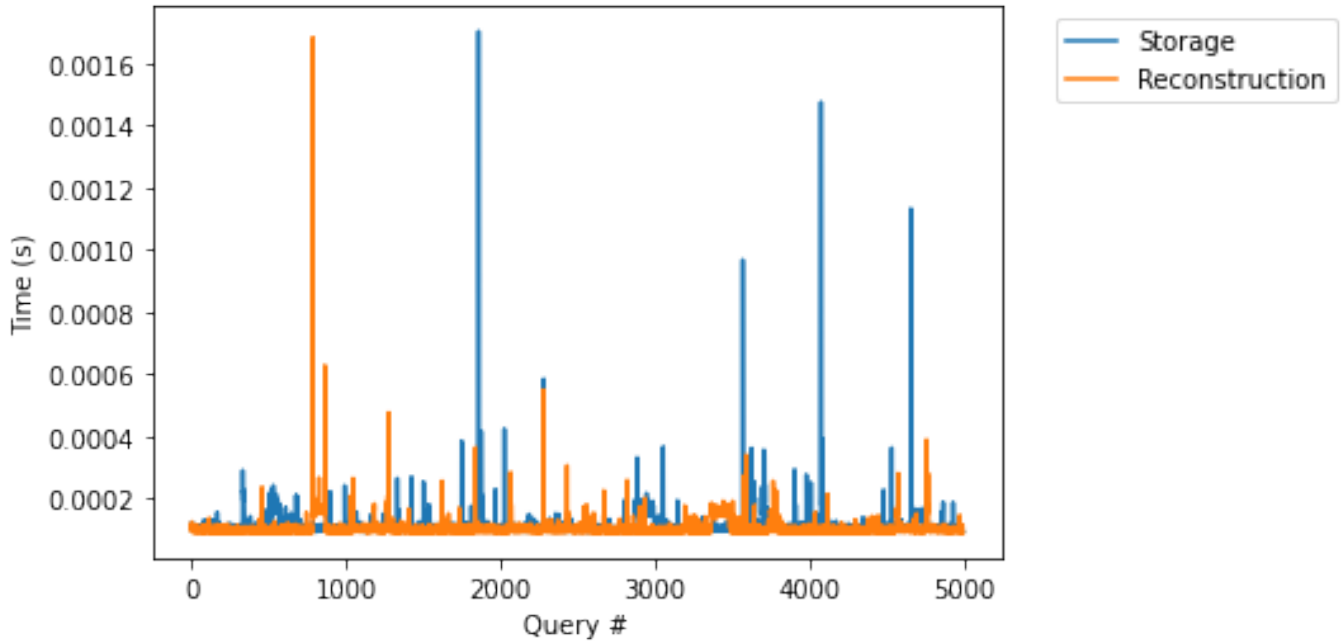
```
times = []
for i in range (0, 50000000, 10000):
  cur_time = time.time()
  con.execute("SELECT x FROM integers WHERE x = " + str(i))
  times.append(time.time() - cur_time)
```



In general, each query is 3x more expensive in the persisted storage format. This is due to two main reasons: 1) Creating the nodes. In the storage version, we do create the nodes lazily, which means that, for each node, all parameters must be allocated, and values like keys and prefixes are loaded. 2) Block Pinning. At every node, we must pin and unpin the blocks where they are stored.

Query Time (Hot)

In this experiment, we execute the same queries as in the previous section.



The times in both versions are comparable since all the nodes in the storage version are already set in memory. In conclusion, when stored indexes are in active use, they present similar performance to fully in-memory indexes.

Future Work

ART index storage has been a long-standing issue in DuckDB, with multiple users claiming it a missing feature that created an impediment for them to use DuckDB. Although now storing and lazily loading ART indexes is possible, there are many future paths we can still pursue to make the ART-Index more performant. Here I list what I believe are the most important next steps:

1. Caching Pinned Blocks. In our current implementation, blocks are constantly pinned and unpinned, even though blocks can store multiple nodes and are most likely reused continuously through lookups. Smartly caching them will result in drastic savings for queries that trigger node loading.
2. Bulk Loading. Our ART-Index currently does not support bulk loading. This means that nodes will be constantly resized when creating an index over a column since elements will be inserted one by one. If we bulk-load the data, we can know exactly which Nodes we must create for that dataset, hence avoiding these frequent resizes.
3. Bulk Insertion. When performing bulk insertion, a similar problem as bulk-loading would happen. A possible solution would be to create a new ART index with Bulk-Loading and then merge it with the existing Art Index
4. Vectorized Lookups/Inserts. DuckDB utilized a vectorized execution engine. However, both our ART lookups and inserts still follow a tuple-at-a-time model.
5. Updatable Index Storage. In our current implementation, ART-Indexes are fully invalidated from disk and stored again. This causes an unnecessary increase in storage time on subsequent storage. Updating nodes directly into disk instead of entirely rewriting the index could drastically decrease future storage costs. In other words, indexes are constantly completely stored at every checkpoint.
6. Combined Pointer/Row Id Leaves. Our current leaf node format allows for storing values that are repeated over multiple tuples. However, since ART indexes are commonly used to keep unique key constraints (e.g., Primary Keys), and a unique `row_id` fits in the same pointer size space, a lot of space can be saved by using the child pointers to point to the actual `row_id` instead of creating an actual leaf node that only stores one `row_id`.

Roadmap

> It's tough to make predictions, especially about the future
> – Yogi Berra

Art Indexes are a core part of both constraint enforcement and keeping access speed up in DuckDB. And as depicted in the previous section, there are many distinct paths we can take in our bag of ART goodies, with advantages for completely different use cases.

To better understand how our indexes are used, it would be extremely helpful if you could answer the following survey created by one of our MSc students.

Querying Postgres Tables Directly from DuckDB

Publication date: 2022-09-30

Author: Hannes Mühleisen

TL;DR: DuckDB can now directly query tables stored in PostgreSQL and speed up complex analytical queries without duplicating data.

Introduction

PostgreSQL is the world's most advanced open source database (self-proclaimed). From its interesting beginnings as an academic DBMS, it has evolved over the past 30 years into a fundamental workhorse of our digital environment.

PostgreSQL is designed for traditional transactional use cases, "OLTP", where rows in tables are created, updated and removed concurrently, and it excels at this. But this design decision makes PostgreSQL far less suitable for analytical use cases, "OLAP", where large chunks

of tables are read to create summaries of the stored data. Yet there are many use cases where both transactional and analytical use cases are important, for example when trying to gain the latest business intelligence insights into transactional data.

There have been some attempts to build database management systems that do well on both workloads, "HTAP", but in general many design decisions between OLTP and OLAP systems are hard trade-offs, making this endeavour difficult. Accepting that one size does not fit all after all, systems are often separated, with the transactional application data living in a purpose-built system like PostgreSQL, and a copy of the data being stored in an entirely different DBMS. Using a purpose-built analytical system speeds up analytical queries by several orders of magnitude.

Unfortunately, maintaining a copy of the data for analytical purposes can be problematic: The copy will immediately be outdated as new transactions are processed, requiring a complex and non-trivial synchronization setup. Storing two copies of the database also will require twice the storage space. For example, OLTP systems like PostgreSQL traditionally use a row-based data representation, and OLAP systems tend to favor a chunked-columnar data representation. You can't have both without maintaining a copy of the data with all the issues that brings with it. Also, the SQL syntaxes between whatever OLAP system you're using and Postgres may differ quite significantly.

But the design space is not as black and white as it seems. For example, the OLAP performance in systems like DuckDB does not only come from a chunked-columnar on-disk data representation. Much of DuckDB's performance comes from its vectorized query processing engine that is custom-tuned for analytical queries. What if DuckDB was able to somehow *read data stored in PostgreSQL*? While it seems daunting, we have embarked on a quest to make just this possible.

To allow for fast and consistent analytical reads of Postgres databases, we designed and implemented the "Postgres Scanner". This scanner leverages the *binary transfer mode* of the Postgres client-server protocol (See the Implementation Section for more details.), allowing us to efficiently transform and use the data directly in DuckDB.

Among other things, DuckDB's design is different from conventional data management systems because DuckDB's query processing engine can run on nearly arbitrary data sources without needing to copy the data into its own storage format. For example, DuckDB can currently directly run queries on Parquet files, CSV files, SQLite files, Pandas, R and Julia data frames as well as Apache Arrow sources. This new extension adds the capability to directly query PostgreSQL tables from DuckDB.

Usage

The Postgres Scanner DuckDB extension source code is available on GitHub, but it is directly installable through DuckDB's new binary extension installation mechanism. To install, just run the following SQL query once:

```
INSTALL postgres_scanner;
```

Then, whenever you want to use the extension, you need to first load it:

```
LOAD postgres_scanner;
```

To make a Postgres database accessible to DuckDB, use the POSTGRES_ATTACH command:

```
CALL postgres_attach('dbname=myshinydb');
```

`postgres_attach` takes a single required string parameter, which is the `libpq` connection string. For example you can pass `'dbname=myshinydb'` to select a different database name. In the simplest case, the parameter is just `''`. There are three additional named parameters to the function:

- `source_schema` the name of a non-standard schema name in Postgres to get tables from. Default is `public`.
- `overwrite` whether we should overwrite existing views in the target schema, default is `false`.
- `filter_pushdown` whether filter predicates that DuckDB derives from the query should be forwarded to Postgres, defaults to `false`. See below for a discussion of what this parameter controls.

The tables in the database are registered as views in DuckDB, you can list them with

```
PRAGMA show_tables;
```

Then you can query those views normally using SQL. Again, no data is being copied, this is just a virtual view on the tables in your Postgres database.

If you prefer to not attach all tables, but just query a single table, that is possible using the POSTGRES_SCAN and POSTGRES_SCAN_PUSHDOWN table-producing functions directly, e.g.

```
SELECT * FROM postgres_scan('dbname=myshinydb', 'public', 'mytable');
SELECT * FROM postgres_scan_pushdown('dbname=myshinydb', 'public', 'mytable');
```

Both functions takes three unnamed string parameters, the `libpq` connection string (see above), a Postgres schema name and a table name. The schema name is often `public`. As the name suggest, the variant with "pushdown" in the name will perform selection pushdown as described below.

The Postgres scanner will only be able to read actual tables, views are not supported. However, you can of course recreate such views within DuckDB, the syntax should be exactly the same!

Implementation

From an architectural perspective, the Postgres Scanner is implemented as a plug-in extension for DuckDB that provides a so-called table scan function (`postgres_scan`) in DuckDB. There are many such functions in DuckDB and in extensions, such as the Parquet and CSV readers, Arrow readers etc.

The Postgres Scanner uses the standard `libpq` library, which it statically links in. Ironically, this makes the Postgres Scanner easier to install than the other Postgres clients. However, Postgres' normal client-server protocol is quite slow, so we spent quite some time optimizing this. As a note, DuckDB's SQLite Scanner does not face this issue, as SQLite is also an in-process database.

We actually implemented a prototype direct reader for Postgres' database files, but while performance was great, there is the issue that committed but not yet checkpointed data would not be stored in the heap files yet. In addition, if a checkpoint was currently running, our reader would frequently overtake the checkpointer, causing additional inconsistencies. We abandoned that approach since we want to be able to query an actively used Postgres database and believe that consistency is important. Another architectural option would have been to implement a DuckDB Foreign Data Wrapper (FDW) for Postgres similar to duckdb_fdw but while this could improve the protocol situation, deployment of a postgres extension is quite risky on production servers so we expect few people will be able to do so.

Instead, we use the rarely-used *binary transfer mode* of the Postgres client-server protocol. This format is quite similar to the on-disk representation of Postgres data files and avoids some of the otherwise expensive to-string and from-string conversions. For example, to read a normal `int32` from the protocol message, all we need to do is to swap byte order (`ntohl`).

The Postgres scanner connects to PostgreSQL and issues a query to read a particular table using the binary protocol. In the simplest case (see optimizations below), to read a table called `lineitem`, we internally run the query:

```
COPY (SELECT * FROM lineitem) TO STDOUT (FORMAT binary);
```

This query will start reading the contents of `lineitem` and write them directly to the protocol stream in binary format.

Parallelization

DuckDB supports automatic intra-query parallelization through pipeline parallelism, so we also want to parallelize scans on Postgres tables: Our scan operator opens multiple connections to Postgres, and reads subsets of the table from each. To efficiently split up reading the table, we use Postgres' rather obscure *TID Scan* (Tuple ID) operator, which allows a query to surgically read a specified range of tuple IDs from a table. The Tuple IDs have the form `(page, tuple)`. We parallelize our scan of a Postgres table based on database page ranges expressed in TIDs. Each scan task reads 1000 pages currently. For example, to read a table consisting of 2500 pages, we would start three scan tasks with TID ranges `[(0,0),(999,0)]`, `[(1000,0),(1999,0)]` and `[(2000,0),(UINT32_MAX,0)]`. Having an open bound for the last range is important because the number of pages (`relpages`) in a table in the `pg_class` table is merely an estimate. For a given page range (P_MIN, P_MAX), our query from above is thus extended to look like this:

```
COPY (
   SELECT
     *
   FROM lineitem
   WHERE
```

```
    ctid BETWEEN '(P_MIN,0)'::tid AND '(P_MAX,0)'::tid
  ) TO STDOUT (FORMAT binary);
```

This way, we can efficiently scan the table in parallel while not relying on the schema in any way. Because page size is fixed in Postgres, this also has the added bonus of equalizing the effort to read a subset of the page independent of the number of columns in each row.

"But wait!", you will say, according to the documentation the tuple ID is not stable and may be changed by operations such as `VACUUM ALL`. How can you use it for synchronizing parallel scans? This is true, and could be problematic, but we found a solution:

Transactional Synchronization

Of course a transactional database such as Postgres is expected to run transactions while we run our table scans for analytical purposes. Therefore we need to address concurrent changes to the table we are scanning in parallel. We solve this by first creating a new read-only transaction in DuckDB's bind phase, where query planning happens. We leave this transaction running until we are completely done reading the table. We use yet another little-known Postgres feature, `pg_export_snapshot()`, which allows us to get the current transaction context in one connection, and then import it into our parallel read connections using `SET TRANSACTION SNAPSHOT ...`. This way, all connections related to one single table scan will see the table state exactly as it appeared at the very beginning of our scan throughout the potentially lengthy read process.

Projection and Selection Push-Down

DuckDB's query optimizer moves selections (filters on rows) and projections (removal of unused columns) as low as possible in the query plan (push down), and even instructs the lowermost scan operators to perform those operations if they support them. For the Postgres scanner, we have implemented both push down variants. Projections are rather straightforward – we can immediately instruct Postgres to only retrieve the columns the query is using. This of course also reduces the number of bytes that need to be transferred, which speeds up queries. For selections, we construct a SQL filter expression from the pushed down filters. For example, if we run a query like `SELECT l_returnflag, l_linestatus FROM lineitem WHERE l_shipdate < '1998-09-02'` through the Postgres scanner, it would run the following queries:

```
COPY (
  SELECT
    "l_returnflag",
    "l_linestatus"
  FROM "public"."lineitem"
  WHERE
    ctid BETWEEN '(0,0)'::tid AND '(1000,0)'::tid AND
    ("l_shipdate" < '1998-09-02' AND "l_shipdate" IS NOT NULL)
  ) TO STDOUT (FORMAT BINARY);
-- and so on
```

As you can see, the projection and selection pushdown has expanded the queries ran against Postgres accordingly. Using the selection push-down is optional. There may be cases where running a filter in Postgres is actually slower than transferring the data and running the filter in DuckDB, for example when filters are not very selective (many rows match).

Performance

To investigate the performance of the Postgres Scanner, we ran the well-known TPC-H benchmark on DuckDB using its internal storage format, on Postgres also using its internal format and with DuckDB reading from Postgres using the new Postgres Scanner. We used DuckDB 0.5.1 and Postgres 14.5, all experiments were run on a MacBook Pro with an M1 Max CPU. The experiment script is available. We run "scale factor" 1 of TPCH, creating a dataset of roughly 1 GB with ca. 6 M rows in the biggest table, `lineitem`. Each of the 22 TPC-H benchmark queries was run 5 times, and we report the median run time in seconds. The time breakdown is given in the following table.

query	duckdb	duckdb/postgres	postgres
1	0.03	0.74	1.12
2	0.01	0.20	0.18
3	0.02	0.55	0.21
4	0.03	0.52	0.11
5	0.02	0.70	0.13
6	0.01	0.24	0.21
7	0.04	0.56	0.20
8	0.02	0.74	0.18
9	0.05	1.34	0.61
10	0.04	0.41	0.35
11	0.01	0.15	0.07
12	0.01	0.27	0.36
13	0.04	0.18	0.32
14	0.01	0.19	0.21
15	0.03	0.36	0.46
16	0.03	0.09	0.12
17	0.05	0.75	> 60.00
18	0.08	0.97	1.05
19	0.03	0.32	0.31
20	0.05	0.37	> 60.00
21	0.09	1.53	0.35
22	0.03	0.15	0.15

Stock Postgres is not able to finish queries 17 and 20 within a one-minute timeout because of correlated subqueries containing a query on the lineitem table. For the other queries, we can see that DuckDB with the Postgres Scanner not only finished all queries, it also was faster than stock Postgres on roughly half of them, which is astonishing given that DuckDB has to read its input data from Postgres through the client/server protocol as described above. Of course, stock DuckDB is still 10× faster with its own storage, but as discussed at the very beginning of this post this requires the data to be imported there first.

Other Use Cases

The Postgres Scanner can also be used to combine live Postgres data with pre-cached data in creative ways. This is especially effective when dealing with an append only table, but could also be used if a modified date column is present. Consider the following SQL template:

```
INSERT INTO my_table_duckdb_cache
SELECT * FROM postgres_scan('dbname=myshinydb', 'public', 'my_table')
WHERE incrementing_id_column > (SELECT max(incrementing_id_column) FROM my_table_duckdb_cache);

SELECT * FROM my_table_duckdb_cache;
```

This provides faster query performance with fully up to date query results, at the cost of data duplication. It also avoids complex data replication technologies.

DuckDB has built-in support to write query results to Parquet files. The Postgres scanner provides a rather simple way to write Postgres tables to Parquet files, it can even directly write to S3 if desired. For example,

```
COPY (SELECT * FROM postgres_scan('dbname=myshinydb', 'public', 'lineitem')) TO 'lineitem.parquet'
(FORMAT parquet);
```

Conclusion

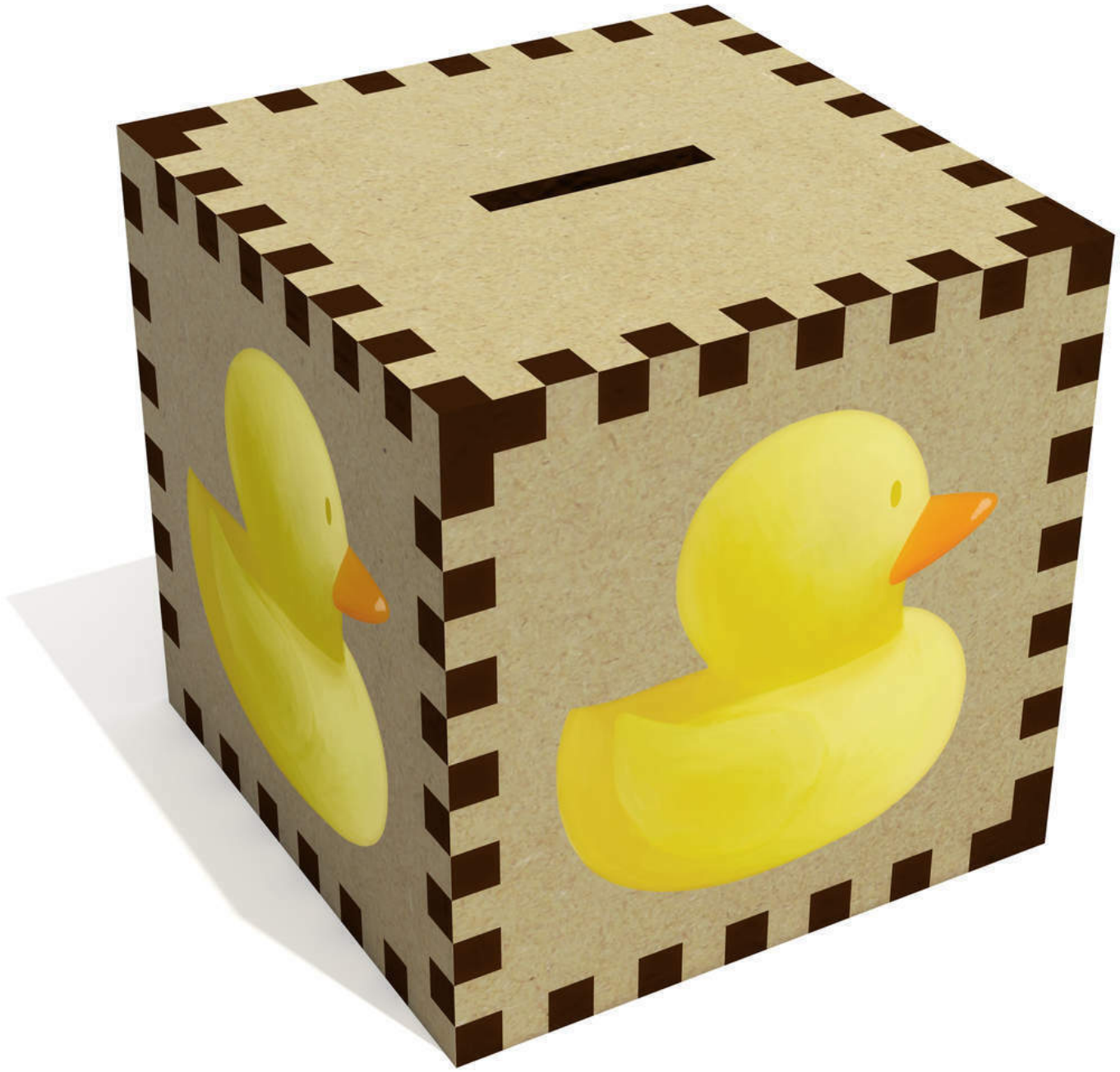
DuckDB's new Postgres Scanner extension can read PostgreSQL's tables while PostgreSQL is running and compute the answers to complex OLAP SQL queries often faster than PostgreSQL itself can without the need to duplicate data. The Postgres Scanner is currently in preview and we are curious to hear what you think. If you find any issues with the Postgres Scanner, please report them.

Modern Data Stack in a Box with DuckDB

Publication date: 2022-10-12

Author: Guest post by Jacob Matson

TL;DR: A fast, free, and open-source Modern Data Stack (MDS) can now be fully deployed on your laptop or to a single machine using the combination of DuckDB, Meltano, dbt, and Apache Superset.



This post is a collaboration with Jacob Matson and cross-posted on dataduel.co.

Summary

There is a large volume of literature, e.g., 1 and 2, about scaling data pipelines. "Use Kafka! Build a lake house! Don't build a lake house, use Snowflake! Don't use Snowflake, use XYZ!" However, with advances in hardware and the rapid maturation of data software, there is a simpler approach. This article will light up the path to highly performant single node analytics with an MDS-in-a-box open source stack: Meltano, DuckDB, dbt, & Apache Superset on Windows using Windows Subsystem for Linux (WSL). There are many options within the MDS, so if you are using another stack to build an MDS-in-a-box, please share it with the community on the DuckDB Twitter, GitHub, or Discord, or the dbt slack! Or just stop by for a friendly debate about our choice of tools!

Motivation

What is the Modern Data Stack, and why use it? The MDS can mean many things (see examples and a historical perspective), but fundamentally it is a return to using SQL for data transformations by combining multiple best-in-class software tools to form a stack. A typical stack would include (at least!) a tool to extract data from sources and load it into a data warehouse, dbt to transform and analyze that data in the warehouse, and a business intelligence tool. The MDS leverages the accessibility of SQL in combination with software development best practices like git to enable analysts to scale their impact across their companies.

Why build a bundled Modern Data Stack on a single machine, rather than on multiple machines and on a data warehouse? There are many advantages!

- Simplify for higher developer productivity
- Reduce costs by removing the data warehouse
- Deploy with ease either locally, on-premise, in the cloud, or all 3
- Eliminate software expenses with a fully free and open-source stack
- Maintain high performance with modern software like DuckDB and increasingly powerful single-node compute instances
- Achieve self-sufficiency by completing an end-to-end proof of concept on your laptop
- Enable development best practices by integrating with GitHub
- Enhance security by (optionally) running entirely locally or on-premise

If you contribute to an open-source community or provide a product within the Modern Data Stack, there is an additional benefit!

- Increase adoption of your tool by providing a free and self-contained example stack
  - Dagster's example project uses DuckDB for this already!
  - Reach out on the DuckDB Twitter, GitHub, or Discord, or the dbt slack to share an example using your tool with the community!

Trade-offs

One key component of the MDS is the unlimited scalability of compute. How does that align with the MDS-in-a-box approach? Today, cloud computing instances can vertically scale significantly more than in the past (for example, 224 cores and 24 TB of RAM on AWS!). Laptops are more powerful than ever. Now that new OLAP tools like DuckDB can take better advantage of that compute, horizontal scaling is no longer necessary for many analyses! Also, this MDS-in-a-box can be duplicated with ease to as many boxes as needed if partitioned by data subject area. So, while infinite compute is sacrificed, significant scale is still easily achievable.

Due to this tradeoff, this approach is more of an "Open Source Analytics Stack in a box" than a traditional MDS. It sacrifices infinite scale for significant simplification and the other benefits above.

Choosing a Problem

Given that the NBA season is starting soon, a monte carlo type simulation of the season is both topical and well-suited for analytical SQL. This is a particularly great scenario to test the limits of DuckDB because it only requires simple inputs and easily scales out to massive numbers of records. This entire project is held in a GitHub repo, which you can find on GitHub.

Building the Environment

The detailed steps to build the project can be found in the repo, but the high-level steps will be repeated here. As a note, Windows Subsystem for Linux (WSL) was chosen to support Apache Superset, but the other components of this stack can run directly on any operating system. Thankfully, using Linux on Windows has become very straightforward.

1. Install Ubuntu 20.04 on WSL.
2. Upgrade your packages (`sudo apt update`).
3. Install python.
4. Clone the git repo.
5. Run `make build` and then `make run` in the terminal.
6. Create super admin user for Superset in the terminal, then login and configure the database.
7. Run test queries in superset to check your work.

Meltano as a Wrapper for Pipeline Plugins

In this example, Meltano pulls together multiple bits and pieces to allow the pipeline to be run with a single statement. The first part is the tap (extractor) which is 'tap-spreadsheets-anywhere'. This tap allows us to get flat data files from various sources. It should be noted that DuckDB can consume directly from flat files (locally and over the network), or SQLite and PostgreSQL databases. However, this tap was chosen to provide a clear example of getting static data into your database that can easily be configured in the meltano.yml file. Meltano also becomes more beneficial as the complexity of your data sources increases.

```
plugins:
  extractors:
  - name: tap-spreadsheets-anywhere
    variant: ets
    pip_url: git+https://github.com/ets/tap-spreadsheets-anywhere.git
# data sources are configured inside of this extractor
```

The next bit is the target (loader), 'target-duckdb'. This target can take data from any Meltano tap and load it into DuckDB. Part of the beauty of this approach is that you don't have to mess with all the extra complexity that comes with a typical database. DuckDB can be dropped in and is ready to go with zero configuration or ongoing maintenance. Furthermore, because the components and the data are co-located, networking is not a consideration and further reduces complexity.

```
  loaders:
  - name: target-duckdb
    variant: jwills
    pip_url: target-duckdb~=0.4
    config:
      filepath: /tmp/mdsbox.db
      default_target_schema: main
```

Next is the transformer: 'dbt-duckdb'. dbt enables transformations using a combination of SQL and Jinja templating for approachable SQL-based analytics engineering. The dbt adapter for DuckDB now supports parallel execution across threads, which makes the MDS-in-a-box run even faster. Since the bulk of the work is happening inside of dbt, this portion will be described in detail later in the post.

```
  transformers:
  - name: dbt-duckdb
    variant: jwills
```

```
    pip_url: dbt-core~=1.2.0 dbt-duckdb~=1.2.0
    config:
      path: /tmp/mdsbox.db
```

Lastly, Apache Superset is included as a Meltano utility to enable some data querying and visualization. Superset leverages DuckDB's SQLAlchemy driver, duckdb_engine, so it can query DuckDB directly as well.

```
  utilities:
  - name: superset
    variant: apache
    pip_url: apache-superset==1.5.0 markupsafe==2.0.1 duckdb-engine==0.6.4
```

With Superset, the engine needs to be configured to open DuckDB in "read-only" mode. Otherwise, only one query can run at a time (simultaneous queries will cause locks). This also prevents refreshing the Superset dashboard while the pipeline is running. In this case, the pipeline runs in under 8 seconds!

Wrangling the Data

The NBA schedule was downloaded from basketball-reference.com, and the Draft Kings win totals from Sept 27th were used for win totals. The schedule and win totals make up the entirety of the data required as inputs for this project. Once converted into CSV format, they were uploaded to the GitHub project, and the meltano.yml file was updated to reference the file locations.

Loading Sources

Once the data is on the web inside of GitHub, Meltano can pull a copy down into DuckDB. With the command `meltano run tap-spreadsheets-anywhere target-duckdb`, the data is loaded into DuckDB, and ready for transformation inside of dbt.

Building dbt Models

After the sources are loaded, the data is transformed with dbt. First, the source models are created as well as the scenario generator. Then the random numbers for that simulation run are generated – it should be noted that the random numbers are recorded as a table, not a view, in order to allow subsequent re-runs of the downstream models with the graph operators for troubleshooting purposes (i.e., `dbt run -s random_num_gen+`). Once the underlying data is laid out, the simulation begins, first by simulating the regular season, then the play-in games, and lastly the playoffs. Since each round of games has a dependency on the previous round, parallelization is limited in this model, which is reflected in the dbt DAG, in this case conveniently hosted on GitHub Pages.

There are a few more design choices worth calling out:

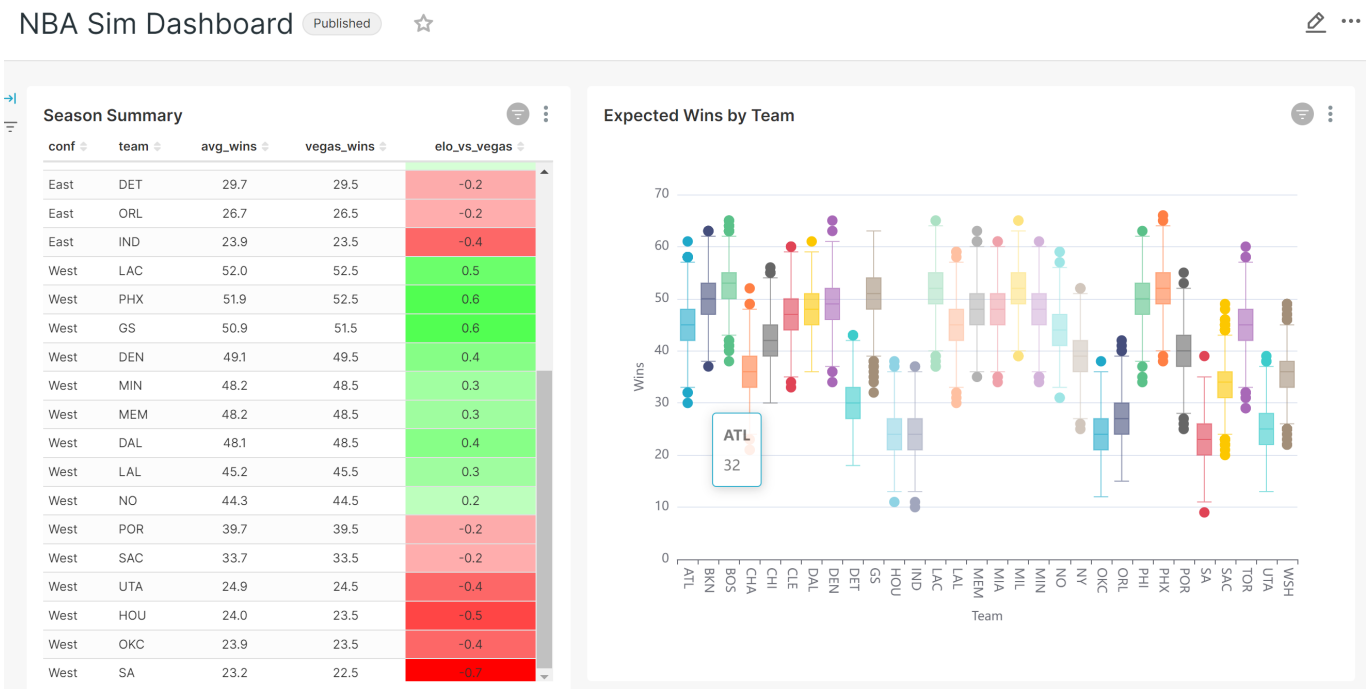
1. Simulation tables and summary tables were split into separate models for ease of use / transparency. So each round of the simulation has a sim model and an end model – this allows visibility into the correct parameters (conference, team, elo rating) to be passed into each subsequent round.
2. To prevent overly deep queries, 'reg_season_end' and 'playoff_sim_r1' have been materialized as tables. While it is slightly slower on build, the performance gains when querying summary tables (i.e., 'season_summary') are more than worth the slowdown. However, it should be noted that even for only 10k sims, the database takes up about 150 MB in disk space. Running at 100k simulations easily expands it to a few GB.

Connecting Superset

Once the dbt models are built, the data visualization can begin. An admin user must be created in superset in order to log in. The instructions for connecting the database can be found in the GitHub project, as well as a note on how to connect it in 'read only mode'.

There are 2 models designed for analysis, although any number of them can be used. 'season_summary' contains various summary statistics for the season, and 'reg_season_sim' contains all simulated game results. This second data set produces an interesting histogram chart. In order to build data visualizations in superset, the dataset must be defined first, the chart built, and lastly, the chart assigned to a dashboard.

Below is an example Superset dashboard containing several charts based on this data. Superset is able to clearly summarize the data as well as display the level of variability within the monte carlo simulation. The duckdb_engine queries can be refreshed quickly when new simulations are run.



Conclusion

The ecosystem around DuckDB has grown such that it integrates well with the Modern Data Stack. The MDS-in-a-box is a viable approach for smaller data projects, and would work especially well for read-heavy analytics. There were a few other learnings from this experiment. Superset dashboards are easy to construct, but they are not scriptable and must be built in the GUI (the paid hosted version, Preset, does support exporting as YAML). Also, while you can do monte carlo analysis in SQL, it may be easier to do in another language. However, this shows how far you can stretch the capabilities of SQL!

Next Steps

There are additional directions to take this project. One next step could be to Dockerize this workflow for even easier deployments. If you want to put together a Docker example, please reach out! Another adjustment to the approach could be to land the final outputs in parquet files, and to read them with in-memory DuckDB connections. Those files could even be landed in an S3-compatible object store (and still read by DuckDB), although that adds complexity compared with the in-a-box approach! Additional MDS components could also be integrated for data quality monitoring, lineage tracking, etc.

Josh Wills is also in the process of making an interesting enhancement to dbt-duckdb! Using the sqlglot library, dbt-duckdb would be able to automatically transpile dbt models written using the SQL dialect of other databases (including Snowflake and BigQuery) to DuckDB. Imagine if you could test out your queries locally before pushing to production... Join the DuckDB channel of the dbt slack to discuss the possibilities!

Please reach out if you use this or another approach to build an MDS-in-a-box! Also, if you are interested in writing a guest post for the DuckDB blog, please reach out on Discord!

Lightweight Compression in DuckDB

Publication date: 2022-10-28

Author: Mark Raasveldt

TL;DR: DuckDB supports efficient lightweight compression that is automatically used to keep data size down without incurring high costs for compression and decompression.

When working with large amounts of data, compression is critical for reducing storage size and egress costs. Compression algorithms typically reduce data set size by **75-95%**, depending on how compressible the data is. Compression not only reduces the storage footprint of a data set, but also often **improves performance** as less data has to be read from disk or over a network connection.

Column store formats, such as DuckDB's native file format or Parquet, benefit especially from compression. That is because data within an

individual column is generally very similar, which can be exploited effectively by compression algorithms. Storing data in row-wise format results in interleaving of data of different columns, leading to lower compression rates.

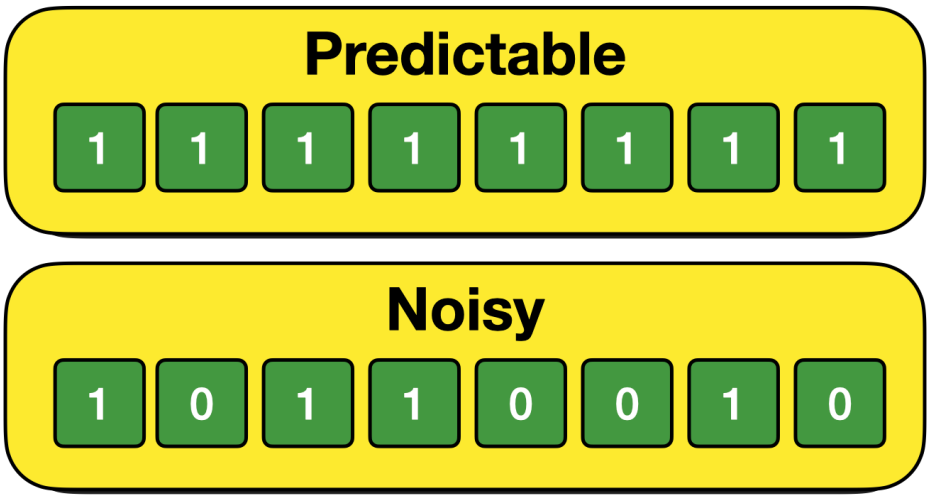
DuckDB added support for compression at the end of last year. As shown in the table below, the compression ratio of DuckDB has continuously improved since then and is still actively being improved. In this blog post, we discuss how compression in DuckDB works, and the design choices and various trade-offs that we have made while implementing compression for DuckDB's storage format.

Version	Taxi	On Time	`lineitem`	Notes	Date
DuckDB v0.2.8	15.3 GB	1.73 GB	0.85 GB	Uncompressed	July 2021
DuckDB v0.2.9	11.2 GB	1.25 GB	0.79 GB	RLE + Constant	September 2021
DuckDB v0.3.2	10.8 GB	0.98 GB	0.56 GB	Bitpacking	February 2022
DuckDB v0.3.3	6.9 GB	0.23 GB	0.32 GB	Dictionary	April 2022
DuckDB v0.5.0	6.6 GB	0.21 GB	0.29 GB	FOR	September 2022
DuckDB dev	4.8 GB	0.21 GB	0.17 GB	FSST + Chimp	now ( )
CSV	17.0 GB	1.11 GB	0.72 GB		
Parquet (Uncompressed)	4.5 GB	0.12 GB	0.31 GB		
Parquet (Snappy)	3.2 GB	0.11 GB	0.18 GB		
Parquet (ZSTD)	2.6 GB	0.08 GB	0.15 GB		

Compression Intro

At its core, compression algorithms try to find patterns in a data set in order to store it more cleverly. **Compressibility** of a data set is therefore dependent on whether or not such patterns can be found, and whether they exist in the first place. Data that follows a fixed pattern can be compressed significantly. Data that does not have any patterns, such as random noise, cannot be compressed. Formally, the compressibility of a dataset is known as its entropy.

As an example of this concept, let us consider the following two data sets.



The constant data set can be compressed by simply storing the value of the pattern and how many times the pattern repeats (e.g., `1x8`). The random noise, on the other hand, has no pattern, and is therefore not compressible.

General Purpose Compression Algorithms

The compression algorithms that most people are familiar with are *general purpose compression algorithms*, such as *zip*, *gzip* or *zstd*. General purpose compression algorithms work by finding patterns in bits. They are therefore agnostic to data types, and can be used on any stream of bits. They can be used to compress files, but they can also be applied to arbitrary data sent over a socket connection.

General purpose compression is flexible and very easy to set up. There are a number of high quality libraries available (such as zstd, snappy or lz4) that provide compression, and they can be applied to any data set stored in any manner.

The downside of general purpose compression is that (de)compression is generally expensive. While this does not matter if we are reading and writing from a hard disk or over a slow internet connection, the speed of (de)compression can become a bottleneck when data is stored in RAM.

Another downside is that these libraries operate as a *black box*. They operate on streams of bits, and do not reveal information of their internal state to the user. While that is not a problem if you are only looking to decrease the size of your data, it prevents the system from taking advantage of the patterns found by the compression algorithm during execution.

Finally, general purpose compression algorithms work better when compressing large chunks of data. As illustrated in the table below, compression ratios suffer significantly when compressing small amounts of data. To achieve a good compression ratio, blocks of at least **256 kB** must be used.

Compression	1 kB	4 kB	16 kB	64 kB	256 kB	1 MB
zstd	1.72	2.1	2.21	2.41	2.54	2.73
lz4	1.29	1.5	1.52	1.58	1.62	1.64
gzip	1.7	2.13	2.28	2.49	2.62	2.67

This is relevant because the block size is the minimum amount of data that must be decompressed when reading a single row from disk. Worse, as DuckDB compresses data on a per-column basis, the block size would be the minimum amount of data that must be decompressed per column. With a block size of 256 kB, fetching a single row could require decompressing multiple megabytes of data. This can cause queries that fetch a low number of rows, such as `SELECT * FROM tbl LIMIT 5` or `SELECT * FROM tbl WHERE id = 42` to incur significant costs, despite appearing to be very cheap on the surface.

Lightweight Compression Algorithms

Another option for achieving compression is to use specialized lightweight compression algorithms. These algorithms also operate by finding patterns in data. However, unlike general purpose compression, they do not attempt to find generic patterns in bitstreams. Instead, they operate by finding **specific patterns** in data sets.

By detecting specific patterns, specialized compression algorithms can be significantly more lightweight, providing much faster compression and decompression. In addition, they can be effective on much smaller data sizes. This allows us to decompress a few rows at a time, rather than requiring large blocks of data to be decompressed at once. These specialized compression algorithms can also offer efficient support for random seeks, making data access through an index significantly faster.

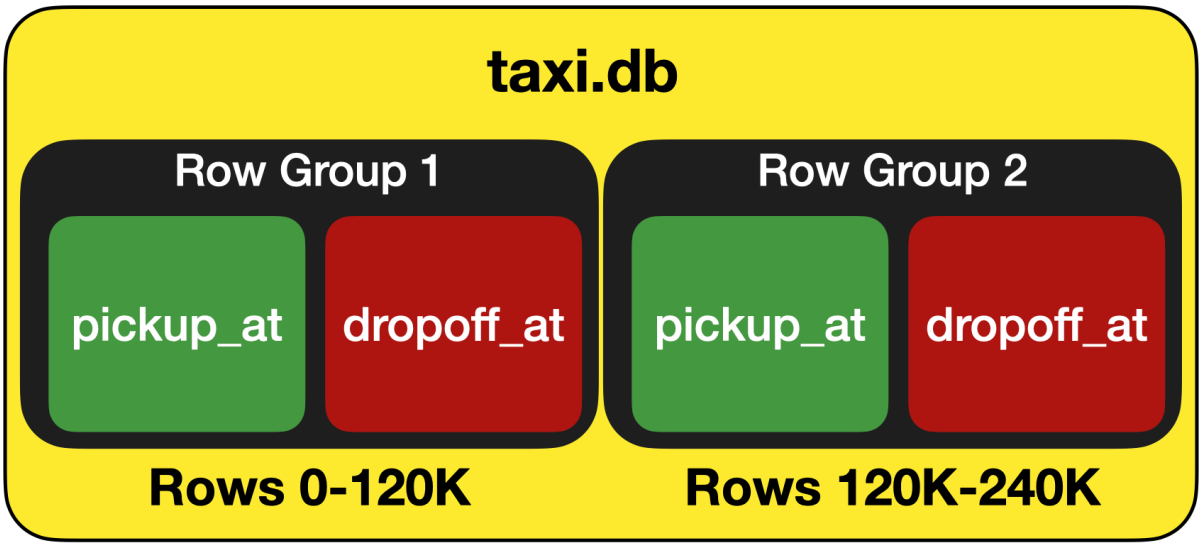
Lightweight compression algorithms also provide us with more fine-grained control over the compression process. This is especially relevant for us as DuckDB's file format uses fixed-size blocks in order to avoid fragmentation for workloads involving deletes and updates. The fine-grained control allows us to fill these blocks more effectively, and avoid having to guess how much compressed data will fit into a buffer.

On the flip side, these algorithms are ineffective if the specific patterns they are designed for do not occur in the data. As a result, individually, these lightweight compression algorithms are no replacement for general purpose algorithms. Instead, multiple specialized algorithms must be combined in order to capture many different common patterns in data sets.

Compression Framework

Because of the advantages described above, DuckDB uses only specialized lightweight compression algorithms. As each of these algorithms work optimally on different patterns in the data, DuckDB's compression framework must first decide on which algorithm to use to store the data of each column.

DuckDB's storage splits tables into *Row Groups*. These are groups of `120K` rows, stored in columnar chunks called *Column Segments*. This storage layout is similar to Parquet – but with an important difference: columns are split into blocks of a fixed-size. This design decision was made because DuckDB's storage format supports in-place ACID modifications to the storage format, including deleting and updating rows, and adding and dropping columns. By partitioning data into fixed size blocks the blocks can be easily reused after they are no longer required and fragmentation is avoided.



The compression framework operates within the context of the individual *Column Segments*. It operates in two phases. First, the data in the column segment is *analyzed*. In this phase, we scan the data in the segment and find out the best compression algorithm for that particular segment. After that, the *compression* is performed, and the compressed data is written to the blocks on disk.

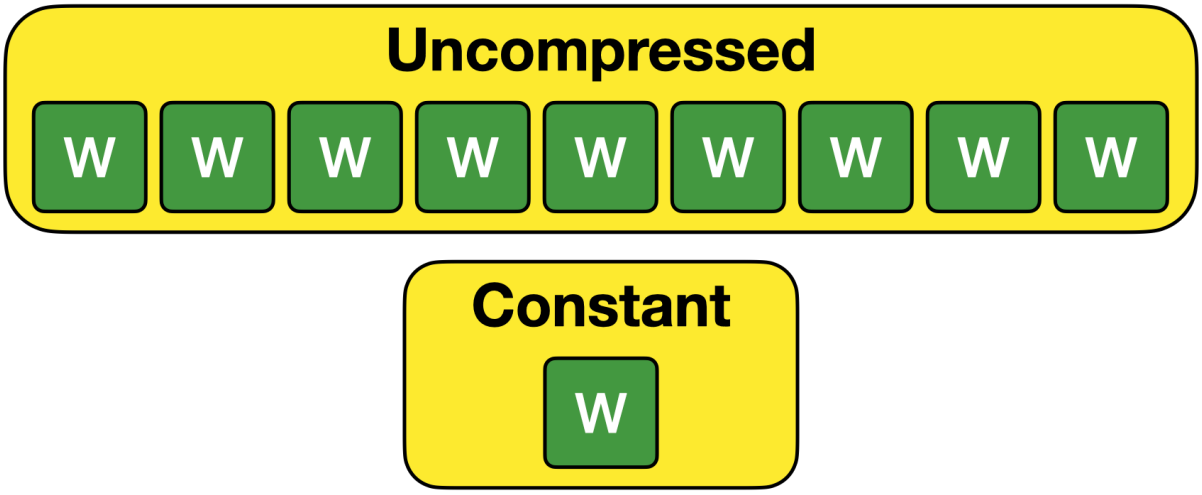
While this approach requires two passes over the data within a segment, this does not incur a significant cost, as the amount of data stored in one segment is generally small enough to fit in the CPU caches. A sampling approach for the analyze step could also be considered, but in general we value choosing the best compression algorithm and reducing file size over a minor increase in compression speed.

Compression Algorithms

DuckDB implements several lightweight compression algorithms, and we are in the process of adding more to the system. We will go over a few of these compression algorithms and how they work in the following sections.

Constant Encoding

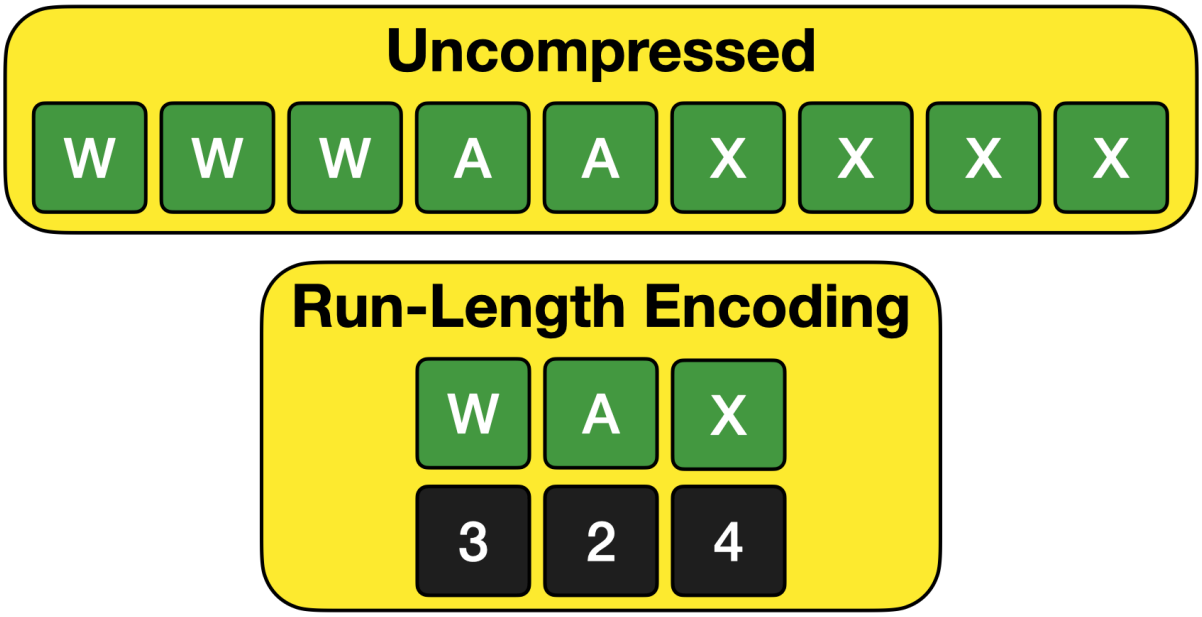
Constant encoding is the most straightforward compression algorithm in DuckDB. Constant encoding is used when every single value in a column segment is the same value. In that case, we store only that single value. This encoding is visualized below.



When applicable, this encoding technique leads to tremendous space savings. While it might seem like this technique is rarely applicable – in practice it occurs relatively frequently. Columns might be filled with `NULL` values, or have values that rarely change (such as e.g., a `year` column in a stream of sensor data). Because of this compression algorithm, such columns take up almost no space in DuckDB.

Run-Length Encoding (RLE)

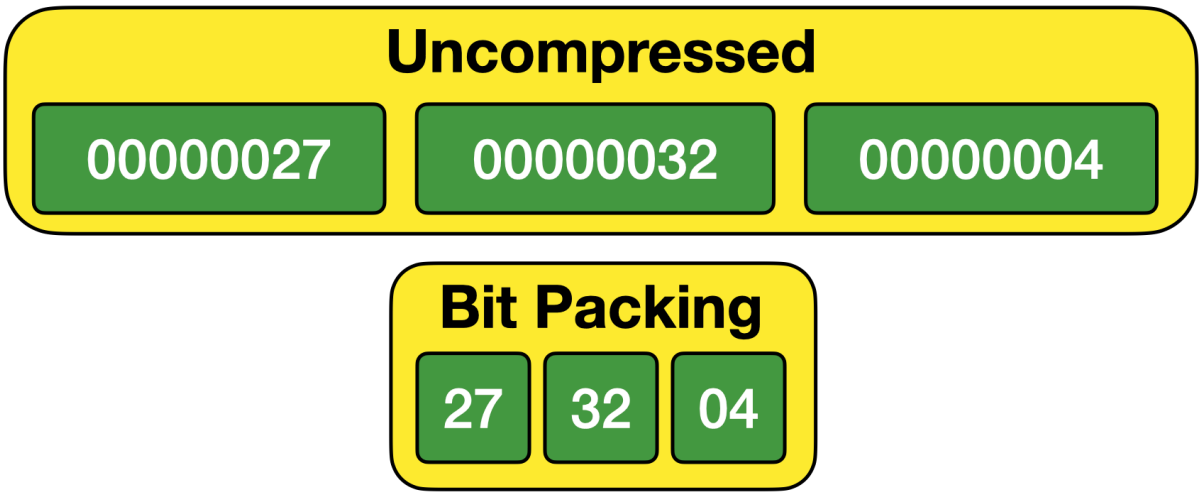
Run-length encoding (RLE) is a compression algorithm that takes advantage of repeated values in a dataset. Rather than storing individual values, the data set is decomposed into a pair of (value, count) tuples, where the count represents how often the value is repeated. This encoding is visualized below.



RLE is powerful when there are many repeating values in the data. This might occur when data is sorted or partitioned on a particular attribute. It is also useful for columns that have many missing (NULL) values.

Bit Packing

Bit Packing is a compression technique that takes advantage of the fact that integral values rarely span the full range of their data type. For example, four-byte integer values can store values from negative two billion to positive two billion. Frequently the full range of this data type is not used, and instead only small numbers are stored. Bit packing takes advantage of this by removing all of the unnecessary leading zeros when storing values. An example (in decimal) is provided below.

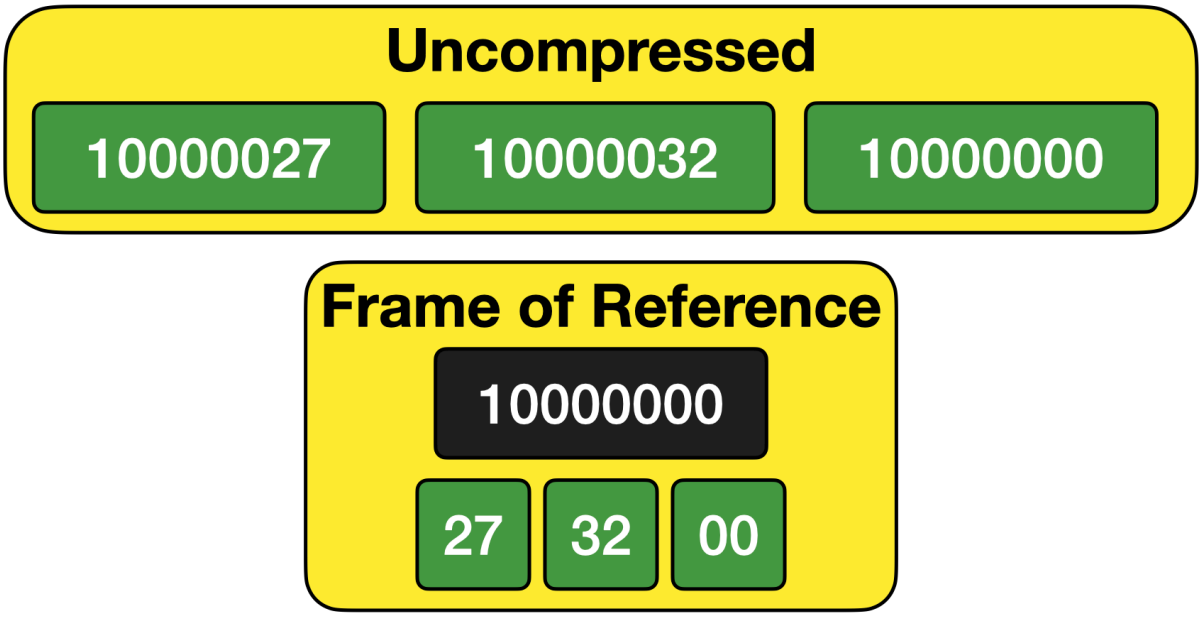


For bit packing compression, we keep track of the maximum value for every `1024` values. The maximum value determines the bit packing width, which is the number of bits necessary to store that value. For example, when storing a set of values with a maximum value of `32`, the bit packing width is `5` bits, down from the `32` bits per value that would be required to store uncompressed four-byte integers.

Bit packing is very powerful in practice. It is also convenient to users – as due to this technique there are no storage size differences between using the various integer types. A `BIGINT` column will be stored in the exact same amount of space as an `INTEGER` column. That relieves the user from having to worry about which integer type to choose.

Frame of Reference

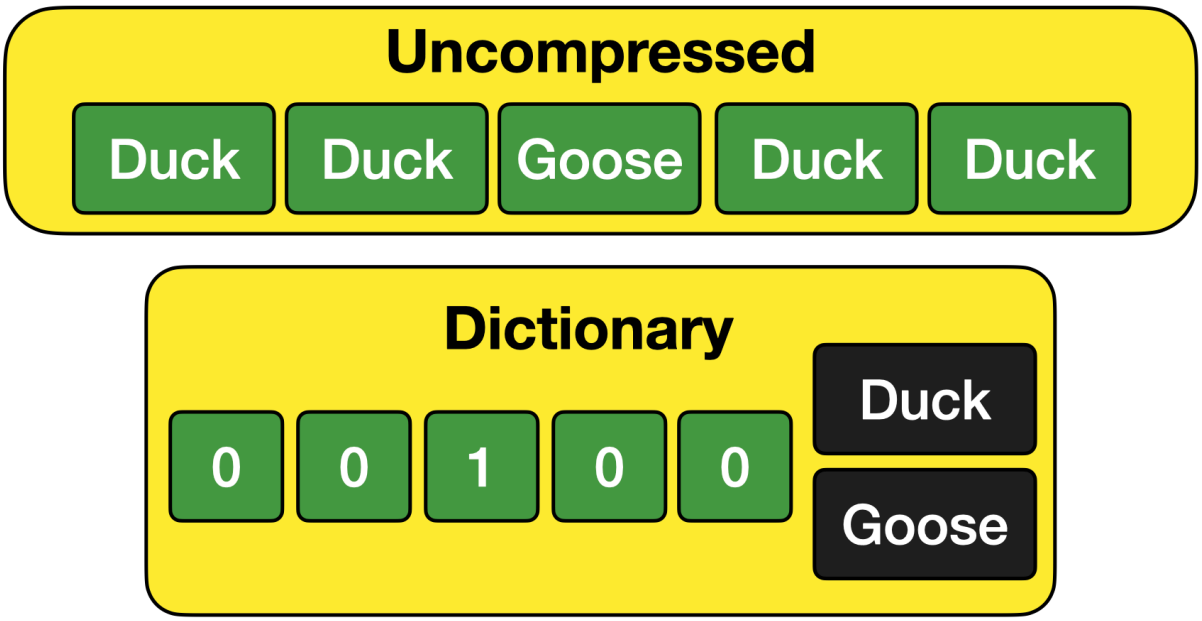
Frame of Reference encoding is an extension of bit packing, where we also include a frame. The frame is the minimum value found in the set of values. The values are stored as the offset from this frame. An example of this is given below.



While this might not seem particularly useful at a first glance, it is very powerful when storing dates and timestamps. That is because dates and timestamps are stored as Unix Timestamps in DuckDB, i.e., the offset since `1970-01-01` in either days (for dates) or microseconds (for timestamps). When we have a set of date or timestamp values, the absolute numbers might be very high, but the numbers are all very close together. By applying a frame before bit packing, we can often improve our compression ratio tremendously.

Dictionary Encoding

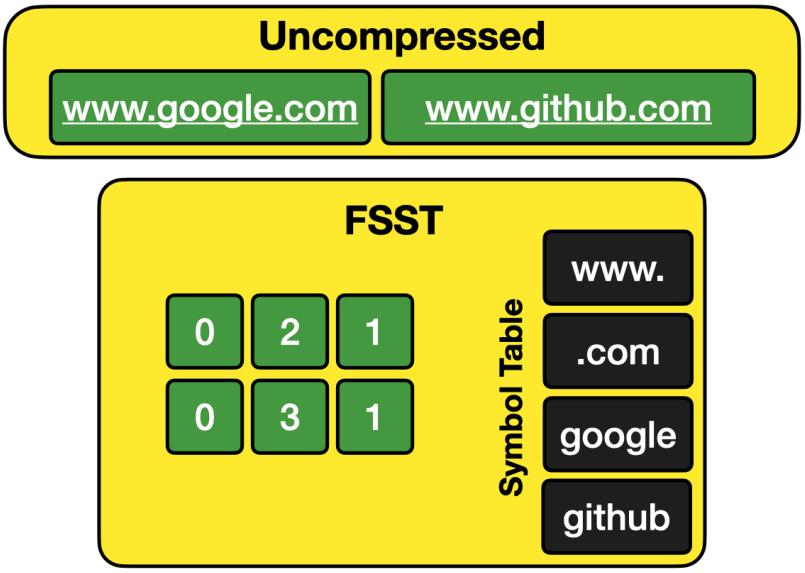
Dictionary encoding works by extracting common values into a separate dictionary, and then replacing the original values with references to said dictionary. An example is provided below.



Dictionary encoding is particularly efficient when storing text columns with many duplicate entries. The much larger text values can be replaced by small numbers, which can in turn be efficiently bit packed together.

FSST

Fast Static Symbol Table compression is an extension to dictionary compression, that not only extracts repetitions of entire strings, but also extracts repetitions *within* strings. This is effective when storing strings that are themselves unique, but have a lot of repetition within the strings, such as URLs or e-mail addresses. An image illustrating how this works is shown below.



For those interested in learning more, watch the talk by Peter Boncz here.

Chimp & Patas

Chimp is a very new compression algorithm that is designed to compress floating point values. It is based on Gorilla compression. The core idea behind Gorilla and Chimp is that floating point values, when XOR'd together, seem to produce small values with many trailing and leading zeros. These algorithms then work on finding an efficient way of storing the trailing and leading zeros.

After implementing Chimp, we have been inspired and worked on implementing Patas, which uses many of the same ideas but optimizes further for higher decompression speed. Expect a future blog post explaining these in more detail soon!

Inspecting Compression

The PRAGMA `storage_info` can be used to inspect the storage layout of tables and columns. This can be used to inspect which compression algorithm has been chosen by DuckDB to compress specific columns of a table.

```
SELECT * EXCLUDE (column_path, segment_id, start, stats, persistent, block_id, block_offset, has_updates)
FROM pragma_storage_info('taxi')
USING SAMPLE 10 ROWS
ORDER BY row_group_id;
```

row_group_id	column_name	column_id	segment_type	count	compression
4	extra	13	FLOAT	65536	Chimp
20	tip_amount	15	FLOAT	65536	Chimp
26	pickup_latitude	6	VALIDITY	65536	Constant
46	tolls_amount	16	FLOAT	65536	RLE
73	store_and_fwd_flag	8	VALIDITY	65536	Uncompressed

row_group_id	column_name	column_id	segment_type	count	compression
96	total_amount	17	VALIDITY	65536	Constant
111	total_amount	17	VALIDITY	65536	Constant
141	pickup_at	1	TIMESTAMP	52224	BitPacking
201	pickup_longitude	5	VALIDITY	65536	Constant
209	passenger_count	3	TINYINT	65536	BitPacking

Conclusion & Future Goals

Compression has been tremendously successful in DuckDB, and we have made great strides in reducing the storage requirements of the system. We are still actively working on extending compression within DuckDB, and are looking to improve the compression ratio of the system even further, both by improving our existing techniques and implementing several others. Our goal is to achieve compression on par with Parquet with Snappy, while using only lightweight specialized compression techniques that are very fast to operate on.

Announcing DuckDB 0.6.0

Publication date: 2022-11-14

Author: Mark Raasveldt

The DuckDB team is happy to announce the latest DuckDB version (0.6.0) has been released. This release of DuckDB is named ”Oxyura” after the White-headed duck (Oxyura leucocephala) which is an endangered species native to Eurasia.

To install the new version, please visit the installation guide. Note that the release is still being rolled out, so not all artifacts may be published yet. The full release notes can be found on GitHub.

What's in 0.6.0

The new release contains many improvements to the storage system, general performance improvements, memory management improvements and new features. Below is a summary of the most impactful changes, together with the linked PRs that implement the features.

Storage Improvements

As we are working towards stabilizing the storage format and moving towards version 1.0, we have been actively working on improving our storage format, including many compression improvements.

Optimistic writing to disk. In previous DuckDB versions, the data of a single transaction was first loaded into memory, and would only be written to disk on a commit. While this works fine when data is loaded in batches that fit in memory, it does not work well when loading a lot of data in a single transaction, such as when ingesting one very large file into the system.

This version introduces optimistic writing to disk. When loading large data sets in a single transaction, data is compressed and streamed to the database file, even before the `COMMIT` has occurred. When the transaction is committed, the data will already have been written to disk, and no further writing has to happen. On a rollback, any optimistically written data is reclaimed by the system.

Parallel data loading. In addition to optimistically writing data to disk, this release includes support for parallel data loading into individual tables. This greatly improves performance of data loading on machines that have multiple cores (i.e., all modern machines).

Below is a benchmark comparing loading time of 150 million rows of the Taxi dataset from a Parquet file on an M1 Max with 10 cores:

Version	Load time
v0.5.1	91.4 s
v0.6.0	17.2 s

DuckDB supports two modes – the `order-preserving` and the `non-order-preserving` parallel data load.

The order-preserving load preserves the insertion order so that e.g., the first line in your CSV file is the first line in the DuckDB table. The non-order-preserving load does not offer such guarantees – and instead might re-order the data on load. By default the order-preserving load is used, which involves some extra book-keeping. The preservation of insertion order can be disabled using the `SET preserve_insertion_order = false` statement.

Compression Improvements

FSST. The Fast Static Symbol Table compression algorithm is introduced in this version. This state-of-the-art compression algorithm compresses data *inside* strings using a dictionary, while maintaining support for efficient scans and random look-ups. This greatly increases the compression ratio of strings that have many unique values but with common elements, such as e-mail addresses or URLs.

The compression ratio improvements of the TPC-H SF1 dataset are shown below:

Compression	Size
Uncompressed	761 MB
Dictionary	510 MB
FSST + Dictionary	251 MB

Chimp. The Chimp compression algorithm is included, which is the state-of-the-art in lightweight floating point compression. Chimp is an improved version of Gorillas, that achieves both a better compression ratio as well as faster decompression speed.

Patas. Patas is a novel floating point compression method that iterates upon the Chimp algorithm by optimizing for a single case in the Chimp algorithm. While Patas generally has a slightly lower compression ratio than Chimp, it has significantly faster decompression speed, almost matching uncompressed data in read speed.

The compression ratio of a dataset containing temperatures of cities stored as double (8-byte floating point numbers) is shown below:

Compression	Size
Uncompressed	25.4 MB
Chimp	9.7 MB
Patas	10.2 MB

Performance Improvements

DuckDB aims to have very high performance for a wide variety of workloads. As such, we are always working to improve performance for various workloads. This release is no different.

Parallel CSV Loading (Experimental). In this release we are launching a new experimental parallel CSV reader. This greatly improves the ingestion speed of large CSV files into the system. While we have done our best to make the parallel CSV reader robust – CSV parsing is a minefield as there is such a wide variety of different files out there – so we have marked the reader as experimental for now.

The parallel CSV reader can be enabled by setting the `experimental_parallel_csv` flag to true. We aim to make the parallel CSV reader the default reader in future DuckDB versions.

```
SET experimental_parallel_csv = true;
```

Below is the load time of a 720 MB CSV file containing the `lineitem` table from the TPC-H benchmark,

Variant	Load time
Single-threaded	3.5 s
Parallel	0.6 s

Parallel CREATE INDEX & Index Memory Management Improvements. Index creation is also sped up significantly in this release, as the `CREATE INDEX` statement can now be executed fully in parallel. In addition, the number of memory allocations done by the ART is greatly reduced through inlining of small structures which both reduces memory size and further improves performance.

The timings of creating an index on a single column with 16 million values is shown below.

Version	Create index time
v0.5.1	5.92 s
v0.6.0	1.38 s

Parallel count(DISTINCT). Aggregates containing `DISTINCT` aggregates, most commonly used for exact distinct count computation (e.g., `count(DISTINCT col)`) previously had to be executed in single-threaded mode. Starting with v0.6.0, DuckDB can execute these queries in parallel, leading to large speed-ups.

SQL Syntax Improvements

SQL is the primary way of interfacing with DuckDB – and DuckDB tries to have an easy to use SQL dialect. This release contains further improvements to the SQL dialect.

UNION Type. This release introduces the UNION type, which allows sum types to be stored and queried in DuckDB. For example:

```
CREATE TABLE messages (u UNION(num INTEGER, error VARCHAR));
INSERT INTO messages VALUES (42);
INSERT INTO messages VALUES ('oh my globs');
SELECT * FROM messages;
```

u
42
oh my globs

Sum types are strongly typed – but they allow a single value in a table to be represented as one of various types. The union page in the documentation contains more information on how to use this new composite type.

FROM-first. Starting with this release, DuckDB supports starting queries with the FROM clause instead of the SELECT clause. In fact, the SELECT clause is fully optional now, and defaults to SELECT *. That means the following queries are now valid in DuckDB:

```
-- SELECT clause is optional, SELECT * is implied (if not included)
FROM tbl;

-- first 5 rows of the table
FROM tbl LIMIT 5;

-- SELECT can be used after the FROM
FROM tbl SELECT l_orderkey;

-- insert all data from tbl1 into tbl2
INSERT INTO tbl2 FROM tbl1;
```

COLUMNS Expression. This release adds support for the COLUMNS expression, inspired by the ClickHouse syntax. The COLUMNS expression allows you to execute expressions or functions on multiple columns without having to duplicate the full expression.

```
CREATE TABLE obs (id INTEGER, val1 INTEGER, val2 INTEGER);
INSERT INTO obs VALUES (1, 10, 100), (2, 20, NULL), (3, NULL, 300);
SELECT min(COLUMNS(*)), count(*) FROM obs;
```

min(obs.id)	min(obs.val1)	min(obs.val2)	count_star()
1	10	100	3

The COLUMNS expression supports all star expressions, including the EXCLUDE and REPLACE syntax. In addition, the COLUMNS expression can take a regular expression as parameter:

```
SELECT COLUMNS('val[0-9]+') FROM obs;
```

val1	val2
10	100
20	NULL
NULL	300

List comprehension support. List comprehension is an elegant and powerful way of defining operations on lists. DuckDB now also supports list comprehension as part of its SQL dialect. For example, the query below now works:

```
SELECT [x + 1 for x in [1, 2, 3]] AS l;
```

```
┌────────────┐
│     l      │
├────────────┤
│ [2, 3, 4]  │
└────────────┘
```

Nested types and structures are very efficiently implemented in DuckDB, and are now also more elegant to work with.

Memory Management Improvements

When working with large data sets, memory management is always a potential pain point. By using a streaming execution engine and buffer manager, DuckDB supports many operations on larger than memory data sets. DuckDB also aims to support queries where *intermediate* results do not fit into memory by using disk-spilling techniques, and has support for an efficient out-of-core sort, out-of-core window functions and an out-of-core hash join.

This release further improves on that by greatly optimizing the out-of-core hash join, resulting in a much more graceful degradation in performance as the data exceeds the memory limit.

Memory limit (GB)	Old time (s)	New time (s)
10	1.97	1.96
9	1.97	1.97
8	2.23	2.22
7	2.23	2.44
6	2.27	2.39
5	2.27	2.32
4	2.81	2.45
3	5.60	3.20
2	7.69	3.28
1	17.73	4.35

jemalloc. In addition, this release bundles the jemalloc allocator with the Linux version of DuckDB by default, which fixes an outstanding issue where the standard `GLIBC` allocator would not return blocks to the operating system, unnecessarily leading to out-of-memory errors on the Linux version. Note that this problem does not occur on macOS or Windows, and as such we continue using the standard allocators there (at least for now).

Shell Improvements

DuckDB has a command-line interface that is adapted from SQLite's command line interface, and therefore supports an extremely similar interface to SQLite. All of the tables in this blog post have been generated using the `.mode markdown` in the CLI.

The DuckDB shell also offers several improvements over the SQLite shell, such as syntax highlighting, and this release includes a few new goodies.

DuckBox Rendering. This release includes a new `.mode duckbox` rendering that is used by default. This box rendering adapts to the size of the shell, and leaves out columns and rows to provide a better overview of a result. It very quickly renders large result sets by leaving

out rows in the middle. That way, typing `SELECT * FROM tbl` in the shell no longer blows it up. In fact, this can now be used to quickly get a good feel of a dataset instead.

The number of rows that are rendered can be changed by using the `.maxrows X` setting, and you can switch back to the old rendering using the `.mode box` command.

```
SELECT * FROM '~/Data/nyctaxi/nyc-taxi/2014/04/data.parquet';
```

vendor_id varchar	pickup_at timestamp	dropoff_at timestamp	…	tip_amount float	tolls_amount float	total_amount float
CMT	2014-04-08 08:59:39	2014-04-08 09:28:57	…	3.7	0.0	22.2
CMT	2014-04-08 14:59:22	2014-04-08 15:04:52	…	1.3	0.0	7.8
CMT	2014-04-08 08:45:28	2014-04-08 08:50:41	…	1.2	0.0	7.2
CMT	2014-04-08 08:00:20	2014-04-08 08:11:31	…	1.7	0.0	10.2
CMT	2014-04-08 08:38:36	2014-04-08 08:44:37	…	1.2	0.0	7.2
CMT	2014-04-08 07:52:53	2014-04-08 07:59:12	…	1.3	0.0	7.8
CMT	2014-04-08 16:08:16	2014-04-08 16:12:38	…	1.4	0.0	8.4
CMT	2014-04-08 12:04:09	2014-04-08 12:14:30	…	1.7	0.0	10.2
CMT	2014-04-08 16:18:38	2014-04-08 16:37:04	…	2.5	0.0	17.5
CMT	2014-04-08 15:28:00	2014-04-08 15:34:44	…	1.4	0.0	8.4
·	·	·	·	·	·	·
·	·	·	·	·	·	·
·	·	·	·	·	·	·
CMT	2014-04-25 00:09:34	2014-04-25 00:14:52	…	2.5	0.0	10.0
CMT	2014-04-25 01:59:39	2014-04-25 02:16:07	…	3.5	0.0	21.0
CMT	2014-04-24 23:02:08	2014-04-24 23:47:10	…	8.8	0.0	52.8
CMT	2014-04-25 01:27:11	2014-04-25 01:56:53	…	4.6	0.0	27.6
CMT	2014-04-25 00:15:46	2014-04-25 00:25:37	…	1.0	0.0	11.5
CMT	2014-04-25 00:17:53	2014-04-25 00:22:52	…	1.3	0.0	7.8
CMT	2014-04-25 03:13:19	2014-04-25 03:21:50	…	2.1	0.0	12.6
CMT	2014-04-24 23:53:03	2014-04-25 00:16:01	…	2.85	0.0	31.35
CMT	2014-04-25 00:26:08	2014-04-25 00:31:25	…	1.4	0.0	8.4
CMT	2014-04-24 23:21:39	2014-04-24 23:33:57	…	1.0	0.0	11.5

14618759 rows (20 shown) 18 columns (6 shown)

Context-Aware Auto-Complete. The shell now also ships with context-aware auto-complete. Auto-complete is triggered by pressing the tab character. The shell auto-completes four different groups: (1) keywords, (2) table names + table functions, (3) column names + scalar functions, and (4) file names. The shell looks at the position in the SQL statement to determine which of these auto-completions to trigger. For example:

```
S -> SELECT

SELECT s -> student_id

SELECT student_id F -> FROM


SELECT student_id FROM g -> grades

SELECT student_id FROM 'd -> data/

SELECT student_id FROM 'data/ -> data/grades.csv
```

Progress Bars. DuckDB has supported progress bars in queries for a while now, but they have always been opt-in. In this release we have prettied up the progress bar and enabled it by default in the shell. The progress bar will pop up when a query is run that takes more than 2 seconds, and display an estimated time-to-completion for the query.

```
COPY lineitem TO 'lineitem-big.parquet';
```

In the future we aim to enable the progress bar by default in other clients. For now, this can be done manually by running the following SQL queries:

```
PRAGMA enable_progress_bar;
PRAGMA enable_print_progress_bar;
```

Announcing DuckDB 0.7.0

Publication date: 2023-02-13

Author: Mark Raasveldt

The DuckDB team is happy to announce the latest DuckDB version (0.7.0) has been released. This release of DuckDB is named ”Labradorius” after the Labrador Duck (Camptorhynchus labradorius) that was native to North America.

To install the new version, please visit the installation guide. The full release notes can be found on GitHub.

What's in 0.7.0

The new release contains many improvements to the JSON support, new SQL features, improvements to data ingestion and export, and other new features. Below is a summary of the most impactful changes, together with the linked PRs that implement the features.

Data Ingestion/Export Improvements

JSON Ingestion. This version introduces the `read_json` and `read_json_auto` methods. These can be used to ingest JSON files into a tabular format. Similar to `read_csv`, the `read_json` method requires a schema to be specified, while the `read_json_auto` automatically infers the schema of the JSON from the file using sampling. Both new-line delimited JSON and regular JSON are supported.

```
FROM 'data/json/with_list.json';
```

id	name
1	[O, Brother,, Where, Art, Thou?]
2	[Home, for, the, Holidays]
3	[The, Firm]
4	[Broadcast, News]
5	[Raising, Arizona]

Partitioned Parquet/CSV Export. DuckDB has been able to ingest Hive-partitioned Parquet and CSV files for a while. After this release DuckDB will also be able to *write* Hive-partitioned data using the `PARTITION_BY` clause. These files can be exported locally or remotely to S3 compatible storage. Here is a local example:

```
COPY orders TO 'orders' (FORMAT parquet, PARTITION_BY (year, month));
```

This will cause the Parquet files to be written in the following directory structure:

```
orders
├── year=2021
│   ├── month=1
│   │   ├── file1.parquet
│   │   └── file2.parquet
│   └── month=2
│       └── file3.parquet
└── year=2022
    ├── month=11
    │   ├── file4.parquet
    │   └── file5.parquet
    └── month=12
        └── file6.parquet
```

Parallel Parquet/CSV Writing. Parquet and CSV writing are sped up tremendously this release with the parallel Parquet and CSV writer support.

Format	Old	New (8T)
CSV	2.6 s	0.4 s
Parquet	7.5 s	1.3 s

Note that currently the parallel writing is currently limited to non-insertion order preserving – which can be toggled by setting the `preserve_insertion_order` setting to false. In a future release we aim to alleviate this restriction and order parallel insertion order preserving writes as well.

Multi-Database Support

Attach Functionality. This release adds support for attaching multiple databases to the same DuckDB instance. This easily allows data to be transferred between separate DuckDB database files, and also allows data from separate database files to be combined together in individual queries. Remote DuckDB instances (stored on a network accessible location like GitHub, for example) may also be attached.

```
ATTACH 'new_db.db';
CREATE TABLE new_db.tbl (i INTEGER);
INSERT INTO new_db.tbl SELECT * FROM range(1000);
DETACH new_db;
```

See the documentation for more information.

SQLite Storage Back-End. In addition to adding support for attaching DuckDB databases – this release also adds support for *pluggable database engines*. This allows extensions to define their own database and catalog engines that can be attached to the system. Once attached, an engine can support both reads and writes. The SQLite extension makes use of this to add native read/write support for SQLite database files to DuckDB.

```
ATTACH 'sqlite_file.db' AS sqlite (TYPE sqlite);
CREATE TABLE sqlite.tbl (i INTEGER);
INSERT INTO sqlite.tbl VALUES (1), (2), (3);
SELECT * FROM sqlite.tbl;
```

Using this, SQLite database files can be attached, queried and modified as if they are native DuckDB database files. This allows data to be quickly transferred between SQLite and DuckDB – and allows you to use DuckDB's rich SQL dialect to query data stored in SQLite tables.

New SQL Features

Upsert Support. Upsert support is added with this release using the `ON CONFLICT` clause, as well as the `SQLite` compatible `INSERT OR REPLACE`/`INSERT OR IGNORE` syntax.

```
CREATE TABLE movies (id INTEGER PRIMARY KEY, name VARCHAR);
INSERT INTO movies VALUES (1, 'A New Hope');
FROM movies;
```

id	name
1	A New Hope

```
INSERT OR REPLACE INTO movies VALUES (1, 'The Phantom Menace');
FROM movies;
```

id	name
1	The Phantom Menace

See the documentation for more information.

Lateral Joins. Support for lateral joins is added in this release. Lateral joins are a more flexible variant of correlated subqueries that make working with nested data easier, as they allow easier unnesting of nested data.

Positional Joins. While SQL formally models unordered sets, in practice the order of datasets does frequently have a meaning. DuckDB offers guarantees around maintaining the order of rows when loading data into tables or when exporting data back out to a file – as well as when executing queries such as `LIMIT` without a corresponding `ORDER BY` clause.

To improve support for this use case – this release introduces the `POSITIONAL JOIN`. Rather than joining on the values of rows – this new join type joins rows based on their position in the table.

```
CREATE TABLE t1 AS FROM (VALUES (1), (2), (3)) t(i);
CREATE TABLE t2 AS FROM (VALUES (4), (5), (6)) t(k);
SELECT * FROM t1 POSITIONAL JOIN t2;
```

i	k
1	4
2	5
3	6

Python API Improvements

Query Building. This release introduces easier incremental query building using the Python API by allowing relations to be queried. This allows you to decompose long SQL queries into multiple smaller SQL queries, and allows you to easily inspect query intermediates.

```
>>> import duckdb
>>> lineitem = duckdb.sql('FROM lineitem.parquet')
>>> lineitem.limit(3).show()
```

l_orderkey int32	l_partkey int32	l_suppkey int32	…	l_shipinstruct varchar	l_shipmode varchar	l_comment varchar
1	155190	7706	…	DELIVER IN PERSON	TRUCK	egular courts abov…
1	67310	7311	…	TAKE BACK RETURN	MAIL	ly final dependenc…
1	63700	3701	…	TAKE BACK RETURN	REG AIR	riously. regular, …
3 rows						16 columns (6 shown)

```
>>> lineitem_filtered = duckdb.sql('FROM lineitem WHERE l_orderkey>5000')
>>> lineitem_filtered.limit(3).show()
```

l_orderkey int32	l_partkey int32	l_suppkey int32	…	l_shipinstruct varchar	l_shipmode varchar	l_comment varchar
5024	165411	444	…	NONE	AIR	to the expre
5024	57578	84	…	COLLECT COD	REG AIR	osits hinder caref…
5024	111009	3521	…	NONE	MAIL	zle carefully saut…
3 rows						16 columns (6 shown)

```
>>> duckdb.sql('SELECT min(l_orderkey), max(l_orderkey) FROM lineitem_filtered').show()
```

min(l_orderkey) int32	max(l_orderkey) int32
5024	6000000

Note that everything is lazily evaluated. The Parquet file is not read from disk until the final query is executed – and queries are optimized in their entirety. Executing the decomposed query will be just as fast as executing the long SQL query all at once.

Python Ingestion APIs. This release adds several familiar data ingestion and export APIs that follow standard conventions used by other libraries. These functions emit relations as well – which can be directly queried again.

```
>>> lineitem = duckdb.read_csv('lineitem.csv')
>>> lineitem.limit(3).show()
```

l_orderkey int32	l_partkey int32	l_suppkey int32	…	l_shipinstruct varchar	l_shipmode varchar	l_comment varchar
1	155190	7706	…	DELIVER IN PERSON	TRUCK	egular courts abov…
1	67310	7311	…	TAKE BACK RETURN	MAIL	ly final dependenc…
1	63700	3701	…	TAKE BACK RETURN	REG AIR	riously. regular, …
3 rows						16 columns (6 shown)

```
>>> duckdb.sql('SELECT min(l_orderkey) FROM lineitem').show()
```

min(l_orderkey) int32
1

Polars Integration. This release adds support for tight integration with the Polars DataFrame library, similar to our integration with Pandas DataFrames. Results can be converted to Polars DataFrames using the `.pl()` function.

```
import duckdb
duckdb.sql('SELECT 42').pl()
```

```
shape: (1, 1)
```

42
i32
42

In addition, Polars DataFrames can be directly queried using the SQL interface.

```
import duckdb
import polars as pl
df = pl.DataFrame({'a': 42})
duckdb.sql('SELECT * FROM df').pl()
```

```
shape: (1, 1)
```

a
i64
42

fsspec Filesystem Support. This release adds support for the fsspec filesystem API. fsspec allows users to define their own filesystem that they can pass to DuckDB. DuckDB will then use this file system to read and write data to and from. This enables support for storage back-ends that may not be natively supported by DuckDB yet, such as FTP.

```
import duckdb
from fsspec import filesystem

duckdb.register_filesystem(filesystem('gcs'))

data = duckdb.query("SELECT * FROM read_csv_auto('gcs:///bucket/file.csv')").fetchall()
```

Have a look at the guide for more information

Storage Improvements

Delta Compression. Compression of numeric values in the storage is improved using the new delta and delta-constant compression. This compression method is particularly effective when compressing values that are equally spaced out. For example, sequences of numbers (`1, 2, 3, ...`) or timestamps with a fixed interval between them (`12:00:01, 12:00:02, 12:00:03, ...`).

Final Thoughts

The full release notes can be found on GitHub. We would like to thank all of the contributors for their hard work on improving DuckDB.

JupySQL Plotting with DuckDB

Publication date: 2023-02-24

Author: Guest post by Eduardo Blancas

TL;DR: JupySQL provides a seamless SQL experience in Jupyter and uses DuckDB to visualize larger than memory datasets in matplotlib.

Introduction

Data visualization is essential for every data practitioner since it allows us to find patterns that otherwise would be hard to see. The typical approach for plotting tabular datasets involves Pandas and Matplotlib. However, this technique quickly falls short as our data grows, given that Pandas introduces a significant memory overhead, making it challenging even to plot a medium-sized dataset.

In this blog post, we'll use JupySQL and DuckDB to efficiently plot *larger-than-memory* datasets in our laptops. JupySQL is a fork of ipython-sql, which adds SQL cells to Jupyter, that is being actively maintained and enhanced by the team at Ploomber.

Combining JupySQL with DuckDB enables a powerful and user friendly local SQL processing experience, especially when combined with JupySQL's new plotting capabilities. There is no need to get beefy (and expensive!) EC2 machines or configure complex distributed frameworks! Get started with JupySQL and DuckDB with our Jupyter Notebook guide, or go directly to an example collab notebook!

We want JupySQL to offer the best SQL experience in Jupyter, so if you have any feedback, please open an issue on GitHub!

The Problem

One significant limitation when using `pandas` and `matplotlib` for data visualization is that we need to load all our data into memory, making it difficult to plot *larger-than-memory* datasets. Furthermore, given the overhead that `pandas` introduces, we might be unable to visualize some smaller datasets that we might think "fit" into memory.

Let's load a sample `.parquet` dataset using pandas to show the memory overhead:

```
from urllib.request import urlretrieve

_ = urlretrieve("https://d37ci6vzurychx.cloudfront.net/trip-data/yellow_tripdata_2022-01.parquet",
                "yellow_tripdata_2022-01.parquet")
```

The downloaded `.parquet` file takes 36 MB of disk space:

```
ls -lh *.parquet
```

```
-rw-r--r--  1 eduardo  staff    36M Jan 18 14:45 yellow_tripdata_2022-01.parquet
```

Now let's load the `.parquet` as a data frame and see how much memory it takes:

```
import pandas as pd

df = pd.read_parquet("yellow_tripdata_2022-01.parquet")

df_mb = df.memory_usage().sum() / (1024 ** 2)
print(f"Data frame takes {df_mb:.0f} MB")
```

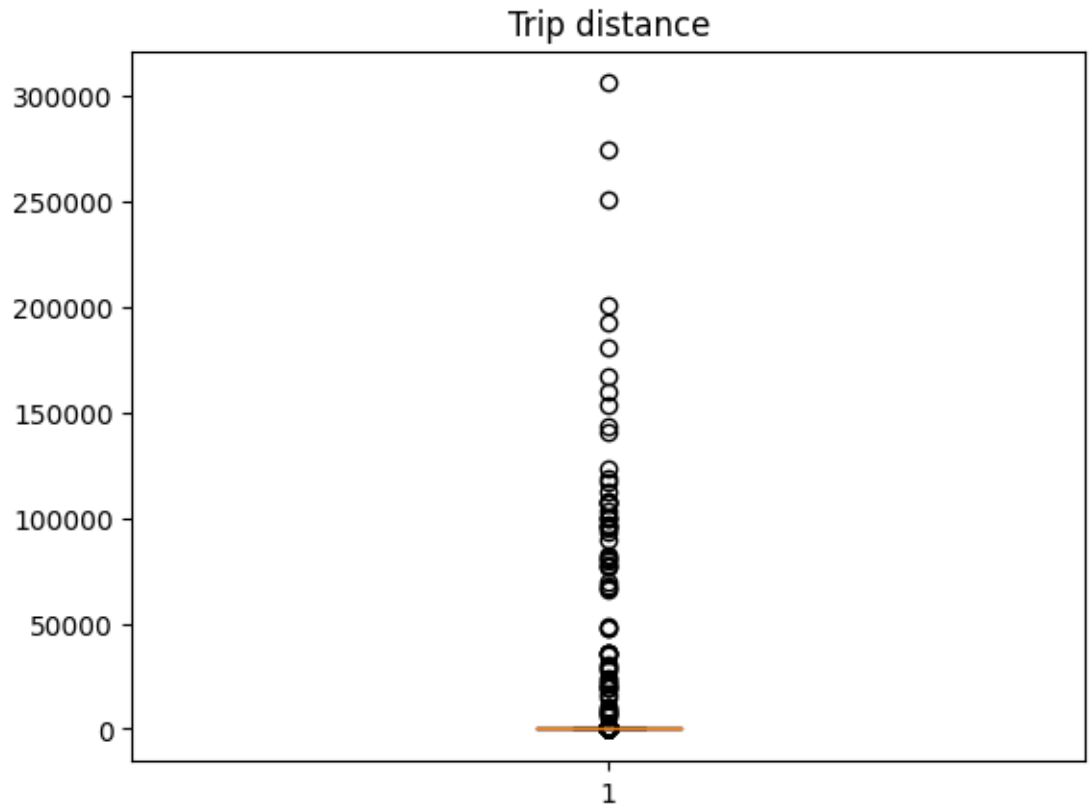
```
Data frame takes 357 MB
```

As you can see, we're using almost 10× as much memory as the file size. Given this overhead, we must be much more conservative about what *larger-than-memory* means, as "medium" files might not fit into memory once loaded. But this is just the beginning of our memory problems.

When plotting data, we often need to preprocess it before it's suitable for visualization. However, if we're not careful, these preprocessing steps will copy our data, dramatically increasing memory. Let's show a practical example.

Our sample dataset contains an observation for each NYC yellow cab trip in January 2022. Let's create a boxplot for the trip's distance:

```
import matplotlib.pyplot as plt

plt.boxplot(df.trip_distance)
_ = plt.title("Trip distance")
```



Wow! It looks like some new yorkers really like taxi rides! Let's put the taxi fans aside to improve the visualization and compute the 99th percentile to use it as the cutoff value:

```
cutoff = df.trip_distance.quantile(q=0.99)
cutoff
```

```
19.7
```

Now, we need to filter out observations larger than the cutoff value; but before we do it, let's create a utility function to capture memory usage:

```
import psutil

def memory_used():
    """Returns memory used in MB"""
    mem = psutil.Process().memory_full_info().uss / (1024 ** 2)
    print(f"Memory used: {mem:.0f} MB")

memory_used()
```

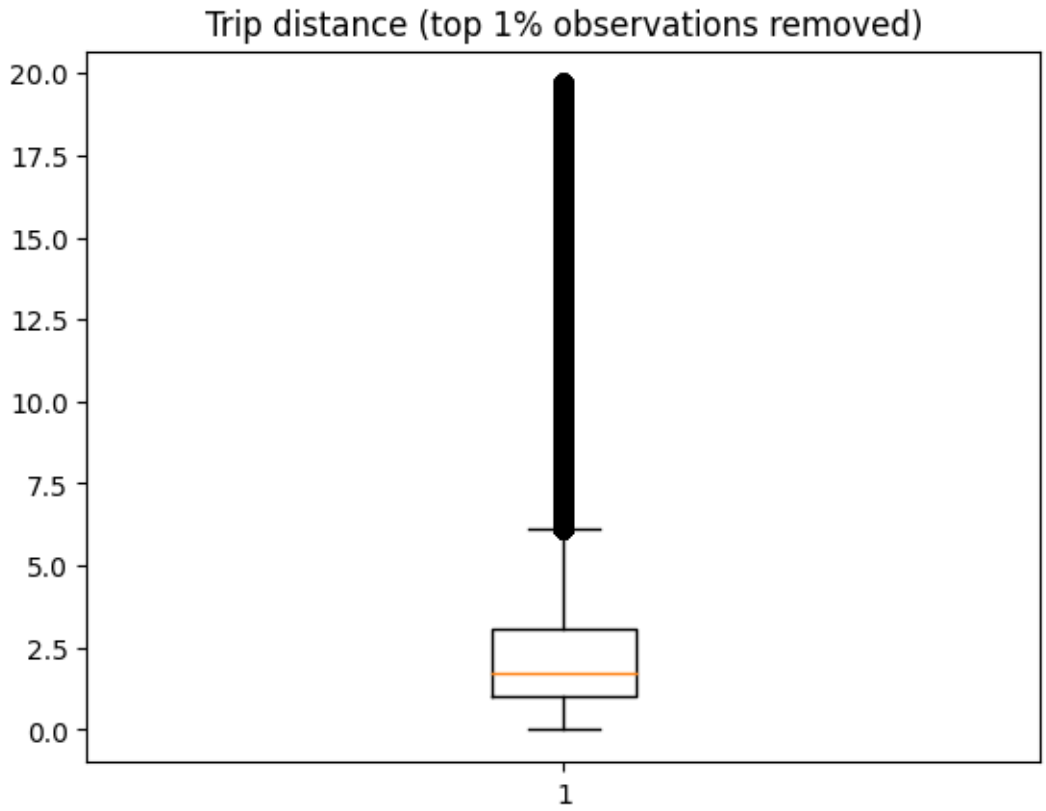
```
Memory used: 941 MB
```

Let's now filter out the observations:

```
df_ = df[df.trip_distance < cutoff]
```

Plot the histogram:

```
plt.boxplot(df_.trip_distance)
_ = plt.title("Trip distance (top 1% observations removed)")
```



We now see more reasonable numbers with the top 1% outliers removed. There are a few trips over 10 miles (perhaps some uptown new yorkers going to Brooklyn for some delicious pizza?)

How much memory are we using now?

```
memory_used()
```

```
Memory used: 1321 MB
```

380 MB more! Loading a 36 MB Parquet file turned into >700 MB in memory after loading and applying one preprocessing step!

So, in reality, when we use `pandas`, what fits in memory is much smaller than we think, and even with a laptop equipped with 16 GB of RAM, we'll be extremely limited in terms of what size of a dataset we process. Of course, we could save a lot of memory by exclusively loading the column we plotted and deleting unneeded data copies; however, let's face it, *this never happens in practice*. When exploring data, we rarely know ahead of time which columns we'll need; furthermore, our time is better spent analyzing and visualizing the data than manually deleting data copies.

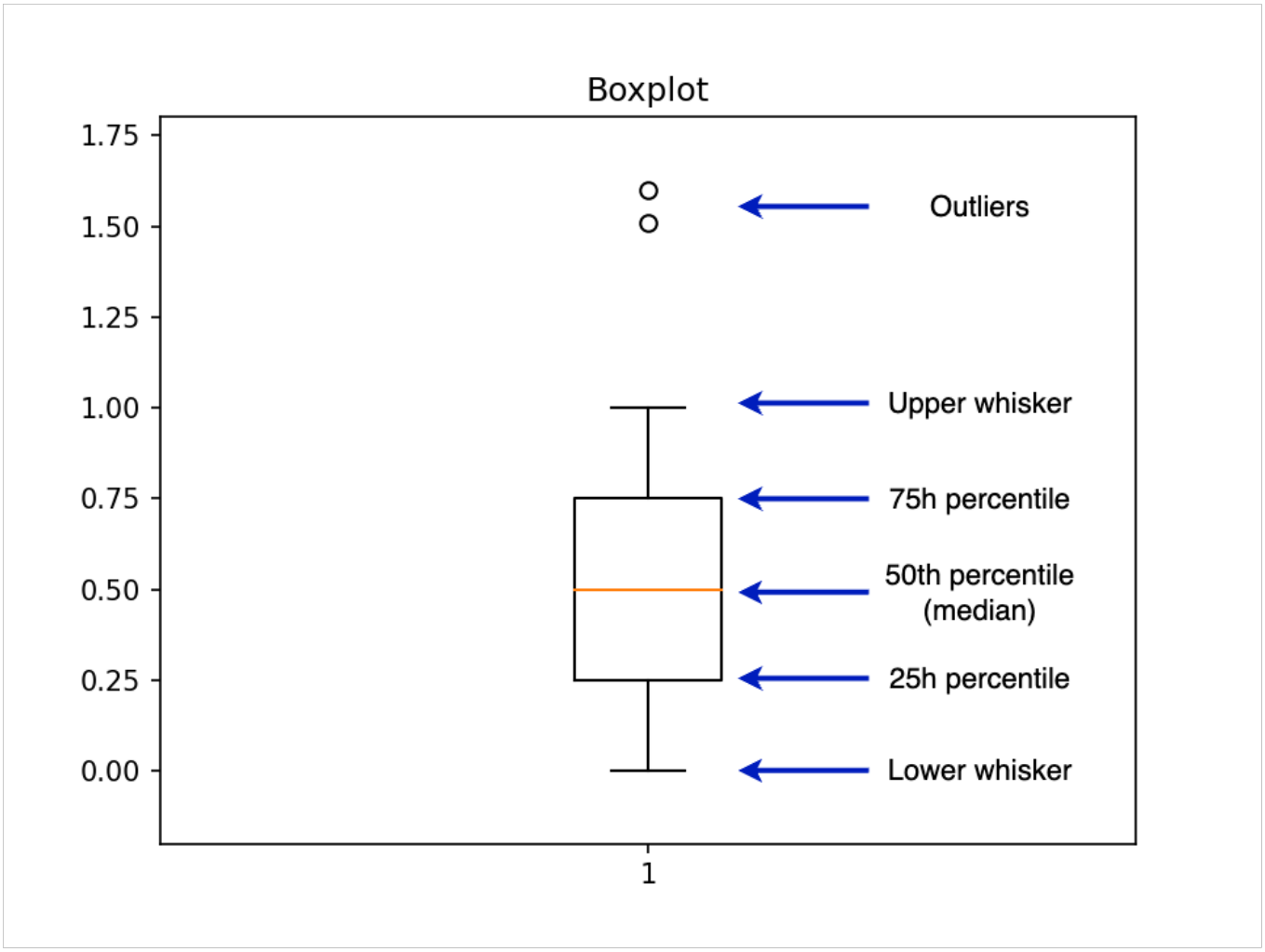
When facing this challenge, we might consider using a distributed framework; however, this adds so much complexity to the process, and it only partially solves the problem since we'd need to write code to compute the statistics in a distributed fashion. Alternatively, we might consider getting a larger machine, a relatively straightforward (but expensive!) approach if we can access cloud resources. However, this still requires us to move our data, set up a new environment, etc. Fortunately for us, there's DuckDB!

DuckDB: A Highly Scalable Backend for Statistical Visualizations

When using functions such as `hist` (histogram) or `boxplot`, `matplotlib` performs two steps:

1. Compute summary statistics
2. Plot data

For example, `boxplot` calls another function called `boxplot_stats` that returns the statistics required to draw the plot. To create a boxplot, we need several summary statistics, such as the 25th percentile, 50th percentile, and 75th percentile. The following diagram shows a boxplot along with the labels for each part:



The bottleneck in the `pandas` + `matplotlib` approach is the `boxplot_stats` function since it requires a `numpy.array` or `pandas.Series` as an input, forcing us to load all our data into memory. However, we can implement a new version of `boxplot_stats` that pushes the data aggregation step to another analytical engine.

We chose DuckDB because it's extremely powerful and easy to use. There is no need to spin up a server or manage complex configurations: install it with `pip install`, point it to your data files, and that's it; you can start aggregating millions and millions of data points in no time!

You can see the full implementation here; essentially, we translated matplotlib's `boxplot_stats` from Python into SQL. For example, the following query will compute the three percentiles we need: 25th, 50th (median), and 75th:

```
%load_ext sql
%sql duckdb://

%%sql
```

```
-- We calculate the percentiles all at once and
-- then convert from list format into separate columns
-- (Improving performance by reducing duplicate work)
WITH stats AS (
  SELECT
    percentile_disc([0.25, 0.50, 0.75]) WITHIN GROUP
      (ORDER BY "trip_distance") AS percentiles
  FROM 'yellow_tripdata_2022-01.parquet'
)
SELECT
  percentiles[1] AS q1,
  percentiles[2] AS median,
  percentiles[3] AS q3
FROM stats;
```

Once we compute all the statistics, we call the `bxp` function, which draws the boxplot from the input statistics.

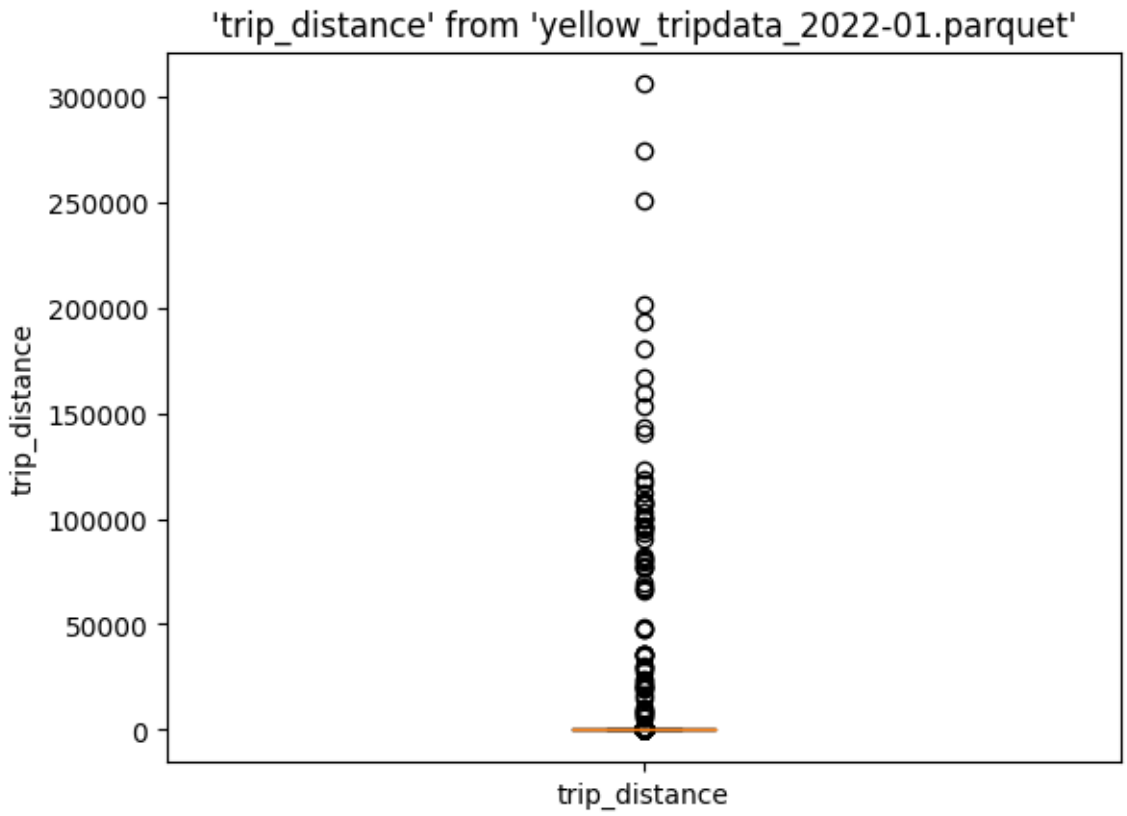
This process is already implemented in JupySQL, and you can create a boxplot with the `%sqlplot boxplot` command. Let's see how. But first, let's check how much memory we're using, so we can compare it to the `pandas` version:

```
memory_used()
```

```
Memory used: 1351 MB
```

Let's create the boxplot:

```
%sqlplot boxplot --table yellow_tripdata_2022-01.parquet --column trip_distance
```



Again, we see all these outliers. Let's compute the cutoff value:

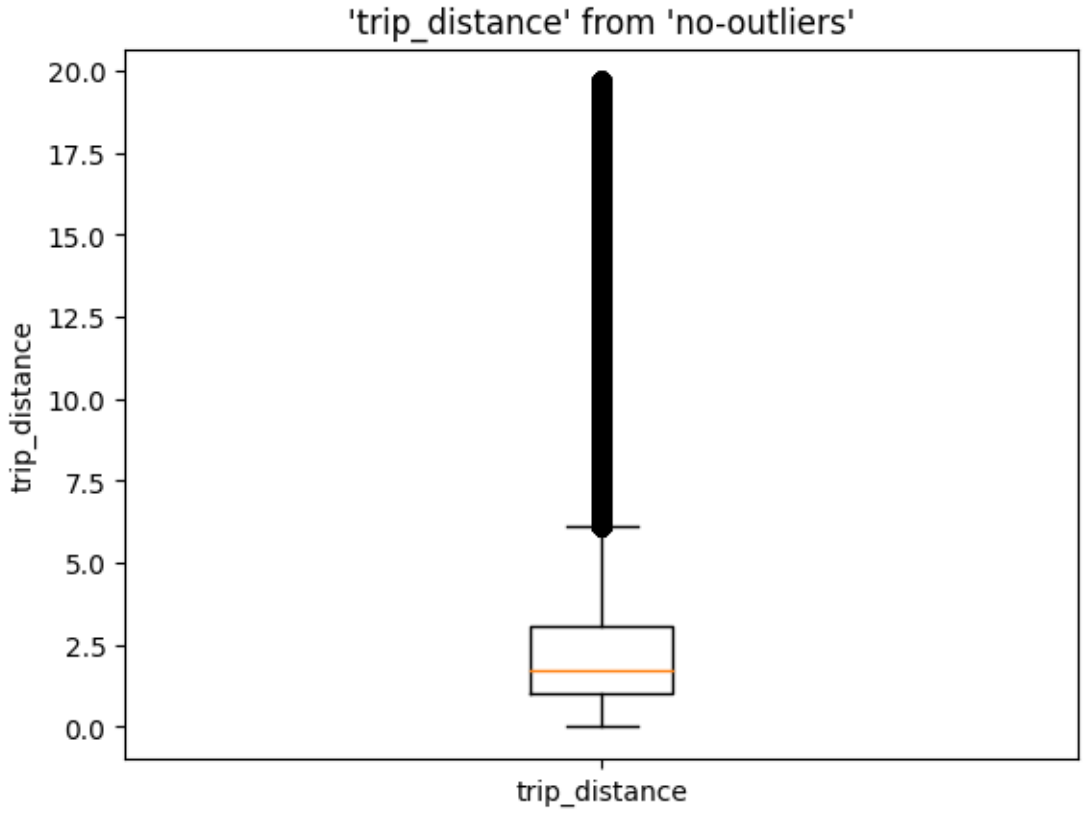
```
%%sql
SELECT percentile_disc(0.99) WITHIN GROUP (ORDER BY trip_distance)
FROM 'yellow_tripdata_2022-01.parquet'
```

Let's define a query that filters out the top 1% of observations. The `--save` option allows us to store this SQL expression and we choose not to execute it.

```
%%sql --save no-outliers --no-execute
SELECT *
FROM 'yellow_tripdata_2022-01.parquet'
WHERE trip_distance < 19.7;
```

We can now use `no-outliers` in our `%sqlplot boxplot` command:

```
%sqlplot boxplot --table no-outliers --column trip_distance --with no-outliers
```




```
memory_used()
```

```
Memory used: 1375 MB
```

Memory usage remained pretty much the same (23 MB difference, mostly due to the newly imported modules). Since we're relying on DuckDB for the data aggregation step, the SQL engine takes care of loading, aggregating, and freeing up memory as soon as we're done; this is much more efficient than loading all our data at the same time and keeping unwanted data copies!

Using DuckDB to Compute Histogram Statistics

We can extend our recipe to other statistical visualizations, such as histograms.

A histogram allows us to visualize the distribution of a dataset, enabling us to find patterns such as modality, outliers, range of values, etc. Like with the boxplot, when using `pandas` + `matplotlib`, creating a histogram involves loading all our data at once into memory; then, `matplotlib` aggregates and plots it.

In our case, we'll push the aggregation to DuckDB, which will compute the bin positions (X-axis) and heights (Y-axis), then we'll pass this to maptlotlib's `bar` function to create the histogram.

The implementation involves two steps.

First, given the number of bins chosen by the user (`N_BINS`), we compute the `BIN_SIZE`:

```
%%sql
SELECT (max(trip_distance) - min(trip_distance)) / N_BINS
FROM 'yellow_tripdata_2022-01.parquet';
```

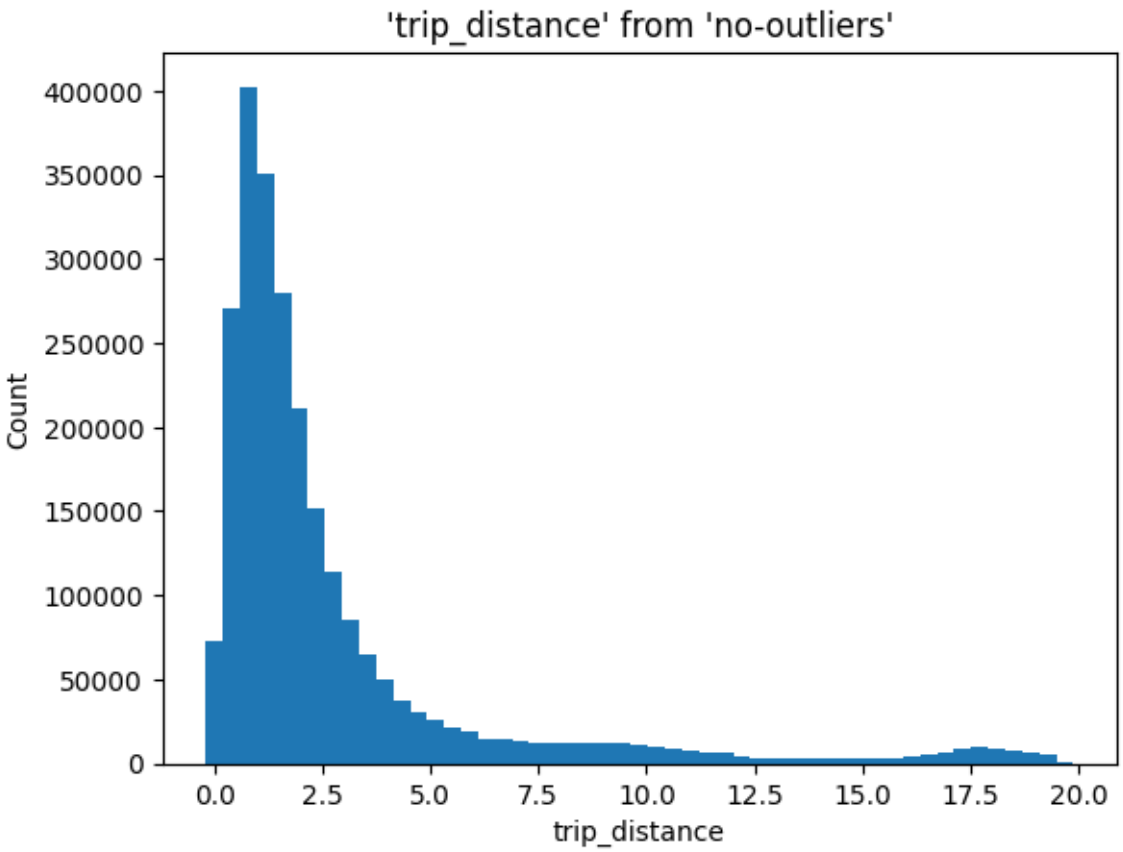
Then, using the `BIN_SIZE`, we find the number of observations that fall into each bin:

```
%%sql
SELECT
    floor("trip_distance" / BIN_SIZE) * BIN_SIZE,
    count(*) AS count
FROM 'yellow_tripdata_2022-01.parquet'
GROUP BY 1
ORDER BY 1;
```

The intuition for the second query is as follows: given that we have `N_BINS`, the `floor("trip_distance" / BIN_SIZE)` portion will assign each observation to their corresponding bin (1, 2, ..., `N_BINS`), then, we multiply by the bin size to get the value in the X axis, while the count represents the value in the Y axis. Once we have that, we call the `bar` plotting function.

All these steps are implemented in the `%sqplot histogram` command:

```
%sqlplot histogram --table no-outliers --column trip_distance --with no-outliers
```



Final Thoughts

This blog post demonstrated a powerful approach for plotting large datasets powered using JupySQL and DuckDB. If you need to visualize large datasets, DuckDB offers unmatched simplicity and flexibility!

At Ploomber, we're working on building a full-fledged SQL client for Jupyter! Exciting features like automated dataset profiling, autocompletion, and more are coming! So keep an eye on updates! If there are features you think we should add to offer the best SQL experience in Jupyter, please open an issue!

JupySQL is an actively maintained fork of `ipython-sql`, and it keeps full compatibility with it. If you want to learn more, check out the GitHub repository and the documentation.

Try It Out

To try it yourself, check out this collab notebook, or here's a snippet you can paste into Jupyter:

```
from urllib.request import urlretrieve

urlretrieve("https://d37ci6vzurychx.cloudfront.net/trip-data/yellow_tripdata_2022-01.parquet",
            "yellow_tripdata_2022-01.parquet")

%pip install jupysql duckdb-engine --quiet
%load_ext sql
%sql duckdb://
%sqlplot boxplot --table yellow_tripdata_2022-01.parquet --column trip_distance
```

Note: the commands that begin with % or %% will only work on Jupyter/IPython. If you want to try this in a regular Python session, check out the Python API.

Shredding Deeply Nested JSON, One Vector at a Time

Publication date: 2023-03-03

Author: Laurens Kuiper

TL;DR: We recently improved DuckDB's JSON extension so JSON files can be directly queried as if they were tables.

> We updated this blog post in December 2024 to reflect the changes in DuckDB's JSON syntax.

DuckDB has a JSON extension that can be installed and loaded through SQL:

```
INSTALL 'json';
LOAD 'json';
```

The JSON extension supports various functions to create, read, and manipulate JSON strings. These functions are similar to the JSON functionality provided by other databases such as PostgreSQL and MySQL. DuckDB uses yyjson internally to parse JSON, a high-performance JSON library written in ANSI C. Many thanks to the yyjson authors and contributors!

Besides these functions, DuckDB is now able to read JSON directly! This is done by automatically detecting the types and column names, then converting the values within the JSON to DuckDB's vectors. The automated schema detection dramatically simplifies working with JSON data and subsequent queries on DuckDB's vectors are significantly faster!

Reading JSON Automatically with DuckDB

Since version 0.7.0, DuckDB supports JSON table functions. To demonstrate these, we will read `todos.json`, a fake TODO list containing 200 fake TODO items (only the first two items are shown):

```
[
  {
    "userId": 1,
    "id": 1,
    "title": "delectus aut autem",
    "completed": false
  },
  {
    "userId": 1,
    "id": 2,
    "title": "quis ut nam facilis et officia qui",
    "completed": false
  },
  ...
]
```

Each TODO item is an entry in the JSON array, but in DuckDB, we'd like a table where each entry is a row. This is as easy as:

```
SELECT * FROM 'todos.json' LIMIT 5;
```

userId	id	title	completed
1	1	delectus aut autem	false
1	2	quis ut nam facilis et officia qui	false
1	3	fugiat veniam minus	false
1	4	et porro tempora	true
1	5	laboriosam mollitia et enim quasi adipisci quia provident illum	false

Finding out which user completed the most TODO items is as simple as:

```
SELECT userId, sum(completed::INTEGER) total_completed
FROM 'todos.json'
GROUP BY userId
ORDER BY total_completed DESC
LIMIT 1;
```

userId	total_completed
5	12

Under the hood, DuckDB recognizes the `.json` file extension in `'todos.json'`, and calls `read_json('todos.json')` instead. This function is similar to our `read_csv` function, which automatically infers column names and types for CSV files.

Like our other table functions, `read_json` supports reading multiple files by passing a list, e.g., `read_json(['file1.json', 'file2.json'])`, but also globbing, e.g., `read_json('file*.json')`. DuckDB will read multiple files in parallel.

Newline Delimited JSON

Not all JSON adheres to the format used in `todos.json`, which is an array of 'records'. Newline-delimited JSON, or NDJSON, stores each row on a new line. DuckDB also supports reading (and writing!) this format. First, let's write our TODO list as NDJSON:

```
COPY (SELECT * FROM 'todos.json') to 'todos2.json';
```

Again, DuckDB recognizes the `.json` suffix in the output file and automatically infers that we mean to use `(FORMAT json)`. The created file looks like this (only the first two records are shown):

```
{"userId":1,"id":1,"title":"delectus aut autem","completed":false}
{"userId":1,"id":2,"title":"quis ut nam facilis et officia qui","completed":false}
...
```

DuckDB can read this file in exactly the same way as the original one:

```
SELECT * FROM 'todos2.json';
```

If your JSON file is newline-delimited, DuckDB can parallelize reading. This can be specified by calling `read_ndjson` or passing the `records = true` parameter to `read_json`:

```
SELECT * FROM read_ndjson('todos2.json');
SELECT * FROM read_json('todos2.json', records = true);
```

You can also set `records = auto` to auto-detect whether the JSON file is newline-delimited.

Other JSON Formats

When the `read_json` function is used directly, the format of the JSON can be specified using the `format` parameter. This parameter defaults to `'auto'`, which tells DuckDB to infer what kind of JSON we are dealing with. The first `format` is `'array'`, while the second is `'nd'`. This can be specified like so:

```
SELECT * FROM read_json('todos.json', format = 'array');
SELECT * FROM read_json('todos2.json', format = 'nd');
```

Another supported format is `unstructured`. With this format, records are not required to be a JSON object but can also be a JSON array, string, or anything supported in JSON.

Manual Schemas

What you may also have noticed is the `auto_detect` parameter. This parameter tells DuckDB to infer the schema, i.e., determine the names and types of the returned columns. These can manually be specified like so:

```
SELECT *
FROM read_json(
    'todos.json',
    columns = {userId: 'INTEGER', id: 'INTEGER', title: 'VARCHAR', completed: 'BOOLEAN'},
    format = 'array'
);
```

You don't have to specify all fields, just the ones you're interested in:

```
SELECT *
FROM read_json(
    'todos.json',
    columns = {userId: 'INTEGER', completed: 'BOOLEAN'},
    format = 'array'
);
```

Now that we know how to use the new DuckDB JSON table functions let's dive into some analytics!

GitHub Archive Examples

GH Archive is a project to record the public GitHub timeline, archive it, and make it easily accessible for further analysis. Every hour, a GZIP compressed, newline-delimited JSON file containing all public events on GitHub is uploaded. I downloaded a whole day (2023-02-08) of activity using `wget` and stored the 24 files (starting with `2023-02-08-0.json.gz`) in a directory called `gharchive_gz`. You can get the full day's archive as `gharchive-2023-02-08.zip`:

Keep in mind that the data is compressed:

```
du -sh gharchive_gz
```

```
2.3G  gharchive_gz
```

Decompressed, one day's worth of GitHub activity amounts to more than 18 GB of JSON.

```
gunzip -dc gharchive_gz/* | wc -c
```

```
18396198934
```

To get a feel of what the data looks like, we run the following query:

```
SELECT json_group_structure(json)
FROM (
    SELECT *
    FROM read_ndjson_objects('gharchive_gz/*.json.gz')
    LIMIT 2048
);
```

Here, we use our `read_ndjson_objects` function, which reads the JSON objects in the file as raw JSON, i.e., as strings. The query reads the first 2048 records of JSON from the JSON files `gharchive_gz` directory and describes the structure. You can also directly query the JSON files from GH Archive using DuckDB's `httpfs` extension, but we will be querying the files multiple times, so it is better to download them in this case.

> **Tip.** In the CLI client, you can use `.mode line` to make the output easier to read.

I formatted the result using an online JSON formatter & validator:

```
{
   "id":"VARCHAR",
   "type":"VARCHAR",
   "actor":{
      "id":"UBIGINT",
      "login":"VARCHAR",
      "display_login":"VARCHAR",
      "gravatar_id":"VARCHAR",
      "url":"VARCHAR",
      "avatar_url":"VARCHAR"
   },
   "repo":{
      "id":"UBIGINT",
      "name":"VARCHAR",
      "url":"VARCHAR"
   },
   "payload":{"..."},
   "public":"BOOLEAN",
   "created_at":"VARCHAR",
   "org":{
      "id":"UBIGINT",
      "login":"VARCHAR",
      "gravatar_id":"VARCHAR",
      "url":"VARCHAR",
      "avatar_url":"VARCHAR"
```

```
   }
}
```

I left "payload" out because it consists of deeply nested JSON, and its formatted structure takes up more than 1000 lines!

So, how many records are we dealing with exactly? Let's count it using DuckDB:

```
SELECT count(*) AS count
FROM 'gharchive_gz/*.json.gz';
```

count
4434953

That's around 4.4M daily events, which amounts to almost 200K events per hour. This query takes around 7.3 seconds on my laptop, a 2020 MacBook Pro with an M1 chip and 16 GB of memory. This is the time it takes to decompress the GZIP compression and parse every JSON record.

To see how much time is spent decompressing GZIP in the query, I also created a `gharchive` directory containing the same data but uncompressed. Running the same query on the uncompressed data takes around 5.4 seconds, almost 2 seconds faster. We got faster, but we also read more than 18 GB of data from storage, as opposed to 2.3 GB when it was compressed. So, this comparison really depends on the speed of your storage. I prefer to keep the data compressed.

As a side note, the speed of this query really shows how fast yyjson is!

So, what kind of events are in the GitHub data?

```
SELECT type, count(*) AS count
FROM 'gharchive_gz/*.json.gz'
GROUP BY type
ORDER BY count DESC;
```

type	count
PushEvent	2359096
CreateEvent	624062
PullRequestEvent	366090
IssueCommentEvent	238660
WatchEvent	231486
DeleteEvent	154383
PullRequestReviewEvent	131107
IssuesEvent	88917
PullRequestReviewCommentEvent	79540
ForkEvent	64233
CommitCommentEvent	36823
ReleaseEvent	23004
MemberEvent	14872
PublicEvent	14500
GollumEvent	8180

This query takes around 7.4 seconds, not much more than the `count(*)` query. So as we can see, data analysis is very fast once everything has been decompressed and parsed.

The most common event type is the `PushEvent`, taking up more than half of all events, unsurprisingly, which is people pushing their committed code to GitHub. The least common event type is the `GollumEvent`, taking up less than 1% of all events, which is a creation or update of a wiki page.

If we want to analyze the same data multiple times, decompressing and parsing every time is redundant and slows down the analysis. Instead, we can create a DuckDB table like so:

```
CREATE TABLE events AS
    SELECT * EXCLUDE (payload)
    FROM 'gharchive_gz/*.json.gz';
```

This takes around 9 seconds if you're using an in-memory database. If you're using an on-disk database, this takes around 13 seconds and results in a database size of 444 MB. When using an on-disk database, DuckDB ensures the table is persistent and performs all kinds of compression. Note that we have temporarily ignored the `payload` field using the convenient `EXCLUDE` clause.

To get a feel of what we've read, we can ask DuckDB to describe the table:

```
DESCRIBE SELECT * FROM events;
```

This gives us the following:

cid	name	type	notnull	dflt_value	pk
0	id	BIGINT	false		false
1	type	VARCHAR	false		false
2	actor	STRUCT(id UBIGINT, login VARCHAR, display_login VARCHAR, gravatar_id VARCHAR, url VARCHAR, avatar_url VARCHAR)	false		false
3	repo	STRUCT(id UBIGINT, name VARCHAR, url VARCHAR)	false		false
4	public	BOOLEAN	false		false
5	created_at	TIMESTAMP	false		false
6	org	STRUCT(id UBIGINT, login VARCHAR, gravatar_id VARCHAR, url VARCHAR, avatar_url VARCHAR)	false		false

As we can see, the `"actor"`, `"repo"` and `"org"` fields, which are JSON objects, have been converted to DuckDB structs. The `"id"` column was a string in the original JSON but has been converted to a `BIGINT` by DuckDB's automatic type detection. DuckDB can also detect a few different `DATE`/`TIMESTAMP` formats within JSON strings, as well as `TIME` and `UUID`.

Now that we created the table, we can analyze it like any other DuckDB table! Let's see how much activity there was in the `duckdb/duckdb` GitHub repository on this specific day:

```
SELECT type, count(*) AS count
FROM events
WHERE repo.name = 'duckdb/duckdb'
GROUP BY type
ORDER BY count DESC;
```

type	count
PullRequestEvent	35
IssueCommentEvent	30
WatchEvent	29
PushEvent	15

type	count
PullRequestReviewEvent	14
IssuesEvent	9
PullRequestReviewCommentEvent	7
ForkEvent	3

That's a lot of pull request activity! Note that this doesn't mean that 35 pull requests were opened on this day, activity within a pull request is also counted. If we search through the pull requests for that day, we see that there are only 15. This is more activity than normal because most of the DuckDB developers were busy fixing bugs for the 0.7.0 release.

Now, let's see who was the most active:

```
SELECT actor.login, count(*) AS count
FROM events
WHERE repo.name = 'duckdb/duckdb'
  AND type = 'PullRequestEvent'
GROUP BY actor.login
ORDER BY count desc
LIMIT 5;
```

login	count
Mytherin	19
Mause	4
carlopi	3
Tmonster	2
lnkuiper	2

As expected, Mark Raasveldt (Mytherin, co-founder of DuckDB Labs) was the most active! My activity (lnkuiper, software engineer at DuckDB Labs) also shows up.

Handling Inconsistent JSON Schemas

So far, we have ignored the "payload" of the events. We ignored it because the contents of this field are different based on the type of event. We can see how they differ with the following query:

```
SELECT json_group_structure(payload) AS structure
FROM (
    SELECT *
    FROM read_json(
        'gharchive_gz/*.json.gz',
        columns = {
            id: 'BIGINT',
            type: 'VARCHAR',
            actor: 'STRUCT(id UBIGINT,
                           login VARCHAR,
                           display_login VARCHAR,
                           gravatar_id VARCHAR,
                           url VARCHAR,
                           avatar_url VARCHAR)',
```

```
            repo: 'STRUCT(id UBIGINT, name VARCHAR, url VARCHAR)',
            payload: 'JSON',
            public: 'BOOLEAN',
            created_at: 'TIMESTAMP',
           org: 'STRUCT(id UBIGINT, login VARCHAR, gravatar_id VARCHAR, url VARCHAR, avatar_url VARCHAR)'
        },
        records = true
    )
    WHERE type = 'WatchEvent'
    LIMIT 2048
);
```

structure
{"action":"VARCHAR"}

The `"payload"` field is simple for events of type `WatchEvent`. However, if we change the type to `PullRequestEvent`, we get a JSON structure of more than 500 lines when formatted with a JSON formatter. We don't want to look through all those fields, so we cannot use our automatic schema detection, which will try to get them all. Instead, we can manually supply the structure of the fields we're interested in. DuckDB will skip reading the other fields. Another approach is to store the `"payload"` field as DuckDB's JSON data type and parse it at query time (see the example later in this post!).

I stripped down the JSON structure for the `"payload"` of events with the type `PullRequestEvent` to the things I'm actually interested in:

```
{
   "action":"VARCHAR",
   "number":"UBIGINT",
   "pull_request":{
      "url":"VARCHAR",
      "id":"UBIGINT",
      "title":"VARCHAR",
      "user":{
         "login":"VARCHAR",
         "id":"UBIGINT",
      },
      "body":"VARCHAR",
      "created_at":"TIMESTAMP",
      "updated_at":"TIMESTAMP",
      "assignee":{
         "login":"VARCHAR",
         "id":"UBIGINT",
      },
      "assignees":[
         {
            "login":"VARCHAR",
            "id":"UBIGINT",
         }
      ],
  }
}
```

This is technically not valid JSON because there are trailing commas. However, we try to allow trailing commas wherever possible in DuckDB, including JSON!

We can now plug this into the `columns` parameter of `read_json`, but we need to convert it to a DuckDB type first. I'm lazy, so I prefer to let DuckDB do this for me:

```
SELECT typeof(
    json_transform('{}', '{
        "action":"VARCHAR",
        "number":"UBIGINT",
        "pull_request":{
          "url":"VARCHAR",
          "id":"UBIGINT",
          "title":"VARCHAR",
          "user":{
              "login":"VARCHAR",
              "id":"UBIGINT",
          },
          "body":"VARCHAR",
          "created_at":"TIMESTAMP",
          "updated_at":"TIMESTAMP",
          "assignee":{
              "login":"VARCHAR",
              "id":"UBIGINT",
          },
          "assignees":[
              {
                "login":"VARCHAR",
                "id":"UBIGINT",
              }
          ],
        }
    }')
);
```

This gives us back a DuckDB type that we can plug the type into our function! Note that because we are not auto-detecting the schema, we have to supply `timestampformat` to be able to parse the timestamps correctly. The key `"user"` must be surrounded by quotes because it is a reserved keyword in SQL:

```
CREATE OR REPLACE TABLE pr_events AS
    SELECT *
    FROM read_json(
        'gharchive_gz/*.json.gz',
        columns = {
            id: 'BIGINT',
            type: 'VARCHAR',
            actor: 'STRUCT(id UBIGINT,
                           login VARCHAR,
                           display_login VARCHAR,
                           gravatar_id VARCHAR,
                           url VARCHAR,
                           avatar_url VARCHAR)',
            repo: 'STRUCT(id UBIGINT, name VARCHAR, url VARCHAR)',
            payload: 'STRUCT(
                        action VARCHAR,
                        number UBIGINT,
                        pull_request STRUCT(
                          url VARCHAR,
                          id UBIGINT,
                          title VARCHAR,
                          "user" STRUCT(
                            login VARCHAR,
                            id UBIGINT
                          ),
                          body VARCHAR,
                          created_at TIMESTAMP,
```

```
                        updated_at TIMESTAMP,
                        assignee STRUCT(login VARCHAR, id UBIGINT),
                        assignees STRUCT(login VARCHAR, id UBIGINT)[]
                    )
                )',
            public: 'BOOLEAN',
            created_at: 'TIMESTAMP',
            org: 'STRUCT(id UBIGINT, login VARCHAR, gravatar_id VARCHAR, url VARCHAR, avatar_url VARCHAR)'
        },
        format = 'newline_delimited',
        records = true,
        timestampformat = '%Y-%m-%dT%H:%M:%SZ'
    )
    WHERE type = 'PullRequestEvent';
```

This query completes in around 36 seconds with an on-disk database (resulting size is 478 MB) and 9 seconds with an in-memory database. If you don't care about preserving insertion order, you can speed the query up with this setting:

```
SET preserve_insertion_order = false;
```

With this setting, the query completes in around 27 seconds with an on-disk database and 8.5 seconds with an in-memory database. The difference between the on-disk and in-memory case is quite substantial here because DuckDB has to compress and persist much more data.

Now we can analyze pull request events! Let's see what the maximum number of assignees is:

```
SELECT max(length(payload.pull_request.assignees)) AS max_assignees
FROM pr_events;
```

max_assignees
10

That's a lot of people reviewing a single pull request!

We can check who was assigned the most:

```
WITH assignees AS (
    SELECT payload.pull_request.assignee.login AS assignee
    FROM pr_events
    UNION ALL
    SELECT unnest(payload.pull_request.assignees).login AS assignee
    FROM pr_events
)
SELECT assignee, count(*) AS count
FROM assignees
WHERE assignee NOT NULL
GROUP BY assignee
ORDER BY count DESC
LIMIT 5;
```

assignee	count
poad	494
vinayakkulkarni	268
tmtmtmtm	198
fisker	98

assignee	count
icemac	84

That's a lot of assignments – although I suspect there are duplicates in here.

Storing as JSON to Parse at Query Time

Specifying the JSON schema of the `"payload"` field was helpful because it allowed us to directly analyze what is there, and subsequent queries are much faster. Still, it can also be quite cumbersome if the schema is complex. If you don't want to specify the schema of a field, you can set the type as `'JSON'`:

```
CREATE OR REPLACE TABLE pr_events AS
    SELECT *
    FROM read_json(
        'gharchive_gz/*.json.gz',
        columns = {
            id: 'BIGINT',
            type: 'VARCHAR',
            actor: 'STRUCT(id UBIGINT,
                           login VARCHAR,
                           display_login VARCHAR,
                           gravatar_id VARCHAR,
                           url VARCHAR,
                           avatar_url VARCHAR)',
            repo: 'STRUCT(id UBIGINT, name VARCHAR, url VARCHAR)',
            payload: 'JSON',
            public: 'BOOLEAN',
            created_at: 'TIMESTAMP',
           org: 'STRUCT(id UBIGINT, login VARCHAR, gravatar_id VARCHAR, url VARCHAR, avatar_url VARCHAR)'
        },
        format = 'newline_delimited',
        records = true,
        timestampformat = '%Y-%m-%dT%H:%M:%SZ'
    )
    WHERE type = 'PullRequestEvent';
```

This will load the `"payload"` field as a JSON string, and we can use DuckDB's JSON functions to analyze it when querying. For example:

```
SELECT DISTINCT payload->>'action' AS action, count(*) AS count
FROM pr_events
GROUP BY action
ORDER BY count DESC;
```

The `->>` arrow is short-hand for our `json_extract_string` function. Creating the entire `"payload"` field as a column with type JSON is not the most efficient way to get just the `"action"` field, but this example is just to show the flexibility of `read_json`. The query results in the following table:

action	count
opened	189096
closed	174914
reopened	2080

As we can see, only a few pull requests have been reopened.

Conclusion

DuckDB tries to be an easy-to-use tool that can read all kinds of data formats. In the 0.7.0 release, we have added support for reading JSON. JSON comes in many formats and all kinds of schemas. DuckDB's rich support for nested types (`LIST`, `STRUCT`) allows it to fully "shred" the JSON to a columnar format for more efficient analysis.

We are excited to hear what you think about our new JSON functionality. If you have any questions or suggestions, please reach out to us on Discord or GitHub!

The Return of the H2O.ai Database-like Ops Benchmark

Publication date: 2023-04-14

Author: Tom Ebergen

TL;DR: We've resurrected the H2O.ai database-like ops benchmark with up to date libraries and plan to keep re-running it.

Skip directly to the results

> We published a new blog post on the H2O.ai benchmark in November 2023 and improved the benchmark setup for reproducibility. For details, see the new post: "Updates to the H2O.ai db-benchmark!"

The H2O.ai Database-like Ops Benchmark is a well-known benchmark in the data analytics and R community. The benchmark measures the groupby and join performance of various analytical tools like data.table, Polars, dplyr, ClickHouse, DuckDB and more. Since July 2nd 2021, the benchmark has been dormant, with no result updates or maintenance. Many of the analytical systems measured in the benchmark have since undergone substantial improvements, leaving many of the maintainers curious as to where their analytical tool ranks on the benchmark.

DuckDB has decided to give the H2O.ai benchmark new life and maintain it for the foreseeable future. One reason the DuckDB project has decided to maintain the benchmark is because DuckDB has had 10 new minor releases since the most recent published results on July 2nd, 2021. After managing to run parts of the benchmark on a r3-8xlarge AWS box, DuckDB ranked as a top performer on the benchmark. Additionally, the DuckDB project wants to demonstrate it's commitment to performance by consistently comparing DuckDB with other analytical systems. While DuckDB delivers differentiated ease of use, raw performance and scalability are critically important for solving tough problems fast. Plus, just like many of our fellow data folks, we have a need for speed. Therefore, the decision was made to fork the benchmark, modernize underlying dependencies and run the benchmark on the latest versions of the included systems. You can find the repository on GitHub.

The results of the new benchmark are very interesting, but first a quick summary of the benchmark and what updates took place.

The H2O.ai Database-like Ops Benchmark

There are 5 basic grouping tests and 5 advanced grouping tests. The 10 grouping queries all focus on a combination of the following

- Low cardinality (a few big groups)
- High cardinality (lots of very small groups)
- Grouping integer types
- Grouping string types

Each query is run only twice with both results being reported. This way we can see the performance of a cold run and any effects data caching may have. The idea is to avoid reporting any potential "best" results on a hot system. Data analysts only need to run a query once to get their answer. No one drives to the store a second time to get another litre of milk faster.

The time reported is the sum of the time it takes to run all 5 queries twice.

More information about the specific queries can be found below.

The Data and Queries

The queries have not changed since the benchmark went dormant. The data is generated in a rather simple manner. Inspecting the datagen files you can see that the columns are generated with small, medium, and large groups of char and int values. Similar generation logic applies to the join data generation.

Query	SQL	Objective
groupby #1	`SELECT id1, sum(v1) AS v1 FROM tbl GROUP BY id1`	Sum over large cardinality groups, grouped by varchar
groupby #2	`SELECT id1, id2, sum(v1) AS v1 FROM tbl GROUP BY id1, id2`	Sum over medium cardinality groups, grouped by varchars
groupby #3	`SELECT id3, sum(v1) AS v1, mean(v3) AS v3 FROM tbl GROUP BY id3`	Sum and mean over many small cardinality groups, grouped by varchar
groupby #4	`SELECT id4, mean(v1) AS v1, mean(v2) AS v2, mean(v3) AS v3 FROM tbl GROUP BY id4`	Mean over many large cardinality groups, grouped by integer
groupby #5	`SELECT id6, sum(v1) AS v1, sum(v2) AS v2, sum(v3) AS v3 FROM tbl GROUP BY id6`	Sum over many small groups, grouped by integer
advanced groupby #1	`SELECT id4, id5, quantile_cont(v3, 0.5) AS median_v3, stddev(v3) AS sd_v3 FROM tbl GROUP BY id4, id5`	`quantile_cont` over medium cardinality group, grouped by integers
advanced groupby #2	`SELECT id3, max(v1)-min(v2) AS range_v1_v2 FROM tbl GROUP BY id3`	Range selection over small cardinality groups, grouped by integer
advanced groupby #3	`SELECT id6, v3 AS largest2_v3 FROM (SELECT id6, v3, row_number() OVER (PARTITION BY id6 ORDER BY v3 DESC) AS order_v3 FROM x WHERE v3 IS NOT NULL) sub_query WHERE order_v3 <= 2`	Advanced group by query
advanced groupby #4	`SELECT id2, id4, pow(corr(v1, v2), 2) AS r2 FROM tbl GROUP BY id2, id4`	Arithmetic over medium sized groups, grouped by varchar, integer.
advanced groupby #5	`SELECT id1, id2, id3, id4, id5, id6, sum(v3) AS v3, count(*) AS count FROM tbl GROUP BY id1, id2, id3, id4, id5, id6`	Many small groups, the number of groups is the cardinality of the dataset
join #1	`SELECT x.*, small.id4 AS small_id4, v2 FROM x JOIN small USING (id1)`	Joining a large table (x) with a small-sized table on integer type
join #2	`SELECT x.*, medium.id1 AS medium_id1, medium.id4 AS medium_id4, medium.id5 AS medium_id5, v2 FROM x JOIN medium USING (id2)`	Joining a large table (x) with a medium-sized table on integer type

Query	SQL	Objective
join #3	`SELECT x.*, medium.id1 AS medium_id1, medium.id4 AS medium_id4, medium.id5 AS medium_id5, v2 FROM x LEFT JOIN medium USING (id2)`	Left join a large table (x) with a medium-sized table on integer type
join #4	`SELECT x.*, medium.id1 AS medium_id1, medium.id2 AS medium_id2, medium.id4 AS medium_id4, v2 FROM x JOIN medium USING (id5)`	Join a large table (x) with a medium table on varchar type
join #5	`SELECT x.*, big.id1 AS big_id1, big.id2 AS big_id2, big.id4 AS big_id4, big.id5 AS big_id5, big.id6 AS big_id6, v2 FROM x JOIN big USING (id3)`	Join a large table (x) with a large table on integer type.

You can find more information about the queries in the Efficiency of Data Processing slides.

Modifications to the Benchmark & Hardware

No modifications have been made to the queries or the data generation. Some scripts required minor modifications so that the current version of the library could be run. The hardware used is slightly different as the exact AWS offering the benchmark previously used is no longer available. Base libraries have been updated as well. GPU libraries were not tested.

AWS is a m4.10xlarge

- CPU model: Intel(R) Xeon(R) CPU E5-2676 v3 @ 2.40 GHz
- CPU cores: 40
- RAM model: Unknown
- Memory: 160 GB
- NO GPU specifications
- R upgraded, 4.0.0 -> 4.2.2
- Python upgraded 3.[6|7] -> 3.10

Changes Made to Install Scripts of Other Systems

Pandas, Polars, Dask, and ClickHouse required changes to their setup/install scripts. The changes were relatively minor consisting mostly of syntax updates and data ingestion updates. Data ingestion did not affect the reporting timing results.

Results

You can also look at the results. DuckDB's timings have improved significantly since v0.2.7 (released over two years ago). A major contributor to our increased performance is parallel grouped aggregation, merged in March 2022, and parallel result set materialization. In addition, DuckDB now supports enum types, which makes DuckDB `group by` aggregation even faster. Improvements to the out-of-core hash join were merged as well, further improving the performance of our joins.

Questions about Certain Results?

Some solutions may report internal errors for some queries. Feel free to investigate the errors by using the `repro.sh` scripts and file a GitHub issue to resolve any confusion. In addition, there are many areas in the code where certain query results are automatically nullified. If you believe that is the case for a query for your system or if you have any other questions, you can create a GitHub issue to discuss.

Maintenance Plan

DuckDB will continue to maintain this benchmark for the forseeable future. The process for re-running the benchmarks with updated library versions must still be decided.

Do you have any other questions? Would you like to have your system added to the benchmark? Please feel free to read the README in the repository, and if you still have questions, you can reach out to me at tom@duckdblabs.com or on our Discord!

Introducing DuckDB for Swift

Publication date: 2023-04-21

Author: Tristan Celder

TL;DR: DuckDB now has a native Swift API. DuckDB on mobile here we go!

Today we're excited to announce the DuckDB API for Swift. It enables developers on Swift platforms to harness the full power of DuckDB using a native Swift interface with support for great Swift features such as strong typing and concurrency. The API is available not only on Apple platforms, but on Linux too, opening up new opportunities for the growing Swift on Server ecosystem.

What's Included

DuckDB is designed to be fast, reliable and easy to use, and it's this philosophy that also guided the creation of our new Swift API.

This initial release supports many of the great features of DuckDB right out of the box, including:

- Queries via DuckDB's enhanced SQL dialect: In addition to basic SQL, DuckDB supports arbitrary and nested correlated subqueries, window functions, collations, complex types (Swift arrays and structs), and more.
- Import and export of JSON, CSV, and Parquet files: Beyond its built-in and super-efficient native file format, DuckDB supports reading in, and exporting out to, JSON, CSV, and Parquet files.
- Strongly typed result sets: DuckDB's strongly typed result sets are a natural fit for Swift. It's simple to cast DuckDB columns to their native Swift equivalents, ready for presentation using SwiftUI or as part of an existing TabularData workflow.
- Swift concurrency support: by virtue of their `Sendable` conformance, many of DuckDB's core underlying types can be safely passed across concurrency contexts, easing the process of designing parallel processing workflows and ensuring responsive UIs.

Usage

To demonstrate just how well DuckDB works together with Swift, we've created an example project that uses raw data from NASA's Exoplanet Archive loaded directly into DuckDB.

You'll see how to:

- Instantiate a DuckDB in-memory Database and Connection
- Populate a DuckDB table with the contents of a remote CSV
- Query a DuckDB database and prepare the results for presentation

Finally, we'll present our analysis with the help of Apple's TabularData Framework and Swift Charts.

Instantiating DuckDB

DuckDB supports both file-based and in-memory databases. In this example, as we don't intend to persist the results of our Exoplanet analysis to disk, we'll opt for an in-memory Database.

```
let database = try Database(store: .inMemory)
```

However, we can't issue queries just yet. Much like other RDMSs, queries must be issued through a *database connection*. DuckDB supports multiple connections per database. This can be useful to support parallel processing, for example. In our project, we'll need just the one connection that we'll eventually access asynchronously.

```
let connection = try database.connect()
```

Finally, we'll create an app-specific type that we'll use to house our database and connection and through which we'll eventually define our app-specific queries.

```
import DuckDB

final class ExoplanetStore {

    let database: Database
    let connection: Connection

    init(database: Database, connection: Connection) {
        self.database = database
        self.connection = connection
    }
}
```

Populating DuckDB with a Remote CSV File

One problem with our current `ExoplanetStore` type is that it doesn't yet contain any data to query. To fix that, we'll load it with the data of every Exoplanet discovered to date from NASA's Exoplanet Archive.

There are hundreds of configuration options for this incredible resource, but today we want each exoplanet's name and its discovery year packaged as a CSV. Checking the docs gives us the following endpoint:

```
https://exoplanetarchive.ipac.caltech.edu/TAP/sync?query=select+pl_name+,+disc_
year+from+pscomppars&format=csv
```

Once we have our CSV downloaded locally, we can use the following SQL command to load it as a new table within our DuckDB in-memory database. DuckDB's `read_csv_auto` command automatically infers our table schema and the data is immediately available for analysis.

```
CREATE TABLE exoplanets AS
    SELECT * FROM read_csv_auto('downloaded_exoplanets.csv');
```

Let's package this up as a new asynchronous factory method on our `ExoplanetStore` type:

```
import DuckDB
import Foundation

final class ExoplanetStore {

    // Factory method to create and prepare a new ExoplanetStore
    static func create() async throws -> ExoplanetStore {

        // Create our database and connection as described above
        let database = try Database(store: .inMemory)
        let connection = try database.connect()

        // Download the CSV from the exoplanet archive
        let (csvFileURL, _) = try await URLSession.shared.download(
            from: URL(string: "https://exoplanetarchive.ipac.caltech.edu/TAP/sync?query=select+pl_
name+,+disc_year+from+pscomppars&format=csv")!)

        // Issue our first query to DuckDB
        try connection.execute("""
            CREATE TABLE exoplanets AS (
                SELECT * FROM read_csv_auto('\(csvFileURL.path)')
            );
```

```
            """)

        // Create our pre-populated ExoplanetStore instance
        return ExoplanetStore(
            database: database,
            connection: connection
        )
    }

    // Let's make the initializer we defined previously
    // private. This prevents anyone accidentally instantiating
    // the store without having pre-loaded our Exoplanet CSV
    // into the database
    private init(database: Database, connection: Connection) {
        // ...
    }
}
```

Querying the Database

Now that the database is populated with data, it's ready to be analyzed. Let's create a query which we can use to plot a chart of the number of exoplanets discovered by year.

```
SELECT disc_year, count(disc_year) AS Count
FROM exoplanets
GROUP BY disc_year
ORDER BY disc_year;
```

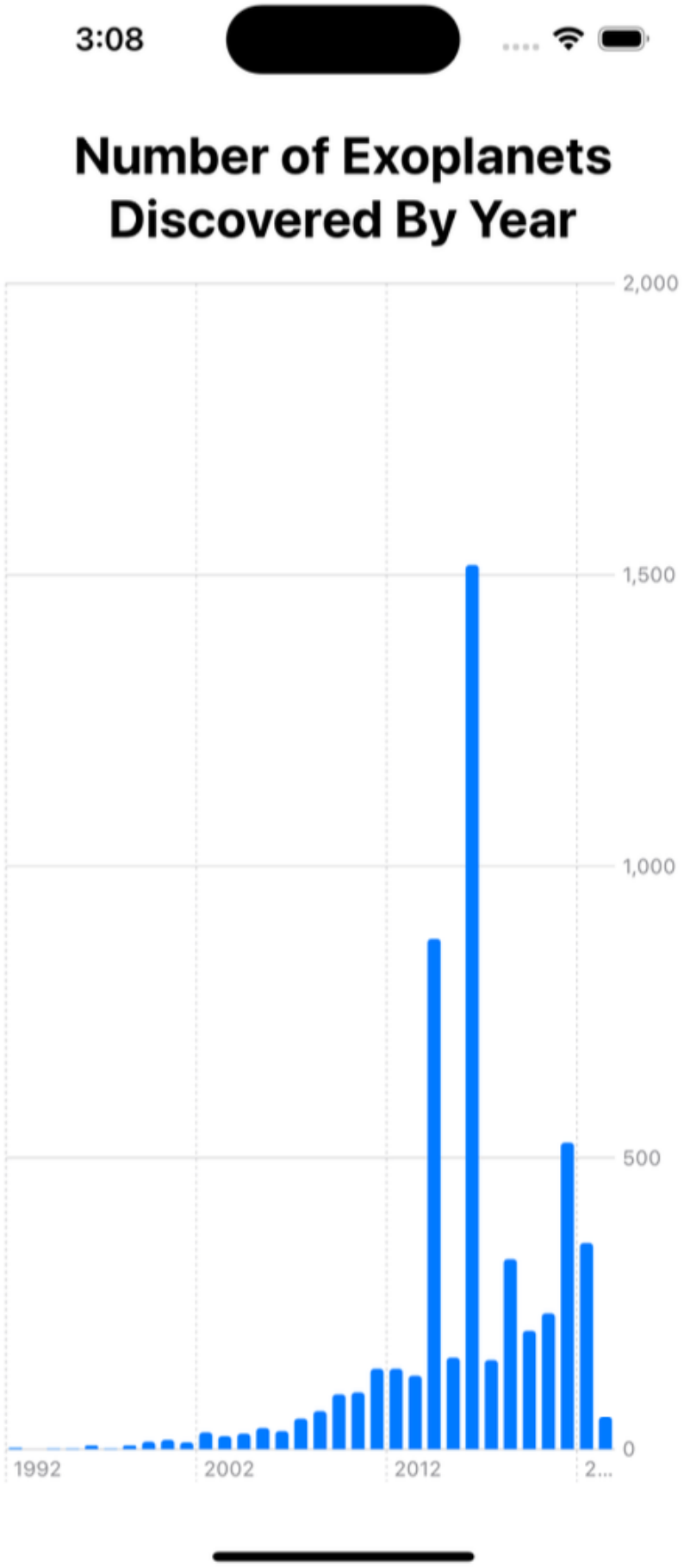
Issuing the query to DuckDB from within Swift is simple. We'll again make use of an async function from which to issue our query. This means the callee won't be blocked while the query is executing. We'll then cast the result columns to Swift native types using DuckDB's `ResultSet cast(to:)` family of methods, before finally wrapping them up in a `DataFrame` from the TabularData framework ready for presentation in the UI.

```
...

import TabularData

extension ExoplanetStore {

    // Retrieves the number of exoplanets discovered by year
    func groupedByDiscoveryYear() async throws -> DataFrame {

        // Issue the query we described above
        let result = try connection.query("""
            SELECT disc_year, count(disc_year) AS Count
            FROM exoplanets
            GROUP BY disc_year
            ORDER BY disc_year
            """)

        // Cast our DuckDB columns to their native Swift
        // equivalent types
        let discoveryYearColumn = result[0].cast(to: Int.self)
        let countColumn = result[1].cast(to: Int.self)

        // Use our DuckDB columns to instantiate TabularData
        // columns and populate a TabularData DataFrame
        return DataFrame(columns: [
            TabularData.Column(discoveryYearColumn)
```

```
                .eraseToAnyColumn(),
            TabularData.Column(countColumn)
                .eraseToAnyColumn(),
        ])
    }
}
```

Visualizing the Results

In just a few lines of code, our database has been created, populated and analyzed – all that's left to do now is present the results.


3:08
Number of Exoplanets Discovered By Year
2,000
1,500
1,000
500
0
1992
2002
2012
2...

And I have a feeling that we're just getting started…

For the complete example project – including the SwiftUI views and Chart definitions used to create the screenshot above – clone the DuckDB Swift repo and open up the runnable app project located in `Examples/SwiftUI/ExoplanetExplorer.xcodeproj`.

We encourage you to modify the code, explore the Exoplanet Archive and DuckDB, and make some discoveries of your own – interplanetary or otherwise!

Conclusion

In this article we've introduced the brand new Swift API for DuckDB and demonstrated how quickly you can get up and running analyzing data.

With DuckDB's incredible performance and analysis capabilities and Swift's vibrant eco-system and platform support, there's never been a better time to begin exploring analytical datasets in Swift.

We can't wait to see what you do with it. Feel free to reach out on our Discord if you have any questions!

The Swift API for DuckDB is packaged using Swift Package Manager and lives in a new top-level repository available at https://github.com/duckdb/duckdb-swift.

PostGEESE? Introducing The DuckDB Spatial Extension

Publication date: 2023-04-28

Author: Max Gabrielsson

TL;DR: DuckDB now has an official Spatial extension to enable geospatial processing.

Geospatial data has become increasingly important and prevalent in modern-day applications and data engineering workflows, with use-cases ranging from location-based services to environmental monitoring.

While there are many great and specialized tools for working with geospatial data, integrating geospatial capabilities directly into DuckDB has multiple advantages. For one, you get to operate, transform and join your geospatial data alongside your regular, unstructured or time-series data using DuckDBs rich type system and extensions like `JSON` and `ICU`. Secondly, spatial queries involving geometric predicates and relations translate surprisingly well to SQL, which is all about expressing relations after all! Not to mention all the other benefits provided by DuckDB such as transactional semantics, high performance multi-threaded vectorized execution and larger-than-memory data processing.

Therefore, we're very excited to announce that DuckDB now has a Spatial extension packed with features easily installable from the DuckDB CLI and other DuckDB clients. Simply execute:

```
INSTALL spatial;
LOAD spatial;
```

And you're good to go!

No, we're not calling it GeoDuck either, that's just gross.

What's in It?

The core of the extension is a `GEOMETRY` type based on the ”Simple Features” geometry model and accompanying functions such as `ST_Area`, `ST_Intersects`. It also provides methods for reading and writing geospatial data formats and converting between coordinate reference systems (details later in the post!). While we're not ready to commit to full compliance with the OGC Simple Feature Access and SQL/MM Standards yet, if you've worked with geospatial functionality in other database systems such as PostGIS or SpatiaLite, you should feel right at home.

Most of the implemented functions are based on the trifecta of foundational geospatial libraries, GEOS, GDAL and PROJ, which provide algorithms, format conversions and coordinate reference system transformations respectively. In particular, we leverage GDAL to provide a set of table and copy functions that enable import and export of tables from and to 50+ different geospatial data formats (so far!), including the most common ones such as Shapefiles, GeoJSON, GeoPackage, KML, GML, WKT, WKB, etc.

Check for yourself by running:

Initially we have prioritized providing a breadth of capabilities by wrapping existing libraries. We're planning to implement more of the core functions and algorithms natively in the future to enable faster performance and more efficient memory management.

As an initial step in this direction, we provide a set of non-standard specialized columnar DuckDB native geometry types such as `POINT_2D`, `BOX_2D`, etc. that should provide better compression and faster execution in exchange for some flexibility, but work around these are still very much experimental.

Example Usage

The following demonstrates how you can use the spatial extension to read and export multiple geospatial data formats, transform geometries between different coordinate reference systems and work with spatial property and predicate functions. While this example may be slightly contrived, we want to showcase the power of the currently available features. You can find the datasets used in this example in the spatial extension repository.

Let's import the NYC taxi ride data provided in Parquet format as well as the accompanying taxi zone data from a shapefile, using the `ST_Read` table function provided by the spatial extension. These taxi zones break NYC into polygons that represent regions, for example the Newark Airport. We then create a table for the rides and a table for the zones. Note that `ST_Read` produces a table with a `wkb_geometry` column that contains the geometry data encoded as a WKB (Well-Known Binary) blob, which we then convert to the `GEOMETRY` type using the `ST_GeomFromWKB` function.

> This may all seem a bit much if you are not familiar with the geospatial ecosystem, but rest assured this is all you really need to get started. In short: – Shapefile (.shp, .shx, .dbf) is a common format for storing geometry vector data and auxiliary metadata such as indexes and attributes. – WKB (Well Known Binary), while not really a file format in itself, is a common binary encoding of vector geometry data, used in e.g., GeoParquet. Comes in multiple flavors, but we're only concerned with "standard" WKB for now. – GEOMETRY is a DuckDB type that represents a Simple Features geometry object, which is based on a set of standards modeling vector geometry data as points, linestrings, polygons or collections of such. This is the core data type used by the spatial extension, and what most of the provided functions take and return.

```
INSTALL spatial;
LOAD spatial;

CREATE TABLE rides AS
    SELECT *
    FROM 'yellow_tripdata_2010-01-limit1mil.parquet';

-- Load the NYC taxi zone data from a shapefile using the gdal-based ST_Read function
CREATE TABLE zones AS
    SELECT zone, LocationId, borough, geom
    FROM ST_Read('taxi_zones/taxi_zones.shx');
```

Let's compare the trip distance to the linear distance between the pickup and dropoff points to figure out how efficient the taxi drivers are (or how dirty the data is, since some diffs seem to be negative). We transform the coordinates from "WGS84" (given by the identifier EPSG:4326), also commonly known as simply latitude/longitude to the "NAD83 / New York Long Island ftUS" (identified as ESRI:102718) coordinate reference system which is a projection with minimal distortion around New York. We then calculate the distance using the `ST_Distance` function. In This case we get the distance in feet since we've converted the coordinates to NAD83 but we can easily convert it into to miles (5280 ft/mile) which is the unit used in the rides dataset so we can compare them correctly.

Trips with a distance shorter than the aerial distance are likely to be erroneous, so we use this query to filter out some bad data. The query below takes advantage of DuckDB's ability to refer to column aliases defined within the same select statement. This is a small example of how DuckDB's rich SQL dialect can simplify geospatial analysis.

```
CREATE TABLE cleaned_rides AS
    SELECT
        ST_Point(pickup_latitude, pickup_longitude) AS pickup_point,
        ST_Point(dropoff_latitude, dropoff_longitude) AS dropoff_point,
        dropoff_datetime::TIMESTAMP - pickup_datetime::TIMESTAMP AS time,
        trip_distance,
        ST_Distance(
            ST_Transform(pickup_point, 'EPSG:4326', 'ESRI:102718'),
            ST_Transform(dropoff_point, 'EPSG:4326', 'ESRI:102718')) / 5280
            AS aerial_distance,
        trip_distance - aerial_distance AS diff
    FROM rides
    WHERE diff > 0
    ORDER BY diff DESC;
```

> It should be noted that this is not entirely accurate since the `ST_Distance` function we use does not take into account the curvature of the earth. However, we'll accept it as a good enough approximation for our purposes. Spherical and geodesic distance calculations are on the roadmap!

Now let's join the taxi rides with the taxi zones to get the start and end zone for each ride. We use the `ST_Within` function as our join condition to check if a pickup or dropoff point is within a taxi zone polygon. Again we need to transform the coordinates from WGS84 to the NAD83 since the taxi zone data also use that projection. Spatial joins like these are the bread and butter of geospatial data processing, but we don't currently have any optimizations in place (such as spatial indexes) to speed up these queries, which is why we only use a subset of the data for the following step.

```
-- Since we don't have spatial indexes yet, use a smaller dataset for the join.
DELETE FROM cleaned_rides WHERE rowid > 5000;

CREATE TABLE joined AS
    SELECT
        pickup_point,
        dropoff_point,
        start_zone.zone AS start_zone,
        end_zone.zone AS end_zone,
        trip_distance,
        time,
    FROM cleaned_rides
    JOIN zones AS start_zone
      ON ST_Within(ST_Transform(pickup_point, 'EPSG:4326', 'ESRI:102718'), start_zone.geom)
    JOIN zones AS end_zone
      ON ST_Within(ST_Transform(dropoff_point, 'EPSG:4326', 'ESRI:102718'), end_zone.geom);
```

We can export the joined table to a `GeoJSONSeq` file using the GDAL copy function, passing in a GDAL layer creation option. Since GeoJSON only supports a single `GEOMETRY` per record, we use the `ST_MakeLine` function to combine the pickup and dropoff points into a single line geometry. The default coordinate reference system for `GeoJSON` is WGS84, but the coordinate pairs are expected to be in longitude/latitude, so we need to flip the geometry using the `ST_FlipCoordinates` function.

```
COPY (
    SELECT
        ST_MakeLine(pickup_point, dropoff_point)
            .ST_FlipCoordinates()
            .ST_AsWKB()
            AS wkb_geometry,
        start_zone,
        end_zone,
        time::VARCHAR AS trip_time
    FROM joined)
TO 'joined.geojsonseq'
WITH (
    FORMAT gdal,
    DRIVER 'GeoJSONSeq',
    LAYER_CREATION_OPTIONS 'WRITE_BBOX=YES'
);
```

And there we have it! We pulled tabular data from Parquet, combined it with geospatial data in a shapefile, cleaned and analyzed that combined data, and output it to a human readable geospatial format. The full set of currently supported functions and their implementation status can be found over at the docs in this table.

What's Next?

While it's probably going to take a while for us to catch up to the full set of functions provided by e.g., PostGIS, we believe that DuckDB's vectorized execution model and columnar storage format will enable a whole new class of optimizations for geospatial processing that we've just begun exploring. Improving the performance of spatial joins and predicates is therefore high on our list of priorities.

There are also some limitations with our GEOMETRY type that we would eventually like to tackle, such as the fact that we don't support additional Z and M dimensions, or don't support the full range of geometry sub-types that are mandated by the OGC standard, like curves or polyhedral surfaces.

We're also interested in supporting spherical and ellipsoidal calculations in the near future, perhaps in the form of a dedicated GEOGRAPHY type.

Wasm builds are also just around the corner!

Please take a look at the GitHub repository for the full roadmap and to see what we're currently working on. If you would like to help build this capability, please reach out on GitHub!

Conclusion

The DuckDB Spatial extension is another step towards making DuckDB a swiss army knife for data engineering and analytics. This extension provides a flexible and familiar GEOMETRY type, reprojectable between thousands of coordinate reference systems, coupled with the capability to export and import geospatial data between more than 50 different data sources. All embedded into a single extension with minimal runtime dependencies. This enables DuckDB to fit seamlessly into your existing GIS workflows regardless of which geospatial data formats or projections you're working with.

We are excited to hear what you make of the DuckDB spatial extension. It's still early days but we hope to have a lot more to share in the future as we continue making progress! If you have any questions, suggestions, ideas or issues, please don't hesitate to reach out to us on Discord or GitHub!

10 000 Stars on GitHub

Publication date: 2023-05-12

Authors: Mark Raasveldt and Hannes Mühleisen

Today, DuckDB reached 10 000 stars on GitHub. We would like to pause for a second to express our gratitude to everyone who contributed to DuckDB and of course all its users. When we started working on DuckDB back in 2018, we would have never dreamt of getting this kind of adoption in such a short time.

From those brave souls who were early adopters of DuckDB back in 2019 to the many today, we are happy you're part of our community. Thank you for your feedback, feature requests and for your enthusiasm in adopting new features and integrations. Thank you for helping each other on our Discord server or in GitHub Discussions. Thank you for spreading the word, too.

We also would like to extend special thanks to the DuckDB foundation supporters, who through their generous donations keep DuckDB independent.

For us, the maintainers of DuckDB, the past few years have also been quite eventful: We spun off from the research group where DuckDB originated to a successful company with close to 20 employees and many excellent partnerships.

We are very much looking forward to what the future will hold for DuckDB. Things are looking bright!

Announcing DuckDB 0.8.0

Publication date: 2023-05-17

Authors: Mark Raasveldt and Hannes Mühleisen

The DuckDB team is happy to announce the latest DuckDB release (0.8.0). This release is named "Fulvigula" after the Mottled Duck (Anas Fulvigula) native to the Gulf of Mexico.

To install the new version, please visit the installation guide. The full release notes can be found on GitHub.

What's New in 0.8.0

There have been too many changes to discuss them each in detail, but we would like to highlight several particularly exciting features!

- New pivot and unpivot statements
- Improvements to parallel data import/export
- Time series joins
- Recursive globbing
- Lazy-loading of storage metadata for faster startup times
- User-defined functions for Python
- Arrow Database Connectivity (ADBC) support
- New Swift integration

Below is a summary of those new features with examples, starting with two breaking changes in our SQL dialect that are designed to produce more intuitive results by default.

Breaking SQL Changes

This release includes two breaking changes to the SQL dialect: The division operator uses floating point division by default, and the default null sort order is changed from `NULLS FIRST` to `NULLS LAST`. While DuckDB is stil in Beta, we recognize that many DuckDB queries are

already used in production. So, the old behavior can be restored using the following settings:

```
SET integer_division = true;
SET default_null_order = 'NULLS_FIRST';
```

Division Operator. The division operator / will now always perform a floating point division even with integer parameters. The new operator // retains the old semantics and can be used to perform integer division. This makes DuckDB's division operator less error prone for beginners, and consistent with the division operator in Python 3 and other systems in the OLAP space like Spark, Snowflake and BigQuery.

```
SELECT 42 / 5, 42 // 5;
```

(42 / 5)	(42 // 5)
8.4	8

Default Null Sort Order. The default null sort order is changed from `NULLS FIRST` to `NULLS LAST`. The reason for this change is that `NULLS LAST` sort-order is more intuitive when combined with `LIMIT`. With `NULLS FIRST`, Top-N queries always return the `NULL` values first. With `NULLS LAST`, the actual Top-N values are returned instead.

```
CREATE TABLE bigdata (col INTEGER);
INSERT INTO bigdata VALUES (NULL), (42), (NULL), (43);
FROM bigdata ORDER BY col DESC LIMIT 3;
```

v0.7.1	v0.8.0
NULL	43
NULL	42
43	NULL

New SQL Features

Pivot and Unpivot. There are many shapes and sizes of data, and we do not always have control over the process in which data is generated. While SQL is well-suited for reshaping datasets, turning columns into rows or rows into columns is tedious in vanilla SQL. With this release, DuckDB introduces the `PIVOT` and `UNPIVOT` statements that allow reshaping data sets so that rows are turned into columns or vice versa. A key advantage of DuckDB's syntax is that the column names to pivot or unpivot can be automatically deduced. Here is a short example:

```
CREATE TABLE sales (year INTEGER, amount INTEGER);
INSERT INTO sales VALUES (2021, 42), (2022, 100), (2021, 42);
PIVOT sales ON year USING sum(amount);
```

2021	2022
84	100

The documentation contains more examples.

ASOF Joins for Time Series. When joining time series data with background fact tables, the timestamps often do not exactly match. In this case it is often desirable to join rows so that the timestamp is joined with the *nearest timestamp*. The ASOF join can be used for this purpose – it performs a fuzzy join to find the closest join partner for each row instead of requiring an exact match.

```
CREATE TABLE a (ts TIMESTAMP);
CREATE TABLE b (ts TIMESTAMP);
INSERT INTO a VALUES (TIMESTAMP '2023-05-15 10:31:00'), (TIMESTAMP '2023-05-15 11:31:00');
INSERT INTO b VALUES (TIMESTAMP '2023-05-15 10:30:00'), (TIMESTAMP '2023-05-15 11:30:00');

FROM a ASOF JOIN b ON a.ts >= b.ts;
```

a.ts	b.ts
2023-05-15 10:31:00	2023-05-15 10:30:00
2023-05-15 11:31:00	2023-05-15 11:30:00

Please refer to the documentation for a more in-depth explanation.

Data Integration Improvements

Default Parallel CSV Reader. In this release, the parallel CSV reader has been vastly improved and is now the default CSV reader. We would like to thank everyone that has tried out the experimental reader for their valuable feedback and reports. The `experimental_parallel_csv` flag has been deprecated and is no longer required. The parallel CSV reader enables much more efficient reading of large CSV files.

```
CREATE TABLE lineitem AS FROM lineitem.csv;
```

v0.7.1	v0.8.0
4.1 s	1.2 s

Parallel Parquet, CSV and JSON Writing. This release includes support for parallel *order-preserving* writing of Parquet, CSV and JSON files. As a result, writing to these file formats is parallel by default, also without disabling insertion order preservation, and writing to these formats is greatly sped up.

```
COPY lineitem TO 'lineitem.csv';
COPY lineitem TO 'lineitem.parquet';
COPY lineitem TO 'lineitem.json';
```

Format	v0.7.1	v0.8.0
CSV	3.9 s	0.6 s
Parquet	8.1 s	1.2 s
JSON	4.4 s	1.1 s

Recursive File Globbing using `**`. This release adds support for recursive globbing where an arbitrary number of subdirectories can be matched using the `**` operator (double-star).

```
FROM 'data/glob/crawl/stackoverflow/**/*.csv';
```

The documentation has been updated with various examples of this syntax.

Storage Improvements

Lazy-Loading Table Metadata. DuckDB's internal storage format stores metadata for every row group in a table, such as min-max indexes and where in the file every row group is stored. In the past, DuckDB would load this metadata immediately once the database was opened. However, once the data gets very big, the metadata can also get quite large, leading to a noticeable delay on database startup. In this release, we have optimized the metadata handling of DuckDB to only read table metadata as its being accessed. As a result, startup is near-instantaneous even for large databases, and metadata is only loaded for columns that are actually used in queries. The benchmarks below are for a database file containing a single large TPC-H `lineitem` table (120× SF1) with ~770 million rows and 16 columns:

Query	v0.6.1	v0.7.1	v0.8.0	Parquet
`SELECT 42`	1.60 s	0.31 s	0.02 s	-
`FROM lineitem LIMIT 1`	1.62 s	0.32 s	0.03 s	0.27 s

Clients

User-Defined Scalar Functions for Python. Arbitrary Python functions can now be registered as scalar functions within SQL queries. This will only work when using DuckDB from Python, because it uses the actual Python runtime that DuckDB is running within. While plain Python values can be passed to the function, there is also a vectorized variant that uses PyArrow under the hood for higher efficiency and better parallelism.

```
import duckdb

from duckdb.typing import *
from faker import Faker

def random_date():
     fake = Faker()
     return fake.date_between()

duckdb.create_function('random_date', random_date, [], DATE)
res = duckdb.sql('SELECT random_date()').fetchall()
print(res)
# [(datetime.date(2019, 5, 15),)]
```

See the documentation for more information.

Arrow Database Connectivity Support (ADBC). ADBC is a database API standard for database access libraries that uses Apache Arrow to transfer query result sets and to ingest data. Using Arrow for this is particularly beneficial for columnar data management systems which traditionally suffered a performance hit by emulating row-based APIs such as JDBC/ODBC. From this release, DuckDB natively supports ADBC. We're happy to be one of the first systems to offer native support, and DuckDB's in-process design fits nicely with ADBC.

Swift Integration. DuckDB has gained another official language integration: Swift. Swift is a language developed by Apple that most notably is used to create Apps for Apple devices, but also increasingly used for server-side development. The DuckDB Swift API allows developers on all swift platforms to harness DuckDB using a native Swift interface with support for Swift features like strong typing and concurrency.

Final Thoughts

The full release notes can be found on GitHub. We would like to thank all of the contributors for their hard work on improving DuckDB.

Correlated Subqueries in SQL

Publication date: 2023-05-26

Author: Mark Raasveldt

Subqueries in SQL are a powerful abstraction that allow simple queries to be used as composable building blocks. They allow you to break down complex problems into smaller parts, and subsequently make it easier to write, understand and maintain large and complex queries.

DuckDB uses a state-of-the-art subquery decorrelation optimizer that allows subqueries to be executed very efficiently. As a result, users can freely use subqueries to create expressive queries without having to worry about manually rewriting subqueries into joins. For more information, skip to the Performance section.

Types of Subqueries

SQL subqueries exist in two main forms: subqueries as *expressions* and subqueries as *tables*. Subqueries that are used as expressions can be used in the `SELECT` or `WHERE` clauses. Subqueries that are used as tables can be used in the `FROM` clause. In this blog post we will focus on subqueries used as *expressions*. A future blog post will discuss subqueries as *tables*.

Subqueries as expressions exist in three forms.

- Scalar subqueries
- `EXISTS`
- `IN`/`ANY`/`ALL`

All of the subqueries can be either *correlated* or *uncorrelated*. An uncorrelated subquery is a query that is independent from the outer query. A correlated subquery is a subquery that contains expressions from the outer query. Correlated subqueries can be seen as *parameterized subqueries*.

Uncorrelated Scalar Subqueries

Uncorrelated scalar subqueries can only return *a single value*. That constant value is then substituted and used in the query. As an example of why this is useful – imagine that we want to select all of the shortest flights in our dataset. We could run the following query to obtain the shortest flight distance:

```
SELECT min(distance)
FROM ontime;
```

min(distance)
31.0

We could manually take this distance and use it in the `WHERE` clause to obtain all flights on this route.

```
SELECT uniquecarrier, origincityname, destcityname, flightdate
FROM ontime
WHERE distance = 31.0;
```

uniquecarrier	origincityname	destcityname	flightdate
AS	Petersburg, AK	Wrangell, AK	2017-01-15
AS	Wrangell, AK	Petersburg, AK	2017-01-15
AS	Petersburg, AK	Wrangell, AK	2017-01-16

However – this requires us to hardcode the constant inside the query. By using the first query as a *subquery* we can compute the minimum distance as part of the query.

```
SELECT uniquecarrier, origincityname, destcityname, flightdate
FROM ontime
WHERE distance = (
     SELECT min(distance)
     FROM ontime
);
```

Correlated Scalar Subqueries

While uncorrelated subqueries are powerful, they come with a hard restriction: only a *single value* can be returned. Often, what we want to do is *parameterize* the query, so that we can return different values per row.

For example, suppose that we want to find all of the shortest flights *for each carrier*. We can find the shortest flight for a *specific carrier* using the following parameterized query:

```
PREPARE min_distance_per_carrier AS
SELECT min(distance)
FROM ontime
WHERE uniquecarrier = ?;
```

We can execute this prepared statement to obtain the minimum distance for a specific carrier.

```
EXECUTE min_distance_per_carrier('UA');
```

min(distance)
67.0

If we want to use this parameterized query as a subquery, we need to use a *correlated subquery*. Correlated subqueries allow us to use parameterized queries as scalar subqueries by referencing columns from *the outer query*. We can obtain the set of shortest flights per carrier using the following query:

```
SELECT uniquecarrier, origincityname, destcityname, flightdate, distance
FROM ontime AS ontime_outer
WHERE distance = (
     SELECT min(distance)
     FROM ontime
     WHERE uniquecarrier = ontime_outer.uniquecarrier
);
```

uniquecarrier	origincityname	destcityname	flightdate	distance
AS	Wrangell, AK	Petersburg, AK	2017-01-01	31.0
NK	Fort Lauderdale, FL	Orlando, FL	2017-01-01	177.0

uniquecarrier	origincityname	destcityname	flightdate	distance
VX	Las Vegas, NV	Los Angeles, CA	2017-01-01	236.0

Notice how the column from the *outer* relation (`ontime_outer`) is used *inside* the query. This is what turns the subquery into a *correlated subquery*. The column from the outer relation (`ontime_outer.uniquecarrier`) is a *parameter* for the subquery. Logically the subquery is executed once for every row that is present in `ontime`, where the value for the column at that row is substituted as a parameter.

In order to make it more clear that the correlated subquery is in essence a *parameterized query*, we can create a scalar macro that contains the query using DuckDB's macros.

```
CREATE MACRO min_distance_per_carrier(param) AS (
     SELECT min(distance)
     FROM ontime
     WHERE uniquecarrier = param
);
```

We can then use the macro in our original query as if it is a function.

```
SELECT uniquecarrier, origincityname, destcityname, flightdate, distance
FROM ontime AS ontime_outer
WHERE distance = min_distance_per_carrier(ontime_outer.uniquecarrier);
```

This gives us the same result as placing the correlated subquery inside of the query, but is cleaner as we can decompose the query into multiple segments more effectively.

EXISTS

`EXISTS` can be used to check if a given subquery has any results. This is powerful when used as a correlated subquery. For example, we can use `EXISTS` if we want to obtain the *last flight that has been flown on each route*.

We can obtain a list of all flights on a given route past a certain date using the following query:

```
PREPARE flights_after_date AS
SELECT uniquecarrier, origincityname, destcityname, flightdate, distance
FROM ontime
WHERE origin = ? AND dest = ? AND flightdate > ?;

EXECUTE flights_after_date('LAX', 'JFK', DATE '2017-05-01');
```

uniquecarrier	origincityname	destcityname	flightdate	distance
AA	Los Angeles, CA	New York, NY	2017-08-01	2475.0
AA	Los Angeles, CA	New York, NY	2017-08-02	2475.0
AA	Los Angeles, CA	New York, NY	2017-08-03	2475.0

Now in order to obtain the *last flight on a route*, we need to find flights *for which no later flight exists*.

```
SELECT uniquecarrier, origincityname, destcityname, flightdate, distance
FROM ontime AS ontime_outer
WHERE NOT EXISTS (
     SELECT uniquecarrier, origincityname, destcityname, flightdate, distance
     FROM ontime
     WHERE origin = ontime_outer.origin
       AND dest = ontime_outer.dest
       AND flightdate > ontime_outer.flightdate
);
```

uniquecarrier	origincityname	destcityname	flightdate	distance
AA	Daytona Beach, FL	Charlotte, NC	2017-02-27	416.0
EV	Abilene, TX	Dallas/Fort Worth, TX	2017-02-15	158.0
EV	Dallas/Fort Worth, TX	Durango, CO	2017-02-13	674.0

IN / ANY / ALL

`IN` can be used to check if a *given value* exists within the result returned by the subquery. For example, we can obtain a list of all carriers that have performed more than `250 000` flights in the dataset using the following query:

```
SELECT uniquecarrier
FROM ontime
GROUP BY uniquecarrier
HAVING count(*) > 250000;
```

We can then use an `IN` clause to obtain all flights performed by those carriers.

```
SELECT *
FROM ontime
WHERE uniquecarrier IN (
     SELECT uniquecarrier
     FROM ontime
     GROUP BY uniquecarrier
     HAVING count(*) > 250000
);
```

A correlated subquery can be useful here if we want to not count the total amount of flights performed by each carrier, but count the total amount of flights *for the given route*. We can select all flights performed by carriers that have performed *at least 1000 flights on a given route* using the following query.

```
SELECT *
FROM ontime AS ontime_outer
WHERE uniquecarrier IN (
     SELECT uniquecarrier
     FROM ontime
     WHERE ontime.origin = ontime_outer.origin
       AND ontime.dest = ontime_outer.dest
     GROUP BY uniquecarrier
     HAVING count(*) > 1000
);
```

`ANY` and `ALL` are generalizations of `IN`. `IN` checks if the value is present in the set returned by the subquery. This is equivalent to `= ANY(...)`. The `ANY` and `ALL` operators can be used to perform other comparison operators (such as `>`, `<`, `<>`). The above query can be rewritten to `ANY` in the following form.

```
SELECT *
FROM ontime AS ontime_outer
WHERE uniquecarrier = ANY (
     SELECT uniquecarrier
     FROM ontime
     WHERE ontime.origin = ontime_outer.origin
       AND ontime.dest = ontime_outer.dest
     GROUP BY uniquecarrier
     HAVING count(*) > 1000
);
```

Performance

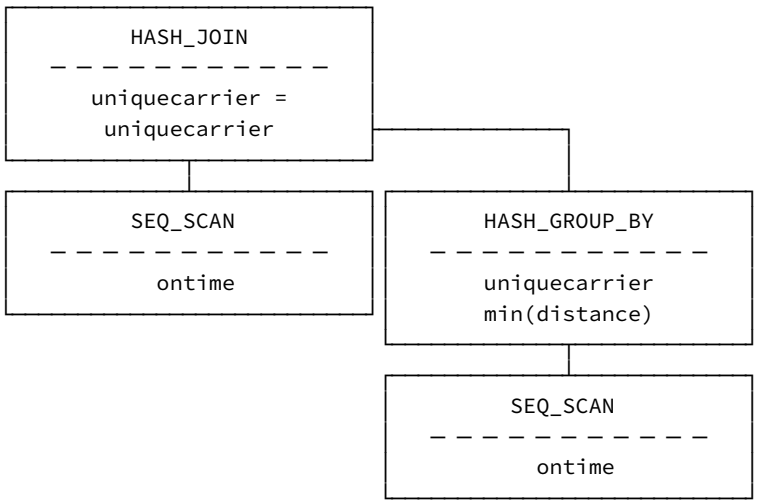
Whereas scalar subqueries are logically executed *once*, correlated subqueries are logically executed *once per row*. As such, it is natural to think that correlated subqueries are very expensive and should be avoided for performance reasons.

While that is true in many SQL systems, it is not the case in DuckDB. In DuckDB, subqueries are **always** *decorrelated*. DuckDB uses a state-of-the-art subquery decorrelation algorithm as described in the Unnesting Arbitrary Queries paper. This allows all subqueries to be decorrelated and executed as a single, much more efficient, query.

In DuckDB, correlation does not imply performance degradation.

If we look at the query plan for the correlated scalar subquery using `EXPLAIN`, we can see that the query has been transformed into a hash aggregate followed by a hash join. This allows the query to be executed very efficiently.

```
EXPLAIN SELECT uniquecarrier, origincityname, destcityname, flightdate, distance
FROM ontime AS ontime_outer
WHERE distance = (
     SELECT min(distance)
     FROM ontime
     WHERE uniquecarrier = ontime_outer.uniquecarrier
);
```



We can see the drastic performance difference that subquery decorrelation has when we compare the run-time of this query in DuckDB with the run-time in Postgres and SQLite. When running the above query on the `ontime` dataset for `2017` with roughly `~4 million` rows, we get the following performance results:

DuckDB	Postgres	SQLite
0.06 s	>48 hours	>48 hours

As Postgres and SQLite do not de-correlate the subquery, the query is not just *logically*, but *actually* executed once for every row. As a result, the subquery is executed *4 million times* in those systems, which takes an immense amount of time.

In this case, it is possible to manually decorrelate the query and generate the following SQL:

```
SELECT ontime.uniquecarrier, origincityname, destcityname, flightdate, distance
FROM ontime
JOIN (
     SELECT uniquecarrier, min(distance) AS min_distance
     FROM ontime
     GROUP BY uniquecarrier
  ) AS subquery
```

```
  ON ontime.uniquecarrier = subquery.uniquecarrier
AND distance = min_distance;
```

By performing the de-correlation manually, the performance of SQLite and Postgres improves significantly. However, both systems remain over 30× slower than DuckDB.

DuckDB	Postgres	SQLite
0.06 s	1.98 s	2.81 s

Note that while it is possible to manually decorrelate certain subqueries by rewriting the SQL, it is not always possible to do so. As described in the Unnesting Arbitrary Queries paper, special join types that are not present in SQL are necessary to decorrelate arbitrary queries.

In DuckDB, these special join types will be automatically generated by the system to decorrelate all subqueries. In fact, DuckDB does not have support for executing subqueries that are not decorrelated. All subqueries will be decorrelated before DuckDB executes them.

Conclusion

Subqueries are a very powerful tool that allow you to take arbitrary queries and convert them into ad-hoc functions. When used in combination with DuckDB's powerful subquery decorrelation, they can be executed extremely efficiently, making previously intractable queries not only possible, but fast.

From Waddle to Flying: Quickly Expanding DuckDB's Functionality with Scalar Python UDFs

Publication date: 2023-07-07

Authors: Pedro Holanda, Thijs Bruineman and Phillip Cloud

TL;DR: DuckDB now supports vectorized Scalar Python User Defined Functions (UDFs). By implementing Python UDFs, users can easily expand the functionality of DuckDB while taking advantage of DuckDB's fast execution model, SQL and data safety.

User Defined Functions (UDFs) enable users to extend the functionality of a Database Management System (DBMS) to perform domain-specific tasks that are not implemented as built-in functions. For instance, users who frequently need to export private data can benefit from an anonymization function that masks the local part of an email while preserving the domain. Ideally, this function would be executed directly in the DBMS. This approach offers several advantages:

1) **Performance.** The function could be executed using the same execution model (e.g., streaming results, beyond-memory/out-of-core execution) of the DBMS, and without any unnecessary transformations.

2) **Easy Use.** UDFs can be seamlessly integrated into SQL queries, allowing users to leverage the power of SQL to call the functions. This eliminates the need for passing data through a separate database connector and executing external code. The functions can be utilized in various SQL contexts (e.g., subqueries, join conditions).

3) **Safety.** The sensitive data never leaves the DBMS process.

There are two main reasons users often refrain from implementing UDFs. 1) There are security concerns associated with UDFs. Since UDFs are custom code created by users and executed within the DBMS process, there is a potential risk of crashing the server. However, when it comes to DuckDB, an embedded database, this concern is mitigated as each analyst runs their own DuckDB process separately. Therefore, the impact on server stability is not a significant worry. 2) The difficulty of implementation is a common deterrent for users. High-Performance UDFs are typically only supported in low-level languages. UDFs in higher-level languages like Python incur significant performance costs. Consequently many users cannot quickly implement their UDFs without investing a significant amount of time in learning a low-level language and understanding the internal details of the DBMS.

DuckDB followed a similar approach. As a DBMS tailored for analytical tasks, performance is a key consideration, leading to the implementation of its core in C++. Consequently, the initial focus of extensibility efforts was centered around C++. However, this duck is not limited to just waddling; it can also fly. So we are delighted to announce the recent addition of Scalar Python UDFs to DuckDB.

DuckDB provides support for two distinct types of Python UDFs, differing in the Python object used for communication between DuckDB's native data types and the Python process. These communication layers include support for Python built-in types and PyArrow Tables.

The two approaches exhibit two key differences:

1) **Zero-Copy.** PyArrow Tables leverage our zero-copy integration with Arrow, enabling efficient translation of data types to Python-Land with zero-copy cost.

2) **Vectorization.** PyArrow Table functions operate on a chunk level, processing chunks of data containing up to 2048 rows. This approach maximizes cache locality and leverages vectorization. On the other hand, the built-in types UDF implementation operates on a per-row basis.

This blog post aims to demonstrate how you can extend DuckDB using Python UDFs, with a particular emphasis on PyArrow-powered UDFs. In our quick-tour section, we will provide examples using the PyArrow UDF types. For those interested in benchmarks, you can jump ahead to the benchmark section below. If you want to see a detailed description of the Python UDF API, please refer to our documentation.

Python UDFs

This section depicts several practical examples of using Python UDFs. Each example uses a different type of Python UDF.

Quick-Tour

To demonstrate the usage of Python UDFs in DuckDB, let's consider the following example. We have a dictionary called `world_cup_titles` that maps countries to the number of World Cups they have won. We want to create a Python UDF that takes a country name as input, searches for the corresponding value in the dictionary, and returns the number of World Cups won by that country. If the country is not found in the dictionary, the UDF will return `NULL`.

Here's an example implementation:

```
import duckdb
from duckdb.typing import *

con = duckdb.connect()

# Dictionary that maps countries and world cups they won
world_cup_titles = {
    "Brazil": 5,
    "Germany": 4,
    "Italy": 4,
    "Argentina": 2,
    "Uruguay": 2,
    "France": 2,
    "England": 1,
    "Spain": 1
}

# Function that will be registered as an UDF, simply does a lookup in the python dictionary
def world_cups(x):
     return world_cup_titles.get(x)

# We register the function
con.create_function("wc_titles", world_cups, [VARCHAR], INTEGER)
```

That's it, the function is then registered and ready to be called through SQL.

```
# Let's create an example countries table with the countries we are interested in using
con.execute("CREATE TABLE countries (country VARCHAR)")
con.execute("INSERT INTO countries VALUES ('Brazil'), ('Germany'), ('Italy'), ('Argentina'),
('Uruguay'), ('France'), ('England'), ('Spain'), ('Netherlands')")
# We can simply call the function through SQL, and even use the function return to eliminate the
countries that never won a world cup
con.sql("SELECT country, wc_titles(country) AS world_cups FROM countries").fetchall()
# [('Brazil', 5), ('Germany', 4), ('Italy', 4), ('Argentina', 2), ('Uruguay', 2), ('France', 2),
('England', 1), ('Spain', 1), ('Netherlands', None)]
```

Generating Fake Data with Faker (Built-In Type UDF)

Here is an example that demonstrates the usage of the Faker library to generate a scalar function in DuckDB, which returns randomly generated dates. The function, named `random_date`, does not require any inputs and outputs a DATE column. Since Faker utilizes built-in Python types, the function directly returns them. One important thing to notice is that a function that is not deterministic based on its input must be marked as having `side_effects`.

```
import duckdb

# By importing duckdb.typing we can specify DuckDB Types directly without using strings
from duckdb.typing import *

from faker import Faker

# Our Python UDF generates a random date every time it's called
def random_date():
     fake = Faker()
     return fake.date_between()
```

We then have to register the Python function in DuckDB using `create_function`. Since our function doesn't require any inputs, we can pass an empty list as the `argument_type_list`. As the function returns a date, we specify DATE from `duckdb.typing` as the `return_type`. Note that since our `random_date()` function returns a built-in Python type (`datetime.date`), we don't need to specify the UDF type.

```
# To exemplify the effect of side-effect, let's first run the function without marking it.
duckdb.create_function('random_date', random_date, [], DATE)

# After registration, we can use the function directly via SQL
# Notice that without side_effect=True, it's not guaranteed that the function will be re-evaluated.
res = duckdb.sql('SELECT random_date() FROM range (3)').fetchall()
# [(datetime.date(2003, 8, 3),), (datetime.date(2003, 8, 3),), (datetime.date(2003, 8, 3),)]

# Now let's re-add the function with side-effects marked as true.
duckdb.remove_function('random_date')
duckdb.create_function('random_date', random_date, [], DATE, side_effects=True)
res = duckdb.sql('SELECT random_date() FROM range (3)').fetchall()
# [(datetime.date(2020, 11, 29),), (datetime.date(2009, 5, 18),), (datetime.date(2018, 5, 24),)]
```

Swap String Case (PyArrow Type UDF)

One issue with using built-in types is that you don't benefit from zero-copy, vectorization and cache locality. Using PyArrow as a UDF type should be favored to leverage these optimizations.

To demonstrate a PyArrow function, let's consider a simple example where we want to transform lowercase characters to uppercase and uppercase characters to lowercase. Fortunately, PyArrow already has a function for this in the compute engine, and it's as simple as calling `pc.utf8_swapcase(x)`.

```python
import duckdb

# By importing duckdb.typing we can specify DuckDB Types directly without using strings
from duckdb.typing import *

import pyarrow as pa
import pyarrow.compute as pc

def swap_case(x):
     # Swap the case of the 'column' using utf8_swapcase and return the result
     return pc.utf8_swapcase(x)

con = duckdb.connect()
# To register the function, we must define it's type to be 'arrow'
con.create_function('swap_case', swap_case, [VARCHAR], VARCHAR, type='arrow')

res = con.sql("SELECT swap_case('PEDRO HOLANDA')").fetchall()
# [('pedro holanda',)]
```

Predicting Taxi Fare Costs (Ibis + PyArrow UDF)

Python UDFs offer significant power as they enable users to leverage the extensive Python ecosystem and tools, including libraries like PyTorch and Tensorflow that efficiently implement machine learning operations.

Additionally the Ibis project offers a DataFrame API with great DuckDB integration and supports both of DuckDB's native Python and PyArrow UDFs.

In this example, we demonstrate the usage of a pre-built PyTorch model to estimate taxi fare costs based on the traveled distance. You can find a complete example in this blog post by the Ibis team.

```python
import torch
import pyarrow as pa
import ibis
import ibis.expr.datatypes as dt

from ibis.expr.operations import udf


# The code to generate the model is not specified in this snippet, please refer to the provided link for
more information
model = ...

# Function that uses the model and a traveled distance input tensor to predict values, please refer to
the provided link for more information
def predict_linear_regression(model, tensor: torch.Tensor) -> torch.Tensor:
    ...


# Indicate to ibis that this is a scalar user-defined function whose input format is pyarrow
@udf.scalar.pyarrow
def predict_fare(x: dt.float64) -> dt.float32:
    # `x` is a pyarrow.ChunkedArray; the `dt.float64` annotation indicate the element type of the
    ChunkedArray.

    # Transform the data from PyArrow to the required torch tensor format and dimension.
    tensor = torch.from_numpy(x.to_numpy()[:, None]).float()

    # Call the actual prediction function, which also returns a torch tensor.
    predicted = predict_linear_regression(model, tensor).ravel()
```

```
    return pa.array(predicted.numpy())


# Execute a query on the NYC Taxi parquet file to showcase our model's predictions, the actual fare
amount, and the distance.
expr = (
    ibis.read_parquet('yellow_tripdata_2016-02.parquet')
    .mutate(
        "fare_amount",
        "trip_distance",
        predicted_fare=lambda t: predict_fare(t.trip_distance),
    )
)
df = expr.execute()
```

By utilizing Python UDFs in DuckDB with Ibis, you can seamlessly incorporate machine learning models and perform predictions directly within your Ibis code and SQL queries. The example demonstrates how to predict taxi fare costs based on distance using a PyTorch model, showcasing the integration of machine learning capabilities within DuckDB's SQL environment driven by Ibis.

Benchmarks

In this section, we will perform simple benchmark comparisons to demonstrate the performance differences between two different types of Python UDFs. The benchmark will measure the execution time, and peak memory consumption. The benchmarks are executed 5 times, and the median value is considered. The benchmark is conducted on a Mac Apple M1 with 16 GB of RAM.

Built-In Python vs. PyArrow

To benchmark these UDF types, we create UDFs that take an integral column as input, add one to each value, and return the result. The code used for this benchmark section can be found in a GitHub Gist.

```
import pyarrow.compute as pc
import duckdb
import pyarrow as pa

# Built-In UDF
def add_built_in_type(x):
    return x + 1

# Arrow UDF
def add_arrow_type(x):
    return pc.add(x, 1)

con = duckdb.connect()

# Registration
con.create_function('built_in_types', add_built_in_type, ['BIGINT'], 'BIGINT', type='native')
con.create_function('add_arrow_type', add_arrow_type, ['BIGINT'], 'BIGINT', type='arrow')

# Integer View with 10,000,000 elements.
con.sql("""
     SELECT i
     FROM range(10000000) tbl(i);
""").to_view("numbers")

# Calls for both UDFs
native_res = con.sql("SELECT sum(add_built_in_type(i)) FROM numbers").fetchall()
arrow_res = con.sql("SELECT sum(add_arrow_type(i)) FROM numbers").fetchall()
```

Name	Time (s)
Built-In	5.37
PyArrow	0.35

We can observe a performance difference of more than one order of magnitude between the two UDFs. The difference in performance is primarily due to three factors:

1) In Python, object construction and general use is rather slow. This is due to several reasons, including automatic memory management, interpretation, and dynamic typing. 2) The PyArrow UDF does not require any data copying. 3) The PyArrow UDF is executed in a vectorized fashion, processing chunks of data instead of individual rows.

Python UDFs vs. External Functions

Here we compare the usage of a Python UDF with an external function. In this case, we have a function that calculates the sum of the lengths of all strings in a column. You can find the code used for this benchmark section in a GitHub Gist.

```
import duckdb
import pyarrow as pa

# Function used in UDF
def string_length_arrow(x):
     tuples = len(x)
     values = [len(i.as_py()) if i.as_py() != None else 0 for i in x]
     array = pa.array(values, type=pa.int32(), size=tuples)
     return array


# Same Function but external to the database
def exec_external(con):
     arrow_table = con.sql("SELECT i FROM strings tbl(i)").arrow()
     arrow_column = arrow_table['i']
     tuples = len(arrow_column)
     values = [len(i.as_py()) if i.as_py() != None else 0 for i in arrow_column]
     array = pa.array(values, type=pa.int32(), size=tuples)
     arrow_tbl = pa.Table.from_arrays([array], names=['i'])
     return con.sql("SELECT sum(i) FROM arrow_tbl").fetchall()


con = duckdb.connect()
con.create_function('strlen_arrow', string_length_arrow, ['VARCHAR'], int, type='arrow')

con.sql("""
     SELECT
          CASE WHEN i != 0 AND i % 42 = 0
          THEN
               NULL
          ELSE
               repeat(chr((65 + (i % 26))::INTEGER), (4 + (i % 12))) END
          FROM range(10000000) tbl(i);
""").to_view("strings")

con.sql("SELECT sum(strlen_arrow(i)) FROM strings tbl(i)").fetchall()

exec_external(con)
```

Name	Time (s)	Peak memory consumption (MB)
External	5.65	584.032
UDF	5.63	112.848

Here we can see that there is no significant regression in performance when utilizing UDFs. However, you still have the benefits of safer execution and the utilization of SQL. In our example, we can also notice that the external function materializes the entire query, resulting in a 5× higher peak memory consumption compared to the UDF approach.

Conclusions and Further Development

Scalar Python UDFs are now supported in DuckDB, marking a significant milestone in extending the functionality of the database. This enhancement empowers users to perform complex computations using a high-level language. Additionally, Python UDFs can leverage DuckDB's zero-copy integration with Arrow, eliminating data transfer costs and ensuring efficient query execution.

While the introduction of Python UDFs is a major step forward, our work in this area is ongoing. Our roadmap includes the following focus areas:

1. **Aggregate/Table-Producing UDFs**: Currently, users can create Scalar UDFs, but we are actively working on supporting Aggregation Functions (which perform calculations on a set of values and return a single result) and Table-Producing Functions (which return tables without limitations on the number of columns and rows).
2. **Types**: Scalar Python UDFs currently support most DuckDB types, with the exception of ENUM types and BIT types. We are working towards expanding the type support to ensure comprehensive functionality.

As always, we are happy to hear your thoughts! Feel free to drop us an email if you have any suggestions, comments or questions.

Last but not least, if you encounter any problems using our Python UDFs, please open an issue in DuckDB's issue tracker.

DuckDB ADBC – Zero-Copy Data Transfer via Arrow Database Connectivity

Publication date: 2023-08-04

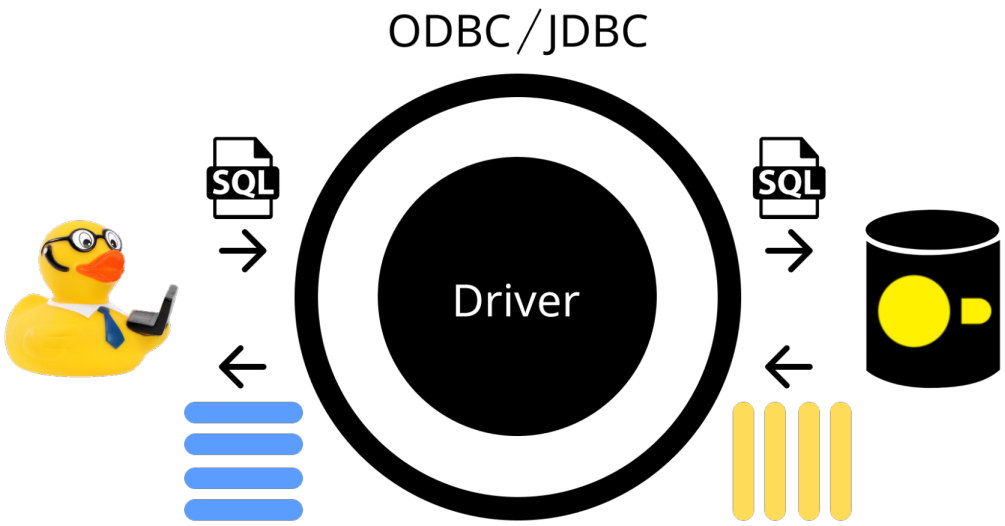
Author: Pedro Holanda

TL;DR: DuckDB has added support for Arrow Database Connectivity (ADBC), an API standard that enables efficient data ingestion and retrieval from database systems, similar to Open Database Connectivity (ODBC) interface. However, unlike ODBC, ADBC specifically caters to the columnar storage model, facilitating fast data transfers between a columnar database and an external application.

Database interface standards allow developers to write application code that is independent of the underlying database management system (DBMS) being used. DuckDB has supported two standards that have gained popularity in the past few decades: the core interface of ODBC and Java Database Connectivity (JDBC). Both interfaces are designed to fully support database connectivity and management, with JDBC being catered for the Java environment. With these APIs, developers can query DBMS agnostically, retrieve query results, run prepared statements, and manage connections.

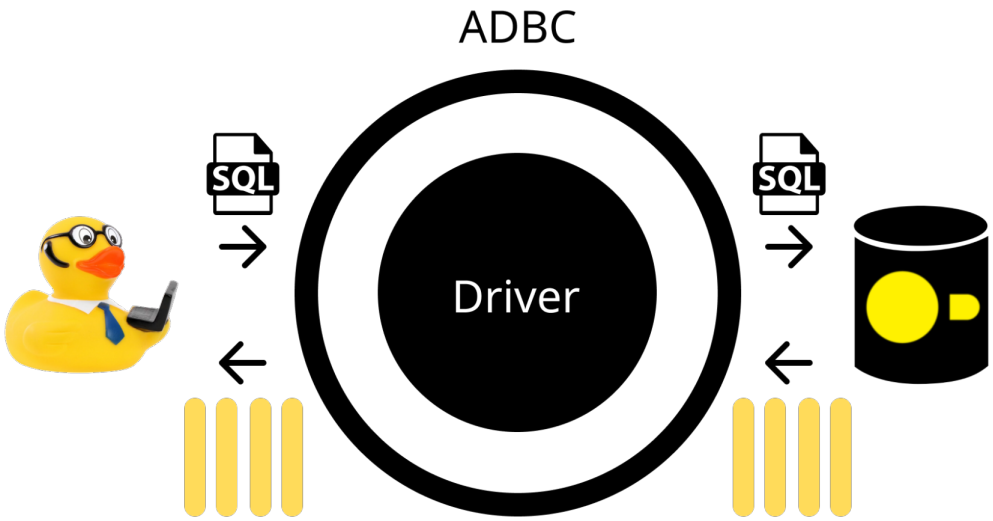
These interfaces were designed in the early 90s when row-wise database systems reigned supreme. As a result, they were primarily intended for transferring data in a row-wise format. However, in the mid-2000s, columnar-wise database systems started gaining a lot of traction due to their drastic performance advantages for data analysis (you can find myself giving a brief exemplification of this difference at EuroPython). This means that these APIs offer no support for transferring data in a columnar-wise format (or, in the case of ODBC, some support with a lot of added complexity). In practice, when analytical, column-wise systems like DuckDB make use of these APIs, converting the data between these representation formats becomes a major bottleneck.

The figure below depicts how a developer can use these APIs to query a DuckDB database. For example, developers can submit SQL queries via the API, which then uses a DuckDB driver to internally call the proper functions. A query result is then produced in DuckDB's internal columnar representation, and the driver takes care of transforming it to the JDBC or ODBC row-wise result format. This transformation has significant costs for rearranging and copying the data, quickly becoming a major bottleneck.



To overcome this transformation cost, ADBC has been proposed, with a generic API to support database operations while using the Apache Arrow memory format to send data in and out of the DBMS. DuckDB now supports the ADBC specification. Due to DuckDB's zero-copy integration with the Arrow format, using ADBC as an interface is rather efficient, since there is only a small *constant* cost to transform DuckDB query results to the Arrow format.

The figure below depicts the query execution flow when using ADBC. Note that the main difference between ODBC/JDBC is that the result does not need to be transformed to a row-wise format.



Quick Tour

For our quick tour, we will illustrate an example of round-tripping data using DuckDB-ADBC via Python. Please note that DuckDB-ADBC can also be utilized with other programming languages. Specifically, you can find C++ DuckDB-ADBC examples and tests in the DuckDB GitHub repository along with usage examples available in C++. For convenience, you can also find a ready-to-run version of this tour in a Colab notebook. If you would like to see a more detailed explanation of the DuckDB-ADBC API or view a C++ example, please refer to our documentation page.

Setup

For this example, you must have a dynamic library from the latest bleeding-edge version of DuckDB, pyarrow, and the adbc-driver-manager. The ADBC driver manager is a Python package developed by Voltron Data. The driver manager is compliant with DB-API 2.0. It wraps ADBC, making its usage more straightforward. For details on ADBC drivers, see the find the documentation of the ADBC Driver Manager.

> While DuckDB is already DB-API compliant in Python, what sets ADBC apart is that you do not need a DuckDB module installed and loaded. Additionally, unlike the DB-API, it does not utilize row-wise as its data transfer format of choice.

```
pip install pyarrow
pip install adbc-driver-manager
```

Insert Data

First, we need to include the necessary libraries that will be used in this tour. Mainly, PyArrow and the DBAPI from the ADBC Driver Manager.

```
import pyarrow
from adbc_driver_manager import dbapi
```

Next, we can create a connection via ADBC with DuckDB. This connection simply requires the path to DuckDB's driver and the entrypoint function name. DuckDB's entrypoint is `duckdb_adbc_init`. By default, connections are established with an in-memory database. However, if desired, you have the option to specify the `path` variable and connect to a local duckdb instance, allowing you to store the data on disk. Note that these are the only variables in ADBC that are not DBMS agnostic; instead, they are set by the user, often through a configuration file.

```
con = dbapi.connect(driver="path/to/duckdb.lib", entrypoint="duckdb_adbc_init", db_kwargs={"path":
"test.db"})
```

To insert the data, we can simply call the `adbc_ingest` function with a cursor from our connection. It requires the name of the table we want to perform the ingestion to and the Arrow Python object we want to ingest. This function also has two modes: `append`, where data is appended to an existing table, and `create`, where the table does not exist yet and will be created with the input data. By default, it's set to create, so we don't need to define it here.

```
table = pyarrow.table(
     [
          ["Tenacious D", "Backstreet Boys", "Wu Tang Clan"],
          [4, 10, 7]

     ],
     names=["Name", "Albums"],
)

with con.cursor() as cursor:
     cursor.adbc_ingest("Bands", table)
```

After calling `adbc_ingest`, the table is created in the DuckDB connection and the data is fully inserted.

Read Data

To read data from DuckDB, one simply needs to use the `execute` function with a SQL query and then return the cursor's result to the desired Arrow format, such as a PyArrow Table in this example.

```
with con.cursor() as cursor:
     cursor.execute("SELECT * FROM Bands")
     cursor.fetch_arrow_table()
```

Benchmark ADBC vs ODBC

In our benchmark section, we aim to evaluate the differences in data reading from DuckDB via ADBC and ODBC. This benchmark was executed on an Apple M1 Max with 32 GB of RAM and involves outputting and inserting the `lineitem` table of TPC-H SF 1. You can find the repository with the code used to run this benchmark on GitHub.

Name	Time (s)
ODBC	28.149
ADBC	0.724

The time difference between ODBC and ADBC is 38x. This significant contrast results from the extra allocations and copies that exist in ODBC.

Conclusions

DuckDB now supports the ADBC standard for database connection. ADBC is particularly efficient when combined with DuckDB, thanks to its use of the Arrow zero-copy integration.

ADBC is particularly interesting because it can drastically decrease interactions between analytic systems compared to ODBC. For example, if software that already support ODBC, e.g., if MS-Excel was to implement ADBC, integrations with columnar systems like DuckDB could benefit from this significant difference in performance.

DuckDB-ADBC is currently supported via the C Interface and through the Python ADBC Driver Manager. We will add more extensive tutorials for other languages to our documentation webpage. Please feel free to let us know your preferred language for interacting with DuckDB via ADBC!

As always, we are happy to hear your thoughts! Feel free to drop us an email if you have any suggestions, comments or questions!

Last but not least, if you encounter any problems using ADBC, please open an issue in DuckDB's issue tracker.

Even Friendlier SQL with DuckDB

Publication date: 2023-08-23

Author: Alex Monahan

TL;DR: DuckDB continues to push the boundaries of SQL syntax to both simplify queries and make more advanced analyses possible. Highlights include dynamic column selection, queries that start with the FROM clause, function chaining, and list comprehensions. We boldly go where no SQL engine has gone before! For more details, see the documentation for friendly SQL features.

Who says that SQL should stay frozen in time, chained to a 1999 version of the specification? As a comparison, do folks remember what JavaScript felt like before Promises? Those didn't launch until 2012! It's clear that innovation at the programming syntax layer can have a profoundly positive impact on an entire language ecosystem.

We believe there are many valid reasons for innovation in the SQL language, among them opportunities to simplify basic queries and also

to make more dynamic analyses possible. Many of these features arose from community suggestions! Please let us know your SQL pain points on Discord or GitHub and join us as we change what it feels like to write SQL!

If you have not had a chance to read the first installment in this series, please take a quick look to the prior blog post, "Friendlier SQL with DuckDB".

The Future Is Now

The first few enhancements in this list were included in the "Ideas for the Future" section of the prior post.

Reusable Column Aliases

When working with incremental calculated expressions in a select statement, traditional SQL dialects force you to either write out the full expression for each column or create a common table expression (CTE) around each step of the calculation. Now, any column alias can be reused by subsequent columns within the same select statement. Not only that, but these aliases can be used in the where and order by clauses as well.

Old Way 1: Repeat Yourself

```
SELECT
    'These are the voyages of the starship Enterprise...' AS intro,
    instr('These are the voyages of the starship Enterprise...', 'starship')
        AS starship_loc
    substr('These are the voyages of the starship Enterprise...',
    instr('These are the voyages of the starship Enterprise...', 'starship')
        + len('starship') + 1) AS trimmed_intro;
```

Old Way 2: All the CTEs

```
WITH intro_cte AS (
    SELECT
        'These are the voyages of the starship Enterprise...' AS intro
), starship_loc_cte AS (
    SELECT
        intro,
        instr(intro, 'starship') AS starship_loc
    FROM intro_cte
)
SELECT
    intro,
    starship_loc,
    substr(intro, starship_loc + len('starship') + 1) AS trimmed_intro
FROM starship_loc_cte;
```

New Way

```
SELECT
     'These are the voyages of the starship Enterprise...' AS intro,
     instr(intro, 'starship') AS starship_loc,
     substr(intro, starship_loc + len('starship') + 1) AS trimmed_intro;
```

intro	starship_loc	trimmed_intro
These are the voyages of the starship Enterprise...	30	Enterprise...

Dynamic Column Selection

Databases typically prefer strictness in column definitions and flexibility in the number of rows. This can help by enforcing data types and recording column level metadata. However, in data science workflows and elsewhere, it is very common to dynamically generate columns (for example during feature engineering).

No longer do you need to know all of your column names up front! DuckDB can select and even modify columns based on regular expression pattern matching, EXCLUDE or REPLACE modifiers, and even lambda functions (see the section on lambda functions below for details!).

Let's take a look at some facts gathered about the first season of Star Trek. Using DuckDB's `httpfs` extension, we can query a CSV dataset directly from GitHub. It has several columns so let's `DESCRIBE` it.

```
INSTALL httpfs;
LOAD httpfs;

CREATE TABLE trek_facts AS
    SELECT *
    FROM 'https://blobs.duckdb.org/data/Star_Trek-Season_1.csv';

DESCRIBE trek_facts;
```

column_name	column_type	null	key	default	extra
season_num	BIGINT	YES	NULL	NULL	NULL
episode_num	BIGINT	YES	NULL	NULL	NULL
aired_date	DATE	YES	NULL	NULL	NULL
cnt_kirk_hookups	BIGINT	YES	NULL	NULL	NULL
cnt_downed_redshirts	BIGINT	YES	NULL	NULL	NULL
bool_aliens_almost_took_over_planet	BIGINT	YES	NULL	NULL	NULL
bool_aliens_almost_took_over_enterprise	BIGINT	YES	NULL	NULL	NULL
cnt_vulcan_nerve_pinch	BIGINT	YES	NULL	NULL	NULL
cnt_warp_speed_orders	BIGINT	YES	NULL	NULL	NULL
highest_warp_speed_issued	BIGINT	YES	NULL	NULL	NULL
bool_hand_phasers_fired	BIGINT	YES	NULL	NULL	NULL
bool_ship_phasers_fired	BIGINT	YES	NULL	NULL	NULL

column_name	column_type	null	key	default	extra
bool_ship_photon_torpedos_fired	BIGINT	YES	NULL	NULL	NULL
cnt_transporter_pax	BIGINT	YES	NULL	NULL	NULL
cnt_damn_it_jim_quote	BIGINT	YES	NULL	NULL	NULL
cnt_im_givin_her_all_shes_got_quote	BIGINT	YES	NULL	NULL	NULL
cnt_highly_illogical_quote	BIGINT	YES	NULL	NULL	NULL
bool_enterprise_saved_the_day	BIGINT	YES	NULL	NULL	NULL

`COLUMNS()` with Regular Expressions

The `COLUMNS` expression can accept a string parameter that is a regular expression and will return all column names that match the pattern. How did warp change over the first season? Let's examine any column name that contains the word `warp`.

```
SELECT
    episode_num,
    COLUMNS('.*warp.*')
FROM trek_facts;
```

episode_num	cnt_warp_speed_orders	highest_warp_speed_issued
0	1	1
1	0	0
2	1	1
3	1	0
...	...	...
27	1	1
28	0	0
29	2	8

The `COLUMNS` expression can also be wrapped by other functions to apply those functions to each selected column. Let's simplify the above query to look at the maximum values across all episodes:

```
SELECT
    max(COLUMNS('.*warp.*'))
FROM trek_facts;
```

max(trek_facts.cnt_warp_speed_orders)	max(trek_facts.highest_warp_speed_issued)
5	8

We can also create a `WHERE` clause that applies across multiple columns. All columns must match the filter criteria, which is equivalent to combining them with `AND`. Which episodes had at least 2 warp speed orders and at least a warp speed level of 2?

```
SELECT
    episode_num,
    COLUMNS('.*warp.*')
FROM trek_facts
WHERE
    COLUMNS('.*warp.*') >= 2;
    -- cnt_warp_speed_orders >= 2
    -- AND
    -- highest_warp_speed_issued >= 2
```

episode_num	cnt_warp_speed_orders	highest_warp_speed_issued
14	3	7
17	2	7
18	2	8
29	2	8

COLUMNS() with EXCLUDE and REPLACE

Individual columns can also be either excluded or replaced prior to applying calculations on them. For example, since our dataset only includes season 1, we do not need to find the `max` of that column. It would be highly illogical.

```
SELECT
    max(COLUMNS(* EXCLUDE season_num))
FROM trek_facts;
```

max(trek_facts.episode_num)	max(trek_facts.aired_date)	max(trek_facts.cnt_kirk_hookups)	...	max(trek_facts.bool_enterprise_saved_the_day)
29	1967-04-13	2	...	1

The REPLACE syntax is also useful when applied to a dynamic set of columns. In this example, we want to convert the dates into timestamps prior to finding the maximum value in each column. Previously this would have required an entire subquery or CTE to pre-process just that single column!

```
SELECT
    max(COLUMNS(* REPLACE aired_date::timestamp AS aired_date))
FROM trek_facts;
```

max(trek_facts.season_num)	max(trek_facts.episode_num)	max(aired_date := CAST(aired_date AS TIMESTAMP))	...	max(trek_facts.bool_enterprise_saved_the_day)
1	29	1967-04-13 00:00:00	...	1

COLUMNS() with Lambda Functions

The most flexible way to query a dynamic set of columns is through a lambda function. This allows for any matching criteria to be applied to the names of the columns, not just regular expressions. See more details about lambda functions below.

For example, if using the `LIKE` syntax is more comfortable, we can select columns matching a `LIKE` pattern rather than with a regular expression.

```
SELECT
    episode_num,
    COLUMNS(col -> col LIKE '%warp%')
FROM trek_facts
WHERE
    COLUMNS(col -> col LIKE '%warp%') >= 2;
```

episode_num	cnt_warp_speed_orders	highest_warp_speed_issued
14	3	7
17	2	7
18	2	8
29	2	8

Automatic JSON to Nested Types Conversion

The first installment in the series mentioned JSON dot notation references as future work. However, the team has gone even further! Instead of referring to JSON-typed columns using dot notation, JSON can now be automatically parsed into DuckDB's native types for significantly faster performance, compression, as well as that friendly dot notation!

First, install and load the `httpfs` and `json` extensions if they don't come bundled with the client you are using. Then query a remote JSON file directly as if it were a table!

```
INSTALL httpfs;
LOAD httpfs;
INSTALL json;
LOAD json;

SELECT
     starfleet[10].model AS starship
FROM 'https://raw.githubusercontent.com/vlad-saling/star-trek-ipsum/master/src/content/content.json';
```

starship
USS Farragut - NCC-1647 - Ship on which James Kirk served as a phaser station operator. Attacked by the Dikironium Cloud Creature, killing half the crew. ad.

Now for some new SQL capabilities beyond the ideas from the prior post!

FROM First in SELECT Statements

When building a query, the first thing you need to know is where your data is coming FROM. Well then why is that the second clause in a SELECT statement?? No longer! DuckDB is building SQL as it should have always been – putting the FROM clause first! This addresses one of the longest standing complaints about SQL, and the DuckDB team implemented it in 2 days.

```
FROM my_table SELECT my_column;
```

Not only that, the SELECT statement can be completely removed and DuckDB will assume all columns should be SELECTed. Taking a look at a table is now as simple as:

```
FROM my_table;
-- SELECT * FROM my_table
```

Other statements like COPY are simplified as well.

```
COPY (FROM trek_facts) TO 'phaser_filled_facts.parquet';
```

This has an additional benefit beyond saving keystrokes and staying in a development flow state: autocomplete will have much more context when you begin to choose columns to query. Give the AI a helping hand!

Note that this syntax is completely optional, so your `SELECT * FROM` keyboard shortcuts are safe, even if they are obsolete...

Function Chaining

Many SQL blogs advise the use of CTEs instead of subqueries. Among other benefits, they are much more readable. Operations are compartmentalized into discrete chunks and they can be read in order top to bottom instead of forcing the reader to work their way inside out.

DuckDB enables the same interpretability improvement for every scalar function! Use the dot operator to chain functions together, just like in Python. The prior expression in the chain is used as the first argument to the subsequent function.

```
SELECT
     ('Make it so')
          .upper()
          .string_split(' ')
          .list_aggr('string_agg','.')
          .concat('.') AS im_not_messing_around_number_one;
```

im_not_messing_around_number_one
MAKE.IT.SO.

Now compare that with the old way...

```
SELECT
     concat(
          list_aggr(
               string_split(
                    upper('Make it stop'),
               ' '),
          'string_agg','.'),
     '.') AS oof;
```

oof
MAKE.IT.STOP.

Union by Name

DuckDB aims to blend the best of databases and dataframes. This new syntax is inspired by the concat function in Pandas. Rather than vertically stacking tables based on column position, columns are matched by name and stacked accordingly. Simply replace `UNION` with `UNION BY NAME` or `UNION ALL` with `UNION ALL BY NAME`.

For example, we had to add some new alien species proverbs in The Next Generation:

```
CREATE TABLE proverbs AS
     SELECT
          'Revenge is a dish best served cold' AS klingon_proverb
     UNION ALL BY NAME
     SELECT
          'You will be assimilated' AS borg_proverb,
          'If winning is not important, why keep score?' AS klingon_proverb;

FROM proverbs;
```

klingon_proverb	borg_proverb
Revenge is a dish best served cold	NULL
If winning is not important, why keep score?	You will be assimilated

This approach has additional benefits. As seen above, not only can tables with different column orders be combined, but so can tables with different numbers of columns entirely. This is helpful as schemas migrate, and is particularly useful for DuckDB's multi-file reading capabilities.

Insert by Name

Another common situation where column order is strict in SQL is when inserting data into a table. Either the columns must match the order exactly, or all of the column names must be repeated in two locations within the query.

Instead, add the keywords BY NAME after the table name when inserting. Any subset of the columns in the table in any order can be inserted.

```
INSERT INTO proverbs BY NAME
     SELECT 'Resistance is futile' AS borg_proverb;

SELECT * FROM proverbs;
```

klingon_proverb	borg_proverb
Revenge is a dish best served cold	NULL
If winning is not important, why keep score?	You will be assimilated
NULL	Resistance is futile

Dynamic `PIVOT` and `UNPIVOT`

Historically, databases are not well-suited for pivoting operations. However, DuckDB's `PIVOT` and `UNPIVOT` clauses can create or stack dynamic column names for a truly flexible pivoting capability! In addition to that flexibility, DuckDB also provides both the SQL standard syntax and a friendlier shorthand.

For example, let's take a look at some procurement forecast data just as the Earth-Romulan war was beginning:

```
CREATE TABLE purchases (item VARCHAR, year INTEGER, count INTEGER);

INSERT INTO purchases
    VALUES ('phasers', 2155, 1035),
           ('phasers', 2156, 25039),
           ('phasers', 2157, 95000),
```

```
          ('photon torpedoes', 2155, 255),
          ('photon torpedoes', 2156, 17899),
          ('photon torpedoes', 2157, 87492);

FROM purchases;
```

item	year	count
phasers	2155	1035
phasers	2156	25039
phasers	2157	95000
photon torpedoes	2155	255
photon torpedoes	2156	17899
photon torpedoes	2157	87492

It is easier to compare our phaser needs to our photon torpedo needs if each year's data is visually close together. Let's pivot this into a friendlier format! Each year should receive its own column (but each year shouldn't need to be specified in the query!), we want to sum up the total `count`, and we still want to keep a separate group (row) for each `item`.

```
CREATE TABLE pivoted_purchases AS
     PIVOT purchases
          ON year
          USING sum(count)
          GROUP BY item;

FROM pivoted_purchases;
```

item	2155	2156	2157
phasers	1035	25039	95000
photon torpedoes	255	17899	87492

Looks like photon torpedoes went on sale...

Now imagine the reverse situation. Scotty in engineering has been visually analyzing and manually constructing his purchases forecast. He prefers things pivoted so it's easier to read. Now you need to fit it back into the database! This war may go on for a bit, so you may need to do this again next year. Let's write an `UNPIVOT` query to return to the original format that can handle any year.

The `COLUMNS` expression will use all columns except `item`. After stacking, the column containing the column names from `pivoted_purchases` should be renamed to `year`, and the values within those columns represent the `count`. The result is the same dataset as the original.

```
UNPIVOT pivoted_purchases
     ON COLUMNS(* EXCLUDE item)
     INTO
          NAME year
          VALUE count;
```

item	year	count
phasers	2155	1035
phasers	2156	25039

item	year	count
phasers	2157	95000
photon torpedoes	2155	255
photon torpedoes	2156	17899
photon torpedoes	2157	87492

More examples are included as a part of our DuckDB 0.8.0 announcement post, and the `PIVOT` and `UNPIVOT` documentation pages highlight more complex queries.

Stay tuned for a future post to cover what is happening behind the scenes!

List Lambda Functions

List lambdas allow for operations to be applied to each item in a list. These do not need to be pre-defined – they are created on the fly within the query.

In this example, a lambda function is used in combination with the `list_transform` function to shorten each official ship name.

```
SELECT
     (['Enterprise NCC-1701', 'Voyager NCC-74656', 'Discovery NCC-1031'])
          .list_transform(x -> x.string_split(' ')[1]) AS short_name;
```

ship_name
[Enterprise, Voyager, Discovery]

Lambdas can also be used to filter down the items in a list. The lambda returns a list of booleans, which is used by the `list_filter` function to select specific items. The `contains` function is using the function chaining described earlier.

```
SELECT
     (['Enterprise NCC-1701', 'Voyager NCC-74656', 'Discovery NCC-1031'])
          .list_filter(x -> x.contains('1701')) AS the_original;
```

the_original
[Enterprise NCC-1701]

List Comprehensions

What if there was a simple syntax to both modify and filter a list? DuckDB takes inspiration from Python's approach to list comprehensions to dramatically simplify the above examples. List comprehensions are syntactic sugar – these queries are rewritten into lambda expressions behind the scenes!

Within brackets, first specify the transformation that is desired, then indicate which list should be iterated over, and finally include the filter criteria.

```
SELECT
     [x.string_split(' ')[1]
     FOR x IN ['Enterprise NCC-1701', 'Voyager NCC-74656', 'Discovery NCC-1031']
     IF x.contains('1701')] AS ready_to_boldly_go;
```

ready_to_boldly_go
[Enterprise]

Exploding Struct.*

A struct in DuckDB is a set of key/value pairs. Behind the scenes, a struct is stored with a separate column for each key. As a result, it is computationally easy to explode a struct into separate columns, and now it is also syntactically simple as well! This is another example of allowing SQL to handle dynamic column names.

```
WITH damage_report AS (
     SELECT {'gold_casualties':5, 'blue_casualties':15, 'red_casualties': 10000} AS casualties
)
FROM damage_report
SELECT
     casualties.*;
```

gold_casualties	blue_casualties	red_casualties
5	15	10000

Automatic Struct Creation

DuckDB exposes an easy way to convert any table into a single-column struct. Instead of SELECTing column names, SELECT the table name itself.

```
WITH officers AS (
     SELECT 'Captain' AS rank, 'Jean-Luc Picard' AS name
     UNION ALL
     SELECT 'Lieutenant Commander', 'Data'
)
FROM officers
SELECT officers;
```

officers
{'rank': Captain, 'name': Jean-Luc Picard}
{'rank': Lieutenant Commander, 'name': Data}

Union Data Type

DuckDB utilizes strong typing to provide high performance and enforce data quality. However, DuckDB is also as forgiving as possible using approaches like implicit casting to avoid always having to cast between data types.

Another way DuckDB enables flexibility is the new UNION data type. A UNION data type allows for a single column to contain multiple types of values. This can be thought of as an "opt-in" to SQLite's flexible data typing rules (the opposite direction of SQLite's recently announced strict tables).

By default DuckDB will seek the common denominator of data types when combining tables together. The below query results in a VARCHAR column:

```
SELECT 'The Motion Picture' AS movie UNION ALL
SELECT 2 UNION ALL
SELECT 3 UNION ALL
SELECT 4 UNION ALL
SELECT 5 UNION ALL
SELECT 6 UNION ALL
SELECT 'First Contact';
```

movie
The Motion Picture
First Contact
6
5
4
3
2

However, if a `UNION` type is used, each individual row retains its original data type. A `UNION` is defined using key-value pairs with the key as a name and the value as the data type. This also allows the specific data types to be pulled out as individual columns:

```
CREATE TABLE movies (
     movie UNION(num INTEGER, name VARCHAR)
);
INSERT INTO movies VALUES
     ('The Motion Picture'), (2), (3), (4), (5), (6), ('First Contact');

FROM movies
SELECT
     movie,
     union_tag(movie) AS type,
     movie.name,
     movie.num;
```

movie	type	name	num
The Motion Picture	name	The Motion Picture	
2	num		2
3	num		3
4	num		4
5	num		5
6	num		6
First Contact	name	First Contact	

Additional Friendly Features

Several other friendly features are worth mentioning and some are powerful enough to warrant their own blog posts.

DuckDB takes a nod from the `describe` function in Pandas and implements a `SUMMARIZE` keyword that will calculate a variety of statistics about each column in a dataset for a quick, high-level overview. Simply prepend `SUMMARIZE` to any table or `SELECT` statement.

Have a look at the correlated subqueries post to see how to use subqueries that refer to each others' columns. DuckDB's advanced optimizer improves correlated subquery performance by orders of magnitude, allowing for queries to be expressed as naturally as possible. What was once an anti-pattern for performance reasons can now be used freely!

DuckDB has added more ways to `JOIN` tables together that make expressing common calculations much easier. Some like `LATERAL`, `ASOF`, `SEMI`, and `ANTI` joins are present in other systems, but have high-performance implementations in DuckDB. DuckDB also adds a new `POSITIONAL` join that combines by the row numbers in each table to match the commonly used Pandas capability of joining on row number indexes. See the `JOIN` documentation for details, and look out for a blog post describing DuckDB's state of the art `ASOF` joins!

Summary and Future Work

DuckDB aims to be the easiest database to use. Fundamental architectural decisions to be in-process, have zero dependencies, and have strong typing contribute to this goal, but the friendliness of its SQL dialect has a strong impact as well. By extending the industry-standard PostgreSQL dialect, DuckDB aims to provide the simplest way to express the data transformations you need. These changes range from altering the ancient clause order of the `SELECT` statement to begin with `FROM`, allowing a fundamentally new way to use functions with chaining, to advanced nested data type calculations like list comprehensions. Each of these features are available in the 0.8.1 release.

Future work for friendlier SQL includes:

- Lambda functions with more than 1 argument, like `list_zip`
- Underscores as digit separators (Ex: `1_000_000` instead of `1000000`)
- Extension user experience, including autoloading
- Improvements to file globbing
- Your suggestions!

Please let us know what areas of SQL can be improved! We welcome your feedback on Discord or GitHub.

Live long and prosper! 🖖

DuckDB's AsOf Joins: Fuzzy Temporal Lookups

Publication date: 2023-09-15

Author: Richard Wesley

TL;DR: DuckDB supports AsOf Joins – a way to match nearby values. They are especially useful for searching event tables for temporal analytics.

Do you have time series data that you want to join, but the timestamps don't quite match? Or do you want to look up a value that changes over time using the times in another table? And did you end up writing convoluted (and slow) inequality joins to get your results? Then this post is for you!

What Is an AsOf Join?

Time series data is not always perfectly aligned. Clocks may be slightly off, or there may be a delay between cause and effect. This can make connecting two sets of ordered data challenging. AsOf Joins are a tool for solving this and other similar problems.

One of the problems that AsOf Joins are used to solve is finding the value of a varying property at a specific point in time. This use case is so common that it is where the name came from: *Give me the value of the property **as of this time.***

More generally, however, AsOf joins embody some common temporal analytic semantics, which can be cumbersome and slow to implement in standard SQL.

Portfolio Example

Let's start with a concrete example. Suppose we have a table of stock `prices` with timestamps:

ticker	when	price
APPL	2001-01-01 00:00:00	1
APPL	2001-01-01 00:01:00	2
APPL	2001-01-01 00:02:00	3
MSFT	2001-01-01 00:00:00	1
MSFT	2001-01-01 00:01:00	2
MSFT	2001-01-01 00:02:00	3
GOOG	2001-01-01 00:00:00	1
GOOG	2001-01-01 00:01:00	2
GOOG	2001-01-01 00:02:00	3

We have another table containing portfolio `holdings` at various points in time:

ticker	when	shares
APPL	2000-12-31 23:59:30	5.16
APPL	2001-01-01 00:00:30	2.94
APPL	2001-01-01 00:01:30	24.13
GOOG	2000-12-31 23:59:30	9.33
GOOG	2001-01-01 00:00:30	23.45
GOOG	2001-01-01 00:01:30	10.58
DATA	2000-12-31 23:59:30	6.65
DATA	2001-01-01 00:00:30	17.95
DATA	2001-01-01 00:01:30	18.37

We can compute the value of each holding at that point in time by finding the most recent price before the holding's timestamp by using an AsOf Join:

```
SELECT h.ticker, h.when, price * shares AS value
FROM holdings h ASOF JOIN prices p
  ON h.ticker = p.ticker
 AND h.when >= p.when;
```

This attaches the value of the holding at that time to each row:

ticker	when	value
APPL	2001-01-01 00:00:30	2.94
APPL	2001-01-01 00:01:30	48.26
GOOG	2001-01-01 00:00:30	23.45
GOOG	2001-01-01 00:01:30	21.16

It essentially executes a function defined by looking up nearby values in the `prices` table. Note also that missing `ticker` values do not have a match and don't appear in the output.

Outer AsOf Joins

Because AsOf produces at most one match from the right hand side, the left side table will not grow as a result of the join, but it could shrink if there are missing times on the right. To handle this situation, you can use an *outer* AsOf Join:

```
SELECT h.ticker, h.when, price * shares AS value
FROM holdings h ASOF LEFT JOIN prices p
  ON h.ticker = p.ticker
 AND h.when >= p.when
ORDER BY ALL;
```

As you might expect, this will produce `NULL` prices and values instead of dropping left side rows when there is no ticker or the time is before the prices begin.

ticker	when	value
APPL	2000-12-31 23:59:30	

ticker	when	value
APPL	2001-01-01 00:00:30	2.94
APPL	2001-01-01 00:01:30	48.26
GOOG	2000-12-31 23:59:30	
GOOG	2001-01-01 00:00:30	23.45
GOOG	2001-01-01 00:01:30	21.16
DATA	2000-12-31 23:59:30	
DATA	2001-01-01 00:00:30	
DATA	2001-01-01 00:01:30	

Windowing Alternative

Standard SQL can implement this kind of join, but you need to use a window function and an inequality join. These can both be fairly expensive operations, but the query would look like this:

```
WITH state AS (
    SELECT
        ticker,
        price,
        "when",
        lead("when", 1, 'infinity')
            OVER (PARTITION BY ticker ORDER BY "when") AS end
    FROM prices
)
SELECT h.ticker, h.when, price * shares AS value
FROM holdings h
INNER JOIN state s
        ON h.ticker = s.ticker
       AND h.when >= s.when
       AND h.when < s.end;
```

The default value of `infinity` is used to make sure there is an end value for the last row that can be compared. Here is what the `state` CTE looks like for our example:

ticker	price	when	end
APPL	1	2001-01-01 00:00:00	2001-01-01 00:01:00
APPL	2	2001-01-01 00:01:00	2001-01-01 00:02:00
APPL	3	2001-01-01 00:02:00	infinity
GOOG	1	2001-01-01 00:00:00	2001-01-01 00:01:00
GOOG	2	2001-01-01 00:01:00	2001-01-01 00:02:00
GOOG	3	2001-01-01 00:02:00	infinity
MSFT	1	2001-01-01 00:00:00	2001-01-01 00:01:00
MSFT	2	2001-01-01 00:01:00	2001-01-01 00:02:00
MSFT	3	2001-01-01 00:02:00	infinity

In the case where there is no equality condition, the planner would have to use an inequality join, which can be very expensive. And even in the equality condition case, the resulting hash join may end up with long chains of identical `ticker` keys that will all match and need pruning.

Why AsOf?

If SQL can compute AsOf joins already, why do we need a new join type? There are two big reasons: expressibility and performance. The windowing alternative is more verbose and harder to understand than the AsOf syntax, so making it easier to say what you are doing helps others (or even you!) understand what is happening.

The syntax also makes it easier for DuckDB to understand what you want and produce your results faster. The window and inequality join version loses the valuable information that the intervals do not overlap. It also prevents the query optimizer from moving the join because SQL insists that windowing happens *after* joins. By treating the operation *as a join* with *known data constraints*, DuckDB can move the join for performance and use a tailored join algorithm. The algorithm we use is to sort the right side table and then do a kind of merge join with the left side values. But unlike a standard merge join, AsOf can stop searching when it finds the first match because there is at most one match.

State Tables

You may be wondering why the Common Table Expression in the `WITH` clause was called *state*. This is because the `prices` table is really an example of what in temporal analytics is called an *event table* . The rows of an event table contain timestamps and what happened at that time (i.e., events). The events in the `prices` table are changes to the price of a stock. Another common example of an event table is a structured log file: Each row of the log records when something "happened" – usually a change to a part of the system.

Event tables are difficult to work with because each fact only has the start time. In order to know whether the fact is still true (or true at a specific time) you need the end time as well. A table with both the start and end time is called a *state table*. Converting event tables to state tables is a common temporal data preparation task, and the windowing CTE above shows how to do it in general using SQL.

Sentinel Values

One limitation of the windowing approach is that the ordering type needs to have sentinel value that can be used if it does not support `infinity`, either an unused value or `NULL`.

Both of these choices are potentially problematic. In the first case, it may not be easy to determine an upper sentinel value (suppose the ordering was a string column?) In the second case, you would need to write the condition as `h.when < s.end OR s.end IS NULL` and using an `OR` like this in a join condition makes comparisons slow and hard to optimize. Moreover, if the ordering column is already using `NULL` to indicate missing values, this option is not available.

For most state tables, there are suitable choices (e.g., large dates) but one of the advantages of AsOf is that it can avoid having to design a state table if it is not needed for the analytic task.

Event Table Variants

So far we have been using a standard type of event table where the timestamps are assumed to be the start of the state transitions. But AsOf can now use any inequality, which allows it to handle other types of event tables.

To explore this, let's use two very simple tables with no equality conditions. The build side will just have four integer "timestamps" with alphabetic values:

Time	Value
1	a
2	b
3	c
4	d

The probe table will just be the time values plus the midpoints, and we can make a table showing what value each probe time matches for greater than or equal to:

Probe	>=
0.5	
1.0	a
1.5	a
2.0	b
2.5	b
3.0	c
3.5	c
4.0	d
4.5	d

This shows us that the interval a probe value matches is in the half-open interval `[Tn, Tn+1)`.

Now let's see what happens if use strictly greater than as the inequality:

Probe	>
0.5	
1.0	
1.5	a
2.0	a
2.5	b
3.0	b
3.5	c
4.0	c
4.5	d

Now we can see that the interval a probe value matches is in the half-open interval `(Tn, Tn+1]`. The only difference is that the interval is closed at the end instead of the beginning. This means that for this inequality type, the time is not part of the interval.

What if the inequality goes in the other direction, say less than or equal to?

Probe	<=
0.5	a
1.0	a
1.5	b
2.0	b
2.5	c
3.0	c
3.5	d
4.0	d

Probe	<=
4.5	

Again, we have half-open intervals, but this time we are matching the *previous* interval `(Tn-1, Tn]`. One way to interpret this is that the times in the build table are the *end* of the interval, instead of the beginning. Also, unlike greater than or equal to, the interval is closed at the end instead of the beginning. Adding this to what we found for strictly greater than, we can interpret this as meaning that the lookup times are part of the interval when non-strict inequalities are used.

We can check this by looking at the last inequality: strictly less than:

Probe	<
0.5	a
1.0	b
1.5	b
2.0	c
2.5	c
3.0	d
3.5	d
4.0	
4.5	

In this case the matching intervals are `[Tn-1, Tn)`. This is a strict inequality, so the table time is not in the interval, and it is a less than, so the time is the end of the interval.

To sum up, here is the full list:

Inequality	Interval
>	(Tn, Tn+1]
>=	[Tn, Tn+1)
<=	(Tn-1, Tn]
<	[Tn-1, Tn)

We now have two natural interpretations of what the inequalities mean:

- The greater (resp. less) than inequalities mean the time is the beginning (resp. end) of the interval.
- The strict (resp. non-strict) inequalities mean the time is excluded from (resp. included in) the interval.

So if we know whether the time marks the start or the end of the event, and whether the time is include or excluded, we can choose the appropriate AsOf inequality.

Usage

So far we have been explicit about specifying the conditions for AsOf, but SQL also has a simplified join condition syntax for the common case where the column names are the same in both tables. This syntax uses the `USING` keyword to list the fields that should be compared for equality. AsOf also supports this syntax, but with two restrictions:

- The last field is the inequality
- The inequality is >= (the most common case)

Our first query can then be written as:

```
SELECT ticker, h.when, price * shares AS value
FROM holdings h
ASOF JOIN prices p
    USING(ticker, "when");
```

Be aware that if you don't explicitly list the columns in the SELECT, the ordering field value will be the probe value, not the build value. For a natural join, this is not an issue because all the conditions are equalities, but for AsOf, one side has to be chosen. Since AsOf can be viewed as a lookup function, it is more natural to return the "function arguments" than the function internals.

Under the Hood

What an AsOf Join is really doing is allowing you to treat an event table as a state table for join operations. By knowing the semantics of the join, it can avoid creating a full state table and be more efficient than a general inequality join.

Let's start by looking at how the windowing version works. Remember that we used this query to convert the event table to a state table:

```
WITH state AS (
    SELECT
        ticker,
        price,
        "when",
        lead("when", 1, 'infinity')
            OVER (PARTITION BY ticker ORDER BY "when") AS end
    FROM prices
);
```

The state table CTE is created by hash partitioning the table on `ticker`, sorting on when and then computing another column that is just when shifted down by one. The join is then implemented with a hash join on `ticker` and two comparisons on when.

If there was no `ticker` column (e.g., the prices were for a single item) then the join would be implemented using our inequality join operator, which would materialise and sort both sides because it doesn't know that the ranges are disjoint.

The AsOf operator uses all three operator pipeline APIs to consolidate and collect rows. During the `sink` phase, AsOf hash partitions and sorts the right hand side to make a temporary state table. (In fact it uses the same code as Window, but without unnecessarily materialising the end column.) During the `operator` phase, it filters out (or returns) rows that cannot match because of `NULL` values in the predicate expressions, and then hash partitions and sorts the remaining rows into a cache. Finally, during the `source` phase, it matches hash partitions and then merge joins the sorted values within each hash partition.

Benchmarks

Because AsOf joins can be implemented in various ways using standard SQL queries, benchmarking is really about comparing the various alternatives.

One alternative is a debugging PRAGMA for AsOf called `debug_asof_iejoin`, which implements the join using Window and IEJoin. This allows us to easily toggle between the implementations and compare runtimes.

Other alternatives combine equi-joins and window functions. The equi-join is used to implement the equality matching conditions, and the window is used to select the closest inequality. We will now look at two different windowing techniques and compare their performance. If you wish to skip this section, the bottom line is that while they are sometimes a bit faster, the AsOf join has the most consistent behavior of all the algorithms.

Window as State Table

The first benchmark compares a hash join with a state table. It probes a 5M row table of values built from 100K timestamps and 50 partitioning keys using a self-join where only 50% of the keys are present and the timestamps have been shifted to be halfway between the originals:

```
CREATE OR REPLACE TABLE build AS (
    SELECT k, '2001-01-01 00:00:00'::TIMESTAMP + INTERVAL (v) MINUTE AS t, v
    FROM range(0, 100_000) vals(v), range(0, 50) keys(k)
);

CREATE OR REPLACE TABLE probe AS (
    SELECT k * 2 AS k, t - INTERVAL (30) SECOND AS t
    FROM build
);
```

The `build` table looks like this:

k	t	v
0	2001-01-01 00:00:00	0
0	2001-01-01 00:01:00	1
0	2001-01-01 00:02:00	2
0	2001-01-01 00:03:00	3
…	…	…

and the probe table looks like this (with only even values for k):

k	t
0	2000-12-31 23:59:30
0	2001-01-01 00:00:30
0	2001-01-01 00:01:30
0	2001-01-01 00:02:30
0	2001-01-01 00:03:30
…	…

The benchmark just does the join and sums up the v column:

```
SELECT sum(v)
FROM probe
ASOF JOIN build USING(k, t);
```

The debugging PRAGMA does not allow us to use a hash join, but we can create the state table in a CTE again and use an inner join:

```
-- Hash Join implementation
WITH state AS (
    SELECT k,
      t AS begin,
      v,
      lead(t, 1, 'infinity'::TIMESTAMP) OVER (PARTITION BY k ORDER BY t) AS end
    FROM build
)
```

```
SELECT sum(v)
FROM probe p
INNER JOIN state s
        ON p.t >= s.begin
       AND p.t < s.end
       AND p.k = s.k;
```

This works because the planner assumes that equality conditions are more selective than inequalities and generates a hash join with a filter.

Running the benchmark, we get results like this:

Algorithm	Median of 5
AsOf	0.425 s
IEJoin	3.522 s
State Join	192.460 s

The runtime improvement of AsOf over IEJoin here is about 9×. The horrible performance of the Hash Join is caused by the long (100K) bucket chains in the hash table.

The second benchmark tests the case where the probe side is about 10× smaller than the build side:

```
CREATE OR REPLACE TABLE probe AS
    SELECT k,
      '2021-01-01T00:00:00'::TIMESTAMP +
          INTERVAL (random() * 60 * 60 * 24 * 365) SECOND AS t,
    FROM range(0, 100_000) tbl(k);

CREATE OR REPLACE TABLE build AS
    SELECT r % 100_000 AS k,
      '2021-01-01T00:00:00'::TIMESTAMP +
          INTERVAL (random() * 60 * 60 * 24 * 365) SECOND AS t,
      (random() * 100_000)::INTEGER AS v
    FROM range(0, 1_000_000) tbl(r);

SELECT sum(v)
FROM probe p
ASOF JOIN build b
  ON p.k = b.k
 AND p.t >= b.t

-- Hash Join Version
WITH state AS (
  SELECT k,
    t AS begin,
    v,
    lead(t, 1, 'infinity'::TIMESTAMP)
        OVER (PARTITION BY k ORDER BY t) AS end
  FROM build
)
SELECT sum(v)
FROM probe p
INNER JOIN state s
        ON p.t >= s.begin
       AND p.t < s.end
       AND p.k = s.k;
```

Algorithm	Median of 5 runs
State Join	0.065 s
AsOf	0.077 s
IEJoin	49.508 s

Now the runtime improvement of AsOf over IEJoin is huge (~500×) because it can leverage the partitioning to eliminate almost all of the equality mismatches.

The Hash Join implementation does much better here because the optimizer notices that the probe side is smaller and builds the hash table on the ”probe” table. Also, the probe values here are unique, so the hash table chains are minimal.

Window with Ranking

Another way to use the window operator is to:

- Join the tables on the equality predicates
- Filter to pairs where the build time is before the probe time
- Partition the result on both the equality keys *and* the probe timestamp
- Sort the partitions on the build timestamp *descending*
- Filter out all value except rank 1 (i.e., the largest build time <= the probe time)

The query looks like:

```
WITH win AS (
    SELECT p.k, p.t, v,
        rank() OVER (PARTITION BY p.k, p.t ORDER BY b.t DESC) AS r
    FROM probe p INNER JOIN build b
      ON p.k = b.k
    AND p.t >= b.t
    QUALIFY r = 1
)
SELECT k, t, v
FROM win;
```

The advantage of this windowing query is that it does not require sentinel values, so it can work with any data type. The disadvantage is that it creates many more partitions because it includes both timestamps, which requires more complex sorting. Moreover, because it applies the window *after* the join, it can produce huge intermediates that can result in external sorting and expensive out-of-memory operations.

For this benchmark, we will be using three build tables, and two probe tables, all containing 10K integer equality keys. The probe tables have either 1 or 15 timestamps per key:

```
CREATE OR REPLACE TABLE probe15 AS
    SELECT k, t
    FROM range(10_000) cs(k),
         range('2022-01-01'::TIMESTAMP, '2023-01-01'::TIMESTAMP, INTERVAL 26 DAY) ts(t);

CREATE OR REPLACE TABLE probe1 AS
    SELECT k, '2022-01-01'::TIMESTAMP t
    FROM range(10_000) cs(k);
```

The build tables are much larger and have approximately 10/100/1000× the number of entries as the 15 element tables:

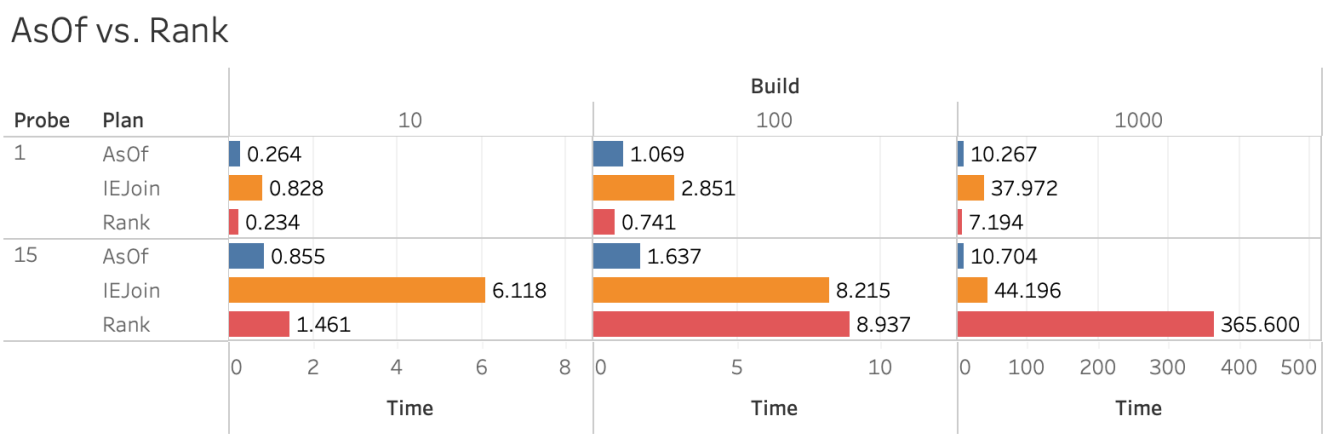
```
-- 10:1
CREATE OR REPLACE TABLE build10 AS
    SELECT k, t, (random() * 1000)::DECIMAL(7, 2) AS v
```

```
    FROM range(10_000) ks(k),
         range('2022-01-01'::TIMESTAMP, '2023-01-01'::TIMESTAMP, INTERVAL 59 HOUR) ts(t);

-- 100:1
CREATE OR REPLACE TABLE build100 AS
    SELECT k, t, (random() * 1000)::DECIMAL(7, 2) AS v
    FROM range(10_000) ks(k),
         range('2022-01-01'::TIMESTAMP, '2023-01-01'::TIMESTAMP, INTERVAL 350 MINUTE) ts(t);

-- 1000:1
CREATE OR REPLACE TABLE build1000 AS
    SELECT k, t, (random() * 1000)::DECIMAL(7, 2) AS v
    FROM range(10_000) ks(k),
         range('2022-01-01'::TIMESTAMP, '2023-01-01'::TIMESTAMP, INTERVAL 35 MINUTE) ts(t);
```

The AsOf join queries are:

```
-- AsOf/IEJoin
SELECT p.k, p.t, v
FROM probe p ASOF JOIN build b
  ON p.k = b.k
 AND p.t >= b.t
ORDER BY 1, 2;

-- Rank
WITH win AS (
    SELECT p.k, p.t, v,
        rank() OVER (PARTITION BY p.k, p.t ORDER BY b.t DESC)  AS r
    FROM probe p INNER JOIN build b
      ON p.k = b.k
    AND p.t >= b.t
    QUALIFY r = 1
)
SELECT k, t, v
FROM win
ORDER BY 1, 2;
```

The results are shown here:



(Median of 5 except for Rank/15/1000).

- For all ratios with 15 probes, AsOf is the most performant.
- For small ratios with 15 probes, Rank beats IEJoin (both with windowing), but by 100:1 it is starting to explode.
- For single element probes, Rank is most effective, but even there, its edge over AsOf is only about 50% at scale.

This shows that AsOf could be possibly be improved upon, but predicting where that happens would be tricky, and getting it wrong would have enormous costs.

Future Work

DuckDB can now execute AsOf joins for all inequality types with reasonable performance. In some cases, the performance gain is several orders of magnitude over the standard SQL versions – even with our fast inequality join operator.

While the current AsOf operator is completely general, there are a couple of planning optimisations that could be applied here.

- When there are selective equality conditions, it is likely that a hash join with filtering against a materialised state table would be significantly faster. If we can detect this and suitable sentinel values are available, the planner could choose to use a hash join instead of the default AsOf implementation.
- There are also use cases where the probe table is much smaller than the build table, along with equality conditions, and performing a hash join against the *probe* table could yield significant performance improvements.

Nevertheless, remember that one of the advantages of SQL is that it is a declarative language:
You specify *what* you want and leave it up to the database to figure out *how*. Now that we have defined the semantics of the AsOf join, you the user can write queries saying this is *what* you want – and we are free to keep improving the *how*!

Happy Joining!

One of the most interesting parts of working on DuckDB is that it stretches the traditional SQL model of unordered data. DuckDB makes it easy to query *ordered* data sets such as data frames and parquet files, and when you have data like that, you expect to be able to do ordered analysis! Implementing Fast Sorting, Fast Windowing and Fast AsOf joins is how we are making this expectation a reality.

Announcing DuckDB 0.9.0

Publication date: 2023-09-26

Authors: Mark Raasveldt and Hannes Mühleisen

The DuckDB team is happy to announce the latest DuckDB release (0.9.0). This release is named Undulata after the Yellow-billed duck native to Africa.

To install the new version, please visit the installation guide. The full release notes can be found on GitHub.

What's New in 0.9.0

There have been too many changes to discuss them each in detail, but we would like to highlight several particularly exciting features!

- Out-of-Core Hash Aggregate
- Storage Improvements
- Index Improvements

- DuckDB-Wasm Extensions
- Extension Auto-Loading
- Improved AWS Support
- Iceberg Support
- Azure Support
- PySpark-Compatible API

Below is a summary of those new features with examples, starting with a change in our SQL dialect that is designed to produce more intuitive results by default.

Breaking SQL Changes

Struct Auto-Casting. Previously the names of struct entries were ignored when determining auto-casting rules. As a result, struct field names could be silently renamed. Starting with this release, this will result in an error instead.

```
CREATE TABLE structs (s STRUCT(i INTEGER));
INSERT INTO structs VALUES ({'k': 42});
```

```
Mismatch Type Error: Type STRUCT(k INTEGER) does not match with STRUCT(i INTEGER). Cannot cast STRUCTs
with different names
```

Unnamed structs constructed using the `ROW` function can still be inserted into struct fields.

```
INSERT INTO structs VALUES (ROW(42));
```

Core System Improvements

Out-of-Core Hash Aggregates and **Hash Aggregate Performance Improvements.** When working with large data sets, memory management is always a potential pain point. By using a streaming execution engine and buffer manager, DuckDB supports many operations on larger than memory data sets. DuckDB also aims to support queries where *intermediate* results do not fit into memory by using disk-spilling techniques.

In this release, support for disk-spilling techniques is further extended through the support for out-of-core hash aggregates. Now, hash tables constructed during `GROUP BY` queries or `DISTINCT` operations that do not fit in memory due to a large number of unique groups will spill data to disk instead of throwing an out-of-memory exception. Due to the clever use of radix partitioning, performance degradation is gradual, and performance cliffs are avoided. Only the subset of the table that does not fit into memory will be spilled to disk.

The performance of our hash aggregate has also improved in general, especially when there are many groups. For example, we compute the number of unique rows in a data set with 30 million rows and 15 columns by using the following query:

```
SELECT count(*) FROM (SELECT DISTINCT * FROM tbl);
```

If we keep all the data in memory, the query should use around 6 GB. However, we can still complete the query if less memory is available. In the table below, we can see how the runtime is affected by lowering the memory limit:

memory limit	v0.8.1	v0.9.0
10 GB	8.52 s	2.91 s
9 GB	8.52 s	3.45 s
8 GB	8.52 s	3.45 s
7 GB	8.52 s	3.47 s
6 GB	OOM	3.41 s
5 GB	OOM	3.67 s

memory limit	v0.8.1	v0.9.0
4 GB	OOM	3.87 s
3 GB	OOM	4.20 s
2 GB	OOM	4.39 s
1 GB	OOM	4.91 s

Compressed Materialization. DuckDB's streaming execution engine has a low memory footprint, but more memory is required for operations such as grouped aggregation. The memory footprint of these operations can be reduced by compression. DuckDB already uses many compression techniques in its storage format, but many of these techniques are too costly to use during query execution. However, certain lightweight compression techniques are so cheap that the benefit of the reducing memory footprint outweight the cost of (de)compression.

In this release, we add support for compression of strings and integer types right before data goes into the grouped aggregation and sorting operators. By using statistics, both types are compressed to the smallest possible integer type. For example, if we have the following table:

id int32	name varchar
300	alice
301	bob
302	eve
303	mallory
304	trent

The `id` column uses a 32-bit integer. From our statistics we know that the minimum value is 300, and the maximum value is 304. We can subtract 300 and cast to an 8-bit integer instead, reducing the width from 4 bytes down to 1.

The `name` column uses our internal string type, which is 16 bytes wide. However, our statistics tell us that the longest string here is only 7 bytes. We can fit this into a 64-bit integer like so:

```
alice   -> alice005
bob     -> bob00003
eve     -> eve00003
mallory -> mallory7
trent   -> trent005
```

This reduces the width from 16 bytes down to 8. To support sorting of compressed strings, we flip the bytes on big-endian machines so that our comparison operators are still correct:

```
alice005 -> 500ecila
bob00003 -> 30000bob
eve00003 -> 30000eve
mallory7 -> 7yrollam
trent005 -> 500tnert
```

By reducing the size of query intermediates, we can prevent/reduce spilling data to disk, reducing the need for costly I/O operations, thereby improving query performance.

Window Function Performance Improvements (#7831, #7996, #8050, #8491). This release features many improvements to the performance of Window functions due to improved vectorization of the code, more re-use of partial aggregates and improved parallelism through work stealing of tasks. As a result, performance of Window functions has improved significantly, particularly in scenarios where there are no or few partitions.

```
SELECT
    sum(driver_pay) OVER (
        ORDER BY dropoff_datetime ASC
        RANGE BETWEEN
        INTERVAL 3 DAYS PRECEDING AND
        INTERVAL 0 DAYS FOLLOWING
    )
FROM tripdata;
```

Version	Run time
v0.8.0	33.8 s
v0.9.0	3.8 s

Storage Improvements

Vacuuming of Deleted Row Groups. Starting with this release, when deleting data using `DELETE` statements, entire row groups that are deleted will be automatically cleaned up. Support is also added to truncate the database file on checkpoint which allows the database file to be reduced in size after data is deleted. Note that this only occurs if the deleted row groups are located at the end of the file. The system does not yet move around data in order to reduce the size of the file on disk. Instead, free blocks earlier on in the file are re-used to store later data.

Index Storage Improvements (#7930, #8112, #8437, #8703). Many improvements have been made to both the in-memory footprint, and the on-disk footprint of ART indexes. In particular for indexes created to maintain `PRIMARY KEY`, `UNIQUE` or `FOREIGN KEY` constraints the storage and in-memory footprint is drastically reduced.

```
CREATE TABLE integers (i INTEGER PRIMARY KEY);
INSERT INTO integers FROM range(10000000);
```

Version	Size
v0.8.0	278 MB
v0.9.0	78 MB

In addition, due to improvements in the manner in which indexes are stored on disk they can now be written to disk incrementally instead of always requiring a full rewrite. This allows for much quicker checkpointing for tables that have indexes.

Extensions

Extension Auto-Loading. Starting from this release, DuckDB supports automatically installing and loading of trusted extensions. As many workflows rely on core extensions that are not bundled, such as `httpfs`, many users found themselves having to remember to load the required extensions up front. With this change, the extensions will instead be automatically loaded (and optionally installed) when used in a query.

For example, in Python the following code snippet now works without needing to explicitly load the `httpfs` or `json` extensions.

```
import duckdb

duckdb.sql("FROM 'https://raw.githubusercontent.com/duckdb/duckdb/main/data/json/example_n.ndjson'")
```

The set of autoloadable extensions is limited to official extensions distributed by DuckDB Labs, and can be found here. The behavior can also be disabled using the `autoinstall_known_extensions` and `autoload_known_extensions` settings, or through the more general `enable_external_access` setting. See the configuration options.

DuckDB-Wasm Extensions. This release adds support for loadable extensions to DuckDB-Wasm. Previously, any extensions that you wanted to use with the Wasm client had to be baked in. With this release, extensions can be loaded dynamically instead. When an extension is loaded, the Wasm bundle is downloaded and the functionality of the extension is enabled. Give it a try in our Wasm shell.

```
LOAD inet;
SELECT '127.0.0.1'::INET;
```

AWS Extension. This release marks the launch of the DuckDB AWS extension. This extension contains AWS related features that rely on the AWS SDK. Currently, the extension contains one function, `LOAD_AWS_CREDENTIALS`, which uses the AWS Credential Provider Chain to automatically fetch and set credentials:

```
CALL load_aws_credentials();
SELECT * FROM 's3://some-bucket/that/requires/authentication.parquet';
```

See the documentation for more information.

Experimental Iceberg Extension. This release marks the launch of the DuckDB Iceberg extension. This extension adds support for reading tables stored in the Iceberg format.

```
SELECT count(*)
FROM iceberg_scan('data/iceberg/lineitem_iceberg', allow_moved_paths = true);
```

See the documentation for more information.

Experimental Azure Extension. This release marks the launch of the DuckDB Azure extension. This extension allows for DuckDB to natively read data stored on Azure, in a similar manner to how it can read data stored on S3.

```
SET azure_storage_connection_string = '<your_connection_string>';
SELECT * FROM 'azure://<my_container>/*.csv';
```

See the documentation for more information.

Clients

Experimental PySpark API. This release features the addition of an experimental Spark API to the Python client. The API aims to be fully compatible with the PySpark API, allowing you to use the Spark API as you are familiar with but while utilizing the power of DuckDB. All statements are translated to DuckDB's internal plans using our relational API and executed using DuckDB's query engine.

```
from duckdb.experimental.spark.sql import SparkSession as session
from duckdb.experimental.spark.sql.functions import lit, col
import pandas as pd

spark = session.builder.getOrCreate()

pandas_df = pd.DataFrame({
    'age': [34, 45, 23, 56],
    'name': ['Joan', 'Peter', 'John', 'Bob']
})

df = spark.createDataFrame(pandas_df)
df = df.withColumn(
    'location', lit('Seattle')
)
res = df.select(
    col('age'),
    col('location')
```

```
).collect()

print(res)
#[
#    Row(age=34, location='Seattle'),
#    Row(age=45, location='Seattle'),
#    Row(age=23, location='Seattle'),
#    Row(age=56, location='Seattle')
#]
```

Note that the API is currently experimental and features are still missing. We are very interested in feedback. Please report any functionality that you are missing, either through Discord or on GitHub.

Final Thoughts

The full release notes can be found on GitHub. We would like to thank all of the contributors for their hard work on improving DuckDB.

DuckDB's CSV Sniffer: Automatic Detection of Types and Dialects

Publication date: 2023-10-27

Author: Pedro Holanda

TL;DR: DuckDB is primarily focused on performance, leveraging the capabilities of modern file formats. At the same time, we also pay attention to flexible, non-performance-driven formats like CSV files. To create a nice and pleasant experience when reading from CSV files, DuckDB implements a CSV sniffer that automatically detects CSV dialect options, column types, and even skips dirty data. The sniffing process allows users to efficiently explore CSV files without needing to provide any input about the file format.

There are many different file formats that users can choose from when storing their data. For example, there are performance-oriented binary formats like Parquet, where data is stored in a columnar format, partitioned into row groups, and heavily compressed. However, Parquet is known for its rigidity, requiring specialized systems to read and write these files.

On the other side of the spectrum, there are files with the CSV (comma-separated values) format, which I like to refer to as the 'Woodstock of data'. CSV files offer the advantage of flexibility; they are structured as text files, allowing users to manipulate them with any text editor, and nearly any data system can read and execute queries on them.

However, this flexibility comes at a cost. Reading a CSV file is not a trivial task, as users need a significant amount of prior knowledge about the file. For instance, DuckDB's CSV reader offers more than 25 configuration options. I've found that people tend to think I'm not working hard enough if I don't introduce at least three new options with each release. *Just kidding.* These options include specifying the delimiter, quote and escape characters, determining the number of columns in the CSV file, and identifying whether a header is present while also defining column types. This can slow down an interactive data exploration process, and make analyzing new datasets a cumbersome and less enjoyable task.

One of the raison d'être of DuckDB is to be pleasant and easy to use, so we don't want our users to have to fiddle with CSV files and input options manually. Manual input should be reserved only for files with rather unusual choices for their CSV dialect (where a dialect comprises the combination of the delimiter, quote, escape, and newline values used to create that file) or for specifying column types.

Automatically detecting CSV options can be a daunting process. Not only are there many options to investigate, but their combinations can easily lead to a search space explosion. This is especially the case for CSV files that are not well-structured. Some might argue that CSV files have a specification, but the truth of the matter is that the ”specification” changes as soon as a single system is capable of reading a flawed file. And, oh boy, I've encountered my fair share of semi-broken CSV files that people wanted DuckDB to read in the past few months.

DuckDB implements a multi-hypothesis CSV sniffer that automatically detects dialects, headers, date/time formats, column types, and identifies dirty rows to be skipped. Our ultimate goal is to automatically read anything resembling a CSV file, to never give up and never let you down! All of this is achieved without incurring a substantial initial cost when reading CSV files. In the bleeding edge version, the sniffer runs when reading a CSV file by default. Note that the sniffer will always prioritize any options set by the user (e.g., if the user sets `,` as the delimiter, the sniffer won't try any other options and will assume that the user input is correct).

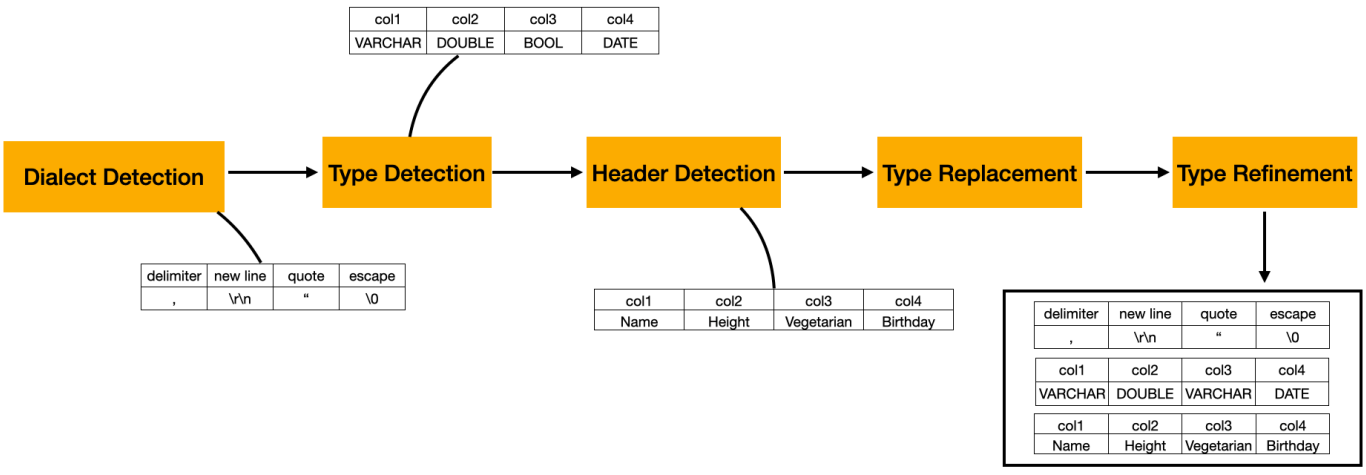
In this blog post, I will explain how the current implementation works, discuss its performance, and provide insights into what comes next!

DuckDB's Automatic Detection

The process of parsing CSV files is depicted in the figure below. It currently consists of five different phases, which will be detailed in the next sections.

The CSV file used in the overview example is as follows:

```
Name, Height, Vegetarian, Birthday
"Pedro", 1.73, False, 30-07-92
... imagine 2048 consistent rows ...
"Mark", 1.72, N/A, 20-09-92
```



In the first phase, we perform *Dialect Detection*, where we select the dialect candidates that generate the most per-row columns in the CSV file while maintaining consistency (i.e., not exhibiting significant variations in the number of columns throughout the file). In our example, we can observe that, after this phase, the sniffer successfully detects the necessary options for the delimiter, quotes, escapes, and new line delimiters.

The second phase, referred to as *Type Detection*, involves identifying the data types for each column in our CSV file. In our example, our sniffer recognizes four column types: `VARCHAR`, `DOUBLE`, `BOOL`, and `DATE`.

The third step, known as *Header Detection*, is employed to ascertain whether our file includes a header. If a header is present, we use it to set the column names; otherwise, we generate them automatically. In our example, there is a header, and each column gets its name defined in there.

Now that our columns have names, we move on to the fourth, optional phase: *Type Replacement*. DuckDB's CSV reader provides users with the option to specify column types by name. If these types are specified, we replace the detected types with the user's specifications.

Finally, we progress to our last phase, *Type Refinement*. In this phase, we analyze additional sections of the file to validate the accuracy of the types determined during the initial type detection phase. If necessary, we refine them. In our example, we can see that the `Vegetarian` column was initially categorized as `BOOL`. However, upon further examination, it was found to contain the string `N/A`, leading to an upgrade of the column type to `VARCHAR` to accommodate all possible values.

The automatic detection is only executed on a sequential sample of the CSV file. By default, the size of the sample is 20,480 tuples (i.e., 10 DuckDB execution chunks). This can be configured via the `sample_size` option, and can be set to `-1` in case the user wants to sniff the complete file. Since the same data is repeatedly read with various options, and users can scan the entire file, all CSV buffers generated during sniffing are cached and efficiently managed to ensure high performance.

Of course, running the CSV Sniffer on very large files will have a drastic impact on the overall performance (see our benchmark section below). In these cases, the sample size should be kept at a reasonable level.

In the next subsections, I will describe each phase in detail.

Dialect Detection

In the *Dialect Detection*, we identify the delimiter, quotes, escapes, and new line delimiters of a CSV file.

Our delimiter search space consists of the following delimiters: `,`, `|`, `;`, `\t`. If the file has a delimiter outside the search space, it must be provided by the user (e.g., `delim='?'`). Our quote search space is `"`, `'` and `\0`, where `\0` is a string terminator indicating no quote is present; again, users can provide custom characters outside the search space (e.g., `quote='?'`). The search space of escape values depends on the value of the quote option, but in summary, they are the same as quotes with the addition of `\`, and again, they can also be provided by the user (`escape='?'`). Finally, the last detected option is the new line delimiters; they can be `\r`, `\n`, `\r\n`, and a mix of everything (trust me, I've seen a real-world CSV file that used a mix).

By default, the dialect detection runs on 24 different combinations of dialect configurations. To determine the most promising configuration, we calculate the number of columns each CSV tuple would produce under each of these configurations. The one that results in the most columns with the most consistent rows will be chosen.

The calculation of consistent rows depends on additional user-defined options. For example, the `null_padding` option will pad missing columns with NULL values. Therefore, rows with missing columns will have the missing columns padded with `NULL`.

If `null_padding` is set to true, CSV files with inconsistent rows will still be considered, but a preference will be given to configurations that minimize the occurrence of padded rows. If `null_padding` is set to false, the dialect detector will skip inconsistent rows at the beginning of the CSV file. As an example, consider the following CSV file.

```
I like my csv files to have notes to make dialect detection harder
I also like commas like this one : ,
A,B,C
1,2,3
4,5,6
```

Here the sniffer would detect that with the delimiter set to `,` the first row has one column, the second has two, but the remaining rows have 3 columns. Hence, if `null_padding` is set to false, it would still select `,` as a delimiter candidate, by assuming the top rows are dirty notes. (Believe me, CSV notes are a thing!). Resulting in the following table:

```
A,B,C
1, 2, 3
4, 5, 6
```

If `null_padding` is set to true, all lines would be accepted, resulting in the following table:

```
'I like my csv files to have notes to make dialect detection harder', None, None
'I also like commas like this one : ', None, None
'A', 'B', 'C'
'1', '2', '3'
'4', '5', '6'
```

If the `ignore_errors` option is set, then the configuration that yields the most columns with the least inconsistent rows will be picked.

Type Detection

After deciding the dialect that will be used, we detect the types of each column. Our *Type Detection* considers the following types: `SQLNULL`, `BOOLEAN`, `BIGINT`, `DOUBLE`, `TIME`, `DATE`, `TIMESTAMP`, `VARCHAR`. These types are ordered in specificity, which means we first check if a column is a `SQLNULL`; if not, if it's a `BOOLEAN`, and so on, until it can only be a `VARCHAR`. DuckDB has more types than the ones used by default. Users can also define which types the sniffer should consider via the `auto_type_candidates` option.

At this phase, the type detection algorithm goes over the first chunk of data (i.e., 2048 tuples). This process starts on the second valid row (i.e., not a note) of the file. The first row is stored separately and not used for type detection. It will be later detected if the first row is a header or not. The type detection runs a per-column, per-value casting trial process to determine the column types. It starts off with a unique, per-column array with all types to be checked. It tries to cast the value of the column to that type; if it fails, it removes the type from the array, attempts to cast with the new type, and continues that process until the whole chunk is finished.

At this phase, we also determine the format of `DATE` and `TIMESTAMP` columns. The following formats are considered for `DATE` columns:

- `%m-%d-%Y`
- `%m-%d-%y`
- `%d-%m-Y`
- `%d-%m-%y`
- `%Y-%m-%d`
- `%y-%m-%d`

The following are considered for `TIMESTAMP` columns:

- `%Y-%m-%dT%H:%M:%S.%f`
- `%Y-%m-%d %H:%M:%S.%f`
- `%m-%d-%Y %I:%M:%S %p`
- `%m-%d-%y %I:%M:%S %p`
- `%d-%m-%Y %H:%M:%S`
- `%d-%m-%y %H:%M:%S`
- `%Y-%m-%d %H:%M:%S`
- `%y-%m-%d %H:%M:%S`

For columns that use formats outside this search space, they must be defined with the `dateformat` and `timestampformat` options.

As an example, let's consider the following CSV file.

```
Name, Age
,
Jack Black, 54
Kyle Gass, 63.2
```

The first row [`Name`, `Age`] will be stored separately for the header detection phase. The second row [`NULL`, `NULL`] will allow us to cast the first and second columns to `SQLNULL`. Therefore, their type candidate arrays will be the same: [`SQLNULL`, `BOOLEAN`, `BIGINT`, `DOUBLE`, `TIME`, `DATE`, `TIMESTAMP`, `VARCHAR`].

In the third row [`Jack Black`, `54`], things become more interesting. With 'Jack Black,' the type candidate array for column 0 will exclude all values with higher specificity, as 'Jack Black' can only be converted to a `VARCHAR`. The second column cannot be converted to either `SQLNULL` or `BOOLEAN`, but it will succeed as a `BIGINT`. Hence, the type candidate for the second column will be [`BIGINT`, `DOUBLE`, `TIME`, `DATE`, `TIMESTAMP`, `VARCHAR`].

In the fourth row, we have [`Kyle Gass`, `63.2`]. For the first column, there's no problem since it's also a valid `VARCHAR`. However, for the second column, a cast to `BIGINT` will fail, but a cast to `DOUBLE` will succeed. Hence, the new array of candidate types for the second column will be [`DOUBLE`, `TIME`, `DATE`, `TIMESTAMP`, `VARCHAR`].

Header Detection

The *Header Detection* phase simply obtains the first valid line of the CSV file and attempts to cast it to the candidate types in our columns. If there is a cast mismatch, we consider that row as the header; if not, we treat the first row as actual data and automatically generate a header.

In our previous example, the first row was [Name, Age], and the column candidate type arrays were [VARCHAR] and [DOUBLE, TIME, DATE, TIMESTAMP, VARCHAR]. Name is a string and can be converted to VARCHAR. Age is also a string, and attempting to cast it to DOUBLE will fail. Since the casting fails, the auto-detection algorithm considers the first row as a header, resulting in the first column being named Name and the second as Age.

If a header is not detected, column names will be automatically generated with the pattern `column${x}`, where x represents the column's position (0-based index) in the CSV file.

Type Replacement

Now that the auto-detection algorithm has discovered the header names, if the user specifies column types, the types detected by the sniffer will be replaced with them in the *Type Replacement* phase. For example, we can replace the Age type with FLOAT by using:

```
SELECT *
FROM read_csv('greatest_band_in_the_world.csv', types = {'Age': 'FLOAT'});
```

This phase is optional and will only be triggered if there are manually defined types.

Type Refinement

The *Type Refinement* phase performs the same tasks as type detection; the only difference is the granularity of the data on which the casting operator works, which is adjusted for performance reasons. During type detection, we conduct cast checks on a per-column, per-value basis.

In this phase, we transition to a more efficient vectorized casting algorithm. The validation process remains the same as in type detection, with types from type candidate arrays being eliminated if a cast fails.

How Fast Is the Sniffing?

To analyze the impact of running DuckDB's automatic detection, we execute the sniffer on the NYC taxi dataset. The file consists of 19 columns, 10,906,858 tuples and is 1.72 GB in size.

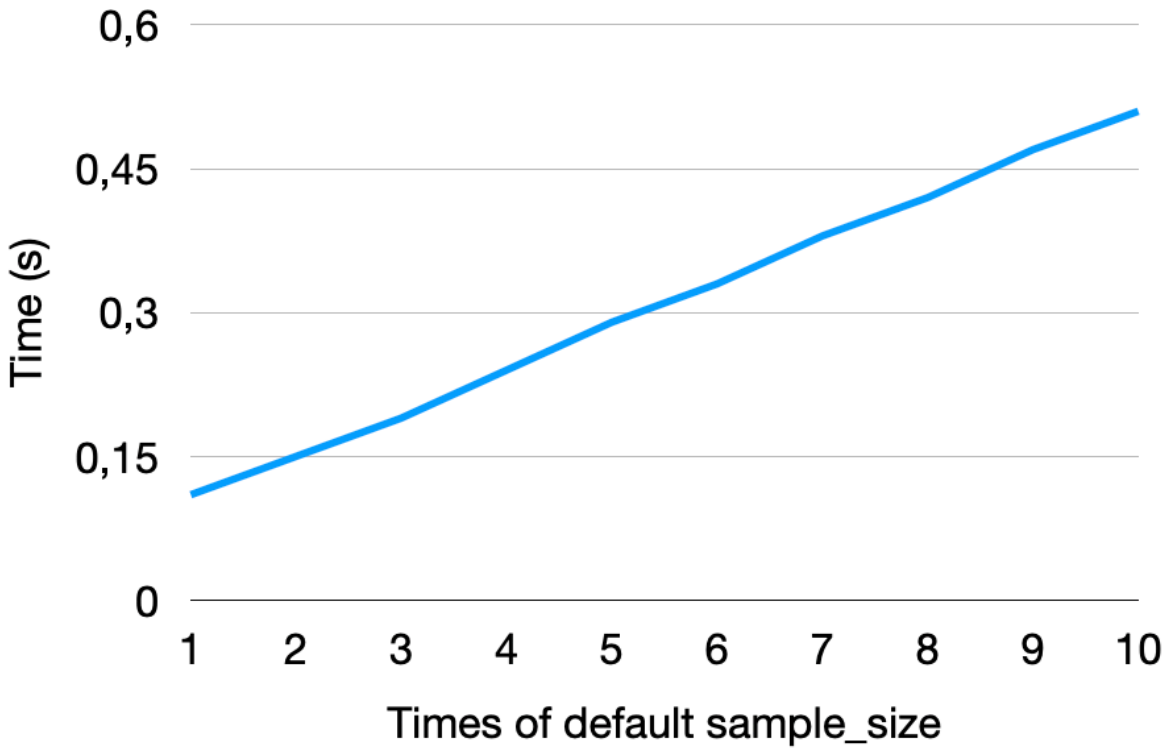
The cost of sniffing the dialect column names and types is approximately 4% of the total cost of loading the data.

Name	Time (s)
Sniffing	0.11
Loading	2.43

Varying Sampling Size

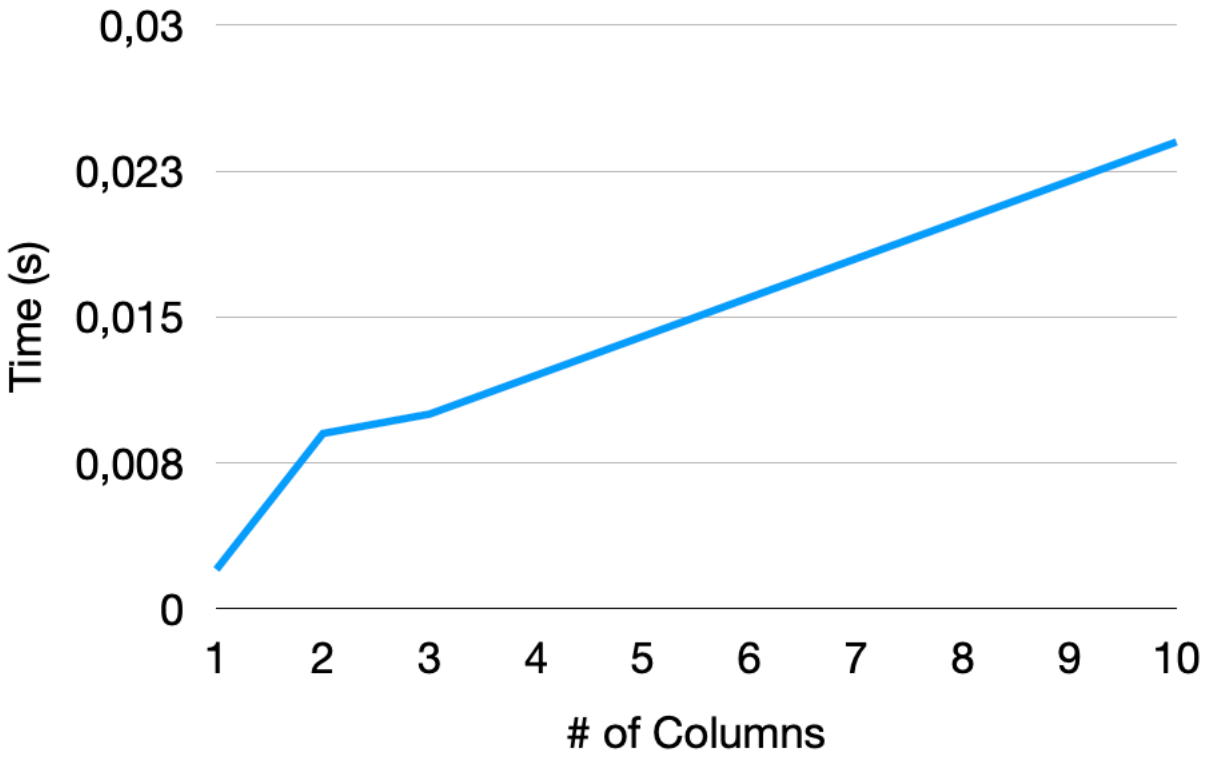
Sometimes, CSV files can have dialect options or more refined types that appear only later in the CSV file. In those cases, the `sample_size` option becomes an important tool for users to ensure that the sniffer examines enough data to make the correct decision. However, increasing the `sample_size` also leads to an increase in the total runtime of the sniffer because it uses more data to detect all possible dialects and types.

Below, you can see how increasing the default sample size by multiplier (see X axis) affects the sniffer's runtime on the NYC dataset. As expected, the total time spent on sniffing increases linearly with the total sample size.



Varying Number of Columns

The other main characteristic of a CSV file that will affect the auto-detection is the number of columns the file has. Here, we test the sniffer against a varying number of `INTEGER` type columns in files with 10,906,858 tuples. The results are depicted in the figure below. We can see that from one column to two, we have a steeper increase in runtime. That's because, for single columns, we have a simplified dialect detection due to the lack of delimiters. For the other columns, as expected, we have a more linear increase in runtime, depending on the number of columns.



Conclusion & Future Work

If you have unusual CSV files and want to query, clean up, or normalize them, DuckDB is already one of the top solutions available. It is very easy to get started. To read a CSV file with the sniffer, you can simply:

```
SELECT *
FROM 'path/to/csv_file.csv';
```

DuckDB's CSV auto-detection algorithm is an important tool to facilitate the exploration of CSV files. With its default options, it has a low impact on the total cost of loading and reading CSV files. Its main goal is to always be capable of reading files, doing a best-effort job even on files that are ill-defined.

We have a list of points related to the sniffer that we would like to improve in the future.

1. *Advanced Header Detection.* We currently determine if a CSV has a header by identifying a type mismatch between the first valid row and the remainder of the CSV file. However, this can generate false negatives if, for example, all the columns of a CSV are of a type VARCHAR. We plan on enhancing our Header Detection to perform matches with commonly used names for headers.
2. *Adding Accuracy and Speed Benchmarks.* We currently implement many accuracy and regression tests; however, due to the CSV's inherent flexibility, manually creating test cases is quite daunting. The plan moving forward is to implement a whole accuracy and regression test suite using the Pollock Benchmark
3. *Improved Sampling.* We currently execute the auto-detection algorithm on a sequential sample of data. However, it's very common that new settings are only introduced later in the file (e.g., quotes might be used only in the last 10% of the file). Hence, being able to execute the sniffer in distinct parts of the file can improve accuracy.
4. *Multi-Table CSV File.* Multiple tables can be present in the same CSV file, which is a common scenario when exporting spreadsheets to CSVs. Therefore, we would like to be able to identify and support these.
5. *Null-String Detection.* We currently do not have an algorithm in place to identify the representation of null strings.
6. *Decimal Precision Detection.* We also don't automatically detect decimal precision yet. This is something that we aim to tackle in the future.

7. *Parallelization.* Despite DuckDB's CSV Reader being fully parallelized, the sniffer is still limited to a single thread. Parallelizing it in a similar fashion to what is done with the CSV Reader (description coming in a future blog post) would significantly enhance sniffing performance and enable full-file sniffing.
8. *Sniffer as a stand-alone function.* Currently, users can utilize the `DESCRIBE` query to acquire information from the sniffer, but it only returns column names and types. We aim to expose the sniffing algorithm as a stand-alone function that provides the complete results from the sniffer. This will allow users to easily configure files using the exact same options without the need to rerun the sniffer.

Updates to the H2O.ai db-benchmark!

Publication date: 2023-11-03

Author: Tom Ebergen

TL;DR: The H2O.ai db-benchmark has been updated with new results. In addition, the AWS EC2 instance used for benchmarking has been changed to a c6id.metal for improved repeatability and fairness across libraries. DuckDB is the fastest library for both join and group by queries at almost every data size.

Skip directly to the results

The Benchmark Has Been Updated!

In April, DuckDB Labs published a blog post reporting updated H2O.ai db-benchmark results. Since then, the results haven't been updated. The original plan was to update the results with every DuckDB release. DuckDB 0.9.1 was recently released, and DuckDB Labs has updated the benchmark. While updating the benchmark, however, we noticed that our initial setup did not lend itself to being fair to all solutions. The machine used had network storage and could suffer from noisy neighbors. To avoid these issues, the whole benchmark was re-run on a c6id.metal machine.

New Benchmark Environment: c6id.metal Instance

Initially, updating the results to the benchmark showed strange results. Even using the same library versions from the prior update, some solutions regressed and others improved. We believe this variance came from the AWS EC2 instance type we chose: `m4.10xlarge`. The `m4.10xlarge` has 40 virtual CPUs and EBS storage. EBS storage is highly available network block storage for EC2 instances. When running compute-heavy benchmarks, a machine like the `m4.10xlarge` can suffer from the following issues:

- **Network storage** is an issue for benchmarking solutions that interact with storage frequently. For the 500 MB and 5 GB workloads, network storage was not an issue on the `m4.10xlarge` since all solutions could execute the queries in memory. For the 50 GB workload, however, network storage was an issue for the solutions that could not execute queries in memory. While the `m4.10xlarge` has dedicated EBS bandwidth, any read/write from storage is still happening over the network, which is usually slower than physically mounted storage. Solutions that frequently read and write to storage for the 50 GB queries end up doing this over the network. This network time becomes a chunk of the execution time of the query. If the network has variable performance, the query performance is then also variable.
- **Noisy neighbors** is a common issue when benchmarking on virtual CPUs. The previous machine most likely shared its compute hardware with other (neighboring) AWS EC2 instances. If these neighbors are also running compute heavy workloads, the physical CPU caches are repeatedly invalidated/flushed by the neighboring instance and the benchmark instance. When the CPU cache is shared between two workloads on two instances, both workloads require extra reads from memory for data that would already be in CPU cache on a non-virtual machine.

In order to be fair to all solutions, we decided to change the instance type to a metal instance with local storage. Metal instance types negate any noisy neighbor problems because the hardware is physical and not shared with any other AWS users/instances. Network storage problems are also fixed because solutions can read and write data to the local instance storage, which is physically mounted on the hardware.

Another benefit of the c6id.metal box is that it stresses parallel performance. There are 128 cores on the c6id.metal. Performance differences between solutions that can effectively use every core and solutions that cannot are clearly visible.

See the updated settings section on how settings were change for each solution when run on the new machine.

Updating the Benchmark

Moving forward we will update the benchmark when PRs with new performance numbers are provided. The PR should include a description of the changes to a solution script or a version update and new entries in the `time.csv` and `logs.csv` files. These entries will be verified using a different c6id.metal instance, and if there is limited variance, the PR will be merged and the results will be updated!

Updated Settings

1. ClickHouse
    - Storage: Any data this gets spilled to disk also needs to be on the NVMe drive. This has been changed in the new `format_and_mount.sh` script and the `clickhouse/clickhouse-mount-config.xml` file.
2. Julia (juliadf & juliads)
    - Threads: The threads were hardcoded for juliadf/juliads to 20/40 threads. Now the max number of threads are used. No option was given to spill to disk, so this was not changed/researched.
3. DuckDB
    - Storage: The DuckDB database file was specified to run on the NVMe mount.
4. Spark
    - Storage: There is an option to spill to disk. I was unsure of how to modify the storage location so that it was on the NVMe drive. Open to a PR with storage location changes and improved results!

Many solutions do not spill to disk, so they did not require any modification to use the instance storage. Other solutions use `parallel::ncores()` or default to a maximum number of cores for parallelism. Solution scripts were run in their current form on github.com/duckdblabs/db-benchmark. Please read the Updating the Benchmark section on how to re-run your solution.

Results

The first results you see are the 50 GB group by results. The benchmark runs every query twice per solution, and both runtimes are reported. The ”first time” can be considered a cold run, and the ”second time” can be considered a hot run. DuckDB and DuckDB-latest perform very well among all dataset sizes and variations.

The team at DuckDB Labs has been hard at work improving the performance of the out-of-core hash aggregates and joins. The most notable improvement is the performance of query 5 in the advanced group by queries. The cold run is almost an order of magnitude better than every other solution! DuckDB is also one of only two solutions to finish the 50 GB join query. Some solutions are experiencing timeouts on the 50 GB datasets. Solutions running the 50 GB group by queries are killed after running for 180 minutes, meaning all 10 group by queries need to finish within the 180 minutes. Solutions running the 50 GB join queries are killed after running for 360 minutes.

Link to result page

Extensions for DuckDB-Wasm

Publication date: 2023-12-18

Author: Carlo Piovesan

TL;DR: DuckDB-Wasm users can now load DuckDB extensions, allowing them to run extensions in the browser.

In this blog post, we will go over two exciting DuckDB features: the DuckDB-Wasm client and DuckDB extensions. I will discuss how these disjoint features have now been adapted to work together. These features are now available for DuckDB-Wasm users and you can try them out at shell.duckdb.org.

DuckDB Extensions

DuckDB's philosophy is to have a lean core system to ensure robustness and portability. However, a competing design goal is to be flexible and allow a wide range of functionality that is necessary to perform advanced analytics. To accommodate this, DuckDB has an extension mechanism for installing and loading extensions during runtime.

Running DuckDB Extensions Locally

For DuckDB, here is a simple end-to-end example using the command line interface:

```
INSTALL tpch;
LOAD tpch;
CALL dbgen(sf = 0.1);
PRAGMA tpch(7);
```

This script first installs the TPC-H extension from the official extension repository, which implements the popular TPC-H benchmark. It then loads the TPC-H extension, uses it to populate the database with generated data using the `dbgen` function. Finally, it runs TPC-H query 7.

This example demonstrates a case where we install an extension to complement DuckDB with a new feature (the TPC-H data generator), which is not part of the base DuckDB executable. Instead, it is downloaded from the extension repository, then loaded and executed it locally within the framework of DuckDB.

Currently, DuckDB has several extensions. These add support for filesystems, file formats, database and network protocols. Additionally, they implement new functions such as full text search.

DuckDB-Wasm

In an effort spearheaded by André Kohn, DuckDB was ported to the WebAssembly platform in 2021. WebAssembly, also known as Wasm, is a W3C standard language developed in recent years. Think of it as a machine-independent binary format that you can execute from within the sandbox of a web browser.

Thanks to DuckDB-Wasm, anyone has access to a DuckDB instance only a browser tab away, with all computation being executed locally within your browser and no data leaving your device. DuckDB-Wasm is a library that can be used in various deployments (e.g., notebooks that run inside your browser without a server). In this post, we will use the Web shell, where SQL statements are entered by the user line by line, with the behavior modeled after the DuckDB CLI shell.

DuckDB Extensions, in DuckDB-Wasm!

DuckDB-Wasm now supports DuckDB extensions. This support comes with four new key features. First, the DuckDB-Wasm library can be compiled with dynamic extension support. Second, DuckDB extensions can be compiled to a single WebAssembly module. Third, users and developers working with DuckDB-Wasm can now select the set of extensions they load. Finally, the DuckDB-Wasm shell's features are now much closer to the native CLI functionality.

Using the TPC-H Extension in DuckDB-Wasm

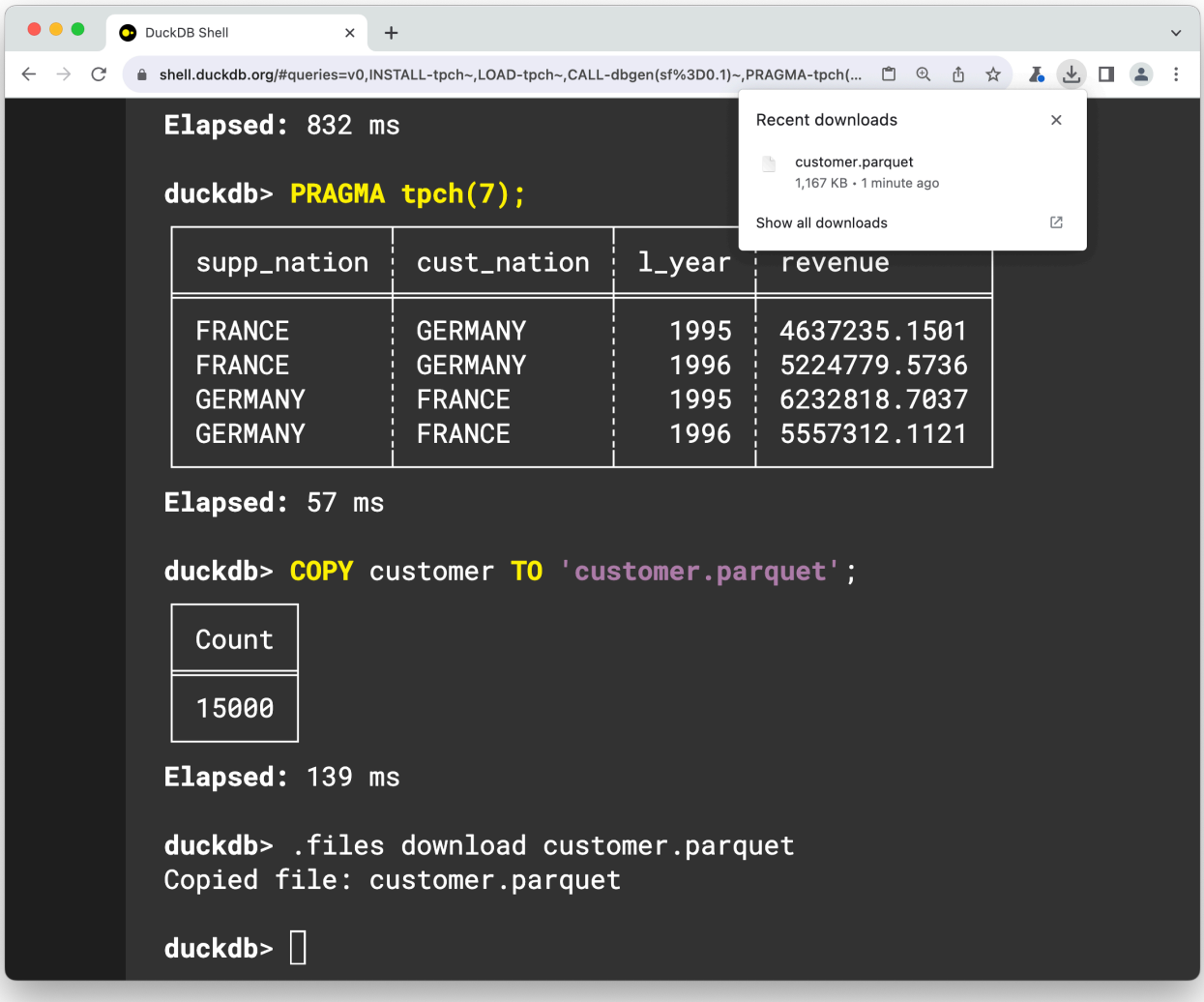
To demonstrate this, we will again use the TPC-H data generation example. To run this script in your browser, start an online DuckDB shell that runs these commands. The script will generate the TPC-H data set at scale factor 0.1, which corresponds to 100 MB in uncompressed CSV format.

Once the script is finished, you can keep executing queries, or you could even download the `customer.parquet` file (1 MB) using the following commands:

```
COPY customer TO 'customer.parquet';
.files download customer.parquet
```

This will first copy the `customer.parquet` to the DuckDB-Wasm file system, then download it via your browser.

In short, your DuckDB instance, which *runs entirely within your browser,* first installed and loaded the TPC-H extension. It then used the extension logic to generate data and convert it to a Parquet file. Finally, you could download the Parquet file as a regular file to your local file system.



Using the Spatial Extension in DuckDB-Wasm

To show the possibilities unlocked by DuckDB-Wasm extensions and test the capabilities of what's possible, what about using the spatial extension within DuckDB-Wasm? This extension implements geospatial types and functions that allow it to work with geospatial data and relevant workloads.

To install and load the spatial extension in DuckDB-Wasm, run:

```
INSTALL spatial;
LOAD spatial;
```

Using the spatial extension, the following query uses the New York taxi dataset, and calculates the area of the taxi zones for each borough:

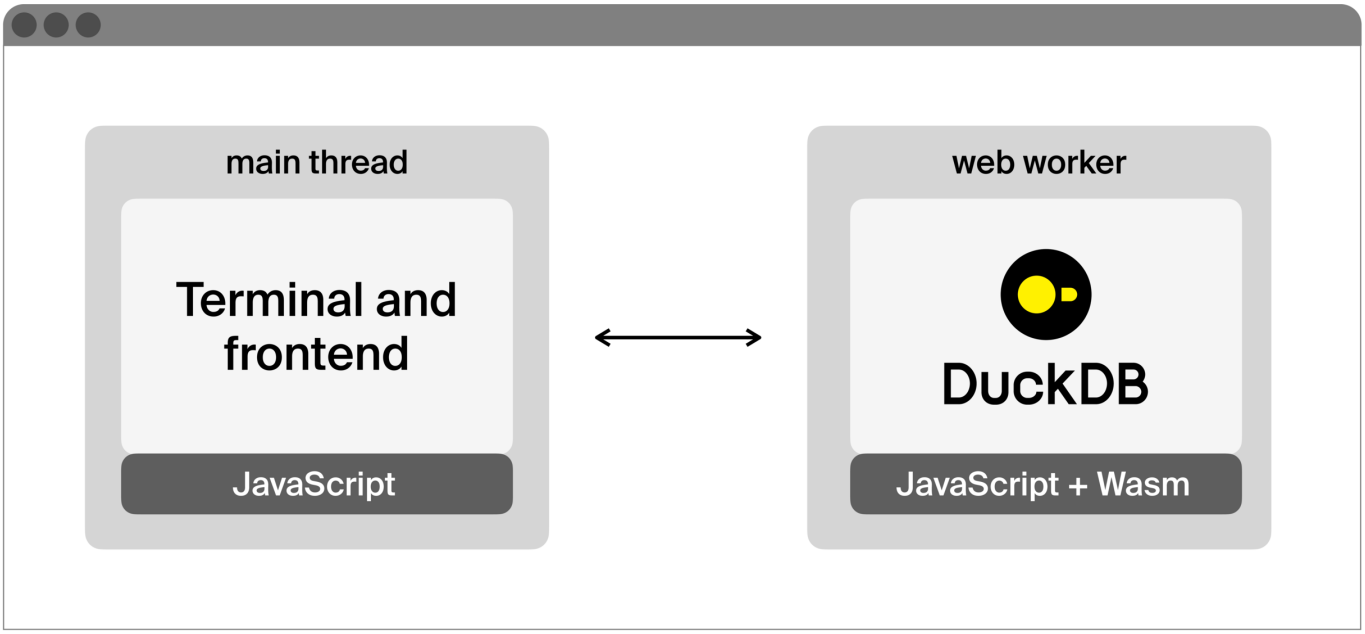
```
CREATE TABLE nyc AS
    SELECT
        borough,
        st_union_agg(geom) AS full_geom,
        st_area(full_geom) AS area,
        st_centroid(full_geom) AS centroid,
        count(*) AS count
    FROM
```

```
        st_read('https://raw.githubusercontent.com/duckdb/duckdb-spatial/main/test/data/nyc_taxi/taxi_
        zones/taxi_zones.shp')
GROUP BY borough;

SELECT borough, area, centroid::VARCHAR, count
FROM nyc;
```

Both your local DuckDB client and the online DuckDB shell will perform the same analysis.

Under the Hood

Let's dig into how this all works. The following figure shows an overview of DuckDB-Wasm's architecture. Both components in the figure run within the web browser.

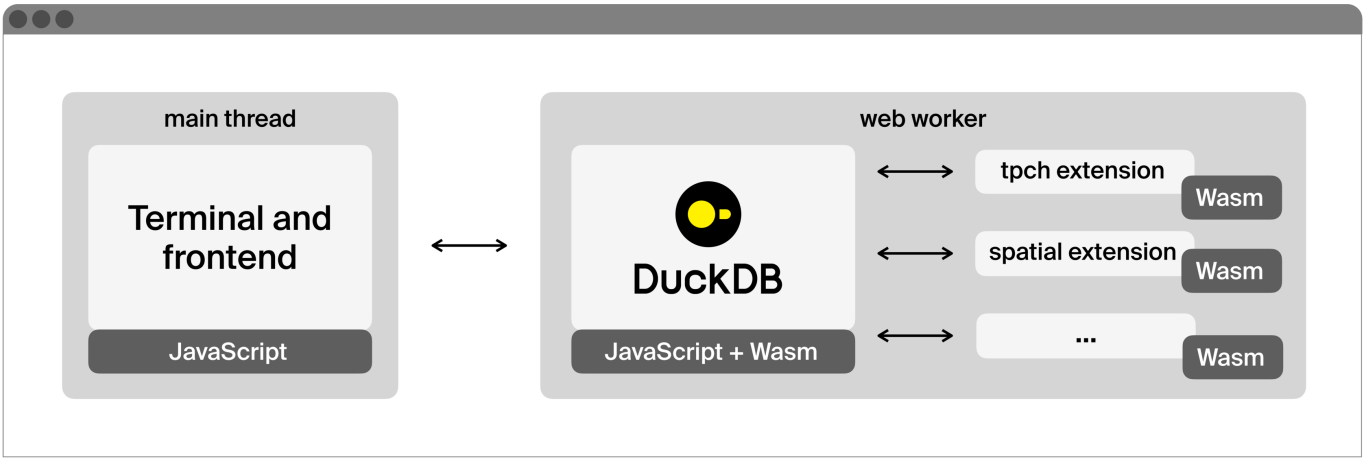


When you load DuckDB-Wasm in your browser, there are two components that will be set up:

(1) A main-thread wrapper library that acts as a bridge between users or code using DuckDB-Wasm and drives the background component.

(2) A DuckDB engine used to execute queries. This component lives in a Web Worker and communicates with the main thread component via messages. This component has a JavaScript layer that handles messages and the original DuckDB C++ logic compiled down to a single WebAssembly file.

What happens when we add extensions to the mix?



Extensions for DuckDB-Wasm are composed of a single WebAssembly module. This will encode the logic and data of the extensions, the list of functions that are going to be imported and exported, and a custom section encoding metadata that allows verification of the extension.

To make extension loading work, the DuckDB engine component blocks, fetches, and validates external WebAssembly code, then links it in, wires together import and export, and then the system will be connected and set to keep executing as if it was a single codebase.

The central code block that makes this possible is the following:

```
EM_ASM(
    {
        const xhr = new XMLHttpRequest();
        xhr.open("GET", UTF8ToString($0), false);
        xhr.responseType = "arraybuffer";
        xhr.send(null);
        var uInt8Array = xhr.response;
        // Check signatures / version compatibility left as an exercise
        WebAssembly.validate(uInt8Array);
        // Here we add the uInt8Array to Emscripten's filesystem,
        // for it to be found by dlopen
        FS.writeFile(UTF8ToString($1), new Uint8Array(uInt8Array));
    },
    filename.c_str(), basename.c_str()
);

auto lib_hdl = dlopen(basename.c_str(), RTLD_NOW | RTLD_LOCAL);
if (!lib_hdl) {
    throw IOException(
      "Extension \"%s\" could not be loaded: %s",
      filename,
      GetDLError()
    );
}
```

Here, we rely on two powerful features of Emscripten, the compiler toolchain we are using to compile DuckDB to WebAssembly.

First, `EM_ASM` allows us to inline JavaScript code directly in C++ code. It means that during runtime when we get to that block of code, the WebAssembly component will go back to JavaScript land, perform a blocking `XMLHttpRequest` on a URL such as https://extensions.duckdb.org/.../tpch.duckdb_extension.wasm, then validate that the package that has been just fetched is actually a valid WebAssembly module.

Second, we leverage Emscripten's `dlopen` implementation, which enables compatible WebAssembly modules to be linked together and act as a single composable codebase.

These enable implementing dynamic loading of extensions, when triggered via the SQL `LOAD` statement.

Developer Guide

We see two main groups of developers using extensions with DuckDB-Wasm.

- Developers working with DuckDB-Wasm: If you are building a website or a library that wraps DuckDB-Wasm, the new extension support means that there is now a wider range of functionality that can be exposed to your users.
- Developers working on DuckDB extensions: If you have written a DuckDB extension, or are thinking of doing so, consider porting it to DuckDB-Wasm. The DuckDB extension template repository contains the configuration required for compiling to DuckDB-Wasm.

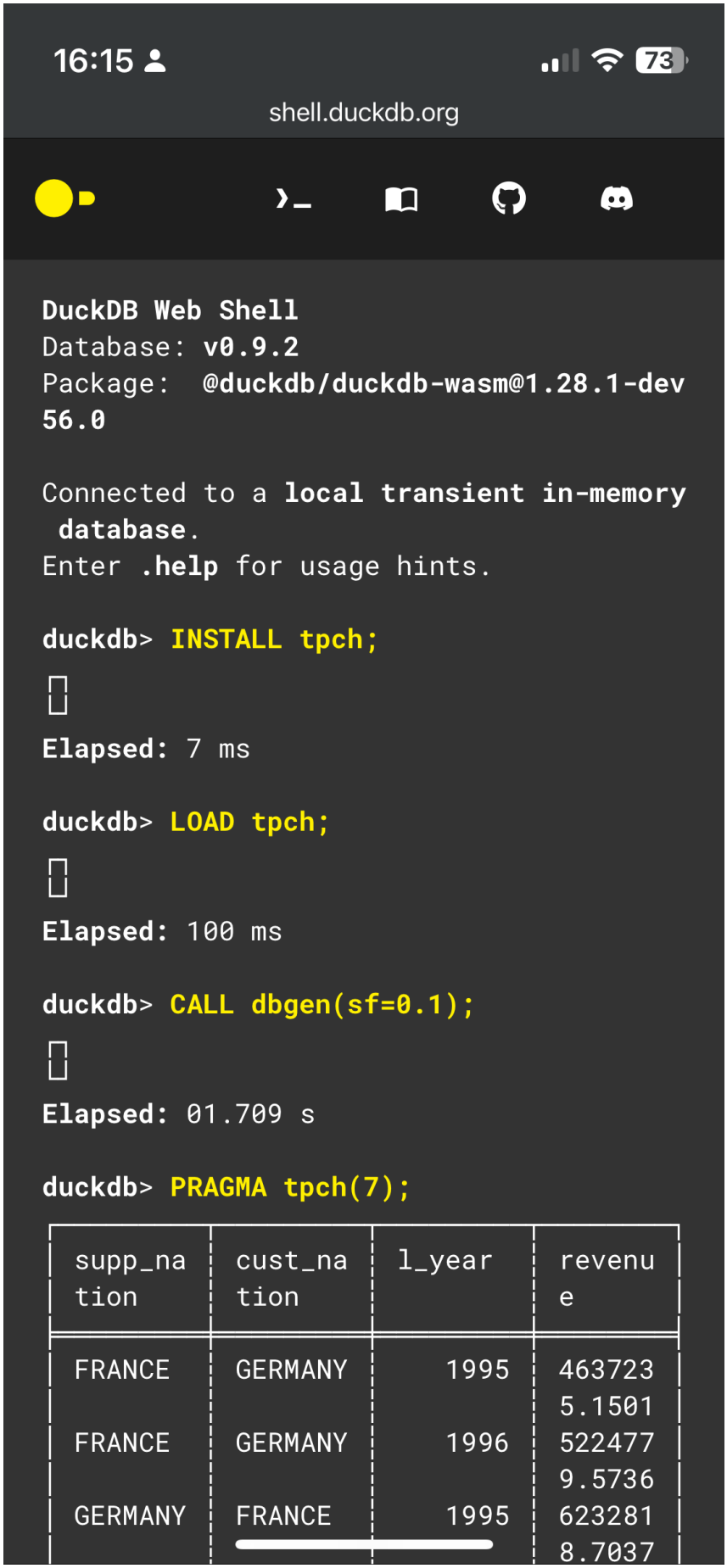
Limitations

DuckDB-Wasm extensions have a few inherent limitations. For example, it is not possible to communicate with native executables living on your machine, which is required by some extensions, such as the `postgres` scanner extension. Moreover, compilation to Wasm may not be currently supported for some libraries you are relying on, or capabilities might not be one-to-one with local executables due to additional requirements imposed on the browser, in particular around non-secure HTTP requests.

Conclusions

In this blog post, we explained how DuckDB-Wasm supports extensions, and demonstrated with multiple extensions: TPC-H, Parquet, and spatial.

Thanks to the portability of DuckDB, the scripts shown in this blog post also work on your smartphone:


16:15
shell.duckdb.org
DuckDB Web Shell
Database: v0.9.2
Package: @duckdb/duckdb-wasm@1.28.1-dev56.0
Connected to a local transient in-memory database.
Enter .help for usage hints.
duckdb> INSTALL tpch;
Elapsed: 7 ms
duckdb> LOAD tpch;
Elapsed: 100 ms
duckdb> CALL dbgen(sf=0.1);
Elapsed: 01.709 s
duckdb> PRAGMA tpch(7);
supp_nation
cust_nation
l_year
revenue
FRANCE
GERMANY
1995
463723 5.1501
FRANCE
GERMANY
1996
522477 9.5736
GERMANY
FRANCE
1995
623281 8.7037

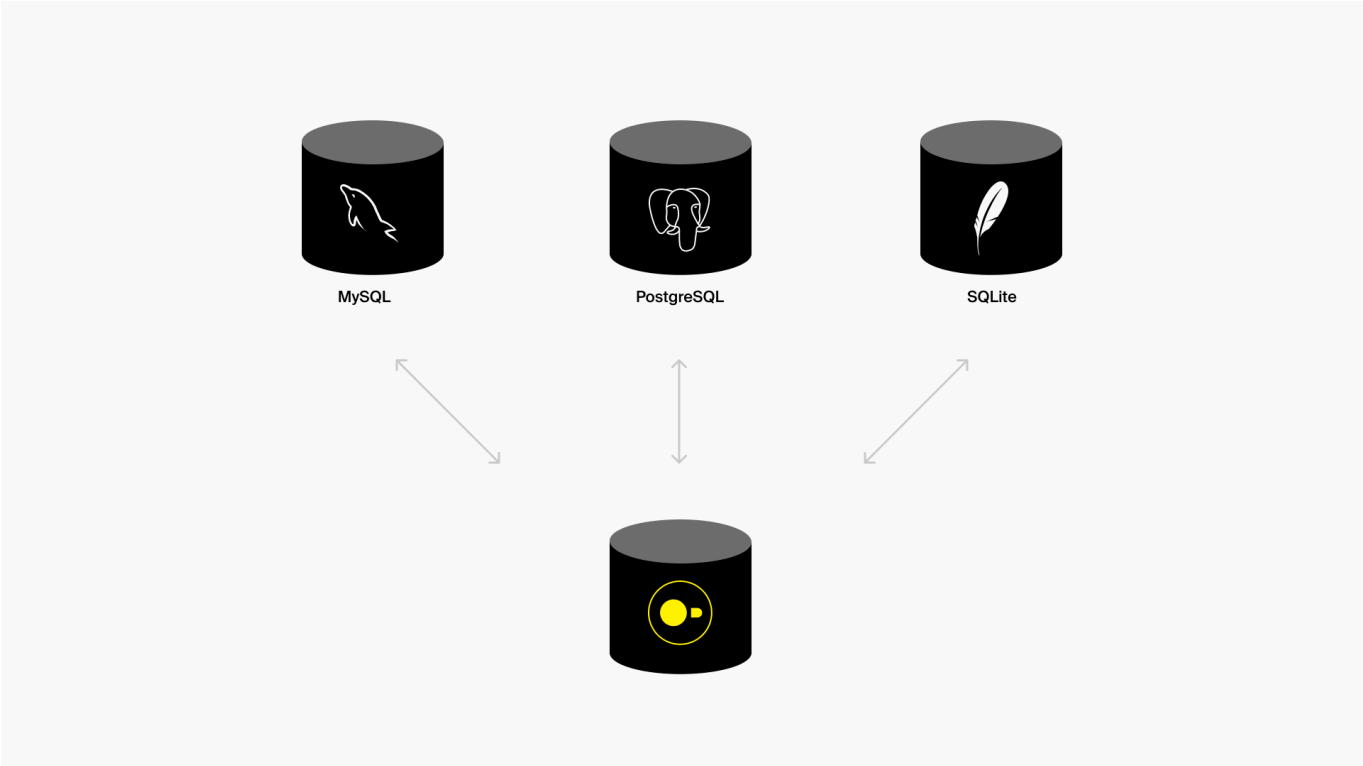
For updates on the latest developments, follow this blog and join the Wasm channel in our Discord. If you have an example of what's possible with extensions in DuckDB, let us know!

Multi-Database Support in DuckDB

Publication date: 2024-01-26

Author: Mark Raasveldt

TL;DR: DuckDB can attach MySQL, Postgres, and SQLite databases in addition to databases stored in its own format. This allows data to be read into DuckDB and moved between these systems in a convenient manner.



In modern data analysis, data must often be combined from a wide variety of different sources. Data might sit in CSV files on your machine, in Parquet files in a data lake, or in an operational database. DuckDB has strong support for moving data between many different data sources. However, this support has previously been limited to reading data and writing data to files.

DuckDB supports advanced operations on its own native storage format – such as deleting rows, updating values, or altering the schema of a table. It supports all of these operations using ACID semantics. This guarantees that your database is always left in a sane state – operations are atomic and do not partially complete.

DuckDB now has a pluggable storage and transactional layer. This flexible layer allows new storage back-ends to be created by DuckDB extensions. These storage back-ends can support all database operations in the same way that DuckDB supports them, including inserting data and even modifying schemas.

The MySQL, Postgres, and SQLite extensions implement this new pluggable storage and transactional layer, allowing DuckDB to connect to those systems and operate on them in the same way that it operates on its own native storage engine.

These extensions enable a number of useful features. For example, using these extensions you can:

- Export data from SQLite to JSON
- Read data from Parquet into Postgres
- Move data from MySQL to Postgres

... and much more.

Attaching Databases

The ATTACH statement can be used to attach a new database to the system. By default, a native DuckDB file will be attached. The TYPE parameter can be used to specify a different storage type. Alternatively, the `{type}:` prefix can be used.

For example, using the SQLite extension, we can open a SQLite database file and query it as we would query a DuckDB database.

```
ATTACH 'sakila.db' AS sakila (TYPE sqlite);
SELECT title, release_year, length FROM sakila.film LIMIT 5;
```

title varchar	release_year varchar	length int64
ACADEMY DINOSAUR	2006	86
ACE GOLDFINGER	2006	48
ADAPTATION HOLES	2006	50
AFFAIR PREJUDICE	2006	117
AFRICAN EGG	2006	130

The USE command switches the main database.

```
USE sakila;
SELECT first_name, last_name FROM actor LIMIT 5;
```

first_name varchar	last_name varchar
PENELOPE	GUINESS
NICK	WAHLBERG
ED	CHASE
JENNIFER	DAVIS
JOHNNY	LOLLOBRIGIDA

The SQLite database can be manipulated as if it were a native DuckDB database. For example, we can create a new table, populate it with values from a Parquet file, delete a few rows from the table and alter the schema of the table.

```
CREATE TABLE lineitem AS FROM 'lineitem.parquet' LIMIT 1000;
DELETE FROM lineitem WHERE l_returnflag = 'N';
ALTER TABLE lineitem DROP COLUMN l_comment;
```

The `duckdb_databases` table contains a list of all attached databases and their types.

```
SELECT database_name, path, type FROM duckdb_databases;
```

database_name varchar	path varchar	type varchar
sakila	sakila.db	sqlite
memory	NULL	duckdb

Mix and Match

While attaching to different database types is useful – it becomes even more powerful when used in combination. For example, we can attach both a SQLite, MySQL and a Postgres database.

```
ATTACH 'sqlite:sakila.db' AS sqlite;
ATTACH 'postgres:dbname=postgresscanner' AS postgres;
ATTACH 'mysql:user=root database=mysqlscanner' AS mysql;
```

Now we can move data between these attached databases and query them together. Let's copy the `film` table to MySQL, and the `actor` table to Postgres:

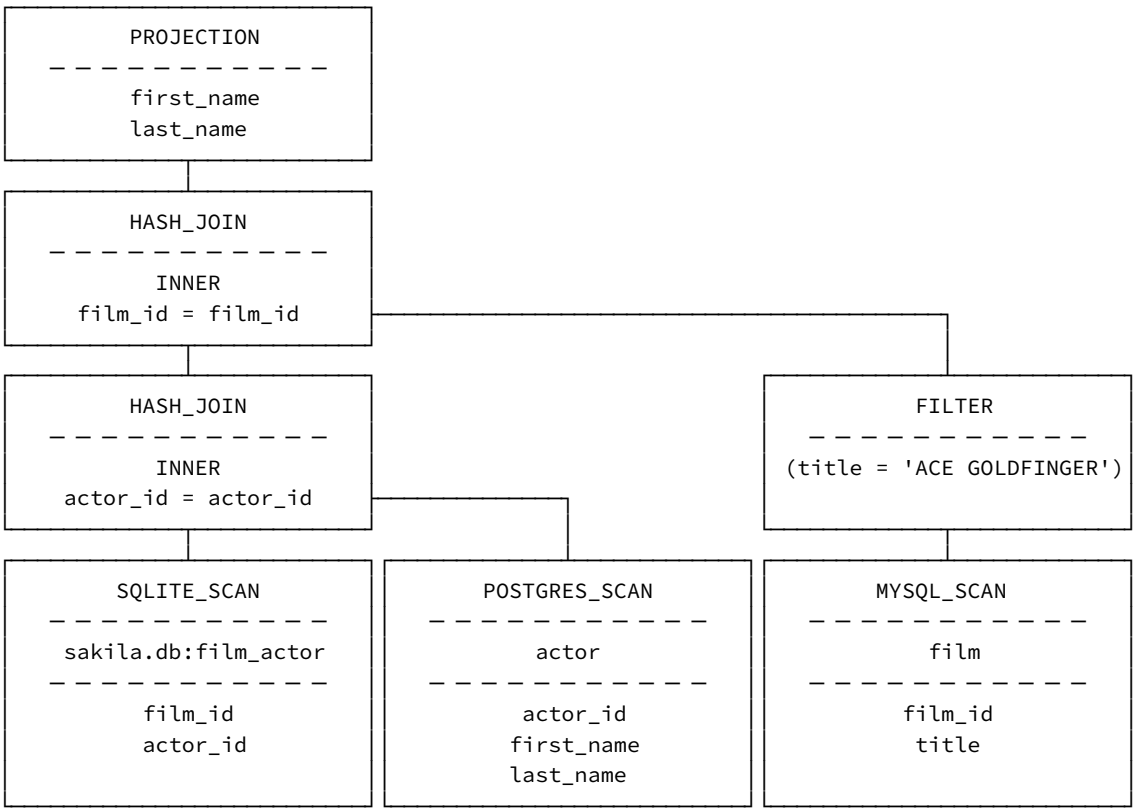
```
CREATE TABLE mysql.film AS FROM sqlite.film;
CREATE TABLE postgres.actor AS FROM sqlite.actor;
```

We can now join tables from these three attached databases together. Let's find all of the actors that starred in `Ace Goldfinger`.

```
SELECT first_name, last_name
FROM mysql.film
JOIN sqlite.film_actor ON (film.film_id = film_actor.film_id)
JOIN postgres.actor ON (actor.actor_id = film_actor.actor_id)
WHERE title = 'ACE GOLDFINGER';
```

first_name varchar	last_name varchar
BOB	FAWCETT
MINNIE	ZELLWEGER
SEAN	GUINESS
CHRIS	DEPP

Running `EXPLAIN` on the query shows how the data from the different engines is combined into the final query result.



Transactions

All statements executed within DuckDB are executed within a transaction. If an explicit `BEGIN TRANSACTION` is not called, every statement will execute in its own transaction. This also applies to queries that are executed over other storage engines. These storage engines also support explicit `BEGIN`, `COMMIT` and `ROLLBACK` statements.

For example, we can begin a transaction within our attached `SQLite` database, make a change, and then roll it back. The original data will be restored.

```
BEGIN;
TRUNCATE film;
SELECT title, release_year, length FROM film;
```

title varchar	release_year varchar	length int64
0 rows		

```
ROLLBACK;
SELECT title, release_year, length FROM film LIMIT 5;
```

title varchar	release_year varchar	length int64
ACADEMY DINOSAUR	2006	86
ACE GOLDFINGER	2006	48
ADAPTATION HOLES	2006	50
AFFAIR PREJUDICE	2006	117
AFRICAN EGG	2006	130

Multi-Database Transactions

Every storage engine has their own transactions that are stand-alone and managed by the storage engine itself. Opening a transaction in Postgres, for example, calls `BEGIN TRANSACTION` in the Postgres client. The transaction is managed by Postgres itself. Similarly, when the transaction is committed or rolled back, the storage engine handles this by itself.

Transactions are used both for **reading** and for **writing** data. For reading data, they are used to provide a consistent snapshot of the database. For writing, they are used to ensure all data in a transaction is packed together and written at the same time.

When executing a transaction that involves multiple attached databases we need to open multiple transactions: one per attached database that is used in the transaction. While this is not a problem when **reading** from the database, it becomes complicated when **writing**. In particular, when we want to `COMMIT` a transaction it is challenging to ensure that either (a) every database has successfully committed, or (b) every database has rolled back.

For that reason, it is currently not supported to **write** to multiple attached databases in a single transaction. Instead, an error is thrown when this is attempted:

```
BEGIN;
CREATE TABLE postgres.new_table (i INTEGER);
CREATE TABLE mysql.new_table (i INTEGER);
```

```
Error: Attempting to write to database "mysql" in a transaction that has
already modified database "postgres" - a single transaction can only write
to a single attached database.
```

Copying Data between Databases

`CREATE TABLE AS`, `INSERT INTO` and `COPY` can be used to copy data between different attached databases. The dedicated `COPY FROM DATABASE ... TO` can be used to copy all data from one database to another. This includes all tables and views that are stored in the source database.

```
-- attach a Postgres database
ATTACH 'postgres:dbname=postgresscanner' AS postgres;
-- attach a DuckDB file
ATTACH 'database.db' AS ddb;
-- export all tables and views from the Postgres database to the DuckDB file
COPY FROM DATABASE postgres TO ddb;
```

Directly Opening a Database

The explicit `ATTACH` statement is not required to connect to a different database type. When instantiating a DuckDB instance a connection can be made directly to a different database type using the `{type}:` prefix. For example, to connect to a SQLite file, use `sqlite:file.db`. To connect to a Postgres instance, use `postgres:dbname=postgresscanner`. This can be done in any client, including the CLI. For instance:

CLI:

```
duckdb sqlite:file.db
```

Python:

```
import duckdb
con = duckdb.connect('sqlite:file.db')
```

This is equivalent to attaching the storage engine and running `USE` afterwards.

Conclusion

DuckDB's pluggable storage engine architecture enables many use cases. By attaching multiple databases, data can be extracted in a transactionally safe manner for bulk ETL or ELT workloads, as well as for on-the-fly data virtualization workloads. These techniques also work well in combination, for example, by moving data in bulk on a regular cadence, while filling in the last few data points on the fly.

Pluggable storage engines also unlock new ways to handle concurrent writers in a data platform. Each separate process could write its output to a transactional database, and the results could be combined within DuckDB – all in a transactionally safe manner. Then, data analysis tasks can occur on the centralized DuckDB database for improved performance.

We look forward to hearing the many creative ways you are able to use this feature!

Future Work

We intend to continue enhancing the performance and capabilities of the existing extensions. In addition, all of these features can be leveraged by the community to connect to other databases.

Announcing DuckDB 0.10.0

Publication date: 2024-02-13

Authors: Mark Raasveldt and Hannes Mühleisen

TL;DR: The DuckDB team is happy to announce the latest DuckDB release (0.10.0). This release is named Fusca after the Velvet scoter native to Europe.

To install the new version, please visit the installation guide. The full release notes can be found on GitHub.

What's New in 0.10.0

There have been too many changes to discuss them each in detail, but we would like to highlight several particularly exciting features!

Below is a summary of those new features with examples, starting with a change in our SQL dialect that is designed to produce more intuitive results by default.

Breaking SQL Changes

Implicit Cast to VARCHAR. Previously, DuckDB would automatically allow any type to be implicitly cast to VARCHAR during function binding. As a result it was possible to e.g., compute the substring of an integer without using an implicit cast. Starting with this release, you will need to use an explicit cast here instead.

```
SELECT substring(42, 1, 1) AS substr;
```

```
No function matches the given name and argument types 'substring(...)'.
You might need to add explicit type casts.
```

To use an explicit cast, run:

```
SELECT substring(42::VARCHAR, 1, 1) AS substr;
```

substr
varchar
4

Alternatively, the `old_implicit_casting` setting can be used to revert this behavior, e.g.:

```
SET old_implicit_casting = true;
SELECT substring(42, 1, 1) AS substr;
```

substr
varchar
4

Literal Typing. Previously, integer and string literals behaved identically to the `INTEGER` and `VARCHAR` types. Starting with this release, `INTEGER_LITERAL` and `STRING_LITERAL` are separate types that have their own binding rules.

- `INTEGER_LITERAL` types can be implicitly converted to any integer type in which the value fits
- `STRING_LITERAL` types can be implicitly converted to **any** other type

This aligns DuckDB with Postgres, and makes operations on literals more intuitive. For example, we can compare string literals with dates – but we cannot compare VARCHAR values with dates.

```
SELECT d > '1992-01-01' AS result
FROM (VALUES (DATE '1992-01-01')) t(d);
```

result
boolean
false

```
SELECT d > '1992-01-01'::VARCHAR AS result
FROM (VALUES (DATE '1992-01-01')) t(d);
```

```
Binder Error:
Cannot compare values of type DATE and type VARCHAR - an explicit cast is required
```

Backward Compatibility

Backward compatibility refers to the ability of a newer DuckDB version to read storage files created by an older DuckDB version. This release is the first release of DuckDB that supports backward compatibility in the storage format. DuckDB v0.10 can read and operate on files created by the previous DuckDB version – DuckDB v0.9. This is made possible by the implementation of a new serialization framework.

Write with v0.9:

```
duckdb_092 v092.db
```

```
CREATE TABLE lineitem AS
FROM lineitem.parquet;
```

Read with v0.10:

```
duckdb_0100 v092.db
```

```
SELECT l_orderkey, l_partkey, l_comment
FROM lineitem
LIMIT 1;
```

l_orderkey int32	l_partkey int32	l_comment varchar
1	155190	to beans x-ray carefull

For future DuckDB versions, our goal is to ensure that any DuckDB version released **after** can read files created by previous versions, starting from this release. We want to ensure that the file format is fully backward compatible. This allows you to keep data stored in DuckDB files around and guarantees that you will be able to read the files without having to worry about which version the file was written with or having to convert files between versions.

Forward Compatibility

Forward compatibility refers to the ability of an older DuckDB version to read storage files produced by a newer DuckDB version. DuckDB v0.9 is **partially** forward compatible with DuckDB v0.10. Certain files created by DuckDB v0.10 can be read by DuckDB v0.9.

Write with v0.10:

```
duckdb_0100 v010.db
```

```
CREATE TABLE lineitem AS
FROM lineitem.parquet;
```

Read with v0.9:

```
duckdb_092 v010.db
```

```
SELECT l_orderkey, l_partkey, l_comment
FROM lineitem
LIMIT 1;
```

l_orderkey int32	l_partkey int32	l_comment varchar
1	155190	to beans x-ray carefull

Forward compatibility is provided on a **best effort** basis. While stability of the storage format is important – there are still many improvements and innovations that we want to make to the storage format in the future. As such, forward compatibility may be (partially) broken on occasion.

For this release, DuckDB v0.9 is able to read files created by DuckDB v0.10 provided that:

- The database file does not contain views
- The database file does not contain new types (`ARRAY`, `UHUGEINT`)
- The database file does not contain indexes (`PRIMARY KEY`, `FOREIGN KEY`, `UNIQUE`, explicit indexes)
- The database file does not contain new compression methods (`ALP`). As ALP is automatically used to compress `FLOAT` and `DOUBLE` columns – that means forward compatibility in practice often does not work for `FLOAT` and `DOUBLE` columns unless `ALP` is explicitly disabled through configuration.

We expect that as the format stabilizes and matures this will happen less frequently – and we hope to offer better guarantees in allowing DuckDB to read files written by future DuckDB versions.

CSV Reader Rework

CSV Reader Rework. The CSV reader has received a major overhaul in this release. The new CSV reader uses efficient state machine transitions to speed through CSV files. This has greatly sped up performance of the CSV reader, particularly in multi-threaded scenarios. In addition, in the case of malformed CSV files, reported error messages should be more clear.

Below is a benchmark comparing the loading time of 11 million rows of the NYC Taxi dataset from a CSV file on an M1 Max with 10 cores:

Version	Load time
v0.9.2	2.6 s
v0.10.0	1.2 s

Furthermore, many optimizations have been done that make running queries over CSV files directly significantly faster as well. Below is a benchmark comparing the execution time of a `SELECT count(*)` query directly over the NYC Taxi CSV file.

Version	Query time
v0.9.2	1.8 s
v0.10.0	0.3 s

Fixed-Length Arrays

Fixed-Length Arrays. This release introduces the fixed-length array type. Fixed-length arrays are similar to lists, however, every value must have the same fixed number of elements in them.

```
CREATE TABLE vectors (v DOUBLE[3]);
INSERT INTO vectors VALUES ([1, 2, 3]);
```

Fixed-length arrays can be operated on faster than variable-length lists as the size of each list element is known ahead of time. This release also introduces specialized functions that operate over these arrays, such as `array_cross_product`, `array_cosine_similarity`, and `array_inner_product`.

```
SELECT array_cross_product(v, [1, 1, 1]) AS result
FROM vectors;
```

```
┌──────────────────┐
│      result      │
│    double[3]     │
├──────────────────┤
│ [-1.0, 2.0, -1.0] │
└──────────────────┘
```

See the Array Type page in the documentation for more information.

Multi-Database Support

DuckDB can now attach MySQL, Postgres, and SQLite databases in addition to databases stored in its own format. This allows data to be read into DuckDB and moved between these systems in a convenient manner, as attached databases are fully functional, appear just as regular tables, and can be updated in a safe, transactional manner. More information about multi-database support can be found in our recent blog post.

```
ATTACH 'sqlite:sakila.db' AS sqlite;
ATTACH 'postgres:dbname=postgresscanner' AS postgres;
ATTACH 'mysql:user=root database=mysqlscanner' AS mysql;
```

Secret Manager

DuckDB integrates with several cloud storage systems such as S3 that require access credentials to access data. In the current version of DuckDB, authentication information is configured through DuckDB settings, e.g., `SET s3_access_key_id = '...';`. While this worked, it had several shortcomings. For example, it was not possible to set different credentials for different S3 buckets without modifying the settings between queries. Because settings are not considered secret, it was also possible to query them using `duckdb_settings()`.

With this release, DuckDB adds a new "Secrets Manager" to manage secrets in a better way. We now have a unified user interface for secrets across all backends that use them. Secrets can be scoped, so different storage prefixes can have different secrets, allowing, for example, joining across organizations in a single query. Secrets can also be persisted, so that they do not need to be specified every time DuckDB is launched.

Secrets are typed, their type identifies which service they are for. For example, this release can manage secrets for S3, Google Cloud Storage, Cloudflare R2 and Azure Blob Storage. For each type, there are one or more "secret providers" that specify how the secret is created. Secrets can also have an optional scope, which is a file path prefix that the secret applies to. When fetching a secret for a path, the secret scopes are compared to the path, returning the matching secret for the path. In the case of multiple matching secrets, the longest prefix is chosen.

Finally, secrets can be temporary or persistent. Temporary secrets are used by default – and are stored in-memory for the life span of the DuckDB instance similar to how settings worked previously. Persistent secrets are stored in unencrypted binary format in the `~/.duckdb/stored_secrets` directory. On startup of DuckDB, persistent secrets are read from this directory and automatically loaded.

For example, to create a temporary unscoped secret to access S3, we can now use the following syntax:

```
CREATE SECRET (
    TYPE s3,
    KEY_ID '<AKIAIOSFODNN7EXAMPLE>',
    SECRET '<wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY>',
    REGION '<us-east-1>'
);
```

If two secrets exist for a service type, the scope can be used to decide which one should be used. For example:

```
CREATE SECRET secret1 (
    TYPE s3,
    KEY_ID 'my_key1',
```

```
    SECRET 'my_secret1',
    SCOPE 's3://<my-bucket>'
);

CREATE SECRET secret2 (
    TYPE s3,
    KEY_ID 'my_key2',
    SECRET 'my_secret2',
    SCOPE 's3://<my-other-bucket>'
);
```

Now, if the user queries something from `s3://<my-other-bucket>/something`{:.language-sql .highlight}, `secret2` will be chosen automatically for that request.

Secrets can be listed using the built-in table-producing function, e.g., by using `FROM duckdb_secrets();`{:.language-sql .highlight}. Sensitive information will be redacted.

In order to persist secrets between DuckDB database instances, we can now use the `CREATE PERSISTENT SECRET` command, e.g.:

```
CREATE PERSISTENT SECRET my_persistent_secret (
    TYPE s3,
    KEY_ID 'my_key',
    SECRET 'my_secret'
);
```

As mentioned, this will write the secret (unencrypted, so beware) to the `~/.duckdb/stored_secrets` directory.

See the Create Secret page in the documentation for more information.

Temporary Memory Manager

DuckDB has support for larger-than-memory operations, which means that memory-hungry operators such as aggregations and joins can offload part of their intermediate results to temporary files on disk should there not be enough memory available.

Before, those operators started offloading to disk if their memory usage reached around 60% of the available memory (as defined by the memory limit). This works well if there is exactly one of these operations happening at the same time. If multiple memory-intensive operations are happening simultaneously, their combined memory usage may exceed the memory limit, causing DuckDB to throw an error.

This release introduces the so-called "Temporary Memory Manager", which manages the temporary memory of concurrent operations. It works as follows: Memory-intensive operations register themselves with the Temporary Manager. Each registration is guaranteed some minimum amount of memory by the manager depending on the number of threads and the current memory limit. Then, the memory-intensive operations communicate how much memory they would currently like to use. The manager can approve this or respond with a reduced allocation. In a case of a reduced allocation, the operator will need to dynamically reduce its memory requirements, for example by switching algorithms.

For example, a hash join might adapt its operation and perform a partitioned hash join instead of a full in-memory one if not enough memory is available.

Here is an example:

```
PRAGMA memory_limit = '5GB';
SET temp_directory = '/tmp/duckdb_temporary_memory_manager';

CREATE TABLE tbl AS
SELECT range AS i,
       range AS j
FROM range(100_000_000);

SELECT max(i),
```

```
       max(t1.j),
       max(t2.j),
       max(t3.j),
FROM tbl AS t1
JOIN tbl AS t2 USING (i)
JOIN tbl AS t3 USING (i);
```

Note that a temporary directory has to be set here, because the operators actually need to offload data to disk to complete this query given this memory limit.

With the new version 0.10.0, this query completes in ca. 5 s on a MacBook, while it would error out on the previous version with `Error: Out of Memory Error: failed to pin block of size ....`

Adaptive Lossless Floating-Point Compression (ALP)

Floating point numbers are notoriously difficult to compress efficiently, both in terms of compression ratio as well as speed of compression and decompression. In the past, DuckDB had support for the then state-of-the-art "Chimp" and the "Patas" compression methods. Turns out, those were not the last word in floating point compression. Researchers Azim Afroozeh, Leonard Kuffo and (the one and only) Peter Boncz have recently published a paper titled "ALP: Adaptive Lossless floating-Point Compression" at SIGMOD, a top-tier academic conference for data management research. In an uncommon yet highly commendable move, they have also sent a pull request to DuckDB. The new compression scheme replaces Chimp and Patas. Inside DuckDB, ALP is **x2-4 times faster** than Patas (at decompression) achieving **compression ratios twice as high** (sometimes even much more).

Compression	Load	Query	Size
ALP	0.434 s	0.020 s	184 MB
Patas	0.603 s	0.080 s	275 MB
Uncompressed	0.316 s	0.012 s	489 MB

As a user, you don't have to do anything to make use of the new ALP compression method, DuckDB will automatically decide during checkpointing whether using ALP is beneficial for the specific dataset.

CLI Improvements

The command-line client has seen a lot of work this release. In particular, multi-line editing has been made the default mode, and has seen many improvements. The query history is now also multi-line. Syntax highlighting has improved – missing brackets and unclosed quotes are highlighted as errors, and matching brackets are highlighted when the cursor moves over them. Compatibility with read-line has also been greatly extended.

```
D SELECT
·     l_returnflag,
·     l_linestatus, -- comment
·     sum(l_quantity) AS sum_qty, -- comment two
·     sum(l_extendedprice) AS sum_base_price,
·     sum(l_extendedprice * (1 - l_discount)) AS sum_disc_price,
·     sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) AS sum_charge,
▸     avg(l_quantity) AS avg_qty,
·     avg(l_extendedprice) AS avg_price,
·     avg(l_discount AS avg_disc,
·     count(*) AS count_order
· FROM
·     lineitem
· WHERE
·     l_shipdate <= CAST('1998-09-02' AS date)
· GROUP BY
·     l_returnflag,
·     l_linestatus
· ORDER BY
·     l_returnflag,
·     l_linestatus
```

See the extended CLI docs for more information.

Final Thoughts

These were a few highlights – but there are many more features and improvements in this release. Below are a few more highlights. The full release notes can be found on GitHub.

New Features

- `COMMENT ON`
- `COPY FROM DATABASE`
- `UHUGEINT` type
- Window `EXCLUDE` and Window `DISTINCT` support
- Parquet encryption support
- Indexes for Lambda parameters
- `EXCEPT ALL`/`INTERSECT ALL`
- `DESCRIBE`/`SHOW`/`SUMMARIZE` as subquery
- Support recursive CTEs in correlated subqueries

New Functions

- `parquet_kv_metadata` and `parquet_file_metadata` functions
- `read_text`/`read_blob` table functions
- `list_reduce`, `list_where`, `list_zip`, `list_select`, `list_grade_up`

Storage Improvements

- Vacuuming partial deletes
- Parallel checkpointing
- Checksum WAL

Optimizations

- Parallel streaming query result
- Struct filter pushdown
- `first(x ORDER BY y)` optimizations

Acknowledgments

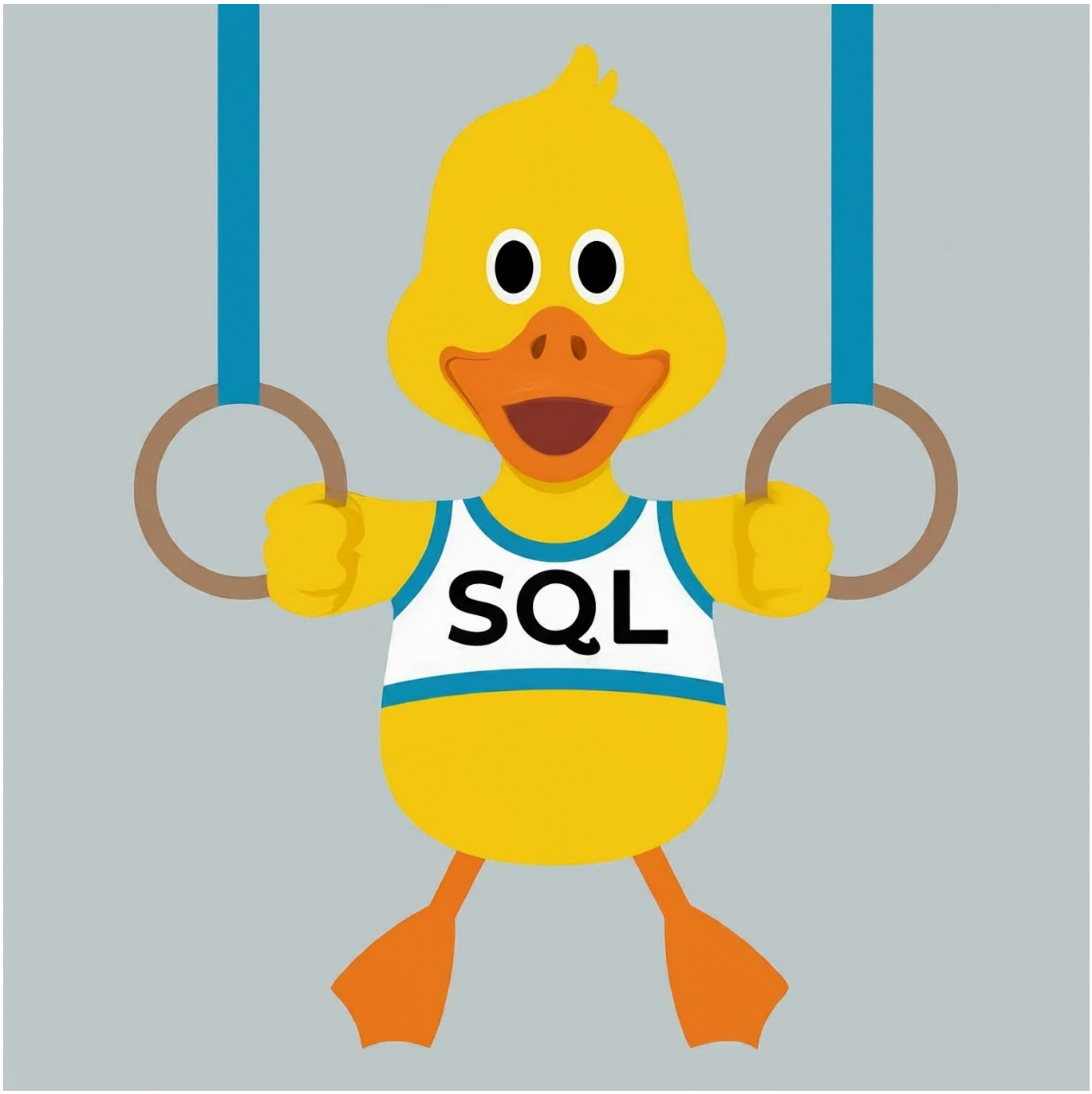
We would like to thank all of the contributors for their hard work on improving DuckDB.

SQL Gymnastics: Bending SQL into Flexible New Shapes

Publication date: 2024-03-01

Author: Alex Monahan

TL;DR: Combining multiple features of DuckDB's friendly SQL allows for highly flexible queries that can be reused across tables.



DuckDB's especially friendly SQL dialect simplifies common query operations. However, these features also unlock new and flexible ways

to write advanced SQL! In this post we will combine multiple friendly features to both move closer to real-world use cases and stretch your imagination. These queries are useful in their own right, but their component pieces are even more valuable to have in your toolbox.

What is the craziest thing you have built with SQL? We want to hear about it! Tag DuckDB on X (the site formerly known as Twitter) or LinkedIn, and join the DuckDB Discord community.

Traditional SQL Is Too Rigid to Reuse

SQL queries are typically crafted specifically for the unique tables within a database. This limits reusability. For example, have you ever seen a library of high-level SQL helper functions? SQL as a language typically is not flexible enough to build reusable functions. Today, we are flying towards a more flexible future!

Dynamic Aggregates Macro

In SQL, typically the columns to `SELECT` and `GROUP BY` must be specified individually. However, in many business intelligence workloads, groupings and aggregate functions must be easily user-adjustable. Imagine an interactive charting workflow – first I want to plot total company revenue over time. Then if I see a dip in revenue in that first plot, I want to adjust the plot to group the revenue by business unit to see which section of the company caused the issue. This typically requires templated SQL, using a language that compiles down to SQL (like Malloy), or building a SQL string using another programming language. How much we can do with just SQL?

Let's have a look at a flexible SQL-only approach and then break down how it is constructed.

col1	col2	col3	col4
1	1	1	1
2	2	0	1
3	3	1	1
4	4	0	1
5	0	1	1
6	1	0	1
7	2	1	1
8	3	0	1
9	4	1	1
10	0	0	1

Creating the Macro

The macro below accepts lists of columns to include or exclude, a list of columns to aggregate, and an aggregate function to apply. All of these can be passed in as parameters from the host language that is querying the database.

```
-- We use a table macro (or function) for reusability
CREATE OR REPLACE MACRO dynamic_aggregates(
        included_columns,
        excluded_columns,
        aggregated_columns,
        aggregate_function
    ) AS TABLE (
    FROM example
    SELECT
```

```
        -- Use a COLUMNS expression to only select the columns
        -- we include or do not exclude
        COLUMNS(c -> (
            -- If we are not using an input parameter (list is empty),
            -- ignore it
            (list_contains(included_columns, c) OR
             len(included_columns) = 0)
            AND
            (NOT list_contains(excluded_columns, c) OR
             len(excluded_columns) = 0)
            )),
        -- Use the list_aggregate function to apply an aggregate
        -- function of our choice
        list_aggregate(
            -- Convert to a list (to enable the use of list_aggregate)
            list(
                -- Use a COLUMNS expression to choose which columns
                -- to aggregate
                COLUMNS(c -> list_contains(aggregated_columns, c))
            ), aggregate_function
        )
    GROUP BY ALL -- Group by all selected but non-aggregated columns
    ORDER BY ALL -- Order by each column from left to right
);
```

Executing the Macro

Now we can use that macro for many different aggregation operations. For illustrative purposes, the 3 queries below show different ways to achieve identical results.

Select col3 and col4, and take the minimum values of col1 and col2:

```
FROM dynamic_aggregates(
    ['col3', 'col4'], [], ['col1', 'col2'], 'min'
);
```

Select all columns except col1 and col2, and take the minimum values of col1 and col2:

```
FROM dynamic_aggregates(
    [], ['col1', 'col2'], ['col1', 'col2'], 'min'
);
```

If the same column is in both the included and excluded list, it is excluded (exclusions win ties). If we include col2, col3, and col4, but we exclude col2, then it is as if we only included col3 and col4:

```
FROM dynamic_aggregates(
    ['col2', 'col3', 'col4'], ['col2'], ['col1', 'col2'], 'min'
);
```

Executing either of those queries will return this result:

col3	col4	list_aggregate(list(example.col1), 'min')	list_aggregate(list(example.col2), 'min')
0	1	2	0
1	1	1	0

Understanding the Design

The first step of our flexible table macro is to choose a specific table using DuckDB's FROM-first syntax. Well that's not very dynamic! If we wanted to, we could work around this by creating a copy of this macro for each table we want to expose to our application. However, we will show another approach in our next example, and completely solve the issue in a follow up blog post with an in-development DuckDB feature. Stay tuned!

Then we `SELECT` our grouping columns based on the list parameters that were passed in. The `COLUMNS` expression will execute a lambda function to decide which columns meet the criteria to be selected.

The first portion of the lambda function checks if a column name was passed in within the `included_columns` list. However, if we choose not to use an inclusion rule (by passing in a blank `included_columns` list), we want to ignore that parameter. If the list is blank, `len(included_columns) = 0` will evaluate to `true` and effectively disable the filtering on `included_columns`. This is a common pattern for optional filtering that is generically useful across a variety of SQL queries. (Shout out to my mentor and friend Paul Bloomquist for teaching me this pattern!)

We repeat that pattern for `excluded_columns` so that it will be used if populated, but ignored if left blank. The `excluded_columns` list will also win ties, so that if a column is in both lists, it will be excluded.

Next, we apply our aggregate function to the columns we want to aggregate. It is easiest to follow the logic of this part of the query by working from the innermost portion outward. The `COLUMNS` expression will acquire the columns that are in our `aggregated_columns` list. Then, we do a little bit of gymnastics (it had to happen sometime...).

If we were to apply a typical aggregation function (like `sum` or `min`), it would need to be specified statically in our macro. To pass it in dynamically as a string (potentially all the way from the application code calling this SQL statement), we take advantage of a unique property of the `list_aggregate` function. It accepts the name of a function (as a string) in its second parameter. So, to use this unique property, we use the `list` aggregate function to transform all the values within each group into a list. Then we use the `list_aggregate` function to apply the `aggregate_function` we passed into the macro to each list.

Almost done! Now `GROUP BY ALL` will automatically choose to group by the columns returned by the first `COLUMNS` expression. The `ORDER BY ALL` expression will order each column in ascending order, moving from left to right.

We made it!

> Extra credit! In the next release of DuckDB, version 0.10.1, we will be able to apply a dynamic alias to the result of a `COLUMNS` expression. For example, each new aggregate column could be renamed in the pattern `agg_[the original column name]`. This will unlock the ability to chain together these type of macros, as the naming will be predictable.

Takeaways

Several of the approaches used within this macro can be applied in a variety of ways in your SQL workflows. Using a lambda function in combination with the `COLUMNS` expression can allow you to select any arbitrary list of columns. The `OR len(my_list) = 0` trick allows list parameters to be ignored when blank. Once you have that arbitrary set of columns, you can even apply a dynamically chosen aggregation function to those columns using `list` and `list_aggregate`.

However, we still had to specify a table at the start. We are also limited to aggregate functions that are available to be used with `list_aggregate`. Let's relax those two constraints!

Creating Version 2 of the Macro

This approach takes advantage of two key concepts:

- Macros can be used to create temporary aggregate functions
- A macro can query a Common Table Expression (CTE) / `WITH` clause that is in scope during execution

```
CREATE OR REPLACE MACRO dynamic_aggregates_any_cte_any_func(
    included_columns,
    excluded_columns,
```

```
    aggregated_columns
    /* No more aggregate_function */
) AS TABLE (
    FROM any_cte -- No longer a fixed table!
    SELECT
        COLUMNS(c -> (
            (list_contains(included_columns, c) OR
            len(included_columns) = 0)
            AND
            (NOT list_contains(excluded_columns, c) OR
            len(excluded_columns) = 0)
            )),
        -- We no longer convert to a list,
        -- and we refer to the latest definition of any_func
        any_func(COLUMNS(c -> list_contains(aggregated_columns, c)))
    GROUP BY ALL
    ORDER BY ALL
);
```

Executing Version 2

When we call this macro, there is additional complexity. We no longer execute a single statement, and our logic is no longer completely parameterizable (so some templating or SQL construction will be needed). However, we can execute this macro against any arbitrary CTE, using any arbitrary aggregation function. Pretty powerful and very reusable!

```
-- We can define or redefine any_func right before calling the macro
CREATE OR REPLACE TEMP FUNCTION any_func(x)
    AS 100.0 * sum(x) / count(x);

-- Any table structure is valid for this CTE!
WITH any_cte AS (
    SELECT
        x % 11 AS id,
        x % 5 AS my_group,
        x % 2 AS another_group,
        1 AS one_big_group
    FROM range(1, 101) t(x)
)
FROM dynamic_aggregates_any_cte_any_func(
    ['another_group', 'one_big_group'], [], ['id', 'my_group']
);
```

another_group	one_big_group	any_func(any_cte.id)	any_func(any_cte.my_group)
0	1	502.0	200.0
1	1	490.0	200.0

Understanding Version 2

Instead of querying the very boldly named `example` table, we query the possibly more generically named `any_cte`. Note that `any_cte` has a different schema than our prior example – the columns in `any_cte` can be anything! When the macro is created, `any_cte` doesn't even exist. When the macro is executed, it searches for a table-like object named `any_cte`, and it was defined in the CTE as the macro was called.

Similarly, `any_func` does not exist initially. It only needs to be created (or recreated) at some point before the macro is executed. Its only requirements are to be an aggregate function that operates on a single column.

> FUNCTION and MACRO are synonyms in DuckDB and can be used interchangeably!

Takeaways from Version 2

A macro can act on any arbitrary table by using a CTE at the time it is called. This makes our macro far more reusable – it can work on any table! Not only that, but any custom aggregate function can be used.

Look how far we have stretched SQL – we have made a truly reusable SQL function! The table is dynamic, the grouping columns are dynamic, the aggregated columns are dynamic, and so is the aggregate function. Our daily gymnastics stretches have paid off. However, stay tuned for a way to achieve similar results with a simpler approach in a future post.

Custom Summaries for Any Dataset

Next we have a truly production-grade example! This query powers a portion of the MotherDuck Web UI's Column Explorer component. Hamilton Ulmer led the creation of this component and is the author of this query as well! The purpose of the Column Explorer, and this query, is to get a high-level overview of the data in all columns within a dataset as quickly and easily as possible.

DuckDB has a built-in SUMMARIZE keyword that can calculate similar metrics across an entire table. However, for larger datasets, SUMMARIZE can take a couple of seconds to load. This query provides a custom summarization capability that can be tailored to the properties of your data that you are most interested in.

Traditionally, databases required that every column be referred to explicitly, and work best when data is arranged in separate columns. This query takes advantage of DuckDB's ability to apply functions to all columns at once, its ability to UNPIVOT (or stack) columns, and its STRUCT data type to store key/value pairs. The result is a clean, pivoted summary of all the rows and columns in a table.

Let's take a look at the entire function, then break it down piece by piece.

This example dataset comes from Hugging Face, which hosts DuckDB-accessible Parquet files for many of their datasets. First, we create a local table populated from this remote Parquet file.

Creation

```
CREATE OR REPLACE TABLE spotify_tracks AS
    FROM 'https://huggingface.co/datasets/maharshipandya/spotify-tracks-
    dataset/resolve/refs%2Fconvert%2Fparquet/default/train/0000.parquet?download=true';
```

Then we create and execute our custom_summarize macro. We use the same any_cte trick from above to allow this to be reused on any query result or table.

```
CREATE OR REPLACE MACRO custom_summarize() AS TABLE (
    WITH metrics AS (
        FROM any_cte
        SELECT
            {
                name: first(alias(COLUMNS(*))),
                type: first(typeof(COLUMNS(*))),
                max: max(COLUMNS(*))::VARCHAR,
                min: min(COLUMNS(*))::VARCHAR,
                approx_unique: approx_count_distinct(COLUMNS(*)),
                nulls: count(*) - count(COLUMNS(*)),
            }
    ), stacked_metrics AS (
        UNPIVOT metrics
        ON COLUMNS(*)
    )
    SELECT value.* FROM stacked_metrics
);
```

Execution

The `spotify_tracks` dataset is effectively renamed to `any_cte` and then summarized.

```
WITH any_cte AS (FROM spotify_tracks)
FROM custom_summarize();
```

The result contains one row for every column in the raw dataset, and several columns of summary statistics.

name	type	max	min	approx_unique	nulls
Unnamed: 0	BIGINT	113999	0	114089	0
track_id	VARCHAR	7zz7iNGIWhmfFE7zlXkMma	0000vdREvCVMxbQTkS888c	89815	0
artists	VARCHAR	▯▯Ryuzo	!nvite	31545	1
album_name	VARCHAR	▯▯▯ ▯▯ ▯▯▯ Pt. 4 Original Television Soundtrack	! ! ! ! ! Whispers ! ! ! ! !	47093	1
track_name	VARCHAR	▯▯▯▯ ▯▯	!I'll Be Back!	72745	1
popularity	BIGINT	100	0	99	0
duration_ms	BIGINT	5237295	0	50168	0
explicit	BOOLEAN	true	false	2	0
danceability	DOUBLE	0.985	0.0	1180	0
energy	DOUBLE	1.0	0.0	2090	0
key	BIGINT	11	0	12	0
loudness	DOUBLE	4.532	-49.531	19436	0
mode	BIGINT	1	0	2	0
speechiness	DOUBLE	0.965	0.0	1475	0
acousticness	DOUBLE	0.996	0.0	4976	0
instrumentalness	DOUBLE	1.0	0.0	5302	0
liveness	DOUBLE	1.0	0.0	1717	0
valence	DOUBLE	0.995	0.0	1787	0
tempo	DOUBLE	243.372	0.0	46221	0
time_signature	BIGINT	5	0	5	0
track_genre	VARCHAR	world-music	acoustic	115	0

So how was this query constructed? Let's break down each CTE step by step.

Step by Step Breakdown

Metrics CTE

First let's have a look at the `metrics` CTE and the shape of the data that is returned:

```
FROM any_cte
SELECT
    {
        name: first(alias(COLUMNS(*))),
        type: first(typeof(COLUMNS(*))),
        max: max(COLUMNS(*))::VARCHAR,
        min: min(COLUMNS(*))::VARCHAR,
```

```
        approx_unique: approx_count_distinct(COLUMNS(*)),
        nulls: count(*) - count(COLUMNS(*)),
    };
```

main.struct_pack(”name” := first(alias(subset.”Unnamed: 0”)), ...	main.struct_pack(”name” := first(alias(subset.track_id)), ...	...	main.struct_pack(”name” := first(alias(subset.time_ signature)), ...	main.struct_pack(”name” := first(alias(subset.track_ genre)), ...
{'name': Unnamed: 0, 'type': BIGINT, 'max': 113999, 'min': 0, 'approx_unique': 114089, 'nulls': 0}	{'name': track_id, 'type': VARCHAR, 'max': 7zz7iNGIWhmfFE7zlXkMma, 'min': 0000vdREvCVMxbQTkS888c, 'approx_unique': 89815, 'nulls': 0}	...	{'name': time_signature, 'type': BIGINT, 'max': 5, 'min': 0, 'approx_unique': 5, 'nulls': 0}	{'name': track_genre, 'type': VARCHAR, 'max': world-music, 'min': acoustic, 'approx_ unique': 115, 'nulls': 0}

This intermediate result maintains the same number of columns as the original dataset, but only returns a single row of summary statistics. The names of the columns are truncated due to their length. The default naming of `COLUMNS` expressions will be improved in DuckDB 0.10.1, so names will be much cleaner!

The data in each column is organized into a STRUCT of key-value pairs. You can also see that a clean name of the original column is stored within the STRUCT thanks to the use of the `alias` function. While we have calculated the summary statistics, the format of those statistics is difficult to visually interpret.

The query achieves this structure using the `COLUMNS(*)` expression to apply multiple summary metrics to all columns, and the `{...}` syntax to create a STRUCT. The keys of the struct represent the names of the metrics (and what we want to use as the column names in the final result). We use this approach since we want to transpose the columns to rows and then split the summary metrics into their own columns.

`stacked_metrics` CTE

Next, the data is unpivoted to reshape the table from one row and multiple columns to two columns and multiple rows.

```
UNPIVOT metrics
ON COLUMNS(*);
```

name	value
main.struct_ pack(”name” := first(alias(spotify_ tracks.”Unnamed: 0”)), ...	{'name': Unnamed: 0, 'type': BIGINT, 'max': 113999, 'min': 0, 'approx_unique': 114089, 'nulls': 0}
main.struct_ pack(”name” := first(alias(spotify_ tracks.track_id)), ...	{'name': track_id, 'type': VARCHAR, 'max': 7zz7iNGIWhmfFE7zlXkMma, 'min': 0000vdREvCVMxbQTkS888c, 'approx_ unique': 89815, 'nulls': 0}
...	...

name	value
main.struct_pack(”name” := first(alias(spotify_tracks.time_signature)), ...	{'name': time_signature, 'type': BIGINT, 'max': 5, 'min': 0, 'approx_unique': 5, 'nulls': 0}
main.struct_pack(”name” := first(alias(spotify_tracks.track_genre)), ...	{'name': track_genre, 'type': VARCHAR, 'max': world-music, 'min': acoustic, 'approx_unique': 115, 'nulls': 0}

By unpivoting on `COLUMNS(*)`, we take all columns and pivot them downward into two columns: one for the auto-generated `name` of the column, and one for the `value` that was within that column.

Return the Results

The final step is the most gymnastics-like portion of this query. We explode the `value` column's struct format so that each key becomes its own column using the `STRUCT.*` syntax. This is another way to make a query less reliant on column names – the split occurs automatically based on the keys in the struct.

```
SELECT value.*
FROM stacked_metrics;
```

We have now split apart the data into multiple columns, so the summary metrics are nice and interpretable.

name	type	max	min	approx_unique	nulls
Unnamed: 0	BIGINT	113999	0	114089	0
track_id	VARCHAR	7zz7iNGIWhmfFE7zlXkMma	0000vdREvCVMxbQTkS888c	89815	0
artists	VARCHAR	▯▯Ryuzo	!nvite	31545	1
album_name	VARCHAR	▯▯▯ ▯▯ ▯▯▯ Pt. 4 Original Television Soundtrack	! ! ! ! ! Whispers ! ! ! ! !	47093	1
track_name	VARCHAR	▯▯▯▯ ▯▯	!I'll Be Back!	72745	1
popularity	BIGINT	100	0	99	0
duration_ms	BIGINT	5237295	0	50168	0
explicit	BOOLEAN	true	false	2	0
danceability	DOUBLE	0.985	0.0	1180	0
energy	DOUBLE	1.0	0.0	2090	0
key	BIGINT	11	0	12	0
loudness	DOUBLE	4.532	-49.531	19436	0
mode	BIGINT	1	0	2	0
speechiness	DOUBLE	0.965	0.0	1475	0
acousticness	DOUBLE	0.996	0.0	4976	0
instrumentalness	DOUBLE	1.0	0.0	5302	0
liveness	DOUBLE	1.0	0.0	1717	0
valence	DOUBLE	0.995	0.0	1787	0

name	type	max	min	approx_unique	nulls
tempo	DOUBLE	243.372	0.0	46221	0
time_signature	BIGINT	5	0	5	0
track_genre	VARCHAR	world-music	acoustic	115	0

Conclusion

We have shown that it is now possible to build reusable SQL macros in a highly flexible way. You can now build a macro that:

- Operates on any dataset
- Selects any columns
- Groups by any columns
- Aggregates any number of columns with any function.

Phew!

Along the way we have covered some useful tricks to have in your toolbox:

- Applying a macro to any dataset using a CTE
- Selecting a dynamic list of columns by combining the `COLUMNS` expression with a lambda and the `list_contains` function
- Passing in an aggregate function as a string using `list_aggregate`
- Applying any custom aggregation function within a macro
- Making list parameters optional using `OR len(list_parameter) = 0`
- Using the `alias` function with a `COLUMNS` expression to store the original name of all columns
- Summarizing all columns and then transposing that summary using `UNPIVOT` and `STRUCT.*`

The combination of these friendly SQL features is more powerful than using any one individually. We hope that we have inspired you to take your SQL to new limits!

As always, we welcome your feedback and suggestions. We also have more flexibility in mind that will be demonstrated in future posts. Please share the times you have stretched SQL in imaginative ways!

Happy analyzing!

Dependency Management in DuckDB Extensions

Publication date: 2024-03-22

Author: Sam Ansmink

TL;DR: While core DuckDB has zero external dependencies, building extensions with dependencies is now very simple, with built-in support for vcpkg, an open-source package manager with support for over 2000 C/C++ packages. Interested in building your own? Check out the extension template.

Introduction

Ever since the birth of DuckDB, one of its main pillars has been its strict no-external-dependencies philosophy. Paraphrasing this 2019 SIGMOD paper on DuckDB: *To achieve the requirement of having practical "embeddability" and portability, the database needs to run in whatever environment the host does. Dependencies on external libraries (e.g., openssh) for either compile- or runtime have been found to be problematic.*

In this blog post, we will cover how DuckDB manages to stay true to this philosophy without forcing DuckDB developers down the path of complete abstinence. Along the way, we will show practical examples of how external dependencies are possible, and how you can use this when creating your own DuckDB extension.

The Difficulties of Complete Abstinence

Having no external dependencies is conceptually very simple. However, in a real-world system with real-world requirements, it is difficult to achieve. Many features require complex implementations of protocols and algorithms, and many high-quality libraries exist that implement them. What this means for DuckDB (and most other systems, for that matter) is that there are basically three options for handling requirements with potential external dependencies:

1. Inlining external code
2. Rewriting the external dependency
3. Breaking the no-dependency rule

The first two options are pretty straightforward: to avoid depending on some external software, just make it part of the codebase. By doing so, the unpredictable nature of depending on somebody else is now eliminated! DuckDB has applied both inlining and rewriting to prevent dependencies. For example, the Postgres parser and MbedTLS libraries are inlined into DuckDB, whereas the S3 support is provided using a custom implementation of the AWS S3 protocol.

Okay, great – problem solved, right? Well, not so fast. Most people with some software engineering experience will realize that both inlining and rewriting come with serious drawbacks. The most fundamental issue is probably related to code maintenance. Every significant piece of software needs some level of maintenance. Ranging from fixing bugs to dealing with changing (build) environments or requirements, code will need to be modified to stay functional and relevant. When inlining/rewriting dependencies, this also copies over the maintenance burden.

For DuckDB, this historically meant that for each dependency, very careful consideration was made to balance the increased maintenance burden against the necessity of dependency. Including a dependency meant the responsibility of maintaining it, so this decision was never taken lightly. This works well in many cases and has the added benefit of forcing developers to think critically about including a dependency and not mindlessly bolt on library after library. However, for some dependencies, this just doesn't work. Take, for example, the SDKs of large cloud providers. They tend to be pretty massive, very frequently updated, and packed with arguably essential functionality

for an increasingly mature analytical database. This leaves an awkward choice: either not provide these essential features or break the no-dependency rule.

DuckDB Extensions

This is where extensions come in. Extensions provide an elegant solution to the dilemma of dependencies by allowing fine-grained breakage of the no-dependency rule. Moving dependencies out of DuckDB's core into extensions, the core codebase can remain, and does remain, dependency-free. This means that DuckDB's "Practical embeddability and portability" remains unthreatened. On the other hand, DuckDB can still provide features that inevitably require depending on some 3rd party library. Furthermore, by moving dependencies to extensions, each extension can have different levels of exposure to instability from dependencies. For example, some extensions may choose to depend only on highly mature, stable libraries with good portability, whereas others may choose to include more experimental dependencies with limited portability. This choice is then forwarded to the user by allowing them to choose which extension to use.

At DuckDB, this realization of the importance of extensions and its relation to the no-dependency rule came very early, and consequently extensibility has been ingrained into DuckDB's design since its early days. Today, many parts of DuckDB can be extended. For example, you can add functions (table, scalar, copy, aggregation), filesystems, parsers, optimizer rules, and much more. Many new features that are added to DuckDB are added in extensions and are grouped by either functionality or by set of dependencies. Some examples of extensions are the SQLite extension for reading/writing to/from SQLite files or the Spatial extension which offers support for a wide range of geospatial processing features. DuckDB's extensions are distributed as loadable binaries for most major platforms (including DuckDB-Wasm), allowing loading and installing extensions with two simple SQL statements:

```
INSTALL spatial;
LOAD spatial;
```

For most core extensions maintained by the DuckDB team, there is even an auto-install and auto-load feature which will detect the required extensions for a SQL statement and automatically install and load them. For a detailed description of which extensions are available and how to use them, check out the docs.

Dependency Management

So far, we've seen how DuckDB avoids external dependencies in its core codebase by moving them out of the core repository into extensions. However, we're not out of the woods yet. As DuckDB is written in C++, the most natural way to write extensions is C++. In C++, though, there is no standard tooling like a package manager and the answer to the question of how to do dependency management in C++ has been, for many years: *"Through much pain and anguish."* Given DuckDB's focus on portability and support for many platforms, managing dependencies manually is not feasible: dependencies generally are built from source, with each their own intricacies requiring special build flags and configuration for different platforms. With a growing ecosystem of extensions, this would quickly turn into an unmaintainable mess.

Fortunately, much has changed in the C++ landscape over the past few years. Today, good dependency managers do exist. One of them is Microsoft's vcpkg. It has become a highly notable player among C++ dependency managers, as proven by its 20k+ GitHub stars and native support from CLion and Visual Studio. vcpkg contains over 2000 dependencies such as Apache Arrow, yyjson, and various cloud provider SDKs.

For anyone who has ever used a package manager, using vcpkg will feel quite natural. Dependencies are specified in a `vcpkg.json` file, and vcpkg is hooked into the build system. Now, when building, vcpkg ensures that the dependencies specified in the `vcpkg.json` are built and available. vcpkg supports integration with multiple build systems, with a focus on its seamless CMake integration.

Using vcpkg with DuckDB

Now that we covered DuckDB extensions and vcpkg, we have shown how DuckDB can manage dependencies without sacrificing portability, maintainability and stability more than necessary. Next, we'll make things a bit more tangible by looking at one of DuckDB's extensions and how it uses vcpkg to manage its dependencies.

Example: Azure extension

The Azure extension provides functionality related to Microsoft Azure, one of the major cloud providers. DuckDB's Azure extension depends on the Azure C++ SDK to support reading directly from Azure Storage. To do so it adds a custom filesystem and secret type, which can be used to easily query from authenticated Azure containers:

```
CREATE SECRET az1 (
    TYPE azure,
    CONNECTION_STRING '<redacted>'
);
SELECT column_a, column_b
FROM 'az://my-container/some-file.parquet';
```

To implement these features, the Azure extension depends on different parts of the Azure SDK. These are specified in the Azure extensions `vcpkg.json`:

```
{
  "dependencies": [
    "azure-identity-cpp",
    "azure-storage-blobs-cpp",
    "azure-storage-files-datalake-cpp"
  ]
}
```

Then, in the Azure extension's `CMakelists.txt` file, we find the following lines:

```
find_package(azure-identity-cpp CONFIG)
find_package(azure-storage-blobs-cpp CONFIG)
find_package(azure-storage-files-datalake-cpp CONFIG)

target_link_libraries(${EXTENSION_NAME} Azure::azure-identity Azure::azure-storage-blobs
Azure::azure-storage-files-datalake)
target_include_directories(${EXTENSION_NAME} PRIVATE Azure::azure-identity Azure::azure-storage-blobs
Azure::azure-storage-files-datalake)
```

And that's basically it! Every time the Azure extension is built, vcpkg will be called first to ensure `azure-identity-cpp`, `azure-storage-blobs-cpp` and `azure-storage-files-datalake-cpp` are built using the correct platform-specific flags and available in CMake through `find_package`.

Building Your Own DuckDB Extension

Up until this part, we've focused on managing dependencies from a point-of-view of the developers of core DuckDB contributors. However, all of this applies to anyone who wants to build an extension. DuckDB maintains a C++ Extension Template, which contains all the necessary build scripts, CI/CD pipeline and vcpkg configuration to build, test and deploy a DuckDB extension in minutes. It can automatically build the loadable extension binaries for all available platforms, including Wasm.

Setting up the Extension Template

To demonstrate how simple this process is, let's go through all the steps of building a DuckDB extension from scratch, including adding a vcpkg-managed external dependency.

Firstly, you will need to install vcpkg:

```
git clone https://github.com/Microsoft/vcpkg.git
./vcpkg/bootstrap-vcpkg.sh
export VCPKG_TOOLCHAIN_PATH=`pwd`/vcpkg/scripts/buildsystems/vcpkg.cmake
```

Then, you create a GitHub repository based on the template by clicking "Use this template".

Now to clone your newly created extension repo (including its submodules) and initialize the template:

```
git clone --recurse-submodules \
    https://github.com/<your_username>/<your_extension_repo>
cd your-extension-repo
./scripts/bootstrap-template.py url_parser
```

Finally, to confirm everything works as expected, run the tests:

```
make test
```

Adding Functionality

In its current state, the extension is, of course, a little boring. Therefore, let's add some functionality! To keep things simple, we'll add a scalar function that parses a URL and returns the scheme. We'll call the function `url_scheme`. We start by adding a dependency to the boost url library in our `vcpkg.json` file:

```
{
  "dependencies": [
    "boost-url"
  ]
}
```

Then, we follow up with changing our `CMakelists.txt` to ensure our dependencies are correctly included in the build.

```
find_package(Boost REQUIRED COMPONENTS url)
target_link_libraries(${EXTENSION_NAME} Boost::url)
target_link_libraries(${LOADABLE_EXTENSION_NAME} Boost::url)
```

Then, in `src/url_parser_extension.cpp`, we remove the default example functions and replace them with our implementation of the `url_scheme` function:

```
inline void UrlParserScalarFun(DataChunk &args, ExpressionState &state, Vector &result) {
  auto &name_vector = args.data[0];
  UnaryExecutor::Execute<string_t, string_t>(
    name_vector, result, args.size(),
    [&](#string_t url) {
          string url_string = url.GetString();
          boost::system::result<boost::urls::url_view> parse_result = boost::urls::parse_uri( url_string
          );
          if (parse_result.has_error() || !parse_result.value().has_scheme()) {
              return string_t();
          }
          string scheme = parse_result.value().scheme();
          return StringVector::AddString(result, scheme);
      });
}

static void LoadInternal(DatabaseInstance &instance) {
  auto url_parser_scalar_function = ScalarFunction("url_scheme", {LogicalType::VARCHAR},
  LogicalType::VARCHAR, UrlParserScalarFun);
  ExtensionUtil::RegisterFunction(instance, url_parser_scalar_function);
}
```

With our extension written, we can run `make` to build both DuckDB and the extension. After the build is finished, we are ready to try out our extension. Since the build process also builds a fresh DuckDB binary with the extension loaded automatically, all we need to do is run `./build/release/duckdb`, and we can use our newly added scalar function:

```
SELECT url_scheme('https://github.com/duckdb/duckdb');
```

Finally, as we are well-behaved developers, we add some tests by overwriting the default test `test/sql/url_parser.test` with:

```
require url_parser

# Confirm the extension works
query I
SELECT url_scheme('https://github.com/duckdb/duckdb')
----
https

# On parser errors or not finding a scheme, the result is also an empty string
query I
SELECT url_scheme('not:\a/valid_url')
----
(empty)
```

Now all that's left to do is confirm everything works as expected with `make test`, and push these changes to the remote repository. Then, GitHub Actions will take over and ensure the extension is built for all of DuckDB's supported platforms.

For more details, check out the template repository. Also, the example extension we built in this blog is published on GitHub. Note that in the demo, the Wasm and MinGW builds have been disabled due to outstanding issues with the boost-url dependency for building on these platforms. As these issues are fixed upstream, re-enabling their builds for the extension is very simple. Of course, as the author of this extension, it could make a lot of sense to fix these compile issues yourself in vcpkg and fix them not only for this extension, but for the whole open-source community!

Conclusion

In this blog post, we've explored DuckDB's journey towards managing dependencies in its extension ecosystem while upholding its core philosophy of zero external dependencies. By leveraging the power of extensions, DuckDB can maintain its portability and embeddability while still providing essential features that require external dependencies. To simplify managing dependencies, Microsoft's vcpkg is integrated into DuckDB's extension build systems both for DuckDB-maintained extension and third-party extensions.

If this blog post sparked your interest in creating your own DuckDB extension, check out the C++ Extension Template, the DuckDB docs on extensions, and the very handy duckdb-extension-radar repository that tracks public DuckDB extensions. Additionally, DuckDB has a Discord server where you can ask for help on extensions or anything DuckDB-related in general.

42.parquet – A Zip Bomb for the Big Data Age

Publication date: 2024-03-26

Author: Hannes Mühleisen

TL;DR: A 42 kB Parquet file can contain over 4 PB of data.

Apache Parquet has become the de-facto standard for tabular data interchange. It is greatly superior to its scary cousin CSV by using a binary, columnar and *compressed* data representation. In addition, Parquet files come with enough metadata so that files can be correctly interpreted without additional information. Most modern data tools and services support reading and writing Parquet files.

However, Parquet files are not without their dangers: For example, corrupt files can crash readers that are not being very careful in interpreting internal offsets and such. But even perfectly valid files can be problematic and lead to crashes and service downtime as we will show below.

A pretty well-known attack on naive firewalls and virus scanners is a Zip Bomb, one famous example being 42.zip, named so because of course 42 is the perfect number and the file is only 42 kilobytes large. This perfectly-valid zip file has a bunch of other zip files in it, which again contain other zip files and so on. Eventually, if one would try to unpack all of that, you would end up with 4 petabytes of data. Big Data indeed.

Parquet files support various methods to compress data. How big of a table can one create with a Parquet file that is only 42 kilobytes large in the spirit of a zip bomb? Let's find out! For reasons of portability, we have implemented our own Parquet reader and writers for DuckDB. It is unavoidable to learn a great deal about the Parquet format when implementing it.

A Parquet file is made up of one or more row groups, which contain columns, which in turn contain so-called pages that contain the actual data in encoded format. Among other encodings, Parquet supports dictionary encoding, where we first have a page with a dictionary, followed by data pages that refer to the dictionary instead of containing plain values. This is more efficient for columns where long values such as categorical strings repeat often, because the dictionary references can be much smaller.

Let's exploit that. We write a dictionary with a single value and refer to it over and over. In our example, we use a single 64-bit integer, the biggest possible value because why not. Then, we refer back to this dictionary entry using the `RLE_DICTIONARY` run-length encoding specified in parquet. The specified encoding is a bit weird because for some reason it combines bit packing and run-length encoding but essentially we can use the biggest run-length possible, which is `2^31-1`, a little over 2 billion. Since the dictionary is tiny (one entry), the value we repeat is 0, referring to the only entry. Including its required metadata headers and footers (like all metadata in Parquet, this is encoded using Thrift), this file is only 133 bytes large. 133 bytes to represent 2 billion 8-byte integers is not too bad, even if they're all the same.

But we can go up from there. Columns can contain multiple pages referring to *the same* dictionary, so we can just repeat our data page over and over, each time only adding 31 bytes to the file, but 2 billion values to the table the file represents. We can also use another trick to blow up the data size: as mentioned, Parquet files contain one or more row groups, those are stored in a Thrift footer at the end of the file. Each column in this row group contains byte offsets (`data_page_offset` and friends) into the file where the pages for the columns are stored. Nothing keeps us from adding multiple row groups that *all refer to the same byte offset*, the one where we stored our slightly mischievous dictionary and data pages. Each row group we add logically repeats all the pages. Of course, adding row groups also requires metadata storage, so there is some sort of trade-off between adding pages (2 billion values) and row groups (2x whatever other row group it duplicates).

With some fiddling, we found that if we repeat the data page 1000 times and repeat the row group 290 times, we end up with a Parquet file that is 42 kilobytes large, yet contains *622 trillion* values (622,770,257,630,000 to be exact). If one would materialize this table in memory, it would require over *4 petabytes* of memory, finally a real example of Big Data, coincidentally roughly the same size as the original `42.zip` mentioned above.

We've made the script that we use to generate this file available as well, we hope it can be used to test Parquet readers better. We hope to have shown that Parquet files can be considered harmful and should certainly not be shoved into some pipeline without being extra careful. And while DuckDB *can* read data from our file (e.g., with a `LIMIT`), if you would make it read through it all, you better get some coffee.

No Memory? No Problem. External Aggregation in DuckDB

Publication date: 2024-03-29

Author: Laurens Kuiper

TL;DR: Since the 0.9.0 release, DuckDB's fully parallel aggregate hash table can efficiently aggregate over many more groups than fit in memory.

Most grouped aggregation queries yield just a few output rows. For example, "How many flights departed from each European capital in the past ten years?" yields one row per European capital, even if the table containing all the flight information has millions of rows. This is not always the case, as "How many orders did each customer place in the past ten years?" yields one row per customer, which could be millions, which significantly increases the memory consumption of the query. However, even if the aggregation does not fit in memory, DuckDB can still complete the query.

Not interested in the implementation? Jump straight to the experiments!

Introduction

Around two years ago, we published our first blog post on DuckDB's hash aggregation, titled "Parallel Grouped Aggregation in DuckDB". So why are we writing another blog post now?

Unlike most database systems, which are servers, DuckDB is used in all kinds of environments, which may not have much memory. However, some database queries, like aggregations with many unique groups, require a lot of memory. The laptop I am writing this on has 16 GB of RAM. What if a query needs 20 GB? If this happens:

```
Out of Memory Error: could not allocate block of size X (Y/Z used)
```

The query is aborted. Sadly, we can't download more RAM. But luckily, this laptop also has a fast SSD with 1 TB of storage. In many cases, we don't need all 20 GB of data to be in memory simultaneously, and we can temporarily place some data in storage. If we load it back whenever needed, we can still complete the query. We must be careful to use storage sparingly because despite modern SSDs being fast, they are still much slower than memory.

In a nutshell, that's what this post is about. Since the 0.9.0 release, DuckDB's hash aggregation can process more unique groups than fit in memory by offloading data to storage. In this post, we'll explain how this works. If you want to know what hash aggregation is, how hash collisions are resolved, or how DuckDB's hash table is structured, check out our first blog post on hash aggregation.

Memory Management

Most database systems store persistent data on "pages". Upon request, these pages can be read from the *database file* in storage, put into memory, and written back again if necessary. The common wisdom is to make all pages the same size: This allows pages to be swapped and avoids fragmentation in memory and storage. When the database is started, a portion of memory is allocated and reserved for these pages, called the "buffer pool". The database component that is responsible for managing the buffer pool is aptly called the "buffer manager".

The remaining memory is reserved for short-lived, i.e., *temporary*, memory allocations, such as hash tables for aggregation. These allocations are done differently, which is good because if there are many unique groups, hash tables may need to be very large, so we wouldn't have been able to use the fixed-size pages for that anyway. If we have more temporary data than fits in memory, operators like aggregation have to decide when to selectively write data to a *temporary file* in storage.

... At least, that's the traditional way of doing things. This made little sense for DuckDB. Why should we manage persistent and temporary data so differently? The difference is that *persistent* data should be *persisted*, and *temporary* data should not. Why can't a buffer manager manage both?

DuckDB's buffer manager is not traditional. Most persistent and temporary data is stored on fixed-size pages and managed by the buffer manager. The buffer manager tries to make the best use of your memory. That means we don't reserve a portion of memory for a buffer pool. This allows DuckDB to use all memory for persistent data, not just a portion if that's what's best for your workload. If you're doing large aggregations that need a lot of memory, DuckDB can evict the persistent data from memory to free up space for a large hash table.

Because DuckDB's buffer manager manages *all* memory, both persistent and temporary data, it is much better at choosing when to write temporary data to storage than operators like aggregation could ever be. Leaving the responsibility of offloading to the buffer manager also saves us the effort of implementing reading and writing data to a temporary file in every operator that needs to process data that does not fit in memory.

Why don't buffer managers in other database systems manage temporary data? There are two problems: *Memory Fragmentation* and *Invalid References*.

Memory Fragmentation

Hash tables and other data structures used in query operators don't exactly have a fixed size like the pages used for persistent data. We also don't want to have a lot of pages with variable sizes floating around in memory alongside the pages with a fixed size, as this would cause memory fragmentation.

Ideally, we would use the fixed size for *all* of our memory allocations, but this is not a good idea: Sometimes, the most efficient way to process a query requires allocating, for example, a large array. So, we settled for using a fixed size for *almost all* of our allocations. These short-lived allocations are immediately deallocated after use, unlike the fixed-size pages for persistent data, which are kept around. These allocations do not cause fragmentation with each other because jemalloc, which DuckDB uses for allocating memory when possible, categorizes allocations using size classes and maintains separate arenas for them.

Invalid References

Temporary data usually cannot be written to storage as-is because it often contains pointers. For example, DuckDB implements the string type proposed by Umbra, which has a fixed width. Strings longer than 12 characters are not stored within the string type, but *somewhere else*, and a pointer to this "somewhere else" is stored instead.

This creates a problem when we want to offload data to storage. Let's say this "somewhere else" where strings longer than 12 characters are stored is one of those pages that the buffer manager can offload to storage at any time to free up some memory. If the page is offloaded and then loaded back, it will most likely be loaded into a different address in memory. The pointers that pointed to the long strings are now *invalid* because they still point to the previous address!

The usual way of writing data containing pointers to storage is by *serializing* it first. When reading it back into memory, it has to be *deserialized* again. (De-)serialization can be an expensive operation, hence why data formats like Arrow Flight exist, which try to minimize the cost. However, we can't use Arrow here because Arrow is a column-major layout, but a row-major layout is more efficient for hash tables.

We could create a row-major version of Arrow Flight, but we can just avoid (de-)serialization altogether: We've created a specialized row-major *page layout* that actually uses the old invalidated pointers to *recompute* new valid pointers after reading the data back into memory.

The page layout places fixed-size rows and variable-size data like strings on separate pages. The size of the rows is fixed for a query: After a SQL query is issued, DuckDB creates and executes a query plan. So, even before executing the said plan, we already know which columns we need, their types, and how wide these types are.

As shown in the image below, a small amount of "MetaData" is needed to recompute the pointers. The fixed-size rows are stored in "Row Pages", and variable-size rows in "Var Pages".

Remember that there are pointers within the fixed-size rows pointing to variable-size data. The MetaData describes which fixed-size rows point to which Var Page and the last known address of the Var Page. For example, MetaData 1 describes 5 rows stored in Row Page 1 at offset 0, with variable-size data stored in Var Page 1, which had an address of `0x42`.

Let's say the buffer manager decides to offload Var Page 1. When we request Var Page 1 again, it's loaded into address `0x500`. The pointers within those 5 rows are now invalid. For example, one of the rows contains the pointer `0x48`, which means that it is stored at offset `0x48 - 0x42 = 6` in Var Page 1. We can recompute the pointer by adding the offset to the new address of the page: `0x500 + 6 = 0x506`. Pointer recomputation is done for rows with their strings stored on the same Row and Var Page, so we create a new MetaData every time a Row Page or Var Page is full.

The advantage of pointer recomputation over (de-)serialization is that it can be done lazily. We can check whether the Var Page was offloaded by comparing the pointer in the MetaData with the current pointer to the page. We don't have to recompute the pointers if they are the same.

External Aggregation

Now that we've figured out how to deal with temporary data, it's finally time to talk about hash aggregation. The first big challenge is to perform the aggregation in parallel.

DuckDB uses Morsel-Driven Parallelism parallelize query execution, which essentially means that query operators, such as aggregation, must be parallelism-aware. This differs from plan-driven parallelism, keeping operators unaware of parallelism.

To briefly summarize our first blog post on aggregation: In DuckDB, all active threads have their own thread-local hash table, which they sink input data into. This will keep threads busy until all input data has been read. Multiple threads will likely have the *exact same group* in their hash table. Therefore, the thread-local hash tables must be combined to complete the grouped aggregation. This can be done in parallel by partitioning the hash tables and assigning each thread to combine the data from each partition. For the most part, we still use this same approach. You'll see this in the image below, which illustrates our new implementation.

We call the first phase *Thread-Local Pre-Aggregation*. The input data are *morsels*, chunks of around 100,000 rows. These are assigned to active threads, which sink them into their thread-local hash table until all input data has been read. We use *linear probing* to resolve collisions and *salt* to reduce the overhead of dealing with said collisions. This is explained in our first blog post on aggregation, so I won't repeat it here.

Now that we've explained what *hasn't* changed, we can talk about what *has* changed. The first difference compared to last time is the way that we partition. Before, if we had, for example, 32 threads, each thread would create 32 hash tables, one for each partition. This totals a whopping 1024 hash tables, which did not scale well when even more threads were active. Now, each thread has one hash table, *but the data within each hash table is partitioned*. The data is also stored on the specialized page layout we presented earlier so that it can easily be offloaded to storage.

The second difference is that the hash tables are not *resized* during Thread-Local Pre-Aggregation. We keep the hash tables' size small, reducing the amount of cache misses during this phase. This means that the hash table will be full at some point. When it's full, we reset it and start over. We can do this because we'll finish the aggregation later in the second phase. When we reset the hash table, we "unpin" the pages that store the actual data, which tells our buffer manager it can write them to storage when it needs to free up memory.

Together, these two changes result in a low memory requirement during the first phase. Each thread only needs to keep a small hash table in memory. We may collect a lot of data by filling up the hash table many times, but the buffer manager can offload almost all of it if needed.

For the second phase, *Partition-Wise Aggregation*, the thread-local partitioned data is exchanged, and each thread combines the data of a single partition into a hash table. This phase is mostly the same as before, except that we now sometimes create many more partitions than threads. Why? The hash table for one partition might fit in memory, but 8 threads could be combining a partition simultaneously, and we might not be able to fit 8 partitions in memory. The easy solution to this problem is to *over-partition*. If we make more partitions than threads, for example, 32 partitions, the size of the partitions will be smaller, and the 8 threads will combine only 8 out of the 32 partitions simultaneously, which won't require nearly as much memory.

Experiments

Aggregations that result in only a few unique groups can easily fit in memory. To evaluate our external hash aggregation implementation, we need aggregations that have many unique groups. For this purpose, we will use the H2O.ai database-like ops benchmark, which we've

resurrected, and now maintain. Specifically, we will use the `G1_1e9_2e0_0_0.csv.zst` file, which is 50 GB uncompressed. The source code for the H2O.ai benchmark can be found on GitHub. You can download the file yourself from https://blobs.duckdb.org/data/G1_1e9_2e0_0_0.csv.zst (18.8 GB compressed).

We use the following queries from the benchmark to load the data:

```
SET preserve_insertion_order = false;
CREATE TABLE y (
    id1 VARCHAR, id2 VARCHAR, id3 VARCHAR,
    id4 INTEGER, id5 INTEGER, id6 INTEGER,
    v1 INTEGER, v2 INTEGER, v3 FLOAT);
COPY y FROM 'G1_1e9_2e0_0_0.csv.zst' (FORMAT csv, AUTO_DETECT true);
CREATE TYPE id1ENUM AS ENUM (SELECT id1 FROM y);
CREATE TYPE id2ENUM AS ENUM (SELECT id2 FROM y);
CREATE TABLE x (
    id1 id1ENUM, id2 id2ENUM, id3 VARCHAR,
    id4 INTEGER, id5 INTEGER, id6 INTEGER,
    v1 INTEGER, v2 INTEGER, v3 FLOAT);
INSERT INTO x (SELECT * FROM y);
DROP TABLE IF EXISTS y;
```

The H2O.ai aggregation benchmark consists of 10 queries, which vary in the number of unique groups:

```
-- Query 1: ~100 unique groups
CREATE OR REPLACE TABLE ans AS
SELECT id1, sum(v1) AS v1
FROM x
GROUP BY id1;

-- Query 2: ~10,000 unique groups
CREATE OR REPLACE TABLE ans AS
SELECT id1, id2, sum(v1) AS v1
FROM x
GROUP BY id1, id2;

-- Query 3: ~10,000,000 unique groups
CREATE OR REPLACE TABLE ans AS
SELECT id3, sum(v1) AS v1, avg(v3) AS v3
FROM x
GROUP BY id3;

-- Query 4: ~100 unique groups
CREATE OR REPLACE TABLE ans AS
SELECT id4, avg(v1) AS v1, avg(v2) AS v2, avg(v3) AS v3
FROM x
GROUP BY id4;

-- Query 5: ~1,000,000 unique groups
CREATE OR REPLACE TABLE ans AS
SELECT id6, sum(v1) AS v1, sum(v2) AS v2, sum(v3) AS v3
FROM x
GROUP BY id6;

-- Query 6: ~10,000 unique groups
CREATE OR REPLACE TABLE ans AS
SELECT
    id4,
    id5,
    quantile_cont(v3, 0.5) AS median_v3,
    stddev(v3) AS sd_v3
FROM x
GROUP BY id4, id5;
```

```sql
-- Query 7: ~10,000,000 unique groups
CREATE OR REPLACE TABLE ans AS
SELECT id3, max(v1) - min(v2) AS range_v1_v2
FROM x
GROUP BY id3;

-- Query 8: ~10,000,000 unique groups
CREATE OR REPLACE TABLE ans AS
SELECT id6, v3 AS largest2_v3
FROM (
    SELECT id6, v3, row_number() OVER (
          PARTITION BY id6
          ORDER BY v3 DESC) AS order_v3
    FROM x
    WHERE v3 IS NOT NULL) sub_query
WHERE order_v3 <= 2;

-- Query 9: ~10,000 unique groups
CREATE OR REPLACE TABLE ans AS
SELECT id2, id4, pow(corr(v1, v2), 2) AS r2
FROM x
GROUP BY id2, id4;

-- Query 10: ~1,000,000,000 unique groups
CREATE OR REPLACE TABLE ans AS
SELECT id1, id2, id3, id4, id5, id6, sum(v3) AS v3, count(*) AS count
FROM x
GROUP BY id1, id2, id3, id4, id5, id6;
```

The results on the benchmark page are obtained using the `c6id.metal` AWS EC2 instance. On this instance, all the queries easily fit in memory, and having many threads doesn't hurt performance either. DuckDB only takes 8.58 seconds to complete even the largest query, query 10, which returns 1 billion unique groups. However, many people will not use such a beefy machine to crunch numbers. On my laptop, a 2020 MacBook Pro, some smaller queries will fit in memory, like query 1, but query 10 will definitely not.

The following table is a summary of the hardware used.

Specs	`c6id.metal`	Laptop	Ratio
Memory	256 GB	16 GB	16×
CPU cores	64	8	8×
CPU threads	128	8	16×
Hourly cost	$6.45	$0.00	NaN

Although the CPU cores of the AWS EC2 instance are not directly comparable with those of my laptop, the instance clearly has much more compute power and memory available. Despite the large differences in hardware, DuckDB can complete all 10 queries without a problem:

Query	`c6id.metal`	Laptop	Ratio
1	0.08	0.74	9.25×
2	0.09	0.76	8.44×
3	8.01	156.63	19.55×
4	0.26	2.07	7.96×
5	6.72	145.00	21.58×
6	17.12	19.28	1.13×

Query	c6id.metal	Laptop	Ratio
7	6.33	124.85	19.72×
8	6.53	126.35	19.35×
9	0.32	1.90	5.94×
10	8.58	264.14	30.79×

The runtime of the queries is reported in seconds, and was obtained by taking the median of 3 runs on my laptop using DuckDB 0.10.1. The `c6id.metal` instance results were obtained from the benchmark website. Despite being unable to *fit* all unique groups in my laptop's memory, DuckDB can *compute* all unique groups and return them. The largest query, query 10, takes almost 4.5 minutes to complete. This is over 30× longer than with the beefy `c6id.metal` instance. The large difference is, of course, explained by the large differences in hardware. Interestingly, this is still faster than Spark on the `c6id.metal` instance, which takes 603.05 seconds!

Conclusion

DuckDB is constantly improving its larger-than-memory query processing capabilities. In this blog post, we showed some of the tricks DuckDB uses for spilling and loading data from storage. These tricks are implemented in DuckDB's external hash aggregation, released since 0.9.0. We took the hash aggregation for a spin on the H2O.ai benchmark, and DuckDB could complete all 50 GB queries on a laptop with only 16 GB of memory.

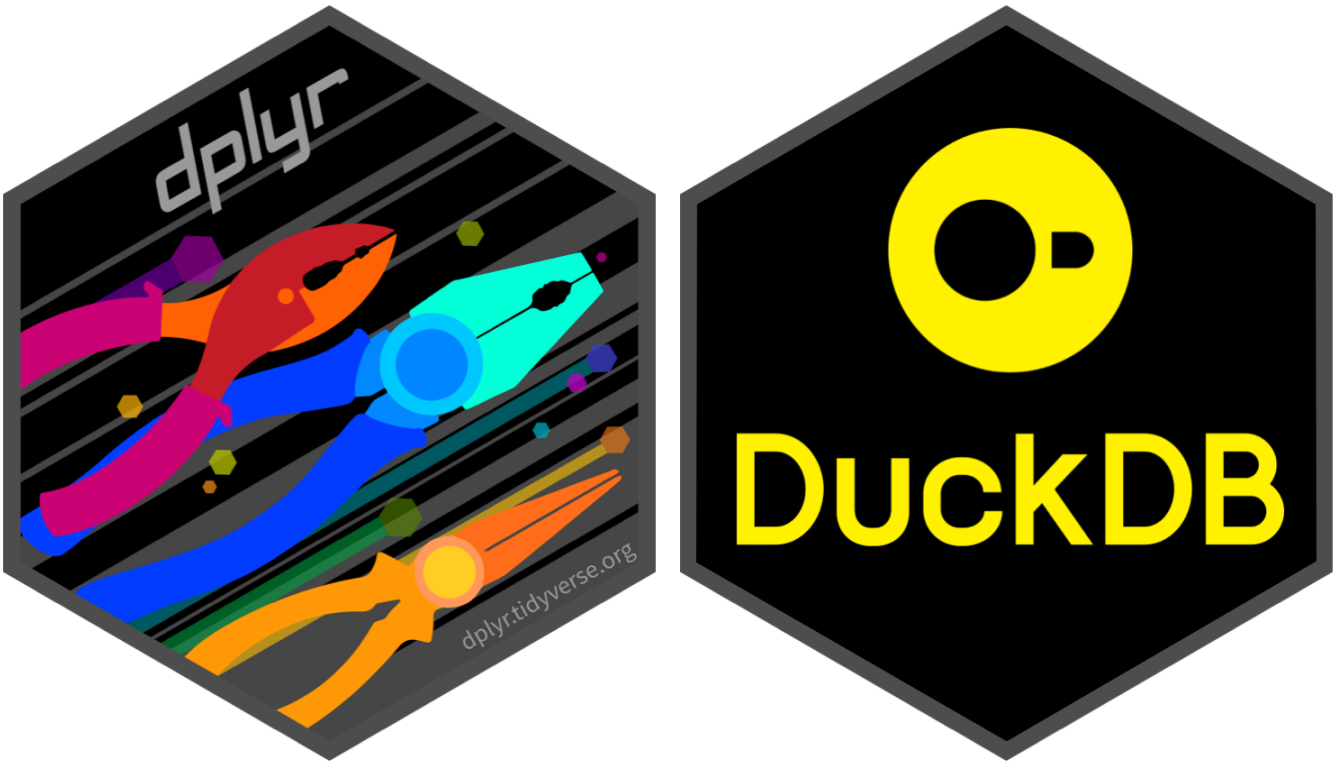
Interested in reading more? Read our paper on external aggregation.

duckplyr: dplyr Powered by DuckDB

Publication date: 2024-04-02

Author: Hannes Mühleisen

TL;DR: The new R package duckplyr translates the dplyr API to DuckDB's execution engine.



> For the duckplyr documentation, visit duckplyr.tidyverse.org.

Background

Wrangling tabular data into a form suitable for analysis can be a challenging task. Somehow, every data set is created differently. Differences between datasets exist in their logical organization of information into rows and columns or in more specific choices like the representation of dates, currency, categorical values, missing data and so on. The task is not simplified by the lack of global consensus on trivial issues like which character to use as a decimal separator. To gain new insights, we also commonly need to combine information from multiple sources, for example by joining two data sets using a common identifier. There are some common recurring operations however, that have been found to be universally useful in reshaping data for analysis. For example, the Structured (English) Query Language, or SQL ("See-Quel") for short describes a set of common operations that can be applied to tabular data like selection, projections, joins, aggregation, sorting, windowing, and more. SQL proved to be a huge success, despite its many warts and many attempts to replace it, it is still the de-facto language for data transformation with a gigantic industry behind it.

```
library("DBI")
con <- dbConnect(...)
df <- dbGetQuery(con, "SELECT something, very, complicated FROM some_table JOIN another_table BY (some_
shared_attribute) GROUP BY group_one, group_two ORDER BY some_column, and_another_column;")
```

A not very ergonomic way of pulling data into R

For data analysts in interactive programming environments like R or Python possibly from within IDEs such as RStudio or Jupyter Notebooks, using SQL to reshape data was never really a natural choice. Sure, sometimes it was required to use SQL to pull data from operational systems as shown above, but when given a choice, analysts much preferred to use the more ergonomic data reshaping facilities provided by those languages. R had built-in data wrangling from the start as part of the language with the data.frame class to represent tabular data. Later on, in 2014, Hadley Wickham defined the logical structure of tabular data for so-called "tidy" data and published the first version of the dplyr package designed to unify and simplify the previously unwieldy R commands to reshape data into a singular, unified and consistent API. In Python-land, the widely popular Pandas project extended Python with a de-facto tabular data representation along with relational-style operators albeit without any attempt at "tidiness".

At some point however, the R and Python data *processing* facilities started to creak under the ever-increasing weight of datasets that people wished to analyze. Datasets quickly grew into millions of rows. For example, one of the early datasets that required special handling was the American Community Survey dataset, because there are just so many Americans. But tools like Pandas and dplyr had been designed for convenience, not necessarily efficiency. For example, they lack the ability to parallelize data reshaping jobs on the now-common multicore processors.

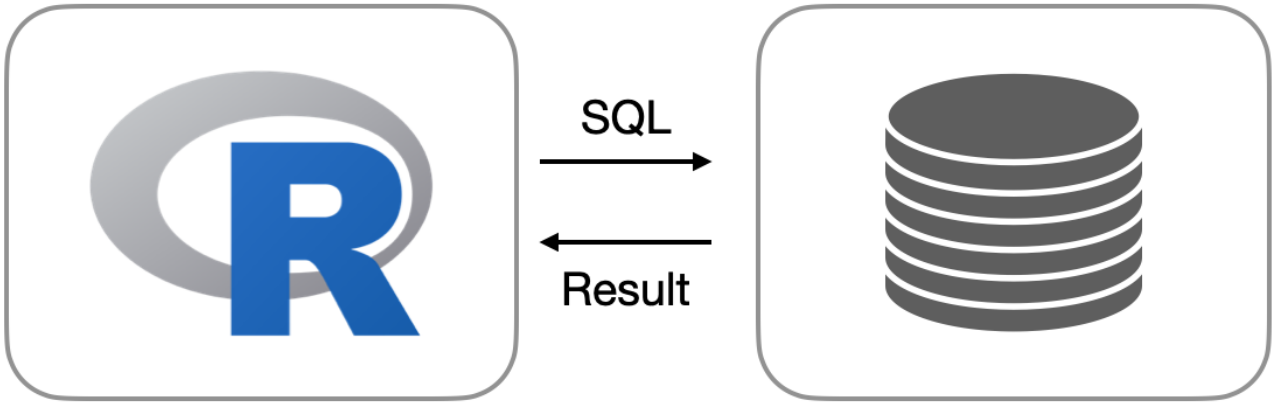
And while there was a whole set of emerging "Big Data" tools, using those from an interactive data analysis environment proved to be a poor developer experience, for example due to multi-second job startup times and very complex setup procedures far beyond the skill set of most data analysts. However, the world of relational data management systems had not stood still in the meantime. Great progress had been made to improve the efficiency of analytical data analysis from SQL: Innovations around columnar data representation, efficient query interpretation or even compilation, and automatic efficient parallelization increased query processing efficiency by several orders of magnitude. Regrettably, those innovations did not find their way into the data analysts toolkit – even as decades passed – due to lack of communication between communities and siloing of innovations into corporate, commercial, and close-source products.

There are two possible ways out of this unfortunate scenario:

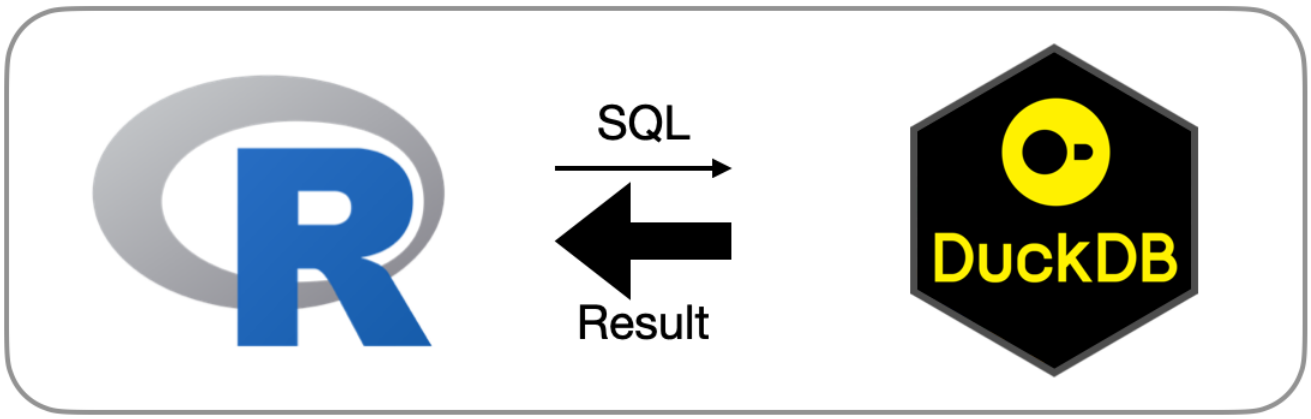
1. improve the data analysis capabilities of R and Python to be able to handle larger datasets through general efficiency improvements, optimization, and parallelization;
2. somehow integrate existing state-of-the-art technology into interactive data analysis environments.

The main issue with approach one is that building a *competitive* analytical query engine from scratch is a multi-million dollar effort requiring a team of highly specialized experts on query engine construction. There are many moving highly complex parts that all have to play together nicely. There are seemingly-obvious questions in query engines that one can get a PhD in data management systems for a solution. Recouping such a massive investment in a space where it is common that tools are built by volunteers in their spare time and released for free is challenging. That being said, there are a few commendable projects in this space like data.table or more recently pola.rs that offer greatly improved performance over older tools.

Approach two is also not without its challenges: State of the art query engine technology is often hidden behind incompatible architectures. For example, the two-tier architecture where a data management system runs on a dedicated database server and client applications use a client protocol to interact with said server is rather incompatible with interactive analysis. Setting up and maintaining a separate database "server" – even on the same computer – is still painful. Moving data back and forth between the analysis environment and the database server has been shown to be quite expensive. Unfortunately, those architectural decisions deeply influence the query engine trade-offs and are therefore difficult to change afterwards.



There has been movement in this space however: One of the stated goals of DuckDB is to unshackle state-of-the-art analytical data management technology from system architecture with its in-process architecture. Simply put, this means there is no separate database server and DuckDB instead runs within a "host" process. This host can be any application that requires data management capabilities or just an interactive data analysis environment like Python or R. Running within the host environment has another massive advantage: Moving data back and forth between the host and DuckDB is very cheap. For R and Python, DuckDB can directly run complex queries on data frames within the analysis environment without any import or conversion steps. Conversely, DuckDB's query results can directly be converted to data frames, greatly reducing the overhead of integrating with downstream libraries for plotting, further analysis or Machine Learning. DuckDB is able to efficiently execute arbitrarily complex relational queries including recursive and correlated queries. DuckDB is able to handle larger-than-memory datasets both in reading and writing but also when dealing with large intermediate results, for example resulting from aggregations with millions of groups. DuckDB has a sophisticated full query optimizer that removes the previously common manual optimization steps. DuckDB also offers persistence, tabular data being stored in files on disk. The tables in those files can be changed, too – while keeping transactional integrity. Those are unheard-of features in interactive data analysis, they are the result of decades of research and engineering in analytical data systems.



One issue remains however, DuckDB speaks SQL. While SQL is a popular language, not all analysts want to express their data transformations in SQL. One of the main issues here is that typically, queries are expressed as strings in R or Python scripts, which are sent to a database system in an opaque way. This means that those queries carry all-or-nothing semantics and it can be challenging to debug problems ("You have an error in your SQL syntax; check the manual…"). APIs like dplyr are often more convenient for the user, they allow an IDE to support things like auto-completion on functions, variable names etc. In addition, the additive nature of the dplyr API allows to build a sequence of data transformation in small steps, which reduces the cognitive load of the analyst considerably compared to writing a hundred-line SQL query. There have been some early experimental attempts to overload R's native data frame API in order to map to SQL databases, but those approaches have been found to be too limited in generality, surprising to users and generally too brittle. A better approach is needed.

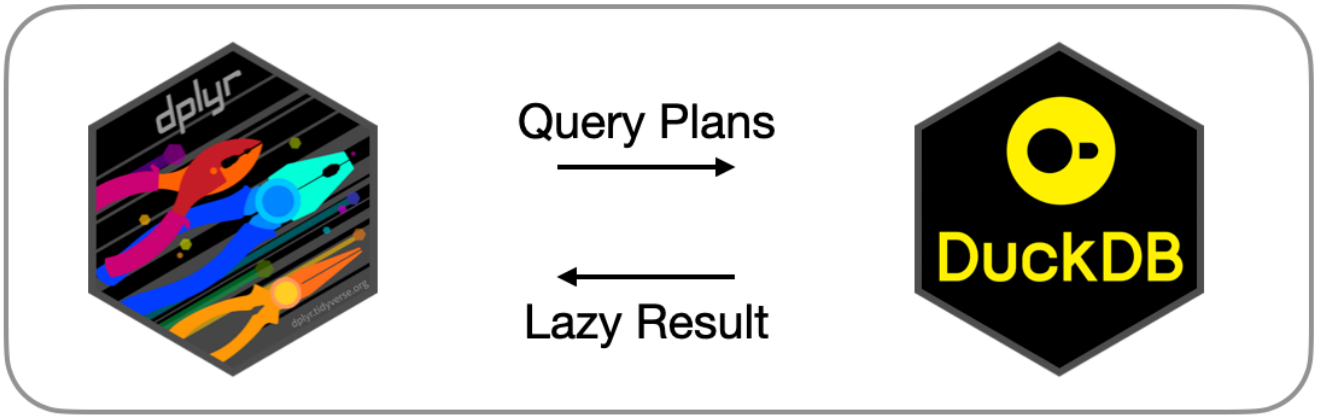
The duckplyr R Package

To address those issues, we have partnered up with the dplyr project team at Posit (formerly RStudio) and cynkra to develop **duckplyr**. duckplyr is a drop-in replacement for dplyr, powered by DuckDB for performance. Duckplyr implements several innovations in the interactive analysis space. First of all, installing duckplyr is just as easy as installing dplyr. DuckDB has been packaged for R as a stand-alone R package that contains the entire data management system code as well as wrappers for R. Both the DuckDB R package as well as duckplyr are available on CRAN, making installation on all major platforms a straightforward:

```
install.packages("duckplyr")
```

Verbs

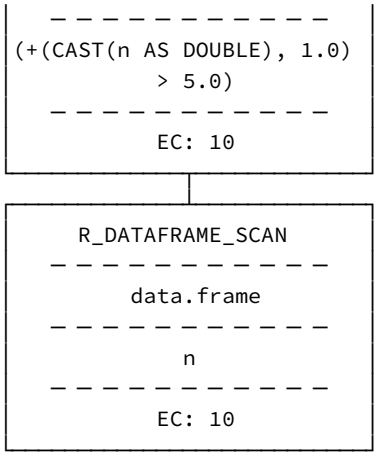
Under the hood, duckplyr translates the sort-of-relational dplyr operations ("verbs") to DuckDB's relational query processing engine. Apart from some naming confusion, there is a mostly straightforward mapping between dplyr's verbs such as select, filter, summarise, etc. and DuckDB's project, filter and aggregate operators. A crucial difference from previous approaches is that duckplyr does not go through DuckDB's SQL interface to create query plans. Instead, duckplyr uses DuckDB's so-called "relational" API to directly construct logical query plans. This API allows to bypass the SQL parser entirely, greatly reducing the difficulty in operator, identifier, constant, and table name escaping that plagues other approaches such as dbplyr.



We have exposed the C++-level relational API to R, so that it is possible to directly construct DuckDB query plans from R. This low-level API is not meant to be used directly, but it is used by duckplyr to transform the dplyr verbs to the DuckDB relational API and thus to query plans. Here is an example:

```
library("duckplyr")
as_duckplyr_df(data.frame(n=1:10)) |>
    mutate(m=n+1) |>
    filter (m > 5) |>
    count() |>
    explain()
```

```
┌─────────────────────────────┐
│         PROJECTION          │
│ — — — — — — — — — — —       │
│             n               │
└─────────────────────────────┘
┌─────────────────────────────┐
│      UNGROUPED_AGGREGATE    │
│ — — — — — — — — — — —       │
│        count_star()         │
└─────────────────────────────┘
┌─────────────────────────────┐
│           FILTER            │
```

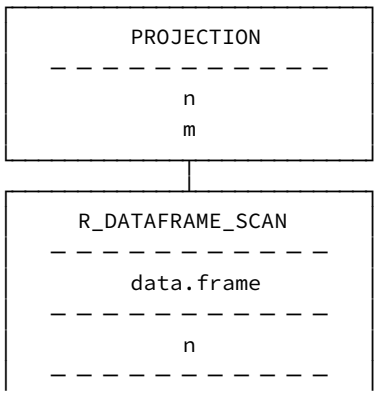


We can see how a sequence of dplyr verbs mutate, filter, and count is "magically" transformed into a DuckDB query plan consisting of a scan, a filter, projections and an aggregate. We can see at the very bottom an `R_DATAFRAME_SCAN` operator is added. This operator directly reads an R data frame as if it were a table in DuckDB, without requiring actual data import. The new verb `explain()` causes DuckDB's logical query plan to be printed so that we can expect what DuckDB intends to execute based on the duckplyr sequence of verbs.

Expressions

An often overlooked yet crucial component of data transformations are so-called expressions. Expressions are (conceptually) scalar transformations of constants and columns from the data that can be used to for example produce derived columns or to transform actual column values to boolean values to be used in filters. For example, one might write an expression like `(amount - discount) * tax` to compute the actual invoiced amount without that amount actually being stored in a column or use an expression like `value > 42` in a filter expression to remove all rows where the value is less than or equal to `42`. Dplyr relies on the base R engine to evaluate expressions with some minor modifications to resolve variable names to columns in the input data. When moving evaluation of expressions over to DuckDB, the process becomes a little bit more involved. DuckDB has its own and independent expression system consisting of a built-in set of functions (e.g., `min`), scalar values and types. To transform R expressions into DuckDB expressions, we use an interesting R feature to capture un-evaluated abstract syntax trees from function arguments. By traversing the tree, we can transform R scalar values into DuckDB scalar values, R function calls into DuckDB function calls, and R-level variable references into DuckDB column references. It should be clear that this transformation cannot be perfect: There are functions in R that DuckDB simply does not support, for example those coming from the myriad of contributed packages. While we are working on expanding the set of supported expressions, there will always be some that cannot be translated. However, in the case of non-translatable expressions, we would still be able to return a result to the user. To achieve this, we have implemented a transparent fall-back mechanism that uses the existing R-level expression evaluation method in the case that an expression cannot be translated to DuckDB's expression language. For example, the following transformation `m = n + 1` can be translated:

```
as_duckplyr_df(data.frame(n=1:10)) |>
    mutate(m=n+1) |>
    explain()
```



```
┌─────────────────────────┐
│          EC: 10         │
└─────────────────────────┘
```

While the following transformation using an inline lambda function cannot (yet):

```
as_duckplyr_df(data.frame(n=1:10)) |>
    mutate(m=(\(x) x+1)(n)) |>
    explain()
```

```
┌─────────────────────────┐
│     R_DATAFRAME_SCAN    │
│ — — — — — — — — — — — — │
│       data.frame        │
│ — — — — — — — — — — — — │
│            n            │
│            m            │
│ — — — — — — — — — — — — │
│          EC: 10         │
└─────────────────────────┘
```

It is a little hard to see (and we are working on improving this), the `explain()` output clearly differs between the two mutate expressions. In the first case, DuckDB computes the `+ 1` as part of the projection operator, in the second case, the translation failed and a fallback was used, leading to the computation happening in the R engine. The upside of automatic fallback is that things "just work". The downside is that there will usually be a performance hit from the fallback due to – for example – the lack of automatic parallelization. We are planning to add a debug mode where users can inspect the translation process and get insight into why translations fail.

Eager vs. Lazy Materialization

Dplyr and Pandas follow an execution strategy known as "eager materialization". Every time an operation is invoked on a data frame, this operation is immediately executed and the result created in memory. This can be problematic. Consider the following example, a ten million row dataset is modified by adding 1 to a column. Then, the `top_n` operation is invoked to retrieve the first ten rows only. Because of eager materialization, the addition operation is executed on ten million rows, the result is created in memory, only for almost all of it to be thrown away immediately because only the first ten rows were requested. Duckplyr solves this problem by using a so-called "lazy materialization" strategy where no action is performed initially but instead the users' intent is being captured. This means that the addition of one to ten million rows will not be performed immediately. The system is instead able to optimize the requested computation and will only perform the addition on the first few rows. Also importantly, the intermediate result of the addition is never actually created in memory, greatly reducing the memory pressure.

However, lazy computation presents a possible integration issue: The result of lazy computation has to be some sort of lazy computation placeholder object, that can be passed to another lazy operation or forced to be evaluated, e.g., via a special print method. However, this would break backwards compatibility with dplyr, where the result of each dplyr operation is a fully materialized data frame itself. This means that those results can be directly passed on to downstream operations like plotting without the plotting package having to be aware of the "lazyness" of the duckplyr result object. To address this, we have creatively used a R feature known as ALTREP. ALTREP allows R objects to have different in-memory representations, and for custom code to be executed whenever those objects are accessed. Duckplyr results are lazy placeholder objects, yes, but they appear to be bog-standard R data frames at the same time. R data frames are essentially named lists of typed vectors with a special row.names attribute. Because DuckDB's lazy query planning already knows the names and types of the resulting table, we can export the names into the lazy data frame. We do not however know the number of rows nor their contents yet. We therefore make both the actual data vectors and the row names vector that contains the data frame length lazy vectors. Those vectors carry a callback that the R engine will invoke whenever downstream code – e.g., plotting code – touches those vectors. The callback will actually trigger computation of the entire pipeline and transformation of the result ot a R data frame. Duckplyr's own operations will refrain from touching those vectors, they instead continue lazily using a special lazy computation object that is also stored in the lazy data frame. This method allows duckplyr to be both lazy and not at the same time, which allows full drop-in replacement with the eagerly evaluated dplyr while keeping the lazy evaluation that is crucial for DuckDB to be able to do a full-query optimization of the various transformation steps.

Here is an example of the duality of the result of duckplyr operations using R's `inspect()` method:

```
dd <- as_duckplyr_df(data.frame(n=1:10)) |> mutate(m=n+1)
.Internal(inspect(dd))
```

```
@12daad988 19 VECSXP g0c2 [OBJ,REF(2),ATT] (len=2, tl=0)
  @13e0c9d60 13 INTSXP g0c0 [REF(4)] DUCKDB_ALTREP_REL_VECTOR n (INTEGER)
  @13e0ca1c0 14 REALSXP g0c0 [REF(4)] DUCKDB_ALTREP_REL_VECTOR m (DOUBLE)
ATTRIB:
  @12817a838 02 LISTSXP g0c0 [REF(1)]
    TAG: @13d80d420 01 SYMSXP g1c0 [MARK,REF(65535),LCK,gp=0x4000] "names" (has value)
    @12daada08 16 STRSXP g0c2 [REF(65535)] (len=2, tl=0)
      @13d852ef0 09 CHARSXP g1c1 [MARK,REF(553),gp=0x61] [ASCII] [cached] "n"
      @13e086338 09 CHARSXP g1c1 [MARK,REF(150),gp=0x61] [ASCII] [cached] "m"
    TAG: @13d80d9d0 01 SYMSXP g1c0 [MARK,REF(56009),LCK,gp=0x4000] "class" (has value)
    @12da9e208 16 STRSXP g0c2 [REF(65535)] (len=2, tl=0)
      @11ff15708 09 CHARSXP g0c2 [MARK,REF(423),gp=0x60] [ASCII] [cached] "duckplyr_df"
      @13d892308 09 CHARSXP g1c2 [MARK,REF(1513),gp=0x61,ATT] [ASCII] [cached] "data.frame"
    TAG: @13d80d1f0 01 SYMSXP g1c0 [MARK,REF(65535),LCK,gp=0x4000] "row.names" (has value)
    @13e0c9970 13 INTSXP g0c0 [REF(65535)] DUCKDB_ALTREP_REL_ROWNAMES
```

We can see that the internal structure of the data frame indeed reflects a data frame, but we can also see the special vectors DUCKDB_ALTREP_REL_VECTOR that hide the un-evaluated data vectors as well as DUCKDB_ALTREP_REL_ROWNAMES that hide the fact that the true dimensions of the data frame are not yet known.

Benchmark: TPC-H Q1

Let's finish with a quick demonstration of duckplyr's performance improvements. We use the data generator from the well known TPC-H benchmark, which is helpfully available as a DuckDB extension. With the "scale factor" of 1, the following DuckDB/R one-liner will generate a data set with a little over 6 million rows and store it in the R data frame named "lineitem":

```
lineitem <- duckdb:::sql("INSTALL tpch; LOAD tpch; CALL dbgen(sf=1); FROM lineitem;")
```

We have transformed the TPC-H benchmark query 1 from its original SQL formulation to dplyr syntax:

```
tpch_01 <- function() {
  lineitem |>
    select(l_shipdate, l_returnflag, l_linestatus, l_quantity, l_extendedprice, l_discount, l_tax) |>
    filter(l_shipdate <= !!as.Date("1998-09-02")) |>
    select(l_returnflag, l_linestatus, l_quantity, l_extendedprice, l_discount, l_tax) |>
    summarise(
      sum_qty = sum(l_quantity),
      sum_base_price = sum(l_extendedprice),
      sum_disc_price = sum(l_extendedprice * (1 - l_discount)),
      sum_charge = sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)),
      avg_qty = mean(l_quantity),
      avg_price = mean(l_extendedprice),
      avg_disc = mean(l_discount),
      count_order = n(),
      .by = c(l_returnflag, l_linestatus)
    ) |>
    arrange(l_returnflag, l_linestatus)
}
```

We can now execute this function with both dplyr and duckplyr and observe the time required to compute the result. "Stock" dplyr takes ca. 400 milliseconds on my MacBook for this query, duckplyr requires only ca 70 milliseconds. Again, this time includes all the magic transforming the sequence of dplyr verbs into a relational operator tree, optimizing said tree, converting the input R data frame into a DuckDB intermediate on-the-fly, and transforming the (admittedly small) result back to a R data frame. Of course, the data set used here is still relatively small and the query is not that complex either, essentially a single grouped aggregation. The differences will be much more pronounced for more complex transformations on larger data sets. duckplyr can also directly access large collections of e.g., Parquet files on storage, and push down filters into those scans, which can also greatly improve performance.

Conclusion

The duckplyr package for R wraps DuckDB's state-of-the-art analytical query processing techniques in a dplyr-compatible API. We have gone to great lengths to ensure compatibility despite switching execution paradigms from eager to lazy and having to translate expressions to a different environment. We continue to work to expand duckplyr's capabilites but would love to hear your experiences trying it out.

Here are two recordings from last year's posit::conf where we present DuckDB for R and duckplyr:

- In-Process Analytical Data Management with DuckDB – posit::conf(2023)
- duckplyr: Tight Integration of duckdb with R and the tidyverse – posit::conf(2023)

Vector Similarity Search in DuckDB

Publication date: 2024-05-03

Author: Max Gabrielsson

TL;DR: This blog post shows a preview of DuckDB's new `vss` extension, which introduces support for HNSW (Hierarchical Navigable Small Worlds) indexes to accelerate vector similarity search.

In DuckDB v0.10.0, we introduced the ARRAY data type, which stores fixed-sized lists, to complement the existing variable-size `LIST` data type.

The initial motivation for adding this data type was to provide optimized operations for lists that can utilize the positional semantics of their child elements and avoid branching as all lists have the same length. Think e.g., the sort of array manipulations you'd do in NumPy: stacking, shifting, multiplying – you name it. Additionally, we wanted to improve our interoperability with Apache Arrow, as previously Arrow's fixed-size list types would be converted to regular variable-size lists when ingested into DuckDB, losing some type information.

However, as the hype for **vector embeddings** and **semantic similarity search** was growing, we also snuck in a couple of distance metric functions for this new ARRAY type: `array_distance`, `array_negative_inner_product` and `array_cosine_distance`

> If you're one of today's lucky 10,000 and haven't heard of word embeddings or vector search, the short version is that it's a technique used to represent documents, images, entities – *data* as high-dimensional *vectors* and then search for *similar* vectors in a vector space, using some sort of mathematical ”distance” expression to measure similarity. This is used in a wide range of applications, from natural language processing to recommendation systems and image recognition, and has recently seen a surge in popularity due to the advent of generative AI and availability of pre-trained models.

This got the community really excited! While we (DuckDB Labs) initially went on record saying that we would not be adding a vector similarity search index to DuckDB as we deemed it to be too far out of scope, we were very interested in supporting custom indexes through extensions in general. Shoot, I've been *personally* nagging on about wanting to plug-in an ”R-Tree” index since the inception of DuckDBs spatial extension! So when one of our client projects evolved into creating a proof-of-concept custom ”HNSW” index extension, we said that we'd give it a shot. And... well, one thing led to another.

Fast forward to now and we're happy to announce the availability of the `vss` vector similarity search extension for DuckDB! While some may say we're late to the vector search party, we'd like to think the party is just getting started!

Alright, so what's in `vss`?

The Vector Similarity Search (VSS) Extension

On the surface, `vss` seems like a comparatively small DuckDB extension. It does not provide any new data types, scalar functions or copy functions, but rather a single new index type: HNSW (Hierarchical Navigable Small Worlds), which is a graph-based index structure that is particularly well-suited for high-dimensional vector similarity search.

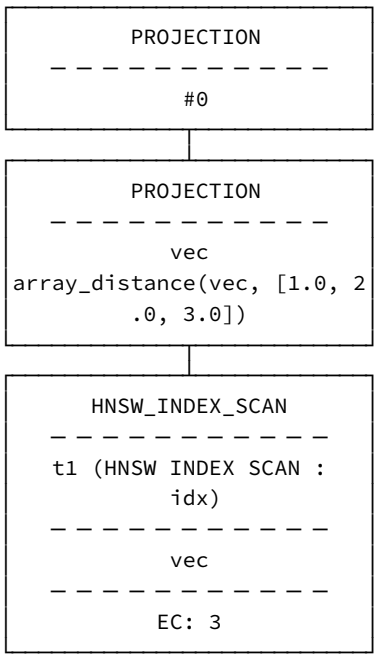
```
-- Create a table with an array column
CREATE TABLE embeddings (vec FLOAT[3]);

-- Create an HNSW index on the column
CREATE INDEX idx ON embeddings USING HNSW (vec);
```

This index type can't be used to enforce constraints or uniqueness like the built-in ART index, and can't be used to speed up joins or index regular columns either. Instead, the HNSW index is only applicable to columns of the ARRAY type containing `FLOAT` elements and will only be used to accelerate queries calculating the ”distance” between a constant `FLOAT` ARRAY and the `FLOAT` ARRAY's in the indexed column, ordered by the resulting distance and returning the top-n results. That is, queries of the form:

```
SELECT *
FROM embeddings
ORDER BY array_distance(vec, [1, 2, 3]::FLOAT[3])
LIMIT 3;
```

will have their logical plan optimized to become a projection over a new HNSW index scan operator, removing the limit and sort altogether. We can verify this by checking the `EXPLAIN` output:

```
EXPLAIN
SELECT *
FROM embeddings
ORDER BY array_distance(vec, [1, 2, 3]::FLOAT[3])
LIMIT 3;
```



You can pass the HNSW index creation statement a `metric` parameter to decide what kind of distance metric to use. The supported metrics are `l2sq`, `cosine` and `inner_product`, matching the three built-in distance functions: `array_distance`, `array_cosine_distance` and `array_negative_inner_product`. The default is `l2sq`, which uses Euclidean distance (`array_distance`):

```
CREATE INDEX l2sq_idx ON embeddings USING HNSW (vec)
WITH (metric = 'l2sq');
```

To use cosine distance (`array_cosine_distance`):

```
CREATE INDEX cos_idx ON embeddings USING HNSW (vec)
WITH (metric = 'cosine');
```

To use inner product (`array_negative_inner_product`):

```
CREATE INDEX ip_idx ON embeddings USING HNSW (vec)
WITH (metric = 'ip');
```

Implementation

The `vss` extension is based on the `usearch` library, which provides a flexible C++ implementation of the HNSW index data structure boasting very impressive performance benchmarks. While we currently only use a subset of all the functionality and tuning options provided by `usearch`, we're excited to explore how we can leverage more of its features in the future. So far we're mostly happy that it aligns so nicely with DuckDB's development ethos. Much like DuckDB itself, `usearch` is written in portable C++11 with no external dependencies and released under a permissive license, making it super smooth to integrate into our extension build and distribution pipeline.

Limitations

The big limitation as of now is that the HNSW index can only be created in in-memory databases, unless the `SET hnsw_enable_experimental_persistence = <bool>`{:.language-sql .highlight} configuration parameter is set to `true`. If this parameter is not set, any attempt to create an HNSW index in a disk-backed database will result in an error message, but if the parameter is set, the index will not only be created in memory, but also persisted to disk as part of the DuckDB database file during checkpointing. After restarting or loading a database file with a persisted HNSW index, the index will be lazily loaded back into memory whenever the associated table is first accessed, which is significantly faster than having to re-create the index from scratch.

The reasoning for locking this feature behind an experimental flag is that we still have some known issues related to persistence of custom indexes that we want to address before enabling it by default. In particular, WAL recovery is not yet properly implemented for custom indexes, meaning that if a crash occurs or the database is shut down unexpectedly while there are uncommited changes to a HNSW-indexed table, you can end up with data loss or corruption of the index. While it is technically possible to recover from a unexpected shutdown manually by first starting DuckDB separately, loading the `vss` extension and then ATTACHing the database file, which ensures that the HNSW index functionality is available during WAL-playback, you should not rely on this for production workloads.

We're actively working on addressing this and other issues related to index persistence, which will hopefully make it into DuckDB v0.10.3, but for now we recommend using the HNSW index in in-memory databases only.

At runtime however, much like the ART the HNSW index must be able to fit into RAM in its entirety, and the memory allocated by the HNSW at runtime is allocated ”outside” of the DuckDB memory management system, meaning that it wont respect DuckDB's `memory_limit` configuration parameter.

Another current limitation with the HNSW index so far are that it only supports the `FLOAT` (a 32-bit, single-precision floating point) type for the array elements and only distance metrics corresponding to the three built in distance functions, `array_distance`, `array_negative_inner_product` and `array_cosine_distance`. But this is also something we're looking to expand upon in the near future as it is much less of a technical limitation and more of a ”we haven't gotten around to it yet” limitation.

Conclusion

The `vss` extension for DuckDB is a new extension that adds support for creating HNSW indexes on fixed-size list columns in DuckDB, accelerating vector similarity search queries. The extension can currently be installed on DuckDB v0.10.2 on all supported platforms (including Wasm!) by running `INSTALL vss; LOAD vss`. The `vss` extension treads new ground for DuckDB extensions by providing a custom index type and we're excited to refine and expand on this functionality going forward.

While we're still working on addressing some of the limitations above, particularly those related to persistence (and performance), we still really want to share this early version the `vss` extension as we believe this will open up a lot of cool opportunities for the community. So make sure to check out the `vss` extension documentation for more information on how to work with this extension!

This work was made possible by the sponsorship of a DuckDB Labs customer! If you are interested in similar work for specific capabilities, please reach out to DuckDB Labs. Alternatively, we're happy to welcome contributors! Please reach out to the DuckDB Labs team over on Discord or on the `vss` extension GitHub repository to keep up with the latest developments.

Access 150k+ Datasets from Hugging Face with DuckDB

Publication date: 2024-05-29

Authors: The Hugging Face and DuckDB teams

TL;DR: DuckDB can now read data from Hugging Face via the `hf://` prefix.

We are excited to announce that we added support for `hf://` paths in DuckDB, providing access to more than 150,000 datasets for artificial intelligence. We worked with Hugging Face to democratize the access, manipulation, and exploration of datasets used to train and evaluate AI models.

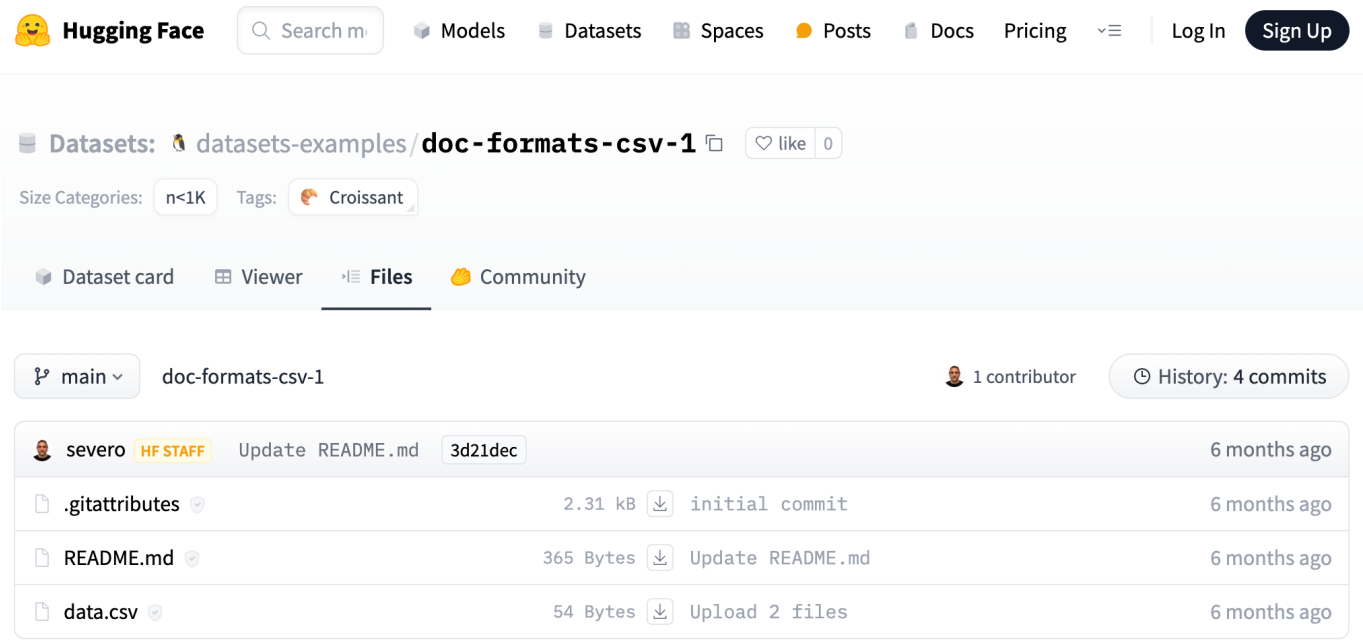
Dataset Repositories

Hugging Face is a popular central platform where users can store, share, and collaborate on machine learning models, datasets, and other resources.

A dataset typically includes the following content:

- A README file: This plain text file provides an overview of the repository and its contents. It often describes the purpose, usage, and specific requirements or dependencies.
- Data files: Depending on the type of repository, it can include data files like CSV, Parquet, JSONL, etc. These are the core components of the repository.

A typical repository looks like this:



Read Using `hf://` Paths

You often need to read files in various formats (such as CSV, JSONL, and Parquet) when working with data. As of version v0.10.3, DuckDB has native support for `hf://` paths as part of the `httpfs` extension, allowing easy access to all these formats.

Now, it is possible to query them using the URL pattern below:

```
hf://datasets/<my_username>/<my_dataset>/<path_to_file>
```

For example, to read a CSV file, you can use the following query:

```
SELECT *
FROM 'hf://datasets/datasets-examples/doc-formats-csv-1/data.csv';
```

Where:

- `datasets-examples` is the name of the user/organization
- `doc-formats-csv-1` is the name of the dataset repository
- `data.csv` is the file path in the repository

The result of the query is:

kind	sound
dog	woof
cat	meow
pokemon	pika
human	hello

To read a JSONL file, you can run:

```
SELECT *
FROM 'hf://datasets/datasets-examples/doc-formats-jsonl-1/data.jsonl';
```

Finally, for reading a Parquet file, use the following query:

```
SELECT *
FROM 'hf://datasets/datasets-examples/doc-formats-parquet-1/data/train-00000-of-00001.parquet';
```

Each of these commands reads the data from the specified file format and displays it in a structured tabular format. Choose the appropriate command based on the file format you are working with.

Creating a Local Table

To avoid accessing the remote endpoint for every query, you can save the data in a DuckDB table by running a `CREATE TABLE ... AS` command. For example:

```
CREATE TABLE data AS
    SELECT *
    FROM 'hf://datasets/datasets-examples/doc-formats-csv-1/data.csv';
```

Then, simply query the `data` table as follows:

```
SELECT *
FROM data;
```

Multiple Files

You might need to query multiple files simultaneously when working with large datasets. Let's see a quick sample using the cais/mmlu (Measuring Massive Multitask Language Understanding) dataset. This dataset captures a test consisting of multiple-choice questions from various branches of knowledge. It covers 57 tasks, including elementary mathematics, US history, computer science, law, and more. To attain high accuracy on this test, AI models must possess extensive world knowledge and problem-solving ability.

First, let's count the number of rows in individual files. To get the row count from a single file in the cais/mmlu dataset, use the following query:

```
SELECT count(*) AS count
FROM 'hf://datasets/cais/mmlu/astronomy/dev-00000-of-00001.parquet';
```

count
5

Similarly, for another file (`test-00000-of-00001.parquet`) in the same dataset, we can run:

```
SELECT count(*) AS count
FROM 'hf://datasets/cais/mmlu/astronomy/test-00000-of-00001.parquet';
```

count
152

To query all files under a specific format, you can use a glob pattern. Here's how you can count the rows in all files that match the pattern `*.parquet`:

```
SELECT count(*) AS count
FROM 'hf://datasets/cais/mmlu/astronomy/*.parquet';
```

count
173

By using glob patterns, you can efficiently handle large datasets and perform comprehensive queries across multiple files, simplifying your data inspections and processing tasks. Here, you can see how you can look for questions that contain the word "planet" in astronomy:

```
SELECT count(*) AS count
FROM 'hf://datasets/cais/mmlu/astronomy/*.parquet'
WHERE question LIKE '%planet%';
```

count
21

And see some examples:

```
SELECT question
FROM 'hf://datasets/cais/mmlu/astronomy/*.parquet'
WHERE question LIKE '%planet%'
LIMIT 3;
```

question
Why isn't there a planet where the asteroid belt is located?
On which planet in our solar system can you find the Great Red Spot?
The lithosphere of a planet is the layer that consists of

Versioning and Revisions

In Hugging Face repositories, dataset versions or revisions are different dataset updates. Each version is a snapshot at a specific time, allowing you to track changes and improvements. In git terms, it can be understood as a branch or specific commit.

You can query different dataset versions/revisions by using the following URL:

```
hf://datasets/<my_username>/<my_dataset>@<my_branch>/<path_to_file>
```

For example:

```
SELECT *
FROM 'hf://datasets/datasets-examples/doc-formats-csv-1@~parquet/**/*.parquet';
```

kind	sound
dog	woof
cat	meow
pokemon	pika
human	hello

The previous query will read all parquet files under the `~parquet` revision. This is a special branch where Hugging Face automatically generates the Parquet files of every dataset to enable efficient scanning.

Authentication

Configure your Hugging Face Token in the DuckDB Secrets Manager to access private or gated datasets. First, visit Hugging Face Settings – Tokens to obtain your access token. Second, set it in your DuckDB session using DuckDB's Secrets Manager. DuckDB supports two providers for managing secrets:

- `CONFIG`: The user must pass all configuration information into the `CREATE SECRET` statement. To create a secret using the `CONFIG` provider, use the following command:

```
CREATE SECRET hf_token (
   TYPE huggingface,
   TOKEN 'your_hf_token'
);
```

- `credential_chain`: Automatically tries to fetch credentials. For the Hugging Face token, it will try to get it from `~/.cache/huggingface/token`. To create a secret using the `credential_chain` provider, use the following command:

```
CREATE SECRET hf_token (
   TYPE huggingface,
   PROVIDER credential_chain
);
```

Conclusion

The integration of `hf://` paths in DuckDB significantly streamlines accessing and querying over 150,000 datasets available on Hugging Face. This feature democratizes data manipulation and exploration, making it easier for users to interact with various file formats such as CSV, JSON, JSONL, and Parquet. By utilizing `hf://` paths, users can execute complex queries, efficiently handle large datasets, and harness the extensive resources of Hugging Face repositories.

The integration supports seamless access to individual files, multiple files using glob patterns, and different dataset versions. DuckDB's robust capabilities ensure a flexible and streamlined data processing experience. This integration is a significant leap forward in making AI dataset access more accessible and efficient for researchers and developers, fostering innovation and accelerating progress in machine learning.

Want to learn more about leveraging DuckDB with Hugging Face datasets? Explore the detailed guide.

Analyzing Railway Traffic in the Netherlands

Publication date: 2024-05-31

Author: Gabor Szarnyas

TL;DR: We use a real-world railway dataset to demonstrate some of DuckDB's key features, including querying different file formats, connecting to remote endpoints, and using advanced SQL features.

Introduction

The Netherlands, the birthplace of DuckDB, has an area of about 42,000 km^2 with a population of about 18 million people. The high density of the country is a key factor in its extensive railway network, which consists of 3,223 km of tracks and 397 stations.

Information about this network's stations and services is available in the form of open datasets. These high-quality datasets are maintained by the team behind the Rijden de Treinen *(Are the trains running?)* application.

In this post, we'll demonstrate some of DuckDB's analytical capabilities on the Dutch railway network dataset. Unlike most of our other blog posts, this one doesn't introduce a new feature or release: instead, it demonstrates several existing features using a single domain. Some of the queries explained in this blog post are shown in simplified form on DuckDB's landing page.

Loading the Data

For our initial queries, we'll use the 2023 railway services dataset. To get this dataset, download the `services-2023.csv.gz` file (330 MB) and load it into DuckDB.

First, start the DuckDB command line client on a persistent database:

```
duckdb railway.db
```

Then, load the `services-2023.csv.gz` file into the `services` table.

```
CREATE TABLE services AS
    FROM 'services-2023.csv.gz';
```

Despite the seemingly simple query, there is quite a lot going on here. Let's deconstruct the query:

- First, there is no need to explicitly define a schema for our `services` table, nor is it necessary to use a `COPY ... FROM` statement. DuckDB automatically detects that the `'services-2023.csv.gz'` refers to a gzip-compressed CSV file, so it calls the `read_csv` function, which decompresses the file and infers its schema from its content using the CSV sniffer.
- Second, the query makes use of DuckDB's FROM-first syntax, which allows users to omit the `SELECT *` clause. Hence, the SQL statement `FROM 'services-2023.csv.gz';` is a shorthand for `SELECT * FROM 'services-2023.csv.gz';`.
- Third, the query creates a table called `services` and populates it with the result from the CSV reader. This is achieved using a `CREATE TABLE ... AS` statement.

Using DuckDB v0.10.3, loading the dataset takes approximately 5 seconds on an M2 MacBook Pro. To check the amount of data loaded, we can run the following query which pretty-prints the number of rows in the `services` table:

```
SELECT format('{:,}', count(*)) AS num_services
FROM services;
```

num_services
21,239,393

We can see that more than 21 million train services ran in the Netherlands in 2023.

Finding the Busiest Station per Month

Let's ask a simple query first: *What were the busiest railway stations in the Netherlands in the first 6 months of 2023?*

First, for every month, let's compute the number of services passing through each station. To do so, we extract the month from the service's date using the `month` function, then perform a group-by aggregation with a `count(*)`:

```
SELECT
    month("Service:Date") AS month,
    "Stop:Station name" AS station,
    count(*) AS num_services
FROM services
GROUP BY month, station
LIMIT 5;
```

Note that this query showcases a common redundancy in SQL: we list the names of non-aggregated columns in both the `SELECT` and the `GROUP BY` clauses. Using DuckDB's `GROUP BY ALL` feature, we can eliminate this. At the same time, let's also turn this result into an intermediate table called `services_per_month` using a `CREATE TABLE ... AS` statement:

```
CREATE TABLE services_per_month AS
    SELECT
        month("Service:Date") AS month,
        "Stop:Station name" AS station,
        count(*) AS num_services
    FROM services
    GROUP BY ALL;
```

To answer the question, we can use the `arg_max(arg, val)` aggregation function, which returns the column `arg` in the row with the maximum value `val`. We filter on the month and return the results:

```
SELECT
    month,
    arg_max(station, num_services) AS station,
    max(num_services) AS num_services
FROM services_per_month
WHERE month <= 6
GROUP BY ALL;
```

month	station	num_services
1	Utrecht Centraal	34760
2	Utrecht Centraal	32300
3	Utrecht Centraal	37386
4	Amsterdam Centraal	33426
5	Utrecht Centraal	35383
6	Utrecht Centraal	35632

Maybe surprisingly, in most months, the busiest railway station is not in Amsterdam but in the country's 4th largest city, Utrecht, thanks to its central geographic location.

Finding the Top-3 Busiest Stations for Each Summer Month

Let's change the question to: *Which are the top-3 busiest stations for each summer month?* The `arg_max()` function only helps us find the top-1 value but it is not sufficient for finding top-k results.

Using a Window Function (OVER)

DuckDB has extensive support for SQL features, including window functions and we can use the `rank()` function to find top-k values. Addtionally, we use `make_date` to reconstruct the date, `strftime` to turn it into the month's name and `array_agg`:

```
SELECT month, month_name, array_agg(station) AS top3_stations
FROM (
    SELECT
        month,
        strftime(make_date(2023, month, 1), '%B') AS month_name,
        rank() OVER
            (PARTITION BY month ORDER BY num_services DESC) AS rank,
        station,
        num_services
    FROM services_per_month
    WHERE month BETWEEN 6 AND 8
)
WHERE rank <= 3
GROUP BY ALL
ORDER BY month;
```

This gives the following result:

month	month_name	top3_stations
6	June	[Utrecht Centraal, Amsterdam Centraal, Schiphol Airport]
7	July	[Utrecht Centraal, Amsterdam Centraal, Schiphol Airport]
8	August	[Utrecht Centraal, Amsterdam Centraal, Amsterdam Sloterdijk]

We can see that the top 3 spots are shared between four stations: Utrecht Centraal, Amsterdam Centraal, Schiphol Airport, and Amsterdam Sloterdijk.

Using the `max_by(arg, val, n)` Function

Starting with DuckDB version 1.1.0, you can use a variant of the `max_by` function that accepts a third parameter, n, for the number of rows. The resulting code is more concise and faster than the one using a window function.

```
SELECT
    month,
    strftime(make_date(2023, month, 1), '%B') AS month_name,
    max_by(station, num_services, 3) AS stations,
FROM services_per_month
WHERE month BETWEEN 6 AND 8
GROUP BY ALL
ORDER BY month;
```

Directly Querying Parquet Files through HTTPS or S3

DuckDB supports querying remote files, including CSV and Parquet, via the HTTP(S) protocol and the S3 API. For example, we can run the following query:

```
SELECT "Service:Date", "Stop:Station name"
FROM 'https://blobs.duckdb.org/nl-railway/services-2023.parquet'
LIMIT 3;
```

It returns the following result:

Service:Date	Stop:Station name
2023-01-01	Rotterdam Centraal
2023-01-01	Delft
2023-01-01	Den Haag HS

Using the remote Parquet file, the query for answering *Which are the top-3 busiest stations for each summer month?* can be run directly on a remote Parquet file without creating any local tables. To do this, we can define the `services_per_month` table as a common table expression in the `WITH` clause. The rest of the query remains the same:

```
WITH services_per_month AS (
    SELECT
        month("Service:Date") AS month,
        "Stop:Station name" AS station,
        count(*) AS num_services
    FROM 'https://blobs.duckdb.org/nl-railway/services-2023.parquet'
    GROUP BY ALL
)
SELECT month, month_name, array_agg(station) AS top3_stations
FROM (
    SELECT
        month,
        strftime(make_date(2023, month, 1), '%B') AS month_name,
        rank() OVER
            (PARTITION BY month ORDER BY num_services DESC) AS rank,
        station,
        num_services
    FROM services_per_month
    WHERE month BETWEEN 6 AND 8
)
WHERE rank <= 3
GROUP BY ALL
ORDER BY month;
```

This query yields the same result as the query above, and completes (depending on the network speed) in about 1–2 seconds. This speed is possible because DuckDB doesn't need to download the whole Parquet file to evaluate the query: while the file size is 309 MB, it only uses about 20 MB of network traffic, approximately 6% of the total file size.

The reduction in network traffic is possible because of partial reading along both the columns and the rows of the data. First, Parquet's columnar layout allows the reader to only access the required columns. Second, the zonemaps available in the Parquet file's metadata allow the filter pushdown optimization (e.g., the reader only fetches row groups with dates in the summer months). Both of these optimizations are implemented via HTTP range requests, saving considerable traffic and time when running queries on remote Parquet files.

Largest Distance between Train Stations in the Netherlands

Let's answer the following question: *Which two train stations in the Netherlands have the largest distance between them when traveling via rail?* For this, we'll use two datasets. The first, `stations-2022-01.csv`, contains information on the railway stations (station name, country, etc.). We can simply load and query this dataset as follows:

```
CREATE TABLE stations AS
    FROM 'https://blobs.duckdb.org/data/stations-2022-01.csv';

SELECT
    id,
    name_short,
    name_long,
    country,
    printf('%.2f', geo_lat) AS latitude,
    printf('%.2f', geo_lng) AS longitude
FROM stations
LIMIT 5;
```

id	name_short	name_long	country	latitude	longitude
266	Den Bosch	's-Hertogenbosch	NL	51.69	5.29
269	Dn Bosch O	's-Hertogenbosch Oost	NL	51.70	5.32
227	't Harde	't Harde	NL	52.41	5.89
8	Aachen	Aachen Hbf	D	50.77	6.09
818	Aachen W	Aachen West	D	50.78	6.07

The second dataset, `tariff-distances-2022-01.csv`, contains the station distances. The distances are defined as the shortest route on the railway network and they are used to calculate the tariffs for ticket. Let's peek into this file:

```
head -n 9 tariff-distances-2022-01.csv | cut -d, -f1-9
```

```
Station,AC,AH,AHP,AHPR,AHZ,AKL,AKM,ALM
AC,XXX,82,83,85,90,71,188,32
AH,82,XXX,1,3,8,77,153,98
AHP,83,1,XXX,2,9,78,152,99
AHPR,85,3,2,XXX,11,80,150,101
AHZ,90,8,9,11,XXX,69,161,106
AKL,71,77,78,80,69,XXX,211,96
AKM,188,153,152,150,161,211,XXX,158
ALM,32,98,99,101,106,96,158,XXX
```

We can see that the distances are encoded as a matrix with the diagonal entries set to `XXX`. As explained in the dataset's description, this string implies that the two stations are the same station. If we just load the values as `XXX`, the CSV reader will assume that all columns have the type `VARCHAR` instead of numeric values. While this can be cleaned up later, it's a lot easier to avoid this problem altogether. To do so, we use the `read_csv` function and set the `nullstr` parameter to `XXX`:

```
CREATE TABLE distances AS
    FROM read_csv(
        'https://blobs.duckdb.org/data/tariff-distances-2022-01.csv',
        nullstr = 'XXX'
    );
```

To make the `NULL` values visible in the command line output, we set the `.nullvalue` dot command to `NULL`:

```
.nullvalue NULL
```

Then, using the DESCRIBE statement, we can confirm that DuckDB has inferred the column correctly as BIGINT:

```
FROM (DESCRIBE distances)
LIMIT 5;
```

column_name	column_type	null	key	default	extra
Station	VARCHAR	YES	NULL	NULL	NULL
AC	BIGINT	YES	NULL	NULL	NULL
AH	BIGINT	YES	NULL	NULL	NULL
AHP	BIGINT	YES	NULL	NULL	NULL
AHPR	BIGINT	YES	NULL	NULL	NULL

To show the first 9 columns, we can run the following query with the #1, #2, etc. column indexes in the SELECT statement:

```
SELECT #1, #2, #3, #4, #5, #6, #7, #8, #9
FROM distances
LIMIT 8;
```

Station	AC	AH	AHP	AHPR	AHZ	AKL	AKM	ALM
AC	NULL	82	83	85	90	71	188	32
AH	82	NULL	1	3	8	77	153	98
AHP	83	1	NULL	2	9	78	152	99
AHPR	85	3	2	NULL	11	80	150	101
AHZ	90	8	9	11	NULL	69	161	106
AKL	71	77	78	80	69	NULL	211	96
AKM	188	153	152	150	161	211	NULL	158
ALM	32	98	99	101	106	96	158	NULL

We can see that the data was loaded correctly but the wide table format is a bit unwieldy for further processing: to query for pairs of stations, we need to first turn it into a long table using the UNPIVOT statement. Naïvely, we would write something like the following:

```
CREATE TABLE distances_long AS
    UNPIVOT distances
    ON AC, AH, AHP, ...
```

However, we have almost 400 stations, so spelling out their names would be quite tedious. Fortunately, DuckDB has a trick to help with this: the COLUMNS(*) expression lists all columns and its optional EXCLUDE clause can remove given column names from the list. Therefore, the expression COLUMNS(* EXCLUDE station) lists all column names except station, precisely what we need for the UNPIVOT command:

```
CREATE TABLE distances_long AS
    UNPIVOT distances
    ON COLUMNS (* EXCLUDE station)
    INTO NAME other_station VALUE distance;
```

This results in the following table:

```
SELECT station, other_station, distance
FROM distances_long
LIMIT 3;
```

Station	other_station	distance
AC	AH	82
AC	AHP	83
AC	AHPR	85

Now we can join the `distances_long` table on the `stations` table along both the start and end stations, then filter for stations which are located in the Netherlands. We introduce symmetry breaking (`station < other_station`) to ensure that the same pair of stations only occurs once in the output. Finally, we select the top-3 results:

```
SELECT
    s1.name_long AS station1,
    s2.name_long AS station2,
    distances_long.distance
FROM distances_long
JOIN stations s1 ON distances_long.station = s1.code
JOIN stations s2 ON distances_long.other_station = s2.code
WHERE s1.country = 'NL'
  AND s2.country = 'NL'
  AND station < other_station
ORDER BY distance DESC
LIMIT 3;
```

The results show that there are pairs of train stations, which are at least 425 km away – quite the distance for such a small country!

station1	station2	distance
Eemshaven	Vlissingen	426
Eemshaven	Vlissingen Souburg	425
Bad Nieuweschans	Vlissingen	425

Conclusion

In this post, we demonstrated some of DuckDB's key features, including automatic detection of formats based on filenames, auto-inferencing the schema of CSV files, direct Parquet querying, remote querying, window functions, unpivot, several friendly SQL features (such as FROM-first, `GROUP BY ALL`, and `COLUMNS(*)`), and so on. The combination of these allows for formulating queries using different file formats (CSV, Parquet), data sources (local, HTTPS, S3), and SQL features. This helps users answer queries quickly and efficiently.

In the next installment, we'll take a look at temporal data using AsOf joins and geospatial data using the DuckDB `spatial` extension.

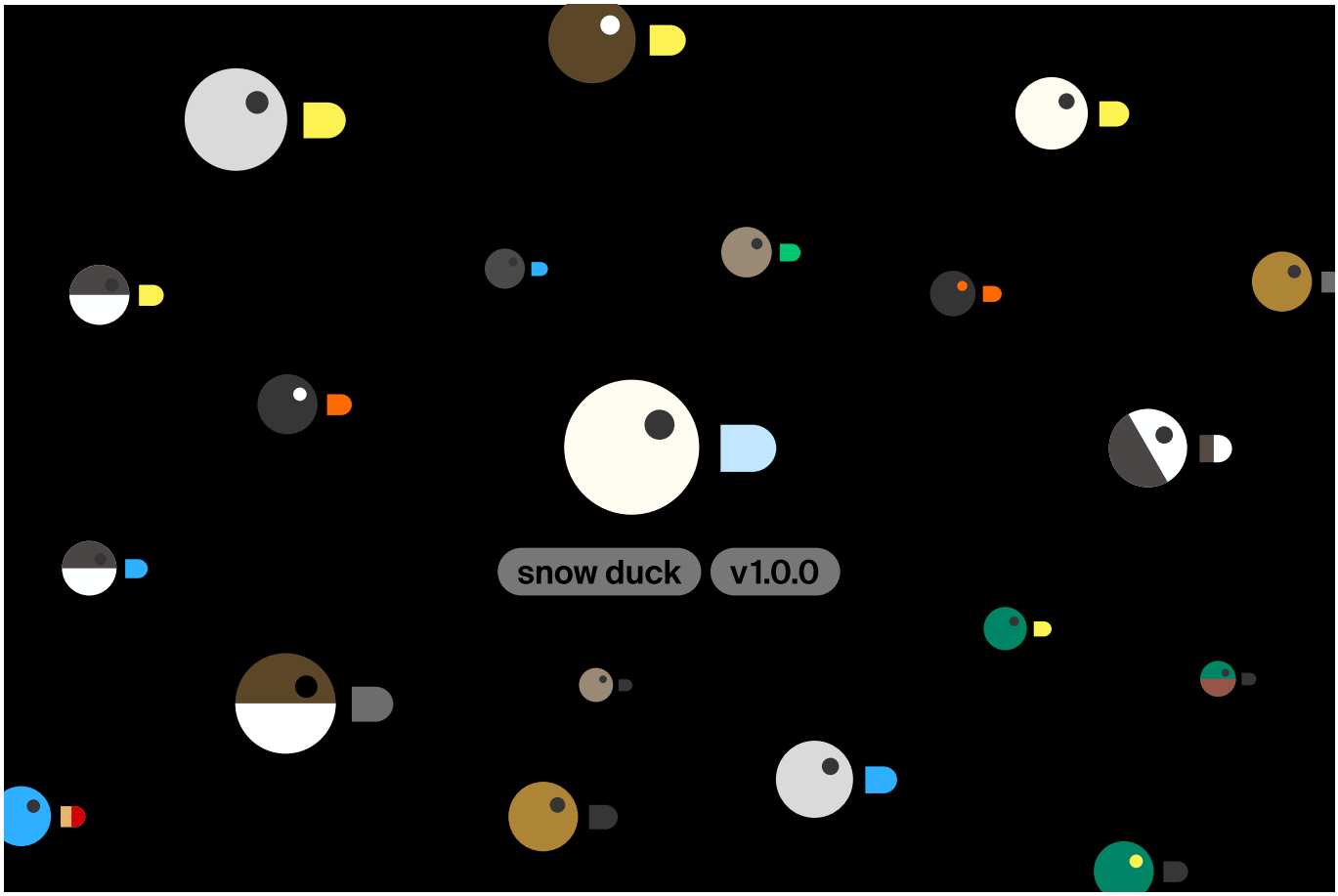
Announcing DuckDB 1.0.0

Publication date: 2024-06-03

Authors: Mark Raasveldt and Hannes Mühleisen

TL;DR: The DuckDB team is very happy to announce that today we're releasing DuckDB version 1.0.0, codename "Snow Duck" (anas nivis).

To install the new version, please visit the installation guide. For the release notes, see the release page.



It has been almost six years since the first source code was written for the project back in 2018, and a *lot* has happened since: There are now over 300 000 lines of C++ engine code, over 42 000 commits and almost 4 000 issues were opened and closed again. DuckDB has also gained significant popularity: the project has attracted tens of thousands of stars and followers on GitHub and social media platforms. Download counts are in the millions each month, and download traffic just for extensions is upwards of four terabytes *each day*. There are even books being written about DuckDB, and – most importantly – now even Wikipedia considers DuckDB notable, albeit barely.

Why Now?

Of course, version numbers are somewhat arbitrary and "feely", despite attempts at making them more mechanical. We could have released DuckDB 1.0.0 back in 2018, or we could have waited ten more years. There is never a great moment, because software (with the exception of TeX) is never "done". Why choose today?

Data management systems – even purely analytical ones – are such core components of any application that there is always an implicit contract of trust between their developers and users. Users rely on databases to provide correct query results and to not lose their data. At the same time, system developers need to be aware of their responsibility of not breaking people's applications willy-nilly. Intuitively, version 1.0.0 means something else for a data management system than it means for an egg timer app (no offense). From the very beginning, we were committed to making DuckDB a reliable base for people to build their applications on. This is also why the 1.0.0 release is named after the non-existent *snow duck (anas nivis),* harking back to Apple's Snow Leopard release some years ago.

For us, one of the major blockers to releasing 1.0.0 was the storage format. DuckDB has its own custom-built data storage format. This format allows users to manage many (possibly very large) tables in a single file with full transactional semantics and state-of-the-art compression. Of course, designing a new file format is not without its challenges, and we had to make significant changes to the format over time. This led to the suboptimal situation that whenever a new DuckDB version was released, the files created with the old version did not work with the new DuckDB version and had to be manually upgraded. This problem was addressed in v0.10.0 back in February – where we introduced backward compatibility and limited forward compatibility for DuckDB's storage format. This feature has now been used in the wild for a while without serious issues – providing us with the confidence to offer a guarantee that DuckDB files created with DuckDB 1.0.0 will be compatible with future DuckDB versions.

Stability

The core theme of the 1.0.0 release is stability. This contrasts it with previous releases where we have had blog posts talk about long lists of new features. Instead, the 1.0.0 release has very limited new features (a few might have snuck in). Instead, our focus has been on stability.

We've observed the frankly staggering growth in the amount and breadth of use of DuckDB in the wild, and have not seen an increase in serious issues being reported. Meanwhile, there are thousands of test cases with millions of test queries being run every night. We run loads of microbenchmarks and standardized benchmark suites to spot performance regressions. DuckDB is constantly being tortured by various fuzzers that construct all manners of wild SQL queries to make sure we don't miss weird corner cases. All told, this has built the necessary confidence in us to release a 1.0.0.

Another core aspect of stability with the 1.0.0 release is stability across versions. While never breaking anyone's workflow is likely impossible, we plan to be much more careful with user-facing changes going forward. In particular, we plan to focus on providing stability for the SQL dialect, as well as the C API. While we do not guarantee that we will never change semantics in these layers in the future – we will try to provide ample warning when doing so, as well as providing workarounds that allow previously working code to keep on working.

Looking ahead

Unlike many open-source projects, DuckDB also has a healthy long-term funding strategy. DuckDB Labs, the company that employs DuckDB's core contributors, has not had any outside investments, and as a result, the company is fully owned by the team. Labs' business model is to provide consulting and support services for DuckDB, and we're happy to report that this is going well. With the revenue from contracts, we fund long-term and strategic DuckDB development with a team of almost 20 people. At the same time, the intellectual property in the project is guarded by the independent DuckDB Foundation. This non-profit foundation ensures that DuckDB will be around long-term under the MIT license.

Regarding long-term plans, there are, of course, many things on the roadmap still. One thing we're very excited about is the ability to expand the extension environment around DuckDB. Extensions are plug-ins that can add new SQL-level functions, file formats, optimizers, etc. while keeping the DuckDB core mean and lean. There are already an impressive number of third-party extensions to DuckDB, and we're working hard to streamline the process of building and distributing community-contributed extensions. We think DuckDB can become the basis for the next revolution in data through community extensions connected by a high-performance data fabric accessible through a unified SQL interface.

Of course, there will be issues found in today's release. But rest assured, there will be a 1.0.1 release. There will be a 1.1.0. And there might also be a 2.0.0 at some point. We're in this for the long run, all of us, together. We have the team and the structures and resources to do so.

Acknowledgments

First of all, we are very, very grateful to you all. Our massive and heartfelt thanks go to everyone who has contributed code, filed issues or engaged in discussions, promoted DuckDB in their environment, and, of course, all DuckDB users. We could not have done it without you!

We would also like to thank the CWI Database Architectures group for providing us with the environment and expertise to build DuckDB, the organizations that provided us with research grants early on, the excellent customers of DuckDB Labs that make it all work (especially the early ones), and the generous donors to the DuckDB Foundation. We are particularly grateful to our long-standing Gold sponsors MotherDuck, Voltron Data and Posit.

Finally, we would like to thank the excellent and amazing team at DuckDB Labs.

So join us now in being nostalgic, teary-eyed and excited for what's to come for DuckDB and celebrate the release of DuckDB 1.0.0 with us. We certainly will.

Mark and Hannes

PS: We are holding our next community event, DuckCon #5, in Seattle on August 15, only a few short weeks from today. Attendance is free. Hope to see you there!

For press inquiries, please reach out to Gabor Szarnyas.

Native Delta Lake Support in DuckDB

Publication date: 2024-06-10

Author: Sam Ansmink

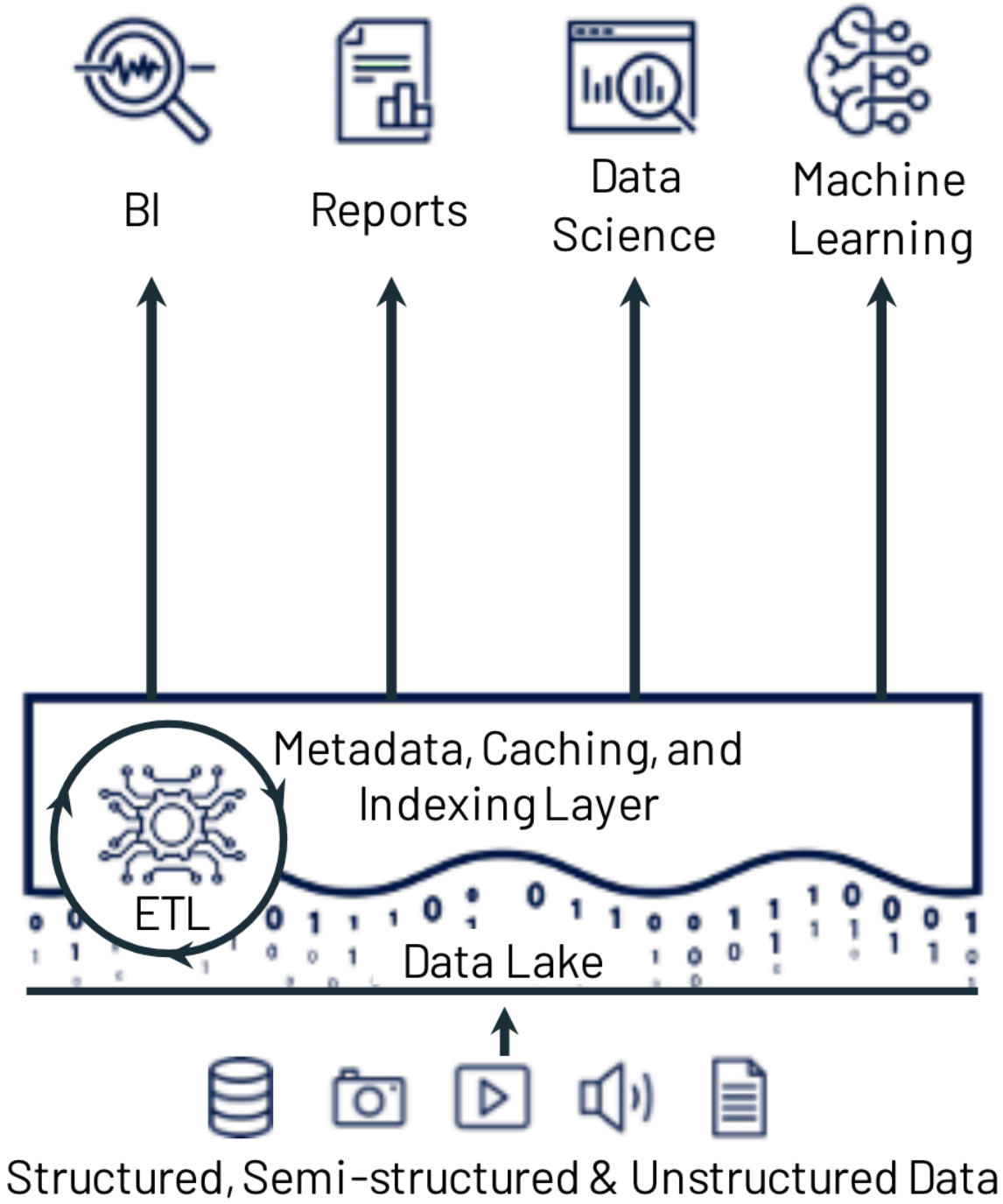
TL;DR: DuckDB now has native support for Delta Lake, an open-source lakehouse framework, with the `delta` extension.

Over the past few months, DuckDB Labs has teamed up with Databricks to add first-party support for Delta Lake in DuckDB using the new `delta-kernel-rs` project. In this blog post we'll give you a short overview of Delta Lake, Delta Kernel and, of course, present the new DuckDB Delta extension.

If you're already dearly familiar with Delta Lake and Delta Kernel, or you are just here to know how to boogie, feel free to skip to the juicy bits on how to use the DuckDB with Delta.

Intro

Delta Lake is an open-source storage framework that enables building a lakehouse architecture. So to understand Delta Lake, we need to understand what the lakehouse architecture is. Lakehouse is a data management architecture that strives to combine the cost-effectiveness of cheap object storage with a smart management layer. In simple terms, lakehouse architectures are a collection of files in various formats, with some additional metadata layers on top. These metadata layers aim to provide extra functionality on top of the raw collection of files such as ACID transactions, time travel, partition- and schema evolution, statistics, and much more. What a lakehouse architecture enables, is to run various types of data-intensive applications such as data analytics and machine learning applications, directly on a vast collection of structured, semi-structured and unstructured data, without the need for an intermediate data warehousing step. If you're ready for the deep dive, we recommend reading the CIDR 2021 paper "Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics" by Michael Armbrust et al. However, if you're (understandably) hesitant to dive into dense scientific literature, this image sums it up pretty well:



Lakehouse architecture (image source: Armburst et al., CIDR 2021)

Delta Lake

Now let's zoom in a little on our star of the show for tonight, Delta Lake. Delta Lake (or simply "Delta") is currently one of the leading open-source lakehouse formats, along with Apache Iceberg™ and Apache HUDI™. The easiest way to get a feeling for what a Delta table is, is to think of a Delta table as a "collection of Parquet files with some metadata". With this slight oversimplification in mind, we will now create a Delta table and examine the files that are created, to improve our understanding. To do this, we'll set up Python with the packages:

duckdb, pandas and deltalake:

```
pip install duckdb pandas deltalake
```

Then, we use DuckDB to create some dataframes with test data, and write that to a Delta table using the `deltalake` package:

```
import duckdb
from deltalake import DeltaTable, write_deltalake
con = duckdb.connect()
df1 = con.query("SELECT i AS id, i % 2 AS part, 'value-' || i AS value FROM range(0, 5) tbl(i)").df()
df2 = con.query("SELECT i AS id, i % 2 AS part, 'value-' || i AS value FROM range(5, 10) tbl(i)").df()
write_deltalake(f"./my_delta_table", df1,  partition_by=["part"])
write_deltalake(f"./my_delta_table", df2,  partition_by=["part"], mode='append')
```

With this script run, we have created a basic Delta table containing 10 rows, split across two partitions that we added in two separate steps. To double-check that everything is going to plan, let's use DuckDB to query the table:

```
SELECT *
FROM delta_scan('./my_delta_table')
ORDER BY id;
```

id	part	value
0	0	value-0
1	1	value-1
2	0	value-2
3	1	value-3
4	0	value-4
5	1	value-5
6	0	value-6
7	1	value-7
8	0	value-8
9	1	value-9

That looks great! All our expected data is there. Now let's take a look at what files have actually been created using `tree`:

```
tree ./my_delta_table`
```

```
my_delta_table
├── _delta_log
│   ├── 00000000000000000000.json
│   └── 00000000000000000001.json
├── part=0
│   ├── 0-f45132f6-2231-4dbd-aabb-1af29bf8724a-0.parquet
│   └── 1-76c82535-d1e7-4c2f-b700-669019d94a0a-0.parquet
└── part=1
    ├── 0-f45132f6-2231-4dbd-aabb-1af29bf8724a-0.parquet
    └── 1-76c82535-d1e7-4c2f-b700-669019d94a0a-0.parquet
```

The `tree` output shows 2 different types of files. While a Delta table can contain various other types of files, these form the basis of any Delta table.

Firstly, there are **data files** in Parquet format. The data files contain all the data that is stored in the table. This is very similar to how data is stored when DuckDB is used to write partitioned Parquet files.

Secondly, there are **delta files** in JSON format. The Delta files contain a log of the changes that have been made to the table. By replaying this log, a reader can construct a valid view of the table. To illustrate this, let's take a small peek into one of the first Delta log files:

```
cat my_delta_table/_delta_log/00000000000000000000.json
```

```
...
{ "add": {
    "path": "part=1/0-f45132f6-2231-4dbd-aabb-1af29bf8724a-0.parquet",
    "partitionValues": { "part": "1" }
  },
  ...
}
{ "add": {
    "path": "part=0/0-f45132f6-2231-4dbd-aabb-1af29bf8724a-0.parquet",
    "partitionValues": { "part": "0" },
  },
  ...
}
...
```

As we can see, this log file contains two `add` objects that describe some data being added to respectively the `1` and `0` partitions. Note also that the partition values themselves are stored in these Delta files explicitly, so even though the file structure looks very similar to a Hive-style partitioning scheme, the folder names are not actually used by Delta internally. Instead, the partition values are read from the metadata.

Now with this simple example, we've shown the basics of how Delta works. For a more thorough understanding of the internals, we refer to the official Delta specification, which is, by protocol specification standards, quite easy to read. The official specification describes in detail how Delta handles every detail, from the basics described here to more complex things like checkpointing, deletes, schema evolution, and much more.

Implementation

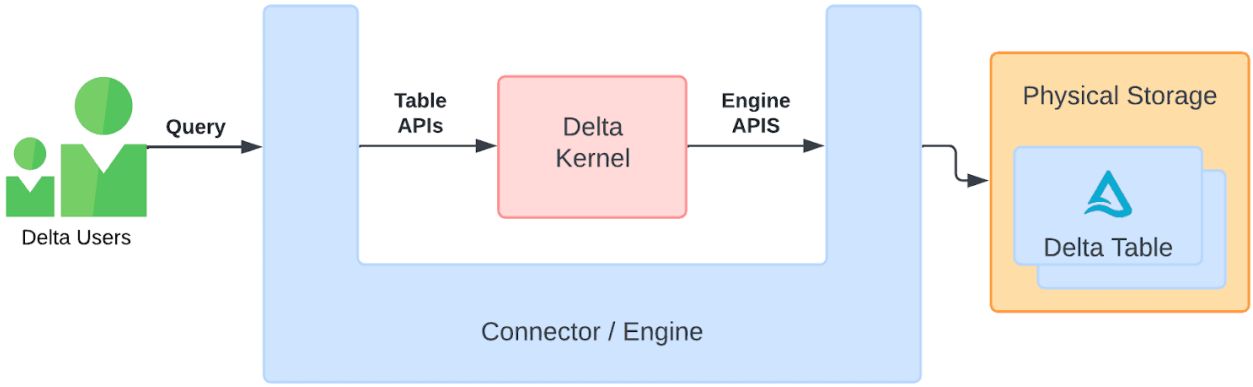
The Delta Kernel

Supporting a relatively complex protocol such as Delta, requires significant development and maintenance effort. For this reason, when looking to add support for such a protocol to an engine, the logical choice would be to look for a ready-to-use library to take care of this. In the case of Delta Lake, we could, for example, opt for the `delta-rs` library. However, when it comes to implementing a native DuckDB Delta extension, this is problematic: if we were to use the `delta-rs` library for implementing the DuckDB extension, all interaction with the Delta tables would go through the `delta-rs` library. But remember, a Delta table is effectively *"just a bunch of Parquet files with some metadata"*. Therefore, this would mean that when DuckDB wants to read a Delta table, the data files will be read by the `delta-rs` Parquet reader, using the`delta-rs` filesystem. But that's annoying: DuckDB already comes shipped with an excellent Parquet reader. Also, DuckDB already has support for a variety of filesystems with its own credential management system. By using a library like `delta-rs` for DuckDB's Delta extension this would actually run into a variety of problems:

- increased extension binary size
- inconsistent user experience between `delta_scan` and `read_parquet`
- increased maintenance load

Now to solve these problems, we would prefer to have some library that implements **only the Delta protocol** while letting DuckDB handle all the things it already knows how to handle.

Fortunately for us, this library exists and it's called the Delta Kernel Project. The Delta Kernel is a "set of libraries for building Delta connectors that can read from and write into Delta tables without the need to understand the Delta protocol details". This is done by exposing two relatively simple sets of APIs that an engine would implement, as shown in the image below:



For more details on the `delta-kernel-rs` project, we refer to this excellent blog post, which goes in-depth into the internals and design rationale.

Now while the `delta-kernel-rs` library is still experimental, it has recently launched its v0.1.0 version, and already offers a lot of functionality. Furthermore, because `delta-kernel-rs` exposes a C/C++ foreign function interface, integrating it into a DuckDB extension has been very straightforward.

DuckDB Delta Extension `delta_scan`

Now we're ready to dive into the nitty-gritties of the DuckDB Delta extension internals. To start, the Delta extension currently implements a single table function: `delta_scan`. It's a simple but powerful function that scans a Delta Table.

To understand how this function is implemented, we first need to establish the four main components involved:

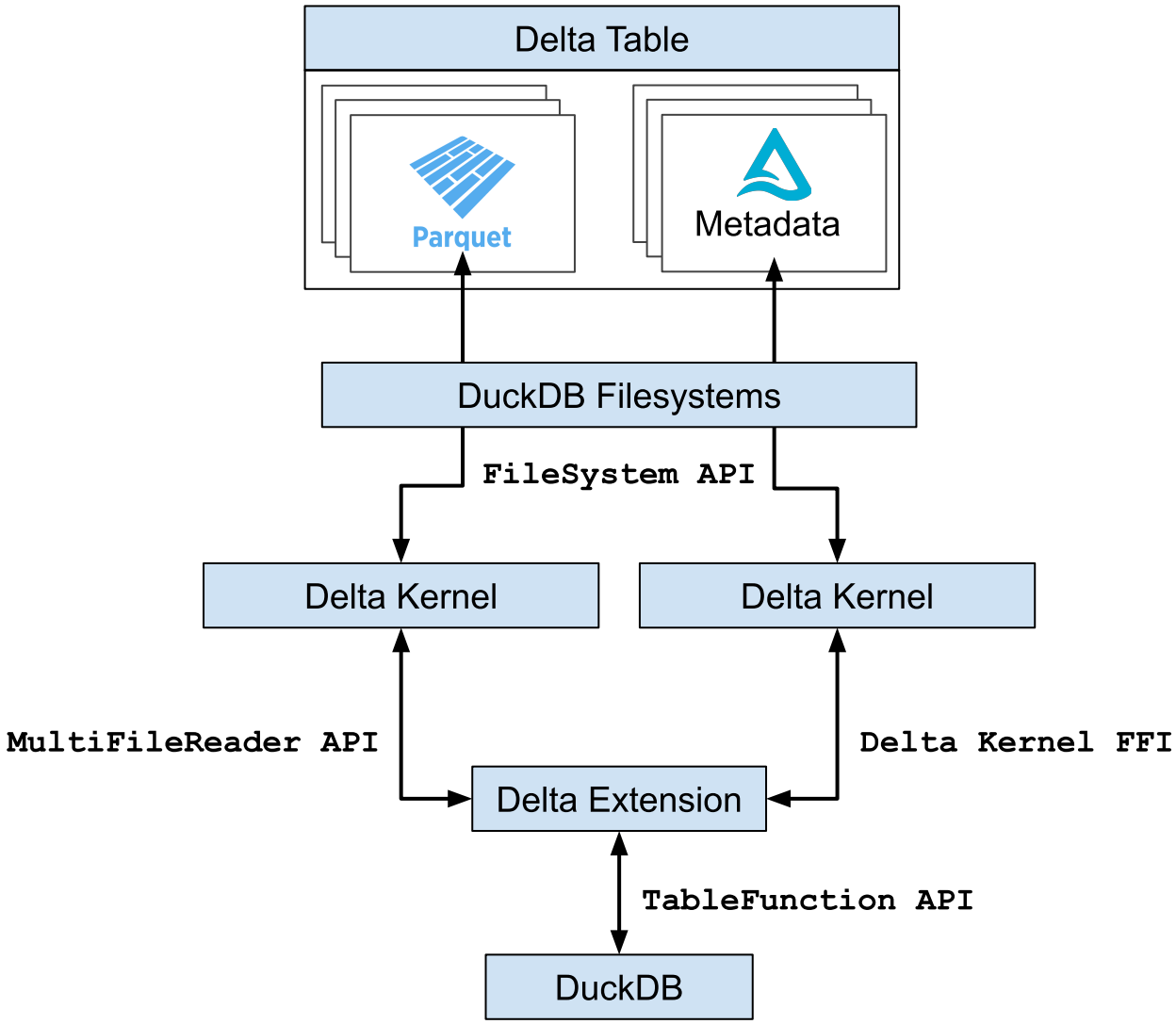
Component	Description
Delta kernel	The delta-kernel-rs library
Delta extension	DuckDB's loadable Delta extension
Parquet extension	DuckDB's loadable Parquet extension
DuckDB	Super cool duck-themed analytical database

Additionally, we need to understand that there are four main APIs involved:

API	Description
`FileSystem`	DuckDB's API for I/O (for local files, Azure, S3, etc.)
`TableFunction`	DuckDB's API for table functions (e.g., `read_parquet`, `read_csv`)
`MultiFileReader`	DuckDB's API for handling multi-file scans
Delta Kernel C/C++ FFI	Delta Kernel FFI for Delta Lake

Now we have all the links, let's tie them all together. When a user runs a query with a `delta_scan` table function, DuckDB will call into the `delta_scan` function from the Delta extension using the `TableFunction` API. The `delta_scan` table function, however, is actually just an exact copy of the regular `read_parquet` function. To change the `read_parquet` into a `delta_scan`, it will replace the regular `MultiFileReader` of the `parquet_scan` (which simply scans a list or glob of files), with a custom `DeltaMultiFileReader` that

will generate a list of files based on the Delta Table metadata. Finally, whenever the Parquet extension requires any IO, it will call into DuckDB using the `FileSystem` API to handle the I/O. This entire interaction is captured in the diagram below.



In this Diagram, we can see all four components involved in the processing of query containing a `delta_scan` table function. The arrows represent the communication that occurs between the components across the four APIs. Now when reading a Delta Table, we can see that the Metadata is handled on the right side going through the Delta Kernel. On the left side we can see how the Parquet data flows through the Parquet extension.

While there are obviously some important details missing here, such as the handling of deletion vectors and column mappings, we have now covered the basic concept of the DuckDB Delta extension. Also, we have demonstrated how the current implementation achieves a very natural logical separation, with component internals being abstracted away by connecting through clearly defined APIs. In doing so, the implementation achieves the following key properties:

1. **The details of the Delta protocol remain largely opaque to any DuckDB component.** The only point of contact with the internals of the Delta protocol is the narrow FFI exposed by the Delta kernel. This is fully handled by the Delta extension, whose only job is to translate this into native DuckDB APIs.
2. **Full reuse of existing Parquet scanning logic** of DuckDB, without any code reuse or compile time dependencies between extensions. Because all interaction between the Delta and Parquet extension is done over DuckDB APIs through the running DuckDB instance, the extensions only interface over the `TableFunction` and `MultiFileReader` APIs. This also means that any future optimizations that are made to the Parquet extension will automatically be available in the Delta extension.

3. **All I/O will go through DuckDB's `FileSystem` API.** This means that all file systems (Azure, S3, etc.) that are available to DuckDB, are available to scan with. This means that any DuckDB file system that can read and list files can be used for Delta. This is also useful in DuckDB-Wasm where custom filesystem implementations are used. *Warning*, two small notes need to be made here. Firstly, currently the DuckDB Delta extension still lets a small part of IO be handled by the Delta kernel through internal filesystem libraries, this is due to the FFI not yet exposing the `FileSystem` APIs, but this will change very soon. Secondly, while the architectural design of the Delta Extension is made with DuckDB-Wasm in mind, the Wasm version of the extension is not yet available.

How to Use Delta in DuckDB

Using the Delta extension in DuckDB is very simple, as it is distributed as one of the core DuckDB extensions, and available for autoloading. What this means is that you can simply start DuckDB (using v0.10.3 or higher) and run:

```
SELECT * FROM delta_scan('./my_delta_table');
```

DuckDB will automatically install and load the Delta Extension. Then it will query the local Delta table `./my_delta_table`.

In case your Delta table lives on S3, there are probably some S3 credentials that you want to set. If these credentials are already in one of the default places, such as an environment variable, or in the `~/.aws/credentials` file? Simply run:

```
CREATE SECRET delta_s1 (
    TYPE s3,
    PROVIDER credential_chain
)
SELECT * FROM delta_scan('s3://<some-bucket>/<path/to/a/delta/table>');
```

Do you prefer remembering your AWS tokens by heart, and would like to type them out? Go with:

```
CREATE SECRET delta_s2 (
    TYPE s3,
    KEY_ID '<AKIAIOSFODNN7EXAMPLE>',
    SECRET '<wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY>',
    REGION '<eu-west-1>'
)
SELECT * FROM delta_scan('s3://<some-bucket>/<path/to/a/delta/table>');
```

Do you have multiple Delta tables, with different credentials? No problem, you can use scoped secrets:

```
CREATE SECRET delta_s3 (
    TYPE s3,
    KEY_ID '<AKIAIOSFODNN7EXAMPLE1>',
    SECRET '<wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY1>',
    REGION '<eu-west-1>',
    SCOPE 's3://<some-bucket-1>'
)
CREATE SECRET delta_s4 (
    TYPE s3,
    KEY_ID '<AKIAIOSFODNN7EXAMPLE2>',
    SECRET '<wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY2>',
    REGION '<us-west-1>',
    SCOPE 's3://<some-bucket-2>'
)
SELECT * FROM delta_scan('s3://<some-bucket-1>/<table1>');
SELECT * FROM delta_scan('s3://<some-bucket-2>/<table2>');
```

Finally, is your table public, but outside the default AWS region? Make sure you set the region using an empty S3 Secret:

```
CREATE SECRET delta_s5 (
    TYPE s3,
    REGION '<eu-west-2>'
)
SELECT * FROM delta_scan('s3://<some-public-bucket>/<table1>');
```

Current State of the Delta Extension

Currently, the Delta Extension is still considered **experimental**. This is partly because the Delta extension itself is still very new, but also because the `delta-kernel-rs` project it relies on is still experimental. Nevertheless, core Delta scanning features are already supported by the current version of the Delta extension, such as:

- All data types
- Filter and projection pushdown
- File skipping based on filter pushdown
- Deletion vectors
- Partitioned tables
- Fully parallel scanning

The Delta extension is available on the platforms `linux_amd64`, `linux_arm64`, `osx_amd64` and `osx_arm64`. Support for the remaining platforms is coming soon. Additionally, we will continue to work together with Databricks on further improving the Delta Extension to add more features such as:

- Write support
- Column mapping
- Time travel
- Variant, RowIds
- Wasm support

For details and info on newly added features, keep an eye on the Delta extension docs and repository.

Conclusion

In this blog post, we presented DuckDB's new Delta extension, enabling easy interaction with Delta Lake directly from the comfort of your own DuckDB environment. To do so, we demonstrated what the Delta Lake format looks like by creating a Delta table and analyzing it using DuckDB.

We want to emphasize the fact that by implementing the Delta extension with the `delta-kernel-rs` library, both DuckDB and the Delta extension have been kept relatively simple and largely agnostic to the internals of the Delta protocol.

We hope you give the Delta extension a try and look forward to any feedback from the community! Also, if you're attending the 2024 Databricks Data + AI Summit be sure to check out DuckDB co-founder Hannes Mühleisen's talk on Thursday during the keynote and the in-depth breakout session, also on Thursday, for more details on the DuckDB–Delta integration.

Command Line Data Processing: Using DuckDB as a Unix Tool

Publication date: 2024-06-20

Author: Gabor Szarnyas

TL;DR: DuckDB's CLI client is portable to many platforms and architectures. It handles CSV files conveniently and offers users the same rich SQL syntax everywhere. These characteristics make DuckDB an ideal tool to complement traditional Unix tools for data processing in the command line.

In this blog post, we dive into the terminal to compare DuckDB with traditional tools used in Unix shells (Bash, Zsh, etc.). We solve several problems requiring operations such as projection and filtering to demonstrate the differences between using SQL queries in DuckDB versus specialized command line tools. In the process, we will show off some cool features such as DuckDB's powerful CSV reader and the positional join operator. Let's get started!

The Unix Philosophy

To set the stage, let's recall the Unix philosophy. This states that programs should:

- do one thing and do it well,
- work together, and
- handle text streams.

Unix-like systems such as macOS, Linux and WSL in Windows have embraced this philosophy. Tools such as `grep`, `sed`, and `sort` are ubiquitious and widely used in shell scripts.

As a purpose-built data processing tool, DuckDB fits the Unix philosophy quite well. First, it was designed to be a fast in-process analytical SQL database system *(do one thing and do it well).* Second, it has a standalone command line client, which can consume and produce CSV files *(work together),* and also supports reading and writing text streams *(handle text streams)*. Thanks to these, DuckDB works well in the ecosystem of Unix CLI tools, as shown in several posts.

Portability and Usability

While Unix CLI tools are fast, robust, and available on all major platforms, they often have cumbersome syntax that's difficult to remember. To make matters worse, these tools often come with slight differences between systems – think of the differences between GNU `sed` and macOS's `sed` or the differences between regex syntax among programs, which is aptly captured by Donald Knuth's quip *"I define Unix as 30 definitions of regular expressions living under one roof."*

While there are shells specialized specifically for dataframe processing, such as the Nushell project, older Unix shells (e.g., the Bourne shell `sh` and Bash) are still the most wide-spread, especially on servers.

At the same time, we have DuckDB, an extremely portable database system which uses the same SQL syntax on all platforms. With version 1.0.0 released recently, DuckDB's syntax – based on the proven and widely used PostgeSQL dialect – is now in a stable state. Another attractive feature of DuckDB is that it offers an interactive shell, which aids quick debugging. Moreover, DuckDB is available in several host languages as well as in the browser via WebAssembly, so if you ever decide to use your SQL scripts outside of the shell, DuckDB SQL scripts can be ported to a wide variety of environments without any changes.

Data Processing with Unix Tools and DuckDB

In the following, we give examples for implementing simple data processing tasks using the CLI tools provided in most Unix shells and using DuckDB SQL queries. We use DuckDB v1.0.0 and run it in in-memory mode. This mode makes sense for the problems we are tackling, as we do not create any tables and the operations are not memory-intensive, so there is no data to persist or to spill on disk.

Datasets

We use the four input files capturing information on cities and airports in the Netherlands.

You can download all input files as a single zip file.

Projecting Columns

Projecting columns is a very common data processing step. Let's take the `pop.csv` file and project the first and last columns, `city` and `population`.

Unix Shell: `cut`

In the Unix shell, we use the `cut` command and specify the file's delimiter (`-d`) and the columns to be projected (`-f`).

```
cut -d , -f 1,3 pop.csv
```

This produces the following output:

```
city,population
Amsterdam,905234
Rotterdam,656050
The Hague,552995
Utrecht,361924
Eindhoven,238478
Groningen,234649
Tilburg,224702
Almere,218096
Breda,184716
Nijmegen,179073
```

DuckDB: SELECT

In DuckDB, we can use the CSV reader to load the data, then use the `SELECT` clause with column indexes (`#i`) to designate the columns to be projected:

```
SELECT #1, #3 FROM 'pop.csv';
```

Note that we did not have to define any schema or load the data to a table. Instead, we simply used `'pop.csv'` in the `FROM` clause as we would do with a regular table. DuckDB detects that this is a CSV file and invokes the `read_csv` function, which automatically infers the CSV file's dialect (delimiter, presence of quotes, etc.) as well as the schema of the table. This allows us to simply project columns using `SELECT #1, #3`. We could also use the more readable syntax `SELECT city, population`.

To make the output of the solutions using Unix tools and DuckDB equivalent, we wrap the query into a `COPY ... TO` statement:

```
COPY (
    SELECT #1, #3 FROM 'pop.csv'
  ) TO '/dev/stdout/';
```

This query produces the same result as the Unix command's output shown above.

To turn this into a standalone CLI command, we can invoke the DuckDB command line client with the `-c <query>`{:.language-sql .highlight} argument, which runs the SQL query and exits once it's finished. Using this technique, the query above can be turned into the following one-liner:

```
duckdb -c "COPY (SELECT #1, #3 FROM 'pop.csv') TO '/dev/stdout/'"
```

In the following, we'll omit the code blocks using the standalone `duckdb`{:.language-sql .highlight} command: all solutions can be executed in the `duckdb -c <query>`{:.language-sql .highlight} template and yield the same result as the solutions using Unix tools.

Sorting Files

Another common task is to sort files based on given columns. Let's rank the cities within provinces based on their populations. To do so, we need to sort the `pop.csv` file first based on the name of the `province` using an ascending order, then on the `population` using a descending order. We then return the `province` column first, followed by the `city` and the `population` columns.

Unix Shell: `sort`

In the Unix shell, we rely on the `sort` tool. We specify the CSV file's separator with the `-t` argument and set the keys to sort on using `-k` arguments. We first sort on the second column (`province`) with `-k 2,2`. Then, we sort on the third column (`population`), setting the ordering to be reversed (`r`) and numeric (`n`) with `-k 3rn`. Note that we need to handle the header of the file separately: we take the first row with `head -n 1` and the rest of the rows with `tail -n +2`, sort the latter, and glue them back together with the header. Finally, we perform a projection to reorder the columns. Unfortunately, the `cut` command cannot reorder the columns, so we use `awk` instead:

```
(head -n 1 pop.csv; tail -n +2 pop.csv \
    | sort -t , -k 2,2 -k 3rn) \
    | awk -F , '{ print $2 "," $1 "," $3 }'
```

The result is the following:

```
province,city,population
Flevoland,Almere,218096
Gelderland,Nijmegen,179073
Groningen,Groningen,234649
North Brabant,Eindhoven,238478
North Brabant,Tilburg,224702
North Brabant,Breda,184716
North Holland,Amsterdam,905234
South Holland,Rotterdam,656050
South Holland,The Hague,552995
Utrecht,Utrecht,361924
```

DuckDB: `ORDER BY`

In DuckDB, we simply load the CSV and specify the column ordering via `SELECT province, city, population`, then set the sorting criteria on the selected columns (`province` ASC and `population` DESC). The CSV reader automatically detects types, so the sorting is numeric by default. Finally, we surround the query with a COPY statement to print the results to the standard output.

```
COPY (
    SELECT province, city, population
    FROM 'pop.csv'
    ORDER BY province ASC, population DESC
  ) TO '/dev/stdout/';
```

Intersecting Columns

A common task is to calculate the intersection of two columns, i.e., to find entities that are present in both. Let's find the cities that are both in the top-10 most populous cities and have their own airports.

Unix Shell: `comm`

The Unix solution for intersection uses the `comm` tool, intended to compare two *sorted* files line-by-line. We first `cut` the relevant colum from both files. Due to the sorting requirement, we apply `sort` on both inputs before performing the intersection. The intersection is performed using `comm -12` where the argument `-12` means that we only want to keep lines that are in both files. We again rely on `head` and `tail` to treat the headers and the rest of the files separately during processing and glue them together at the end.

```
head -n 1 pop.csv | cut -d , -f 1; \
    comm -12 \
        <(tail -n +2 pop.csv | cut -d , -f 1 | sort) \
        <(tail -n +2 cities-airports.csv | cut -d , -f 1 | sort)
```

The script produces the following output:

```
city
Amsterdam
Eindhoven
Groningen
Rotterdam
The Hague
```

DuckDB: `INTERSECT ALL`

The DuckDB solution reads the CSV files, projects the `city` fields and applies the `INTERSECT ALL` clause to calculate the intersection:

```
COPY (
    SELECT city FROM 'pop.csv'
    INTERSECT ALL
    SELECT city FROM 'cities-airports.csv'
  ) TO '/dev/stdout/';
```

Pasting Rows Together

Pasting rows together line-by-line is a recurring task. In our example, we know that the `pop.csv` and the `area.csv` files have an equal number of rows, so we can produce a single file that contains both the population and the area of every city in the dataset.

Unix Shell: `paste`

In the Unix shell, we use the `paste` command and remove the duplicate `city` field using `cut`:

```
paste -d , pop.csv area.csv | cut -d , -f 1,2,3,5
```

The output is the following:

```
city,province,population,area
Amsterdam,North Holland,905234,219.32
Rotterdam,South Holland,656050,324.14
The Hague,South Holland,552995,98.13
Utrecht,Utrecht,361924,99.21
Eindhoven,North Brabant,238478,88.92
Groningen,Groningen,234649,197.96
Tilburg,North Brabant,224702,118.13
```

```
Almere,Flevoland,218096,248.77
Breda,North Brabant,184716,128.68
Nijmegen,Gelderland,179073,57.63
```

DuckDB: `POSITIONAL JOIN`

In DuckDB, we can use a `POSITIONAL JOIN`. This join type is one of DuckDB's SQL extensions and it provides a concise syntax to combine tables row-by-row based on each row's position in the table. Joining the two tables together using `POSITIONAL JOIN` results in two `city` columns – we use the `EXCLUDE` clause to remove the duplicate column:

```
COPY (
    SELECT pop.*, area.* EXCLUDE city
    FROM 'pop.csv'
    POSITIONAL JOIN 'area.csv'
  ) TO '/dev/stdout/';
```

Filtering

Filtering is another very common operation. For this, we'll use `cities-airports.csv` file. For each airport, this file contains its `IATA` code and the main cities that it serves:

```
city,IATA
Amsterdam,AMS
Haarlemmermeer,AMS
Eindhoven,EIN
...
```

Let's try to formulate two queries:

1. Find all cities whose name ends in `dam`.
2. Find all airports whose IATA code is equivalent to the first three letters of a served city's name, but the city's name does *not* end in `dam`.

Unix Shell: `grep`

To answer the first question in the Unix shell, we use `grep` and the regular expression `^[^,]*dam,`:

```
grep "^[^,]*dam," cities-airports.csv
```

In this expression, `^` denotes the start of the line, `[^,]*` searches for a string that does not contain the comma character (the separator). The expression `dam,` ensures that the end of the string in the first field is `dam`. The output is:

```
Amsterdam,AMS
Rotterdam,RTM
```

Let's try to answer the second question. For this, we need to match the first three characters in the `city` field to the `IATA` field but we need to do so in a case-insensitive manner. We also need to use a negative condition to exclude the lines where the city's name ends in `dam`. Both of these requirements are difficult to achieve with a single `grep` or `egrep` command as they lack support for two features. First, they do not support case-insensitive matching *using a backreference* (`grep -i` alone is not sufficient to ensure this). Second, they do not support negative lookbehinds. Therefore, we use `pcregrep`, and formulate our question as follows:

```
pcregrep -i '^([a-z]{3}).*?(?<!dam),\1$' cities-airports.csv
```

Here, we call `pcregrep` with the case-insensitive flag (`-i`), which in `pcregrep` also affects backreferences such as `\1`. We capture the first three letters with `([a-z]{3})` (e.g., `Ams`) and match it to the second field with the backreference: `,\1$`. We use a non-greedy `.*?` to seek to the end of the first field, then apply a negative lookbehind with the `(?<!dam)` expression to ensure that the field does not end in `dam`. The result is a single line:

```
Eindhoven,EIN
```

DuckDB: `WHERE ... LIKE`

Let's answer the questions now in DuckDB. To answer the first question, we can use `LIKE` for pattern matching. The header should not be part of the output, so we disable it with `HEADER false`. The complete query looks like follows:

```
COPY (
    FROM 'cities-airports.csv'
    WHERE city LIKE '%dam'
  ) TO '/dev/stdout/' (HEADER false);
```

For the second question, we use string slicing to extract the first three characters, `upper` to ensure case-insensitivity, and `NOT LIKE` for the negative condition:

```
COPY (
    FROM 'cities-airports.csv'
    WHERE upper(city[1:3]) = IATA
      AND city NOT LIKE '%dam'
  ) TO '/dev/stdout/' (HEADER false);
```

These queries return exactly the same results as the solutions using `grep` and `pcregrep`.

In both of these queries, we used the FROM-first syntax. If the `SELECT` clause is omitted, the query is executed as if `SELECT *` was used, i.e., it returns all columns.

Joining Files

Joining tables is an essential task in data processing. Our next example is going to use a join to return city name–airport name combinations. This is achieved by joining the `cities-airports.csv` and the `airport-names.csv` files on their IATA code fields.

Unix Shell: `join`

Unix tools support joining files via the `join` command, which joins lines of two *sorted* inputs on a common field. To make this work, we sort the files based on their `IATA` fields, then perform the join on the first file's 2nd column (`-1 2`) and the second file's 1st column (`-2 1`). We have to omit the header for the `join` command to work, so we do just that and construct a new header with an `echo` command:

```
echo "IATA,city,airport name"; \
    join -t , -1 2 -2 1 \
        <(tail -n +2 cities-airports.csv | sort -t , -k 2,2) \
        <(tail -n +2 airport-names.csv   | sort -t , -k 1,1)
```

The result is the following:

```
IATA,city,airport name
AMS,Amsterdam,Amsterdam Airport Schiphol
AMS,Haarlemmermeer,Amsterdam Airport Schiphol
EIN,Eindhoven,Eindhoven Airport
GRQ,Eelde,Groningen Airport Eelde
GRQ,Groningen,Groningen Airport Eelde
MST,Beek,Maastricht Aachen Airport
MST,Maastricht,Maastricht Aachen Airport
RTM,Rotterdam,Rotterdam The Hague Airport
RTM,The Hague,Rotterdam The Hague Airport
```

DuckDB

In DuckDB, we load the CSV files and connect them using the `NATURAL JOIN` clause, which joins on column(s) with the same name. To ensure that the result matches with that of the Unix solution, we use the `ORDER BY ALL` clause, which sorts the result on all columns, starting from the first one, and stepping through them for tie-breaking to the last column.

```
COPY (
    SELECT "IATA", "city", "airport name"
    FROM 'cities-airports.csv'
    NATURAL JOIN 'airport-names.csv'
    ORDER BY ALL
  ) TO '/dev/stdout/';
```

Replacing Strings

You may have noticed that we are using very clean datasets. This is of course very unrealistic, so in an evil twist, let's reduce the data quality a bit:

- Replace the space in the province's name with an underscore, e.g., turning `North Holland` to `North_Holland`.
- Add thousand separating commas, e.g., turning `905234` to `905,234`.
- Change the CSV's separator to the semicolon character (`;`).

And while we're at it, also fetch the data set via HTTPS this time, using the URL `https://duckdb.org/data/cli/pop.csv`.

Unix Shell: `curl` and `sed`

In Unix, remote data sets are typically fetched via `curl`. The output of `curl` is piped into the subsequent processing steps, in this case, a bunch of `sed` commands.

```
curl -s https://duckdb.org/data/cli/pop.csv \
    | sed 's/\([^,]*,.*\) \(.*,[^,]*\)/\1_\2/g' \
    | sed 's/,/;/g' \
    | sed 's/\([0-9][0-9][0-9]\)$/,\1/'
```

This results in the following output:

```
city;province;population
Amsterdam;North_Holland;905,234
Rotterdam;South_Holland;656,050
The Hague;South_Holland;552,995
Utrecht;Utrecht;361,924
Eindhoven;North_Brabant;238,478
Groningen;Groningen;234,649
Tilburg;North_Brabant;224,702
Almere;Flevoland;218,096
Breda;North_Brabant;184,716
Nijmegen;Gelderland;179,073
```

DuckDB: `httpfs` and `regexp_replace`

In DuckDB, we use the following query:

```
COPY (
    SELECT
        city,
        replace(province, ' ', '_') AS province,
        regexp_replace(population::VARCHAR, '([0-9][0-9][0-9])$', ',\1')
            AS population
    FROM 'https://duckdb.org/data/cli/pop.csv'
  ) TO '/dev/stdout/' (DELIMITER ';');
```

Note that the FROM clause now has an HTTPS URL instead of a simple CSV file. The presence of the `https://` prefix triggers DuckDB to load the `httpfs` extension and use it to fetch the JSON document. We use the `replace` function to substitute the spaces with underscores, and the `regexp_replace` function for the replacement using a regular expression. (We could have also used string formatting functions

such as `format` and `printf`). To change the separator to a semicolon, we serialize the file using the COPY statement with the `DELIMITER ';'` option.

Reading JSON

As a final exercise, let's query the number of stars given to the duckdb/duckdb repository on GitHub.

Unix Shell: `curl` and `jq`

In Unix tools, we can use `curl` to get the JSON file from `https://api.github.com` and pipe its output to `jq` to query the JSON object.

```
curl -s https://api.github.com/repos/duckdb/duckdb \
    | jq ".stargazers_count"
```

DuckDB: `read_json`

In DuckDB, we use the `read_json` function, invoking it with the remote HTTPS endpoint's URL. The schema of the JSON file is detected automatically, so we can simply use SELECT to return the required field.

```
SELECT stargazers_count
  FROM read_json('https://api.github.com/repos/duckdb/duckdb');
```

Output

Both of these commands return the current number of stars of the repository.

Performance

At this point, you might be wondering about the performance of the DuckDB solutions. After all, all of our prior examples have only consisted of a few lines, so benchmarking them against each other will not result in any measurable performance differences. So, let's switch to the Dutch railway services dataset that we used in a previous blog post and formulate a different problem.

We'll use the 2023 railway services file (`services-2023.csv.gz`) and count the number of Intercity services that operated in that year.

In Unix, we can use the `gzcat` command to decompress the `csv.gz` file into a pipeline. Then, we can use `grep` or `pcregrep` (which is more performant), and top it off with the `wc` command to count the number of lines (`-l`). In DuckDB, the built-in CSV reader also supports compressed CSV files, so we can use that without any extra configuration.

```
gzcat services-2023.csv.gz | grep '^[^,]*,[^,]*,Intercity,' | wc -l
gzcat services-2023.csv.gz | pcregrep '^[^,]*,[^,]*,Intercity,' | wc -l
duckdb -c "SELECT count(*) FROM 'services-2023.csv.gz' WHERE \"Service:Type\" = 'Intercity';"
```

We also test the tools on uncompressed input:

```
gunzip -k services-2023.csv.gz
grep '^[^,]*,[^,]*,Intercity,' services-2023.csv | wc -l
pcregrep '^[^,]*,[^,]*,Intercity,' services-2023.csv | wc -l
duckdb -c "SELECT count(*) FROM 'services-2023.csv' WHERE \"Service:Type\" = 'Intercity';"
```

To reduce the noise in the measurements, we used the `hyperfine` benchmarking tool and took the mean execution time of 10 runs. The experiments were carried out on a MacBook Pro with a 12-core M2 Pro CPU and 32 GB RAM, running macOS Sonoma 14.5. To reproduce them, run the `grep-vs-duckdb-microbenchmark.sh` script. The following table shows the runtimes of the solutions on both compressed and uncompressed inputs:

Tool	Runtime (compressed)	Runtime (uncompressed)
grep 2.6.0-FreeBSD	20.9 s	20.5 s
pcregrep 8.45	3.1 s	2.9 s
DuckDB 1.0.0	4.2 s	1.2 s

The results show that on compressed input, `grep` was the slowest, while DuckDB is slightly edged out by `gzcat+pcregrep`, which ran in 3.1 seconds compared to DuckDB's 4.2 seconds. On uncompressed input, DuckDB can utilize all CPU cores from the get-go (instead of starting with a single-threaded decompression step), allowing it to outperform both `grep` and `pcregrep` by a significant margin: 2.5× faster than `pcregrep` and more than 15× faster than `grep`.

While this example is quite simple, as queries get more complex, there are more opportunities for optimization and larger intermediate dataset may be produced. While both of these can be tackled within a shell script (by manually implementing optimizations and writing the intermediate datasets to disk), these will likely be less efficient than what a DBMS can come up with. Shell scripts implementing complex pipelines can also be very brittle and need to be rethought even for small changes, making the performance advantage of using a database even more significant for more complex problems.

Summary

In this post, we used DuckDB as a standalone CLI application, and explored its abilities to complement or substitute existing command line tools (`sort`, `grep`, `comm`, `join`, etc.). While we obviously like DuckDB a lot and prefer to use it in many cases, we also believe Unix tools have their place: on most systems, they are already pre-installed and a well-chosen toolchain of Unix commands *can* be fast, efficient, and portable (thanks to POSIX-compliance). Additionally, they can be very concise for certain problems. However, to reap their benefits, you will need to learn the syntax and quirks of each tool such as `grep` variants, awk as well as advanced ones such as `xargs` and `parallel`. In the meantime, DuckDB's SQL is easy-to-learn (you likely know quite a bit of it already) and DuckDB handles most of the optimization for you.

If you have a favorite CLI use case for DuckDB, let us know on social media or submit it to DuckDB snippets. Happy hacking!

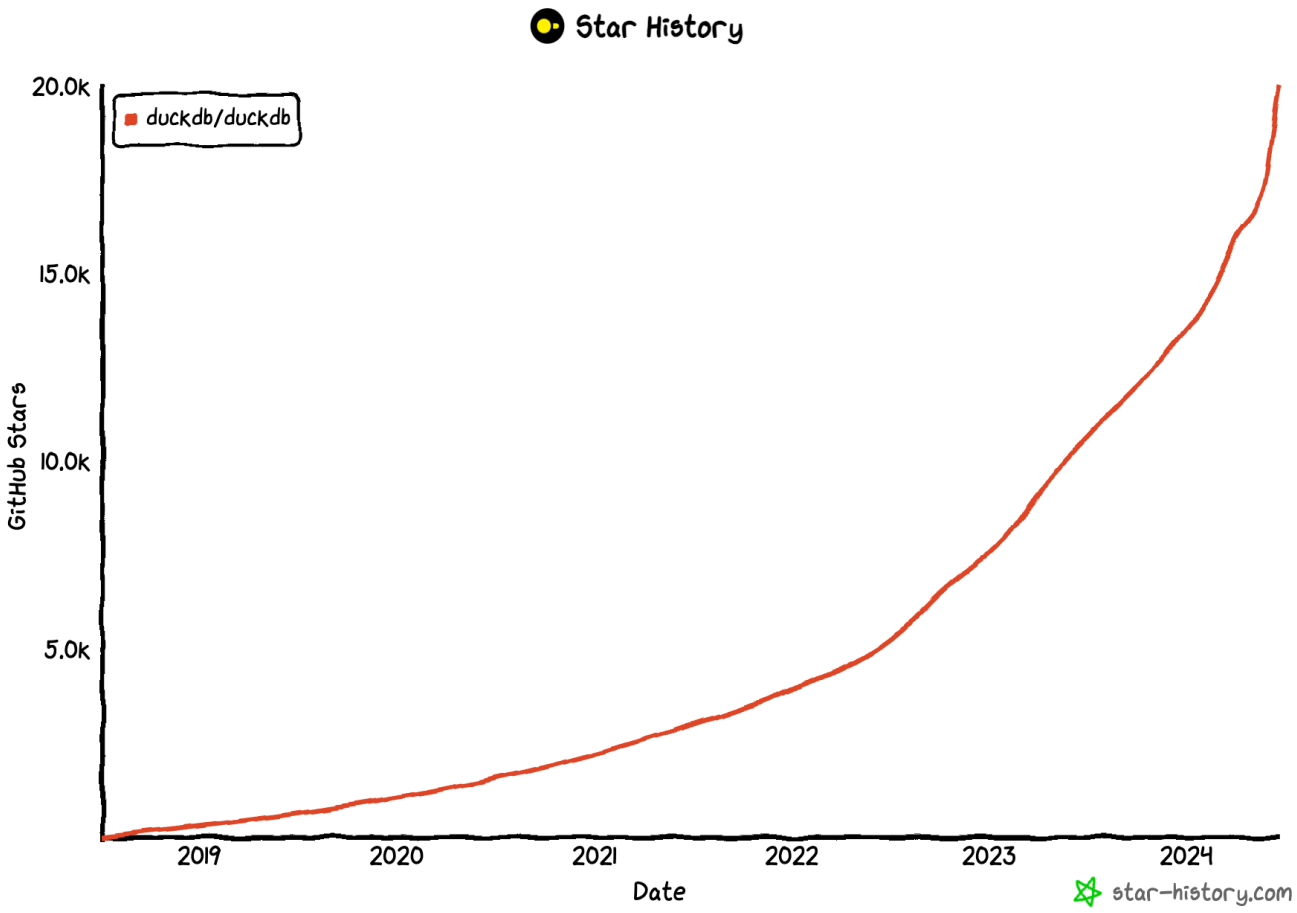
20 000 Stars on GitHub

Publication date: 2024-06-22

Author: The DuckDB Team

DuckDB reached 20 000 stars today on GitHub. We would like to thank our amazing community of users and contributors.

To this day, we continue to be amazed by the adoption of DuckDB. We hit the previous milestone, 10 000 stars just a little over a year ago. Since then, the growth of stars has been slowly increasing, until the release of version 1.0.0 in early June gave the boost that propelled the star count to 20 000.



(image source: star-history.com)

What Else Happened in June?

The last few weeks since the release were quite eventful:

1. MotherDuck, a DuckDB-based cloud warehouse, just reached General Availability last week. Congratulations to the team on the successful release!

2. We added support to DuckDB for Delta Lake, an open-source lakehouse framework. This feature was described in Sam Ansmink's blog post and Hannes Mühleisen's keynote segment at the DATA+AI summit.

   With extensions for both Delta Lake and Iceberg, DuckDB can now read the two most popular data lake formats.

3. We ran a poster campaign for DuckDB in Amsterdam:

4. DuckDB Labs sponsored the Hack4Her event, a female-focused student hackathon in the Netherlands. During the DuckDB Challenge of the event, teams built a community-driven app providing safe walking routes in Amsterdam using DuckDB and its geospatial library.

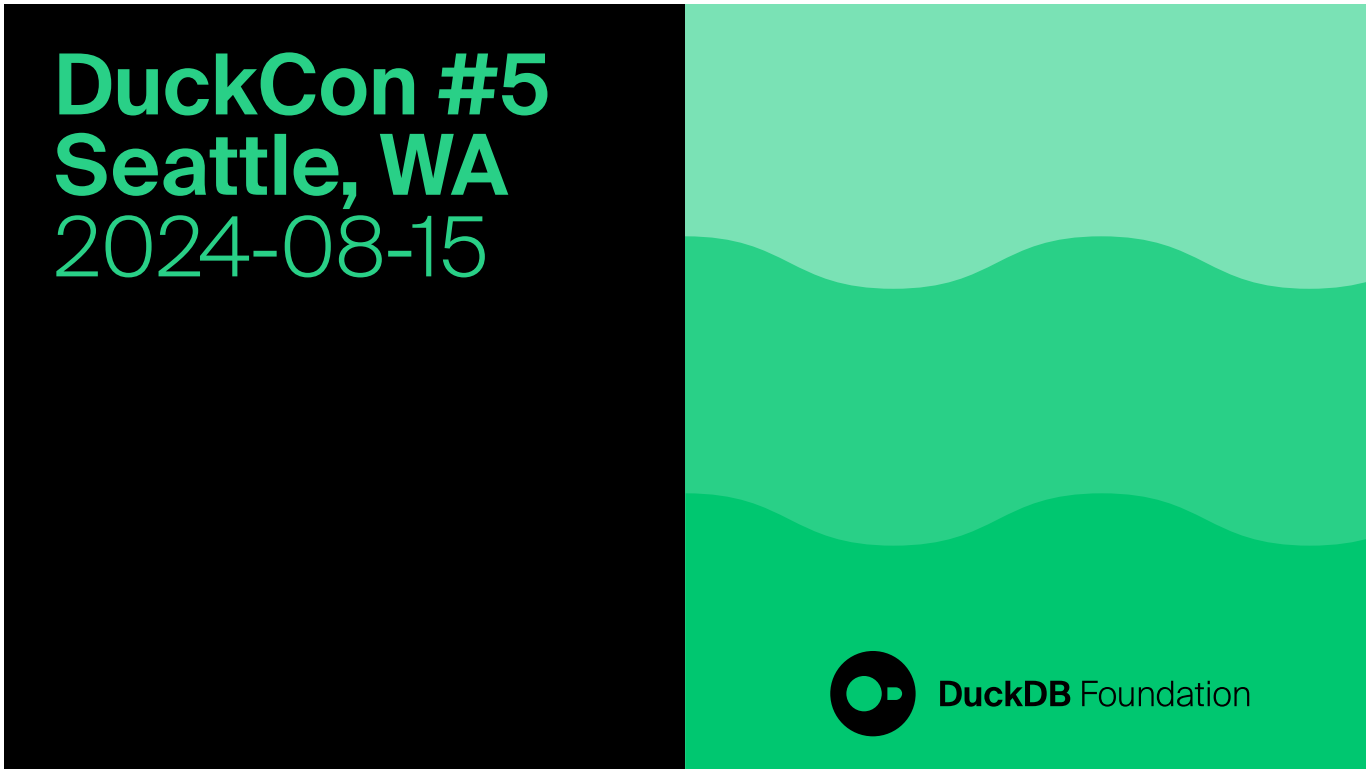
Looking Ahead

There are several interesting events lined up for the summer.

First, two books about DuckDB are expected to be released:

- **Getting Started with DuckDB**, authored by Simon Aubury and Ned Letcher, and published by Packt Publishing
- **DuckDB in Action**, authored by Mark Needham, Michael Hunger and Michael Simons, and published by Manning Publications

Second, we are holding our next user community meeting, DuckCon #5 in Seattle on August 15 with the regular ”State of the Duck” update as well as three regular talks and several lightning talks.



Third, we will improve DuckDB's extension ecosystem and streamline the publication process for community extensions.

Finally, we have a series of blog posts lined up for publication. These will discuss DuckDB's performance over time, the results of the user survey we conducted during the spring, DuckDB's storage format, and many more. Stay tuned!

We are looking forward to next part of our journey and, of course, the next 10 000 stars on GitHub.

Benchmarking Ourselves over Time at DuckDB

Publication date: 2024-06-26

Author: Alex Monahan

TL;DR: In the last 3 years, DuckDB has become 3-25× faster and can analyze ~10× larger datasets all on the same hardware.

A big part of DuckDB's focus is on the developer experience of working with data. However, performance is an important consideration when investigating data management systems. Fairly comparing data processing systems using benchmarks is very difficult. Whoever creates the benchmark is likely to know one system better than the rest, influencing benchmark selection, how much time is spent tuning parameters, and more.

Instead, this post focuses on benchmarking *our own* performance over time.

This approach avoids many comparison pitfalls, and also provides several valuable data points to consider when selecting a system.

- **How fast is it improving?** Learning a new tool is an investment. Picking a vibrant, rapidly improving database ensures your choice pays dividends for years to come. Plus, if you haven't experimented with a tool in a while, you can see how much faster it has become since you last checked!
- **What is it especially good at?** The choice of benchmark is an indicator of what types of workloads a tool is useful for. The higher the variety of analyses in the benchmark, the more broadly useful the tool can be.
- **What scale of data can it handle?** Many benchmarks are deliberately smaller than typical workloads. This allows the benchmark to complete in a reasonable amount of time when run with many configurations. However, an important question to answer when selecting a system is whether the size of your data can be handled within the size of your compute resources.

There are some limitations when looking at the performance of a system over time. If a feature is brand new, there is no prior performance to compare to! As a result, this post focuses on fundamental workloads rather than DuckDB's ever-increasing set of integrations with different lakehouse data formats, cloud services, and more.

The code used to run the benchmark also avoids many of DuckDB's Friendlier SQL additions, as those have also been added more recently. (When writing these queries, it felt like going back in time!)

Benchmark Design Summary

This post measures DuckDB's performance over time using the H2O.ai benchmark, plus some new benchmarks added for importing, exporting, and using window functions. Please see our previous blog posts for details on why we believe the H2O.ai benchmark is a good approach! The full details of the benchmark design are in the appendix.

- H2O.ai, plus import/export and window function tests
- Python instead of R
- 5 GB scale for everything, plus 50 GB scale for group bys and joins
- Median of 3 runs
- Using a MacBook Pro M1 with 16 GB RAM
- DuckDB versions 0.2.7 through 1.0.0
  - Nearly 3 years, from 2021-06-14 to 2024-06-03
- Default settings
- Pandas pre-version 0.5.1, Apache Arrow 0.5.1+

Overall Benchmark Results

The latest DuckDB can complete one run of the full benchmark suite in under 35 seconds, while version 0.2.7 required nearly 500 seconds for the same task in June 2021. **That is 14 times faster, in only 3 years!**

Performance over Time

> **Note.** These graphs are interactive, thanks to Plotly.js! Feel free to filter the various series (single click to hide, double click to show only that series) and click-and-drag to zoom in. Individual benchmark results are visible on hover.

The above plot shows the median runtime in seconds for all tests. Due to the variety of uses for window functions, and their relative algorithmic complexity, the 16 window function tests require the most time of any category.

This plot normalizes performance to the latest version of DuckDB to show relative improvements over time. If you look at the point in time when you most recently measured DuckDB performance, that number will show you how many times faster DuckDB is now!

A portion of the overall improvement is DuckDB's addition of multi-threading, which became the default in November 2021 with version 0.3.1. DuckDB also moved to a push-based execution model in that version for additional gains. Parallel data loading boosted performance in December 2022 with version 0.6.1, as did improvements to the core `JOIN` algorithm. We will explore other improvements in detail later in the post.

However, we see that all aspects of the system have seen improvements, not just raw query performance! DuckDB focuses on the entire data analysis workflow, not just aggregate or join performance. CSV parsing has seen significant gains, import and export have improved significantly, and window functions have improved the most of all.

What was the slight regression from December 2022 to June 2023? Window functions received additional capabilities and experienced a slight performance degradation in the process. However, from June 2023 onward we see substantial performance improvement across the board for window functions. If window functions are filtered out of the chart, we see a smoother trend.

You may also notice that starting with version 0.9 in September 2023, the performance appears to plateau. What is happening here? First, don't forget to zoom in! Over the last year, DuckDB has still improved over 3×! More recently, the DuckDB Labs team focused on scalability by developing algorithms that support larger-than-memory calculations. We will see the fruits of those labors in the scale section later on! In addition, DuckDB focused exclusively on bug fixes in versions 0.10.1, 0.10.2, and 0.10.3 in preparation for an especially robust DuckDB 1.0. Now that those two major milestones (larger than memory calculations and DuckDB 1.0) have been accomplished, performance improvements will resume! It is worth noting that the boost from moving to multi-threading will only occur once, but there are still many opportunities moving forward.

Performance by Version

We can also recreate the overall plot by version rather than by time. This demonstrates that DuckDB has been doing more frequent releases recently. See DuckDB's release calendar for the full version history.

If you remember the version that you last tested, you can compare how much faster things are now with 1.0!

Results by Workload

CSV Reader

DuckDB has invested substantially in building a fast and robust CSV parser. This is often the first task in a data analysis workload, and it tends to be undervalued and underbenchmarked. DuckDB has **improved CSV reader performance by nearly 3×**, while adding the ability to handle many more CSV dialects automatically.

Group By

Group by or aggregation operations are critical steps in OLAP workloads, and have therefore received substantial focus in DuckDB, **improving over 12× in the last 3 years**.

In November 2021, version 0.3.1 enabled multithreaded aggregation by default, providing a significant speedup.

In December 2022, data loads into tables were parallelized with the release of version 0.6.1. This is another example of improving the entire data workflow, as this group by benchmark actually stressed the insertion performance substantially. Inserting the results was taking the majority of the time!

Enums were also used in place of strings for categorical columns in version 0.6.1. This means that DuckDB was able to use integers rather than strings when operating on those columns, further boosting performance.

Despite what appears at first glance to be a performance plateau, zooming in to 2023 and 2024 reveals a ~20% improvement. In addition, aggregations have received significant attention in the most recent versions to enable larger-than-memory aggregations. You can see that this was achieved while continuing to improve performance for the smaller-than-memory case.

Join

Join operations are another area of focus for analytical databases, and DuckDB in particular. Join speeds have **improved by 4× in the last 3 years**!

Version 0.6.1 in December 2022 introduced improvements to the out-of-core hash join that actually improved the smaller-than-memory case as well. Parallel data loading from 0.6.1 also helps in this benchmark as well, as some results are the same size as the input table.

In recent versions, joins have also been upgraded to support larger-than-memory capabilities. This focus has also benefitted the smaller-than-memory case and has led to the improvements in 0.10, launched in February 2024.

Window Functions

Over the time horizon studied, window functions have **improved a dramatic 25×**!

Window function performance was improved substantially with the 0.9.0 release in September 2023. 14 different performance optimizations contributed. Aggregate computation was vectorized (with special focus on the segment tree data structure). Work stealing enabled multi-threaded processing and sorting was adapted to run in parallel. Care was also taken to pre-allocate memory in larger batches.

DuckDB's window functions are also capable of processing larger-than-memory datasets. We leave benchmarking that feature for future work!

Export

Often DuckDB is not the final step in a workflow, so export performance has an impact. Exports are **10× faster now!** Until recently, the DuckDB format was not backward compatible, so the recommended long term persistence format was Parquet. Parquet is also critical to interoperability with many other systems, especially data lakes. DuckDB works well as a workflow engine, so exporting to other in-memory formats is quite common as well.

In the September 2022 release (version 0.5.1) we see significant improvements driven by switching from Pandas to Apache Arrow as the recommended in-memory export format. DuckDB's underlying data types share many similarities with Arrow, so data transfer is quite quick.

Parquet export performance has improved by 4–5× over the course of the benchmark, with dramatic improvements in versions 0.8.1 (June 2023) and 0.10.2 (April 2024). Version 0.8.1 added parallel Parquet writing while continuing to preserve insertion order.

The change driving the improvement in 0.10.2 was more subtle. When exporting strings with high cardinality, DuckDB decides whether or not to do dictionary compression depending on if it reduces file size. From 0.10.2 onward, the compression ratio is tested after a sample of the values are inserted into the dictionary, rather than after all values are added. This prevents substantial unnecessary processing for high-cardinality columns where dictionary compression is unhelpful.

Exporting Apache Arrow vs. Pandas vs. Parquet

This plot shows the performance of all three export formats over the entire time horizon (rather than picking the winner between Pandas and Arrow). It allows us to see at what point Apache Arrow passes Pandas in performance.

Pandas export performance has improved substantially over the course of the benchmark. However, Apache Arrow has proven to be the more efficient data format, so Arrow is now preferred for in-memory exports. Interestingly, DuckDB's Parquet export is now so efficient that it is faster to write a persistent Parquet file than it is to write to an in-memory Pandas dataframe! It is even competitive with Apache Arrow.

Scan Other Formats

In some use cases, DuckDB does not need to store the raw data, but instead should simply read and analyze it. This allows DuckDB to fit seamlessly into other workflows. This benchmark measures how fast DuckDB can scan and aggregate various data formats.

To enable comparisons over time, we switch from Pandas to Arrow at version 0.5.1 as mentioned. DuckDB is **over 8× faster in this workload**, and the absolute time required is very short. DuckDB is a great fit for this type of work!

Scanning Apache Arrow vs. Pandas vs. Parquet

Once again, we examine all three formats over the entire time horizon.

When scanning data, Apache Arrow and Pandas are more comparable in performance. As a result, while Arrow is clearly preferable for exports, DuckDB will happily read Pandas with similar speed. However, in this case, the in-memory nature of both Arrow and Pandas allow them to perform 2–3× faster than Parquet. In absolute terms, the time required to complete this operation is a very small fraction of the benchmark, so other operations should be the deciding factor.

Scale tests

Analyzing larger-than-memory data is a superpower for DuckDB, allowing it to be used for substantially larger data analysis tasks than were previously possible.

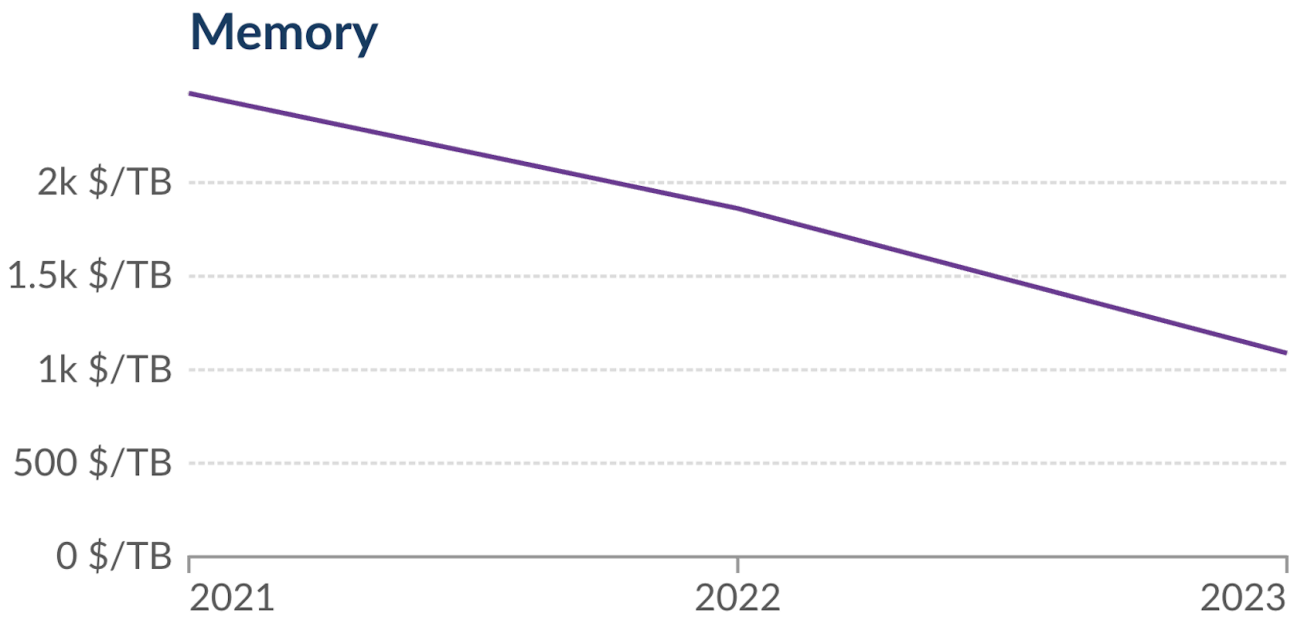
In version 0.9.0, launched in September 2023, DuckDB's hash aggregate was enhanced to handle out-of-core (larger than memory) intermediates. The details of the algorithm, along with some benchmarks, are available in this blog post. This allows for DuckDB to aggregate one billion rows of data (50 GB in size) on a MacBook Pro with only 16 GB of RAM, even when the number of unique groups in the group by is large. This represents at least a 10× improvement in aggregate processing scale over the course of the 3 years of the benchmark.

DuckDB's hash join operator has supported larger-than-memory joins since version 0.6.1 in December 2022. However, the scale of this benchmark (coupled with the limited RAM of the benchmarking hardware), meant that this benchmark could still not complete successfully. In version 0.10.0, launched in February 2024, DuckDB's memory management received a significant upgrade to handle multiple concurrent operators all requiring significant memory. The 0.10.0 release blog post shares additional details about this feature.

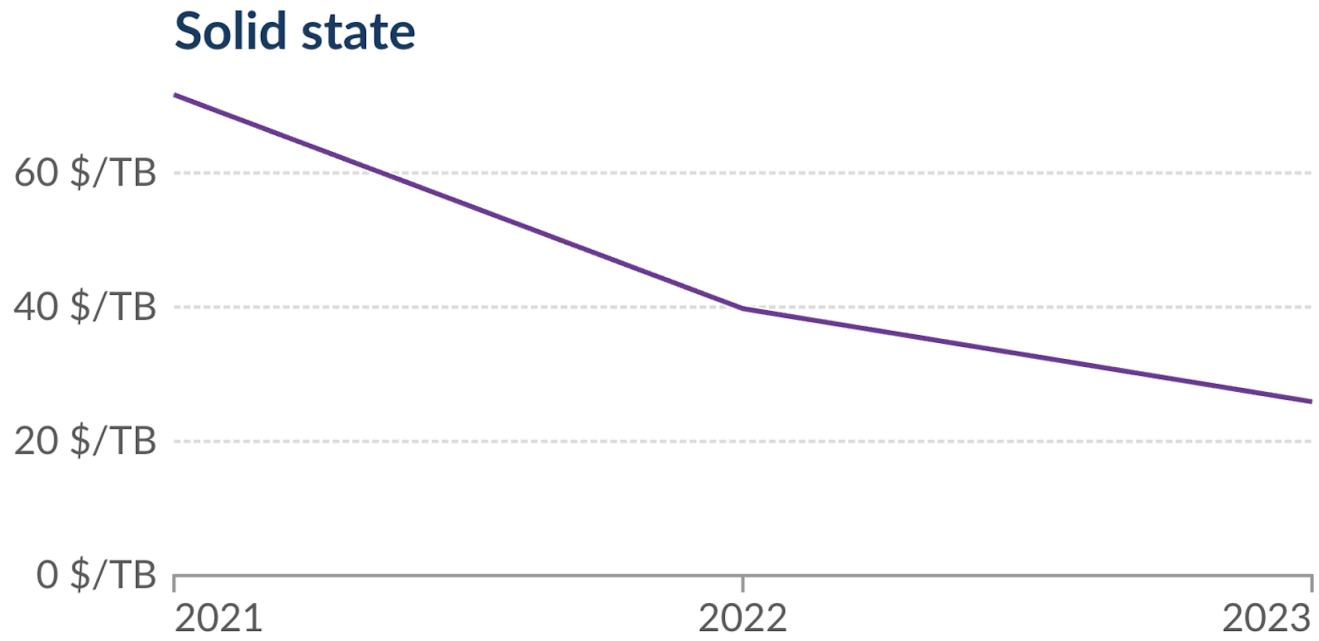
As a result, by version 0.10.0 DuckDB was able to handle calculations on data that is significantly larger than memory, even if the intermediate calculations are large in size. All operators are supported, including sorting, aggregating, joining, and windowing. Future work can further test the boundaries of what is possible with DuckDB's out-of-core support, including window functions and even larger data sizes.

Hardware Capabilities over Time

DuckDB's performance on the same hardware has improved dramatically, and at the same time, the capabilities of hardware are increasing rapidly as well.



{:



width="360"}
width="360" }

Source: Our World in Data

The price of RAM has declined by 2.2× and the price of SSD storage has decreased by 2.7× from 2021 to 2023 alone. Thanks to the combination of DuckDB enhancements and hardware prices, the scale of analysis possible on a single node has increased by substantially more than an order of magnitude in just 3 years!

Analyzing the Results Yourself

A DuckDB 1.0 database containing the results of these benchmarks is available at https://blobs.duckdb.org/data/duckdb_perf_over_time.duckdb. Any DuckDB client with the `httpfs` extension can read that file.

You can even use the DuckDB Wasm web shell to **query the file directly from your browser** (with the queries pre-populated and automatically executed!):

```
LOAD httpfs;
ATTACH 'https://blobs.duckdb.org/data/duckdb_perf_over_time.duckdb' AS performance_results;
USE performance_results;
```

The file contains two tables: `benchmark_results` and `scale_benchmark_results`. Please let us know if you uncover any interesting findings!

Conclusion

In summary, not only is DuckDB's feature set growing substantially with each release, DuckDB is getting faster very fast! Overall, performance has **improved by 14× in only 3 years!**

Yet query performance is only part of the story! The variety of workloads that DuckDB can handle is wide and growing wider thanks to a full-featured SQL dialect, including high performance window functions. Additionally, critical workloads like data import, CSV parsing, and data export have improved dramatically over time. The complete developer experience is critical for DuckDB!

Finally, DuckDB now supports larger-than-memory calculations across all operators: sorting, aggregating, joining, and windowing. The size of problem that you can handle on your current compute resources just got 10× bigger, or more!

If you have made it this far, welcome to the flock! 🦆 Join us on Discord, we value your feedback!

Appendix

Benchmark Design

H2O.ai as the Foundation

This post measures DuckDB's performance over time on the H2O.ai benchmark for both joins and group by queries.

The result of each H2O.ai query is written to a table in a persistent DuckDB file. This does require additional work when compared with an in-memory workflow (especially the burden on the SSD rather than RAM), but improves scalability and is a common approach for larger analyses.

As in the current H2O.ai benchmark, categorical-type columns (VARCHAR columns with low cardinality) were converted to the ENUM type as a part of the benchmark. The time for converting into ENUM columns was included in the benchmark time, and resulted in a lower total amount of time (so the upfront conversion was worthwhile). However, the ENUM data type was not fully operational in DuckDB until version 0.6.1 (December 2022), so earlier versions skip this step.

Python Client

To measure interoperability with other dataframe formats, we have used Python rather than R (used by H2O.ai) for this analysis. We do continue to use R for the data generation step for consistency with the benchmark. Python is DuckDB's most popular client, great for data science, and also the author's favorite language for this type of work.

Export and Replacement Scans

We now extend this benchmark in several important ways. In addition to considering raw query performance, we measure import and export performance with several formats: Pandas, Apache Arrow, and Apache Parquet. The results of both the join and group by benchmarks are exported to each format.

When exporting to dataframes, we measured the performance in both cases. However, when summarizing the total performance, we chose the best performing format at the time. This likely mirrors the behavior of performance-sensitive users (as they would likely not write to both formats!). In version 0.5.1, released September 2022, DuckDB's performance when writing to and reading from the Apache Arrow format surpassed Pandas. As a result, versions 0.2.7 to 0.4.0 use Pandas, and 0.5.1 onward uses Arrow.

On the import side, replacement scans allow DuckDB to read those same formats without a prior import step. In the replacement scan benchmark, the data that is scanned is the output of the final H2O.ai group by benchmark query. At the 5 GB scale it is a 100 million row dataset. Only one column is read, and a single aggregate is calculated. This focuses the benchmark on the speed of scanning the data rather than DuckDB's aggregation algorithms or speed of outputting results. The query used follows the format:

```
SELECT
    sum(v3) AS v3
FROM <dataframe_or_Parquet_file>;
```

Window Functions

We also added an entire series of window function benchmarks. Window functions are a critical workload in real world data analysis scenarios, and can stress test a system in other ways. DuckDB has implemented state of the art algorithms to quickly process even the most complex window functions. We use the largest table from the join benchmark as the raw data for these new tests to help with comparability to the rest of the benchmark.

Window function benchmarks are much less common than more traditional joins and aggregations, and we were unable to find a suitable suite off the shelf. These queries were designed to showcase the variety of uses for window functions, but there are certainly more that could be added. We are open to your suggestions for queries to add, and hope these queries could prove useful for other systems!

Since the window functions benchmark is new, the window functions from each of the queries included are shown in the appendix at the end of the post.

Workload Size

We test only the middle 5 GB dataset size for the workloads mentioned thus far, primarily because some import and export operations to external formats like Pandas must fit in memory (and we used a MacBook Pro M1 with only 16 GB of RAM). Additionally, running the tests for 21 DuckDB versions was time-intensive even at that scale, due to the performance of older versions.

Scale Tests

Using only 5 GB of data does not answer our second key question: “What scale of data can it handle?”! We also ran only the group by and join related operations (avoiding in-memory imports and exports) at the 5 GB and the 50 GB scale. Older versions of DuckDB could not handle the 50 GB dataset when joining or aggregating, but modern versions can handle both, even on a memory-constrained 16 GB RAM laptop. Instead of measuring performance, we measure the size of the benchmark that was able to complete on a given version.

Summary Metrics

With the exception of the scale tests, each benchmark was run 3 times and the median time was used for reporting results. The scale tests were run once and produced a binary metric, success or failure, at each data size tested. As older versions would not fail gracefully, the scale metrics were accumulated across multiple partial runs.

Computing Resources

All tests use a MacBook Pro M1 with 16 GB of RAM. In 2024, this is far from state of the art! If you have more powerful hardware, you will see both improved performance and scalability.

DuckDB Versions

Version 0.2.7, published in June 2021, was the first version to include a Python client compiled for ARM64, so it was the first version that could easily run on the benchmarking compute resources. Version 1.0.0 is the latest available at the time of publication (June 2024), although we also provide a sneak preview of an in-development feature branch.

Default Settings

All versions were run with the default settings. As a result, improvements from a new feature only appear in these results once that feature became the default and was therefore ready for production workloads.

Window Functions Benchmark

Each benchmark query follows the format below, but with different sets of window functions in the `<window_function(s)>{:.language-sql .highlight}` placeholder. The table in use is the largest table from the H2O.ai join benchmark, and in this case the 5 GB scale was used.

```
DROP TABLE IF EXISTS windowing_results;

CREATE TABLE windowing_results AS
    SELECT
        id1,
        id2,
        id3,
        v2,
        <window_function(s)>
    FROM join_benchmark_largest_table;
```

The various window functions that replace the placeholder are below and are labelled to match the result graphs. These were selected to showcase the variety of use cases for window functions, as well as the variety of algorithms required to support the full range of the syntax. The DuckDB documentation contains a full railroad diagram of the available syntax. If there are common use cases for window functions that are not well-covered in this benchmark, please let us know!

```
/* 302 Basic Window */
sum(v2) OVER () AS window_basic

/* 303 Sorted Window */
first(v2) OVER (ORDER BY id3) AS first_order_by,
row_number() OVER (ORDER BY id3) AS row_number_order_by

/* 304 Quantiles Entire Dataset */
quantile_cont(v2, [0, 0.25, 0.50, 0.75, 1]) OVER ()
    AS quantile_entire_dataset

/* 305 PARTITION BY */
sum(v2) OVER (PARTITION BY id1) AS sum_by_id1,
sum(v2) OVER (PARTITION BY id2) AS sum_by_id2,
sum(v2) OVER (PARTITION BY id3) AS sum_by_id3

/* 306 PARTITION BY ORDER BY */
first(v2) OVER
    (PARTITION BY id2 ORDER BY id3) AS first_by_id2_ordered_by_id3

/* 307 Lead and Lag */
first(v2) OVER
    (ORDER BY id3 ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING)
    AS my_lag,
```

```
first(v2) OVER
    (ORDER BY id3 ROWS BETWEEN 1 FOLLOWING AND 1 FOLLOWING)
    AS my_lead

/* 308 Moving Averages */
avg(v2) OVER
    (ORDER BY id3 ROWS BETWEEN 100 PRECEDING AND CURRENT ROW)
    AS my_moving_average,
avg(v2) OVER
    (ORDER BY id3 ROWS BETWEEN id1 PRECEDING AND CURRENT ROW)
    AS my_dynamic_moving_average

/* 309 Rolling Sum */
sum(v2) OVER
    (ORDER BY id3 ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
    AS my_rolling_sum

/* 310 RANGE BETWEEN */
sum(v2) OVER
    (ORDER BY v2 RANGE BETWEEN 3 PRECEDING AND CURRENT ROW)
    AS my_range_between,
sum(v2) OVER
    (ORDER BY v2 RANGE BETWEEN id1 PRECEDING AND CURRENT ROW)
    AS my_dynamic_range_between

/* 311 Quantiles PARTITION BY */
quantile_cont(v2, [0, 0.25, 0.50, 0.75, 1])
    OVER (PARTITION BY id2)
    AS my_quantiles_by_id2

/* 312 Quantiles PARTITION BY ROWS BETWEEN */
first(v2) OVER
    (PARTITION BY id2 ORDER BY id3 ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING)
    AS my_lag_by_id2,
first(v2) OVER
    (PARTITION BY id2 ORDER BY id3 ROWS BETWEEN 1 FOLLOWING AND 1 FOLLOWING)
    AS my_lead_by_id2

/* 313 Moving Averages PARTITION BY */
avg(v2) OVER
    (PARTITION BY id2 ORDER BY id3 ROWS BETWEEN 100 PRECEDING AND CURRENT ROW)
    AS my_moving_average_by_id2,
avg(v2) OVER
    (PARTITION BY id2 ORDER BY id3 ROWS BETWEEN id1 PRECEDING AND CURRENT ROW)
    AS my_dynamic_moving_average_by_id2

/* 314 Rolling Sum PARTITION BY */
sum(v2) OVER
    (PARTITION BY id2 ORDER BY id3 ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
    AS my_rolling_sum_by_id2

/* 315 RANGE BETWEEN PARTITION BY */
sum(v2) OVER
    (PARTITION BY id2 ORDER BY v2 RANGE BETWEEN 3 PRECEDING AND CURRENT ROW)
    AS my_range_between_by_id2,
sum(v2) OVER
    (PARTITION BY id2 ORDER BY v2 RANGE BETWEEN id1 PRECEDING AND CURRENT ROW)
    AS my_dynamic_range_between_by_id2

/* 316 Quantiles PARTITION BY ROWS BETWEEN */
```

```
quantile_cont(v2, [0, 0.25, 0.50, 0.75, 1]) OVER
    (PARTITION BY id2 ORDER BY id3 ROWS BETWEEN 100 PRECEDING AND CURRENT ROW)
    AS my_quantiles_by_id2_rows_between
```

DuckDB Community Extensions

Publication date: 2024-07-05

Author: The DuckDB team

TL;DR: DuckDB extensions can now be published as DuckDB Community Extensions. The repository makes it easier for users to install extensions using the `INSTALL extension_name FROM community` syntax. Extension developers avoid the burdens of compilation and distribution.

> To browse existing community extensions, visit the DuckDB Community Extensions documentation page.

DuckDB Extensions

Design Philosophy

One of the main design goals of DuckDB is *simplicity*, which – to us – implies that the system should be rather nimble, very light on dependencies, and generally small enough to run on constrained platforms like WebAssembly. This goal is in direct conflict with very reasonable user requests to support advanced features like spatial data analysis, vector indexes, connectivity to various other databases, support for data formats, etc. Baking all those features into a monolithic binary is certainly possible and the route some systems take. But we want to preserve DuckDB's simplicity. Also, shipping all possible features would be quite excessive for most users because no use cases require *all* extensions at the same time (the "Microsoft Word paradox", where even power users only use a few features of the system, but the exact set of features vary between users).

To achieve this, DuckDB has a powerful extension mechanism, which allows users to add new functionalities to DuckDB. This mechanism allows for registering new functions, supporting new file formats and compression methods, handling new network protocols, etc. In fact, many of DuckDB's popular features are implemented as extensions: the Parquet reader, the JSON reader, and the HTTPS/S3 connector all use the extension mechanism.

Using Extensions

Since version 0.3.2, we have already greatly simplified the discovery and installation by hosting them on a centralized extension repository. So, for example, to install the spatial extension, one can just run the following commands using DuckDB's SQL interface:

```
INSTALL spatial; -- once
LOAD spatial;    -- on each use
```

What happens behind the scenes is that DuckDB downloads an extension binary suitable to the current operating system and processor architecture (e.g., macOS on ARM64) and stores it in the `~/.duckdb` folder. On each `LOAD`, this file is loaded into the running DuckDB instance, and things happily continue from there. Of course, for this to work, we compile, sign and host the extensions for a rather large and growing list of processor architecture – operating system combinations. This mechanism is already heavily used, currently, we see around six million extension downloads *each week* with a corresponding data transfer volume of around 40 terabytes!

Until now, publishing third-party extensions has been a *difficult process* which required the extension developer to build the extensions in their repositories for a host of platforms. Moreover, they were unable to sign the extensions using official keys, forcing users to use the `allow_unsigned_extensions` option that disables signature checks which is problematic in itself.

DuckDB Community Extensions

Distributing software in a safe way has never been easier, allowing us to reach a wide base of users across pip, conda, cran, npm, brew, etc. We want to provide a similar experience both to users who can easily grab the extension they will want to use, and developers who should not be burdened with distribution details. We are also interested in lowering the bar to package utilities and scripts as a DuckDB extension, empowering users to package useful functionality connected to their area of expertise (or pain points).

We believe that fostering a community extension ecosystem is the next logical step for DuckDB. That's why we're very excited about launching our Community Extensions repository which was announced at the Data + AI Summit.

For users, this repository allows for easy discovery, installation and maintenance of community extensions directly from the DuckDB SQL prompt. For developers, it greatly streamlines the publication process of extensions. In the following, we'll discuss how the new extension repository enhances the experiences of these groups.

User Experience

We are going to use the h3 extension as our example. This extension implements hierarchical hexagonal indexing for geospatial data.

Using the DuckDB Community Extensions repository, you can now install and load the h3 extension as follows:

```
INSTALL h3 FROM community;
LOAD h3;
```

Then, you can instantly start using it. Note that the sample data is 500 MB:

```
SELECT
    h3_latlng_to_cell(pickup_latitude, pickup_longitude, 9) AS cell_id,
    h3_cell_to_boundary_wkt(cell_id) AS boundary,
    count() AS cnt
FROM read_parquet('https://blobs.duckdb.org/data/yellow_tripdata_2010-01.parquet')
GROUP BY cell_id
HAVING cnt > 10;
```

On load, the extension's signature is checked, both to ensure platform and versions are compatible, and to verify that the source of the binary is the Community Extensions repository. Extensions are built, signed and distributed for Linux, macOS, Windows, and WebAssembly. This allows extensions to be available to any DuckDB client using version 1.0.0 and upcoming versions.

The h3 extension's documentation is available at https://duckdb.org/community_extensions/extensions/h3.

Developer Experience

From the developer's perspective, the Community Extensions repository performs the steps required for publishing extensions, including building the extensions for all relevant platforms, signing the extension binaries and serving them from the repository.

For the maintainer of h3, the publication process required performing the following steps:

1. Sending a PR with a metadata file `description.yml` contains the description of the extension:

```
extension:
  name: h3
  description: Hierarchical hexagonal indexing for geospatial data
  version: 1.0.0
  language: C++
  build: cmake
  license: Apache-2.0
  maintainers:
    - isaacbrodsky

repo:
```

```
    github: isaacbrodsky/h3-duckdb
    ref: 3c8a5358e42ab8d11e0253c70f7cc7d37781b2ef
```

2. The CI will build and test the extension. The checks performed by the CI are aligned with the `extension-template` repository, so iterations can be done independently.
3. Wait for approval from the DuckDB Community Extensions repository's maintainers and for the build process to complete.

Published Extensions

To show that it's feasible to publish extensions, we reached out to a few developers of key extensions. At the time of the publication of this blog post, the DuckDB Community Extensions repository already contains the following extensions.

Name	Description
crypto	Adds cryptographic hash functions and HMAC.
h3	Implements hierarchical hexagonal indexing for geospatial data.
lindel	Implements linearization/delinearization, Z-Order, Hilbert and Morton curves.
prql	Allows running PRQL commands directly within DuckDB.
scrooge	Supports a set of aggregation functions and data scanners for financial data.
shellfs	Allows shell commands to be used for input and output.

DuckDB Labs and the DuckDB Foundation do not vet the code within community extensions and, therefore, cannot guarantee that DuckDB community extensions are safe to use. The loading of community extensions can be explicitly disabled with the following one-way configuration option:

```
SET allow_community_extensions = false;
```

For more details, see the documentation's Securing DuckDB page.

Summary and Looking Ahead

In this blog post, we introduced the DuckDB Community Extensions repository, which allows easy installation of third-party DuckDB extensions.

We are looking forward to continuously extending this repository. If you have an idea for creating an extension, take a look at the already published extension source codes, which provide good examples of how to package community extensions, and join the `#extensions` channel on our Discord. Once you have an extension, please contribute it via a pull request.

Finally, we would like to thank the early adopters of DuckDB's extension mechanism and Community Extensions repository. Thanks for iterating with us and providing feedback to us.

Memory Management in DuckDB

Publication date: 2024-07-09

Author: Mark Raasveldt

Memory is an important resource when processing large amounts of data. Memory is a fast caching layer that can provide immense speed-ups to query processing. However, memory is finite and expensive, and when working with large data sets there is generally not enough memory available to keep all necessary data structures cached. Managing memory effectively is critical for a high-performance query engine – as memory must be utilized in order to provide that high performance, but we must be careful so that we do not use excessive memory which can cause out-of-memory errors or can cause the ominous OOM killer to zap the process out of existence.

DuckDB is built to effectively utilize available memory while avoiding running out of memory:

- The streaming execution engine allows small chunks of data to flow through the system without requiring entire data sets to be materialized in memory.
- Data from intermediates can be spilled to disk temporarily in order to free up space in memory, allowing computation of complex queries that would otherwise exceed the available memory.
- The buffer manager caches as many pages as possible from any attached databases without exceeding the pre-defined memory limits.

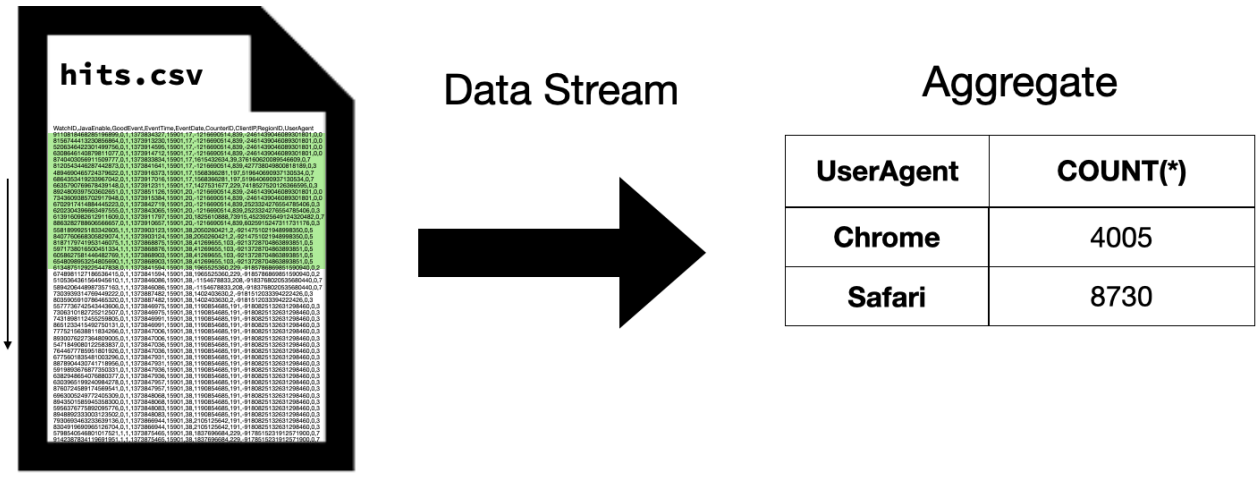
In this blog post we will cover these aspects of memory management within DuckDB – and provide examples of where they are utilized.

Streaming Execution

DuckDB uses a streaming execution engine to process queries. Data sources, such as tables, CSV files or Parquet files, are never fully materialized in memory. Instead, data is read and processed one chunk at a time. For example, consider the execution of the following query:

```
SELECT
    UserAgent,
    count(*)
FROM 'hits.csv'
GROUP BY UserAgent;
```

Instead of reading the entire CSV file at once, DuckDB reads data from the CSV file in pieces, and computes the aggregation incrementally using the data read from those pieces. This happens continuously until the entire CSV file is read, at which point the entire aggregation result is computed.



In the above example we are only showing a single data stream. In practice, DuckDB uses multiple data streams to enable multi-threaded execution – each thread executes its own data stream. The aggregation results of the different threads are combined to compute the final result.

While streaming execution is conceptually simple, it is powerful, and is sufficient to provide larger-than-memory support for many simple use cases. For example, streaming execution enables larger-than-memory support for:

- Computing aggregations where the total number of groups is small
- Reading data from one file and writing to another (e.g., reading from CSV and writing to Parquet)
- Computing a Top-N over the data (where N is small)

Note that nothing needs to be done to enable streaming execution – DuckDB always processes queries in this manner.

Intermediate Spilling

While streaming execution enables larger-than-memory processing for simple queries, there are many cases where streaming execution alone is not sufficient.

In the previous example, streaming execution enabled larger-than-memory processing because the computed aggregate result was very small – as there are very few unique user agents in comparison to the total number of web requests. As a result, the aggregate hash table would always remain small, and never exceed the amount of available memory.

Streaming execution is not sufficient if the intermediates required to process a query are larger than memory. For example, suppose we group by the source IP in the previous example:

```
SELECT
    IPNetworkID,
    count(*)
FROM 'hits.csv'
GROUP BY IPNetworkID;
```

Since there are many more unique source IPs, the hash table we need to maintain is significantly larger. If the size of the aggregate hash table exceeds memory, the streaming execution engine is not sufficient to prevent out-of-memory issues.

Larger-than-memory intermediates can happen in many scenarios, in particular when executing more complex queries. For example, the following scenarios can lead to larger-than-memory intermediates:

- Computing an aggregation with many unique groups
- Computing an exact distinct count of a column with many distinct values

- Joining two tables together that are both larger than memory
- Sorting a larger-than-memory dataset
- Computing a complex window over a larger-than-memory table

DuckDB deals with these scenarios by disk spilling. Larger-than-memory intermediates are (partially) written to disk in the temporary directory when required. While powerful, disk spilling reduces performance – as additional I/O must be performed. For that reason, DuckDB tries to minimize disk spilling. Disk spilling is adaptively used only when the size of the intermediates increases past the memory limit. Even in those scenarios, as much data is kept in memory as possible to maximize performance. The exact way this is done depends on the operators and is detailed in other blog posts (aggregation, sorting).

The `memory_limit` setting controls how much data DuckDB is allowed to keep in memory. By default, this is set to 80% of the physical RAM of your system (e.g., if your system has 16 GB RAM, this defaults to 12.8 GB). The memory limit can be changed using the following command:

```
SET memory_limit = '4GB';
```

The location of the temporary directory can be chosen using the `temp_directory` setting, and is by default the connected database with a `.tmp` suffix (e.g., `database.db.tmp`), or only `.tmp` if connecting to an in-memory database. The maximum size of the temporary directory can be limited using the `max_temp_directory_size` setting, which defaults to 90% of the remaining disk space on the drive where the temporary files are stored. These settings can be adjusted as follows:

```
SET temp_directory = '/tmp/duckdb_swap';
SET max_temp_directory_size = '100GB';
```

If the memory limit is exceeded and disk spilling cannot be used, either because disk spilling is explicitly disabled, the temporary directory size exceeds the provided limit, or a system limitation means that disk spilling cannot be used for a given query – an out-of-memory error is reported and the query is canceled.

Buffer Manager

Another core component of memory management in DuckDB is the buffer manager. The buffer manager is responsible for caching pages from DuckDB's own persistent storage. Conceptually the buffer manager works in a similar fashion to the intermediate spilling. Pages are kept in memory as much as possible, and evicted from memory when space is required for other data structures. The buffer manager abides by the same memory limit as any intermediate data structures. Pages in the buffer manager can be freed up to make space for intermediate data structures, or vice versa.

There are two main differences between the buffer manager and intermediate data structures:

- As the buffer manager caches pages that already exist on disk (in DuckDB's persistent storage) – they do not need to be written to the temporary directory when evicted. Instead, when they are required again, they can be re-read from the attached storage file directly.
- Query intermediates have a natural life-cycle, namely when the query is finished processing the intermediates are no longer required. Pages that are buffer managed from the persistent storage are useful across queries. As such, the pages kept by the buffer manager are kept cached until either the persistent database is closed, or until space must be freed up for other operations.

The performance boost of the buffer manager depends on the speed of the underlying storage medium. When data is stored on a very fast disk, reading data is fast and the speed-up is minimal. When data is stored on a network drive or read over http/S3, reading requires performing network requests, and the speed-up can be very large.

Profiling Memory Usage

DuckDB contains a number of tools that can be used to profile memory usage.

The `duckdb_memory()` function can be used to inspect which components of the system are using memory. Memory used by the buffer manager is labeled as `BASE_TABLE`, while query intermediates are divided into separate groups.

```
FROM duckdb_memory();
```

tag varchar	memory_usage_bytes int64	temporary_storage_bytes int64
BASE_TABLE	168558592	0
HASH_TABLE	0	0
PARQUET_READER	0	0
CSV_READER	0	0
ORDER_BY	0	0
ART_INDEX	0	0
COLUMN_DATA	0	0
METADATA	0	0
OVERFLOW_STRINGS	0	0
IN_MEMORY_TABLE	0	0
ALLOCATOR	0	0
EXTENSION	0	0
12 rows		3 columns

The `duckdb_temporary_files` function can be used to examine the current contents of the temporary directory.

```
FROM duckdb_temporary_files();
```

path varchar	size int64
.tmp/duckdb_temp_storage-0.tmp	967049216

Conclusion

Memory management is critical for a high-performance analytics engine. DuckDB is built to take advantage of any available memory to speed up query processing, while gracefully dealing with larger-than-memory datasets using intermediate spilling. Memory management is still an active area of development and has continuously improved across DuckDB versions. Amongst others, we are working on improving memory management for complex queries that involve multiple operators with larger-than-memory intermediates.

Friendly Lists and Their Buddies, the Lambdas

Publication date: 2024-08-08

Authors: Tania Bogatsch and Maia de Graaf

Introduction

Nested data types, such as lists and structs, are widespread in analytics. Several popular formats, such as Parquet and JSON, support nested types. Traditionally, working with nested types requires normalizing steps before any analysis. Then, to return nested results, systems need to (re-)aggregate their data. Normalization and aggregation are undesirable from both a usability and performance perspective. To streamline the operation on nested data, analytical systems, including DuckDB, provide native functionality on these nested types.

In this blog post, we'll first cover the basics of lists and lambdas. Then, we dive into their technical details. Finally, we'll show some examples from the community. Feel free to skip ahead if you're already familiar with lists and lambdas and are just here for our out-of-the-box examples!

Lists

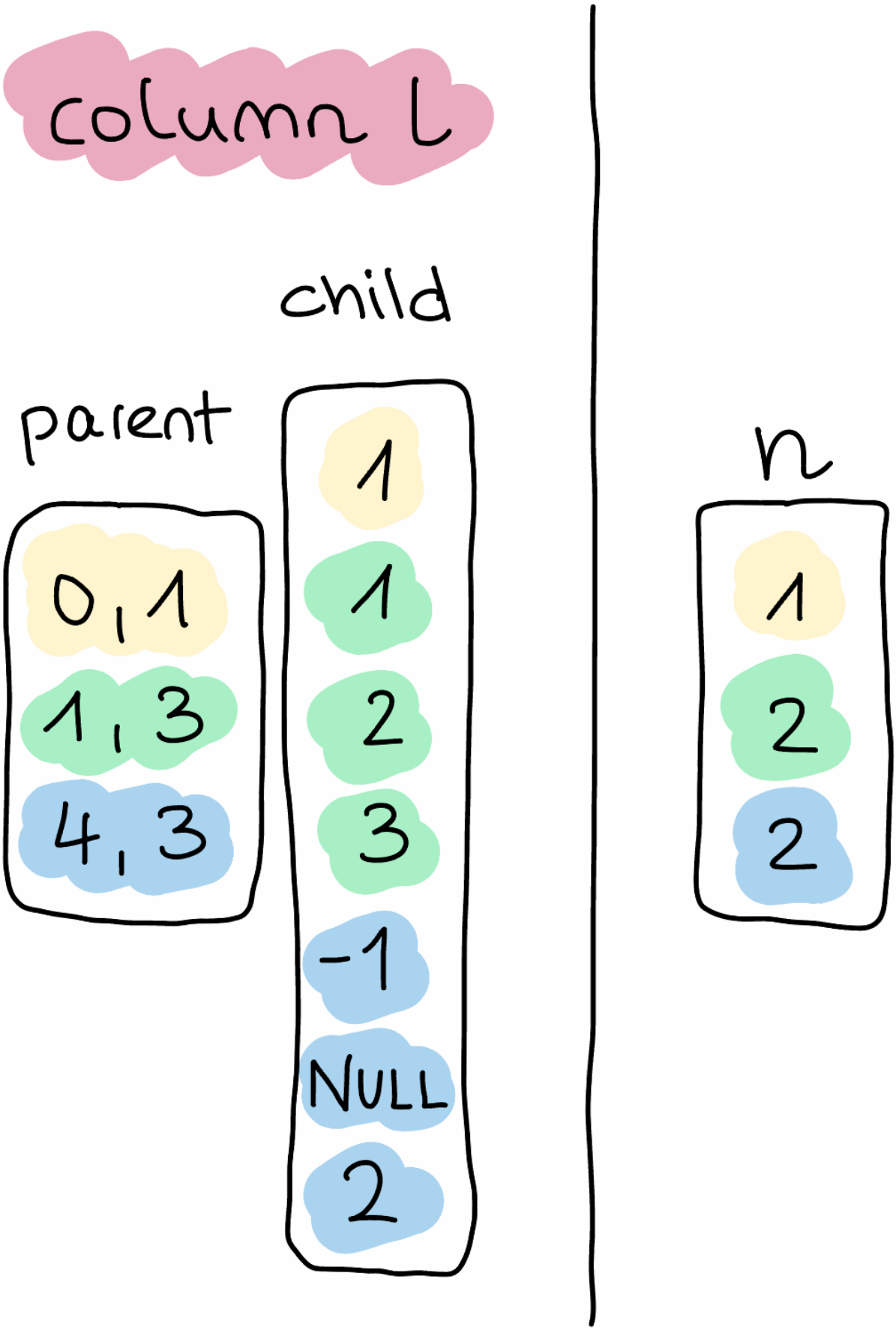
Before jumping into lambdas, let's take a quick detour into DuckDB's `LIST` type. A list contains any number of elements with the same data type. Below is a table containing two columns, `l` and `n`. `l` contains lists of integers, and `n` contains integers.

```
CREATE OR REPLACE TABLE my_lists (l INTEGER[], n INTEGER);
INSERT INTO my_lists VALUES ([1], 1), ([1, 2, 3], 2), ([-1, NULL, 2], 2);
FROM my_lists;
```

l int32[]	n int32
[1]	1
[1, 2, 3]	2
[-1, NULL, 2]	2

Internally, all data moves through DuckDB's execution engine in `Vectors`. For more details on `Vectors` and vectorized execution, please refer to the documentation and respective research papers (1 and 2). In this case, we get two vectors, as depicted below. This representation is mostly similar to Arrow's physical list representation.

When examined closely, we can observe that the nested child vector of `l` looks suspiciously similar to the vector `n`. These nested vector representations enable our execution engine to reuse existing components on nested types. We'll elaborate more on why this is relevant later.


column L
child
parent
1
0, 1
1
1, 3
2
4, 3
3
-1
NULL
2
n
1
2
2

Lambdas

A **lambda function** is an anonymous function, i.e., a function without a name. In DuckDB, a lambda function's syntax is `(param1, param2, ...) -> expression`. The parameters can have any name, and the `expression` can be any SQL expression.

Currently, DuckDB has three scalar functions for working with lambdas: `list_transform`, `list_filter`, and `list_reduce`, along with their aliases. Each accepts a `LIST` as its first argument and a lambda function as its second argument.

Lambdas were the guest star in our SQL Gymnastics: Bending SQL into Flexible New Shapes blog post. This time, we want to put them in the spotlight.

Zooming In: List Transformations

To return to our previous example, let's say we want to add `n` to each element of the corresponding list `l`.

Pure Relational Solution

Using pure relational operators, i.e., avoiding list-native functions, we would need to perform the following steps:

1. Unnest the lists while keeping the connection to their respective rows. We can achieve this by inventing a temporary unique identifier, such as a `rowid` or a `UUID`.
2. Transform each element by adding `n`.
3. Using our temporary identifier `rowid`, we can reaggregate the transformed elements by grouping them into lists.

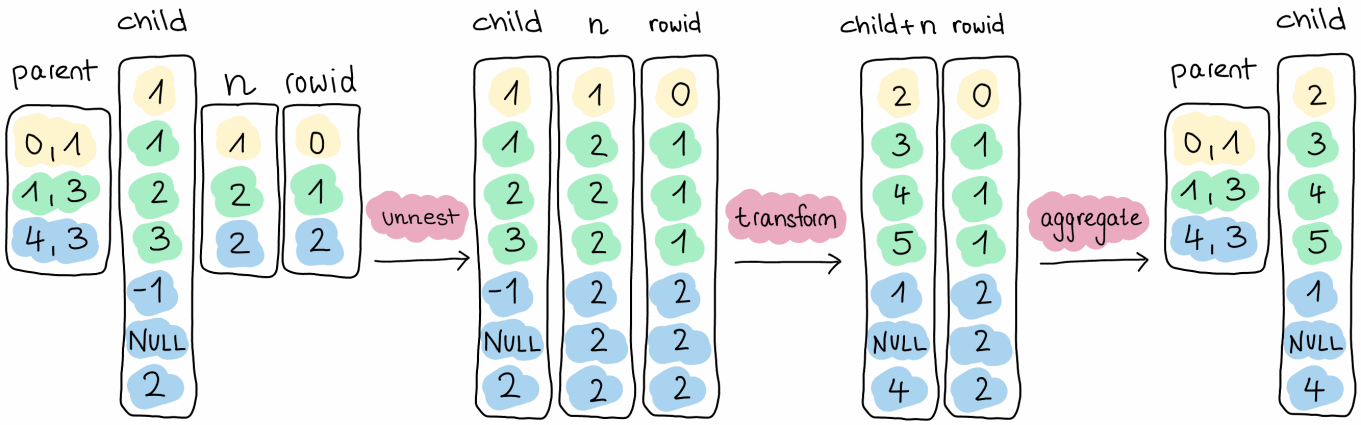
In SQL, it would look like this:

```
WITH flattened_tbl AS (
    SELECT unnest(l) AS elements, n, rowid
    FROM my_lists
)
SELECT array_agg(elements + n) AS result
FROM flattened_tbl
GROUP BY rowid
ORDER BY rowid;
```

```
┌──────────────┐
│    result    │
│   int32[]    │
├──────────────┤
│ [2]          │
│ [3, 4, 5]    │
│ [1, NULL, 4] │
└──────────────┘
```

While the above example is reasonably readable, more complex transformations can become lengthy queries, which are difficult to compose and maintain. More importantly, this query adds an `unnest` operation and an aggregation (`array_agg`) with a `GROUP BY`. Adding a `GROUP BY` can be costly, especially for large datasets.

We have to dive into the technical implications to fully understand why the above query yields suboptimal performance. Internally, the query execution performs the steps depicted in the diagram below. We can directly emit the child vector for the `unnest` operation, i.e., without copying any data. For the correlated columns `rowid` and `n`, we use selection vectors, which again prevents the copying of data. This way, we can fire our expression execution on the child vector, another nested vector, and the expanded vector `n`.



The heavy-hitting operation is the last one, reaggregating the transformed elements into their respective lists. As we don't propagate the parent vector, we have no information about the resulting element's correlation to the initial lists. Recreating these lists requires a full copy of the data and partitioning, which impacts performance even with DuckDB's high-performance aggregation operator.

As a consequence, the normalized approach is both cumbersome to write and it is inefficient as it produces a significant (and unnecessary) overhead despite the relative simplicity of the query. This is yet another example of how shaping nested data into relational forms or forcing it through rectangles can have a significant negative performance impact.

Native List Functions

With support for native list functions, DuckDB mitigates these drawbacks by operating directly on the `LIST` data structure. Since, as we've seen, lists are essentially nested columns, we can reshape these functions into concepts already understood by our execution engine and leverage their full potential.

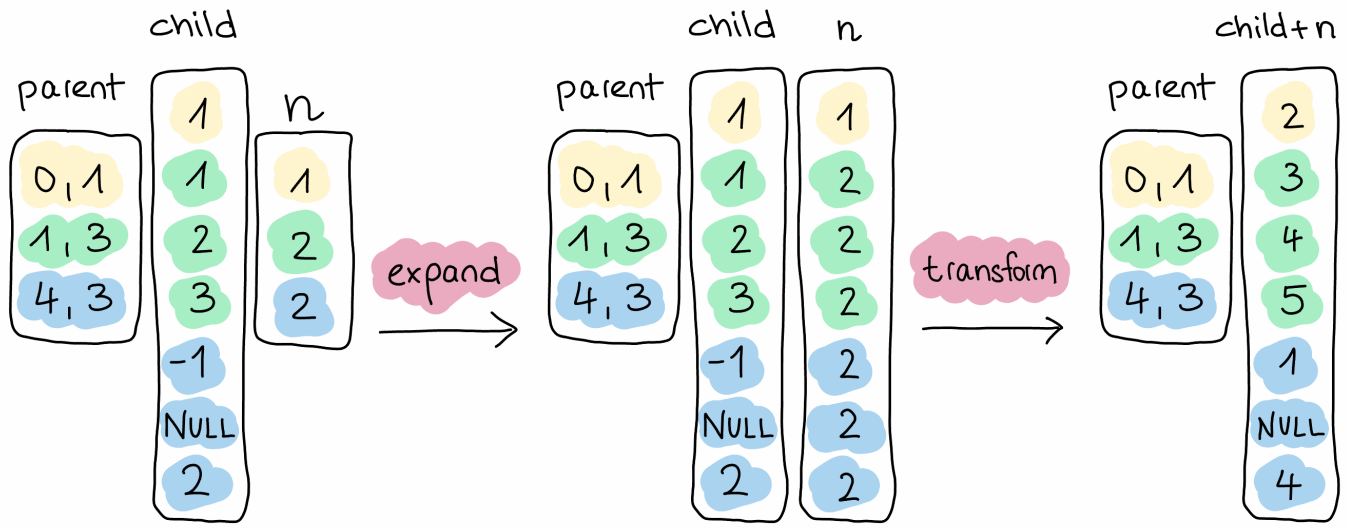
In the case of transformations, the corresponding list-native function is `list_transform`. Here is the rewritten query:

```
SELECT list_transform(l, x -> x + n) AS result
FROM my_lists;
```

Alternatively, with Python's list comprehension syntax:

```
SELECT [x + n FOR x IN l] AS result
FROM my_lists;
```

Internally, this query expands all related vectors, which is just n in this case. Just like before, we employ selection vectors to avoid any data copies. Then, we use the lambda function `x -> x + n` to fire our expression execution on the child vector and the expanded vector n. As this is a list-native function, we're aware of the existence of a parent vector and keep it alive. So, once we get the result from the transformation, we can completely omit the reaggregation step.



To see the efficiency of `list_transform` in action, we executed a simple benchmark. Firstly, we added 1M rows to our table `my_lists`, each containing five elements.

```
INSERT INTO my_lists
    SELECT [r, r % 10, r + 5, r + 11, r % 2], r
    FROM range(1_000_000) AS tbl(r);
```

Then, we ran both our normalized and list-native queries on this data. Both queries were run in the CLI with DuckDB v1.0.0 on a MacBook Pro 2021 with a M1 Max chip.

Normalized	Native
0.522 s	0.027 s

As we can see, the native query is more than 10× faster. Amazing! If we look at the execution plan using `EXPLAIN ANALYZE` (not shown in this blog post), we can see that DuckDB spends most of its time in the `HASH_GROUP_BY` and `UNNEST` operators. In comparison, these operators no longer exist in the list-native query plan.

Lists and Lambdas in the Community

To better present what's possible by combining our `LIST` type and lambda functions, we've scoured the community Discord and GitHub, as well as some far corners of the internet, for exciting use cases.

`list_transform`

As established earlier, `list_transform` applies a lambda function to each element of the input list and returns a new list with the transformed elements. Here, one of our users implemented a `list_shuffle` function by nesting different `LIST` native functions.

```
CREATE OR REPLACE MACRO list_shuffle(l) AS (
    list_select(l, list_grade_up([random() FOR _ IN l]))
);
```

Another user investigated querying remote Parquet files using DuckDB. In their query, they first use `list_transform` to generate a list of URLs for Parquet files. This is followed by the `read_parquet` function, which reads the Parquet files and calculates the total size of the data. The query looks like this:

```
SELECT
    sum(size) AS size
FROM read_parquet(
    ['https://huggingface.co/datasets/vivym/midjourney-messages/resolve/main/data/' ||
        format('{:06d}', n) || '.parquet'
        FOR n IN generate_series(0, 55)
    ]
);
```

`list_filter`

The `list_filter` function filters all elements of the input list for which the lambda function returns `true`.

Here is an example using `list_filter` from a discussion on our Discord where the user wanted to remove the element at index `idx` from each list.

```
CREATE OR REPLACE MACRO remove_idx(l, idx) AS (
    list_filter(l, (_, i) -> i != idx)
);
```

So far, we've primarily focused on showcasing our lambda function support in this blog post. Yet, there are often many possible paths with SQL and its rich dialects. We couldn't help but show how we can achieve the same functionality with some of our other native list functions. In this case, we used `list_slice` and `list_concat`.

```
CREATE OR REPLACE MACRO remove_idx(l, idx) AS (
    l[:idx - 1] || l[idx + 1:]
);
```

`list_reduce`

Most recently, we've added `list_reduce`, which applies a lambda function to an accumulator value. The accumulator is the result of the previous lambda function and is also what the function ultimately returns.

We took the following example from a discussion on GitHub. The user wanted to use a lambda to validate BSN numbers, the Dutch equivalent of social security numbers. A BSN must be 8 or 9 digits, but to limit our scope we'll just focus on BSNs that are 9 digits long. After multiplying each digit by its index, from 9 down to 2, and the last digit by -1, the sum must be divisible by 11 to be valid.

Setup

For our example, we assume that input BSNs are of type `INTEGER[]`.

```
CREATE OR REPLACE TABLE bsn_tbl AS
    FROM VALUES
        ([2, 4, 6, 7, 4, 7, 5, 9, 6]),
        ([1, 2, 3, 4, 5, 6, 7, 8, 9]),
        ([7, 6, 7, 4, 4, 5, 2, 1, 1]),
        ([8, 7, 9, 0, 2, 3, 4, 1, 7]),
        ([1, 2, 3, 4, 5, 6, 7, 8, 9, 0])
        tbl(bsn);
```

Solution

When this problem was initially proposed, DuckDB didn't have support for `list_reduce`. Instead, the user came up with the following:

```
CREATE OR REPLACE MACRO valid_bsn(bsn) AS (
    list_sum(
        [array_extract(bsn, x)::INTEGER * (IF (x = 9, -1, 10 - x))
        FOR x IN range(1, 10, 1)]
    ) % 11 = 0
);
```

With `list_reduce`, we can rewrite the query as follows. We also added a check validating that the length is always nine digits.

```
CREATE OR REPLACE MACRO valid_bsn(bsn) AS (
    list_reduce(list_reverse(bsn),
        (x, y, i) -> IF (i = 1, -x, x) + y * (i + 1)) % 11 = 0
    AND len(bsn) = 9
);
```

Using our macro with the example table we get the following result:

```
SELECT bsn, valid_bsn(bsn) AS valid
FROM bsn_tbl;
```

bsn int32[]	valid boolean
[2, 4, 6, 7, 4, 7, 5, 9, 6]	true
[1, 2, 3, 4, 5, 6, 7, 8, 9]	false
[7, 6, 7, 4, 4, 5, 2, 1, 1]	true
[8, 7, 9, 0, 2, 3, 4, 1, 7]	true
[1, 2, 3, 4, 5, 6, 7, 8, 9, 0]	false

Conclusion

Native nested type support is critical for analytical systems. As such, DuckDB offers native nested type support and many functions to work with these types directly. These functions make working with nested types easier and substantially faster. In this blog post, we looked at the technical details of working with nested types by diving into our `list_transform` function. Additionally, we highlighted various use cases that we came across in our community.

DuckDB Tricks – Part 1

Publication date: 2024-08-19

Author: Gabor Szarnyas

TL;DR: We use a simple example data set to present a few tricks that are useful when using DuckDB.

In this blog post, we present five simple DuckDB operations that we found particularly useful for interactive use cases. The operations are summarized in the following table:

Operation	Snippet
Pretty-printing floats	`SELECT (10 / 9)::DECIMAL(15, 3)`{:.language-sql .highlight}
Copying the schema	`CREATE TABLE tbl AS FROM example LIMIT 0`{:.language-sql .highlight}
Shuffling data	`FROM example ORDER BY hash(rowid + 42)`{:.language-sql .highlight}
Specifying types when reading CSVs	`FROM read_csv('example.csv', types = {'x': 'DECIMAL(15, 3)'})`{:.language-sql .highlight}
Updating CSV files in-place	`COPY (SELECT s FROM 'example.csv') TO 'example.csv'`{:.language-sql .highlight}

Creating the Example Data Set

We start by creating a data set that we'll use in the rest of the blog post. To this end, we define a table, populate it with some data and export it to a CSV file.

```
CREATE TABLE example (s STRING, x DOUBLE);
INSERT INTO example VALUES ('foo', 10/9), ('bar', 50/7), ('qux', 9/4);
COPY example TO 'example.csv';
```

Wait a bit, that's way too verbose! DuckDB's syntax has several SQL shorthands including the "friendly SQL" clauses. Here, we combine the VALUES clause with the FROM-first syntax, which makes the SELECT clause optional. With these, we can compress the data creation script to ~60% of its original size. The new formulation omits the schema definition and creates the CSV with a single command:

```
COPY (FROM VALUES ('foo', 10/9), ('bar', 50/7), ('qux', 9/4) t(s, x))
TO 'example.csv';
```

Regardless of which script we run, the resulting CSV file will look like this:

```
s,x
foo,1.1111111111111112
bar,7.142857142857143
qux,2.25
```

Let's continue with the code snippets and their explanations.

Pretty-Printing Floating-Point Numbers

When printing a floating-point number to the output, the fractional parts can be difficult to read and compare. For example, the following query returns three numbers between 1 and 8 but their printed widths are very different due to their fractional parts.

```
SELECT x
FROM 'example.csv';
```

x
double
1.1111111111111112
7.142857142857143
2.25

By casting a column to a `DECIMAL` with a fixed number of digits after the decimal point, we can pretty-print it as follows:

```
SELECT x::DECIMAL(15, 3) AS x
FROM 'example.csv';
```

x
decimal(15,3)
1.111
7.143
2.250

A typical alternative solution is to use the `printf` or `format` functions, e.g.:

```
SELECT printf('%.3f', x)
FROM 'example.csv';
```

However, these approaches require us to specify a formatting string that's easy to forget. What's worse, the statement above returns string values, which makes subsequent operations (e.g., sorting) more difficult. Therefore, unless keeping the full precision of the floating-point numbers is a concern, casting to `DECIMAL` values should be the preferred solution for most use cases.

Copying the Schema of a Table

To copy the schema from a table without copying its data, we can use `LIMIT 0`.

```
CREATE TABLE example AS
    FROM 'example.csv';
CREATE TABLE tbl AS
    FROM example
    LIMIT 0;
```

This will result in an empty table with the same schema as the source table:

```
DESCRIBE tbl;
```

column_name varchar	column_type varchar	null varchar	key varchar	default varchar	extra varchar
s	VARCHAR	YES			
x	DOUBLE	YES			

Alternatively, in the CLI client, we can run the `.schema` dot command:

```
.schema
```

This will return the schema of the table.

```
CREATE TABLE example (s VARCHAR, x DOUBLE);
```

After editing the table's name (e.g., `example` to `tbl`), this query can be used to create a new table with the same schema.

Shuffling Data

Sometimes, we need to introduce some entropy into the ordering of the data by shuffling it. To shuffle *non-deterministically*, we can simply sort on a random value provided the `random()` function:

```
FROM 'example.csv' ORDER BY random();
```

Shuffling *deterministically* is a bit more tricky. To achieve this, we can order on the hash, of the `rowid` pseudocolumn. Note that this column is only available in physical tables, so we first have to load the CSV in a table, then perform the shuffle operation as follows:

```
CREATE OR REPLACE TABLE example AS FROM 'example.csv';
FROM example ORDER BY hash(rowid + 42);
```

The result of this shuffle operation is deterministic – if we run the script repeatedly, it will always return the following table:

s varchar	x double
bar	7.142857142857143
qux	2.25
foo	1.1111111111111112

Note that the `+ 42` is only necessary to nudge the first row from its position – as `hash(0)` returns `0`, the smallest possible value, using it for ordering leaves the first row in its place.

Specifying Types in the CSV Loader

DuckDB's CSV loader auto-detects types from a short list of `BOOLEAN`, `BIGINT`, `DOUBLE`, `TIME`, `DATE`, `TIMESTAMP` and `VARCHAR`. In some cases, it's desirable to override the detected type of a given column with a type outside of this list. For example, we may want to treat column `x` as a `DECIMAL` value from the get-go. We can do this on a per-column basis with the `types` argument of the `read_csv` function:

```
CREATE OR REPLACE TABLE example AS
    FROM read_csv('example.csv', types = {'x': 'DECIMAL(15, 3)'});
```

Then, we can simply query the table to see the result:

```
FROM example;
```

s varchar	x decimal(15,3)
foo	1.111
bar	7.143
qux	2.250

Updating CSV Files In-Place

In DuckDB, it is possible to read, process and write CSV files in-place. For example, to project the column `s` into the same file, we can simply run:

```
COPY (SELECT s FROM 'example.csv') TO 'example.csv';
```

The resulting `example.csv` file will have the following content:

```
s
foo
bar
qux
```

Note that this trick is not possible in Unix shells without a workaround. One might be tempted to run the following command on the `example.csv` file and expect the same result:

```
cut -d, -f1 example.csv > example.csv
```

However, due to the intricacies of Unix pipelines, executing this command leaves us with an empty `example.csv` file. The solution is to use different file names, then perform a rename operation:

```
cut -d, -f1 example.csv > tmp.csv && mv tmp.csv example.csv
```

Closing Thoughts

That's it for today. The tricks shown in this post are available on duckdbsnippets.com. If you have a trick that would like to share, please submit it there, or send it to us via social media or Discord. Happy hacking!

Announcing DuckDB 1.1.0

Publication date: 2024-09-09

Author: The DuckDB team

TL;DR: The DuckDB team is happy to announce that today we're releasing DuckDB version 1.1.0, codenamed “Eatoni”.

To install the new version, please visit the installation guide. For the release notes, see the release page.

> Some packages (R, Java) take a few extra days to release due to the reviews required in the release pipelines.

We are proud to release DuckDB 1.1.0, our first release since we released version 1.0.0 three months ago. This release is codenamed “Eatoni” after the Eaton's pintail (Anas eatoni), a dabbling duck that occurs only on two very remote island groups in the southern Indian Ocean.

What's New in 1.1.0

There have been far too many changes to discuss them each in detail, but we would like to highlight several particularly exciting features! Below is a summary of those new features with examples.

Breaking SQL Changes

IEEE-754 semantics for division by zero. The IEEE-754 floating point standard states that division by zero returns `inf`. Previously, DuckDB would return `NULL` when dividing by zero, also for floating point division. Starting with this release, DuckDB will return `inf` instead.

```
SELECT 1 / 0 AS division_by_zero;
```

division_by_zero
double
inf

The `ieee_floating_point_ops` can be set to `false` to revert this behavior:

```
SET ieee_floating_point_ops = false;
SELECT 1 / 0 AS division_by_zero;
```

division_by_zero
double
NULL

Error when scalar subquery returns multiple values. Scalar subqueries can only return a single value per input row. Previously, DuckDB would match SQLite's behavior and select an arbitrary row to return when multiple rows were returned. In practice this behavior often led to confusion. Starting with this release, an error is returned instead, matching the behavior of Postgres. The subquery can be wrapped with ARRAY to collect all of the results of the subquery in a list.

```
SELECT (SELECT unnest(range(10)));
```

```
Invalid Input Error: More than one row returned by a subquery used as
an expression - scalar subqueries can only return a single row.
```

```sql
SELECT ARRAY(SELECT unnest(range(10))) AS subquery_result;
```

```
┌──────────────────────────────────┐
│        subquery_result           │
│            int64[]               │
├──────────────────────────────────┤
│ [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]   │
└──────────────────────────────────┘
```

The `scalar_subquery_error_on_multiple_rows` setting can be set to `false` to revert this behavior.

```sql
SET scalar_subquery_error_on_multiple_rows = false;
SELECT (SELECT unnest(range(10))) AS result;
```

```
┌──────────┐
│ result   │
│ int64    │
├──────────┤
│       0  │
└──────────┘
```

Community Extensions

Recently we introduced Community Extensions. Community extensions allow anyone to build extensions for DuckDB, that are then built and distributed by us. The list of community extensions has been growing since then.

In this release, we have been working towards making community extensions easier to build and produce. This release includes a new method of registering extensions using the C API in addition to a lot of extensions to the C API allowing scalar functions, aggregate functions and custom types to be defined. These changes will enable building extensions against a stable API, that are smaller in size, that will work across different DuckDB versions. In addition, these changes will enable building extensions in other programming languages in the future.

Friendly SQL

Histogram. This version introduces the `histogram` function that can be used to compute histograms over columns of a dataset. The histogram function works for columns of any type, and allows for various different binning strategies and a custom amount of bins.

```sql
FROM histogram(
    'https://blobs.duckdb.org/data/ontime.parquet',
    UniqueCarrier,
    bin_count := 5
);
```

```
┌──────────────────┬──────────┬──────────────────────────────────────────────────────────────────────────────┐
│      bin         │ count    │                                   bar                                        │
│    varchar       │ uint64   │                                 varchar                                      │
├──────────────────┼──────────┼──────────────────────────────────────────────────────────────────────────────┤
│ AA               │ 677215   │ ████████████████████████████████████████████████████████████████████████████
│
│ DL               │ 696931   │ ████████████████████████████████████████████████████████████████████████████
│
│ OO              │ 521956   │ ████████████████████████████████████████████████████████████████████████████
│ UA              │ 435757   │ ████████████████████████████████████████████████████████████████████████████
│ WN               │ 999114   │
████████████████████████████████████████████████████████████████████████████████████████████████████████████ │
```

```
│ (other values) │ 945484 │ ████████████████████████████████████████████████████████████████████████████████ │
└────────────────┴────────┴──────────────────────────────────────────────────────────────────────────────────┘
```

SQL variables. This release introduces support for variables that can be defined in SQL. Variables can hold a single value of any type – including nested types like lists or structs. Variables can be set as literals, or from scalar subqueries.

The value stored within variables can be read using `getvariable`. When used in a query, `getvariable` is treated as a literal during query planning and optimization. This allows variables to be used in places where we normally cannot read values from within tables, for example, when specifying which CSV files to read:

```
SET VARIABLE list_of_files = (SELECT LIST(file) FROM csv_files);
SELECT * FROM read_csv(getvariable('list_of_files'), filename := true);
```

a int64	filename varchar
42	test.csv
84	test2.csv

Unpacked Columns

The COLUMNS expression allows users to write dynamic SQL over a set of columns without needing to explicitly list the columns in the SQL string. Instead, the columns can be selected through either a regex or computed with a lambda function.

This release expands this capability by allowing the COLUMNS expression to be *unpacked* into a function. This is especially useful when combined with nested functions like `struct_pack` or `list_value`.

```
CREATE TABLE many_measurements (
    id INTEGER, m1 INTEGER, m2 INTEGER, m3 INTEGER
);
INSERT INTO many_measurements VALUES (1, 10, 100, 20);

SELECT id, struct_pack(*COLUMNS('m\d')) AS measurements
FROM many_measurements;
```

id int32	measurements struct(m1 integer, m2 integer, m3 integer)
1	{'m1': 10, 'm2': 100, 'm3': 20}

`query` and `query_table` Functions

The `query` and `query_table` functions take a string literal, and convert it into a `SELECT` subquery or a table reference. Note that these functions can only take literal strings. As such, they are not as powerful (or dangerous) as a generic `eval`.

These functions are conceptually simple, but enable powerful and more dynamic SQL. For example, they allow passing in a table name as a prepared statement parameter:

```
CREATE TABLE my_table (i INTEGER);
INSERT INTO my_table VALUES (42);

PREPARE select_from_table AS SELECT * FROM query_table($1);
EXECUTE select_from_table('my_table');
```

i
int32
42

When combined with the COLUMNS expression, we can write very generic SQL-only macros. For example, below is a custom version of SUMMARIZE that computes the `min` and `max` of every column in a table:

```
CREATE OR REPLACE MACRO my_summarize(table_name) AS TABLE
SELECT
    unnest([*COLUMNS('alias_.*')]) AS column_name,
    unnest([*COLUMNS('min_.*')]) AS min_value,
    unnest([*COLUMNS('max_.*')]) AS max_value
FROM (
    SELECT
        any_value(alias(COLUMNS(*))) AS "alias_\0",
        min(COLUMNS(*))::VARCHAR AS "min_\0",
        max(COLUMNS(*))::VARCHAR AS "max_\0"
    FROM query_table(table_name::VARCHAR)
);

SELECT *
FROM my_summarize('https://blobs.duckdb.org/data/ontime.parquet')
LIMIT 3;
```

column_name	min_value	max_value
varchar	varchar	varchar
year	2017	2017
quarter	1	3
month	1	9

Performance

Dynamic Filter Pushdown from Joins

This release adds a *very cool* optimization for joins: DuckDB now automatically creates filters for the larger table in the join during execution. Say we are joining two tables A and B. A has 100 rows, and B has one million rows. We are joining on a shared key `i`. If there were any filter on `i`, DuckDB would already push that filter into the scan, greatly reducing the cost to complete the query. But we are now filtering on another column from A, namely `j`:

```
CREATE TABLE A AS
    SELECT range AS i, range AS j
    FROM range(100);

CREATE TABLE B AS
    SELECT t1.range AS i
    FROM range(100) t1, range(10_000) t2;

SELECT count(*)
FROM A
JOIN B
USING (i) WHERE j > 90;
```

DuckDB will execute this join by building a hash table on the smaller table A, and then probe said hash table with the contents of B. DuckDB will now observe the values of `i` during construction of the hash table on A. It will then create a min-max range filter of those values of `i` and then *automatically* apply that filter to the values of `i` in B! That way, we early remove (in this case) 90% of data from the large table before even looking at the hash table. In this example, this leads to a roughly 10× improvement in query performance. The optimization can also be observed in the output of `EXPLAIN ANALYZE`.

Automatic CTE Materialization

Common Table Expressions (CTE) are a convenient way to break up complex queries into manageable pieces without endless nesting of subqueries. Here is a small example for a CTE:

```
WITH my_cte AS (SELECT range AS i FROM range(10))
SELECT i FROM my_cte WHERE i > 5;
```

Sometimes, the same CTE is referenced multiple times in the same query. Previously, the CTE would be "copied" wherever it appeared. This creates a potential performance problem: if computing the CTE is computationally expensive, it would be better to cache ("materialize") its results instead of computing the result multiple times in different places within the same query. However, different filter conditions might apply for different instantiations of the CTE, which could drastically reduce their computation cost. A classical no-win scenario in databases. It was already possible to explicitly mark a CTE as materialized using the `MATERIALIZED` keyword, but that required manual intervention.

This release adds a feature where DuckDB automatically decides whether a CTE result should be materialized or not using a heuristic. The heuristic currently is that if the CTE performs aggregation and is queried more than once, it should be materialized. We plan to expand that heuristic in the future.

Parallel Streaming Queries

DuckDB has two different methods for fetching results: *materialized* results and *streaming* results. Materialized results fetch all of the data that is present in a result at once, and return it. Streaming results instead allow iterating over the data in incremental steps. Streaming results are critical when working with large result sets as they do not require the entire result set to fit in memory. However, in previous releases, the final streaming phase was limited to a single thread.

Parallelism is critical for obtaining good query performance on modern hardware, and this release adds support for parallel streaming of query results. The system will use all available threads to fill up a query result buffer of a limited size (a few megabytes). When data is consumed from the result buffer, the threads will restart and start filling up the buffer again. The size of the buffer can be configured through the `streaming_buffer_size` parameter.

Below is a small benchmark using `ontime.parquet` to illustrate the performance benefits that can be obtained using the Python streaming result interface:

```
import duckdb
duckdb.sql("SELECT * FROM 'ontime.parquet' WHERE flightnum = 6805;").fetchone()
```

v1.0	v1.1
1.17 s	0.12 s

Parallel `union_by_name`

The `union_by_name` parameter allows combination of – for example – CSV files that have the same columns in them but not in the same order. This release adds support for parallelism when using `union_by_name`. This greatly improves reading performance when using the union by name feature on multiple files.

Nested ART Rework (Foreign Key Load Speed-Up)

We have greatly improved index insertion and deletion performance for foreign keys. Normally, we directly inline row identifiers into the tree structure. However, this is impossible for indexes that contain a lot of duplicates, as is the case with foreign keys. Instead, we now actually create another index entry for each key that is itself another "recursive" index tree in its own right. That way, we can achieve good insertion and deletion performance inside index entries. The performance results of this change are drastic, consider the following example where a has 100 rows and b has one million rows that all reference a:

```
CREATE TABLE a (i INTEGER, PRIMARY KEY (i));
CREATE TABLE b (i INTEGER, FOREIGN KEY (i) REFERENCES a(i));

INSERT INTO a FROM range(100);
INSERT INTO b SELECT a.range FROM range(100) a, range(10_000) b;
```

On the previous version, this would take ca. 10 seconds on a MacBook to complete. It now takes 0.2 seconds thanks to the new index structure, a ca. 50× improvement!

Window Function Improvements

Window functions see a lot of use in DuckDB, which is why we are continuously improving performance of executing Window functions over large datasets.

The `DISTINCT` and `FILTER` window function modifiers can now be executed in streaming mode. Streaming mode means that the input data for the operator does not need to be completely collected and buffered before the operator can execute. For large intermediate results, this can have a very large performance impact. For example, the following query will now use the streaming window operator:

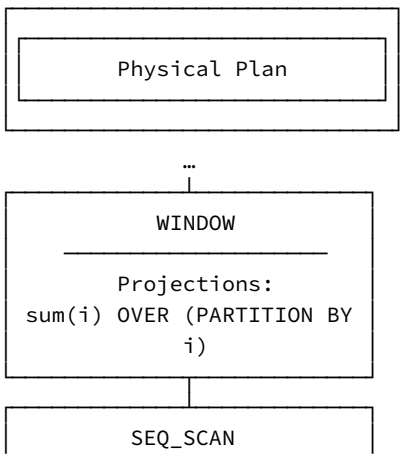
```
SELECT
    sum(DISTINCT i)
        FILTER (i % 3 = 0)
        OVER (ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
FROM range(10) tbl(i);
```

We have also implemented streaming mode for positive `lead` offsets.

We can now push filters on columns through window functions that are partitioned by the same column. For example, consider the following scenario:

```
CREATE TABLE tbl2 AS SELECT range i FROM range(10);
SELECT i
FROM (SELECT i, SUM(i) OVER (PARTITION BY i) FROM tbl)
WHERE i > 5;
```

Previously, the filter on `i` could not be pushed into the scan on `tbl`. But we now recognize that pushing this filter "through" the window is safe and the optimizer will do so. This can be verified through `EXPLAIN`:



```
│     ─────────────────────    │
│             tbl              │
│                              │
│        Projections: i        │
│                              │
│           Filters:           │
│    i>5 AND i IS NOT NULL     │
│                              │
│           ~2 Rows            │
└──────────────────────────────┘
```

The blocking (non-streaming) version of the window operator now processes input data in parallel. This greatly reduces the footprint of the window operator.

See also Richard's talk on the topic at DuckCon #5 in Seattle a few weeks ago.

Spatial Features

GeoParquet

GeoParquet is an extension format of the ubiquitous Parquet format that standardizes how to encode vector geometries and their metadata in Parquet files. This can be used to store geographic data sets in Parquet files efficiently. When the `spatial` extension is installed and loaded, reading from a GeoParquet file through DuckDB's normal Parquet reader will now automatically convert geometry columns to the `GEOMETRY` type, for example:

```sql
INSTALL spatial;
LOAD spatial;

FROM 'https://blobs.duckdb.org/data/geoparquet-example.parquet'
SELECT GEOMETRY g
LIMIT 10;
```

```
┌──────────────────────────────────────────────────────────────────────────────────────────────────────
│                                                 g                                                    │
│                                              geometry                                                │
├──────────────────────────────────────────────────────────────────────────────────────────────────────
│ MULTIPOLYGON (((180 -16.067132663642447, 180 -16.555216566639196, 179.36414266196414
-16.801354076946883, 17…  │
│ POLYGON ((33.90371119710453 -0.95, 34.07261999999997 -1.059819999999945, 37.69868999999994
-3.09698999999994…  │
│ POLYGON ((-8.665589565454809 27.656425889592356, -8.665124477564191 27.589479071558227,
-8.684399786809053 2…  │
│ MULTIPOLYGON (((-122.84000000000003 49.000000000000114, -122.97421000000001 49.00253777777778,
-124.91024 49…  │
│ MULTIPOLYGON (((-122.84000000000003 49.000000000000114, -120 49.000000000000114, -117.03121 49,
-116.04818 4…  │
└──────────────────────────────────────────────────────────────────────────────────────────────────────
```

R-Tree

The spatial extension accompanying this release also implements initial support for creating "R-Tree" spatial indexes. An R-Tree index stores the approximate bounding boxes of each geometry in a column into an auxiliary hierarchical tree-like data structure where every "node" contains a bounding box covering all of its child nodes. This makes it really fast to check what geometries intersect a specific region of interest as you can quickly prune out a lot of candidates by recursively moving down the tree.

Support for spatial indexes has been a long-requested feature on the spatial extension roadmap, and now that we have one, a ton of new use cases and directions for further development are opening up. However, as of now they are only used to accelerate simple queries that

select from a table with a filter using one out of a hardcoded set of spatial predicate functions applied on an indexed geometry column and a constant geometry. This makes R-Tree indexes useful when you have a very large table of geometries that you repeatedly query, but you don't want to perform a full table scan when you're only interested in the rows whose geometries intersect or fit within a certain region anyway. Here is an example where we can see that the `RTREE_INDEX_SCAN` operator is used:

```
INSTALL spatial;
LOAD spatial;

-- Create a table with 10_000_000 random points
CREATE TABLE t1 AS SELECT point::GEOMETRY AS geom
FROM st_generatepoints(
        {min_x: 0, min_y: 0, max_x: 10_000, max_y: 10_000}::BOX_2D,
        10_000_000,
        1337
    );

-- Create an index on the table
CREATE INDEX my_idx ON t1 USING RTREE (geom);

-- Perform a query with a "spatial predicate" on the indexed geometry
-- column. Note how the second argument in this case,
-- the ST_MakeEnvelope call is a "constant"
SELECT count(*)
FROM t1
WHERE ST_Within(geom, ST_MakeEnvelope(450, 450, 650, 650));
```

```
3986
```

R-Tree indexes mostly share the same feature-set as DuckDB's built-in ART index. They are buffer-managed, persistent, lazily-loaded from disk and support inserts, updates and deletes to the base table. Although they can not be used to enforce constraints.

Final Thoughts

These were a few highlights – but there are many more features and improvements in this release. The full release notes can be found on GitHub.

We would like to thank again our amazing community for using DuckDB, building cool projects on DuckDB and improving DuckDB by providing us feedback. Your contributions truly mean a lot!

Changing Data with Confidence and ACID

Publication date: 2024-09-25

Authors: Hannes Mühleisen and Mark Raasveldt

TL;DR: Transactions are key features in database management systems and are also beneficial for data analysis workloads. DuckDB supports fully ACID transactions, confirmed by the TPC-H benchmark's test suite.

The great quote “Everything changes and nothing stays the same” from Heraclitus, according to Socrates, according to Plato is not very controversial: change is as old as the universe. Yet somehow, when dealing with data, we often consider change as merely an afterthought.

Static datasets are split-second snapshots of whatever the world looked like at one moment. But very quickly, the world moves on, and the dataset needs to catch up to remain useful. In the world of tables, new rows can be added, old rows may be deleted and sometimes rows have to be changed to reflect a new situation. Often, changes are interconnected. A row in a table that maps orders to customers is not very useful without the corresponding entry in the `orders` table. Most, if not all, datasets eventually get changed. As a data management system, managing change is thus not optional. However, managing changes properly is difficult.

Early data management systems researchers invented a concept called “transactions”, the notions of which were first formalized in the 1980s. In essence, transactionality and the well-known ACID principles describe a set of guarantees that a data management system has to provide in order to be considered safe. ACID is an acronym that stands for Atomicity, Consistency, Isolation and Durability.

The ACID principles are not a theoretical exercise. Much like the rules governing airplanes or trains, they have been “written in blood” – they are hard-won lessons from decades of data management practice. It is very hard for an application to reason correctly when dealing with non-ACID systems. The end result of such problems is often corrupted data or data that no longer reflects reality accurately. For example, rows can be duplicated or missing.

DuckDB provides full ACID guarantees by default without additional configuration. In this blog post, we will describe in detail what that means together with concrete examples, and show how you can take advantage of this functionality.

Atomicity

Atomicity means that *either all changes in a set of updates happen or none of them happen*. Consider the example below, where we insert two rows in two separate tables. The inserts themselves are separate statements, but they can be made atomic by wrapping them in a transaction:

```
CREATE TABLE customer (id INTEGER, name VARCHAR);
CREATE TABLE orders (customer_id INTEGER, item VARCHAR);

BEGIN TRANSACTION;
INSERT INTO customer VALUES (42, 'DuckDB Labs');
INSERT INTO orders VALUES (42, 'stale bread');
COMMIT;

SELECT * FROM orders;
```

customer_id int32	item varchar
42	stale bread

By wrapping the changes in a transaction, we can be sure that *either both rows are written, or none of them are written*. The `BEGIN TRANSACTION` statement signifies all following statements belong to that transaction. The `COMMIT` signifies the end of the transaction – and will persist the changes to disk.

It is also possible to undo a set of changes by issuing a `ROLLBACK` at the end of a transaction. This will ensure that none of the changes made in the transaction are persisted.

```
BEGIN TRANSACTION;
INSERT INTO orders VALUES (42, 'iceberg lettuce');
INSERT INTO orders VALUES (42, 'dried worms');
ROLLBACK;
SELECT * FROM orders;
```

customer_id int32	item varchar
42	stale bread

As we can see, the two new rows have not been inserted permanently.

Atomicity is great to have because it allows the application to move the database from one consistent state to another consistent state without ever having to worry about intermediate states being visible to an application.

We should note that queries by default run in the so-called "auto-commit" mode, where each query will automatically be run in its own transaction. That said, even for these single-statement queries, transactions are very useful. For example, when bulk loading data into a table using an `INSERT` or `COPY` command, either *all* of the data is loaded, or *none* of the data is loaded. The system will not partially load a CSV file into a table.

We should also note that in DuckDB *schema changes are also transactional*. This means that you can create or delete tables, as well as alter the schema of a table, all within the safety of a transaction. It also means that you can undo any of these operations by issuing a `ROLLBACK`.

Consistency

Consistency means that all of the constraints that are defined in the database must always hold, both before and after a transaction. The constraints can never be violated. Examples of constraints are `PRIMARY KEY` or `FOREIGN KEY` constraints.

```
CREATE TABLE customer (id INTEGER, name VARCHAR, PRIMARY KEY (id));

INSERT INTO customer VALUES (42, 'DuckDB Labs');
INSERT INTO customer VALUES (42, 'Wilbur the Duck');
```

In the example above, the `customer` table requires the `id` column to be unique for all entries, otherwise multiple customers would be associated with the same orders. We can enforce this constraint by defining a so-called `PRIMARY KEY` on that column. When we insert two entries with the same id, the consistency check fails, and we get an error message:

```
Constraint Error: Duplicate key "id: 42" violates primary key
constraint. (...)
```

Having these kinds of constraints in place is a great way to make sure data *remains* consistent even after many updates have taken place.

Isolation

Isolation means that concurrent transactions are isolated from one another. A database can have many clients interacting with it *at the same time,* causing many transactions to happen all at once. An easy way of isolating these transactions is to execute them one after another. However, that would be prohibitively slow. Thousands of requests might have to wait for one particularly slow one.

To avoid this problem, transactions are typically executed *interleaved*. However, as those transactions change data, one must ensure that each transaction is logically *isolated* – it only ever sees a consistent state of the database and can – for example – never read data from a transaction that has not yet committed.

DuckDB does not have connections in the typical sense – as it is not a client/server database that allows separate applications to connect to it. However, DuckDB has full multi-client support within a single application. The user can create multiple clients that all connect to the same DuckDB instance. The transactions can be run concurrently and they are isolated using Snapshot Isolation.

The way that multiple connections are created differs per client. Below is an example where we showcase the transactionality of the system using the Python client.

```
import duckdb

con1 = duckdb.connect(":memory:mydb")
con1.sql("CREATE TABLE customer (id INTEGER, name VARCHAR)")

con1.sql("INSERT INTO customer VALUES (42, 'DuckDB Labs')")

con1.begin()
con1.sql("INSERT INTO customer VALUES (43, 'Wilbur the Duck')")
# no commit!

# start a new connection
con2 = duckdb.connect(":memory:mydb")
con2.sql("SELECT name FROM customer").show()

#
# ┌──────────────┐
# │     name     │
# │   varchar    │
# ├──────────────┤
# │ DuckDB Labs  │
# └──────────────┘
#

# commit from the first connection
con1.commit()

# now the changes are visible
con2.sql("SELECT name FROM customer").show()

#
# ┌──────────────────┐
# │       name       │
# │     varchar      │
# ├──────────────────┤
# │ DuckDB Labs      │
# │ Wilbur the Duck  │
# └──────────────────┘
#
```

As you can see, we have two connections to the same database, and the first connection inserts the `Wilbur the Duck` customer but *does not yet commit the change*. Meanwhile, the second connection reads from the customer table. The result does not yet show the new entry, because the two transactions are isolated from each other with regards to uncommitted changes. After the first connection commits, the second connection can read its changes.

Durability

Finally, **durability** is the behavior of a system under failure. This is important as a process might crash or power to a computer may be lost. A database system now needs to ensure that *all committed transactions* are durable, meaning their effects will be visible after restarting the database. Transactions that have not yet completed cannot leave any visible traces behind. Databases typically guarantee this property by keeping close tabs on the various caches, for example by using `fsync` to force changes to disk as transactions complete. Skipping the `fsync` is a common "optimization" that endangers durability.

Here is an example, again using Python:

```
import duckdb
import os
import signal

con = duckdb.connect("mydb.duckdb")
con.sql("CREATE TABLE customer (id INTEGER, name VARCHAR)")
con.sql("INSERT INTO customer VALUES (42, 'DuckDB Labs')")

# begin a transaction
con.begin()
con.sql("INSERT INTO customer VALUES (43, 'Wilbur the Duck')")
# no commit!

os.kill(os.getpid(), signal.SIGKILL)
```

After restarting, we can check the `customer` table:

```
import duckdb

con = duckdb.connect("mydb.duckdb")
con.sql("SELECT name FROM customer").show()
```

```
┌──────────────┐
│     name     │
│   varchar    │
├──────────────┤
│ DuckDB Labs  │
└──────────────┘
```

In this example, we first create the customer table in the database file `mydb.duckdb`. We then insert a single row with DuckDB Labs as a first transaction. Then, we begin but *do not commit* a second transaction that adds the `Wilbur the Duck` entry. If we then kill the process and with it the database, we can see that upon restart only the `DuckDB Labs` entry has survived. This is because the second transaction was not committed and hence not subject to durability. Of course, this gets more complicated when non-clean exits such as operating system crashes have to be considered. DuckDB also guarantees durability in those circumstances, some more on this below.

Why ACID in OLAP?

There are two main classes of data management systems, transactional systems (OLTP) and analytical systems (OLAP). As the name implies, transactional systems are far more concerned with guaranteeing the ACID properties than analytical ones. Systems like the venerable PostgreSQL deservedly pride themselves on doing the "right thing" with regard to providing transactional guarantees by default. Even NoSQL transactional systems such as MongoDB that swore off guaranteeing the ACID principles "for performance" early on had to eventually "roll back" to offering ACID guarantees with one or two hurdles along the way.

Analytical systems such as DuckDB – in principle – have less of an imperative to provide strong transactional guarantees. They are often not the so-called "system of record", which is the data management system that is considered the source truth. In fact, DuckDB offers various connectors to load data from systems of record, like the PostgreSQL scanner. If an OLAP database would become corrupted, it is often possible to recover from that source of truth. Of course, that first requires that users notice that something has gone wrong, which is not always simple to detect. For example, a common mistake is ingesting data from the same CSV file twice into a database because the first attempt went wrong at some point. This can lead to duplicate rows causing incorrect aggregate results. ACID prevents these kinds of problems. ACID properties enable useful functionality in OLAP systems. For example:

Concurrent Ingestion and Reporting. As change is continuous, we often have data ingestion streams adding new data to a database system. In analytical systems, it is common to have a single connection append new data to a database, while other connections read from the database in order to e.g., generate graphs and reports. If these connections are isolated, then the generated graphs and aggregates will always be executed over a complete and consistent snapshot of the database, ensuring that the generated graphs and aggregates are correct.

Rolling Back Incorrect Transformations. When analyzing data, a common pattern is loading data from data sets stored in flat files followed by performing a number of transformations on that data. For example, we might load a data set from a CSV file, followed by cleaning up `NULL` values and then deleting incomplete rows. If we make an incorrect transformation, it is possible we accidentally delete too many rows.

This is not the end of the world, as we can recover by re-reading from the original CSV files. However, we can save ourselves a lot of time by wrapping the transformations in a transaction and rolling back when something goes wrong. For example:

```
CREATE TABLE people AS SELECT * FROM 'people.csv';

BEGIN TRANSACTION;
UPDATE people SET age = NULL WHERE age = -99;
-- oops, we deleted all rows!
DELETE FROM people WHERE name <> 'non-existent name';
-- we can recover our original table by rolling back the delete
ROLLBACK;
```

SQL Assertions. When a (non-syntax) error occurs in a transaction, the transaction is automatically aborted, and the changes cannot be committed. We can use this property of transactions to add assertions to our transactions. When one of these assertions is triggered, an error is raised, and the transaction cannot be committed. We can use the `error` function to define our own `assert` macro:

```
CREATE MACRO assert(condition, message) AS
    CASE WHEN NOT condition THEN error(message) END;
```

We can then use this `assert` macro to assert that the `people` table is not empty:

```
CREATE TABLE people AS SELECT * FROM 'people.csv';

BEGIN TRANSACTION;
UPDATE people SET age = NULL WHERE age = -99;
-- oops, we deleted all rows!
DELETE FROM people WHERE name <> 'non-existent name';
SELECT assert(
           (SELECT count(*) FROM people) > 0,
           'People should not be empty'
       );
COMMIT;
```

When the assertion triggers, the transaction is automatically aborted, and the changes are rolled back.

Full TPC-H Benchmark Implementation

The Transaction Processing Performance Council (TPC) is an industry association of data management systems and hardware vendors. TPC publishes database benchmark specifications and oversees auditing of benchmark results, which it then publishes on its website. There are various benchmarks aimed at different use cases. The TPC-H decision support benchmark is specifically aimed at analytical query processing on large volumes of data. Its famous 22 SQL queries and data generator specifics have been thourougly analyzed by both database vendors and academics ad nauseam.

It is less well known that the official TPC-H benchmark includes *data modification transactions* that require ACID compliance, which is not too-surprising given the name of the organization. For one-off performance shoot-outs, the updates are typically ignored and only the runtimes of the 22 queries on a static dataset are reported. Such results are purely informational and cannot be audited or formally published by the TPC. But as we have argued above, change is inevitable, so let's perform the TPC-H experiments *with updates* with DuckDB.

TPC-H generates data for a fictional company selling things. The largest tables are `orders` and `lineitem`, which contains elements of each order. The benchmark can generate data of different sizes, the size is controlled by a so-called "scale factor" (SF). The specification defines two "refresh functions", that modify the database. The first refresh function will add `SF * 1500` new rows into the `orders` table, and randomly between 1 and 7 new entries for each order into the `lineitem` table. The second refresh function will delete `SF * 1500` entries from the `orders` table along with the associated `lineitem` entries. The benchmark data generator `dbgen` can generate an arbitrary amount of refresh function CSV files with new entries for `orders` and `lineitem` along with rows to be deleted.

Metrics

TPC-H's main benchmark metric is combined from both a "power" and a "throughput" test result.

The power test will run the first refresh function and time it, then run the 22 queries, then run the second refresh function, and calculate the geometric mean of all timings. With a scale factor of 100 and DuckDB 1.1.1 on a MacBook Pro with an M3 Max CPU and 64 GB of RAM, we get a *Power@Size value of 650 536*.

The throughput test will run a number of concurrent query "streams" that execute the 22 benchmark queries in shuffled order in parallel. In addition, a single refresh stream will run both refresh functions a number of times. The number of query streams and refresh sets is derived from the scale factor. For SF100, there are 5 query streams and 10 refresh sets. For our experiment, we get a *Throughput@Size of 452 571*. Results are hard to compare, but the result does not look too shabby when compared with the official result list.

ACID Tests

Section 3 of the TPC-H benchmark specification discusses the ACID properties in detail. The specification defines a set of tests to stress the ACID guarantees of a data management system. The spec duly notes that no test can prove that the ACID properties are fully supported, passing them is a "necessary but not sufficient condition" of compliance. Below, we will give an overview of what is tested.

The tests specify an "ACID Transaction", which modifies the `lineitem` and `orders` tables in such a way that an invariant holds: the `orders` table contains a total sum of all the prices of all the lineitems that belong to this order. The transaction picks a random order, and modifies the last lineitem to have a new price. It then re-calculates the order total price and updates the `orders` table with that. Finally, the transaction inserts information about which row was updated when and the price delta used in a `history` table.

To test *atomicity*, the ACID transaction is ran for a random order and then committed. It is verified that the database has been changed accordingly with the specified values. The test is repeated but this time the transaction is aborted. It is verified that the database has not been changed.

For *consistency*, a number of threads run the ACID transaction in parallel 100 times on random orders. Before and after the test, a consistency condition is checked, which essentially makes sure that the sum of all lineitem prices for an order is consistent with the sum in the order.

To test *isolation*, one thread will run the transaction, but not commit or rollback yet. Another connection will make sure the changes are not visible to it. Another set of tests will have two threads running transactions on the same order, and ensure that one of them is aborted by the system due to the conflict.

Finally, to test *durability*, a number of threads run the ACID transaction and log the results. They are allowed to complete at least 100 transactions each. Then, a failure is caused, in our case, we simply killed the process (using `SIGKILL`). Then, the database system is allowed to recover the committed changes from the write-ahead log. The log is checked to ensure that there are no log entries that are not reflected in the `history` table and there are no history entries that don't have log entries, minus very few that might have been lost in flight (i.e., persisted by the database but not yet logged by the benchmark driver). Finally, the consistency is checked again.

We're happy to report that DuckDB passed all tests.

Our scripts to run the benchmark are available on GitHub. We are planning to perform a formal audit of our results in the future. We will update this post when that happens.

Conclusion

Change in datasets is inevitable, and data management systems need to be able to safely manage change. DuckDB supports strong ACID guarantees that allow for safe and concurrent data modification. We have run extensive experiments with TPC-H's transactional validation tests and found that they pass.

Creating a SQL-Only Extension for Excel-Style Pivoting in DuckDB

Publication date: 2024-09-27

Author: Alex Monahan

TL;DR: Easily create sharable extensions using only SQL macros that can apply to any table and any columns. We demonstrate the power of this capability with the pivot_table extension that provides Excel-style pivoting.

The Power of SQL-Only Extensions

SQL is not a new language. As a result, it has historically been missing some of the modern luxuries we take for granted. With version 1.1, DuckDB has launched community extensions, bringing the incredible power of a package manager to the SQL language. A bold goal of ours is for DuckDB to become a convenient way to wrap any C++ library, much the way that Python does today, but across any language with a DuckDB client.

For extension builders, compilation and distribution are much easier. For the user community, installation is as simple as two commands:

```
INSTALL pivot_table FROM community;
LOAD pivot_table;
```

The extension can then be used in any query through SQL functions.

However, **not all of us are C++ developers**! Can we, as a SQL community, build up a set of SQL helper functions? What would it take to build these extensions with *just SQL?*

Reusability

Traditionally, SQL is highly customized to the schema of the database on which it was written. Can we make it reusable? Some techniques for reusability were discussed in the SQL Gymnasics post, but now we can go even further. With version 1.1, DuckDB's world-class friendly SQL dialect makes it possible to create macros that can be applied:

- To any tables
- On any columns
- Using any functions

The new ability to work **on any tables** is thanks to the `query` and `query_table` functions! The `query` function is a safe way to execute `SELECT` statements defined by SQL strings, while `query_table` is a way to make a `FROM` clause pull from multiple tables at once. They are very powerful when used in combination with other friendly SQL features like the `COLUMNS` expression and `LIST` lambda functions.

Community Extensions as a Central Repository

Traditionally, there has been no central repository for SQL functions across databases, let alone across companies! DuckDB's community extensions can be that knowledge base. DuckDB extensions can be used across all languages with a DuckDB client, including Python, NodeJS, Java, Rust, Go, and even WebAssembly (Wasm)!

If you are a DuckDB fan and a SQL user, you can share your expertise back to the community with an extension. This post will show you how! No C++ knowledge is needed – just a little bit of copy/paste and GitHub Actions handles all the compilation. If I can do it, you can do it!

Powerful SQL

All that said, just how valuable can a SQL MACRO be? Can we do more than make small snippets? I'll make the case that you can do quite complex and powerful operations in DuckDB SQL using the `pivot_table` extension as an example. The `pivot_table` function allows for Excel-style pivots, including `subtotals`, `grand_totals`, and more. It is also very similar to the Pandas `pivot_table` function, but with all the scalability and speed benefits of DuckDB. It contains over **250 tests**, so it is intended to be useful beyond just an example!

To achieve this level of flexibility, the `pivot_table` extension uses many friendly and advanced SQL features:

- The `query` function to execute a SQL string
- The `query_table` function to query a list of tables
- The `COLUMNS` expression to select a dynamic list of columns
- List lambda functions to build up the SQL statement passed into `query`
  - `list_transform` for string manipulation like quoting
  - `list_reduce` to concatenate strings together
  - `list_aggregate` to sum multiple columns and identify subtotal and grand total rows
- Bracket notation for string slicing
- `UNION ALL BY NAME` to stack data by column name for subtotals and grand totals
- `SELECT * REPLACE` to dynamically clean up subtotal columns
- `SELECT * EXCLUDE` to remove internally generated columns from the final result
- `GROUPING SETS` and `ROLLUP` to generate subtotals and grand totals
- `UNNEST` to convert lists into separate rows for `values_axis := 'rows'`
- `MACRO`s to modularize the code
- `ORDER BY ALL` to order the result dynamically
- `ENUM`s to determine what columns to pivot horizontally
- And of course the `PIVOT` function for horizontal pivoting!

DuckDB's innovative syntax makes this extension possible!

So, we now have all 3 ingredients we will need: a central package manager, reusable macros, and enough syntactic flexibility to do valuable work.

Create Your Own SQL Extension

Let's walk through the steps to creating your own SQL-only extension.

Writing the Extension

Extension Setup

The first step is to create your own GitHub repo from the DuckDB Extension Template for SQL by clicking *Use this template.*

Then clone your new repository onto your local machine using the terminal:

```
git clone --recurse-submodules \
    https://github.com/<your_github_username>/<your_extension_repo>.git
```

Note that `--recurse-submodules` will ensure DuckDB is pulled which is required to build the extension.

Next, replace the name of the example extension with the name of your extension in all the right places by running the Python script below.

> **Note.** If you don't have Python installed, head to python.org and follow those instructions. This script doesn't require any libraries, so Python is all you need! (No need to set up any environments.)

```
python3 ./scripts/bootstrap-template.py <extension_name_you_want>
```

Initial Extension Test

At this point, you can follow the directions in the README to build and test locally if you would like. However, even easier, you can simply commit your changes to git and push them to GitHub, and GitHub Actions can do the compilation for you! GitHub Actions will also run tests on your extension to validate it is working properly.

> **Note.** The instructions are not written for a Windows audience, so we recommend GitHub Actions in that case!

```
git add -A
git commit -m "Initial commit of my SQL extension!"
git push
```

Write Your SQL Macros

It is likely a bit faster to iterate if you test your macros directly in DuckDB. After you have written your SQL, we will move it into the extension. The example we will use demonstrates how to pull a dynamic set of columns from a dynamic table name (or a view name!).

```
CREATE OR REPLACE MACRO select_distinct_columns_from_table(table_name, columns_list) AS TABLE (
    SELECT DISTINCT
        COLUMNS(column_name -> list_contains(columns_list, column_name))
    FROM query_table(table_name)
    ORDER BY ALL
);

FROM select_distinct_columns_from_table('duckdb_types', ['type_category']);
```

type_category
BOOLEAN
COMPOSITE
DATETIME
NUMERIC
STRING
NULL

Add SQL Macros

Technically, this is the C++ part, but we are going to do some copy/paste and use GitHub Actions for compiling so it won't feel that way!

DuckDB supports both scalar and table macros, and they have slightly different syntax. The extension template has an example for each (and code comments too!) inside the file named `<your_extension_name>.cpp`. Let's add a table macro here since it is the more complex one. We will copy the example and modify it!

{% raw %}

```
static const DefaultTableMacro <your_extension_name>_table_macros[] = {
    {DEFAULT_SCHEMA, "times_two_table", {"x", nullptr}, {{"two", "2"}, {nullptr, nullptr}}, R"(SELECT x
    * two AS output_column;)"},
    {
        DEFAULT_SCHEMA, // Leave the schema as the default
        "select_distinct_columns_from_table", // Function name
        {"table_name", "columns_list", nullptr}, // Parameters
        {{nullptr, nullptr}}, // Optional parameter names and values (we choose not to have any here)
        // The SQL text inside of your SQL Macro, wrapped in R"( )", which is a raw string in C++
        R"(
```

```
            SELECT DISTINCT
                COLUMNS(column_name -> list_contains(columns_list, column_name))
            FROM query_table(table_name)
            ORDER BY ALL
        )"
    },
    {nullptr, nullptr, {nullptr}, {{nullptr, nullptr}}, nullptr}
    };
```

{% endraw %}

That's it! All we had to provide were the name of the function, the names of the parameters, and the text of our SQL macro.

Testing the Extension

We also recommend adding some tests for your extension to the `<your_extension_name>.test` file. This uses sqllogictest to test with just SQL! Let's add the example from above.

> **Note.** In sqllogictest, `query I` indicates that there will be 1 column in the result. We then add `----` and the resultset in tab separated format with no column names.

```
query I
FROM select_distinct_columns_from_table('duckdb_types', ['type_category']);
----
BOOLEAN
COMPOSITE
DATETIME
NUMERIC
STRING
NULL
```

Now, just add, commit, and push your changes to GitHub like before, and GitHub Actions will compile your extension and test it!

If you would like to do further ad-hoc testing of your extension, you can download the extension from your GitHub Actions run's artifacts and then install it locally using these steps.

Uploading to the Community Extensions Repository

Once you are happy with your extension, it's time to share it with the DuckDB community! Follow the steps in the Community Extensions post. A summary of those steps is:

1. Send a PR with a metadata file `description.yml` that contains the description of the extension. For example, the h3 community extension uses the following YAML configuration:

```yaml
extension:
  name: h3
  description: Hierarchical hexagonal indexing for geospatial data
  version: 1.0.0
  language: C++
  build: cmake
  license: Apache-2.0
  maintainers:
    - isaacbrodsky

repo:
  github: isaacbrodsky/h3-duckdb
  ref: 3c8a5358e42ab8d11e0253c70f7cc7d37781b2ef
```

2. Wait for approval from the maintainers.

And there you have it! You have created a shareable DuckDB community extension. Now let's have a look at the `pivot_table` extension as an example of just how powerful a SQL-only extension can be.

Capabilities of the `pivot_table` Extension

The `pivot_table` extension supports advanced pivoting functionality that was previously only available in spreadsheets, dataframe libraries, or custom host language functions. It uses the Excel pivoting API: `values`, `rows`, `columns`, and `filters` – handling 0 or more of each of those parameters. However, not only that, but it supports `subtotals` and `grand_totals`. If multiple `values` are passed in, the `values_axis` parameter allows the user to choose if each value should get its own column or its own row.

Why is this a good example of how DuckDB moves beyond traditional SQL? The Excel pivoting API requires dramatically different SQL syntax depending on which parameters are in use. If no `columns` are pivoted outward, a `GROUP BY` is all that is needed. However, once `columns` are involved, a `PIVOT` is required.

This function can operate on one or more `table_names` that are passed in as a parameter. Any set of tables (or views!) will first be vertically stacked and then pivoted.

Example Using `pivot_table`

Check out a live example using the extension in the DuckDB Wasm shell here!

product_line	product	year	quarter	revenue	cost
Waterfowl watercraft	Duck boats	2022	Q1	100	100
Waterfowl watercraft	Duck boats	2022	Q2	200	100
Waterfowl watercraft	Duck boats	2022	Q3	300	100
Waterfowl watercraft	Duck boats	2022	Q4	400	100
Waterfowl watercraft	Duck boats	2023	Q1	500	100
Waterfowl watercraft	Duck boats	2023	Q2	600	100
Waterfowl watercraft	Duck boats	2023	Q3	700	100
Waterfowl watercraft	Duck boats	2023	Q4	800	100
Duck Duds	Duck suits	2022	Q1	10	10
Duck Duds	Duck suits	2022	Q2	20	10
Duck Duds	Duck suits	2022	Q3	30	10
Duck Duds	Duck suits	2022	Q4	40	10
Duck Duds	Duck suits	2023	Q1	50	10
Duck Duds	Duck suits	2023	Q2	60	10
Duck Duds	Duck suits	2023	Q3	70	10
Duck Duds	Duck suits	2023	Q4	80	10
Duck Duds	Duck neckties	2022	Q1	1	1
Duck Duds	Duck neckties	2022	Q2	2	1
Duck Duds	Duck neckties	2022	Q3	3	1
Duck Duds	Duck neckties	2022	Q4	4	1
Duck Duds	Duck neckties	2023	Q1	5	1
Duck Duds	Duck neckties	2023	Q2	6	1

product_line	product	year	quarter	revenue	cost
Duck Duds	Duck neckties	2023	Q3	7	1
Duck Duds	Duck neckties	2023	Q4	8	1

Next, we install the extension from the community repository:

```
INSTALL pivot_table FROM community;
LOAD pivot_table;
```

Now we can build pivot tables like the one below. There is a little bit of boilerplate required, and the details of how this works will be explained shortly.

```
DROP TYPE IF EXISTS columns_parameter_enum;

CREATE TYPE columns_parameter_enum AS ENUM (
    FROM build_my_enum(['business_metrics'],     -- table_names
                       ['year', 'quarter'],      -- columns
                       [])                       -- filters
);

FROM pivot_table(['business_metrics'],           -- table_names
                 ['sum(revenue)', 'sum(cost)'], -- values
                 ['product_line', 'product'],    -- rows
                 ['year', 'quarter'],            -- columns
                 [],                             -- filters
                 subtotals := 1,
                 grand_totals := 1,
                 values_axis := 'rows'
                 );
```

product_line	product	value_names	2022_Q1	2022_Q2	2022_Q3	2022_Q4	2023_Q1	2023_Q2	2023_Q3	2023_Q4
Duck Duds	Duck neckties	sum(cost)	1	1	1	1	1	1	1	1
Duck Duds	Duck neckties	sum(revenue)	1	2	3	4	5	6	7	8
Duck Duds	Duck suits	sum(cost)	10	10	10	10	10	10	10	10
Duck Duds	Duck suits	sum(revenue)	10	20	30	40	50	60	70	80
Duck Duds	Subtotal	sum(cost)	11	11	11	11	11	11	11	11
Duck Duds	Subtotal	sum(revenue)	11	22	33	44	55	66	77	88
Waterfowl water-craft	Duck boats	sum(cost)	100	100	100	100	100	100	100	100
Waterfowl water-craft	Duck boats	sum(revenue)	100	200	300	400	500	600	700	800

product_line	product	value_names	2022_Q1	2022_Q2	2022_Q3	2022_Q4	2023_Q1	2023_Q2	2023_Q3	2023_Q4
Waterfowl watercraft	Subtotal	sum(cost)	100	100	100	100	100	100	100	100
Waterfowl watercraft	Subtotal	sum(revenue)	100	200	300	400	500	600	700	800
Grand Total	Grand Total	sum(cost)	111	111	111	111	111	111	111	111
Grand Total	Grand Total	sum(revenue)	111	222	333	444	555	666	777	888

How the `pivot_table` Extension Works

The `pivot_table` extension is a collection of multiple scalar and table SQL macros. This allows the logic to be modularized. You can see below that the functions are used as building blocks to create more complex functions. This is typically difficult to do in SQL, but it is easy in DuckDB!

The functions and a brief description of each follows.

Building Block Scalar Functions

- `nq`: "No quotes" – Escape semicolons in a string to prevent SQL injection
- `sq`: "Single quotes" – Wrap a string in single quotes and escape embedded single quotes
- `dq`: "Double quotes" – Wrap in double quotes and escape embedded double quotes
- `nq_list`: Escape semicolons for each string in a list. Uses `nq`.
- `sq_list`: Wrap each string in a list in single quotes. Uses `sq`.
- `dq_list`: Wrap each string in a list in double quotes. Uses `dq`.
- `nq_concat`: Concatenate a list of strings together with semicolon escaping. Uses `nq_list`.
- `sq_concat`: Concatenate a list of strings together, wrapping each in single quotes. Uses `sq_list`.
- `dq_concat`: Concatenate a list of strings together, wrapping each in double quotes. Uses `dq_list`.

Functions Creating During Refactoring for Modularity

- `totals_list`: Build up a list as a part of enabling `subtotals` and `grand_totals`.
- `replace_zzz`: Rename `subtotal` and `grand_total` indicators after sorting so they are more friendly.

Core Pivoting Logic Functions

- `build_my_enum`: Determine which new columns to create when pivoting horizontally. Returns a table. See below for details.
- `pivot_table`: Based on inputs, decide whether to call `no_columns`, `columns_values_axis_columns` or `columns_values_axis_rows`. Execute `query` on the SQL string that is generated. Returns a table. See below for details.
  - `no_columns`: Build up the SQL string for `query` to execute when no `columns` are pivoted out.
  - `columns_values_axis_columns`: Build up the SQL string for `query` to execute when pivoting horizontally with each entry in `values` receiving a separate column.
  - `columns_values_axis_rows`: Build up the SQL string for `query` to execute when pivoting horizontally with each entry in `values` receiving a separate row.

- `pivot_table_show_sql`: Return the SQL string that would have been executed by `query` for debugging purposes.

The `build_my_enum` Function

The first step in using the `pivot_table` extension's capabilities is to define an `ENUM` (a user-defined type) containing all of the new column names to create when pivoting horizontally called `columns_parameter_enum`. DuckDB's automatic `PIVOT` syntax can automatically define this, but in our case, we need 2 explicit steps. The reason for this is that automatic pivoting runs 2 statements behind the scenes, but a `MACRO` must only be a single statement. If the `columns` parameter is not in use, this step is essentially a no-op, so it can be omitted or included for consistency (recommended).

The `query` and `query_table` functions only support `SELECT` statements (for security reasons), so the dynamic portion of the `ENUM` creation occurs in the function `build_my_enum`. If this type of usage becomes common, features could be added to DuckDB to enable a `CREATE OR REPLACE` syntax for `ENUM` types, or possibly even temporary enums. That would reduce this pattern from 3 statements down to 2. Please let us know!

The `build_my_enum` function uses a combination of `query_table` to pull from multiple input tables, and the `query` function so that double quotes (and correct character escaping) can be completed prior to passing in the list of table names. It uses a similar pattern to the core `pivot_table` function: build up a SQL query as a string, then call it with `query`. The SQL string is constructed using list lambda functions and the building block functions for quoting.

The `pivot_table` Function

At its core, the `pivot_table` function determines the SQL required to generate the desired pivot based on which parameters are in use.

Since this SQL statement is a string at the end of the day, we can use a hierarchy of scalar SQL macros rather than a single large macro. This is a common traditional issue with SQL – it tends to not be very modular or reusable, but we are able to compartmentalize our logic wth DuckDB's syntax.

> **Note.** If a non-optional parameter is not in use, an empty list (`[]`) should be passed in.

- `table_names`: A list of table or view names to aggregate or pivot. Multiple tables are combined with `UNION ALL BY NAME` prior to any other processing.
- `values`: A list of aggregation metrics in the format `['agg_fn_1(col_1)', 'agg_fn_2(col_2)', ...]`.
- `rows`: A list of column names to `SELECT` and `GROUP BY`.
- `columns`: A list of column names to `PIVOT` horizontally into a separate column per value in the original column. If multiple column names are passed in, only unique combinations of data that appear in the dataset are pivoted.
  - Ex: If passing in a `columns` parameter like `['continent', 'country']`, only valid `continent` / `country` pairs will be included.
  - (no `Europe_Canada` column would be generated).
- `filters`: A list of `WHERE` clause expressions to be applied to the raw dataset prior to aggregating in the format `['col_1 = 123', 'col_2 LIKE ''woot%''', ...]`.
  - The `filters` are combined with `AND`.
- `values_axis` (Optional): If multiple `values` are passed in, determine whether to create a separate row or column for each value. Either `rows` or `columns`, defaulting to `columns`.
- `subtotals` (Optional): If enabled, calculate the aggregate metric at multiple levels of detail based on the `rows` parameter. Either 0 or 1, defaulting to 0.
- `grand_totals` (Optional): If enabled, calculate the aggregate metric across all rows in the raw data in addition to at the granularity defined by `rows`. Either 0 or 1, defaulting to 0.

No Horizontal Pivoting (No `columns` in Use)

If not using the `columns` parameter, no columns need to be pivoted horizontally. As a result, a `GROUP BY` statement is used. If `subtotals` are in use, the `ROLLUP` expression is used to calculate the `values` at the different levels of granularity. If `grand_totals` are in use, but not `subtotals`, the `GROUPING SETS` expression is used instead of `ROLLUP` to evaluate across all rows.

In this example, we build a summary of the `revenue` and `cost` of each `product_line` and `product`.

```
FROM pivot_table(['business_metrics'],
                 ['sum(revenue)', 'sum(cost)'],
                 ['product_line', 'product'],
                 [],
                 [],
                 subtotals := 1,
                 grand_totals := 1,
                 values_axis := 'columns'
                 );
```

product_line	product	sum(revenue)	sum("cost")
Duck Duds	Duck neckties	36	8
Duck Duds	Duck suits	360	80
Duck Duds	Subtotal	396	88
Waterfowl watercraft	Duck boats	3600	800
Waterfowl watercraft	Subtotal	3600	800
Grand Total	Grand Total	3996	888

Pivot Horizontally, One Column per Metric in `values`

Build up a `PIVOT` statement that will pivot out all valid combinations of raw data values within the `columns` parameter. If `subtotals` or `grand_totals` are in use, make multiple copies of the input data, but replace appropriate column names in the `rows` parameter with a string constant. Pass all expressions in `values` to the `PIVOT` statement's `USING` clause so they each receive their own column.

We enhance our previous example to pivot out a separate column for each `year` / `value` combination:

```
DROP TYPE IF EXISTS columns_parameter_enum;

CREATE TYPE columns_parameter_enum AS ENUM (
    FROM build_my_enum(['business_metrics'],
                       ['year'],
                       [])
);

FROM pivot_table(['business_metrics'],
                 ['sum(revenue)', 'sum(cost)'],
                 ['product_line', 'product'],
                 ['year'],
                 [],
                 subtotals := 1,
                 grand_totals := 1,
                 values_axis := 'columns'
                 );
```

product_line	product	2022_sum(revenue)	2022_sum("cost")	2023_sum(revenue)	2023_sum("cost")
Duck Duds	Duck neckties	10	4	26	4
Duck Duds	Duck suits	100	40	260	40
Duck Duds	Subtotal	110	44	286	44
Waterfowl watercraft	Duck boats	1000	400	2600	400
Waterfowl watercraft	Subtotal	1000	400	2600	400
Grand Total	Grand Total	1110	444	2886	444

Pivot Horizontally, One Row per Metric in `values`

Build up a separate `PIVOT` statement for each metric in `values` and combine them with `UNION ALL BY NAME`. If `subtotals` or `grand_totals` are in use, make multiple copies of the input data, but replace appropriate column names in the `rows` parameter with a string constant.

To simplify the appearance slightly, we adjust one parameter in our previous query and set `values_axis := 'rows'`:

```
DROP TYPE IF EXISTS columns_parameter_enum;

CREATE TYPE columns_parameter_enum AS ENUM (
    FROM build_my_enum(['business_metrics'],
                       ['year'],
                       [])
);

FROM pivot_table(['business_metrics'],
                 ['sum(revenue)', 'sum(cost)'],
                 ['product_line', 'product'],
                 ['year'],
                 [],
                 subtotals := 1,
                 grand_totals := 1,
                 values_axis := 'rows'
                 );
```

product_line	product	value_names	2022	2023
Duck Duds	Duck neckties	sum(cost)	4	4
Duck Duds	Duck neckties	sum(revenue)	10	26
Duck Duds	Duck suits	sum(cost)	40	40
Duck Duds	Duck suits	sum(revenue)	100	260
Duck Duds	Subtotal	sum(cost)	44	44
Duck Duds	Subtotal	sum(revenue)	110	286
Waterfowl watercraft	Duck boats	sum(cost)	400	400
Waterfowl watercraft	Duck boats	sum(revenue)	1000	2600
Waterfowl watercraft	Subtotal	sum(cost)	400	400
Waterfowl watercraft	Subtotal	sum(revenue)	1000	2600
Grand Total	Grand Total	sum(cost)	444	444
Grand Total	Grand Total	sum(revenue)	1110	2886

Conclusion

With DuckDB 1.1, sharing your SQL knowledge with the community has never been easier! DuckDB's community extension repository is truly a package manager for the SQL language. Macros in DuckDB are now highly reusable (thanks to `query` and `query_table`), and DuckDB's SQL syntax provides plenty of power to accomplish complex tasks.

Please let us know if the `pivot_table` extension is helpful to you – we are open to both contributions and feature requests! Together we can write the ultimate pivoting capability just once and use it everywhere.

In the future, we have plans to further simplify the creation of SQL extensions. Of course, we would love your feedback! Join us on Discord in the `community-extensions` channel.

Happy analyzing!

DuckDB in Python in the Browser with Pyodide, PyScript, and JupyterLite

Publication date: 2024-10-02

Author: Alex Monahan

TL;DR: Run DuckDB in an in-browser Python environment to enable simple querying on remote files, interactive documentation, and easy to use training materials.

{:/nomarkdown}

Time to “Hello World”

The first time that you are using a new library, the most important thing is how quickly you can get to “Hello World”.

> **Note.** Want to see “Hello World”? Jump to the **fully interactive** examples!

Likewise, if someone is visiting any documentation you have written, you want them to quickly and easily get your tool up and running. When you are giving a demo, you want to avoid ”demo hell” and have it work the first try!

If you want to try “expert mode”, try leading an entire conference room of people through those setup steps! The classroom or conference workshop environment makes it far more critical that installation be bulletproof.

Python is one of our favorite ways to use DuckDB, but Python is notoriously difficult to set up – doubly so for a novice programmer. What the heck is a virtual environment? Are you on Windows, Linux, or Mac? Pip or Conda? The new kid on the block uv?

Experienced Pythonistas are not immune either! Many, like me, have been forced to celebrate the time honored and xkcd-chronicled tradition of just wiping everything related to Python and starting from scratch.

How can we make it as easy and as fast as possible to test out DuckDB in Python?

Difficulties of Server-Side Python

One response to this challenge is to host a Python environment on a server for each of your users. This has a number of issues.

Hosting Python on a server yourself is not free. If you have many users, it can be far from free.

If you want to use a free solution like Google Colab, each visitor will need a Google account, and you'll need to be comfortable with Google accessing your data. Plus, it is hard to embed within an existing web page for a seamless experience.

Enter Pyodide

Pyodide is a way to run Python directly in your browser with no installation and no setup, thanks to the power of WebAssembly. That makes it the easiest and fastest way to get a Python environment up and running – just load a web page! All computation happens locally, so it can be served like any static website with tools like GitHub Pages. No server-side Python required!

Another benefit is that Pyodide is nicely sandboxed in the browser environment. Each user gets their own workspace, and since it is all local, it is nice and secure.

Part of what sets Pyodide apart from other in-browser-Python approaches is that it can even run libraries that are written in C, C++, or even Fortran, including much of the Python data science stack. This means that now you can use DuckDB in Pyodide as well! You can even combine it with NumPy, SciPy, and Pandas (in addition to many pure-Python libraries). PyArrow and Ibis have experimental support also.

Use Cases for Pyodide DuckDB

Want to quickly analyze some remote data using either Python or DuckDB?

Pyodide is the fastest way to get your questions answered using Python.

Want to quickly analyze some local data?

Pyodide can also query local files!

Want to make your documentation interactive?

Let your users test out your DuckDB-powered library with ease. We will see an example below that demonstrates the `magic-duckdb` Jupyter plugin to enable SQL cells.

Leading a training session with DuckDB and Python?

Skip the hassles of local installation. There is no need to work 1:1 with the 15% of folks in the audience with some quirky setup! Everyone will get this to work on the first try, in seconds, so you can get to the content you want to teach. Plus, it is free, with no signup required of any kind!

Pyodide Examples

We will cover multiple ways to embed Pyodide-powered-Python directly into your site, so your users can try out your new DuckDB-backed tool with a single click!

- PyScript Editor
  - An editor with nice syntax highlighting
- JupyterLite Notebook
  - A classic notebook environment
- JupyterLite Lab IDE
  - A full development environment

PyScript Editor

This HTML snippet will embed a runnable PyScript editor into any page!

```
<script type="module" src="https://pyscript.net/releases/2024.8.2/core.js"></script>
<script type="py-editor" config='{"packages":["duckdb"]}'>
    import duckdb
    print(duckdb.sql("SELECT '42 in an editor' AS s").fetchall())
</script>
```

Just click the play button and you can execute a DuckDB query directly in the browser. You can edit the code, add new lines, etc. Try it out!

{:/nomarkdown}

JupyterLite Notebook

Here is an example of using an `iframe` that points to a JupyterLite environment that was deployed to GitHub Pages!

This is a fully interactive Python notebook environment, with DuckDB running inside. Feel free to give it a run!

{:/nomarkdown}

Configuring a full JupyterLite environment is only a few steps! The JupyterLite folks have built a demo page that serves as a template and have some great documentation. The main steps are to:

1. Use the JupyterLite Demo Template to create your own repo
2. Enable GitHub Pages for that repo
3. Add and commit a .ipynb file in the `content` folder
4. Visit `https://<your_github_username>.github.io/<YOUR_REPOSITORY_NAME>/notebooks/index.html?path=<your_notebook_name>.ipynb`{:.language-sql .highlight}

Note that it can take a couple of minutes for GitHub Pages to deploy. You can monitor the progress on GitHub's Actions tab.

JupyterLite Lab IDE

After following the steps in the JupterLite Notebook setup, if you change your URL from `/notebooks/` to `/lab/`, you can have a full IDE experience instead! This form factor is a bit harder to embed in another page, but great for interactive use.

This example uses the `magic-duckdb` Jupyter extension that allows us to create SQL cells using `%%dql`.

Follow this link to see the Lab IDE interface, or experiment with the Notebook-style version below.

{:/nomarkdown}

Architecture of DuckDB in Pyodide

So how does DuckDB work in Pyodide exactly? The DuckDB Python client is compiled to WebAssembly (Wasm) in its entirety. This is different than the existing DuckDB Wasm approach, since that is compiling the C++ side of the library only and wrapping it with a JavaScript API. Both approaches use the Emscripten toolchain to do the Wasm compilation. It is DuckDB's design decision to avoid dependencies and the prior investments in DuckDB-Wasm that made this feasible to build in such a short period of time!

The Pyodide team has added DuckDB to their hosted repository of libraries, and even set up DuckDB to run as a part of their CI/CD workflow. That is what enables JupyterLite to simply run `%pip install duckdb`, and PyScript to specify DuckDB as a package in the `py-editor config` parameter or in the `<py-config>` tag. Pyodide then downloads the Wasm-compiled version of the DuckDB library from Pyodide's repository. We want to send a big thank you to the Pyodide team, including Hood Chatham and Gyeongjae Choi, as well as the Voltron Data team including Phillip Cloud for leading the effort to get this to work.

Limitations

Running in the browser is a more restrictive environment (for security purposes), so there are some limitations when using DuckDB in Pyodide. There is no free lunch!

- Single-threaded
  - Pyodide currently limits execution to a single thread
- A few extra steps to query remote files

  - Remote files can't be accessed by DuckDB directly
  - Instead, pull the files locally with Pyodide first
  - DuckDB-Wasm has custom enhancements to make this possible, but these are not present in DuckDB's Python client
- No runtime-loaded extensions
  - Several extensions are automatically included: `parquet`, `json`, `icu`, `tpcds`, and `tpch`.
- Release cadence aligned with Pyodide
  - At the time of writing, duckdb-pyodide is at 1.0.0 rather than 1.1.1

Conclusion

Pyodide is now the fastest way to use Python and DuckDB together! It is also an approach that scales to an arbitrary number of users because Pyodide's computations happen entirely locally.

We have seen how to embed Pyodide in a static site in multiple ways, as well as how to read remote files.

If you are excited about DuckDB in Pyodide, feel free to join us on Discord. We have a `#show-and-tell` channel where you can share what you build with the community. You are also welcome to explore the duckdb-pyodide repo and report any issues you find. We would also really love some help with enabling runtime-loaded extensions – please reach out if you can help!

Happy quacking about!

DuckDB User Survey Analysis

Publication date: 2024-10-04

Author: Gabor Szarnyas

TL;DR: We share the findings from a survey of 500+ DuckDB users.

Earlier this year, we conducted a survey in the DuckDB community. We were mostly curious about the following topics:

1. How do people use DuckDB?
2. Where do people use DuckDB?
3. What do they like about DuckDB?
4. What improvements would they like to see in future releases?

The survey was open for about three weeks. More than 500 people submitted their answers, and we raffled 20 t-shirts and hoodies among the participants.

Summary

We summarize the key findings of the survey below:

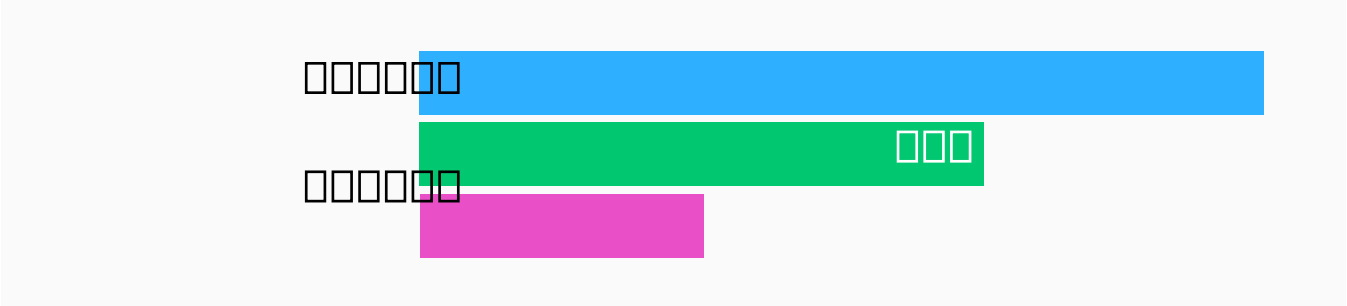
- Users run DuckDB most often on laptops but servers are also very popular.
- The most popular clients are the Python API and the standalone CLI client.
- Most users don't have huge data sets but they appreciate high performance very much.
- Users would like to see performance optimizations related to time series and partitioned data.
- DuckDB is popular among data engineers, analysts and scientists, and also among software engineers.

Let's dive into the details!

Using DuckDB

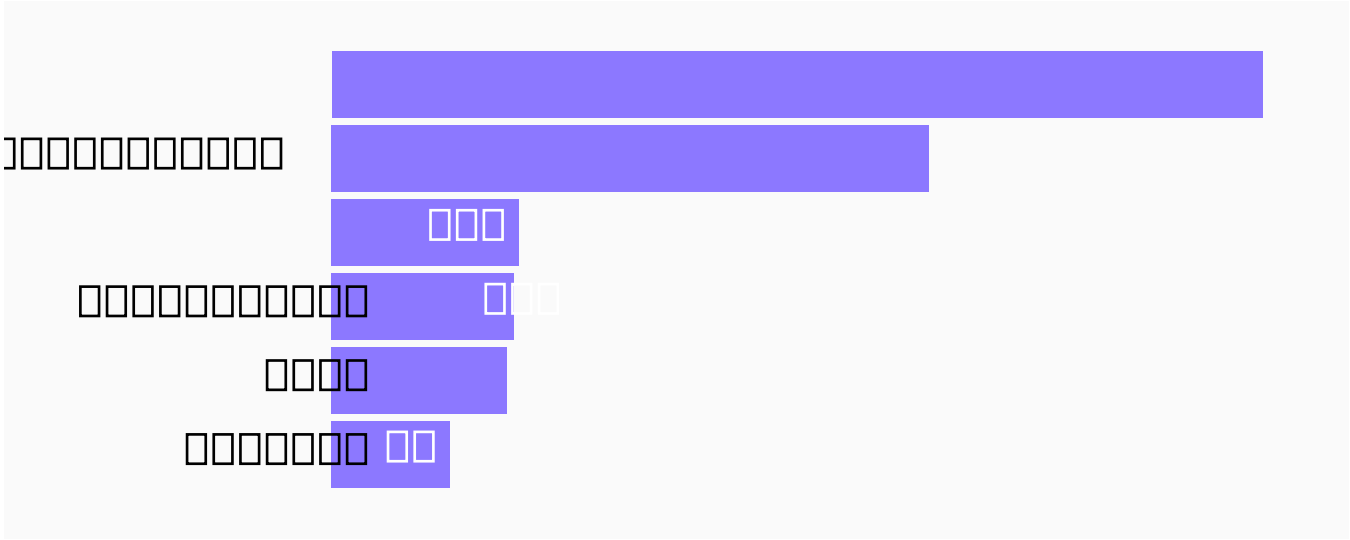
Environments

We asked users about the environment where DuckDB is deployed and found that most of them, 87%, run DuckDB on their laptops. This is in line with the vision that originally drove the creation of DuckDB: creating a system that harnesses the power of hardware available in modern end-user devices. 29% run DuckDB on desktop workstations, and 58% run it on servers (see the breakdown later in the “Server Types” section).



Clients

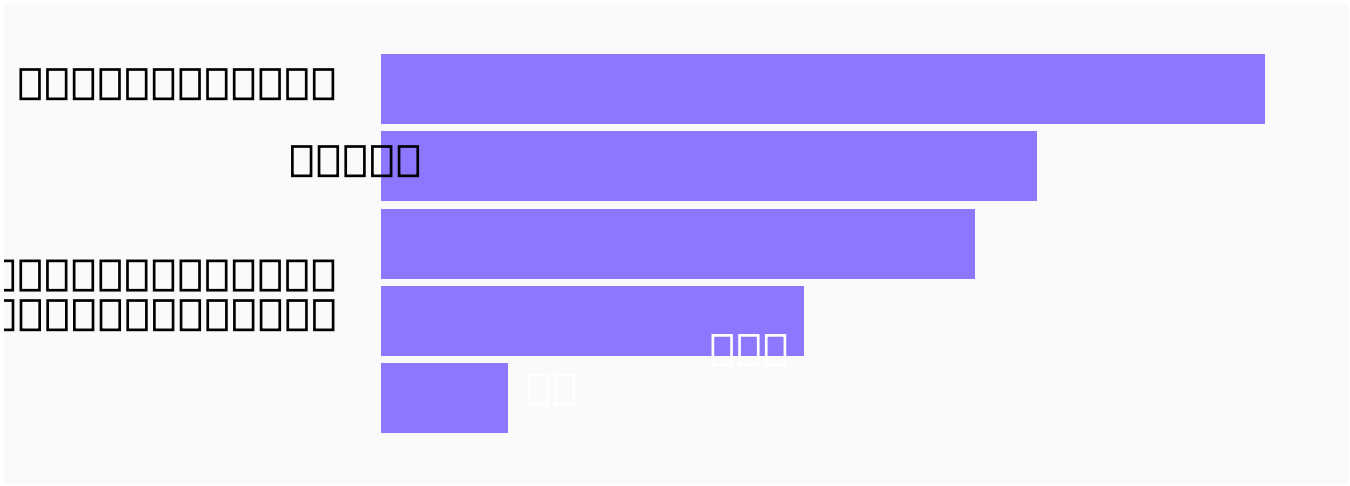
Unsurprisingly, DuckDB is most often used from Python (73%), followed by the standalone command-line application (47%). The third spot is hotly contested with R, WebAssembly (!) and Java all achieving around 14%, followed by Node.js (Javascript) at 9%



The next few places, with 6-7% each, are occupied by ODBC, Rust, and Go. Finally, Arrow (ADBC) rounds off the top 10 with 5%.

Operating Systems

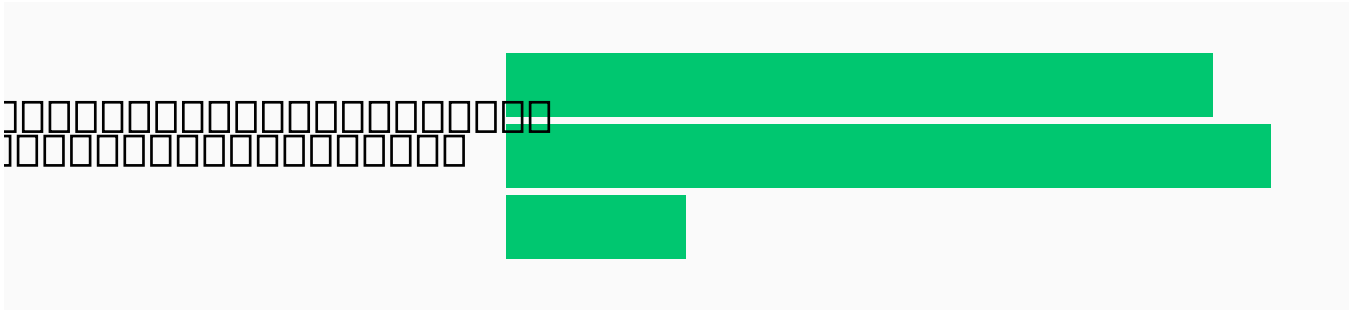
We found that most users, 61%, run DuckDB on Linux servers. These deployments include cloud instances, on-premises installations, and continuous integration (CI) runners. Windows desktop and macOS have a similar share of users, 41–45%. A further 9% run DuckDB on Windows servers.



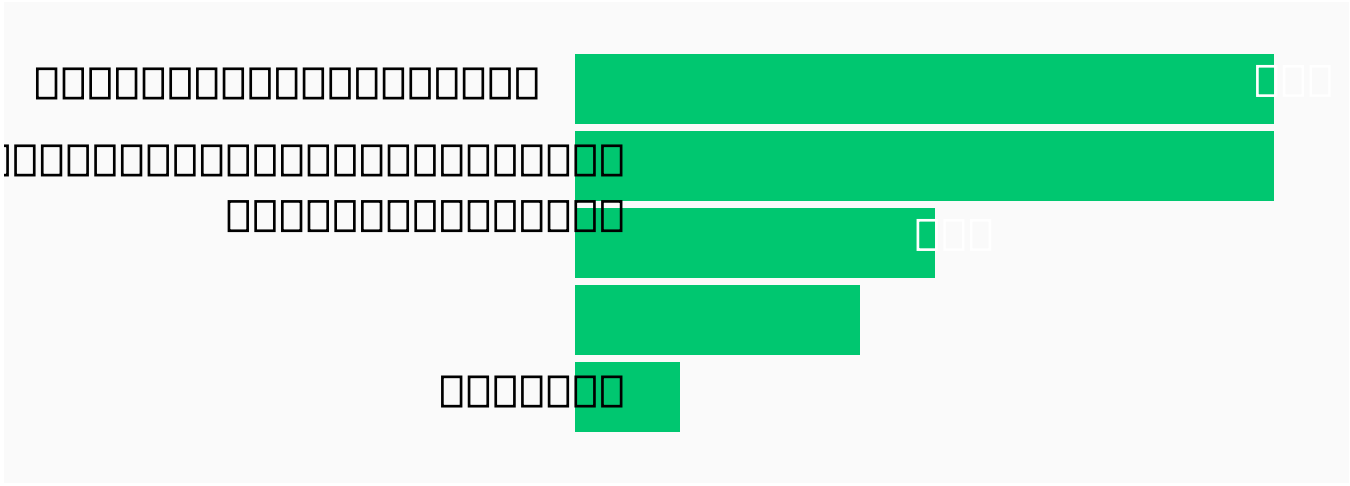
We found the number of Linux desktop users quite striking. While the overall market share of Linux desktop is around 4.5%, *29% of respondents indicated that they run DuckDB on Linux desktop!* We suspect that this is thanks to DuckDB's popularity among data engineers, who often use Linux desktop due to its customizability and similarity to the Linux server-based deployment environments.

Server Types

As we discussed in the "Environments" section, DuckDB is often run on servers. But how big are these servers, and where are they operated? Both small servers (less than 16 GB of memory) and medium-sized servers (16-512 GB of memory) are popular, with 56% and 61% of users reporting that they run DuckDB on these. About 14% of respondents run DuckDB on servers with more than 0.5 TB of memory.



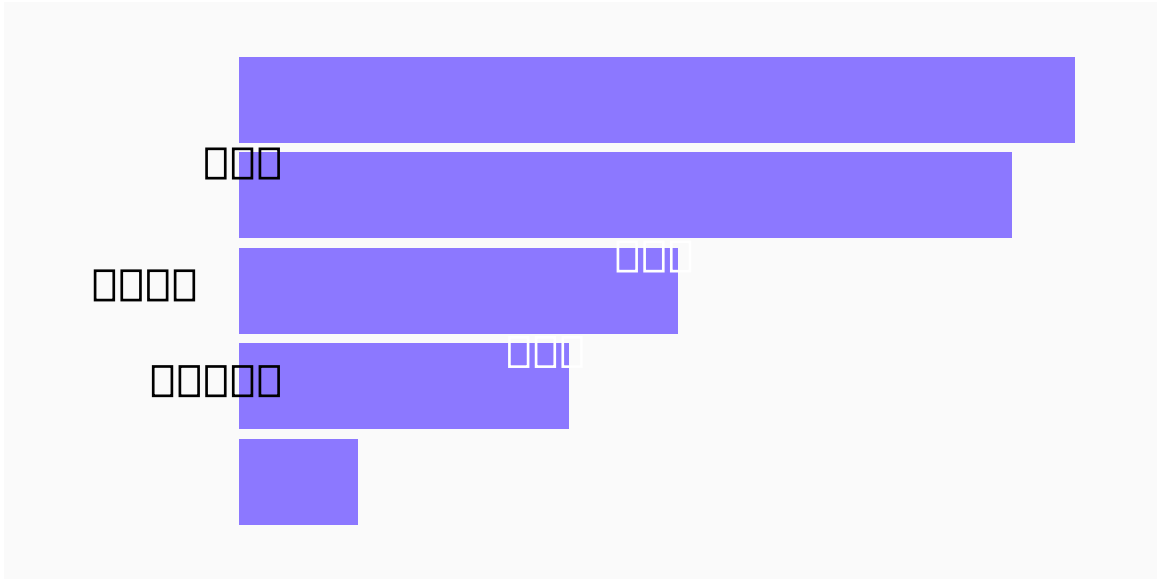
Regarding *where* the servers run, on-premises deployments and AWS are neck-and-neck with 27%. They are followed by two other clouds, Microsoft Azure and the Google Cloud Platform. Finally, about 4% of users run DuckDB on Hetzner servers.



Data

Data Formats

We inquired about the data formats used when working with DuckDB. Parquet is the most popular format: 79% of users reporting to use it. CSV is a close second with 73%. JSON is also popular with vanilla JSON achieving 42% and NDJSON 11%. About ⅓ reported to use Arrow.



Dataset Sizes

We asked users about the size of the largest dataset they processed with DuckDB. We defined *dataset size* as the size of the data when stored in uncompressed CSV format. For Parquet files and DuckDB database files, we asked users to approximate the CSV size by multiplying their file sizes by 5.

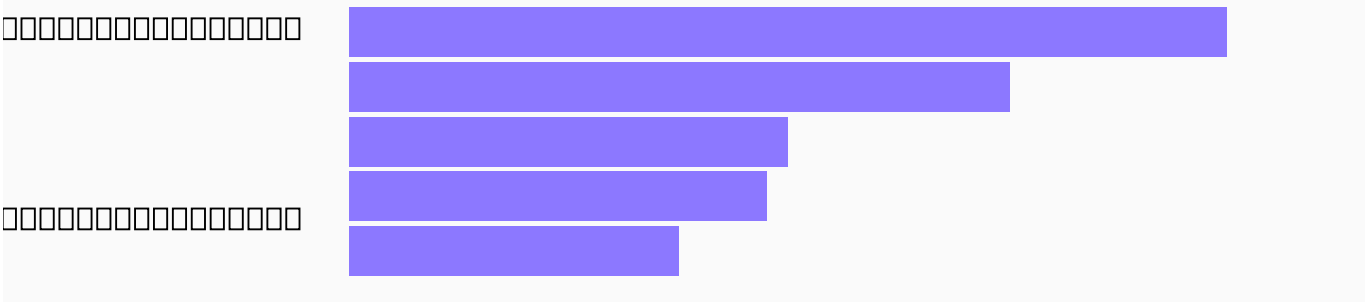
The responses showed that only a few respondents use DuckDB to process Big Data. For ¾ of users, their largest dataset size was less than 100 GB data, 20% of users processed a dataset between 100 GB and 1 TB, and approximately 5% of the users ventured into the 1 TB+ territory. About 1% processed 10 TB+ datasets. These findings are in line with statistics derived from a recent RedShift usage dataset by Jordan Tigani of MotherDuck, and the recent analysis of the Snowflake and RedShift datasets by George Fraser of Fivetran.

While these results obviously are somewhat biased – users who need to crunch through huge datasets may not work with DuckDB (yet!) –, the skew towards smaller datasets is quite significant and shows that many real-world use cases can be tackled using small to medium-sized datasets. The results also show that DuckDB *can* solve many problems on datasets larger than 1 TB.

Features

Most Liked Features

We were curious: what do users like most about DuckDB? The plot shows the most frequent responses:



The most liked feature is **high performance**. Users also enjoy **file format support** (CSV, Parquet, JSON, etc.), **ease of use**, **extensive SQL support** (including friendly SQL) and **in-memory integrations** such as support for Pandas, Arrow and NumPy. Finally, users mentioned low memory usage, protocol support (e.g., HTTPS, S3), database integrations, and portability.

Feature Requests

We asked users about the features that they'd most like to see in future DuckDB versions. The most popular requests are listed in the table below:

Feature	Percentage
Improved partitioning and optimizations related to partitioning	39%
Improved support for time series and optimizations for pre-sorted data	35%
Support for materialized views	28%
Support for vector search	24%
Support for attaching to database systems via ODBC	24%
Support for time travel queries (query the database as of a given time)	23%
Support for the Delta Lake format	22%
Improved support for Iceberg (including writes)	17%

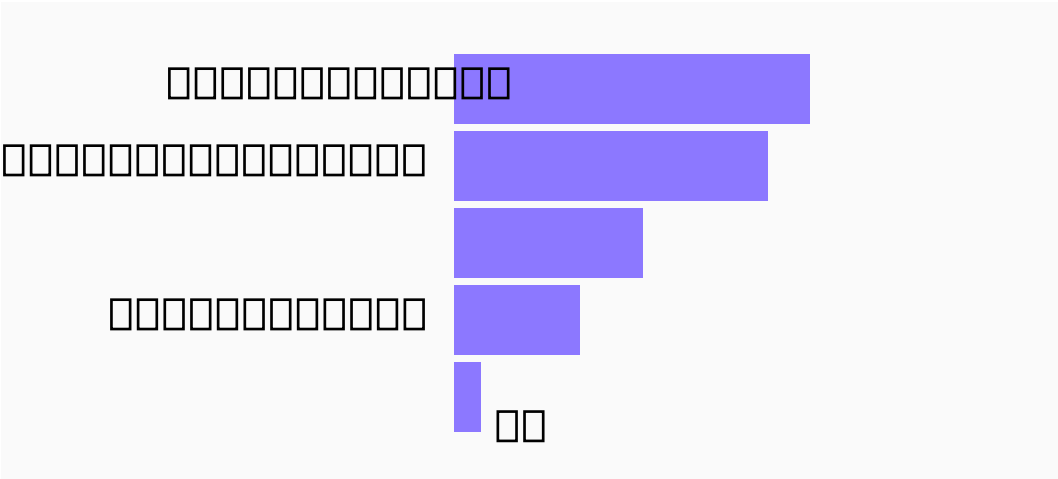
We are happy to report that, since the survey was conducted pre-v1.0.0 and DuckDB is now at version 1.1.1, some of these requests are already a reality:

- Reading Delta Lake is now possible via the `delta` extension.
- Vector search is now supported via the `vss` extension.

For the rest of the requested features, several ones are in the making at DuckDB Labs. Stay tuned!

User Roles

We asked respondents to indicate their main roles in their organization. The top-5 answers were as follows:



It's no surprise that DuckDB is popular in the “data” roles: 26% of the respondents are data engineers, 14% are data scientists, and 9% are data analysts. The form had a surprisingly high share of software engineers, 23%. Finally, about 2% of respondents indicated that their primary role is DBA.

Conclusion

We would like to thank all participants for taking the time to complete the survey. We will use the answers to guide the future development of DuckDB, and we hope that readers of this analysis find it informative as well.

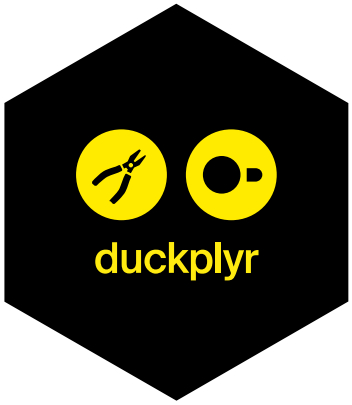
Analyzing Open Government Data with duckplyr

Publication date: 2024-10-09

Author: Hannes Mühleisen

TL;DR: We use the duckplyr R library to clean and analyze an Open Data set published by the government of New Zealand.

> For the duckplyr documentation, visit duckplyr.tidyverse.org.



Wrangling data by throwing SQL strings at it is not the most ergonomic way to perform interactive data analysis in R. For a while now, we have been working with the dplyr project team at Posit (formerly RStudio) and Kirill Müller to develop *duckplyr*. duckplyr is a high-performance drop-in replacement for dplyr, powered by DuckDB. You can read more about duckplyr in the announcement blog post. In this post, we are going to walk through a challenging real-world use case with duckplyr. For those of you wishing to follow along, we have prepared a Google Colab notebook with all the code snippets in this post. Timings reported below are also from Colab.

Like many government statistics agencies, New Zealand's "Stats NZ Tatauranga Aotearoa" thankfully provides some of the datasets they maintain as Open Data for download. The largest file available for download on that page contains "Age and sex by ethnic group (grouped total responses), for census usually resident population counts, 2006, 2013, and 2018 Censuses", CSV zipped file.

We can download that file (mirrored from our CDN, we don't want to DDoS poor Stats NZ) and unzip like so:

```
download.file("https://blobs.duckdb.org/nzcensus.zip", "nzcensus.zip")
unzip("nzcensus.zip")
```

Let's explore the CSV files in the zip and what their sizes are:

```
file.info(Sys.glob("*.csv"))["size"]
```

```
                              size
Data8277.csv             857672667
DimenLookupAge8277.csv        2720
DimenLookupArea8277.csv      65400
DimenLookupEthnic8277.csv      272
DimenLookupSex8277.csv          74
DimenLookupYear8277.csv         67
```

As we can see, there is one large (~800 MB) `Data` file and a bunch of `Dimen...` dimension files. This is a fairly common data layout, sometimes called a "star schema". From this, it's clear there are some joins in our future. But first lets focus on the main file, `Data8277.csv`. Reading sizeable CSV files is not trivial and can be very frustrating. But enough whinging, as the Kiwis would say.

To start with, let's just have a quick look what the file looks like:

```
cat(paste(readLines("Data8277.csv", n=10), collapse="\n"))
```

```
Year,Age,Ethnic,Sex,Area,count
2018,000,1,1,01,795
2018,000,1,1,02,5067
2018,000,1,1,03,2229
2018,000,1,1,04,1356
2018,000,1,1,05,180
2018,000,1,1,06,738
2018,000,1,1,07,630
2018,000,1,1,08,1188
2018,000,1,1,09,2157
```

So far this looks rather tame, there seem to be five columns. Thankfully, they have names. From just eyeballing the column values, it looks like they are all numeric and even integer values. However, looks can be deceiving, and the columns `Age`, `Area`, `count` contain character values somewhere down the line. Fun fact: we have to wait till line 431 741 until the `Area` column contains a non-integer value. Clearly we need a good CSV parser. R has no shortage of CSV readers, for example the `readr` package contains a flexible CSV parser. Reading this file with `readr` takes about a minute (on Colab).

But let's now start using DuckDB and duckplyr. First, we install duckplyr (and DuckDB, which is a dependency):

```
install.packages("duckplyr")
duckdb:::sql("SELECT version()")
```

This command prints out the installed DuckDB version, as of this writing the latest version on CRAN is 1.1.0. We can now use DuckDB's advanced data wrangling capabilities. First off, DuckDB contains probably the world's most advanced CSV parser. For the extra curious, here is a presentation on DuckDB's CSV parser. We use DuckDB's CSV reader to only read the first 10 rows from the CSV file:

```
duckdb:::sql("FROM Data8277.csv LIMIT 10")
```

```
   Year Age Ethnic Sex Area count
1  2018 000      1   1   01   795
2  2018 000      1   1   02  5067
3  2018 000      1   1   03  2229
4  2018 000      1   1   04  1356
5  2018 000      1   1   05   180
6  2018 000      1   1   06   738
7  2018 000      1   1   07   630
8  2018 000      1   1   08  1188
9  2018 000      1   1   09  2157
10 2018 000      1   1   12   177
```

This only takes a few milliseconds because DuckDB's CSV reader produces results in a streaming fashion, and because we have only requested 10 rows we are done fairly quickly.

DuckDB can also print out the schema it detected from the CSV file using the `DESCRIBE` keyword:

```
duckdb:::sql("DESCRIBE FROM Data8277.csv")
```

```
  column_name column_type ...
1        Year      BIGINT ...
2         Age     VARCHAR ...
3      Ethnic      BIGINT ...
4         Sex      BIGINT ...
5        Area     VARCHAR ...
6       count     VARCHAR ...
```

We can see that we have correctly detected the various data types for the columns. We can use the `SUMMARIZE` keyword to compute various summary statistics for all the columns in the file:

```
duckdb:::sql("SUMMARIZE FROM Data8277.csv")
```

This will take a little bit longer, but the results are very interesting:

```
# A tibble: 6 × 12
  column_name column_type min   max     approx_unique avg       std    q25   q50
  <chr>       <chr>       <chr> <chr>           <dbl> <chr>     <chr>  <chr> <chr>
1 Year        BIGINT      2006  2018                3 2012.33… 4.92…  2006  2013
2 Age         VARCHAR     000   999999            149 NA        NA     NA    NA
3 Ethnic      BIGINT      1     9999               11 930.545… 2867…  3     6
4 Sex         BIGINT      1     9                   3 4.0       3.55…  1     2
5 Area        VARCHAR     001   DHB9999          2048 NA        NA     NA    NA
6 count       VARCHAR     ..C   9999            16825 NA        NA     NA    NA
# ℹ 3 more variables: q75 <chr>, count <dbl>, null_percentage <dbl>
```

This will show again the column names and their types, but also the summary statistics for minimum and maximum value, approximate count of unique values, average, standard deviations, 25, 50, and 75 quantiles, and percentage of NULL/NA values. So one gets a pretty good overview of what the data is like.

But we're not here to ogle summary statistics, we want to do actual analysis of the data. In this use case, we would like to compute the number of non-Europeans between 20 and 40 that live in the Auckland area using the 2018 census data and the results should be grouped by sex. To do so, we need to join the dimension CSV files with the main data file in order to properly filter the dimension values. In SQL, the lingua franca of large-scale data analysis, this looks like this:

We first join everything together:

```
FROM 'Data8277.csv' data
JOIN 'DimenLookupAge8277.csv' age ON data.Age = age.Code
JOIN 'DimenLookupArea8277.csv' area ON data.Area = area.Code
JOIN 'DimenLookupEthnic8277.csv' ethnic ON data.Ethnic = ethnic.Code
JOIN 'DimenLookupSex8277.csv' sex ON data.Sex = sex.Code
JOIN 'DimenLookupYear8277.csv' year ON data.Year = year.Code
```

Next, we use the `SELECT` projection to perform some basic renames and data cleaning:

```
SELECT
    year.Description AS year_,
    area.Description AS area_,
    ethnic.Description AS ethnic_,
    sex.Description AS sex_,
    TRY_CAST(replace(age.Description, ' years', '') AS INTEGER) AS age_,
    TRY_CAST(data.count AS INTEGER) AS count_
```

The data set contains various totals, so we remove them before proceeding:

```
WHERE count_ > 0
  AND age_ IS NOT NULL
  AND area_ NOT LIKE 'Total%'
  AND ethnic_ NOT LIKE 'Total%'
  AND sex_ NOT LIKE 'Total%'
```

We wrap the previous statements as a common-table-expression `expanded_cleaned_data`, and we can then compute the actual aggregation using DuckDB

```
SELECT sex_, sum(count_) AS group_count
FROM expanded_cleaned_data
WHERE age_ BETWEEN 20 AND 40
  AND area_ LIKE 'Auckland%'
  AND ethnic_ <> 'European'
  AND year_ = 2018
GROUP BY sex_
ORDER BY sex_
```

This takes ca. 20 s on the limited Colab free tier compute. The result is:

```
    sex_ group_count
1 Female      398556
2   Male      397326
```

So far, so good. However, writing SQL queries is not for everyone. The ergonomics of creating SQL strings in an interactive data analysis environment like R are questionable to say the least. Frameworks like `dplyr` have shown how data wrangling ergonomics can be massively improved. Let's express our analysis using dplyr then after first reading the data into RAM from CSV:

```
library(dplyr)

data   <- readr::read_csv("Data8277.csv")
age    <- readr::read_csv("DimenLookupAge8277.csv")
area   <- readr::read_csv("DimenLookupArea8277.csv")
ethnic <- readr::read_csv("DimenLookupEthnic8277.csv")
sex    <- readr::read_csv("DimenLookupSex8277.csv")
year   <- readr::read_csv("DimenLookupYear8277.csv")

expanded_cleaned_data <- data |>
  filter(grepl("^\\d+$", count)) |>
  mutate(count_ = as.integer(count)) |>
  filter(count_ > 0) |>
  inner_join(
    age |>
      filter(grepl("^\\d+ years$", Description)) |>
      mutate(age_ = as.integer(Code)),
    join_by(Age == Code)
  ) |>
  inner_join(area |>
    mutate(area_ = Description) |>
    filter(!grepl("^Total", area_)), join_by(Area == Code)) |>
  inner_join(ethnic |>
    mutate(ethnic_ = Description) |>
    filter(!grepl("^Total", ethnic_)), join_by(Ethnic == Code)) |>
  inner_join(sex |>
    mutate(sex_ = Description) |>
    filter(!grepl("^Total", sex_)), join_by(Sex == Code)) |>
  inner_join(year |> mutate(year_ = Description), join_by(Year == Code))

# create final aggregation, still completely lazily
twenty_till_fourty_non_european_in_auckland_area <-
  expanded_cleaned_data |>
  filter(
    age_ >= 20, age_ <= 40,
    grepl("^Auckland", area_),
    year_ == "2018",
    ethnic_ != "European"
  ) |>
  summarise(group_count = sum(count_), .by = sex_) |> arrange(sex_)

print(twenty_till_fourty_non_european_in_auckland_area)
```

This looks nicer and completes in ca. one minute, but there are several hidden issues. First, we read the *entire* dataset into RAM. While for this dataset this is likely possible because most computers have more than 1 GB of RAM, this will of course not work for larger datasets. Then, we execute a series of dplyr verbs. However, dplyr executes those eagerly, meaning it does not holistically optimize the sequence of verbs. For example, it cannot see that we are filtering out all non-European ethnicities in the last step and happily computes all of those for the intermediate result. The same happens with survey years that are not 2018, only in the last step we filter those out. We have computed an expensive join on all other years for nothing. Depending on data distributions, this can be extremely wasteful. And yes, it is possible to manually move the filters around but this is tedious and error-prone. At least the result is exactly the same as the SQL version above:

```
# A tibble: 2 × 2
  sex_   group_count
  <chr>        <int>
1 Female      398556
2 Male        397326
```

Now we switch the exact same script over to duckplyr. Instead of reading the CSV files into RAM entirely using `readr`, we instead use the `duckplyr_df_from_csv` function from `duckplyr`:

```
library("duckplyr")

data   <- duckplyr_df_from_csv("Data8277.csv")
age    <- duckplyr_df_from_csv("DimenLookupAge8277.csv")
area   <- duckplyr_df_from_csv("DimenLookupArea8277.csv")
ethnic <- duckplyr_df_from_csv("DimenLookupEthnic8277.csv")
sex    <- duckplyr_df_from_csv("DimenLookupSex8277.csv")
year   <- duckplyr_df_from_csv("DimenLookupYear8277.csv")
```

This takes exactly 0 seconds, because duckplyr is not actually doing much. We detect the schema of the CSV files using our award-winning "sniffer", and create the six placeholder objects for each of those files. Part of the unique design of duckplyr is that those objects are "Heisenbergian", they behave like completely normal R `data.frame`s once they are treated as such, but they can *also* act as lazy evaluation placeholders when they are passed to downstream analysis steps. This is made possible by a little-known R feature known as `ALTREP` which allows R vectors to be computed on-demand among other things.

Now we re-run the exact same dplyr pipeline as above. Only this time we are "done" in less than a second. This is because all we have done is *lazily* constructing a so-called relation tree which encapsulates the entirety of the transformations. This allows *holistic* optimization, for example pushing the year and ethnicity all the way down to the reading of the CSV file *before* joining. We can also eliminate the reading of columns that are not used in the query at all.

Only when we finally print the result

```
print(twenty_till_fourty_non_european_in_auckland_area)
```

actual computation is triggered. This finishes in the same time as the hand-rolled SQL query above, only that this time we had a much more pleasant experience from using the dplyr syntax. And, thankfully, the result is still exactly the same.

This use case was also presented as part of my keynote at this year's posit::conf:

Finally, we should note that duckplyr is still being developed. We have taking great care in not breaking anything and will fall back on the existing dplyr implementation if anything cannot be run in DuckDB (yet). But we would love to hear from you if anything does not work as expected.

DuckDB Tricks – Part 2

Publication date: 2024-10-11

Author: Gabor Szarnyas

TL;DR: We continue our "DuckDB tricks" series, focusing on queries that clean, transform and summarize data.

Overview

This post is the latest installment of the DuckDB Tricks series, where we show you nifty SQL tricks in DuckDB. Here's a summary of what we're going to cover:

Operation	SQL instructions
Fixing timestamps in CSV files	`regexp_replace()`{:.language-sql .highlight} and `strptime()`{:.language-sql .highlight}
Filling in missing values	`CROSS JOIN`{:.language-sql .highlight}, `LEFT JOIN`{:.language-sql .highlight} and `coalesce()`{:.language-sql .highlight}
Repeated transformation steps	`CREATE OR REPLACE TABLE t AS ... FROM t ...`{:.language-sql .highlight}
Computing checksums for columns	`bit_xor(md5_number(COLUMNS(*)::VARCHAR))`{:.language-sql .highlight}
Creating a macro for checksum	`CREATE MACRO checksum(tbl) AS TABLE ...`{:.language-sql .highlight}

Dataset

For our example dataset, we'll use `schedule.csv`, a hand-written CSV file that encodes a conference schedule. The schedule contains the timeslots, the locations and the events scheduled.

```
timeslot,location,event
2024-10-10 9am,room Mallard,Keynote
2024-10-10 10.30am,room Mallard,Customer stories
2024-10-10 10.30am,room Fusca,Deep dive 1
2024-10-10 12.30pm,main hall,Lunch
2024-10-10 2pm,room Fusca,Deep dive 2
```

Fixing Timestamps in CSV Files

As usual in real use case, the input CSV is messy with irregular timestamps such as `2024-10-10 9am`. Therefore, if we load the `schedule.csv` file using DuckDB's CSV reader, the CSV sniffer will detect the first column as a `VARCHAR` field:

```
CREATE TABLE schedule_raw AS
    SELECT * FROM 'https://duckdb.org/data/schedule.csv';

SELECT * FROM schedule_raw;
```

timeslot varchar	location varchar	event varchar
2024-10-10 9am	room Mallard	Keynote
2024-10-10 10.30am	room Mallard	Customer stories
2024-10-10 10.30am	room Fusca	Deep dive 1
2024-10-10 12.30pm	main hall	Lunch
2024-10-10 2pm	room Fusca	Deep dive 2

Ideally, we would like the `timeslot` column to have the type `TIMESTAMP` so we can treat it as a timestamp in the queries later. To achieve this, we can use the table we just loaded and fix the problematic entities by using a regular expression-based search and replace operation, which unifies the format to `hours.minutes` followed by `am` or `pm`. Then, we convert the string to timestamps using `strptime` with the `%p` format specifier capturing the `am`/`pm` part of the string.

```
CREATE TABLE schedule_cleaned AS
    SELECT
        timeslot
            .regexp_replace(' (\d+)(am|pm)$', ' \1.00\2')
            .strptime('%Y-%m-%d %H.%M%p') AS timeslot,
        location,
        event
    FROM schedule_raw;
```

Note that we use the dot operator for function chaining to improve readability. For example, `regexp_replace(string, pattern, replacement)` is formulated as `string.regexp_replace(pattern, replacement)`. The result is the following table:

timeslot timestamp	location varchar	event varchar
2024-10-10 09:00:00	room Mallard	Keynote
2024-10-10 10:30:00	room Mallard	Customer stories
2024-10-10 10:30:00	room Fusca	Deep dive 1
2024-10-10 12:30:00	main hall	Lunch
2024-10-10 14:00:00	room Fusca	Deep dive 2

Filling in Missing Values

Next, we would like to derive a schedule that includes the full picture: *every timeslot* for *every location* should have its line in the table. For the timeslot-location combinations, where there is no event specified, we would like to explicitly add a string that says `<empty>`.

To achieve this, we first create a table `timeslot_location_combinations` containing all possible combinations using a `CROSS JOIN`. Then, we can connect the original table on the combinations using a `LEFT JOIN`. Finally, we replace `NULL` values with the `<empty>` string using the `coalesce` function.

> The `CROSS JOIN` clause is equivalent to simply listing the tables in the `FROM` clause without specifying join conditions. By explicitly spelling out `CROSS JOIN`, we communicate that we intend to compute a Cartesian product – which is an expensive operation on large tables and should be avoided in most use cases.

```
CREATE TABLE timeslot_location_combinations AS
    SELECT timeslot, location
```

```
    FROM (SELECT DISTINCT timeslot FROM schedule_cleaned)
    CROSS JOIN (SELECT DISTINCT location FROM schedule_cleaned);

CREATE TABLE schedule_filled AS
    SELECT timeslot, location, coalesce(event, '<empty>') AS event
    FROM timeslot_location_combinations
    LEFT JOIN schedule_cleaned
        USING (timeslot, location)
    ORDER BY ALL;

SELECT * FROM schedule_filled;
```

timeslot timestamp	location varchar	event varchar
2024-10-10 09:00:00	main hall	<empty>
2024-10-10 09:00:00	room Fusca	<empty>
2024-10-10 09:00:00	room Mallard	Keynote
2024-10-10 10:30:00	main hall	<empty>
2024-10-10 10:30:00	room Fusca	Deep dive 1
2024-10-10 10:30:00	room Mallard	Customer stories
2024-10-10 12:30:00	main hall	Lunch
2024-10-10 12:30:00	room Fusca	<empty>
2024-10-10 12:30:00	room Mallard	<empty>
2024-10-10 14:00:00	main hall	<empty>
2024-10-10 14:00:00	room Fusca	Deep dive 2
2024-10-10 14:00:00	room Mallard	<empty>
12 rows		3 columns

We can also put everything together in a single query using a WITH clause:

```
WITH timeslot_location_combinations AS (
    SELECT timeslot, location
    FROM (SELECT DISTINCT timeslot FROM schedule_cleaned)
    CROSS JOIN (SELECT DISTINCT location FROM schedule_cleaned)
)
SELECT timeslot, location, coalesce(event, '<empty>') AS event
FROM timeslot_location_combinations
LEFT JOIN schedule_cleaned
    USING (timeslot, location)
ORDER BY ALL;
```

Repeated Data Transformation Steps

Data cleaning and transformation usually happens as a sequence of transformations that shape the data into a form that's best fitted to later analysis. These transformations are often done by defining newer and newer tables using CREATE TABLE ... AS SELECT statements.

For example, in the sections above, we created `schedule_raw`, `schedule_cleaned`, and `schedule_filled`. If, for some reason, we want to skip the cleaning steps for the timestamps, we have to reformulate the query computing `schedule_filled` to use `schedule_raw` instead of `schedule_cleaned`. This can be tedious and error-prone, and it results in a lot of unused temporary data – data that may accidentally get picked up by queries that we forgot to update!

In interactive analysis, it's often better to use the same table name by running CREATE OR REPLACE statements:

```
CREATE OR REPLACE TABLE <table_name> AS
```

```
    ...
    FROM <table_name>
    ...;
```

Using this trick, we can run our analysis as follows:

```
CREATE OR REPLACE TABLE schedule AS
    SELECT * FROM 'https://duckdb.org/data/schedule.csv';

CREATE OR REPLACE TABLE schedule AS
    SELECT
        timeslot
            .regexp_replace(' (\d+)(am|pm)$', ' \1.00\2')
            .strptime('%Y-%m-%d %H.%M%p') AS timeslot,
        location,
        event
    FROM schedule;

CREATE OR REPLACE TABLE schedule AS
    WITH timeslot_location_combinations AS (
        SELECT timeslot, location
        FROM (SELECT DISTINCT timeslot FROM schedule)
        CROSS JOIN (SELECT DISTINCT location FROM schedule)
    )
    SELECT timeslot, location, coalesce(event, '<empty>') AS event
    FROM timeslot_location_combinations
    LEFT JOIN schedule
        USING (timeslot, location)
    ORDER BY ALL;

SELECT * FROM schedule;
```

Using this approach, we can skip any step and continue the analysis without adjusting the next one.

What's more, our script can now be re-run from the beginning without explicitly deleting any tables: the `CREATE OR REPLACE` statements will automatically replace any existing tables.

Computing Checksums for Columns

It's often beneficial to compute a checksum for each column in a table, e.g., to see whether a column's content has changed between two operations. We can compute a checksum for the `schedule` table as follows:

```
SELECT bit_xor(md5_number(COLUMNS(*)::VARCHAR))
FROM schedule;
```

What's going on here? We first list columns (`COLUMNS(*)`) and cast all of them to VARCHAR values. Then, we compute the numeric MD5 hashes with the `md5_number` function and aggregate them using the `bit_xor` aggregate function. This produces a single `HUGEINT` (`INT128`) value per column that can be used to compare the content of tables.

If we run this query in the script above, we get the following results:

```
┌──────────────────────────────────────────┬──────────────────────────────────────────┬──────────────────────────────────────────┐
│                 timeslot                 │                 location                 │                  event                   │
│                  int128                  │                  int128                  │                  int128                  │
├──────────────────────────────────────────┼──────────────────────────────────────────┼──────────────────────────────────────────┤
│ -134063647976146309049043791223896883700 │ 851812273645607500489714593303929888 15 │ -650144045653398519678796832146127680 44 │
└──────────────────────────────────────────┴──────────────────────────────────────────┴──────────────────────────────────────────┘
```

timeslot	location	event
int128	int128	int128
629010110167473189774697778517845645961	85181227364560750048971459330392988815	-650144045653398519678796832146127680 44

timeslot int128	location int128	event int128
-162418013182718436871288818115274808663	0	-135609337521255080720676586176293337793

Creating a Macro for the Checksum Query

We can turn the checksum query into a table macro with the new `query_table` function:

```
CREATE MACRO checksum(table_name) AS TABLE
    SELECT bit_xor(md5_number(COLUMNS(*)::VARCHAR))
    FROM query_table(table_name);
```

This way, we can simply invoke it on the `schedule` table as follows (also leveraging DuckDB's FROM-first syntax):

```
FROM checksum('schedule');
```

timeslot	location	event
int128	int128	int128
-134063647976146309049043791223896883700	85181227364560750048971459330392988815	-650144045653398519678796832146127680 44

Closing Thoughts

That's it for today! We'll be back soon with more DuckDB tricks and case studies. In the meantime, if you have a trick that would like to share, please share it with the DuckDB team on our social media sites, or submit it to the DuckDB Snippets site (maintained by our friends at MotherDuck).

Driving CSV Performance: Benchmarking DuckDB with the NYC Taxi Dataset

Publication date: 2024-10-16

Author: Pedro Holanda

TL;DR: DuckDB's benchmark suite now includes the NYC Taxi Benchmark. We explain how our CSV reader performs on the Taxi Dataset and provide steps to reproduce the benchmark.

The NYC taxi dataset is a collection of many years of taxi rides that occurred in New York City. It is a very influential dataset, used for database benchmarks, machine learning, data visualization, and more.

In 2022, the data provider has decided to distribute the dataset as a series of Parquet files instead of CSV files. Performance-wise, this is a wise choice, as Parquet files are much smaller than CSV files, and their native columnar format allows for fast execution directly on them. However, this change hinders the number of systems that can natively load the files.

In the "Billion Taxi Rides in Redshift" blog post, a new database benchmark is proposed to evaluate the performance of aggregations over the taxi dataset. The dataset is also joined and denormalized with other datasets that contain information about the weather, cab types, and pickup/dropoff locations. It is then stored as multiple compressed, gzipped CSV files, each containing 20 million rows.

The Taxi Data Set as CSV Files

Since DuckDB is well-known for its CSV reader performance, we were intrigued to explore whether the loading process of this benchmark could help us identify new performance bottlenecks in our CSV loader. This curiosity led us on a journey to generate these datasets and analyze their performance in DuckDB. According to the recent study conducted on the AWS RedShift fleet, CSV files are the most used external source data type in S3, and 99% of them are gzipped. Therefore, the fact that the proposed benchmark also used split gzipped files caught my attention.

In this blog post, we'll guide you through how to run this benchmark in DuckDB and discuss some lessons learned and future ideas for our CSV Reader. The dataset used in this benchmark is publicly available. The dataset is partitioned and distributed as a collection of 65 gzipped CSV files, each containing 20 million rows and totaling up to 1.8 GB per file. The total dataset is 111 GB compressed and 518 GB uncompressed. We also provide more details on how we generated this dataset and highlight the differences between the dataset we distribute and the original one described in the "Billion Taxi Rides in Redshift" blog post.

Reproducing the Benchmark

Doing fair benchmarking is a difficult problem, especially when the data, queries, and results used for the benchmark are not easy to access and run. We have made the benchmark discussed in this blog post easy to run by providing scripts available in the `taxi-benchmark` GitHub repository.

This repository contains three main Python scripts:

1. `generate_prepare_data.py`: Downloads all necessary files and prepares them for the benchmark.
2. `benchmark.py`: Runs the benchmark and performs result verification.
3. `analyse.py`: Analyzes the benchmark results and produces some of the insights discussed in this blog post.

The benchmark is not intended to be flawless – no benchmark is. However, we believe that sharing these scripts is a positive step, and we welcome any contributions to make them cleaner and more efficient.

The repository also includes a README file with detailed instructions on how to use it. This repository will serve as the foundation for the experiments conducted in this blog post.

Preparing the Dataset

To start, you first need to download and prepare the files by executing `python generate_prepare_data.py`. This will download all 65 files to the `./data` folder. Additionally, the files will be uncompressed and combined into a single large file.

As a result, the `./data` folder will have 65 gzipped CSV files (i.e., from `trips_xaa.csv.gz` to `trips_xcm.csv.gz`) and a single large uncompressed CSV file containing the full data (i.e., `decompressed.csv`).

Our benchmark then run in two different settings:

1. Over 65 compressed files.
2. Over a single uncompressed file.

Once the files have been prepared, you can run the benchmark by running `python benchmark.py`.

Loading

The loading phase of the benchmark runs six times for each benchmark setting. From the first five runs, we take the median loading time. During the sixth run, we collect resource usage data (e.g., CPU usage and disk reads/writes).

Loading is performed using an in-memory DuckDB instance, meaning the data is not persisted to DuckDB storage and only exists while the connection is active. This is important to note because, as the dataset does not fit in memory and is spilled into a temporary space on disk. The decision to not persist the data has a substantial impact on performance: it makes loading the dataset significantly faster, while querying it will be somewhat slower as DuckDB will use an uncompressed representation. We made this choice for the benchmark since our primary focus is on testing the CSV loader rather than the queries.

Our table schema is defined in `schema.sql`.

The loader for the 65 files uses the following query:

```
COPY trips FROM 'data/trips_*.csv.gz' (HEADER false);
```

The loader for the single uncompressed file uses this query:

```
COPY trips FROM 'data/decompressed.csv' (HEADER false);
```

Querying

After loading, the benchmark script will run each of the benchmark queries five times to measure their execution time. It is also important to note that the results of the queries are validated against their corresponding answers. This allows us to verify the correctness of the benchmark. Additionally, the queries are identical to those used in the original "Billion Taxi Rides" benchmark.

Results

Loading Time

Although we are talking about many rows of a CSV file with 51 columns, DuckDB can ingest them rather fast.

Note that, by default, DuckDB preserves the insertion order of the data, which negatively impacts performance. In the following results, all datasets have been loaded with this option set to `false`.

```
SET preserve_insertion_order = false;
```

All experiments were run on my Apple M1 Max with 64 GB of RAM, and we compare the loading times for a single uncompressed CSV file, and the 65 compressed CSV files.

Name	Time (min)	Avg deviation of CPU usage from 100%
Single File – Uncompressed	11:52	31.57
Multiple Files – Compressed	13:52	27.13

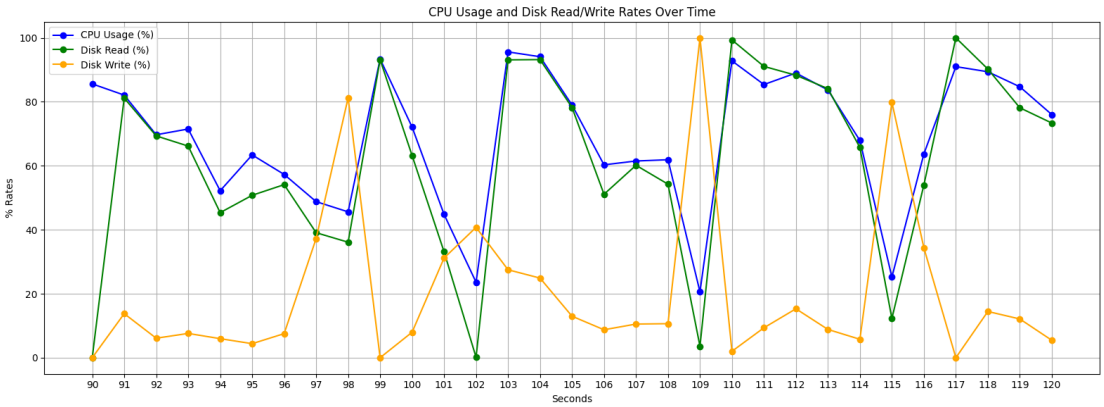
Unsurprisingly, loading data from multiple compressed files is more CPU-efficient than loading from a single uncompressed file. This is evident from the lower average deviation in CPU usage for multiple compressed files, indicating fewer wasted CPU cycles. There are two main reasons for this: (1) The compressed files are approximately eight times smaller than the uncompressed file, drastically reducing the amount of data that needs to be loaded from disk and, consequently, minimizing CPU stalls while waiting for data to be processed. (2) It is much easier to parallelize the loading of multiple files than a single file, as each thread can handle on a single file.

The difference in CPU efficiency is also reflected in execution times: reading from a single uncompressed file is 2 minutes faster than reading from multiple compressed files. The reason for this lies in our decompression algorithm, which is admittedly not optimally designed. Reading a compressed file involves three tasks: (1) loading data from disk into a compressed buffer, (2) decompressing that data into a decompressed buffer, and (3) processing the decompressed buffer. In our current implementation, tasks 1 and 2 are combined into a single operation, meaning we cannot continue reading until the current buffer is fully decompressed, resulting in idle cycles.

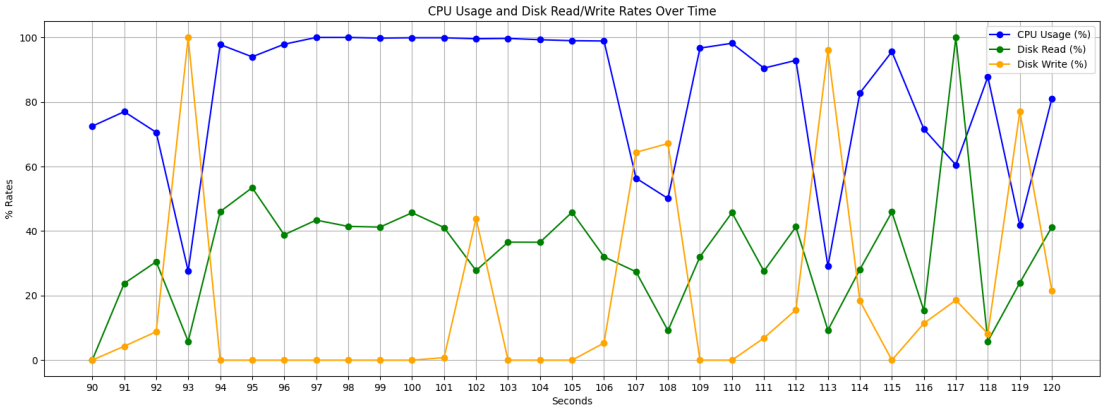
Under the Hood

We can also see what happens under the hood to verify our conclusion regarding the loading time.

In the figure below, you can see a snapshot of CPU and disk utilization for the “Single File – Uncompressed” run. We observe that achieving 100% CPU utilization is challenging, and we frequently experience stalls due to data writes to disk, as we are creating a table from a dataset that does not fit into our memory. Another key point is that CPU utilization is closely tied to disk reads, indicating that our threads often wait for data before processing it. Implementing async IO for the CSV Reader/Writer could significantly improve performance for parallel processing, as a single thread could handle most of our disk I/O without negatively affecting CPU utilization.



Below, you can see a similar snapshot for loading the 65 compressed files. We frequently encounter stalls during data writes; however, CPU utilization is significantly better because we wait less time for the data to load (remember, the data is approximately 8 times smaller than in the uncompressed case). In this scenario, parallelization is also much easier. Like in the uncompressed case, these gaps in CPU utilization could be mitigated by async I/O, with the addition of a decomposed decompression algorithm.



Query Times

For completeness, we also provide the results of the four queries on a MacBook Pro with an M1 Pro CPU. This comparison demonstrates the time differences between querying a database that does not fit in memory using a purely in-memory connection (i.e., without storage) versus one where the data is first loaded and persisted in the database.

Name	Time – without storage (s)	Time – with storage (s)
Q 01	2.45	1.45
Q 02	3.89	0.80
Q 03	5.21	2.20
Q 04	11.2	3.12

The main difference between these times is that when DuckDB uses a storage file, the data is highly compressed, resulting in much faster access when querying the dataset. In contrast, when we do not use persistent storage, our in-memory database temporarily stores data in an uncompressed `.tmp` file to allow for memory overflow, which increases disk I/O and leads to slower query results. This observation raises a potential area for exploration: determining whether applying compression to temporary data would be beneficial.

How This Dataset Was Generated

The original blog post generated the dataset using CSV files distributed by the NYC Taxi and Limousine Commission. Originally, these files included precise latitude and longitude coordinates for pickups and drop-offs. However, starting in mid-2016, these precise coordinates were anonymized using pickup and drop-off geometry objects to address privacy concerns. (There are even stories of broken marriages resulting from checking the actual destinations of taxis.) Furthermore, in recent years, the TLC decided to redistribute the data as Parquet files and to fully anonymize these data points, including data prior to mid-2016.

This is a problem, as the dataset from the "Billion Taxi Rides in Redshift" blog post relies on having this detailed information. Let's take the following snippet of the data:

```
649084905,VTS,2012-08-31 22:00:00,2012-08-31 22:07:00,0,1,-73.993908,40.741383000000006,-
73.989915,40.75273800000001,1,1.32,6.1,0.5,0.5,0,0,0,0,7.1,CSH,0,0101000020E6100000E6CE4C309C7F52C0BA675DA3E55E44
Yards-Chelsea-Flatiron-Union Square,3807,132,109,1,Manhattan,010900,1010900,I,MN17,Midtown-Midtown
South,3807
```

We see precise longitude and latitude data points: `-73.993908`, `40.741383000000006`, `-73.989915`, `40.75273800000001`, along with a PostGIS Geometry hex blob created from this longitude and latitude information: `0101000020E6100000E6CE4C309C7F52C0BA675DA3E55E4440`,

`0101000020E610000078B471C45A7F52C06D3A02B859604440` (generated as `ST_SetSRID(ST_Point(longitude, latitude), 4326)`).

Since this information is essential to the dataset, producing files as described in the "Billion Taxi Rides in Redshift" blog post is no longer feasible due to the missing detailed location data. However, the internet never forgets. Hence, we located instances of the original dataset distributed by various sources, such as [1], [2], and [3]. Using these sources, we combined the original CSV files with weather information from the scripts referenced in the "Billion Taxi Rides in Redshift" blog post.

How Does This Dataset Differ from the Original One?

There are two significant differences between the dataset we distribute and the one from the "Billion Taxi Rides in Redshift" blog post:

1. Our dataset includes data up to the last date that longitude and latitude information was available (June 30, 2016), whereas the original post only included data up to the end of 2015 (understandable, as the post was written in February 2016).
2. We also included Uber trips, which were excluded from the original post.

If you wish to run the benchmark with a dataset as close to the original as possible, you can generate a new table by filtering out the additional data. For example:

```
CREATE TABLE trips_og AS
    FROM trips
    WHERE pickup_datetime < '2016-01-01'
      AND cab_type != 'uber';
```

Conclusion

In this blog post, we discussed how to run the taxi benchmark on DuckDB, and we've made all scripts available so you can benchmark your preferred system as well. We also demonstrated how this highly relevant benchmark can be used to evaluate our operators and gain insights into areas for further improvement.

What's New in the Vector Similarity Search Extension?

Publication date: 2024-10-23

Author: Max Gabrielsson

TL;DR: DuckDB is another step closer to becoming a vector database! In this post, we show the new performance optimizations implemented in the vector search extension.

In the previous blog post, we introduced the DuckDB Vector Similarity Search (VSS) extension. While the extension is still quite experimental, we figured it would be interesting to dive into the details of some of the new features and improvements that we've been working on since the initial release.

Indexing Speed Improvements

As previously documented, creating an HNSW (Hierarchical Navigable Small Worlds) index over an already populated table is much more efficient than first creating the index and then inserting into the table. This is because it is much easier to predict how large the index will be if the total amount of rows are known up-front, which makes its possible to divide the work into chunks large enough to distribute over multiple threads. However, in the initial release this work distribution was a bit too coarse-grained as we would only schedule an additional worker thread for each *row group* (about 120,000 rows by default) in the table.

We've now introduced an extra buffer step in the index creation pipeline which enables more fine-grained work distribution, smarter memory allocation and less contention between worker threads. This results in much higher CPU saturation and a significant speedup when building HNSW indexes in environments with many threads available, regardless of how big or small the underlying table is.

Another bonus of this change is that we can now emit a progress bar when building the index, which is a nice touch when you still need to wait a while for the index creation to finish (despite the now much better use of system resources!).

New Distance Functions

In the initial release of VSS we supported three different distance functions: `array_distance`, `array_cosine_similarity` and `array_inner_product`. However, only the `array_distance` function is actually a *distance* function in that it returns results closer to 0 when the vectors are similar, and close to 1 when they are dissimilar, in contrast to, e.g., `array_cosine_similarity` that returns 1 when the vectors are identical. Oops!

To remedy this we've introduced two new distances:

- `array_cosine_distance`, equivalent to `1 - array_cosine_simililarity`
- `array_negative_inner_productequivalent` to `-array_inner_product`

These will now be accelerated with the use of the HNSW index instead, making the query patterns and ordering consistent for all supported metrics regardless if you make use of the HNSW index or not. Additionally, if you have an HNSW using, e.g., the `cosine` metric and write a top-k style query using `1 - array_cosine_similarity` as the ranking criterium, the optimizer should be able to normalize the expression to `array_cosine_distance` and use the index for this function as well.

For completeness we've also added the equivalent distance functions for the dynamically-sized `LIST` datatype (prefixed with `list_` instead of `array_`) and changed the `<=>` binary operator to now be an alias of `array_cosine_distance`, matching the semantics of the `pgvector` extension for PostgreSQL.

Index Accelerated ”Top-K” Aggregates

Another cool thing that's happened in core DuckDB since last time is that DuckDB now has extra overloads for the `min_by` and `max_by` aggregate functions (and their aliases `arg_min` and `arg_max`) These new overloads take an optional third n argument that specifies the number of top-k (or top-n) elements to keep and outputs them into a sorted `LIST` value. Here's an example:

```
-- Create a table with some example data
CREATE OR REPLACE TABLE vecs AS
    SELECT
        row_number() OVER () AS id,
        [a, b, c]::FLOAT[3] AS vec
    FROM
        range(1,4) AS x(a), range(1,4) AS y(b), range(1,4) AS z(c);

-- Find the top 3 rows with the vector closest to [2, 2, 2]
SELECT
    arg_min(vecs, array_distance(vec, [2, 2, 2]::FLOAT[3]), 3)
FROM
    vecs;
```

```
[{'id': 14, 'vec': [2.0, 2.0, 2.0]}, {'id': 13, 'vec': [2.0, 1.0, 2.0]}, {'id': 11, 'vec': [1.0, 2.0,
2.0]}]
```

Of course, the VSS extension now includes optimizer rules to use to the HNSW index to accelerate these top-k aggregates when the ordering input is a distance function that references an indexed vector column, similarly to the `SELECT a FROM b ORDER BY array_distance(a.vec, query_vec) LIMIT k` query pattern that we discussed in the previous blog post. These new overloads allow you to express the same query in a more concise and readable way, while still avoiding the need for a full scan and sort of the underlying table (as long as the table has a matching HNSW index).

Index Accelerated LATERAL Joins

After running some benchmarking on the initial version of VSS, we realized that even though index-lookups on our HNSW index is really fast (thanks to the USearch library that it is based on!), using DuckDB to search for individual vectors at a time has a lot of latency compared to other solutions. The reasons for this are many and nuanced, but we want to be clear that our choice of HNSW implementation, USearch, is not the bottleneck here as profiling revelead only about 2% of the runtime is actually spent inside of usearch.

Instead, most of the per-query overhead comes from the fact that DuckDB is just not optimized for *point queries,* i.e., queries that only really fetch and process a single row. Because DuckDB is based on a vectorized execution engine, the smallest unit of work is not 1 row but 2,048, and because we expect to crunch through a ton of data, we generally favor spending a lot of time up front to optimize the query plan and pre-allocate large buffers and caches so that everything is as efficient as possible once we start executing. But a lot of this work becomes unneccessary when the actual working set is so small. For example, is it really worthwile to inspect and hash every single element of a constant 768-long query vector to attempt to look for common subexpressions if you know there is only going to be a handful of rows in the result?

While we have some ideas on how to improve this scenario in the future, we decided to take another approach for now and instead try focus not on our weaknesses, but on our strengths. That is, crunching through a ton of data! So instead of trying to optimize the “1:N”, i.e., “given this one embedding, give me the closes N embeddings” query, what if we instead focused on the “N:M”, “given all these N embeddings, pair them up with the closest M embeddings each”. What would that look like? Well, that would be a LATERAL join of course!

Basically, we are now able to make use of HNSW indexes to accelerate `LATERAL` joins where the “inner” query looks just like the top-k style queries we normally target, for example:

```
SELECT a
FROM b
ORDER BY array_distance(a.vec, query_vec)
LIMIT k;
```

But where the `query_vec` array is now a reference to an "outer" join table. The only requirement is for the inner table to have an HNSW index on the vector column matching the distance function. Here's an example:

```
-- Set the random seed for reproducibility
SELECT setseed(0.42);

-- Create some example tables
CREATE TABLE queries AS
    SELECT
        i AS id,
        [random(), random(), random()]::FLOAT[3] AS embedding
    FROM generate_series(1, 10_000) r(i);

CREATE TABLE items AS
    SELECT
        i AS id,
        [random(), random(), random()]::FLOAT[3] AS embedding
FROM generate_series(1, 10_000) r(i);

-- Collect the 5 closest items to each query embedding
SELECT queries.id AS id, list(inner_id) AS matches
    FROM queries, LATERAL (
        SELECT
            items.id AS inner_id,
            array_distance(queries.embedding, items.embedding) AS dist
        FROM items
        ORDER BY dist
        LIMIT 5
    )
GROUP BY queries.id;
```

Executing this on my Apple M3 Pro-equipped MacBook with 36 GB memory takes about 10 seconds.

If we `EXPLAIN` this query plan, we'll see a lot of advanced operators:

```
PRAGMA explain_output = 'optimized_only';
EXPLAIN ...
```

While this plan looks very complicated, the most worrysome among these operators is the `CROSS_PRODUCT` towards the bottom of the plan, which blows up the expected cardinality and is a sign that we are doing a lot of work that we probably don't want to do. However, if we create an HNSW index on the `items` table using

```
CREATE INDEX my_hnsw_idx ON items USING HNSW(embedding);
```

and re-run `EXPLAIN`, we get this plan instead:

We can see that this plan is drastically simplified, but most importantly, the new `HNSW_INDEX_JOIN` operator replaces the `CROSS_PRODUCT` node that was there before and the estimated cardinality went from 5,000,000 to 50,000! Executing this query now takes about 0.15 seconds. That's an almost 66× speedup!

This optimization was just recently added to the VSS extension, so if you've already installed `vss` for DuckDB v1.1.2, run the following command to get the latest version:

```
UPDATE EXTENSIONS (vss);
```

Conclusion

That's all for this time folks! We hope you've enjoyed this update on the DuckDB Vector Similarity Search extension. While this update has focused a lot on new features and improvements such as faster indexing, additional distance functions and more optimizer rules, we're still working on improving some of the limitations mentioned in the previous blog post. We hope to have more to share related to custom

indexes and index-based optimizations soon! If you have any questions or feedback, feel free to reach out to us on the `duckdb-vss` GitHub repository or on the DuckDB Discord. Hope to see you around!

Fast Top N Aggregation and Filtering with DuckDB

Publication date: 2024-10-25

Author: Alex Monahan

TL;DR: Find the top N values or filter to the latest N rows more quickly and easily with the N parameter in the `min`, `max`, `min_by`, and `max_by` aggregate functions.

Introduction to Top N

A common pattern when analyzing data is to look for the rows of data that are the highest or lowest in a particular metric. When interested in the highest or lowest N rows in an entire dataset, SQL's standard `ORDER BY` and `LIMIT` clauses will sort by the metric of interest and only return N rows. For example, using the scale factor 1 (SF1) data set of the TPC-H benchmark:

```
INSTALL tpch;
LOAD tpch;
-- Generate an example TPC-H dataset
CALL dbgen(sf = 1);

-- Return the most recent 3 rows by l_shipdate
FROM lineitem
ORDER BY
    l_shipdate DESC
LIMIT 3;
```

l_orderkey	l_partkey	...	l_shipmode	l_comment
354528	6116	...	MAIL	wake according to the u
413956	16402	...	SHIP	usual patterns. carefull
484581	10970	...	TRUCK	ccounts maintain. dogged accounts a

This is useful to quickly get the oldest or newest values in a dataset or to find outliers in a particular metric.

Another common approach is to query the min/max summary statistics of one or more columns. This can find outliers, but the row that contains the outlier can be different for each column, so it is answering a different question. DuckDB's helpful `COLUMNS` expression allows us to calculate the maximum value for all columns.

```
FROM lineitem
SELECT
    max(COLUMNS(*));
```

> The queries in this post make extensive use of DuckDB's FROM-first syntax. This allows the `FROM` and `SELECT` clauses to be swapped, and it even allows omitting the latter entirely.

l_orderkey	l_partkey	...	l_shipmode	l_comment
600000	20000	...	TRUCK	zzle. slyly

However, these two approaches can only answer certain kinds of questions. There are many scenarios where the goal is to understand the top N values *within a group*. In the first example above, how would we calculate the last 10 shipments from each supplier? SQL's `LIMIT` clause is not able to handle that situation. Let's call this type of analysis the top N by group.

This type of analysis is a common tool for exploring new datasets. Use cases include pulling the most recent few rows for each group or finding the most extreme few values in a group. Sticking with our shipment example, we could look at the last 10 shipments of each part number, or find the 5 highest priced orders per customer.

Traditional Top N by Group

In most databases, the way to filter to the top N within a group is to use a window function and a common table expression (CTE). This approach also works in DuckDB. For example, this query returns the 3 most recent shipments for each supplier:

```
WITH ranked_lineitem AS (
    FROM lineitem
    SELECT
        *,
        row_number() OVER
            (PARTITION BY l_suppkey ORDER BY l_shipdate DESC)
            AS my_ranking
)
FROM ranked_lineitem
WHERE
    my_ranking <= 3;
```

l_orderkey	l_partkey	l_suppkey	...	l_shipmode	l_comment	my_ranking
1310688	169532	7081	...	RAIL	ully final exc	1
910561	194561	7081	...	SHIP	ly bold excuses caj	2
4406883	179529	7081	...	RAIL	tions. furious	3
4792742	52095	7106	...	RAIL	onic, ironic courts. final deposits sleep	1
4010212	122081	7106	...	MAIL	accounts cajole finally ironic instruc	2
1220871	94596	7106	...	TRUCK	regular requests above t	3
...	...	...	...	...	...	...

In DuckDB, this can be simplified using the `QUALIFY` clause. `QUALIFY` acts like a `WHERE` clause, but specifically operates on the results of window functions. By making this adjustment, the CTE can be avoided while returning the same results.

```
FROM lineitem
SELECT
    *,
    row_number() OVER
        (PARTITION BY l_suppkey ORDER BY l_shipdate DESC)
        AS my_ranking
QUALIFY
    my_ranking <= 3;
```

This is certainly a viable approach! However, what are its weaknesses? Even though the query is interested in only the 3 most recent shipments, it must sort every shipment just to retrieve those top 3. Sorting in DuckDB has a complexity of `O(kn)` due to DuckDB's innovative Radix sort implementation, but this is still higher than the `O(n)` of DuckDB's hash aggregate, for example. Sorting is also a memory intensive operation when compared with aggregation.

Top N in DuckDB

DuckDB 1.1 added a new capability to dramatically simplify and improve performance of top N calculations. Namely, the functions `min`, `max`, `min_by`, and `max_by` all now accept an optional parameter N. If N is greater than 1 (the default), they will return an array of the top values.

As a simple example, let's query the most recent (top 3) shipment dates:

```
FROM lineitem
SELECT
    max(l_shipdate, 3) AS top_3_shipdates;
```

top_3_shipdates
[1998-12-01, 1998-12-01, 1998-12-01]

Top N by Column in DuckDB

The top N selection can become even more useful thanks to the `COLUMNS` expression once again – we can retrieve the 3 top values in each column. We can call this a *top N by column analysis.* It is particularly messy to try to do this analysis with ordinary SQL! You would need a subquery or window function for every single column... In DuckDB, simply:

```
FROM lineitem
SELECT
    max(COLUMNS(*), 3) AS "top_3_\0";
```

top_3_l_orderkey	top_3_l_partkey	...	top_3_l_shipmode	top_3_l_comment
[600000, 600000, 599975]	[20000, 20000, 20000]	...	[TRUCK, TRUCK, TRUCK]	[zzle. slyly, zzle. quickly bold a, zzle. pinto beans boost slyly slyly fin]

Top N by Group in DuckDB

Armed with the new N parameter, how can we speed up a top N by group analysis?

Want to cut to the chase and see the final output? Feel free to skip ahead!

We will take advantage of three other DuckDB SQL features to make this possible:

- The `max_by` function (also known as `arg_max`)
- The `unnest` function
- Automatically packing an entire row into a `STRUCT` column

The `max` function will return the max (or now the max N!) of a specific column. In contrast, the `max_by` function will find the maximum value in a column, and then retrieve a value from the same row, but a different column. For example, this query will return the ids of the 3 most recently shipped orders for each supplier:

```
FROM lineitem
SELECT
    l_suppkey,
    max_by(l_orderkey, l_shipdate, 3) AS recent_orders
GROUP BY
    l_suppkey;
```

l_suppkey	recent_orders
2992	[233573, 3597639, 3060227]
8516	[4675968, 5431174, 4626530]
3205	[3844610, 4396966, 3405255]
2152	[1672000, 4209601, 3831138]
1880	[4852999, 2863747, 1650084]
...	...

The `max_by` function is an aggregate function, so it takes advantage of DuckDB's fast hash aggregation rather than sorting. Instead of sorting by `l_shipdate`, the `max_by` function scans through the dataset just once and keeps track of the N highest `l_shipdate` values. It then returns the order id that corresponds with each of the most recent shipment dates. The radix sort in DuckDB must scan through the dataset once per byte, so scanning only once provides a significant speedup. For example, if sorting by a 64-bit integer, the sort algorithm must loop through the dataset 8 times vs. 1 with this approach! A simple micro-benchmark is included in the Performance Comparisons section.

However, this SQL query has a few gaps. The query returns results as a `LIST` rather than as separate rows. Thankfully the `unnest` function can split a `LIST` into separate rows:

```
FROM lineitem
SELECT
    l_suppkey,
    unnest(
        max_by(l_orderkey, l_shipdate, 3)
    ) AS recent_orders
GROUP BY
    l_suppkey;
```

l_suppkey	recent_orders
2576	930468
2576	2248354
2576	3640711
5559	4022148
5559	1675680
5559	4976259
...	...

The next gap is that there is no way to easily see the `l_shipdate` associated with the returned `l_orderkey` values. This query only returns a single column, while typically a top N by group analysis will require the entire row.

Fortunately, DuckDB allows us to refer to the entire contents of a row as if it were just a single column! By referring to the name of the table itself (here, `lineitem`) instead of the name of a column, the `max_by` function can retrieve all columns.

```
FROM lineitem
SELECT
    l_suppkey,
    unnest(
        max_by(lineitem, l_shipdate, 3)
    ) AS recent_orders
GROUP BY
    l_suppkey;
```

l_suppkey	recent_orders
5411	{'l_orderkey': 2543618, 'l_partkey': 105410, 'l_suppkey': 5411, ...
5411	{'l_orderkey': 580547, 'l_partkey': 130384, 'l_suppkey': 5411, ...
5411	{'l_orderkey': 3908642, 'l_partkey': 132897, 'l_suppkey': 5411, ...
90	{'l_orderkey': 4529697, 'l_partkey': 122553, 'l_suppkey': 90, ...
90	{'l_orderkey': 4473346, 'l_partkey': 160089, 'l_suppkey': 90, ...
...	...

Let's make that a bit friendlier looking by splitting the STRUCT out into separate columns to match our original dataset.

The Final Top N by Group Query

Passing in one more argument to UNNEST will split this out into separate columns by running recursively. In this case, that means that UNNEST will run twice: once to convert each `LIST` into separate rows, and then again to convert each STRUCT into separate columns. The `l_suppkey` column can also be excluded, since it will automatically be included already.

```
FROM lineitem
SELECT
    unnest(
        max_by(lineitem, l_shipdate, 3),
        recursive := 1
    ) AS recent_orders
GROUP BY
    l_suppkey;
```

l_orderkey	l_partkey	l_suppkey	...	l_shipinstruct	l_shipmode	l_comment
1234726	6875	6876	...	COLLECT COD	FOB	cajole carefully slyly fin
2584193	51865	6876	...	TAKE BACK RETURN	TRUCK	fully regular deposits at the q
2375524	26875	6876	...	DELIVER IN PERSON	AIR	nusual ideas. busily bold deposi
5751559	95626	8136	...	NONE	SHIP	ers nag fluffily against the spe
3103457	103115	8136	...	TAKE BACK RETURN	FOB	y slyly express warthogs-- unusual, e
5759105	178135	8136	...	COLLECT COD	TRUCK	es. regular pinto beans haggle.
...	...	...	...	...	...	...

> This approach can also be useful for the common task of de-duplicating by finding the latest value within a group. One pattern is to find the current state of a dataset by returning the most recent event in an events table. Simply use an N of 1!

We now have a way to use an aggregate function to calculate the top N rows per group! So, how much more efficient is it?

Performance Comparisons

We will compare the QUALIFY approach with the max_by approach for solving the top N by group problem. We have discussed both queries, but for reference they are repeated below.

While the main query is running, we will also kick off a background thread to periodically measure DuckDB's memory use. This uses the built in table function duckdb_memory() and includes information about Memory usage as well as temporary disk usage. The small Python script used for benchmarking is included below the results. The machine used for benchmarking was an M1 MacBook Pro with 16 GB RAM.

SF	max_memory	Metric	QUALIFY	max_by	Improvement
1	Default	Total time	0.58 s	0.24 s	2.4×
5	Default	Total time	6.15 s	1.26 s	4.9×
10	36 GB	Total time	36.8 s	25.4 s	1.4×
1	Default	Memory usage	1.7 GB	0.2 GB	8.5×
5	Default	Memory usage	7.9 GB	1.5 GB	5.3×
10	36 GB	Memory usage	15.7 GB	17.1 GB	0.9×

We can see that in each of these situations, the max_by approach is faster, in some cases nearly 5× faster! However, as the data grows larger, the max_by approach begins to weaken relative to QUALIFY.

In some cases, the memory use is significantly lower with max_by also. However, the memory use of the max_by approach becomes more significant as scale increases, because the number of distinct l_suppkey values increases linearly with the scale factor. This increased memory use likely explains the performance decrease, as both algorithms approached the maximum amount of RAM on my machine and began to swap to disk.

In order to reduce the memory pressure, let's re-run the scale factor 10 (SF10) benchmark using fewer threads (4 threads and 1 thread). We continue to use a max_memory setting of 36 GB. The prior SF10 results with all 10 threads are included for reference.

SF	Threads	Metric	QUALIFY	max_by	Improvement
10	10	Total time	36.8 s	25.4 s	1.4×
10	4	Total time	49.0 s	21.0 s	2.3×
10	1	Total time	115.7 s	12.7 s	9.1×
10	10	Memory usage	15.7 GB	17.1 GB	0.9×
10	4	Memory usage	15.9 GB	17.3 GB	0.9×
10	1	Memory usage	14.5 GB	1.8 GB	8.1×

The max_by approach is so computationally efficient that even with 1 thread it is dramatically faster than the QUALIFY approach that uses all 10 threads! Reducing the thread count very effectively lowered the memory use as well (a nearly 10× reduction).

So, when should we use each? As with all database things, *it depends!* If memory is constrained, max_by may also offer benefits, especially when the thread count is tuned to avoid spilling to disk. However, if there are approximately as many groups as there are rows, consider QUALIFY since we lose some of the memory efficiency of the max_by approach.

Conclusion

DuckDB now offers a convenient way to calculate the top N values of both min and max aggregate functions, as well as their advanced cousins min_by and max_by. They are easy to get started with, and also enable more complex analyses like calculating the top N for all columns or the top N by group. There are also possible performance benefits when compared with a window function approach.

We would love to hear about the creative ways you are able to use this new feature!

Happy analyzing!

Analytics-Optimized Concurrent Transactions

Publication date: 2024-10-30

Authors: Mark Raasveldt and Hannes Mühleisen

TL;DR: DuckDB employs unique analytics-optimized optimistic multi-version concurrency control techniques. These allow DuckDB to perform large-scale in-place updates efficiently.

> This is the second post on DuckDB's ACID support. If you have not read the first post, Changing Data with Confidence and ACID, it may be a good idea to start there.

In our previous post, we have discussed why changes to data are much saner if the formal "ACID" transaction properties hold. A data system should not allow importing "half" a CSV file into a table because of some unexpected string in line 431,741.

Ensuring the ACID properties of transactions under concurrency is very challenging and one of the "holy grails" of databases. DuckDB implements advanced methods for concurrency control and logging. In this post, we describe DuckDB's Multi-Version Concurrency (MVCC) and Write-Ahead-Logging (WAL) schemes that are specifically designed for efficiently ensuring the transactional guarantees for analytical use cases under concurrent workloads.

Concurrency Control

Pessimistic Concurrency Control. Traditional database systems use locks to manage concurrency. A transaction obtains locks in order to ensure that (a) no other transaction can see its uncommitted changes, and (b) it does not see uncommitted changes of other transactions. Locks need to be obtained both when **reading** (shared locks) and when **writing** (exclusive locks). When a different transaction tries to read data that has been written to by another transaction – it must wait for the other transaction to complete and release its exclusive lock on the data. This type of concurrency control is called **pessimistic**, because locks are always obtained, even if there are no conflicts between transactions.

This strategy works well for transactional workloads. These workloads consist of small transactions that read or modify a few rows. A typical transaction only locks a few rows, and keeps those rows locked only for a short period of time. For analytical workloads, on the other hand, this strategy does not work well. These workloads consist of large transactions that read or modify large parts of the table. An analytical transaction executed in a system that uses pessimistic concurrency control will therefore lock many rows, and keep those rows locked for a long period of time, preventing other transactions from executing.

Optimistic Concurrency Control. DuckDB uses a different approach to manage concurrency conflicts. Transactions do not hold locks – they can always read and write to any row in any table. When a conflict occurs and multiple transactions try to write to the same row at the same time – one of the conflicting transactions is instead aborted. The aborted transaction can then be retried if desired. This type of concurrency control is called **optimistic**.

In case there are never any concurrency conflicts – this strategy is very efficient as we have not unnecessarily slowed down transactions by pessimistically grabbing locks. This strategy works well for analytical workloads – as read-only transactions can never conflict with one another, and multiple writers that modify the same rows are rare in these workloads.

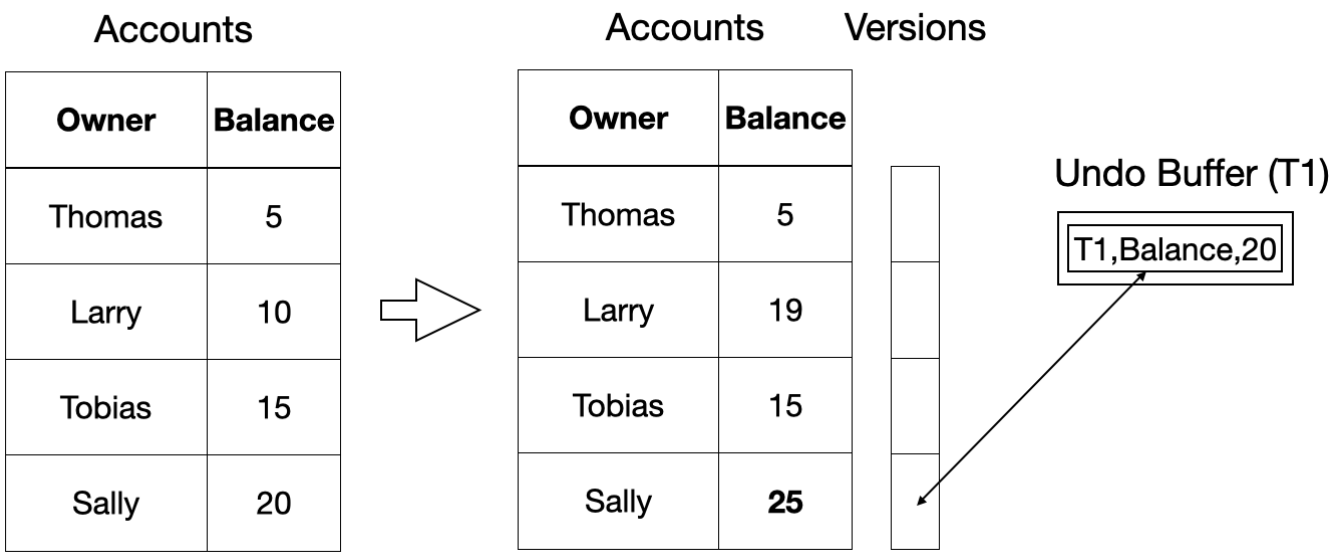
Multi-Version Concurrency Control

In an optimistic concurrency control model – multiple transactions can read and make changes to the same tables at the same time. We have to ensure these transactions cannot see each others' *half-done* changes in order to maintain ACID isolation. A well-known technique to

achieve this is Multi-Version Concurrency Control (MVCC). MVCC works by keeping **multiple versions** of modified rows. When a transaction modifies a row – we can create a copy of that row and modify that instead. This allows other transactions to keep on reading the original version of the row. This allows for each transaction to see their own, consistent state of the database. Often that state is the "version" that existed when the transaction was started. MVCC is widely used in database systems, for example PostgreSQL also uses MVCC.

DuckDB implements MVCC using a technique inspired by the paper "Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems" by the one and only Thomas Neumann. This MVCC implementation works by maintaining a list of previous versions to each row in a table. Transactions will update the table data in-place, but will save the previous version of the updated row in the undo buffers. Below is an illustrated example.

```
-- add 5 to Sally's balance
UPDATE Accounts SET Balance = Balance + 5 WHERE Name = 'Sally';
```



When reading a row, a transaction will first check if there is version information for that row. If there is none, which is the common case, the transaction can read the original data. If there is version information, the transaction has to compare the transaction number at the transaction's start time with those in the undo buffers and pick the right version to read.

Efficient MVCC for Analytics

The above approach works well for transactional workloads where individual rows are changed frequently. For *analytical* use cases, we observe a very different usage pattern: changes are much more "bulky" and they often only affect a subset of columns. For example, we do not usually delete individual rows but instead delete all rows matching a pattern, e.g.:

```
DELETE FROM orders WHERE order_time < DATE '2010-01-01';
```

We also commonly bulk update columns, e.g., to fix the evergreen annoyance of people using nonsensical in-domain values to express `NULL`:

```
UPDATE people SET age = NULL WHERE age = -99;
```

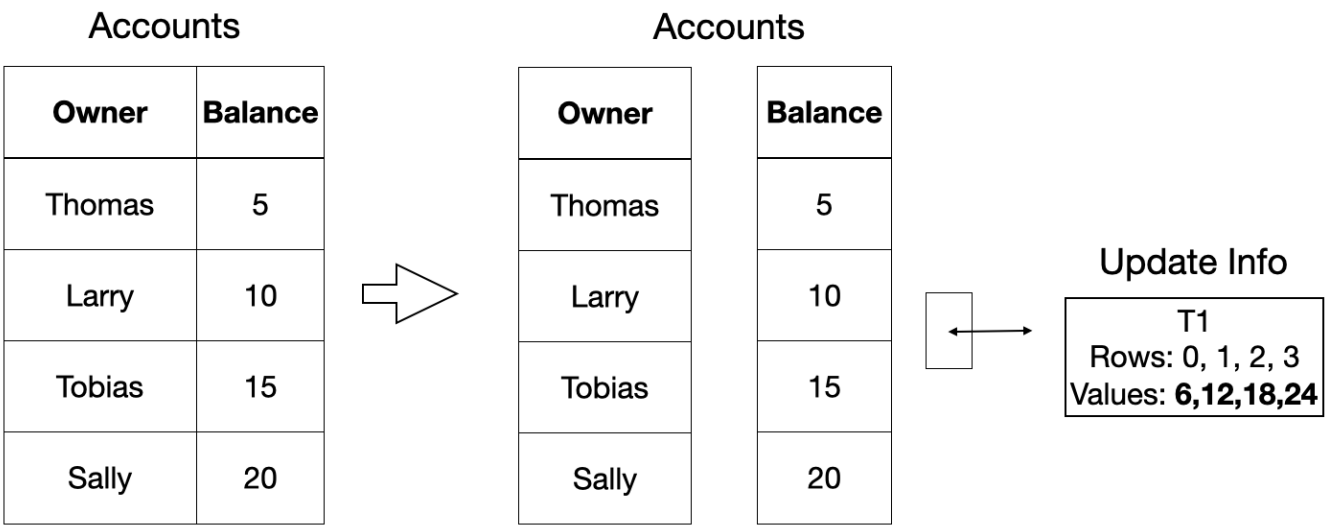
If every row has version information, such bulk changes create a *huge* amount of entries in the undo buffers, which consume a lot of memory and are inefficient to operate on and read from.

There is also an added complication – the original approach relies on performing *in-place updates*. While we can efficiently perform in-place updates on uncompressed data, this is not possible when data is compressed. As DuckDB keeps data compressed, both on-disk and in-memory, in-place updates cannot be performed.

In order to address these issues – DuckDB instead stores **bulk version information** on a per-column basis. For every batch of `2048` rows, a single version information entry is stored. The version information stores the changes made to the data, instead of the old data, as we

cannot modify the original data in-place. Instead, any changes made to the data are flushed to disk during a checkpoint. Below is an illustrated example.

```
-- add 20% interest to all accounts
UPDATE Accounts SET Balance = Balance + Balance / 5;
```



One beautiful aspect of this undo buffer scheme is that it is largely performance-transparent: if no changes are made, there are no extra computational cost associated with providing support for transactions. To the best of our knowledge, DuckDB is the *only transactional data management system that is optimized for bulk changes to data* that are common in analytical use cases. But even with changes present, our transaction scheme is very fast for the kind of transactions that we expect for analytical use cases.

Benchmarks

Here is a small experiment, comparing DuckDB 1.1.0, HyPer 9.1.0, SQLite 3.43.2, and PosgreSQL 14.13 on a recent MacBook Pro, showing some of the effects that an OLAP-optimized transaction scheme will have. We should note that HyPer implements the MVCC scheme from the Neumann paper mentioned above. SQLite does not actually implement MVCC, it is mostly included as a comparison point.

We create two tables with either 1 or 100 columns, each with 10 million rows, containing the integer values 1-100 repeating.

```
CREATE TABLE mvcc_test_1 (i INTEGER);
INSERT INTO mvcc_test_1
    SELECT s1
    FROM
        generate_series(1, 100) s1(s1),
        generate_series(1, 100_000) s2(s2);

CREATE TABLE mvcc_test_100 (i INTEGER,
    j1 INTEGER, j2 INTEGER, ..., j99 INTEGER);
INSERT INTO mvcc_test_100
    SELECT s1, s1, s1, ..., s1
    FROM
        generate_series(1, 100) s1(s1),
        generate_series(1, 100_000) s2(s2);
```

We then run three transactions on both tables that increment a single column, with an increasing number of affected rows, 1%, 10% and 100%:

```
UPDATE mvcc_test_... SET i = i + 1 WHERE i <= 1;
UPDATE mvcc_test_... SET i = i + 1 WHERE i <= 10;
UPDATE mvcc_test_... SET i = i + 1 WHERE i <= 100;
```

For the **single-column case**, there should not be huge differences between using a row-major or a column-major concurrency control scheme, and indeed the results show this:

1 Column	1%	10%	100%
DuckDB	0.02	0.07	0.43
SQLite	0.21	0.25	0.61
HyPer	0.66	0.28	2.37
PostgreSQL	1.44	2.48	19.07

Changing more rows took more time. The rows are small, each row only contain a single value. DuckDB and HyPer, having more modern MVCC scheme based on undo buffers as outlined above, are generally much faster than PostgreSQL. SQLite is doing well, but of course it does not have any MVCC. Timings increase roughly 10× as the amount of rows changed is increased tenfold. So far so good.

For the **100 column case**, results look drastically different:

100 Columns	1%	10%	100%
DuckDB	0.02	0.07	0.43
SQLite	0.51	1.79	12.93
HyPer	0.66	6.06	61.54
PostgreSQL	1.42	5.45	50.05

Recall that here we are changing a single column out of 100, a common use case in wrangling analytical data sets. Because DuckDB's MVCC scheme is *designed* for those use cases, it shows exactly the same runtime as in the single-column experiment above. In SQLite, there is a clear impact of the larger row size on the time taken to complete the updates even without MVCC. HyPer and PostgreSQL also show much larger, up to 100× (!) slowdowns as the amount of changed rows is increased.

This neatly brings us to checkpointing.

Write-Ahead Logging and Checkpointing

Any data that's not written to disk but instead still lingers in CPU caches or main memory will be lost in case the operating system crashes or if power is lost. To guarantee durability of changes in the presence of those adverse events, DuckDB needs to *ensure that any committed changes are written to persistent storage*. However, changes in a transaction can be scattered all over potentially large tables, and fully writing them to disk can be quite slow, especially if it has to happen before any transaction can commit. Also, we don't yet know if we actually want to persist a change, we may encounter a failure in the very process of committing.

The traditional approach of transactional data management systems to balance the requirement of writing changes to persistent storage with the requirement of not taking forever is the write-ahead log (WAL). The WAL can be thought of as a log file of all changes to the database. On each transaction commit, its changes are written to the WAL. On restart, the database files are re-loaded from disk, the changes in the WAL are re-applied (if present), and things happily continue. After some amount of changes, the changes in the WAL need to be physically applied to the table, a process known as “checkpointing”. Afterward, the WAL entries can be discarded, a process known as “truncating”. This scheme ensures that changes persist even if a crash occurs or power is lost immediately after a commit.

DuckDB implements write-ahead logging and you may have seen a `.wal` file appearing here and there. Checkpointing normally happens *automatically* whenever the WAL file reached a limit, by default 16 MB but this can be adjusted with the `checkpoint_threshold` setting. Checkpoints also automatically happen at database shutdown. Checkpoints can also be explicitly triggered with the `CHECKPOINT` and `FORCE CHECKPOINT` commands, the difference being that the latter will abort (rollback) any active transactions to ensure the checkpoing is happening *right now* while the former will wait.

DuckDB explicitly calls the `fsync()` system call to make sure any WAL entries will be forced to be written to persistent storage, ignoring the many caches on the way. This is *necessary* because those caches may also be lost in the event of, e.g., power failure, so it's no use to only write log entries to the WAL if they end up not being actually written to storage because the operating system or the disk decided that it was better to wait for performance reasons. However, `fsync()` does take some time, and while it's generally considered bad practice, there are systems out there that don't do this at all or not by default in order to boast about more transactions per second.

In DuckDB, even bulk loads such as loading large files into tables (e.g., using the COPY statement) are fully transactional. This means you can do something like this:

```
BEGIN TRANSACTION;
CREATE TABLE people (age INTEGER, ...);
COPY people FROM 'many_people.csv';
UPDATE people SET age = NULL WHERE age = -99;
SELECT
    CASE
        WHEN (SELECT count(*) FROM people) = 1_000_000 THEN true
        ELSE error('expected 1m rows')
    END;
COMMIT;
```

This transaction creates a table, copies a large CSV file into the table, and then updates the table to replace a magic value. Finally, a check is performed to see if there is the expected number of rows in the table. All this is bound together into a *single transaction*. If anything goes wrong at any point in the process or the check fails, the transaction will be aborted and zero changes to the database will have happened, the table will not even exist. This is great because it allows implementing all-or-nothing semantics for complex loading tasks, possibly into many tables.

However, logging large changes is a problem. Imagine the `many_people.csv` file being large, say ten gigabytes. As discussed, all changes are written to the WAL and eventually checkpointed. The changes in the file are large enough to immediately trigger a checkpoint. So now we're first writing ten gigabytes to the WAL, and then reading them again, and then writing them again to the database file. Instead of reading ten and writing ten, we have read twenty and written twenty. This is not ideal, but rather than allowing to bypass transactions for bulk loads, DuckDB will instead *optimistically write large changes to new blocks in the database file directly*, and merely add a reference to the WAL. On commit, these new blocks are added to the table. On rollback, the blocks are marked as free space. So while this can lead to the database file pointlessly increasing in size if transactions are aborted, the common case will benefit greatly. Again, this means that users experience near-zero-cost transactionality.

More Experiments

Making concurrency control and write-ahead looking work correctly in the face of failure is very challenging. Software engineers are biased towards the “happy path”, where everything works as intended. The well-known TPC-H benchmark actually contains tests that stress concurrency and logging schemes (Section 3.5.4, “Durability Tests”). Our previous blog post also implemented this test and DuckDB passed.

In addition, we also defined our own, even more challenging test for durability: we run the TPC-H refresh sets one-by-one, in a sub-process. The sub-process reports the last commited refresh. As they are run, after a random (short) time interval, that sub-process is being killed (using `SIGKILL`). Then, DuckDB is restarted, it will likely start recovering from WAL and then continue with the refresh sets. Because of the random time interval, it is likely that DuckDB also gets killed during WAL recovery. This of course should not have any impact on the contents of the database. Finally, we have pre-computed the correct result after running 4000 refresh sets using DuckDB, and after all is set and done we check if there are any differences. There were none, luckily.

To stress our implementation further, we have repeated this experiment on a special file system, LazyFS. This FUSE file system is specifically designed to help uncover bugs in database systems by – among other things – not properly flushing changes to disk using `fsync()`. In our LazyFS configuration, any change that is written to a file is discarded *unless* sync-ed, which also happens if a file is closed. So in our experiment where we kill the database any un-sync-ed entries in the WAL would be lost. We've re-run our durability tests described above on LazyFS and are also **happy to report that no issues were found**.

Conclusion

In this post, we described DuckDB's approaches to concurrency control and write-ahead logging. Of course, we are constantly working on improving them. One nasty failure mode that can appear in real-world systems are partial (“torn”) writes to files, where only parts of write requests actually make it to the file. Luckily, LazyFS can be configured to be even more hostile, for example failing read and write system calls entirely, returning partial or wrong data, or only partially writing the data to disk. We plan to expand our experimentation on this, to make sure DuckDB's transaction handling is as bullet-proof as it can be.

And who knows, maybe we even dare to unleash the famous Kyle of Jepsen on DuckDB at some point.

Optimizers: The Low-Key MVP

Publication date: 2024-11-14

Author: Tom Ebergen

TL;DR: The query optimizer is an important part of any analytical database system as it provides considerable performance improvements compared to hand-optimized queries, even as the state of your data changes.

Optimizers don't often give ”main character” energy in the database community. Databases are usually popular because of their performance, ease of integration, or reliability. As someone who mostly works on the optimizer in DuckDB, I have been wanting to write a blog post about how important optimizers are and why they merit more recognition. In this blog post we will analyze queries that fall into one of three categories: unoptimized, hand-optimized, and optimized by the DuckDB query optimizer. I will also explain why built-in optimizers are almost always better than any hand optimizations. Hopefully, by the end of this blog post, you will agree that optimizers play a silent, but vital role when using a database. Let's first start by understanding where in the execution pipeline query optimization happens.

Before any data is read from the database, the given SQL text must be parsed and validated. If this process finishes successfully, a tree-based query plan is created. The query plan produced by the parser is naïve, and can be extremely inefficient depending on the query. This is where the optimizer comes in, the inefficient query plan is passed to the optimizer for modification and, you guessed it, optimization. The optimizer is made up of many optimization rules. Each rule has the ability to reorder, insert, and delete query operations to create a slightly more efficient query plan that is also logically equivalent. Once all the optimization rules are applied, the optimized plan can be much more efficient than the plan produced by the parser.

> In practice an optimization rule can also be called an optimizer. For the rest of this blog post, optimizer rule will be used for a specific optimization, and optimizer will refer to the database optimizer, unless the word optimizer names a specific optimization rule, (i.e., *Join Order Optimizer*).

Normal Queries vs. Optimized Queries

To examine the effect of the DuckDB query optimizer, let's use a subset of the NYC taxi dataset. You can create native DuckDB tables with the following commands (note that `taxi-data-2019.parquet` is approximately 1.3 GB):

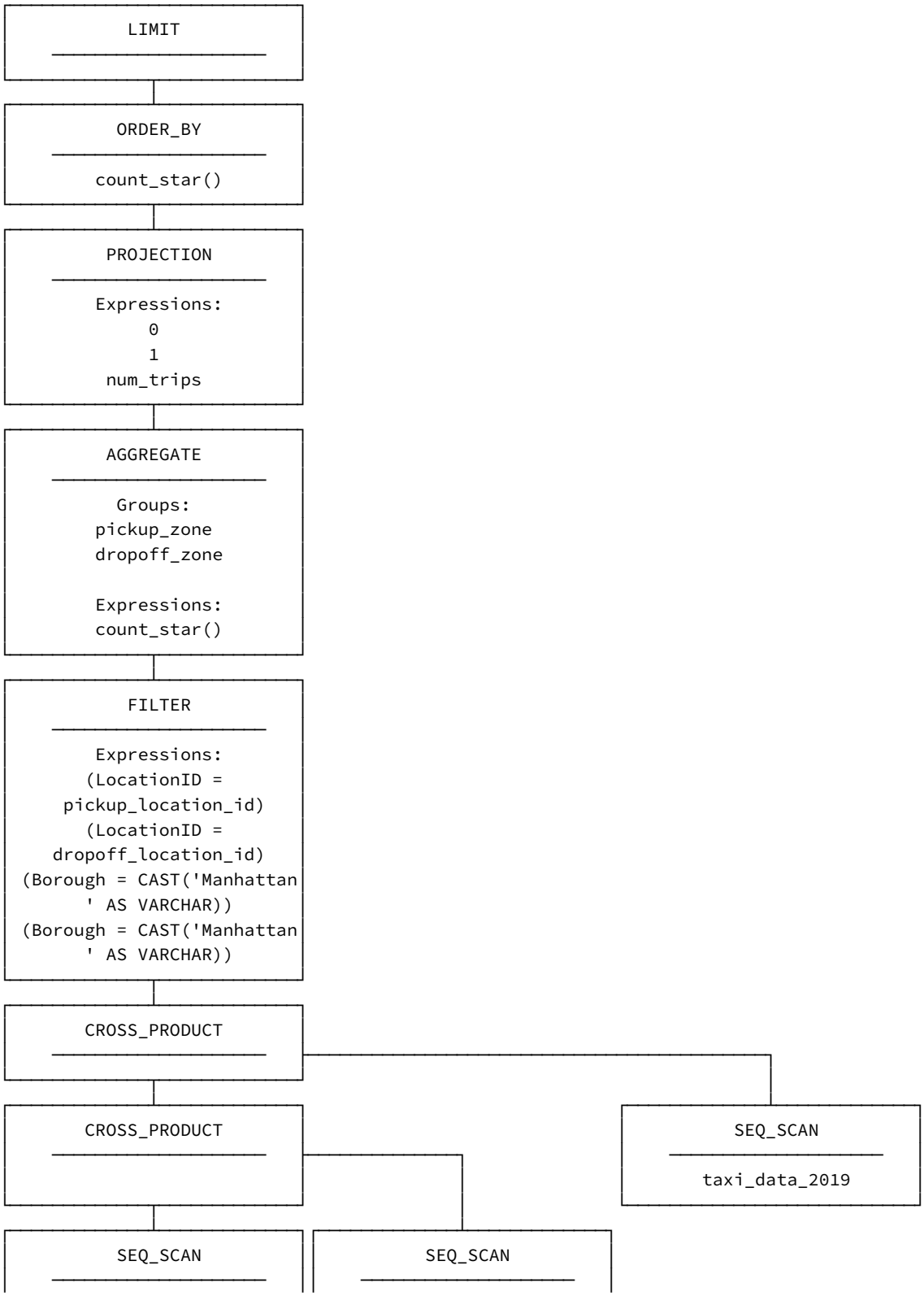
```
CREATE TABLE taxi_data_2019 AS
    FROM 'https://blobs.duckdb.org/data/taxi-data-2019.parquet';
CREATE TABLE zone_lookups AS
    FROM 'https://blobs.duckdb.org/data/zone-lookups.parquet';
```

Now that we have all 2019 data, let's look at the unoptimized vs. optimized plans for a simple query. The following SQL query gets us the most common pickup and drop-off pairs in the Manhattan borough.

```
PRAGMA disable_optimizer;
PRAGMA explain_output = 'optimized_only';
EXPLAIN SELECT
    pickup.zone AS pickup_zone,
    dropoff.zone AS dropoff_zone,
    count(*) AS num_trips
FROM
    zone_lookups AS pickup,
    zone_lookups AS dropoff,
    taxi_data_2019 AS data
WHERE pickup.LocationID = data.pickup_location_id
  AND dropoff.LocationID = data.dropoff_location_id
```

```
  AND pickup.Borough = 'Manhattan'
  AND dropoff.Borough = 'Manhattan'
GROUP BY pickup_zone, dropoff_zone
ORDER BY num_trips DESC
LIMIT 5;
```

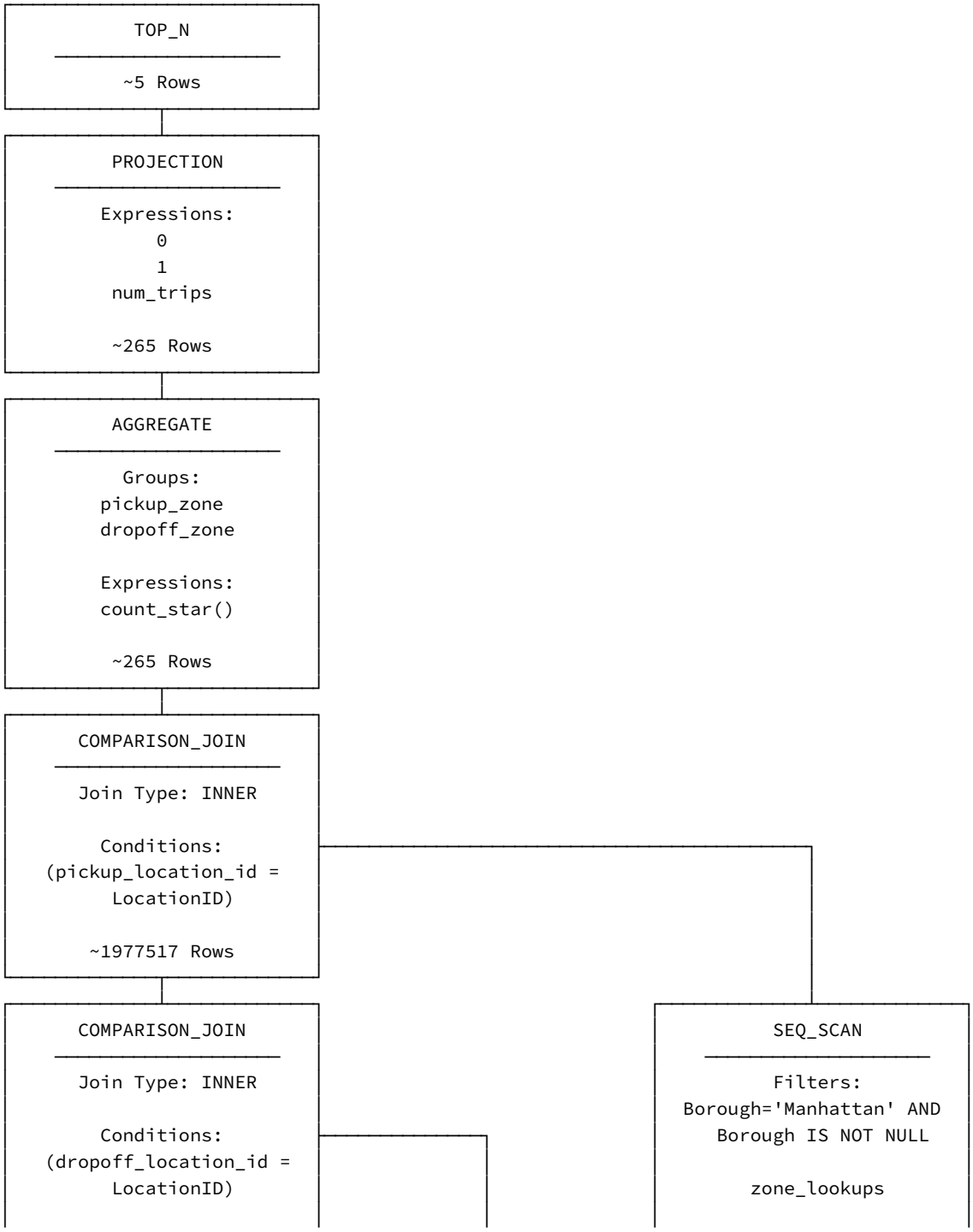
Running this EXPLAIN query gives us the following plan.

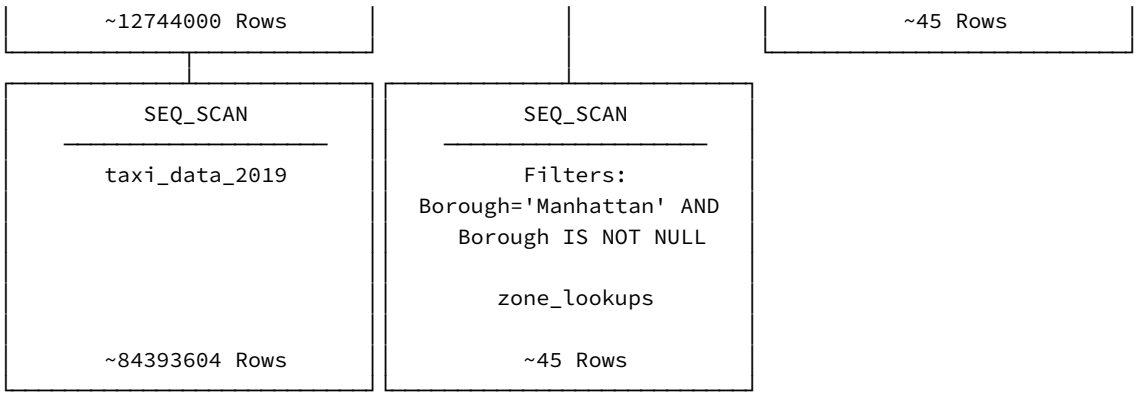


```
┌──────────────────────────┐ ┌──────────────────────────┐
│       zone_lookups       │ │       zone_lookups       │
└──────────────────────────┘ └──────────────────────────┘
```

The cross products alone make this query extremely inefficient. The cross-products produce `256 * 256 * |taxi_data_2019|` rows of data, which is 5 trillion rows of data. The filter only matches 71 million rows, which is only 0.001% of the data. The aggregate produces 4,373 rows of data, which need to be sorted by the `ORDER BY` operation, which runs in `O(N * log N)`. Producing 5 trillion tuples alone is an enormous amount of data processing, which becomes clear when you try to run the query and notice it doesn't complete. With the optimizer enabled, the query plan produced is much more efficient because the operations are re-ordered to avoid many trillions of rows of intermediate data. Below is the query plan with the optimizer enabled:

```
PRAGMA enable_optimizer;
EXPLAIN ...;
```





Let's first look at the difference in execution times on my MacBook with an M1 Max and 32 GB of memory before talking about the optimizations that have taken place.

	Unoptimized	Optimized
Runtime	>24 hours	0.769 s

Hopefully this performance benefit illustrates how powerful the DuckDB Optimizer is. So what optimization rules are responsible for these drastic performance improvements? For the query above, there are three powerful rules that are applied when optimizing the query: *Filter Pushdown, Join Order Optimization,* and *TopN Optimization*.

The *Filter Pushdown Optimizer* is very useful since it reduces the amount of intermediate data being processed. It is an optimization rule that is sometimes easy to miss for humans and will always result in faster execution times if the filter is selective in any way. It takes a filter, like `Borough = 'Manhattan'` and pushes it down to the operator that first introduces the filtered column, in this case the table scan. In addition, it will also detect when a filtered column like `col1` is used in an equality condition (i.e., `WHERE col1 = col2`). In these cases, the filter is duplicated and applied to the other column, `col2`, further reducing the amount of intermediate data being processed.

The *Join Order Optimizer* recognizes that the filters `pickup.LocationID = data.pickup_location_id` and `dropoff.LocationID = data.dropoff_location_id` can be used as join conditions and rearranges the scans and joins accordingly. This optimizer rule does a lot of heavy lifting to reduce the amount of intermediate data being processed since it is responsible for removing the cross products.

The *TopN Optimizer* is very useful when aggregate data needs to be sorted. If a query has an `ORDER BY` and a `LIMIT` operator, a TopN operator can replace these two operators. The TopN operator orders only the highest/lowest `N` values, instead of all values. If `N` is 5, then DuckDB only needs to keep 5 rows with the minimum/maximum values in memory and can throw away the rest. So if you are only interested in the top `N` values out of `M`, where `N << M`, the TopN operator can run in `O(M + N * log N)` instead of `O(M * log M)`.

These are just a few of the optimizations DuckDB has. More optimizations are explained in the section Summary of All Optimizers.

Hand-Optimized Queries

For the query above, it is possible to achieve almost the same plan by carefully writing the SQL query by hand. To achieve a similar plan as the one generated by DuckDB, you can write the following.

```
SELECT
    pickup.zone AS pickup_zone,
    dropoff.zone AS dropoff_zone,
    count(*) AS num_trips
FROM
    taxi_data_2019 data
INNER JOIN
    (SELECT * FROM zone_lookups WHERE Borough = 'Manhattan') pickup
```

```
    ON pickup.LocationID = data.pickup_location_id
INNER JOIN
    (SELECT * FROM zone_lookups WHERE Borough = 'Manhattan') dropoff
    ON dropoff.LocationID = data.dropoff_location_id
GROUP BY pickup_zone, dropoff_zone
ORDER BY num_trips desc
LIMIT 5;
```

Inspecting the runtimes again we get:

	Unoptimized	Hand-optimized	Optimized
Runtime	>24 hours	0.926 s	0.769 s

The SQL above results in a plan similar to the DuckDB optimized plan, but it is wordier and more error-prone to write, which can potentially lead to bugs. In very rare cases, it is possible to hand write a query that produces a more efficient plan than an optimizer. These cases are extreme outliers, and in all other cases the optimizer will produce a better plan. Moreover, a hand-optimized query is optimized for the current state of the data, which can change with many updates over time. Once a sufficient amount of changes are applied to the data, the assumptions of a hand-optimized query may no longer hold, leading to bad performance. Let's take a look at the following example.

Suppose an upstart company has an `orders` and `parts` table and every time some dashboard loads, the most popular ordered parts needs to be calculated. Since the company is still relatively new, they only have a small amount orders, but their catalog of parts is still quite large. A hand-optimized query would look like this:

```
CREATE OR REPLACE TABLE orders AS
    SELECT RANGE order_id, range % 10_000 pid FROM range(1_000);
CREATE TABLE parts AS
    SELECT range p_id, range::VARCHAR AS part_name FROM range(10_000);
SELECT
    parts.p_id,
    parts.part_name,
    count(*) AS ordered_amount
FROM parts
INNER JOIN orders
    ON orders.pid = parts.p_id
GROUP BY ALL;
```

Naturally, the number of orders will increase as this company gains customers and grows in popularity. If the query above continues to run without the use of an optimizer, the performance will slowly decline. This is because the execution engine will build the hash table on the orders table, which potentially will have 100 million rows. If the optimizer is enabled, the Join Order Optimizer will be able to inspect the statistics of the table during the optimization process and produce a new plan according to the new state of the data.

Here is a breakdown of running the queries with and without the optimizer as the orders table increases.

	Unoptimized	Optimized
|orders| = 1K	0.004 s	0.003 s
|orders| = 10K	0.005 s	0.005 s
|orders| = 100K	0.013 s	0.008 s
|orders| = 1M	0.055 s	0.014 s
|orders| = 10M	0.240 s	0.044 s
|orders| = 100M	2.266 s	0.259 s

At first the difference in execution time is not really noticeable, so no one would think a query rewrite would be the solution. But once enough orders are reached, waiting 2 seconds every time the dashboard loads becomes tedious. If the optimizer is enabled, the query

performance improves by a factor of 10×. So if you ever think you have identified a scenario where you are smarter than the optimizer, make sure you have also thought about all possible updates to the data and have hand-optimized for those as well.

Optimizations That Are Impossible by Hand

Some optimization rules are also impossible to write by hand. For example, the TopN optimization can not be optimized by hand.

Another good example is the Join Filter Pushdown optimization. The Join Filter Pushdown optimization works in scenarios where the build side of a hash join has a subset of the join keys. In its current state the join filter pushdown optimization keeps track of the minimum value key and maximum value key and pushes a table filter into the probe side to filter out keys greater than the maximum join value and smaller than the minimum join value.

With a small change, we can use the query from above to demonstrate this. Suppose we first filter our `parts` table to only include parts with a specific prefix in the `part_name`. When the `orders` table has 100 million rows and the `parts` table only has ~20,000 after filtering, then the `orders` table will be the probe side and the `parts` table will be the hash/build side. When the hash table is built, the min and max `p_id` values in the `parts` table are recorded, in this case it could be 20,000 and 80,000. These min and max values get pushed as a filter into the `orders` table scan, filtering out all parts with `p_id > 80,000` and `pid < 20,000`. 40% of the `orders` table has a `pid` greater than 80,000, and less than 20,000 so this optimization does a lot of heavy lifting in join queries.

Imagine trying to express this logic in your favorite data frame API; it would be extremely difficult and error-prone. The library would need to implement this optimization automatically for all hash joins. The Join Filter Pushdown optimization can improve query performance by 10x, so it should be a key factor when deciding what analytical system to use.

If you use a data frame library like collapse, pandas, data.table, modin, then you are most likely not enjoying the benefits of query optimization techniques. This means your optimizations need to be applied by hand, which is not sustainable if your data starts changing. Moreover, you are most likely writing imperatively, using a syntax specific to the dataframe library. This means the scripts responsible for analyzing data are not very portable. SQL, on the other hand, can be much more intuitive to write since it is a declarative language, and can be ported to practically any other database system.

Summary of All Optimizers

Below is a non-exhaustive list of all the optimization rules that DuckDB applies.

Expression Rewriter

The *Expression Rewriter* simplifies expressions within each operator. Sometimes queries are written with expressions that are not completely evaluated or they can be rewritten in a way that takes advantage of features within the execution engine. Below is a table of common expression rewrites and the optimization rules that are responsible for them. Many of these rules rewrite expressions to use specialized DuckDB functions so expression evaluation is much faster during execution. If an expression can be evaluated to `true` in the optimizer phase, there is no need to pass the original expression to the execution engine. In addition, the optimized expressions are more likely to allow DuckDB to make further improvements to the query plan. For example, the "Move constants" rule could enable filter pushdown to occur.

Rewriter rule	Original expression	Optimized expression
Move constants	`x + 1 = 6`	`x = 5`
Constant folding	`2 + 2 = 4`	`true`
Conjunction simplification	`(1 = 2 AND b)`	`false`
Arithmetic simplification	`x * 1`	`x`
Case simplification	`CASE WHEN true THEN x ELSE y END`	`x`
Equal or NULL simplification	`a = b OR (a IS NULL AND b IS NULL)`	`a IS NOT DISTINCT FROM b`

Rewriter rule	Original expression	Optimized expression
Distributivity	`(x AND b) OR (x AND c) OR (x AND d)`	`x AND (b OR c OR d)`
Like optimization	`regexp_matches(c, '^Prefix')`	`LIKE 'Prefix%'`

Filter Pull-Up & Filter Pushdown

Filter Pushdown was explained briefly above. *Filter Pull-Up* is also important to identify cases where a filter can be applied on columns in other tables. For example, the query below scans column a from both `t1` and `t2`. `t1.a` has a filter, but in the presence of the equality condition, `t2.a` can have the same filter. For example:

```
SELECT *
FROM t1, t2
WHERE t1.a = t2.a
  AND t1.a = 50;
```

This can be optimized to:

```
SELECT *
FROM t1, t2
WHERE t1.a = t2.a
  AND t1.a = 50
  AND t2.a = 50;
```

Filter Pull-Up pulls up the filter `t1.a = 50` above the join, and when the filter is pushed down again, the optimizer rule recognizes the filter can be applied to both columns `t1.a` and `t2.a`.

IN Clause Rewriter

If there is a filter with an `IN` clause, sometimes it can be re-written so execution is more efficient. Some examples are below:

Original	Optimized
`c1 IN (1)`	`c1 = 1`
`c1 IN (3, 4, 5)`	`c1 >= 3 AND c1 <= 5`

In addition, the *IN Clause Rewriter* will transform expensive `IN` expressions into `MARK` joins. If a query has an expression like `c1 IN (x1, ..., xn)` where `n` is quite large, it can be expensive to evaluate this expression for every row in the table. The runtime would be `O(n * m)` where `n` is the number of rows and `m` is the length of the list. The `IN` clause rewriter will transform the expression into `SELECT c1 FROM t1, VALUES (x1, ..., xn) t(c0) WHERE c1 = c0` turning the expression into a `HASH` join that can complete in `O(n + m)` time!

Join Order Optimizer

The *Join Order Optimizer* can provide an enormous performance benefit by limiting the number of intermediate tuples that are processed between joins. By processing fewer intermediate tuples, the query can execute faster.

Statistics Propagation

Statistics Propagation is another optimization that works even when the state of the data changes. By traversing the query plan and keeping note of all equality join conditions, the Statistics Propagation optimizer can create new filters by inspecting the statistics of the columns

that are eventually joined. For example, suppose `t1.a` and `t2.a` will be joined with the equality condition `t1.a = t2.a`. If our internal statistics tell us `t1.a` has a maximum value of `50` and a minimum value of `25`, the optimizer can create a new filter when scanning table `t2`. The filter would be `t2.a >= 25 AND t2.a <= 50`.

Reorder Filters

If there are multiple filters on a column, the order in which these filters are executed also becomes important. It's best to execute the most efficient filters first, saving execution of expensive filters for later. For example, DuckDB can evaluate equality very quickly. So for a query like `... WHERE a = 50 AND md5(b) LIKE '%d77%'`, the optimizer will tell DuckDB to evaluate `a = 50` on every column first. If the value in column `a` passes the check `a = 50`, DuckDB will evaluate the `md5` hash for the values in column `b`.

Conclusion

A well-written optimizer can provide significant performance improvements when allowed to optimize freely. Not only can the optimizer apply the many optimization rules a human might naturally miss, an optimizer can respond to changes in the data. Some optimizations can result in a performance improvement of 100×, which might be the difference when deciding to use analytical system *A* vs. analytical system *B*. With DuckDB, all optimization rules are applied automatically to every query, so you can continually enjoy the benefits. Hopefully this blog post has convinced you to consider the optimizer next time you hear about the next database that has everyone's ears burning.

Runtime-Extensible SQL Parsers Using PEG

Publication date: 2024-11-22

Authors: Hannes Mühleisen and Mark Raasveldt

TL;DR: Despite their central role in processing queries, parsers have not received any noticeable attention in the data systems space. State-of-the art systems are content with ancient old parser generators. These generators create monolithic, inflexible and unforgiving parsers that hinder innovation in query languages and frustrate users. Instead, parsers should be rewritten using modern abstractions like Parser Expression Grammars (PEG), which allow dynamic changes to the accepted query syntax and better error recovery. In this post, we discuss how parsers could be re-designed using PEG, and validate our recommendations using experiments for both effectiveness and efficiency.

> This post is a shortened version of our peer-reviewed research paper "Runtime-Extensible Parsers" that was accepted for publication and presentation at the 2025 Conference on Innovative Data Systems Research (CIDR) that is going to be held in Amsterdam between January 19 and 22, 2025. You can read the full paper if you prefer.

The parser is the DBMS component that is responsible for turning a query in string format into an internal representation which is usually tree-shaped. The parser defines which queries are going to be accepted at all. Every single SQL query starts its journey in a parser. Despite its prominent position in the stack, very little research has been published on parsing queries for data management systems. There seems to have been very little movement on the topic in the past decades and their implementations are largely stuck in sixty-year-old abstractions and technologies.

The constant growth of the SQL specification with niche features (e.g., support for graph queries in SQL/PGQ or XML support) as well as the desire to support alternative query notations like dplyr, piped SQL, PRQL or SaneQL makes monolithic parsers less and less practical: in their traditional design, parser construction is a *compile-time* activity where enormous grammar files are translated into state machine transition lookup tables which are then baked in a system binary. Having those *always* be present in the parser might be wasteful especially for size-conscious binary distributions like WebAssembly (Wasm).

Many if not most SQL systems use a static parser created using a YACC-style parser toolkit: we are able to easily confirm this for open-source systems like PostgreSQL and MySQL/MariaDB. From analyzing their binaries' symbol names, we also found indications that Oracle, SQL Server and IBM Db2 use YACC. Internally, YACC and its slightly more recent variant GNU Bison as well as the "Lemon" parser generator used by SQLite all use a "single look-ahead left-to-right rightmost derivation" LALR(1) parser generator. This generator translates a formal context-free set of grammar rules in Extended Backus-Naur Form (EBNF) to a parser state machine. LALR parsers are a more space-efficient specialization of LR(k) parsers as first described by Knuth. But in effect, **the most advanced SQL systems of 2024 use parser technology from the 1960s**. Given that the rest of data management systems have been greatly overhauled since this should raise the question of why the parser did not receive any serious engineering attention.

Database systems are moving towards becoming *ecosystems* instead of pre-built monoliths. Much of the innovation in the PostgreSQL, SQLite, and DuckDB communities now comes from extensions, which are shared libraries that are loaded into the database system at run-time to extend the database system with features like vector similarity search, geospatial support, file systems, or graph processing. Bundling all those features upfront would be difficult due to additional binary size, external dependencies. In addition, they are often maintained independently by their communities. Thus far, at least in part due to the ubiquity of YACC-style parsers, those community extensions have been restricted from extending syntax. While this is also true in other ecosystems like Python, the design of SQL with its heavy focus on syntax and not function calls makes the extensions second-class citizens that have to somehow work around the restrictions by the original parser, e.g., by embedding custom expressions in strings.

We propose to *re-think data management system parser design* to create modern, *extensible* parsers, which allow a dynamic configuration of the accepted syntax *at run-time*, for example to allow syntax extensions, new statements, or to add entirely new query languages. This would allow to break up the monolithic grammars currently in use and enable more creativity and flexibility in what syntax a data management system can accept, both for industrial and research use. Extensible parsers allow for new grammar features to be easily integrated and tested, and can also help bridge the gap between different SQL dialects by adding support for the dialect of one system to the parser

of another. Conversely, it might also be desirable in some use cases to *restrict* the acceptable grammar, e.g., to restrict the complexity of queries, or to enforce strict compliance with the SQL standard.

Modernizing parser infrastructure also has additional benefits: one of the most-reported support issues with data management systems are unhelpful syntax errors. Some systems go to great lengths to try to provide a meaningful error message, e.g., `this column does not exist, did you mean ...`, but this is typically limited to resolving identifiers following the actual parsing. YACC-style parsers exhibit "all-or-nothing" behavior, the *entire* query or set of queries either is accepted entirely or not at all. This is why queries with actual syntactical errors (e.g., `SELEXT` instead of `SELECT` are usually harshly rejected by a DBMS. MySQL for example is notorious for its unhelpful error messages:

```
You have an error in your SQL syntax; check the manual that corresponds
to your MySQL server version for the right syntax to use near 'SELEXT'
at line 1.
```

Parsing Expression Grammar

Parsing Expression Grammar (PEG) parsers represent a more modern approach to parsing. PEG parsers are top-down parsers that effectively generate a recursive-descent style parser from a grammar. Through the "packrat" memoization technique PEG parsers exhibit linear time complexity in parsing at the expense of a grammar-dependent amount of extra memory. The biggest difference from a grammar author perspective is the choice operator where multiple syntax options can be matched. In LALR parsers options with similar syntax can create ambiguity and reduce conflicts. In PEG parsers the *first* matching option is always selected. Because of this, PEG parsers cannot be ambiguous by design.

As their name suggests, parsing expression grammar consists of a set of *parsing expressions*. Expressions can contain references to other rules, or literal token references, both as actual strings or character classes similar to regular expressions. Expressions can be combined through sequences, quantifiers, optionals, groupings and both positive and negative look-ahead. Each expression can either match or not, but it is required to consume a part of the input if it matches. Expressions are able to look ahead and consider the remaining input but are not required to consume it. Lexical analysis is typically part of the PEG parser itself, which removes the need for a separate step.

One big advantage is that PEG parsers *do not require a compilation step* where the grammar is converted to for example a finite state automaton based on lookup tables. PEG can be executed directly on the input with minimal grammar transformation, making it feasible to re-create a parser at runtime. PEG parsers are gaining popularity, for example, the Python programming language has recently switched to a PEG parser.

Another big advantage of PEG parsers is *error handling*: the paper "Syntax Error Recovery in Parsing Expression Grammars" describes a practical technique where parser rules are annotated with "recovery" actions, which can (1) show more than a single error and (2) annotate errors with a more meaningful error message.

A possible disadvantage of memoized packrat parsing is the memory required for memoization: the amount required is *proportional to the input size*, not the stack size. Of course, memory limitations have relaxed significantly since the invention of LALR parsers sixty years ago and queries typically are not "Big Data"‘ themselves.

Proof-of-Concept Experiments

To perform experiments on parser extensibility, we have implemented an – admittedly simplistic – experimental prototype PEG parser for enough of SQL to parse *all* the TPC-H and TPC-DS queries. This grammar is compatible with the `cpp-peglib` single-header C++17 PEG execution engine.

`cpp-peglib` uses a slightly different grammar syntax, where `/` is used to denote choices. The symbol `?` shows an optional element, and `*` defines arbitrary repetition. The special rules `Parens()` and `List()` are grammar macros that simplify the grammar for common elements. The special `%whitespace` rule is used to describe tokenization.

Below is an abridged version of our experimental SQL grammar, with the `Expression` and `Identifier` syntax parsing rules omitted for brevity:

```
Statements <- SingleStmt (';' SingleStmt )* ';'*
SingleStmt <- SelectStmt
SelectStmt <- SimpleSelect (SetopClause SimpleSelect)*
SetopClause <-
    ('UNION' / 'EXCEPT' / 'INTERSECT') 'ALL'?
SimpleSelect <- WithClause? SelectClause FromClause?
    WhereClause? GroupByClause? HavingClause?
    OrderByClause? LimitClause?
WithStatement <- Identifier 'AS' SubqueryReference
WithClause <- 'WITH' List(WithStatement)
SelectClause <- 'SELECT' ('*' / List(AliasExpression))
ColumnsAlias <- Parens(List(Identifier))
TableReference <-
    (SubqueryReference 'AS'? Identifier ColumnsAlias?) /
    (Identifier ('AS'? Identifier)?)
ExplicitJoin <- ('LEFT' / 'FULL')? 'OUTER'?
    'JOIN' TableReference 'ON' Expression
FromClause <- 'FROM' TableReference
    ((',' TableReference) / ExplicitJoin)*
WhereClause <- 'WHERE' Expression
GroupByClause <- 'GROUP' 'BY' List(Expression)
HavingClause <- 'HAVING' Expression
SubqueryReference <- Parens(SelectStmt)
OrderByExpression <- Expression ('DESC' / 'ASC')?
    ('NULLS' 'FIRST' / 'LAST')?
OrderByClause <- 'ORDER' 'BY' List(OrderByExpression)
LimitClause <- 'LIMIT' NumberLiteral
AliasExpression <- Expression ('AS'? Identifier)?
%whitespace <- [ \t\n\r]*
List(D) <- D (',' D)*
Parens(D) <- '(' D ')'
```

All experiments were run on a 2021 MacBook Pro with the M1 Max CPU and 64 GB of RAM. The experimental grammar and the code for experiments are available on GitHub.

Loading the base grammar from its text representation into the `cpp-peglib` grammar dictionary with symbolic rule representations takes 3 ms. In case that delay should become an issue, the library also allows to define rules programmatically instead of as strings. It would be straightforward to pre-compile the grammar file into source code for compilation, YACC-style. While somewhat counter-intuitive, it would reduce the time required to initialize the initial, unmodified parser. This difference matters for some applications of e.g., DuckDB where the database instance only lives for a few short milliseconds.

For the actual parsing, YACC parses TPC-H Query 1 in ca. 0.03 ms, where `cpp-peglib` takes ca. 0.3 ms, a ca. 10 times increase. To further stress parsing performance, we repeated all TPC-H and TPC-DS queries six times to create a 36,840 line SQL script weighing in at ca. 1 MB. Note that a recent study has found that the 99-percentile of read queries in the Amazon Redshift cloud data warehouse are smaller than 16.5 kB.

Postgres takes on average 24 ms to parse this file using YACC. Note that this time includes the execution of grammar actions that create Postgres' parse tree. `cpp-peglib` takes on average 266 ms to parse the test file. However, our experimental parser does not have grammar actions defined yet. When simulating actions by generating default AST actions for every rule, parsing time increases to 339 ms. Note that the AST generation is more expensive than required, because a node is created for each matching rule, even if there is no semantic meaning in the grammar at hand.

Overall, we can observe a ca. 10 times slowdown in parsing performance when using the `cpp-peglib` parser. However, it should be noted that the *absolute duration* of those two processes is still tiny; at least for analytical queries, sub-millisecond parsing time is more than acceptable as parsing still only accounts for a tiny fraction of overall query processing time. Furthermore, there are still ample optimization opportunities in the experimental parsers we created using an off-the-shelf PEG library. For example, the library makes heavy use of recursive function calls, which can be optimized e.g., by using a loop abstraction.

In the following, we present some experiments in extending the prototype parser with support for new statements, entirely new syntax and with improvements in error messages.

> It is already possible to replace DuckDB's parser by providing an alternative parser. Several community extensions such as `duckpgq`, `prql` and `psql` use this approach. When trying to parse a query string, DuckDB first attempts to use the default parser. If this fails, it switches to the extension parsers as failover. Therefore, these extensions cannot simply extend the parser with a few extra rules – instead, they implement the complete grammar of their target language.

Adding the `UNPIVOT` Statement

Let's assume we would want to add a new top-level `UNPIVOT` statement to turn columns into rows to a SQL dialect. `UNPIVOT` should work on the same level as e.g., `SELECT`, for example to unpivot a table `t1` on a specific list of columns or all columns (`*`), we would like to be able to write:

```
UNPIVOT t1 ON (c1, c2, c3);
UNPIVOT t1 ON (*);
```

It is clear that we would have to somehow modify the parser to allow this new syntax. However, when using a YACC parser, this would require modifying the grammar, re-running the parser generator, hoping for the absence of shift-reduce conflicts, and then recompiling the actual database system. However, this is not practical at run-time which is when extensions are loaded, ideally within milliseconds.

In order to add `UNPIVOT`, we have to define a grammar rule and then modify `SingleStmt` to allow the statement in a global sequence of SQL statements. This is shown below. We define the new `UnpivotStatement` grammar rule by adding it to the dictionary, and we then modify the `SingleStmt` rule entry in the dictionary to also allow the new statement.

```
UnpivotStatement <- 'UNPIVOT' Identifier
    'ON' Parens(List(Identifier) / '*')

SingleStmt <- SelectStatement / UnpivotStatement
```

Note that we re-use other machinery from the grammar like the `Identifier` rule as well as the `Parens()` and `List()` macros to define the `ON` clause. The rest of the grammar dictionary remains unchanged. After modification, the parser can be re-initialized in another 3 ms. Parser execution time was unaffected.

Extending `SELECT` with `GRAPH_TABLE`

Let's now assume we would want to modify the `SELECT` syntax to add support for SQL/PGQ graph matching patterns. Below is an example query in SQL/PGQ that finds the university name and year for all students called Bob:

```
SELECT study.classYear, study.name
FROM GRAPH_TABLE (pg,
    MATCH
        (a:Person WHERE a.firstName = 'Bob')-[s:studyAt]->(u:University)
        COLUMNS (s.classYear, u.name)
) study;
```

We can see that this new syntax adds the `GRAPH_TABLE` clause and the pattern matching domain-specific language (DSL) within. To add support for this syntax to a SQL parser at runtime, we need to modify the grammar for the `SELECT` statement itself. This is fairly straightforward when using a PEG. We replace the rule that describes the `FROM` clause to also accept a sub-grammar starting at the `GRAPH_TABLE` keyword following by parentheses. Because the parser does not need to generate a state machine, we are immediately able to accept the new syntax.

Below we show a small set of grammar rules that are sufficient to extend our experimental parser with support for the SQL/PGQ `GRAPH_TABLE` clause and the containing property graph patterns. With this addition, the parser can parse the query above. Parser construction and parser execution timings were unaffected.

```
Name <- (Identifier? ':' Identifier) / Identifier
Edge <- ('-' / '<-') '[' Name ']' ('->' / '-')
Pattern <- Parens(Name WhereClause?) Edge
   Parens(Name WhereClause?)
```

```
PropertyGraphReference <- 'GRAPH_TABLE'i '('
        Identifier ','
        'MATCH'i List(Pattern)
        'COLUMNS'i Parens(List(ColumnReference))
    ')' Identifier?

TableReference <-
    PropertyGraphReference / ...
```

`dplyr`, the "Grammar of Data Manipulation", is the de facto standard data transformation language in the R Environment for Statistical Computing. The language uses function calls and a special chaining operator (%>%) to combine operators. Below is an example dplyr query:

```
df %>%
  group_by(species) %>%
  summarise(
    n = n(),
    mass = mean(mass, na.rm = TRUE)
  ) %>%
  filter(n > 1, mass > 50)
```

For those unfamiliar with dplyr, the query is equivalent to this SQL query:

```
SELECT * FROM (
    SELECT count(*) AS n, AVG(mass) AS mass
        FROM df
        GROUP BY species)
    WHERE n > 1 AND mass > 50;
```

With an extensible parser, it is feasible to add support for completely new query languages like `dplyr` to a SQL parser. Below is a simplified grammar snippet that enables our SQL parser to accept the `dplyr` example from above.

```
DplyrStatement <- Identifier Pipe Verb (Pipe Verb)*
Verb <- VerbName Parens(List(Argument))
VerbName <- 'group_by' / 'summarise' / 'filter'
Argument <- Expression / (Identifier '=' Expression)
Pipe <- '%>%'

SingleStmt <- SelectStatement /
    UnpivotStatement / DplyrStatement
```

It is important to note that the rest of the experimental SQL parser *still works*, i.e., the `dplyr` syntax now *also* works. Parser construction and parser execution timings were again unaffected.

Better Error Messages

As mentioned above, PEG parsers are able to generate better error messages elegantly. A common novice SQL user mistake is to mix up the order of keywords in a query, for example, the `ORDER BY` must come after the `GROUP BY`. Assume an inexperienced user types the following query:

```
SELECT customer, SUM(sales)
FROM revenue
ORDER BY customer
GROUP BY customer;
```

By default, both the YACC and the PEG parsers will report a similar error message about an `unexpected 'GROUP' keyword` with a byte position. However, with a PEG parser we can define a "recovery" syntax rule that will create a useful error message. We modify the `OrderByClause` from our experimental grammar like so:

```
OrderByClause <- 'ORDER'i 'BY'i List(OrderByExpression)
    %recover(WrongGroupBy)?
WrongGroupBy <- GroupByClause
    { error_message "GROUP BY must precede ORDER BY" }
```

Here, we use the `%recover` construct to match a misplaced `GROUP BY` clause, re-using the original definition, and then trigger a custom error message that advises the user on how to fix their query. And indeed, when we parse the wrong SQL example, the parser will output the custom message.

Conclusion and Future Work

In this post, we have proposed to modernize the ancient art of SQL parsing using more modern parser generators like PEG. We have shown how by using PEG, a parser can be extended at run-time at minimal cost without re-compilation. In our experiments we have demonstrated how minor grammar adjustments can fundamentally extend and change the accepted syntax.

An obvious next step is to address the observed performance drawback observed in our prototype. Using more efficient implementation techniques, it should be possible to narrow the gap in parsing performance between YACC-based LALR parsers and a dynamic PEG parser. Another next step is to address some detail questions for implementation: for example, parser extension load order should ideally not influence the final grammar. Furthermore, while parser actions can in principle execute arbitrary code, they may have to be restrictions on return types and input handling.

We plan to switch DuckDB's parser, which started as a fork of the Postgres YACC parser, to a PEG parser in the near future. As an initial step, we have performed an experiment where we found that it is possible to interpret the current Postgres YACC grammar with PEG. This should greatly simplify the transitioning process, since it ensures that the same grammar will be accepted in both parsing frameworks.

Acknowledgments

We would like to thank **Torsten Grust**, **Gábor Szárnyas** and **Daniël ten Wolde** for their valuable suggestions. We would also like to thank **Carlo Piovesan** for his translation of the Postgres YACC grammar to PEG.

DuckDB Tricks – Part 3

Publication date: 2024-11-29

Authors: Andra Ionescu and Gabor Szarnyas

TL;DR: In this new installment of the DuckDB Tricks series, we present features for convenient handling of tables and performance optimization tips for Parquet and CSV files.

Overview

We continue our DuckDB Tricks series with a third part, where we showcase friendly SQL features and performance optimizations.

Operation	SQL instructions
Excluding columns from a table	`EXCLUDE`{:.language-sql .highlight}/`COLUMNS(...)`{:.language-sql .highlight}, `NOT SIMILAR TO`{:.language-sql .highlight}
Renaming columns with pattern matching	`COLUMNS(...) AS ...`{:.language-sql .highlight}
Loading with globbing	`FROM '*.csv'`{:.language-sql .highlight}
Reordering Parquet files	`COPY (FROM ... ORDER BY ...) TO ...`{:.language-sql .highlight}
Hive partitioning	`hive_partitioning = true`{:.language-sql .highlight}

Dataset

We'll use a subset of the Dutch railway services dataset, which was already featured in a blog post earlier this year. This time, we'll use the CSV files between January and October 2024: `services-2024-01-to-10.zip`. If you would like to follow the examples, download and decompress the data set before proceeding.

Excluding Columns from a Table

First, let's look at the data in the CSV files. We pick the CSV file for August and inspect it with the `DESCRIBE` statement.

```
DESCRIBE FROM 'services-2024-08.csv';
```

The result is a table with the column names and the column types.

column_name	column_type	null	key	default	extra
Service:RDT-ID	BIGINT	YES	NULL	NULL	NULL
Service:Date	DATE	YES	NULL	NULL	NULL
Service:Type	VARCHAR	YES	NULL	NULL	NULL
Service:Company	VARCHAR	YES	NULL	NULL	NULL

column_name	column_type	null	key	default	extra
Service:Train number	BIGINT	YES	NULL	NULL	NULL
...	...	...	...	...	...

Now, let's use `SUMMARIZE` to inspect some statistics about the columns.

```
SUMMARIZE FROM 'services-2024-08.csv';
```

With `SUMMARIZE`, we get 10 statistics about our data (`min`, `max`, `approx_unique`, etc.). If we want to remove a few of them the result, we can use the `EXCLUDE` modifier. For example, to exclude `min`, `max` and the quantiles `q25`, `q50`, `q75`, we can use issue the following command:

```
SELECT * EXCLUDE(min, max, q25, q50, q75)
FROM (SUMMARIZE FROM 'services-2024-08.csv');
```

Alternatively, we can use the `COLUMNS` expression with the `NOT SIMILAR TO` operator. This works with a regular expression:

```
SELECT COLUMNS(c -> c NOT SIMILAR TO 'min|max|q.*')
FROM (SUMMARIZE FROM 'services-2024-08.csv');
```

In both cases, the resulting table will contain the 5 remaining statistical columns:

column_name	column_type	approx_unique	avg	std	count	null_percentage
Service:RDT-ID	BIGINT	259022	14200071.03736433	59022.836209662266	1846574	0.00
Service:Date	DATE	32	NULL	NULL	1846574	0.00
Service:Type	VARCHAR	20	NULL	NULL	1846574	0.00
Service:Company	VARCHAR	12	NULL	NULL	1846574	0.00
Service:Train number	BIGINT	17264	57781.81688196628	186353.76365744913	1846574	0.00
...	...	...	...	...	...	...

Renaming Columns with Pattern Matching

Upon inspecting the columns, we see that their names contain spaces and semicolons (`:`). These special characters makes writing queries a bit tedious as they necessitate quoting column names with double quotes. For example, we have to write `"Service:Company"` in the following query:

```
SELECT DISTINCT "Service:Company" AS company,
FROM 'services-2024-08.csv'
ORDER BY company;
```

Let's see how we can rename the columns using the `COLUMNS` expression. To replace the special characters (up to 2), we can write the following query:

```
SELECT COLUMNS('(.*?)_*$') AS "\1"
FROM (
    SELECT COLUMNS('(\w*)\W*(\w*)\W*(\w*)') AS "\1_\2_\3"
    FROM 'services-2024-08.csv'
);
```

Add `DESCRIBE` at the beginning of the query and we can see the renamed columns:

column_name	column_type	null	key	default	extra
Service_RDT_ID	BIGINT	YES	NULL	NULL	NULL
Service_Date	DATE	YES	NULL	NULL	NULL
Service_Type	VARCHAR	YES	NULL	NULL	NULL
Service_Company	VARCHAR	YES	NULL	NULL	NULL
Service_Train_number	BIGINT	YES	NULL	NULL	NULL
...	...	...	...	...	...

Let's break down the query starting with the first `COLUMNS` expression:

```
SELECT COLUMNS('(\w*)\W*(\w*)\W*(\w*)') AS "\1_\2_\3"
```

Here, we use regular expression with `(\w*)` groups that capture 0...n word characters (`[0-9A-Za-z_]`). Meanwhile, the expression `\W*` captures 0...n non-word characters (`[^0-9A-Za-z_]`). In the alias part we refer to the capture group `i` with `\i` so `"\1_\2_\3"` means that we only keep the word characters and separate their groups with underscores (`_`). However, because some column names contain words separated by a space, while others don't, after this `SELECT` statement we get column names with a trailing underscore (`_`), e.g., `Service_Date_`. Thus, we need an additional processing step:

```
SELECT COLUMNS('(.*?)_*$') AS "\1"
```

Here, we capture the group of characters without the trailing underscore(s) and rename the columns to `\1`, which removes the trailing underscores.

To make writing queries even more convenient, we can rely on the case-insensitivity of identifiers to query the column names in lowercase:

```
SELECT DISTINCT service_company
FROM (
    SELECT COLUMNS('(.*?)_*$') AS "\1"
    FROM (
       SELECT COLUMNS('(\w*)\W*(\w*)\W*(\w*)') AS "\1_\2_\3"
       FROM 'services-2024-08.csv'
    )
)
ORDER BY service_company;
```

Service_Company
Arriva
Blauwnet
Breng
DB
Eu Sleeper
...

> The returned column name preserves its original cases even though we used lowercase letters in the query.

Loading with Globbing

Now that we can simplify the column names, let's ingest all 3 months of data to a table:

```
CREATE OR REPLACE TABLE services AS
    SELECT COLUMNS('(.*?)_*$') AS "\1"
    FROM (
        SELECT COLUMNS('(\w*)\W*(\w*)\W*(\w*)') AS "\1_\2_\3"
        FROM 'services-2024-*.csv'
    );
```

In the inner FROM clause, we use the * glob syntax to match all files. DuckDB automatically detects that all files have the same schema and unions them together. We have now a table with all the data from January to October, amounting to almost 20 million rows.

Reordering Parquet Files

Suppose we want to analyze the average delay of the Intercity Direct trains operated by the Nederlandse Spoorwegen (NS), measured at the final destination of the train service. While we can run this analysis directly on the `.csv` files, the lack of metadata (such as schema and min-max indexes) will limit the performance. Let's measure this in the CLI client by turning on the timer:

```
.timer on
```

```
SELECT avg("Stop:Arrival delay")
FROM 'services-*.csv'
WHERE "Service:Company" = 'NS'
  AND "Service:Type" = 'Intercity direct'
  AND "Stop:Departure time" IS NULL;
```

This query takes about 1.8 seconds. Now, if we run the same query on `services` table that's already loaded to DuckDB, the query is much faster:

```
SELECT avg(Stop_Arrival_delay)
FROM services
WHERE Service_Company = 'NS'
  AND Service_Type = 'Intercity direct'
  AND Stop_Departure_time IS NULL;
```

The run time is about 35 milliseconds.

If we would like to use an external binary file format, we can also export the database to a single Parquet file:

```
EXPORT DATABASE 'railway' (FORMAT parquet);
```

We can then directly query it as follows:

```
SELECT avg(Stop_Arrival_delay)
FROM 'railway/services.parquet'
WHERE Service_Company = 'NS'
  AND Service_Type = 'Intercity direct'
  AND Stop_Departure_time IS NULL;
```

The runtime for this format is about 90 milliseconds – somewhat slower than DuckDB's own file format but about 20× faster than reading the raw CSV files.

If we have a priori knowledge of the fields a query filters on, we can reorder the Parquet file to improve query performance.

```
COPY
(FROM 'railway/services.parquet' ORDER BY Service_Company, Service_Type)
TO 'railway/services.parquet';
```

If we run the query again, it's noticeably faster, taking only 35 milliseconds. This is thanks to partial reading, which uses the zonemaps (min-max indexes) to limit the amount of data that has to be scanned. Reordering the file allows DuckDB to skip more data, leading to faster query times.

Hive Partitioning

To speed up queries even further, we can use Hive partitioning to create a directory layout on disk that matches the filtering used in the queries.

```
COPY services
TO 'services-parquet-hive'
(FORMAT parquet, PARTITION_BY (Service_Company, Service_Type));
```

Let's peek into the directory from DuckDB's CLI using the `.sh` dot command:

```
.sh tree services-parquet-hive
```

```
services-parquet-hive
├── Service_Company=Arriva
│   ├── Service_Type=Extra%20trein
│   │   └── data_0.parquet
│   ├── Service_Type=Nachttrein
│   │   └── data_0.parquet
│   ├── Service_Type=Snelbus%20ipv%20trein
│   │   └── data_0.parquet
│   ├── Service_Type=Sneltrein
│   │   └── data_0.parquet
│   ├── Service_Type=Stopbus%20ipv%20trein
│   │   └── data_0.parquet
│   ├── Service_Type=Stoptrein
│   │   └── data_0.parquet
│   └── Service_Type=Taxibus%20ipv%20trein
│       └── data_0.parquet
├── Service_Company=Blauwnet
│   ├── Service_Type=Intercity
│   │   └── data_0.parquet
...
```

We can now run the query on the Hive partitioned data set by passing the `hive_partitioning = true` flag:

```
SELECT avg(Stop_Arrival_delay)
FROM read_parquet(
         'services-parquet-hive/**/*.parquet',
         hive_partitioning = true
     )
WHERE Service_Company = 'NS'
  AND Service_Type = 'Intercity direct'
  AND Stop_Departure_time IS NULL;
```

This query now takes about 20 milliseconds as DuckDB can use the directory structure to limit the reads even further. And the neat thing about Hive partitioning is that it even works with CSV files!

```
COPY services
TO 'services-csv-hive'
(FORMAT csv, PARTITION_BY (Service_Company, Service_Type));

SELECT avg(Stop_Arrival_delay)
FROM read_csv('services-csv-hive/**/*.csv', hive_partitioning = true)
WHERE Service_Company = 'NS'
  AND Service_Type = 'Intercity direct'
  AND Stop_Departure_time IS NULL;
```

While the CSV files lack any sort of metadata, DuckDB can rely on the directory structure to limit the scans to the relevant directories, resulting in execution times around 150 milliseconds, more than 10× faster compared to reading all CSV files.

If all these formats and results got your head spinning, no worries. We got your covered with this summary table:

Format	Query runtime (ms)
DuckDB file format	35
CSV (vanilla)	1800
CSV (Hive-partitioned)	150
Parquet (vanilla)	90
Parquet (reordered)	35
Parquet (Hive-partitioned)	20

Oh, and we forgot to report the result. The average delay of Intercity Direct trains is 3 minutes!

Closing Thoughts

That's it for part three of DuckDB tricks. If you have a trick that would like to share, please share it with the DuckDB team on our social media sites, or submit it to the DuckDB Snippets site (maintained by our friends at MotherDuck).

CSV Files: Dethroning Parquet as the Ultimate Storage File Format — or Not?

Publication date: 2024-12-05

Author: Pedro Holanda

TL;DR: Data analytics primarily uses two types of storage format files: human-readable text files like CSV and performance-driven binary files like Parquet. This blog post compares these two formats in an ultimate showdown of performance and flexibility, where there can be only one winner.

File Formats

CSV Files

Data is most commonly stored in human-readable file formats, like JSON or CSV files. These file formats are easy to operate on, since anyone with a text editor can simply open, alter, and understand them.

For many years, CSV files have had a bad reputation for being slow and cumbersome to work with. In practice, if you want to operate on a CSV file using your favorite database system, you must follow this recipe:

1. Manually discover its schema by opening the file in a text editor.
2. Create a table with the given schema.
3. Manually figure out the dialect of the file (e.g., which character is used for a quote?)
4. Load the file into the table using a COPY statement and with the dialect set.
5. Start querying it.

Not only is this process tedious, but parallelizing a CSV file reader is far from trivial. This means most systems either process it single-threaded or use a two-pass approach.

Additionally, CSV files are wild: although RFC-4180 exists as a CSV standard, it is commonly ignored. Systems must therefore be sufficiently robust to handle these files as if they come straight from the wild west.

Last but not least, CSV files are wasteful: data is always laid out as strings. For example, numeric values like `1000000000` take 10 bytes instead of 4 bytes if stored as an `int32`. Additionally, since the data layout is row-wise, opportunities to apply lightweight columnar compression are lost.

Parquet Files

Due to these shortcomings, performance-driven file formats like Parquet have gained significant popularity in recent years. Parquet files cannot be opened by general text editors, cannot be easily edited, and have a rigid schema. However, they store data in columns, apply various compression techniques, partition the data into row groups, maintain statistics about these row groups, and define their schema directly in the file.

These features make Parquet a monolith of a file format — highly inflexible but efficient and fast. It is easy to read data from a Parquet file since the schema is well-defined. Parallelizing a scanner is straightforward, as each thread can independently process a row group. Filter pushdown is also simple to implement, as each row group contains statistical metadata, and the file sizes are very small.

The conclusion should be simple: if you have small files and need flexibility, CSV files are fine. However, for data analysis, one should pivot to Parquet files, right? Well, this pivot may not be a hard requirement anymore – read on to find out why!

Reading CSV Files in DuckDB

For the past few releases, DuckDB has doubled down on delivering not only an easy-to-use CSV scanner but also an extremely performant one. This scanner features its own custom CSV sniffer, parallelization algorithm, buffer manager, casting mechanisms, and state machine-based parser.

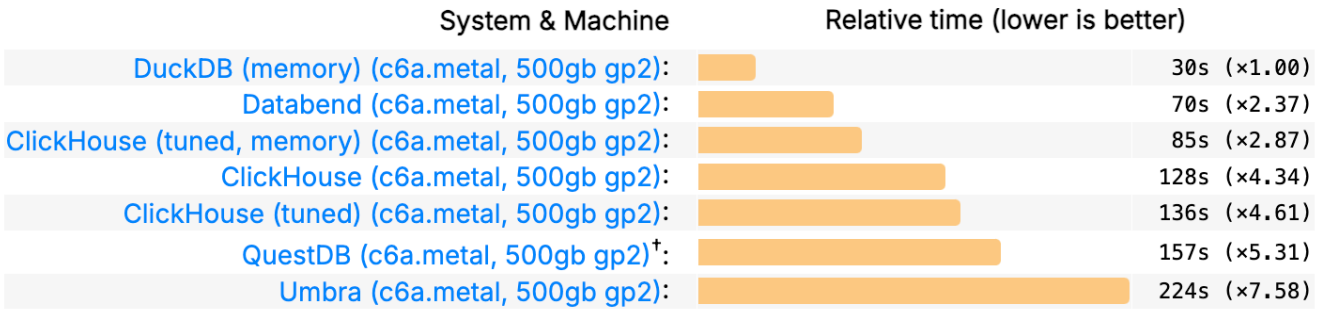
For usability, the previous paradigm of manual schema discovery and table creation has been changed. Instead, DuckDB now utilizes a CSV Sniffer, similar to those found in dataframe libraries like Pandas. This allows for querying CSV files as easily as:

```
FROM 'path/to/file.csv';
```

Or tables to be created from CSV files, without any prior schema definition with:

```
CREATE TABLE t AS FROM 'path/to/file.csv';
```

Furthermore, the reader became one of the fastest CSV readers in analytical systems, as can be seen by the load times of the latest iteration of ClickBench. In this benchmark, the data is loaded from an 82 GB uncompressed CSV file into a database table.



ClickBench CSV loading times (2024-12-05)

Comparing CSV and Parquet

With the large boost in usability and performance for the CSV reader, one might ask: what is the actual difference in performance when loading a CSV file compared to a Parquet file into a table? Additionally, how do these formats differ when running queries directly on them?

To find out, we will run a few examples using both CSV and Parquet files containing TPC-H data to shed light on their differences. All scripts used to generate the benchmarks of this blogpost can be found in a repository.

Usability

In terms of usability, scanning CSV files and Parquet files can differ significantly.

In simple cases, where all options are correctly detected by DuckDB, running queries on either CSV or Parquet files can be done directly.

```
FROM 'path/to/file.csv';
FROM 'path/to/file.parquet';
```

Things can differ drastically for wild, rule-breaking Arthur Morgan-like CSV files. This is evident from the number of parameters that can be set for each scanner. The Parquet scanner has a total of six parameters that can alter how the file is read. For the majority of cases, the user will never need to manually adjust any of them.

The CSV reader, on the other hand, depends on the sniffer being able to automatically detect many different configuration options. For example: What is the delimiter? How many rows should it skip from the top of the file? Are there any comments? And so on. This results in over 30 configuration options that the user might have to manually adjust to properly parse their CSV file. Again, this number of options is

necessary due to the lack of a widely adopted standard. However, in most scenarios, users can rely on the sniffer or, at most, change one or two options.

The CSV reader also has an extensive error-handling system and will always provide suggestions for options to review if something goes wrong.

To give you an example of how the DuckDB error-reporting system works, consider the following CSV file:

```
Clint Eastwood;94
Samuel L. Jackson
```

In this file, the second line is missing the value for the second column.

```
Invalid Input Error: CSV Error on Line: 2
Original Line: Samuel L. Jackson
Expected Number of Columns: 2 Found: 1
Possible fixes:
* Enable null padding (null_padding=true) to replace missing values with NULL
* Enable ignore errors (ignore_errors=true) to skip this row

  file = western_actors.csv
  delimiter = , (Auto-Detected)
  quote = " (Auto-Detected)
  escape = " (Auto-Detected)
  new_line = \n (Auto-Detected)
  header = false (Auto-Detected)
  skip_rows = 0 (Auto-Detected)
  comment = \0 (Auto-Detected)
  date_format =  (Auto-Detected)
  timestamp_format =  (Auto-Detected)
  null_padding = 0
  sample_size = 20480
  ignore_errors = false
  all_varchar = 0
```

DuckDB provides detailed information about any errors encountered. It highlights the line of the CSV file where the issue occurred, presents the original line, and suggests possible fixes for the error, such as ignoring the problematic line or filling missing values with `NULL`. It also displays the full configuration used to scan the file and indicates whether the options were auto-detected or manually set.

The bottom line here is that, even with the advancements in CSV usage, the strictness of Parquet files make them much easier to operate on.

Of course, if you need to open your file in a text editor or Excel, you will need to have your data in CSV format. Note that Parquet files do have some visualizers, like TAD.

Performance

There are primarily two ways to operate on files using DuckDB:

1. The user creates a DuckDB table from the file and uses the table in future queries. This is a loading process, commonly used if you want to store your data as DuckDB tables or if you will run many queries on them. Also, note that these are the only possible scenarios for most database systems (e.g., Oracle, SQL Server, PostgreSQL, SQLite, ...).
2. One might run a query directly on the file scanner without creating a table. This is useful for scenarios where the user has limitations on memory and disk space, or if queries on these files are only executed once. Note that this scenario is typically not supported by database systems but is common for dataframe libraries (e.g., Pandas).

To fairly compare the scanners, we provide the table schemas upfront, ensuring that the scanners produce the exact same data types. We also set `preserve_insertion_order = false`, as this can impact the parallelization of both scanners, and set `max_temp_directory_size = '0GB'` to ensure no data is spilled to disk, with all experiments running fully in memory.

We use the default writers for both CSV files and Parquet (with the default Snappy compression), and also run a variation of Parquet with `CODEC 'zstd', COMPRESSION_LEVEL 1`, as this can speed up querying/loading times.

For all experiments, we use an Apple M1 Max, with 64 GB RAM. We use TPC-H scale factor 20 and report the median times from 5 runs.

Creating Tables

For creating the table, we focus on the `lineitem` table.

After defining the schema, both files can be loaded with a simple COPY statement, with no additional parameters set. Note that even with the schema defined, the CSV sniffer will still be executed to determine the dialect (e.g., quote character, delimiter character, etc.) and match types and names.

Name	Time (s)	Size (GB)
CSV	11.76	15.95
Parquet Snappy	5.21	3.78
Parquet ZSTD	5.52	3.22

We can see that the Parquet files are definitely smaller. About 5× smaller than the CSV file, but the performance difference is not drastic.

The CSV scanner is only about 2× slower than the Parquet scanner. It's also important to note that some of the cost associated with these operations (~1-2 seconds) is related to the insertion into the DuckDB table, not the scanner itself.

However, it is still important to consider this in the comparison. In practice, the raw CSV scanner is about 3× slower than the Parquet scanner, which is a considerable difference but much smaller than one might initially think.

Directly Querying Files

We will run two different TPC-H queries on our files.

Query 01. First, we run TPC-H Q01. This query operates solely on the `lineitem` table, performing an aggregation and grouping with a filter. It filters on one column and projects 7 out of the 16 columns from `lineitem`.

Therefore, this query will stress the filter pushdown, which is supported by the Parquet reader but not the CSV reader, and the projection pushdown, which is supported by both.

```
SELECT
    l_returnflag,
    l_linestatus,
    sum(l_quantity) AS sum_qty,
    sum(l_extendedprice) AS sum_base_price,
    sum(l_extendedprice * (1 - l_discount)) AS sum_disc_price,
    sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) AS sum_charge,
    avg(l_quantity) AS avg_qty,
    avg(l_extendedprice) AS avg_price,
    avg(l_discount) AS avg_disc,
    count(*) AS count_order
FROM
    lineitem
WHERE
    l_shipdate <= CAST('1996-09-02' AS date)
GROUP BY
    l_returnflag,
    l_linestatus
ORDER BY
    l_returnflag,
    l_linestatus;
```

Name	Time (s)
CSV	6.72
Parquet Snappy	0.88
Parquet ZSTD	0.95

We can see that running this query directly on our file presents a much larger performance gap of approximately 7x compared to simply loading the data into the table. In the Parquet file, we can directly skip row groups that do not match our filter `l_shipdate <= CAST('1996-09-02' AS date)`. Note that this filter, eliminates approximately 30% of the data. Not only that, but we can also skip individual rows that do not match the filter. Additionally, since the Parquet format is column-oriented, we can completely skip any computation on columns that are not projected.

Unfortunately, the CSV reader does not benefit from these filters. Since it lacks partitions, it can't efficiently skip parts of the data. Theoretically, a CSV scanner could skip the computation of rows that do not match a filter, but this is not currently implemented.

Furthermore, the CSV projection skips much of the computation on a column (e.g., it does not cast or copy the value), but it still must parse the value to be able to skip it.

Query 21. Query 21 is a query that not only heavily depends on filter and projection pushdown but also relies significantly on join ordering based on statistics to achieve good performance. In this query, four different files are used and joined together.

```
SELECT
    s_name,
    count(*) AS numwait
FROM
    supplier,
    lineitem l1,
    orders,
    nation
WHERE
    s_suppkey = l1.l_suppkey
    AND o_orderkey = l1.l_orderkey
    AND o_orderstatus = 'F'
    AND l1.l_receiptdate > l1.l_commitdate
    AND EXISTS (
        SELECT
            *
        FROM
            lineitem l2
        WHERE
            l2.l_orderkey = l1.l_orderkey
            AND l2.l_suppkey <> l1.l_suppkey)
    AND NOT EXISTS (
        SELECT
            *
        FROM
            lineitem l3
        WHERE
            l3.l_orderkey = l1.l_orderkey
            AND l3.l_suppkey <> l1.l_suppkey
            AND l3.l_receiptdate > l3.l_commitdate)
    AND s_nationkey = n_nationkey
    AND n_name = 'SAUDI ARABIA'
GROUP BY
    s_name
ORDER BY
    numwait DESC,
    s_name
```

```
LIMIT 100;
```

Name	Time (s)
CSV	19.95
Parquet Snappy	2.08
Parquet ZSTD	2.12

We can see that this query now has a performance difference of approximately 10×. We observe an effect similar to Query 01, but now we also incur the additional cost of performing join ordering with no statistical information for the CSV file.

Conclusion

There is no doubt that the performance of CSV file scanning has drastically increased over the years. If we were to take a guess at the performance difference in table creation a few years ago, the answer would probably have been at least one order of magnitude.

This is excellent, as it allows data to be exported from legacy systems that do not support performance-driven file formats.

But oh boy, don't let super-convenient and fast CSV readers fool you. Your data is still best kept in self-describing, column-binary compressed formats like Parquet — or the DuckDB file format, of course! They are much smaller and more consistent. Additionally, running queries directly on Parquet files is much more beneficial due to efficient projection/filter pushdown and available statistics.

One thing to note is that there exists an extensive body of work on indexing CSV files (i.e., building statistics in a way) to speed up future queries and enable filter pushdown. However, DuckDB does not perform these operations yet.

Bottom line: Parquet is still the undisputed champion for most scenarios, but we will continue working on closing this gap wherever possible.

DuckDB: Running TPC-H SF100 on Mobile Phones

Publication date: 2024-12-06

Authors: Gabor Szarnyas, Laurens Kuiper, Hannes Mühleisen

TL;DR: DuckDB runs on mobile platforms such as iOS and Android, and completes the TPC-H benchmark faster than state-of-the-art research systems on big iron machines 20 years ago.

A few weeks ago, we set out to perform a series of experiments to answer two simple questions:

1. Can DuckDB complete the TPC-H queries on the SF100 data set when running on a new smartphone?
2. If so, can DuckDB complete a run in less than 400 seconds, i.e., faster than the system in the research paper that originally introduced vectorized query processing?

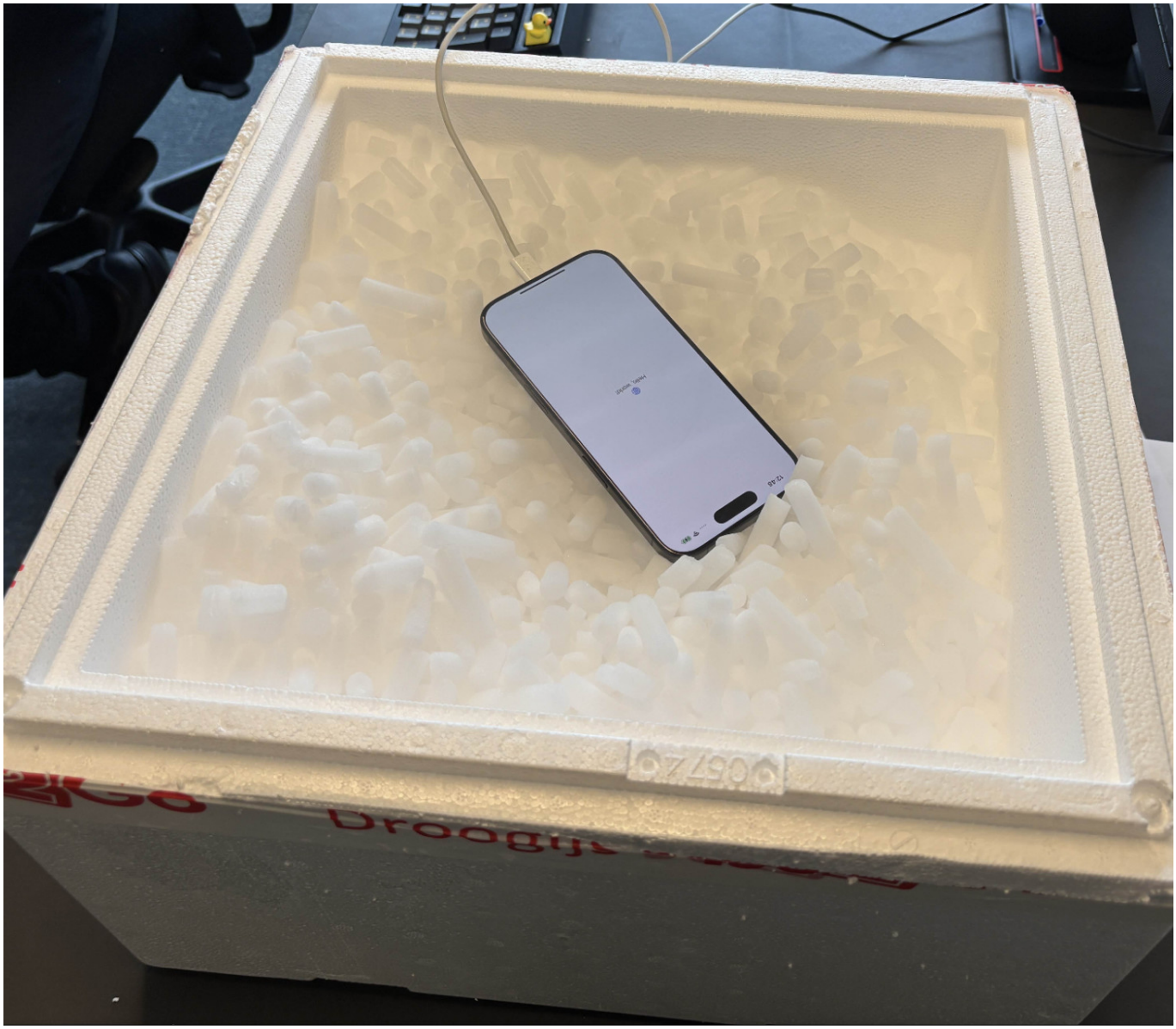
These questions took us on an interesting quest. Along the way, we had a lot of fun and learned the difference between a cold run and a *really cold* run. Read on to find out more.

A Song of Dry Ice and Fire

Our first attempt was to use an iPhone, namely an iPhone 16 Pro. This phone has 8 GB memory and a 6-core CPU with 2 performance cores (running at 4.05 GHz) and 4 efficiency cores (running at 2.42 GHz).

We implemented the application using the DuckDB Swift client and loaded the benchmark on the phone, all 30 GB of it. We quickly found that the iPhone can indeed run the workload without any problems – except that it heated up during the workload. This prompted the phone to perform thermal throttling, slowing down the CPU to reduce heat production. Due to this, DuckDB took 615.1 seconds. Not bad but not enough to reach our goal.

The results got us thinking: what if we improve the cooling of the phone? To this end, we purchased a box of dry ice, which has a temperature below -50 degrees Celsius, and put the phone in the box for the duration of the experiments.



iPhone in a box of dry ice, running TPC-H. Don't try this at home.

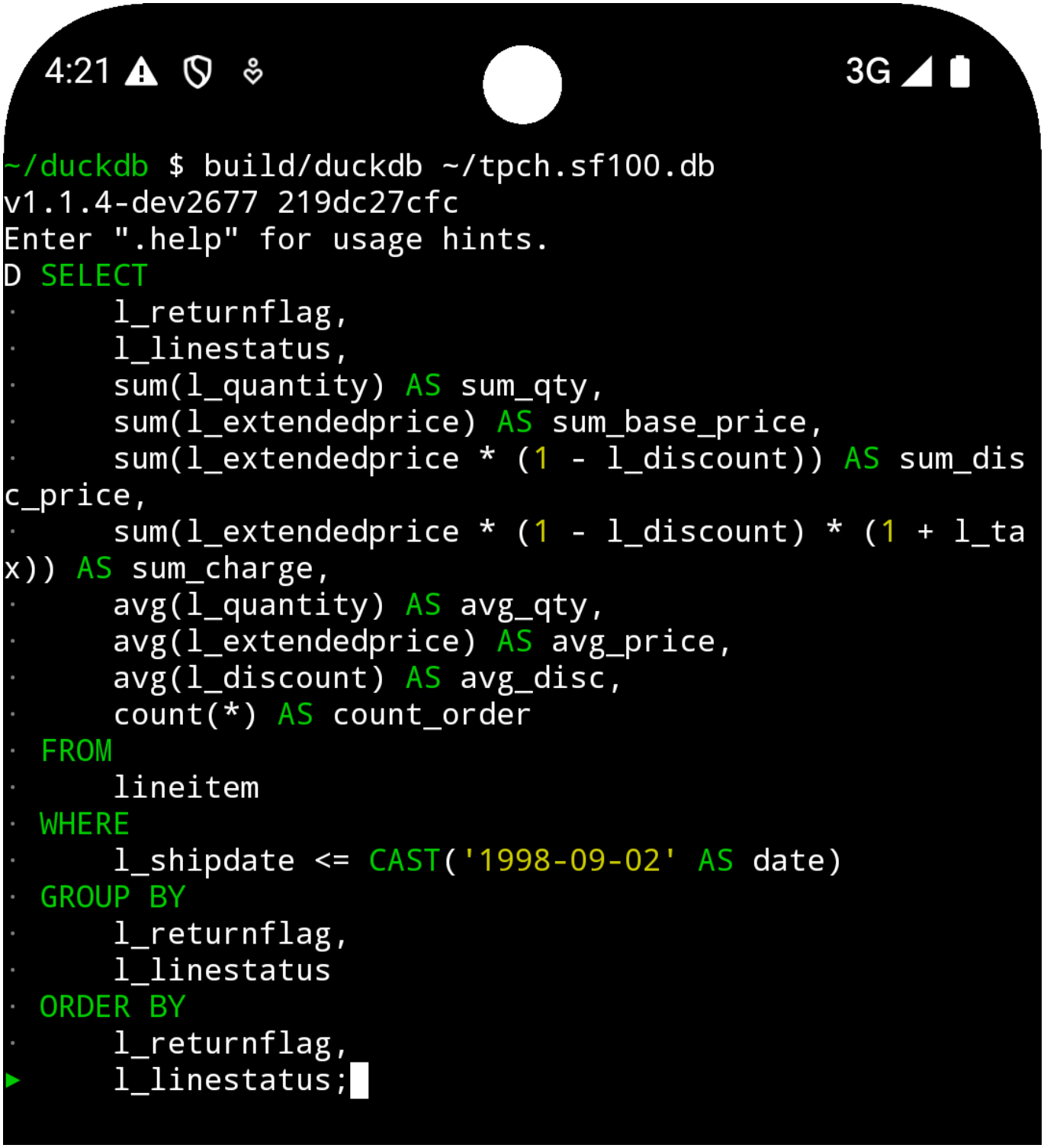
This helped a lot: DuckDB completed in 478.2 seconds. This is a more than 20% improvement – but we still didn't manage to be under 400 seconds.

The phone a few minutes after finishing the benchmark. It no longer booted because the battery was too cold!

Do Androids Dream of Electric Ducks?

In our next experiment, we picked up a Samsung Galaxy S24 Ultra phone, which runs Android 14. This phone is full of interesting hardware. First, it has an 8-core CPU with 4 different core types (1×3.39 GHz, 3×3.10 GHz, 2×2.90 GHz and 2×2.20 GHz). Second, it has a huge amount of RAM – 12 GB to be precise. Finally, its cooling system includes a vapor chamber for improved heat dissipation.

We ran DuckDB in the Termux terminal emulator. We compiled DuckDB CLI client from source following the Android build instructions and ran the experiments from the command line.



DuckDB in Termux, running in the Android emulator

In the end, it wasn't even close. The Android phone completed the benchmark in 235.0 seconds, outperforming our baseline by around 40%.

Never Was a Cloudy Day

The results got us thinking: how do the results stack up among cloud servers? We picked two x86-based cloud instances in AWS EC2 with instance-attached NVMe storage.

The details of these benchmarks are far less interesting than those of the previous ones. We booted up the instances with Ubuntu 24.04 and ran DuckDB in the command line. We found that an `r6id.large` instance (2 vCPUs with 16 GB RAM) completes the queries in 570.8 seconds, which is roughly on-par with an air-cooled iPhone. However, an `r6id.xlarge` (4 vCPUs with 32 GB RAM) completes the benchmark in 166.2 seconds, faster than any result we achieved on phones.

Summary of DuckDB Results

The table contains a summary of the DuckDB benchmark results.

Setup	CPU cores	Memory	Runtime
iPhone 16 Pro (air-cooled)	6	8 GB	615.1 s
iPhone 16 Pro (dry ice-cooled)	6	8 GB	478.2 s
Samsung Galaxy S24 Ultra	8	12 GB	235.0 s
AWS EC2 `r6id.large`	2	16 GB	570.8 s
AWS EC2 `r6id.xlarge`	4	32 GB	166.2 s

Historical Context

So why did we set out to run these experiments in the first place?

Just a few weeks ago, CWI, the birthplace of DuckDB, held a ceremony for the Dijkstra Fellowship. The fellowship was awarded to Marcin Żukowski for his pioneering role in the development of database management systems and his successful entrepreneurial career that resulted in systems such as VectorWise and Snowflake.

A lot of ideas that originate in Marcin's research are used in DuckDB. Most importantly, *vectorized query processing* allows DuckDB to be both fast and portable at the same time. With his co-authors Peter Boncz and Niels Nes, he first described this paradigm in the CIDR 2005 paper "MonetDB/X100: Hyper-Pipelining Query Execution".

> The terms *vectorization, hyper-pipelining,* and *superscalar* refer to the same idea: processing data in slices, which turns out to be a good compromise between row-at-a-time or column-at-a-time. DuckDB's query engine uses the same principle.

This paper was published in January 2005, so it's safe to assume that it was finalized in late 2004 – almost exactly 20 years ago!

If we read the paper, we learn that the experiments were carried out on an HP workstation equipped with 12 GB of memory (the same amount as the Samsung phone has today!). It also had an Itanium CPU and looked like this:

The Itanium2 workstation used in original the experiments (source: Wikimedia)

> Upon its release in 2001, the Itanium was aimed at the high-end market with the goal of eventually replacing the then-dominant x86 architecture with a new instruction set that focused heavily on SIMD (single instruction, multiple data). While this ambition did not work out, the Itanium was the state-of-the-art architecture of its day. Due to the focus on the server market, the Itanium CPUs had a large amount of cache: the 1.3 GHz Itanium2 model used in the experiments had 3 MB of L2 cache, while Pentium 4 CPUs released around that time only had 0.5–1 MB.

The paper provides a detailed breakdown of the runtimes:

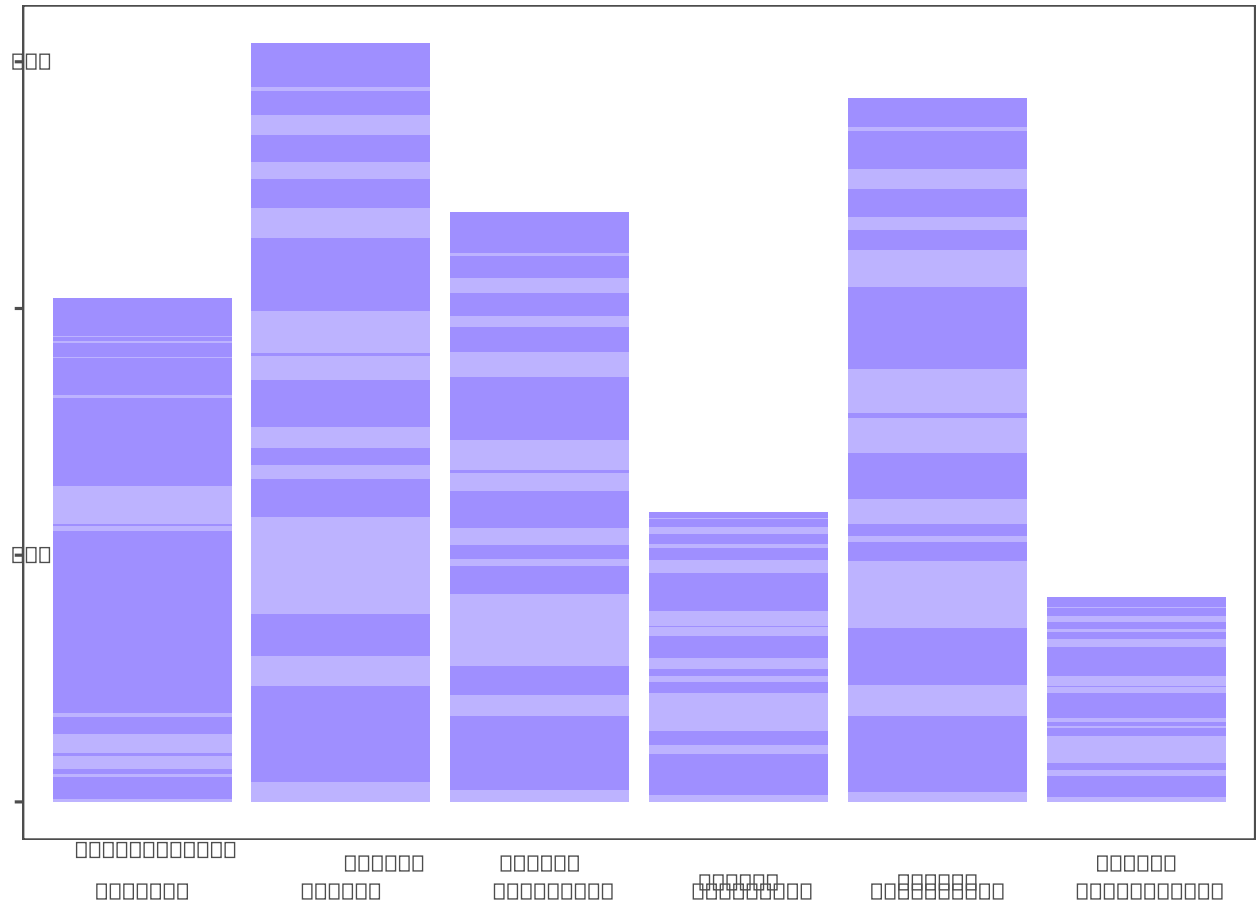
MonetDB/MIL		MonetDB/X100, 1CPU			DB2, 4CPU
Q	SF=1	SF=1	SF=1	SF=100	SF=100
1	3.72	0.50	0.31	30.25	229
2	0.46	0.01	0.01	0.81	19
3	2.52	0.04	0.02	3.77	16
4	1.56	0.05	0.02	1.15	14
5	2.72	0.08	0.04	11.02	72
6	2.24	0.09	0.02	1.44	12
7	3.26	0.22	0.22	29.47	81
8	2.23	0.06	0.03	2.78	65
9	6.78	0.44	0.44	71.24	274
10	4.40	0.22	0.19	30.73	47
11	0.43	0.03	0.02	1.66	20
12	3.73	0.09	0.04	3.68	19
13	11.42	1.26	1.04	148.22	343
14	1.03	0.02	0.02	2.64	14
15	1.39	0.09	0.04	14.36	30
16	2.25	0.21	0.14	15.77	64
17	2.30	0.02	0.02	1.75	77
18	5.20	0.15	0.11	10.37	600
19	12.46	0.05	0.05	4.47	81
20	2.75	0.08	0.05	2.45	35
21	8.85	0.29	0.17	17.61	428
22	3.07	0.07	0.04	2.30	93
	AthlonMP		Itanium2		

Table 4: TPC-H Performance (seconds)

Benchmark results from the paper “MonetDB/X100: Hyper-Pipelining Query Execution”

The total runtime of the TPC-H SF100 queries was 407.9 seconds – hence our baseline for the experiments. Here is a video of Hannes presenting the results at the event:

And here are all results visualized on a plot:



TPC-H SF100 total query runtimes for MonetDB/X100 and DuckDB

Conclusion

It was a long journey from the original vectorized execution paper to running an analytical database on a phone. Many key innovations happened that allowed these results, and the big improvement in hardware is just one of them. Another crucial component is that compiler optimizations became a lot more sophisticated. Thanks to this, while the MonetDB/X100 system needed to use explicit SIMD, DuckDB can rely on the auto-vectorization of our (carefully constructed) loops.

All that's left is to answer questions that we posed at the beginning of our journey. Yes, DuckDB can run TPC-H SF100 on a mobile phone. And yes, in some cases it can even outperform a research prototype running on a high-end machine of 2004 – on a modern smartphone that fits in your pocket.

And with newer hardware, smarter compilers and yet-to-be-discovered database optimizations, future versions are only going to be faster.

The DuckDB Avro Extension

Publication date: 2024-12-09

Author: Hannes Mühleisen

TL;DR: DuckDB now supports reading Avro files through the `avro` community extension.

> **Update (May 2025):** Avro support is now available as the `avro` core extension.

The Apache™ Avro™ Format

Avro is a binary format for record data. Like many innovations in the data space, Avro was developed by Doug Cutting as part of the Apache Hadoop project in around 2009. Avro gets its name – somewhat obscurely – from a defunct British aircraft manufacturer. The company famously built over 7,000 Avro Lancaster heavy bombers under the challenging conditions of World War 2. But we digress.

The Avro format is yet another attempt to solve the dimensionality reduction problem that occurs when transforming a complex *multi-dimensional data structure* like tables (possibly with nested types) to a *single-dimensional storage layout* like a flat file, which is just a sequence of bytes. The most fundamental question that arises here is whether to use a columnar or a row-major layout. Avro uses a row-major layout, which differentiates it from its famous cousin, the Apache™ Parquet™ format. There are valid use cases for a row-major format: for example, appending a few rows to a Parquet file is difficult and inefficient because of Parquet's columnar layout and due to the fact the Parquet metadata is stored *at the back* of the file. In a row-major format like Avro with the metadata *up top*, we can "just" add those rows to the end of the files and we're done. This enables Avro to handle appends of a few rows somewhat efficiently.

Avro-encoded data can appear in several ways, e.g., in RPC messages but also in files. In the following, we focus on files since those survive long-term.

Header Block

Avro "object container" files are encoded using a comparatively simple binary format: each file starts with a **header block** that first has the magic bytes `Obj1`. Then, a metadata "map" (a list of string-bytearray key-value pairs) follows. The map is only strictly required to contain a single entry for the `avro.schema` key. This key contains the Avro file schema encoded as JSON. Here is an example for such a schema:

```
{
  "namespace": "example.avro",
  "type": "record",
  "name": "User",
  "fields": [
     {"name": "name", "type": "string"},
     {"name": "favorite_number", "type": ["int", "null"]},
     {"name": "favorite_color", "type": ["string", "null"]}
  ]
}
```

The Avro schema defines a record structure. Records can contain scalar data fields (like `int`, `double`, `string`, etc.) but also more complex types like records (similar to DuckDB STRUCTs), unions and lists. As a sidenote, it is quite strange that a data format for the definition of record structures would fall back to another format like JSON to describe itself, but such are the oddities of Avro.

Data Blocks

The header concludes with 16 randomly chosen bytes as a "sync marker". The header is followed by an arbitrary amount of **data blocks**: each data block starts with a record count, followed by a size and a byte array containing the actual records. Optionally, the bytes can be compressed with deflate (gzip), which will be known from the header metadata.

The data bytes can only be decoded using the schema. The object file specification contains the details on how each type is encoded. For example, in the example schema we know each value is a record of three fields. The root-level record will encode its entries in the order they are declared. There are no actual bytes required for this. First we will be reading the `name` field. Strings consist of a length followed by the string bytes. Like other formats (e.g., Thrift), Avro uses variable-length integers with zigzag encoding to store lengths and counts and the like. After reading the string, we can proceed to `favorite_number`. This field is a union type (encoded with the `[]` syntax). This union can have values of two types, `int` and `null`. The `null` type is a bit odd, it can only be used to encode the fact that a value is missing. To decode the `favorite_number` fields, we first read an `int` that encodes which choice of the union was used. Afterward, we use the "normal" decoders to read the values (e.g., `int` or `null`). The same can be done for `favorite_color`. Each data block again ends with the sync marker. The sync marker can be used to verify that the block was fully written and that there is no garbage in the file.

The DuckDB `avro` Community Extension

We have developed a DuckDB community extension that enables DuckDB to *read* Apache Avro™ files.

The extension does not contain Avro *write* functionality. This is on purpose, by not providing a writer we hope to decrease the amount of Avro files in the world over time.

Installation & Loading

Installation is simple through the DuckDB community extension repository, just type

```
INSTALL avro FROM community;
LOAD avro;
```

in a DuckDB instance near you.

> Since DuckDB v1.2.1, DuckDB's WebAssembly client is also supported.

The `read_avro` Function

The extension adds a single DuckDB function, `read_avro`. This function can be used like so:

```
FROM read_avro('some_example_file.avro');
```

This function will expose the contents of the Avro file as a DuckDB table. You can then use any arbitrary SQL constructs to further transform this table.

File IO

The `read_avro` function is integrated into DuckDB's file system abstraction, meaning you can read Avro files directly from e.g., HTTP or S3 sources. For example:

```
FROM read_avro('http://blobs.duckdb.org/data/userdata1.avro');
FROM read_avro('s3://<my-example-bucket>/some_example_file.avro');
```

should "just" work.

You can also *glob* multiple files in a single read call or pass a list of files to the functions:

```
FROM read_avro('some-example-file-*.avro');
FROM read_avro(['some-example-file-1.avro', 'some-example-file-2.avro']);
```

If the filenames somehow contain valuable information (as is unfortunately all-too-common), you can pass the `filename` argument to `read_avro`:

```
FROM read_avro('some-example-file-*.avro', filename = true);
```

This will result in an additional column in the result set that contains the actual filename of the Avro file.

Schema Conversion

This extension automatically translates the Avro Schema to the DuckDB schema. *All* Avro types can be translated, except for *recursive type definitions*, which DuckDB does not support.

The type mapping is very straightforward except for Avro's "unique" way of handling `NULL`. Unlike other systems, Avro does not treat `NULL` as a possible value in a range of e.g., `INTEGER` but instead represents `NULL` as a union of the actual type with a special `NULL` type. This is different to DuckDB, where any value can be `NULL`. Of course DuckDB also supports `UNION` types, but this would be quite cumbersome to work with.

This extension *simplifies* the Avro schema where possible: an Avro union of any type and the special null type is simplified to just the non-null type. For example, an Avro record of the union type `["int", "null"]` (like `favorite_number` in the example) becomes a DuckDB `INTEGER`, which just happens to be `NULL` sometimes. Similarly, an Avro union that contains only a single type is converted to the type it contains. For example, an Avro record of the union type `["int"]` also becomes a DuckDB `INTEGER`.

The extension also "flattens" the Avro schema. Avro defines tables as root-level "record" fields, which are the same as DuckDB `STRUCT` fields. For more convenient handling, this extension turns the entries of a single top-level record into top-level columns.

Implementation

Internally, this extension uses the "official" Apache Avro C API, albeit with some minor patching to allow reading Avro files from memory.

Limitations & Next Steps

In the following, we disclose the limitations of the `avro` DuckDB extension along with our plans to mitigate them in the future:

- The extension currently does not make use of **parallelism** when reading either a single (large) Avro file or when reading a list of files. Adding support for parallelism in the latter case is on the roadmap.
- There is currently no support for projection or filter **pushdown**, but this is also planned at a later stage.
- There is currently no support for ~~the Wasm or~~ the Windows-MinGW builds of DuckDB due to issues with the Avro library dependency (sigh again). We plan to fix this eventually.
- As mentioned above, DuckDB cannot express recursive type definitions that Avro has. This is unlikely to ever change.
- There is no support to allow users to provide a separate Avro schema file. This is unlikely to change, all Avro files we have seen so far had their schema embedded.
- There is currently no support for the `union_by_name` flag that other readers in DuckDB support. This is planned for the future.

Conclusion

The new `avro` community extension for DuckDB enables DuckDB to read Avro files directly as if they were tables. If you have a bunch of Avro files, go ahead and try it out! We'd love to hear from you if you run into any issues.

25 000 Stars on GitHub

Publication date: 2024-12-16

Author: The DuckDB team

TL;DR: We have recently reached 25 000 stars on GitHub. We would like to use this occasion to stop and reflect about DuckDB's recent year and our future plans.

Our GitHub repository has just passed 25,000 stars. This is great news and since it is also the end of the year it is a good moment to reflect on DuckDB's trajectory. There has been a lot of new and exciting adoption of DuckDB across the industry.

We would like to highlight two main events that have happened this year:

- We released DuckDB 1.0.0. This version introduced a stable storage format which guarantees backwards compatibility and limited forward compatibility.
- We started the DuckDB Community Extensions project. Community extensions allow developers to contribute packages to DuckDB and users to easily install these extensions using the simple command `INSTALL xyz FROM community`.

Besides the GitHub stars we have also observed a lot of growth in various metrics.

- Each month, our website handles over 1.5 million unique visitors. In addition, we see over 300 TB in traffic from ca. 30 million extension downloads. Thanks again to Cloudflare for sponsoring the project with free content delivery services!
- In one year, we rose in the DB Engines ranking from position 91 to 55 on the general board and from position 47 to 33 in the relational board, which makes DuckDB the fastest growing relational system in the top-50.
- We count 7.5M+ monthly downloads in PyPI.
- Maven Central downloads for the JDBC driver have also shot up, we now see over 500k+ downloads per month.

We should note that we're not glorifying those numbers and they are not a target per se for our much-beloved optimization in accordance with Goodhart's law. Still, they are just motivating to see grow.

As an aside, we have recently opened a Bluesky account and are seeing great discussions happening over there. The account has already exceeded 4 thousand followers!

Following our ancient two-year tradition, we hosted two DuckCon events, one in Amsterdam and another in Seattle. We also organized the first DuckDB Amsterdam Meetup.

Early next year, we are going to host DuckCon in Amsterdam, which is going to be the first event that we live stream in order to be more accessible to the growing DuckDB users in, e.g., Asia. But for now, let's sit around the syntax tree and be merry thinking about what's to come.

SELECT
INSERT
CASE
QUALIFY
UNION
CAST
ANY
CREATE
LIMIT

DuckDB Node Neo Client

Publication date: 2024-12-18

Author: Jeff Raymakers

TL;DR: The new DuckDB Node client, “Neo”, provides a powerful and friendly way to use your favorite database

Meet the newest DuckDB client API: DuckDB Node “Neo”!

You may be familiar with DuckDB’s old Node client. While it has served the community well over the years, “Neo” aims to learn from and improve upon its predecessor. It presents a friendlier API, supports more features, and uses a more robust and maintainable architecture. It provides both high-level conveniences and low-level access. Let’s take a tour!

What Does It Offer?

Friendly, Modern API

The old Node client’s API is based on SQLite’s. While familiar to many, it uses an awkward, dated callback-based style. Neo uses Promises natively.

```
const result = await connection.run(` SELECT 'Hello, Neo!'`);
```

Additionally, Neo is built from the ground up in TypeScript. Carefully chosen names and types minimize the need to check documentation.

```
const columnNames = result.columnNames();
const columnTypes = result.columnTypes();
```

Neo also provides convenient helpers to read only as many rows as needed and return them in either column-major or row-major format.

```
const reader = await connection.runAndReadUtil('FROM range(5000)',
    1000);
const rows = reader.getRows();
// OR: const columns = reader.getColumns();
```

Full Data Type Support

DuckDB supports a rich variety of data types. Neo supports every built-in type as well as custom types such as JSON. For example, ARRAY:

```
if (columnType.typeId === DuckDBTypeId.ARRAY) {
  const arrayValueType = columnType.valueType;
  const arrayLength = columnType.length;
}
```

DECIMAL:

```
if (columnType.typeId === DuckDBTypeId.DECIMAL) {
  const decimalWidth = columnType.width;
  const decimalScale = columnType.scale;
}
```

And JSON:

```
if (columnType.alias === 'JSON') {
  const json = JSON.parse(columnValue);
}
```

Type-specific utilities ease common conversions such as producing human-readable strings from TIMESTAMPs or DECIMALs, while preserving access to the raw values for lossless processing.

```
if (columnType.typeId === DuckDBTypeId.TIMESTAMP) {
  const timestampMicros = columnValue.micros; // bigint
  const timestampString = columnValue.toString();
  const {
    date: { year, month, day },
    time: { hour, min, sec, micros },
  } = columnValue.toParts();
}
```

Advanced Features

Need to bind specific types of values to prepared statements, or precisely control SQL execution? Perhaps you want to leverage DuckDB's parser to extract statements, or efficiently append data to a table. Neo has you covered, providing full access to these powerful features of DuckDB.

Binding Values to Prepared Statements

When binding values to parameters of prepared statements, you can select the SQL data type. This is useful for types that don't have a natural equivalent in JavaScript.

```
const prepared = await connection.prepare('SELECT $1, $2');
prepared.bindTimestamp(1, new DuckDBTimestampValue(micros));
prepared.bindDecimal(2, new DuckDBDecimalValue(value, width, scale));
const result = await prepared.run();
```

Controlling Task Execution

Using pending results allows pausing or stopping SQL execution at any point, even before the result is ready.

```
import { DuckDBPendingResultState } from '@duckdb/node-api';

// Placeholder to demonstrate doing other work between tasks.
async function sleep(ms) {
  return new Promise((resolve) => {
    setTimeout(resolve, ms);
  });
}

const prepared = await connection.prepare('FROM range(10_000_000)');
const pending = prepared.start();
// Run tasks until the result is ready.
// This allows execution to be paused and resumed as needed.
// Other work can be done between tasks.
while (pending.runTask() !== DuckDBPendingResultState.RESULT_READY) {
  console.log('not ready');
  await sleep(1);
}
console.log('ready');
const result = await pending.getResult();
// ...
```

Extracting Statements and Running Them with Parameters

You can run multi-statement SQL containing parameters using the extract statements API.

```
// Parse this multi-statement input into separate statements.
const extractedStatements = await connection.extractStatements(`
  CREATE OR REPLACE TABLE numbers AS FROM range(?);
  FROM numbers WHERE range < ?;
  DROP TABLE numbers;
`);
const parameterValues = [10, 7];
const stmtCount = extractedStatements.count;
// Run each statement, binding values as needed.
for (let stmtIndex = 0; stmtIndex < stmtCount; stmtIndex++) {
  const prepared = await extractedStatements.prepare(stmtIndex);
  const paramCount = prepared.parameterCount;
  for (let paramIndex = 1; paramIndex <= paramCount; paramIndex++) {
    prepared.bindInteger(paramIndex, parameterValues.shift());
  }
  const result = await prepared.run();
  // ...
}
```

Appending Data to a Table

The appender API is the most efficient way to bulk insert data into a table.

```
await connection.run(
  `CREATE OR REPLACE TABLE target_table(i INTEGER, v VARCHAR)`
);

const appender = await connection.createAppender('main', 'target_table');

appender.appendInteger(100);
appender.appendVarchar('walk');
appender.endRow();

appender.appendInteger(200);
appender.appendVarchar('swim');
appender.endRow();

appender.appendInteger(300);
appender.appendVarchar('fly');
appender.endRow();

appender.close();
```

How Is It Built?

Dependencies

Neo uses a different implementation approach from most other DuckDB client APIs, including the old Node client. It binds to DuckDB's C API instead of the C++ API.

Why should you care? Using DuckDB's C++ API means building all of DuckDB from scratch. Each client API using this approach ships with a slightly different build of DuckDB. This can create headaches for both library maintainers and consumers.

Maintainers need to pull in the entire DuckDB source code. This increases the cost and complexity of the build, and thus the cost of code changes and especially DuckDB version updates. These costs often lead to significant delays in fixing bugs or supporting new versions.

Consumers are impacted by these delays. There's also the possibility of subtle behavioral differences between the builds in each client, perhaps introduced by different compile-time configuration.

> Some client APIs reside in the main DuckDB repository. This addresses some of the problems above, but increases the cost and complexity of maintaining DuckDB itself.

To use DuckDB's C API, on the other hand, one only needs to depend on released binaries. This significantly simplifies the maintenance required, speeds up builds, and minimizes the cost of updates. It removes the uncertainty and risk of rebuilding DuckDB.

Packages

DuckDB requires different binaries for each platform. Distributing platform-specific binaries in Node packages is notoriously challenging. It can often lead to inscrutable errors when installing, when the package manager attempts to rebuild some component from source, using whatever build and configuration tools happen to be around.

Neo uses a package design aimed to avoid these problems. Inspired by ESBuild, Neo packages pre-built binaries for each supported platform in a separate package. Each of these packages declares the particular platform (e.g., `os` and `cpu`) it supports. Then, the main package depends on all these platform-specific packages using `optionalDependencies`.

When the main package is installed, the package manager will only install optionalDependencies for supported platforms. So you only get exactly the binaries you need, no more. If installed on an unsupported platform, no binaries will be installed. At no point will an attempt to build from source occur during install.

Layers

The DuckDB Node Neo client has multiple layers. Most people will want to use Neo's main "api" package, @duckdb/node-api. This contains the friendly API with convenient helpers. But, for advanced use cases, Neo also exposes the lower-level "bindings" package, @duckdb/node-bindings, which implements a more direct translation of DuckDB's C API into Node.

This API has TypeScript definitions, but, as it follows the conventions of C, it can be awkward to use from Node. However, it provides a relatively unopinionated way to access DuckDB, which supports building special-purpose applications or alternate higher-level APIs.

Where Is It Headed?

Neo is currently marked "alpha". This is an indication of completeness and maturity, not robustness. Most of the functionality of DuckDB's C API is exposed, and what is exposed has extensive tests. But it's relatively new, so it may contain undiscovered bugs.

Additionally, some areas of functionality are not yet complete:

- Appending and binding advanced data types. These require additional functions in DuckDB's C API. The goal is to add these for the next release of DuckDB 1.2, currently planned for January 2025.
- Writing to data chunk vectors. Modifying binary buffers in a way that can be seen by a native layer presents special challenges in the Node environment. This is a high priority to work on in the near future.
- User-defined types & functions. The necessary functions and types were added to the DuckDB C API relatively recently, in v1.1.0. This is on the near-term roadmap.
- Profiling info. This was added in v1.1.0. It's on the roadmap.
- Table descriptions. This was also added in v1.1.0. It's on the roadmap.

New versions of DuckDB will include additions to the C API. Since Neo aims to cover all the functionality of the C API, these additions will be added to the roadmap as they are released.

If you have a feature request, or other feedback, let us know! Pull requests are also welcome.

What Now?

DuckDB Node Neo provides a friendly and powerful way to use DuckDB with Node. By leveraging DuckDB's C API, it exemplifies a new, more maintainable way to build on DuckDB, providing benefits to maintainers and consumers alike. It's still young, but growing up fast. Try it yourself!

Vertical Stacking as the Relational Model Intended: UNION ALL BY NAME

Publication date: 2025-01-10

Author: Alex Monahan

TL;DR: DuckDB allows vertical stacking of datasets by column name rather than position. This allows DuckDB to read files with schemas that evolve over time and finally aligns SQL with Codd's relational model.

Overview

Ever heard of SQL's `CORRESPONDING` keyword? Yeah, me neither! Well, it has been in the SQL standard since at least 1992, and almost nobody implemented it! `CORRESPONDING` was an attempt to fix a flaw in SQL – but it failed. It's time for SQL to get back to the relational model's roots when stacking data. Let's wind the clocks back to 1969...

You just picked up your own Ford Mustang Boss 302, drifting around the corner at every street to make it to the library to read the latest research report out of IBM by Edgar Codd. (Do we need a Neflix special about databases?) Reading that report, wearing plenty of plaid, you gain a critical insight: data should be treated as unordered sets! (Technically multisets – duplicates are everywhere...) Rows should be treated as unordered and so should columns. The relational model is *the way*. Any language built atop the relational model should absolutely follow those core principles.

A few years later, you learn about SQL, and it looks like a pretty cool idea. Declarative, relational – none of this maintaining order business. You don't want to be tied down by an ordering, after all. What if you change your mind about how to query your data? Sets are the best way to think about these things.

More time passes, and then, you have the need to stack some data in SQL. Should be easy enough – I can just take two tables and stack them, and the corresponding attributes will map together. No need to worry about ordering, and certainly no need to make sure that the relations are exactly the same width.

Wait. This can't be right.

I have to get the order of my columns exactly right? And I have to have the exact same number of columns in both relations? Did these SQL folks forget about Codd??

Fast forward just a couple of decades, and DuckDB is making stacking in SQL totally groovy again.

Making Vertical Stacking Groovy Again

In addition to the traditional `UNION` and `UNION ALL` operators, DuckDB adds both `UNION BY NAME` and `UNION ALL BY NAME`. These will vertically stack multiple relations (e.g., `SELECT` statements) by matching on the names of columns independent of their order. As an example, we provide columns a and b out of order, and even introduce the entirely new column c and stacking will still succeed:

```
SELECT
    42 AS a,
    'woot' AS b

UNION ALL BY NAME
```

```
SELECT
    'woot2' AS b,
    9001 AS a,
    'more wooting' AS c;
```

a	b	c
42	woot	NULL
9001	woot2	more wooting

> Any column that is not present in all relations is filled in with NULL in the places where it is missing.

This capability unlocks a variety of useful patterns that can add flexibility and save time. Some examples include:

- Stacking datasets that have different column orders
- Adding new columns to an analysis, but only for a portion of the rows
- Combining completely unrelated datasets into a single resultset
    - This can be useful if your IDE, BI tool, or API can only return a single resultset at a time, but you need to view multiple datasets

> DuckDB has had this capability since August of 2022, but the performance and scalability of this feature has recently been greatly improved! See the end of the post for some micro-benchmarks.

`UNION` vs. `UNION ALL`

If only using the keyword `UNION`, duplicates are removed when stacking. With `UNION ALL`, duplicates are permitted and the stacking occurs without additional processing.

Unfortunately we have Codd to thank for this confusing bit! If only `UNION ALL` were the default... Typically, `UNION ALL` (and its new counterpart `UNION ALL BY NAME`!) are the desired behavior as they faithfully reproduce the input relations, just stacked together. This is higher performance as well, since the deduplication that occurs with `UNION` can be quite time intensive with large datasets. And finally, `UNION ALL` preserves the original row order.

Reading Multiple Files

This column matching functionality becomes particularly useful when querying data from multiple files with different schemas. DuckDB provides a `union_by_name` boolean parameter in the table functions used to pull external flat files:

- `read_csv`
- `read_json`
- `read_parquet`

To read multiple files, DuckDB can use glob patterns within the file path parameter (or a list of files, or a list of glob patterns!). If those files could have different schemas, adding `union_by_name=True` will allow them to be read and stacked! Any columns that do not appear in a particular file will be filled with `NULL` values. For example:

```
COPY (SELECT 'Star' AS col1) TO 'star.parquet';
COPY (SELECT 'Wars' AS col2) TO 'wars.parquet';

FROM read_parquet(
    ['star.parquet', 'wars.parquet'],
    union_by_name = true);
```

col1	col2
Star	NULL
NULL	Wars

> If your files have different schemas and you did not expect it, DuckDB's friendly error messages will suggest the `union_by_name` parameter! There is no need for memorization:
>
> `If you are trying to read files with different schemas, try setting union_by_name=True`

Data Lakes

It is very common to have schema changes over time in data lakes, so this unlocks many additional uses for DuckDB in those environments. The secondary effect of this feature is that you may now feel free to change your data lake schemas freely! Now it is painless to add more attributes to your data lake over time – DuckDB will be ready to handle the analysis!

> DuckDB's extensions to read lakehouse table formats like Delta and Iceberg handle schema evolution within the formats' own metadata, so `union_by_name` is not needed.

Inserting Data by Name

Another use case for vertically stacking data is when inserting into an existing table. The DuckDB syntax of `INSERT INTO <my_table> BY NAME{:.language-sql .highlight}` offers the same flexibility of referring to columns by name rather than by position. This allows you to provide the data to insert with any column order and even including only a subset of columns. For example:

```sql
CREATE TABLE year_info (year INTEGER, status VARCHAR);

INSERT INTO year_info BY NAME
    SELECT
        'The planet made it through' AS status,
        2024 AS year;

INSERT INTO year_info BY NAME
    SELECT
        2025 AS year;

FROM year_info;
```

year	status
2024	The planet made it through
2025	NULL

The pre-existing alternative approach was to provide an additional clause that specified the list of columns to be added in the same order as the dataset. However, this requires the ordering and number of columns to be known up front rather than determined dynamically. In many cases it also requires specifying columns in two locations: the `INSERT` statement and the `SELECT` statement producing the data. Ignoring the sage advice of "Don't Repeat Yourself" has led to more than a few unintended consequences in my own code... It is always nicer to have a single location to edit rather than having to keep things in sync!

The Inspirations for `UNION ALL BY NAME`

Other systems and communities have tackled the challenges of stacking messy data for many years. DuckDB takes inspiration from them and brings their improvements back into SQL!

The most direct inspiration is the Pandas `concat` function. It was added in January of 2012, and from the very beginning it supported the addition of new columns. Pandas is incredibly widely used and is a significant contributor to the popularity of Python today. Bringing this capability to SQL can broaden its impact beyond Python and into the other languages that DuckDB supports (Java, Node.js, Go, Rust, etc.). Databases should learn from dataframes!

PySpark added the function `unionByName` in 2018 and added the abilty to handle the addition of new columns in version 3.1 in March of 2021. This is another option for Pythonistas, but carries with it the requirement for a Spark cluster and its overhead.

SQL's `UNION` clause had the `CORRESPONDING` keyword since 1992 (!) at the latest, but critically it lacks the ability to handle new or missing columns. As a result, it is useless for handling schema evolution.

It is our hope that we inspire other SQL engines to become "friendlier" and allow for this flexibility!

Improved Performance in DuckDB 1.1

DuckDB has supported `UNION ALL BY NAME` since 2022, but version 1.1 brought some significant scalability and performance improvements. This feature used to be an "if you have to" approach, but can now be used more broadly!

The first change reduced memory usage when reading multiple files over the network using `union_by_name`. This provides scalability benefits when querying from cloud object storage like S3, especially when the files are large relative to available memory.

The second change was to parallelize reads across files when using `union_by_name`. This expectedly provides a dramatic performance improvement (~6× in the microbenchmark in the PR).

Micro-Benchmark

This micro-benchmark is a reproduction of the work done by Daniel Beach (@DataEngDude) in this post. Thanks to Daniel for his permission to reuse his benchmark for this post!

The benchmark requires reading 16 GB of CSV files stored on S3 that have changing schemas on a cloud instance with 4 GB of memory. The intent behind it is to process large datasets on small commodity hardware (which is a use case where we want to see DuckDB be helpful!). The original post uses Linode, but for this post we selected the most similar AWS instance having the same amount of memory (`c5d.large`).

We use two quarters' of CSV files from the Backblaze dataset (2023 Q2 and 2023 Q3), which are placed in an S3 bucket.

I modified the query from here very slightly to remove the `ignore_errors = true` option. The benchmark continued to use Python, but I'm just showing the SQL here for better syntax highlighting:

```
CREATE OR REPLACE VIEW metrics AS
    SELECT
        date,
        sum(failure) AS failures
    FROM read_csv_auto('<s3_path>/*.csv', union_by_name = true)
    GROUP BY date;

COPY metrics TO '<s3_path>/results/results.csv';
```

When using a 4 GB instance and an older version of DuckDB (1.0.0), I am able to replicate the out of memory errors that Daniel encountered. If I upgrade to DuckDB 1.1.3, the queries run successfully! However, they required about 5.8 minutes to complete.

As I dug more deeply into the dataset, I discovered that the columns selected in the benchmark query are present in each file. In prior versions of DuckDB, just having files with different sets of columns would require the `union_by_name = True` flag, even if the inconsistent or new columns were not used in the query. However, between the original post and version 1.1.3, DuckDB added the capability to

do projection pushdown into CSV files! This means that only the columns used in the query are actually read from the CSV, not all columns. As a result, we can actually remove the `union_by_name = true` for the benchmark query and run successfully. This requires less overhead (since we do not need to invest time checking if all schemas match – we can rely on the first schema that is read). The simplified query runs in only 4 minutes, but it fails to exercise the capability we discussed – handling schema evolution!

To exercise the `BY NAME` capability, we add a column to the SQL query that is present only in some of the files.

```
CREATE OR REPLACE VIEW metrics AS
    SELECT
        date,
        count(DISTINCT datacenter) AS datacenters,
        sum(failure) AS failures
    FROM read_csv_auto('<s3_path>/*.csv', union_by_name = true)
    GROUP BY date;

COPY metrics TO '<s3_path>/results/results.csv';
```

This query runs in approximately the same amount of time as the original (5.6 minutes), so it is a good proxy for the original while showcasing how DuckDB handles schema evolution!

I then made a few tweaks to improve the performance. The first change is to skip the creation of a view and complete the operations all in one step. The reason this improves performance is that DuckDB will try to ensure that a view is correctly defined by binding it when it is created. Normally, this has negligible overhead (views are a great abstraction!), however when reading from cloud object storage and using `UNION ALL BY NAME`, this triggers a check of the schema of each file, which can take time. In this case, around 2 minutes! The updated SQL statement looks like this:

```
COPY (
    SELECT
        date,
        count(DISTINCT datacenter) AS datacenters,
        sum(failure) AS failures
    FROM read_csv_auto('<s3_path>/*.csv', union_by_name = true)
    GROUP BY date
) TO '<s3_path>/results/results.csv';
```

Performance improves to about 4.1 minutes with this change and also reduces the test down to a single query.

We can quantify the overhead of the flexibility that `UNION ALL BY NAME` provides if we keep the improved subquery syntax, but once again remove the `datacenter` column and the `union_by_name` flag.

```
COPY (
    SELECT
        date,
        sum(failure) AS failures
    FROM read_csv_auto('<s3_path>/*.csv')
    GROUP BY date
) TO '<s3_path>/results/results.csv';
```

This query runs in 3.7 minutes, so the overhead of handling schema evolution is only about 10%! That is a small price to pay for flexibilty and ease of use.

However, we can improve performance further still. The next change was to increase the number of threads that DuckDB uses. By default, DuckDB will use a single thread per core. However, this is a very I/O intensive query (due to the network hops reading from then writing to S3) and less of a CPU intensive one. DuckDB uses synchronous I/O, so with the default thread count, if a thread is doing I/O, that CPU core is idle. As a result, using more threads might be more likely to fully utilize network resources, which is the bottleneck in this test. Here I just made an educated guess that this would help, but monitoring CPU utilization is a better approach.

With 4 threads, instead of the default of 2, performance improves to 3 minutes!

Adding more threads did not meaningfully improve performance any further. Additional threads do use more memory, but with the improvements in 1.1, this is no longer a significant issue (I tested up to 16 threads with only 2.2 GB of memory used).

The table below summarizes the results achieved on a `c5d.large` instance, which has 2 vCPUs and 4 GB RAM. We report the total runtime and the maximum memory usage for each query.

Query syntax	`UNION` type	Threads	Runtime	Memory
create view, copy	`BY NAME`	2	5.8 min	0.47 GB
create view, copy	`BY POSITION`	2	4.0 min	0.47 GB
create view, copy, new column	`BY NAME`	2	5.6 min	0.47 GB
copy subquery, new column	`BY NAME`	2	4.1 min	0.47 GB
copy subquery	`BY POSITION`	2	3.7 min	0.49 GB
copy subquery, new column	`BY NAME`	4	3.0 min	0.77 GB

Closing Thoughts

When stacking data, DuckDB brings the spirit of the relational model back to SQL! After all, stacking data should not require column orders to match... The `BY NAME` keywords can simplify common operations like combining relations with different orders or sets of columns, inserting the results of a query into a table, or querying a data lake with a changing schema. As of DuckDB version 1.1, this is now a performant and scalable approach!

Happy analyzing!

TPC-H SF300 on a Raspberry Pi

Publication date: 2025-01-17

Author: Gábor Szárnyas

TL;DR: DuckDB can run all TPC-H SF300 queries on a Raspberry Pi board.

> Update (April 2025): The setup described in this post can now run **all TPC-H queries on the SF1,000 dataset** with the latest stable version of DuckDB, v1.2.2!

Introduction

The Raspberry Pi is an initiative to provide affordable and easy-to-program microcomputer boards. The initial model in 2012 had a single CPU core running at 0.7 GHz, 256 MB RAM and an SD card slot, making it a great educational platform. Over time the Raspberry Pi Foundation introduced more and more powerful models. The latest model, the Raspberry Pi 5, has a 2.4 GHz quad-core CPU and – with extra connectors – can even make use of NVMe SSDs for storage.

Last week, the Pi 5 got another upgrade: it can now be purchased with 16 GB RAM. The DuckDB team likes to experiment with DuckDB in unusual setups such as smartphones dipped into dry ice, so we were eager to get our hands on a new Raspberry Pi 5. After all, 16 GB or memory is equivalent to the amount found in the median gaming machine as reported in the 2024 December Steam survey. So, surely, the Pi must be able to handle a few dozen GBs of data, right? We ventured to find out.

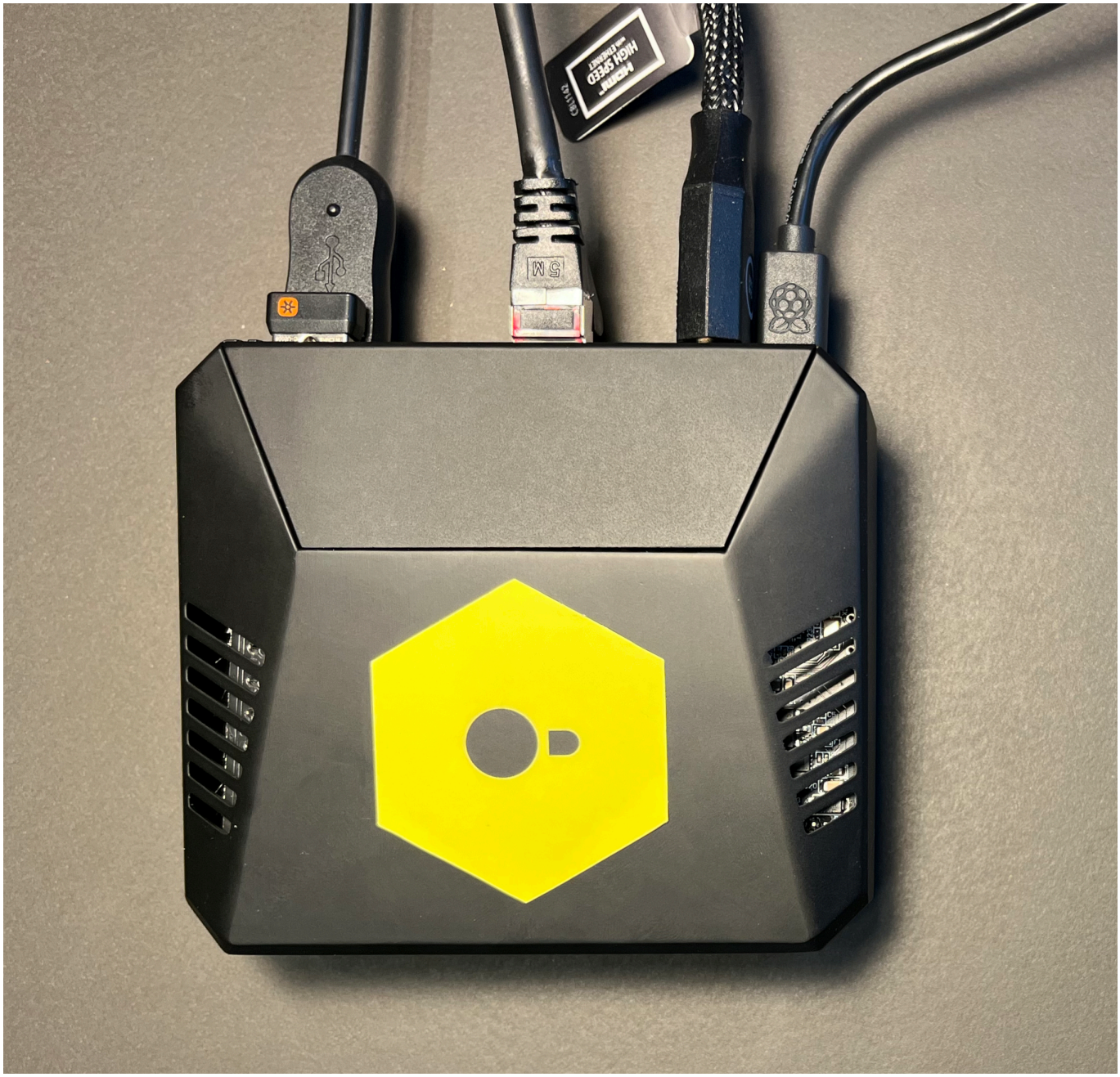
> Do you have a cool DuckDB setup? We would like to hear about it! Please post about it on social media or email it to gabor@duckdblabs.com.

Setup

Our setup consisted of the following components, priced at a total of $300:

Component	Price (USD)
Raspberry Pi 5 with 16 GB RAM	120.00
Raspberry Pi 27 W USB-C power supply	13.60
Raspberry Pi microSD card (128 GB)	33.40
Samsung 980 NVMe SSD (1 TB)	84.00
Argon ONE V3 Case	49.00
Total	**$300.00**

We installed the heat sinks, popped the SSD into place, and assembled the house. Here is a photo of our machine:



Experiments

So what is this little box capable of? We used the TPC-H workload to find out.

We first updated the Raspberry Pi OS (a fork of Debian Linux) to its latest version, 2024-11-19. We then compiled DuckDB version `0024e5d4be` using the Raspberry Pi build instructions. To make the queries easy to run, we also included the TPC-H extension in the build:

```
GEN=ninja CORE_EXTENSIONS="tpch" make
```

We then downloaded DuckDB database files containing the TPC-H datasets at different scale factors (SF):

```
wget https://blobs.duckdb.org/data/tpch-sf100.db
wget https://blobs.duckdb.org/data/tpch-sf300.db
```

We used two different storage options for both scale factors. For the first run, we stored the database files on the microSD card, which is the storage that most Raspberry Pi setups have. This card works fine for serving the OS and programs, and it can also store DuckDB databases but it's rather slow, especially for write-intensive operations, which include spilling to disk. Therefore, for the second run, we placed the database files on the 1 TB NVMe SSD disk.

For the measurements, we used our `tpch-queries.sql` script, which performs a cold run of all TPC-H queries, from Q1 to Q22.

We ran the script as follows:

```
duckdb tpch-sf${SF}.db -f tpch-queries.sql
```

We did not encounter any crashes, errors or incorrect results. The following table contains the aggregated runtimes:

Scale factor	Storage	Geometric mean runtime	Total runtime
SF100	microSD card	23.8 s	769.9 s
SF100	NVMe SSD	11.7 s	372.3 s
SF300	microSD card	171.9 s	4,866.5 s
SF300	NVMe SSD	55.2 s	1,561.8 s

Aggregated Runtimes

For the SF100 dataset, the geometric mean runtimes were 23.8 seconds with the microSD card and 11.7 seconds with the NVMe disk. The latter isn't that far off from the 7.5 seconds we achieved with the Samsung S24 Ultra, which has 8 CPU cores, most of which run at a higher frequency than the Raspberry Pi's cores.

For the SF300 dataset, DuckDB had to spill more to disk due to the limited system memory. This resulted in relatively slow queries for the microSD card setup with a geometric mean of 171.9 seconds. However, switching to the NVMe disk gave a 3× improvement, bringing the geometric mean down to 55.2 seconds.

Individual Query Runtimes

If you are interested in the individual query runtimes, you can find them below.

Perspective

To put our results into context, we looked at historical TPC-H results, and found that several enterprise solutions from 20 years ago had similar query performance, often reporting more than 60 seconds as the geometric mean of their query runtimes. Back then, these systems – with their software license and maintenance costs factored in – were priced *around $300,000!* This means that – if you ignore the maintenance aspects –, the "bang for your buck" metric (a.k.a. price–performance ratio) for TPC read queries has increased by around 1,000× over the last 20 years. This is a great demonstration of what the continuous innovation in hardware and software enables in modern systems.

> Disclaimer: The results presented here are not official TPC-H results and only include the read queries of TPC-H.

Summary

We showed that you can use DuckDB in a Raspberry Pi setup that costs $300 and runs all queries on the TPC-H SF300 dataset in less than 30 minutes.

We hope you enjoyed this blog post. If you have an interesting DuckDB setup, don't forget to share it with us!

Query Engines: Gatekeepers of the Parquet File Format

Publication date: 2025-01-22

Author: Laurens Kuiper

TL;DR: Mainstream query engines do not support reading newer Parquet encodings, forcing systems like DuckDB to default to writing older encodings, thereby sacrificing compression.

The Apache® Parquet™ Format

Apache Parquet is a popular, free, open-source, column-oriented data storage format. Whereas database systems typically load data from formats such as CSV and JSON into database tables before analyzing them, Parquet is designed to be efficiently queried directly. Parquet considers that users often only want to read some of the data, not all of it. To accommodate this, Parquet is designed to read individual columns instead of always having to read all of them. Furthermore, statistics can be used to filter out parts of files without fully reading them (this is also referred to as zone maps). Furthermore, storing data in Parquet typically results in a much smaller file than CSV or JSON due to a combination of lightweight columnar compression and general-purpose compression.

Many query engines implement reading and writing Parquet files. Therefore, it is also useful as a *data interchange* format. For example, Parquet files written by Spark in a large distributed data pipeline can later be analyzed using DuckDB. Because so many systems can read and write Parquet, it is the data format of choice for data lake solutions like Delta Lake™ and Iceberg™. While Parquet certainly has flaws, which researchers and companies are trying to address with new data formats, like it or not, it seems like Parquet is here to stay, at least for a while.

So, while we're here, we might try to make the best of it, right? SQL also has its flaws, and while researchers have certainly tried to create different query languages, we're still stuck with SQL. DuckDB embraces this and tries to make the best of it. The Parquet developers are doing the same for their format by updating it occasionally, bringing new features that make the format better.

Updates

If DuckDB adds a new compression method to its internal file format in a release, all subsequent releases must be able to read it. Otherwise, you couldn't read a database file created by DuckDB 1.1.0 after updating it to 1.2.0. This is called *backward compatibility*, and it can be challenging for developers. It sometimes requires holding onto legacy code and creating conversions from old to new. It is important to keep supporting older formats because updating DuckDB is much easier than rewriting entire database files.

Backward compatibility is also valuable for Parquet: it should be possible to read a Parquet file written years ago today. Luckily, most mainstream query engines can still read files in the Parquet 1.0 format, which was released in 2013, over ten years ago. Updates to the format do not threaten backward compatibility, as query engines simply need to continue being able to read the old files. However, it is also important that query engines add support for *reading* newer files alongside the older ones so that we can start *writing* new and improved Parquet files as well.

Here's where it gets tricky. We cannot expect query engines to be able to read the bleeding-edge Parquet format *tomorrow* if Parquet developers roll out an update *today*. We cannot start writing the new format for some time because many query engines will not be able to read it. The robustness principle states, “Be conservative in what you send, be liberal in what you accept.” If we apply this to Parquet files, query engines should strive to read new Parquet files but not write them yet, at least by default.

Encodings

DuckDB really likes lightweight compression. So, for the upcoming DuckDB 1.2.0 version, we're excited to have implemented the `DELTA_BINARY_PACKED`, `DELTA_LENGTH_BYTE_ARRAY` (added in Parquet 2.2.0 in 2015), and `BYTE_STREAM_SPLIT` (added in Parquet 2.8.0 in 2019) encodings in our Parquet writer. DuckDB, initially created in 2018, has been able to read Parquet since 2020, and has been able to read the encodings `DELTA_BINARY_PACKED` and `DELTA_LENGTH_BYTE_ARRAY` since 2022, and `BYTE_STREAM_SPLIT` since 2023.

However, despite these new encodings being available in 1.2.0, DuckDB will not write them by default. If DuckDB did this, many of our users would have a frustrating experience because some mainstream query engines still do not support reading these encodings. Having a good compression ratio does not help users if their downstream application cannot read the file. Therefore, we had to disable writing these encodings by default. They are only used when setting `PARQUET_VERSION v2` in a `COPY` command.

> DuckDB versions as old as 0.9.1 (released in late 2023) can already read files serialized with the setting `PARQUET_VERSION v2`.

Compressing data is almost always a trade-off between file size and the time it takes to write. Let's take a look at the following example (ran on a MacBook Pro with an M1 Max):

```
-- Generate TPC-H scale factor 1
INSTALL tpch;
LOAD tpch;
CALL dbgen(sf = 1);

-- Export to Parquet using Snappy compression
COPY lineitem TO 'snappy_v1.parquet'
    (COMPRESSION snappy, PARQUET_VERSION v1); -- 244 MB, ~0.46 s
COPY lineitem TO 'snappy_v2.parquet'
    (COMPRESSION snappy, PARQUET_VERSION v2); -- 170 MB, ~0.39 s

-- Export to Parquet using zstd compression
COPY lineitem TO 'zstd_v1.parquet'
    (COMPRESSION zstd, PARQUET_VERSION v1); -- 152 MB, ~0.58 s
COPY lineitem TO 'zstd_v2.parquet'
    (COMPRESSION zstd, PARQUET_VERSION v2); -- 135 MB, ~0.44 s
```

When using Snappy, DuckDB's default page compression algorithm for Parquet, which focuses mostly on speed, not compression ratio, the file is ~30% smaller and writing is ~15% faster with the encodings enabled. When using zstd, which focuses more on compression ratio than speed, the file is ~11% smaller, and writing is ~24% faster with the encodings enabled.

The compression ratio highly depends on how well data can be compressed. Here are some more extreme examples:

```
CREATE TABLE range AS FROM range(1e9::BIGINT);
COPY range TO 'v1.parquet' (PARQUET_VERSION v1); -- 3.7 GB, ~2.96 s
COPY range TO 'v2.parquet' (PARQUET_VERSION v2); -- 1.3 MB, ~1.68 s
```

The integer sequence 0, 1, 2, ... compresses extremely well with `DELTA_BINARY_PACKED`. In this case, the file is ~99% smaller, and writing is almost twice as fast.

Compressing floating points is much more difficult. Nonetheless, if there is a pattern, the data will compress quite well:

```
CREATE TABLE range AS SELECT range / 1e9 FROM range(1e9::BIGINT);
COPY range TO 'v1.parquet' (PARQUET_VERSION v1); -- 6.3 GB, ~3.83 s
COPY range TO 'v2.parquet' (PARQUET_VERSION v2); -- 610 MB, ~2.63 s
```

This sequence compresses really well with `BYTE_STREAM_SPLIT`. It is ~90% smaller and writes ~31% faster. Real-world data often does not have such extremely compressible patterns. Still, there are patterns, nonetheless, which will be exploited by these encodings.

If the query engines you're using support reading them, you can start using these encodings once DuckDB 1.2.0 is released!

Wasted Bits

Although it's difficult to get exact numbers, it's safe to assume that many TBs of data are written in Parquet each day. A large chunk of the bits written are wasted because query engines haven't implemented these newer encodings. The solution to this is surprisingly easy. There's no need to invent anything new to stop wasting all that space. Just read the specification on Parquet encodings, and implement them. Some of these "newer" encodings are almost 10 years old by now!

By reducing the size of Parquet files, we can reduce the amount of data we store in data centers. Reducing the amount of data we store even a little bit can have a big impact, as it can eventually reduce the need to build new data centers. This is not to say that data centers are evil; we will certainly need more of them in the future. However, making the most out of the data centers that we already have wouldn't hurt anyone.

Conclusion

Parquet is currently the industry standard tabular data format. Because it is also used as a data interchange format, the effectiveness of Parquet's features depends on the query engines that use it. If *some* of the mainstream query engines (you know who you are) refuse to implement these features, we *all* lose. This is not to say that all query engines must be on Parquet's bleeding edge, and DuckDB certainly isn't. However, query engine developers have a shared responsibility to make Parquet more useful.

We hope that more query engines will implement these newer encodings. Then, more query engines can write them by default and stop wasting so many bits.

Announcing DuckDB 1.2.0

Publication date: 2025-02-05

Author: The DuckDB team

TL;DR: The DuckDB team is happy to announce that today we're releasing DuckDB version 1.2.0, codenamed “Histrionicus”.

To install the new version, please visit the installation guide. For the release notes, see the release page.

> Some packages (Go, R, Java) take a few extra days to release due to the reviews required in the release pipelines.

We are proud to release DuckDB 1.2.0. This release is codenamed “Histrionicus” after the good-looking Harlequin duck (Histrionicus histrionicus), that inhabits “cold fast moving streams in North America, Greenland, Iceland and eastern Russia”.

What's New in 1.2.0

There have been far too many changes to discuss them each in detail, but we would like to highlight several particularly important and exciting features! Below is a summary of those new features with examples.

Breaking Changes

The `random` function now uses a larger state. This means that it's *even more random*™ now. Due to this change fixed seeds will now produce different values than in the previous versions of DuckDB.

`map['entry']` now returns a value, instead of a *list* of entries. For example, `map(['k'], ['v'])['k']` now returns `'v'`, while previously it returned `['v']`. We also introduced the `map_extract_value` function, which is now the alias for the bracket operator `[]`. If you would like to return a list, use the `map_extract` function: `map_extract(map(['k'], ['v']), 'k') = ['v']`.

The indexing of `list_reduce` is fixed. When indexing is applied in `list_reduce`, the index points to the last parameter of the lambda function and indexing starts from 1. Therefore, `list_reduce(['a', 'b'], (x, y, i) -> x || y || i)` returns ab2.

Explicit Storage Versions

DuckDB v1.2.0 ships new compression methods but *they are not yet enabled by default* to ensure that older DuckDB versions can read files produced by DuckDB v1.2.0.

In practice, this means that DuckDB v1.2.0 can read database files written by past stable DuckDB versions such as v1.0.0. When using DuckDB v1.2.0 with default settings, older versions can read files written by DuckDB v1.2.0.

You can *opt-in to newer forwards-incompatible features* using the following syntax:

```
ATTACH 'file.db' (STORAGE_VERSION 'v1.2.0');
```

This setting specifies the minimum DuckDB version that should be able to read the database file. When database files are written with this option, the resulting files cannot be opened by older DuckDB released versions than the specified version. They can be read by the specified version and all newer versions of DuckDB.

If you attach to DuckDB databases, you can query the storage versions using the following command:

```
SELECT database_name, tags FROM duckdb_databases();
```

This shows the storage versions:

database_name varchar	tags map(varchar, varchar)
file1 file2 ...	{storage_version=v1.2.0} {storage_version=v1.0.0 - v1.1.3} ...

This means that `file2` can be opened by past DuckDB versions while `file1` is compatible only with `v1.2.0` (or future versions).

To convert from the new format to the old format for compatibility, use the following sequence in DuckDB v1.2.0:

```
ATTACH 'file1.db';
ATTACH 'converted_file.db' (STORAGE_VERSION 'v1.0.0');
COPY FROM DATABASE file1 TO converted_file;
```

Indexing

`ALTER TABLE ... ADD PRIMARY KEY`. After a long while, DuckDB is finally able to add a primary key to an existing table . So it is now possible to run this:

```
CREATE TABLE tbl (id INTEGER);
INSERT INTO tbl VALUES (42);
ALTER TABLE tbl ADD PRIMARY KEY (id);
```

Over-eager constraint checking addressed. We also resolved a long-standing issue with over-eager unique constraint checking. For example, the following sequence of commands used to throw an error but now works:

```
CREATE TABLE students (id INTEGER PRIMARY KEY, name VARCHAR);
INSERT INTO students VALUES (1, 'John Doe');

BEGIN; -- start transaction
DELETE FROM students WHERE id = 1;
INSERT INTO students VALUES (1, 'Jane Doe');
```

CSV Features

Latin-1 and UTF-16 encodings. Previously, DuckDB's CSV reader was limited to UTF-8 files. It can now read Latin-1 and UTF-16 files. For example:

```
FROM read_csv('cities-latin-1.csv', encoding = 'latin-1');
```

Multi-byte delimiters. DuckDB now supports delimiters of up to 4 bytes. This means that you can finally use the duck emoji as a column delimiter. For example:

```
a🦆b
hello🦆world
```

```
FROM read_csv('example.dsv', sep = '🦆');
```

Strict CSV parsing. The RFC 4180 specification defines requirements for well-formed CSV files, e.g., having a single line delimiter. By default, DuckDB now parses CSVs in so-called strict mode (‘ strict_mode = true). For example, the following CSV file gets rejected because of mixed newline characters:

```
echo "a,b\r\nhello,42\nworld,84" > rfc_4180-defiant.csv
```

```
FROM read_csv('rfc_4180-defiant.csv');
```

```
Invalid Input Error:
Error when sniffing file "rfc_4180-defiant.csv".
It was not possible to automatically detect the CSV Parsing dialect/types
```

But it's parsed with the more lenient option `strict_mode = false`:

```
FROM read_csv('rfc_4180-defiant.csv', strict_mode = false);
```

a varchar	b int64
hello world	42 84

Performance improvements. The CSV parser in the new release uses a new algorithm to find a new line on parallel execution. This leads to speedups of around 15%.

Unlimited row length. Previously, DuckDB was limited to CSV files with rows of up to 8 MB. The new version lifts this restriction, and lines can be of arbitrary length.

Parquet Features

Parquet dictionary and Bloom filter support. DuckDB now supports writing many more types using dictionary encoding. This should reduce file size in some cases. DuckDB is now also able to read and write Parquet Bloom filters. Bloom filters are small indexing data structures that can be used to exclude row groups if a filter is set. This is particularly useful for often-repeated but unordered data (e.g., categorical values). A separate blog post will follow.

Delta binary packed compression for Parquet. DuckDB now supports the `DELTA_BINARY_PACKED` compression as well as the `DELTA_LENGTH_BYTE_ARRAY` and `BYTE_STREAM_SPLIT` option for Parquet files. A few weeks ago, we elaborated on these in a blog post.

CLI Improvements

Safe mode. The DuckDB command line client now supports *safe mode*, which can be activated with the `-safe` flag or the `.safe_mode` dot command. In this mode, the CLI client is prevented from accessing external files other than the database file that it was initially connected to and prevented from interacting with the host file system. For more information, see the Securing DuckDB page in the Operations Manual.

Better autocomplete. The autocomplete in CLI now uses a Parsing Expression Grammar (PEG) for better autocomplete, as well as improved error messages and suggestions.

Pretty-printing large numbers. The CLI provides a summary of the number printed if the client is only rendering only a single row.

```
SELECT 100_000_000 AS x, pi() * 1e9 AS y;
```

x int32	y double
100000000 (100.00 million)	3141592653.589793 (3.14 billion)

Friendly SQL

Prefix aliases. SQL Expression and table aliases can now be specified before the thing they are referring to (instead of using the well-known syntax of using ASs). This can improve readability in some cases, for example:

```
SELECT
    e1: some_long_and_winding_expression,
    e2: t2.a_column_name
FROM
    t1: long_schema.some_long_table_name,
    t2: short_s.tbl;
```

Credit for this idea goes to Michael Toy. A separate blog post will follow soon. **Update:** the blog post on prefix aliases is out.

`RENAME` clause. DuckDB now supports the `RENAME` clause in `SELECT`. This allows renaming fields emitted by the `*` expression:

```
CREATE TABLE integers (col1 INTEGER, col2 INTEGER);
INSERT INTO integers VALUES (42, 84);
SELECT * RENAME (col1 AS new_col1) FROM integers;
```

Star `LIKE`. The `LIKE` and `SIMILAR TO` clauses can now be used on `*` expressions as a short-hand for the `COLUMNS` syntax.

```
CREATE TABLE key_val (key VARCHAR, val1 INTEGER, val2 INTEGER);
INSERT INTO key_val VALUES ('v', 42, 84);
SELECT * LIKE 'val%' FROM key_val;
```

val1	val2
int32	int32
42	84

Optimizations

We have spent a lot of time on DuckDB's optimizer. It is hard to quantify optimizer improvements, but as a result of these optimizations, DuckDB for example achieves a **13% improvement** on the total runtime of TPC-H SF100 queries when run on a MacBook Pro over the previous release.

C API for Extensions

Currently, DuckDB extensions use DuckDB's internal C++ structures. This – along with some fun linking issues – requires a lock-step development of extensions with mainline DuckDB and constant updates. Starting with this release, we expose a new C-style API for extensions in `duckdb_extension.h`. This API can be used to create for example scalar, aggregate or table functions in DuckDB. There are two main advantages of using this API: first, many programming languages (e.g., Go, Rust and even Java) have direct bindings to C APIs, making it rather easy to integrate. Secondly, the C Extension API is stable and backwards-compatible, meaning that extensions that target this API will keep working for new DuckDB versions. We will follow up with a new extension template.

musl Extensions

Distributing extensions for musl. The `musl` C library is often used in lightweight setups such as Docker setups running Alpine Linux. Starting with this release, we officially support musl and we distribute extensions for the `linux_amd64_musl` platform (but not yet for `linux_arm64_musl`). Note that DuckDB binaries (e.g., the CLI client) are not yet distributed for musl platforms, so you have to build them from source.

Final Thoughts

These were a few highlights – but there are many more features and improvements in this release. There have been **over 5 000 commits** by over 70 contributors since we released 1.1.3. The full – very long – release notes can be found on GitHub.

We would like to thank again our amazing community for using DuckDB, building cool projects on DuckDB and improving DuckDB by providing us feedback. Your contributions truly mean a lot!

Catching up with Windowing

Publication date: 2025-02-10

Author: Richard Wesley

TL;DR: DuckDB implements a number of modern windowing features, some of which are extensions to the SQL standard. This posts presents a few of these features, including GROUPS framing, QUALIFY and aggregate/function modifiers.

Background

In the beginning, the relational data processing model was all about sets. This was Codd's great insight and for many years, relational processing took little notice of data ordering.

But it turns out that there are a lot of analytic operations that are related to ordering. For example, smoothing out noise in time series is very difficult to do in traditional SQL queries – it involves self-joins with inequality conditions! So in the late 1990s database vendors started adding *windowing* operations. By making the user's intent clear, the operations could be implemented much more efficiently, and these operations were eventually added to the SQL:2003 standard.

DuckDB has had support for window functions since the early days, but if they are new to you, you might want to start with my earlier blog posts on Windowing in DuckDB and Fast Moving Holistic Aggregates, or just the window function documentation. In this post, I will start by introducing the more recent functionality additions. In a follow-up post, I will spelunk into the internals to talk about some performance and scaling improvements.

For the examples in this post, I will mostly stick to using a table of athletic `results`:

Field	Type	Description
`event`	`VARCHAR`	The name of the event (e.g., 200 meter butterfly).
`athlete`	`VARCHAR`	The name of the competitor (e.g., Michael Phelps).
`date`	`TIMESTAMP`	The start time of the event.
`time`	`DECIMAL(18, 3)`	The athlete's time in that event (in seconds).

GROUPS Framing

In addition to the ROWS and RANGE frame boundary types (which we have supported for a while now), the standard also defines GROUPS as a boundary type. ROWS is pretty simple: it just counts the number of rows. RANGE is trickier: it treats its counts as distances from the value of the ORDER BY expression at the current row. This means that there can only be one such expression and you have to be able to do arithmetic on it.

GROUPS is somewhere in between. A “group” in the standard's language is all the “peers” of a row, which are all the rows with the same value of the ORDER BY expression at the current row. In the original windowing code, this was not easy to implement, but after several years of work, the infrastructure has evolved, and as of v1.2.0 we now support this last type of framing.

Frame Exclusion

Another missing piece of the 2003 specification was the EXCLUDE clause. Thanks to work by a community member, we have supported this since v0.10.0, but we somehow never got around to mentioning it in a blog post!

EXCLUDE is an optional modifier to the frame clause for excluding rows around the CURRENT ROW. This is useful when you want to compute some aggregate value of nearby rows to see how the current row compares to it. In this example, we want to know how an athlete's time in an event compares to the average of all the times recorded for their event within ±10 days:

```
SELECT
    event,
    date,
    athlete,
    avg(time) OVER w AS recent,
FROM results
WINDOW w AS (
    PARTITION BY event
    ORDER BY date
    RANGE BETWEEN INTERVAL 10 DAYS PRECEDING AND INTERVAL 10 DAYS FOLLOWING
        EXCLUDE CURRENT ROW
)
ORDER BY event, date, athlete;
```

There are four options for EXCLUDE that specify how to treat the current row:

- CURRENT ROW – exclude just the current row
- GROUP – exclude the current row and all its "peers" (rows that have the same ORDER BY value)
- TIES – exclude all peer rows, but *not* the current row (this makes a hole on either side)
- NO OTHERS – don't exclude anything (the default)

Exclusion is implemented for both windowed aggregates and for the `first`, `last` and `nth_value` functions.

QUALIFY Clause

It may not be immediately obvious, but the SQL language has rules for the order in which various expressions are computed. For example, aggregates (like sum) are computed after row-level expressions (like +). This is why SQL has two filtering clauses: WHERE and HAVING: WHERE is for row-level computations and HAVING is applied after GROUP BY.

When windowing was introduced, it added another layer of computation: window functions are computed *after* aggregates. That is great, but then how do you filter the results of an OVER function? Originally, you had to put the query in a Common Table Expression (or CTE) which is defined by a WITH clause:

```
-- Find the third fastest times in each event
WITH windowed AS (
    SELECT
        event,
        athlete,
        time,
        row_number() OVER w AS r
    FROM results
    WINDOW w AS (
        PARTITION BY event
        ORDER BY time
    )
)
SELECT event, athlete, time
FROM windowed
WHERE r = 3;
```

This was kind of clunky, so eventually the `QUALIFY` clause was proposed for filtering window functions. DuckDB supports this, making it easier to filter the results of window functions:

```
-- Find the third fastest times in each event
SELECT event, athlete, time
FROM results
WINDOW w AS (
    PARTITION BY event
    ORDER BY time
)
QUALIFY row_number() OVER w = 3;
```

Aggregate Modifiers

There are several modifiers for ordinary aggregate functions (`FILTER`, `DISTINCT` and `ORDER BY` as an argument) that are not part of the SQL:2003 standard for windowing, but which are also useful in a windowing context. `FILTER` is pretty straightforward (and DuckDB has supported it for a while) but the others are not easy to implement efficiently.

They can of course be implemented naïvely (academic-speak for "slow"!) by just computing each row independently:

- re-read all the values,
- filter out the ones we don't want,
- stick them into a hash table to remove duplicates,
- sort the results,
- send them off to the aggregate function to get the result.

We have an implementation that does this (which you can access by turning off the optimizer) and we use it to check fancier implementations, but it is horribly slow.

Fortunately, these last two modifiers have been the subject of research published in the last 10 years, and we have now added those algorithms to the windowing aggregates. We can then use the `DISTINCT` modifier to exclude duplicates in the frame:

```
-- Count the number of distinct athletes at a given point in time
SELECT count(DISTINCT athlete) OVER (ORDER BY date) FROM results;
-- Concatenate those distinct athletes into a list
SELECT list(DISTINCT athlete) OVER (ORDER BY date) FROM results;
```

We can also use the `ORDER BY` modifier with order-sensitive aggregates to get sorted results:

```
-- Return an alphabetized list of athletes who made or beat a time
SELECT list(athlete ORDER BY athlete) OVER (
    PARTITION BY event, date
    ORDER BY time DESC
)
FROM results;
```

I should mention that the research on these extensions is ongoing, and combining them will often force us to use the naïve implementation. So for example, if we wished to exclude the athlete who made the time in the previous example:

```
-- Return an alphabetized list athletes who beat the each time
SELECT list(athlete ORDER BY athlete) OVER (
    PARTITION BY event, date
    ORDER BY time DESC
    EXCLUDE CURRENT ROW
)
FROM results;
```

DuckDB will still compute this for you, but it may be very slow.

Function Modifiers

The ORDER BY modifier also makes sense for some non-aggregate window functions, especially if we let them use framing with it:

```
-- Compute the current world record holder over time for each event
SELECT
    event,
    date,
    first_value(time ORDER BY time DESC) OVER w AS record_time,
    first_value(athlete ORDER BY time DESC) OVER w AS record_holder,
FROM results
WINDOW w AS (
    PARTITION BY event
    ORDER BY date
    ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
)
ORDER BY event, date;
```

All of the non-aggregate window functions (except dense_rank) now support ordering arguments and will use the frame instead of the entire partition when an ordering argument is supplied.

> **Tip.** If you wish to use the entire frame with an ordering argument, then you will need to be explicit and use RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING.

> **Note.** If you wish to use the frame ordering *and* the frame boundaries with a non-aggregate function, you will need to specify the ORDER BY *twice* (once in the frame specification and once in the argument list). This has not yet been optimized, but it will be in the v1.3.0 release.

Conclusion

Windowing is a very natural way to think about order-dependent analysis, but it is at odds with traditional unordered query processing. Nevertheless, since 2003 the SQL language has provided syntax for expressing a wide range of such queries. In recent years, the community has also considered further extensions to the language (such as QUALIFY and argument modifiers like DISTINCT and ORDER BY) to improve expressivity. Here at DuckDB we love providing this kind of expressiveness as part of our "friendly SQL" work. What may be less obvious is that when we enable users to express their problem more naturally, it helps us provide more performant solutions! In my next post, I will go deeper into recent improvements in windowing's performance and resource utilization.

Flying Through Windows

Publication date: 2025-02-14

Author: Richard Wesley

TL;DR: Dive into the details of recent DuckDB windowing performance improvements.

Introduction

In the previous post I went into some new windowing functionality in DuckDB available through SQL. But there are other changes that improve our use of resources (such as memory) without adding new functionality. So let's get “under the feathers” and look at these changes.

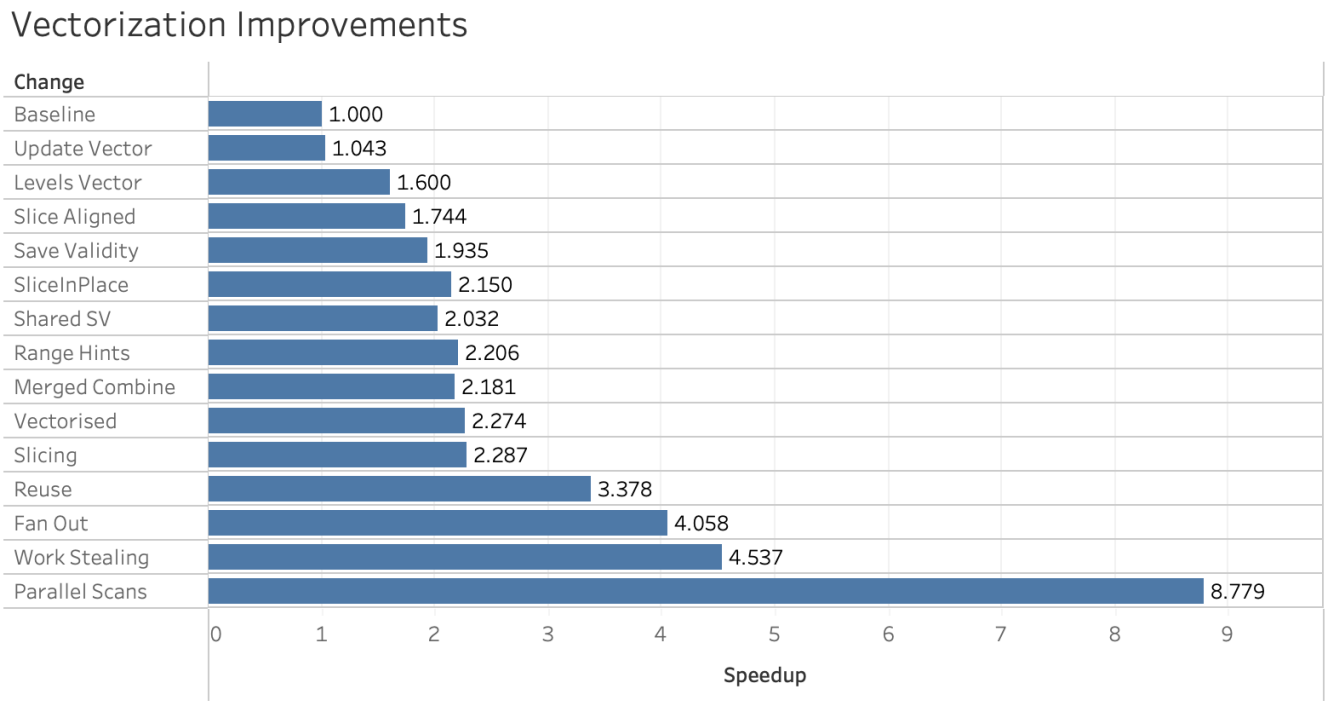
> We previously benchmarked ourselves on a window function-heavy workload, which showed great performance improvements over time. The optimizations presented in this blog post push the performance of DuckDB's window operator even further.

Segment Tree Vectorization

One important improvement that was made in the summer of 2023 was converting the segment tree evaluation code to vectorized evaluation from single-value evaluation. You might wonder why it wasn't that way to begin with in a “vectorized relational database” (!), but the answer is lost in the mists of time. My best guess is either that the published algorithm was written for values or that the aggregation API had not been nailed down yet (or both).

In the old version, we used the aggregate's `update` or `combine` APIs, but with only the values and tree states for a single row. To vectorize the segment tree aggregation, we accumulate *vectors* of leaf values and tree states and flush them into each output row's state when we reach the vector capacity of 2048 rows. Some care needed to be taken to handle order-sensitive aggregates by accumulating values in the correct order. `FILTER` and `EXCLUDE` clauses also provided some entertainment, but the segment trees are now fully vectorized. The performance gains here were about a factor of four (from “Baseline” to “Fan Out”).

The chart below shows the speedups achieved by the vectorization improvements:



Once segment trees were vectorized, we could use the same approach when implementing `DISTINCT` aggregates with merge sort trees. It may be worth updating the custom window API to handle vectorization at some point, because although most custom window aggregates are quite slow (e.g., `quantile`, `mad` and `mode`), `count(*)` is also implemented as a custom aggregate and would likely benefit from a vectorized implementation.

Constant Aggregation

A lot of window computations are aggregates over frames, and a common analytic task with these results is to compare a partial aggregate to the same aggregate over *the entire partition*. Computing this value repeatedly is expensive and potentially wasteful of memory (e.g, the old implementation would construct a segment tree even though only one value was needed.)

The previous performance workaround for this was to compute the aggregate in a subquery and join it in on the partition keys, but that was, well, unfriendly. Instead, we have added an optimization that checks for *partition-wide aggregates* and computes that value once per partition. This not only reduces memory and compute time for the aggregate itself, but we can often return a constant vector that shares the values across all rows in a chunk, reducing copy costs and potentially even downstream evaluation costs.

Returning a constant vector can yield surprisingly large memory and performance benefits. In the issue that drove this improvement, the user was constructing a constant 100k element list (!) and then computing the median with a list aggregation lambda. By returning a single constant list, we build and reduce that list only once instead of once per row!

Streaming Windows

Computing window functions is usually quite expensive! The entire relation has to be materialized, broken up into partitions, and each partition needs to be sorted.

But what if there is no partitioning or ordering? This just means that the window function is computed over the entire relation, in the "natural order", using a frame that starts with the first row and continues to the current row. Examples might be assigning row numbers or computing a running sum. This is simple enough that we can *stream* the evaluation of the function on a single thread.

First, let's step back a bit and talk about the window *operator*. During parsing and optimization of a query, all the window functions are attached to a single *logical* window operator. When it comes time to plan the query, we group the functions that have common partitions and "compatible" orderings (see Cao et al., *Optimization of Analytic Window Functions* for more information) and hand each group off to a separate *physical* window operator that handles that partitioning and ordering. In order to use the "natural order" we have to group those functions that can be streamed and execute them first (or the order will have been destroyed!) and hand them off to the *streaming* physical window operator.

So which window functions can we stream? It turns out there are quite a few:

- Aggregates `BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW` (we just update the aggregate)
- `first_value` – it's always the same
- `percent rank` – it's always 0
- `rank` – it's always 0
- `dense_rank` – it's always 0
- `row_number` – we just count the rows
- `lead` and `lag` – we just keep a buffer

There are a few more restrictions:

- `IGNORE NULLS`, `EXCLUDE` and `ORDER BY` arguments are not allowed
- `lead` and `lag` distances are restricted to a constant within ±2048 (one vector). this is not really a big deal because the distance is usually 1.

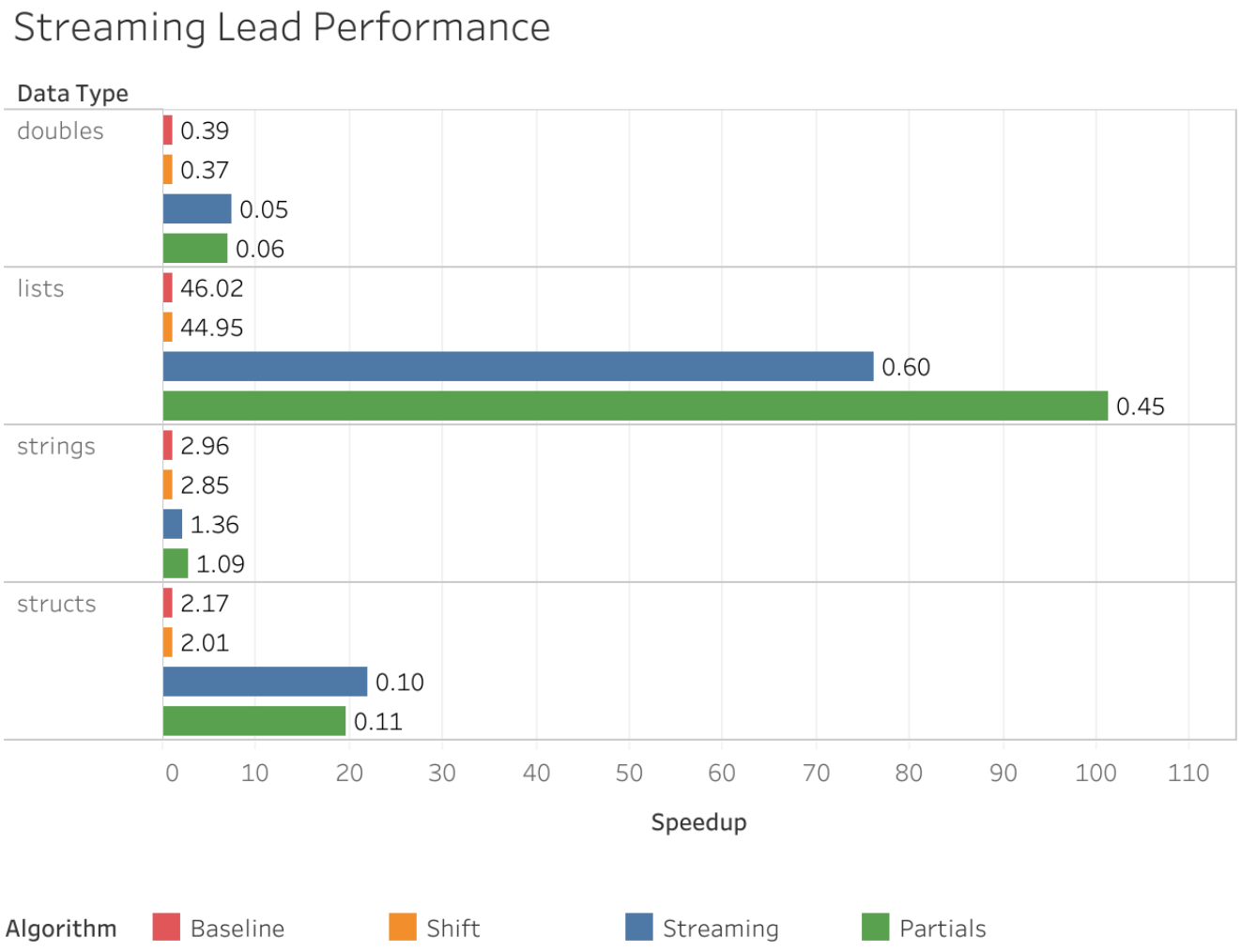
The improvements for streaming LEAD were quite dramatic:

```
SELECT setseed(0.8675309);
CREATE OR REPLACE TABLE df AS
    SELECT
        random() AS a,
        random() AS b,
        random() AS c,
    FROM range(10_000_000);

SELECT sum(a_1 + a_2 + b_1 + b_2)
FROM (
    SELECT
        lead(a, 1) OVER () AS a_1,
        lead(a, 2) OVER () AS a_2,
        lead(b, 1) OVER () AS b_1,
        lead(b, 2) OVER () AS b_2
    FROM df
) t;
```

The chart below shows the performance improvements.

- Baseline – Where we started
- Shift – Copying blocks of rows inside the non-streaming implementation, instead of one at a time
- Streaming – First implementation of streaming LEAD
- Partials – Avoid copying when the entire chunk can just be referenced

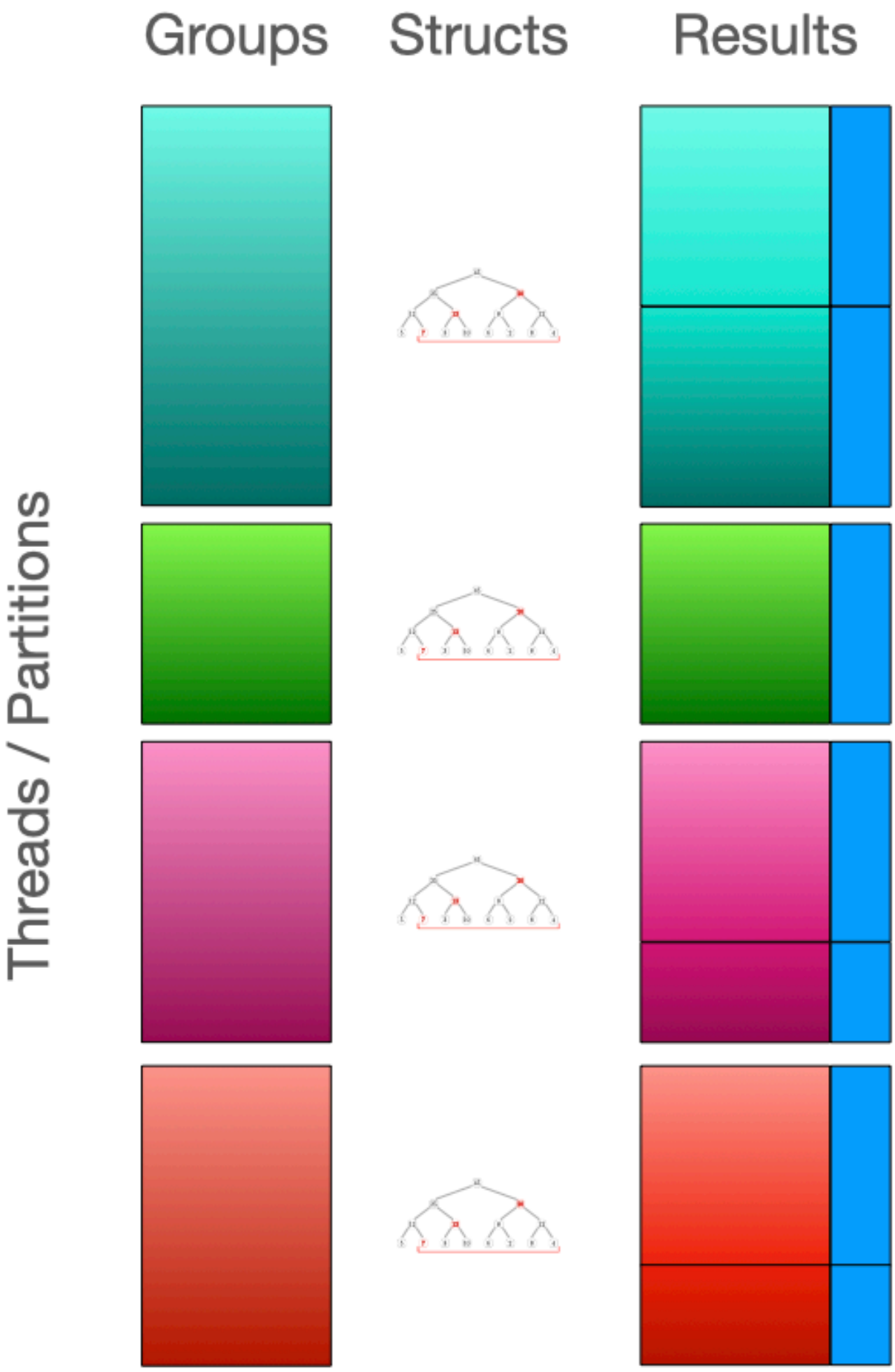
Note that the *x* axis shows the speedups compared to the baseline (1×) and the absolute runtimes (in seconds) are shown as labels.



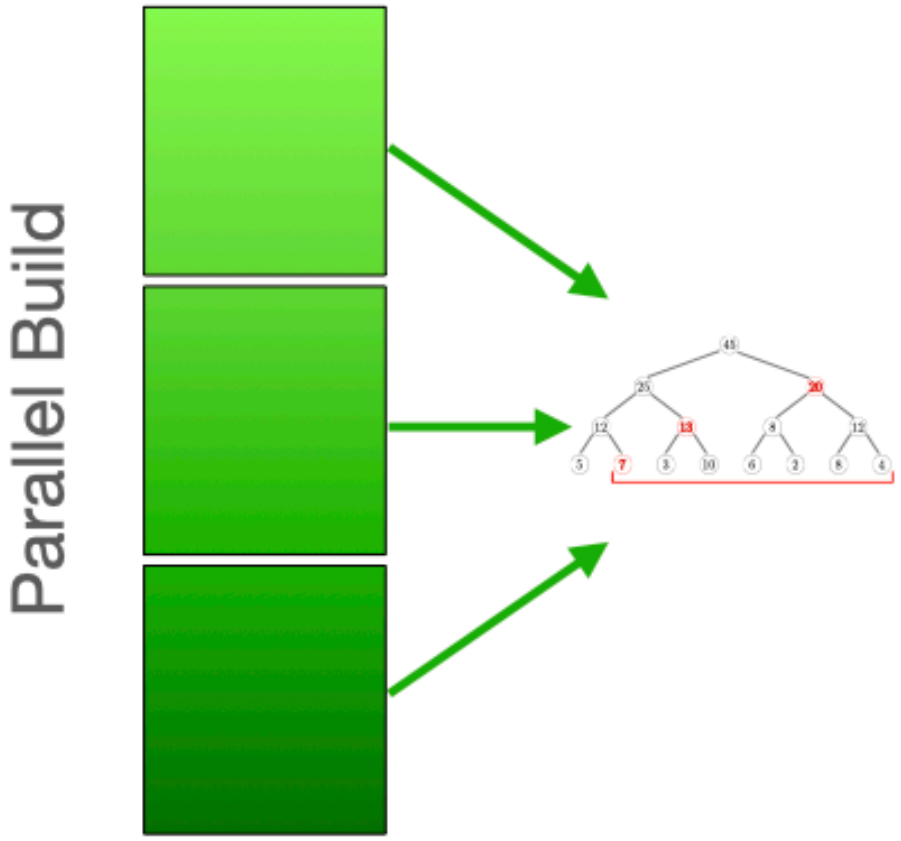
In the future we may be able to relax the end of the frame to a constant distance from the current row that fits inside the buffer length (e.g., `BETWEEN UNBOUNDED PRECEDING AND 1 PRECEDING`) but that hasn't been investigated yet.

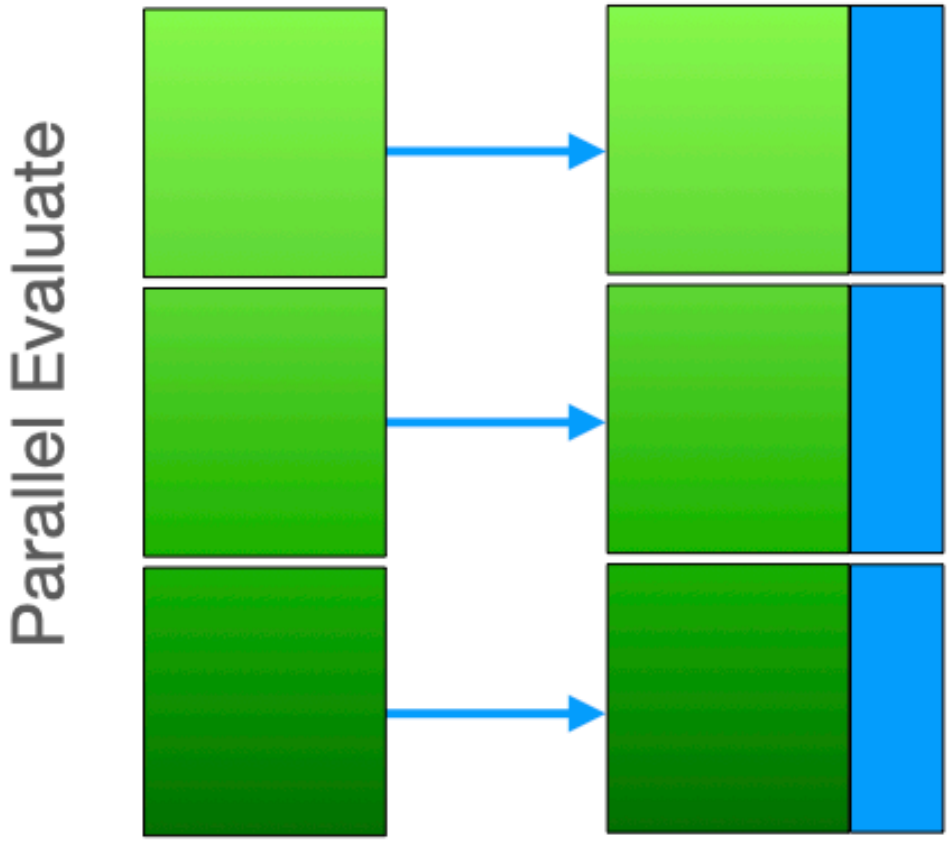
Partition Major Evaluation

Window partitions are completely independent, so evaluating them separately is attractive. Our first implementation took advantage of this and evaluated each partition on a separate thread. This works well if you have more partitions than threads, and they are all roughly the same size, but if the partition sizes are skewed or if there is only one partition (a common situation), then most of the cores will be idle, reducing throughput.


Groups
Structs
Results
Threads / Partitions

To improve the CPU utilization, we changed the execution model for v1.1 to evaluate partitions in parallel. The partitions are evaluated from largest to smallest and we then distribute each partition across as many cores as we can, while synchronizing access to shared data structures. This was a lot more challenging than independent single-threaded evaluation of partitions, and we had some synchronization issues (hopefully all sorted now!) that were dealt with in the v1.1.x releases. But we now have much better core utilization, especially for unpartitioned data. As a side benefit, we were able to reduce the memory footprint because fewer partitions were in memory at a time.


Parallel Build


Parallel Evaluate

One remaining issue is the coarseness of the sub-partitions. At the moment to avoid copying they use the blocks produced by the sorting code, which are often larger than we would like. Reducing the size of these chunks is future work, but hopefully we will get to it as part of some proposed changes to the sorting code.

Out of Memory Operation

Because windowing materializes the entire relation, it was very easy to blow out the memory budget of a query. For v1.2 we have switched from materializing active partitions in memory to using a pageable collection. So now not only do we have fewer active partitions (due to partition major evaluation above), but those partitions themselves can now spool to disk. This further reduces memory pressure during evaluation of large partitions.

Windowing is so complex, however, that there are still some remaining large data structures. The segment trees and merge sort trees used for accelerating aggregation are still in memory, especially the intermediate aggregate states in the middle of the trees. Solving this completely, will require a general method of serializing aggregate states to disk, which we do not yet have. Still, most aggregates can be serialized as binary data without special handling, so in the short term we can probably cover a lot of cases with the current aggregation infrastructure, just as we do for GROUP BY.

Shared Expressions

Window expressions are evaluated independently, but they often share expressions. Some of those expressions can be expensive to evaluate and others need to be materialized over the entire partition. As an example of the latter, aggregate functions can reference values anywhere in the partition. This could result in computing and materializing the same values multiple times:

```
-- Compute the moving average and range of x over a large window
SELECT
    x,
    min(x) OVER w AS min_x,
    avg(x) OVER w AS avg_x,
    max(x) OVER w AS max_x,
FROM data
WINDOW w AS (
    PARTITION BY p
    ORDER BY s
    ROWS BETWEEN 1_000_000 PRECEDING and 1_000_000 FOLLOWING
);
```

Paging the data for `x` reduces the memory footprint, but the segment trees used to evaluate the three aggregates will contain duplicate copies of `x` – along with the with operator itself (which has to return `x`). With v1.2 we have added a mechanism for sharing evaluation of expressions like these between functions. This not only reduces memory, but in this example we will also reduce disk paging because all three functions will be accessing the same values.

There are a number of places where we are sharing expressions, including ORDER BY arguments, `range` expressions and "value" functions like `lead`, `lag` and `nth_value`, and we are always on the lookout for more (such as frame boundaries – or even segment trees).

Future Work

While I have mentioned a number of things we would like to get to in the future, one that doesn't fit nicely into any of the topics so far is query rewriting. It turns out that some window functions can be evaluated using other techniques such as self-joins and some of our smarter aggregates (like `arg_max`). Generating these alternate query plans can have large performance benefits and we plan to investigate them.

Conclusion

As you can see, windowing is a big hairy beast! There is also not a lot of published research on effective algorithms (I've linked to pretty much all of it), so we are often stuck making it up ourselves or falling back to extremely simple and slow approaches. But I hope that many of you will find something new and exciting in what we have been up to for the past 2-3 years – and I will try to be more timely in blogging about future window improvements.

Planning AsOf Joins

Publication date: 2025-02-19

Author: Richard Wesley

TL;DR: AsOf Joins are a great example of how DuckDB can choose different implementations for an expensive operation.

Introduction

AsOf joins are a very useful kind of operation for temporal analytics. As the name suggests, they are a kind of lookup for when you have a table of values that change over time, and you want to look up the most recent value at another set of times. Put another way, they let you ask *"What was the value of the property **as of this time**?"*

DuckDB added AsOf joins about 18 months ago and you can read that post to learn about their semantics.

What's the Plan?

In that earlier post, I explained why we have a custom operator and syntax for AsOf joins when you can implement them in traditional SQL. The superpower of SQL is that it is *declarative:* you tell us *what* you want and we figure out an efficient *how.* By allowing you to say you want an AsOf join, we can think about how to get you results faster!

Still, the AsOf operator has to do a lot of work. Namely:

- Read all the data in the right side (lookup) table
- Partition it on any equality conditions
- Sort it on the inequality condition
- Repeat the process for the left side (probe) table
- Do a merge join on the two tables that only returns the "most recent" value

That's a lot of data movement! Plus, if any of the tables are large, we may end up exceeding memory and spilling to disk, slowing the operation down even further. Still, as we will see, it is much faster than the plain SQL implementation.

This is such a burden, that many databases that support AsOf joins require the right side table to be partitioned and ordered on any keys you might want to join on. That doesn't fit well with DuckDB's "friendly SQL" approach, so (for now) we have to do it every time.

Let's Get Small

It turns out there is a very common case for AsOf where the left table is small. Suppose you have a year's worth of price data, recorded at high granularity (e.g., fractions of a second), but you only want to look up a small number (say 20) values that are the times you actually bought or sold?

The price table could run to hundreds of millions, if not billions of rows, and just sorting that will take a lot of time and memory. Given how expensive that is, one might wonder if there is a way to avoid all that sorting? Happily the answer is *yes!*

Simple Joins

Suppose we were to swap the sides of the join, build the old left side as a small right side table, and stream the huge table through the left side of the join. We could use the AsOf conditions for the join, and hopefully find a way to throw out the older matches (we only want to keep the latest match). This would use very little memory, and the streaming could be highly parallelized.

There are two streaming physical join operators we could use for this:

- *Nested Loop Join* – Literally what it sounds like: loop over each left block and the right side table, checking for matches;
- *Piecewise Merge Join* – A tricky join for one inequality condition that sorts the right side and each left block before merging to find matches;

We can try both of these once we have a way to eliminate the duplicates. One thing to be aware of, though, is that they are both `N^2` algorithms, so there will be a limit on how big "small" can be.

Grouping

If you have been around databases long enough, you know that the phrase "eliminate the duplicates" means GROUP BY! So to eliminate the duplicates, we want to add an aggregation operator onto the output. The tricky part is that we want to keep only the matched values that have the "largest" times. Fortunately, DuckDB has a pair of aggregate functions that do just that: `arg_max` and `arg_min` (also called `max_by` and `min_by`).

That takes care of the fields from the lookup table, but what about the fields from the small table? Well, those values will all be the same, so we can just use the `first` aggregate function for them.

Streaming Window

But what should we group on? One might be tempted to group on the times that are being looked up, but that could be problematic if there are duplicate lookup times (only one of the rows would be returned!). Instead, we need to have a unique identifier for each row being looked up. The simplest way to do this is to use the *streaming window operator* with the `row_number()` window function. We then group on this row number.

Coming Together

This all sounds good, but how does it work in practice? How big can "small" get? To answer that I ran a number of benchmarks joining small tables against large ones. The tables are called `prices` and `times`:

```
CREATE OR REPLACE TABLE prices_{prices_size} AS
    SELECT
        r AS id,
        '2021-01-01T00:00:00'::TIMESTAMP +
            INTERVAL (random() * 365 * 24 * 60 * 60) SECOND
            AS time,
        (random() * 100000)::INTEGER AS price,
    FROM range({prices_size}) tbl(r);

CREATE OR REPLACE TABLE times_{times_size} AS
    SELECT
        r AS id,
        '2021-01-01'::TIMESTAMP +
            INTERVAL ((random() * 365 * 24 * 60 * 60)::INTEGER) SECONDS
            AS probe
    FROM range({times_size}) tbl(r);
```

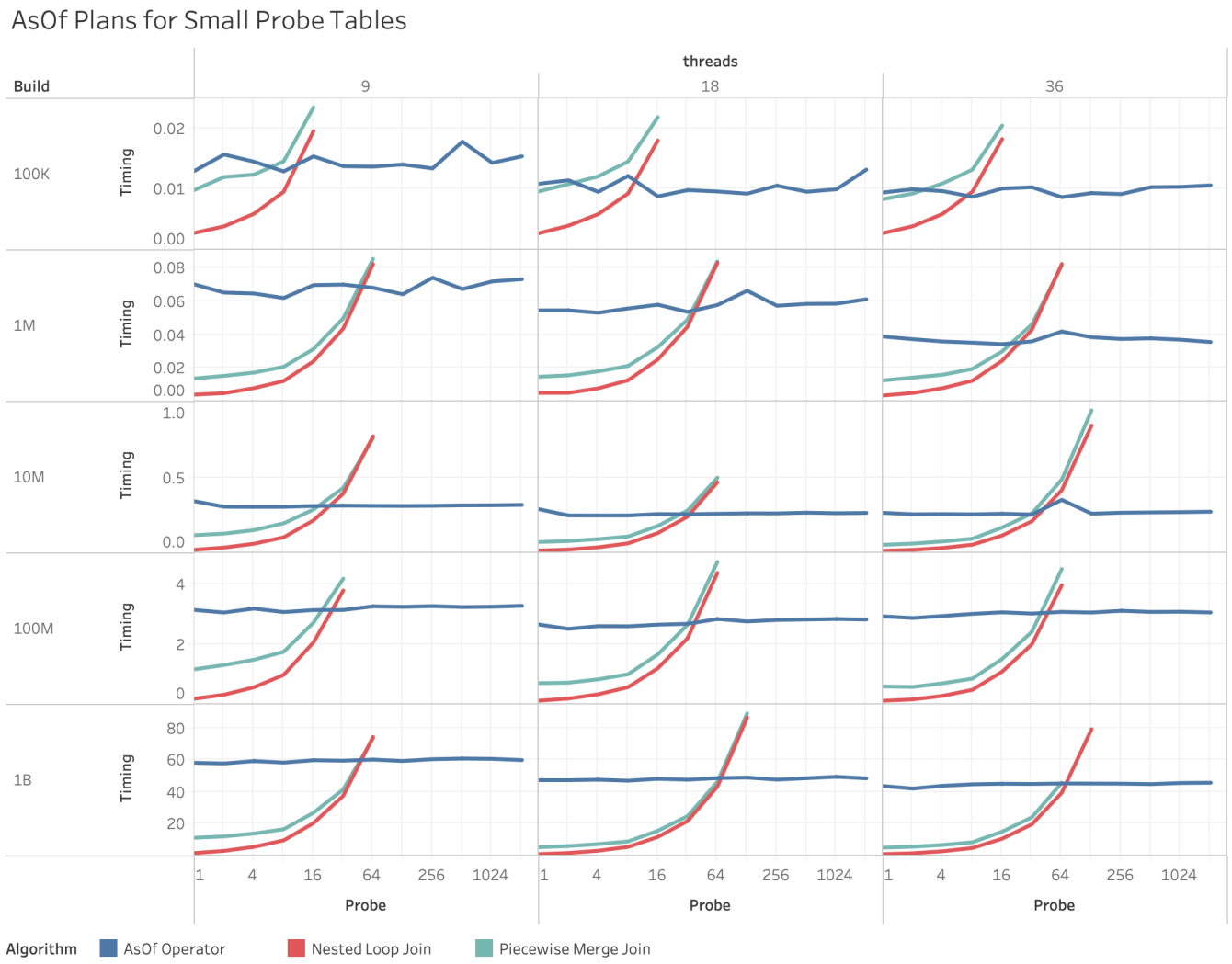
I then ran a benchmark query:

```
SELECT count(*)
FROM (
    SELECT
        t.probe,
        p.price
    FROM times_{times_size} t
        ASOF JOIN prices_{prices_size} p
        ON t.probe >= p.time
    ) t;
```

for a matrix of the following values:

- Prices – 100K to 1B rows in steps of 10x;
- Time – 1 to 2048 rows in steps of 2x (until it got too slow);
- Threads – 36, 18 and 9;

Here are the results:



As you can see, the quadratic nature of the joins means that "small" means "<= 64". That is pretty small, but the table in the original user issue had only 21 values.

We can also see that the sorting provided by piecewise merge join does not seem to help much, so plain old Nested Loop Join is the best choice.

It is clear that the performance of the standard operator is stable at each size, but decreases slowly as the number of threads increases.

This makes sense because sorting is compute-intensive and the fewer cores we can assign, the longer it will take.

If you want to play with the data more, you can find the interactive vizualization on our Tableau Public site.

Memory

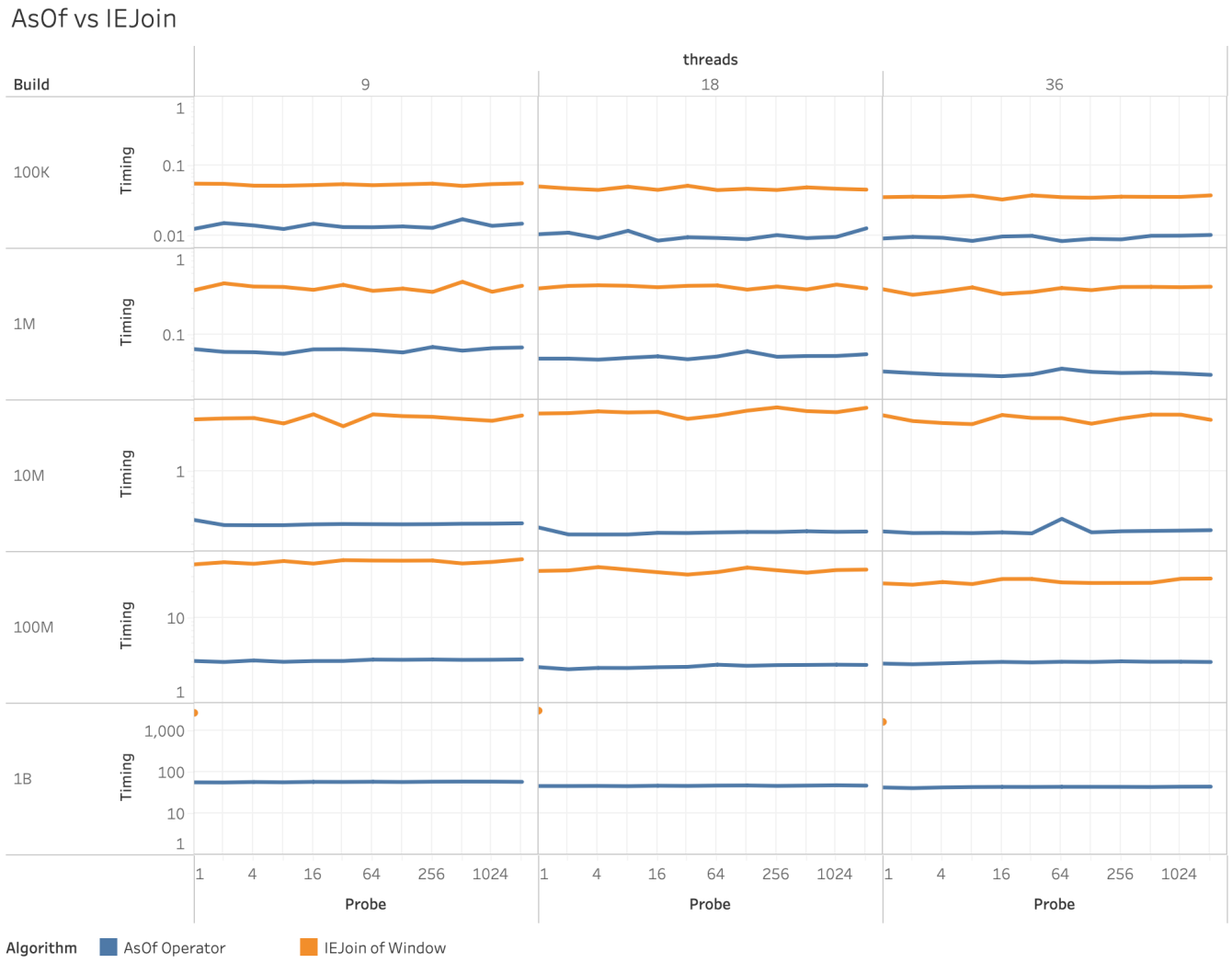
The Loop Join plan is clearly faster at small sizes, but how much memory do the two plans use? These are the rough amounts of memory needed before excessive paging or allocation failures occur:

Price Rows	AsOf Memory	Loop Join Memory
1B	48 GB	64 MB
100M	6 GB	64 MB
10M	256 MB	64 MB
1M	32 MB	64 MB
100K	32 MB	64 MB

In other words, the Loop Join plan only needs enough memory to page in the lookup table! So if the table is large and you have limited memory, the Loop Join plan is the best option, even if it is painfully slow. Just remember that the Loop Join plan has to compete with the speed of the standard operator under paging, and that may still be faster past a certain point.

Backup Plans

As part of the experiment, I also measured how the old SQL implementation would perform, and it did not fare well. At the 1B row level I had to cut it off after one run to avoid wasting time:



Note that the Y-axis is a log scale here!

Setting

While it is nice that we provide a default value for making plan choices like this, your mileage may vary as they say. If you have more time than memory, it might be worth it to you to bump up the Loop Join threshold a bit. The threshold is a new setting called `asof_loop_join_threshold` with a default value of 64, and you can change it using a PRAGMA statement:

```
PRAGMA asof_loop_join_threshold = 128;
```

Remember, though, this is a quadratic operation, and pushing it up too high might take a Very Long Time (especially if you express it in Old Entish!).

If you wish to disable the feature, you can just set it to zero:

```
PRAGMA asof_loop_join_threshold = 0;
```

Roll Your Own

This Loop Join plan optimization will not ship until v1.3, but if you are having problems today, you can always write your own version like this:

```
SELECT
    first(t.probe) AS probe,
    arg_max(p.price, p.time) AS price
FROM prices p
INNER JOIN (
    SELECT
        *,
        row_number() OVER () AS pk
    FROM times
) t
ON t.probe >= p.time
GROUP BY pk
ORDER BY 1;
```

If you know the probe times are unique, you can simplify this to:

```
SELECT
    t.probe,
    arg_max(p.price, p.time) AS price
FROM prices p
INNER JOIN times t
ON t.probe >= p.time
GROUP BY 1
ORDER BY 1;
```

Future Work

The new AsOf Loop Join plan feature only covers a common but very specific situation, and the standard operator could be made a lot more efficient if it knew that the data was already sorted. This is often the case, but we do not yet have the ability to track partitioning and ordering between operators. Tracking that kind of metadata would be very useful for speeding up a large number of operations, including sorting (!), partitioned aggregation, windowing, AsOf joins and merge joins. This is work we are very interested in, so stay tuned!

Conclusion

With apologies to Guido van Rossum, there is usually more than one way to do something, but each way may have radically different performance characteristics. One of the jobs of a relational database with a declarative query language like SQL is to make intelligent choices between the options so you the user can focus on the result. Here at DuckDB we look forward to finding more ways to plan your queries so you can focus on what you do best!

Notes

- The tests were all run on an iMac Pro with a 2.3 GHz 18-core Intel Xeon W CPU and 128 GB of RAM.
- The raw test data and visualizations are available on our Tableau Public repository.
- The script to generate the data is in my public DuckDB tools repository.

Prefix Aliases in SQL

Publication date: 2025-02-25

Author: Hannes Mühleisen

TL;DR: You can now put your aliases first in DuckDB's SQL dialect with a colon, e.g., `SELECT a: 42;`

Syntax

There is often more than one way to do things in our beloved SQL. For example, you can define join conditions *implicitly* (and dangerously) in the `WHERE` clause or use the (better) `JOIN ... ON ...` syntax. Generally, having "more than one way to do things" can be confusing and even outright dangerous sometimes.

Having said that we here at DuckDB pride ourselves in *friendlier* SQL. There are just too many people typing this stuff by hand, especially in the more ad-hoc world of analytics.

If there is a *good* reason to expand the SQL syntax with something useful we are at least considering it. Others seem to be watching closely. For example, our `GROUP BY ALL` syntax has by now been picked up by almost every SQL system out there (and their little brothers).

Aliases

In SQL, a user can define *aliases* for lots of things like `SELECT` expressions, table names, subqueries, etc. This is sometimes just nice to have readable column names in the result and sometimes required to refer back to a complex expression in for example the `ORDER BY` clause without just repeating it and praying for the optimizer. Aliases are defined *after* the thing they alias using `AS` but actually typing the `AS` term is optional. For example, those two statements are equivalent:

```
SELECT 42 AS fortytwo;
SELECT 42 fortytwo;
```

We can see the alias follows the expression (`42`) and the `AS` is optional. Having the alias *behind* the thing it describes is actually somewhat rare in programming, it is much more typical to define the alias first and then provide the expression. For example, in C:

```
int fortytwo = 42;
```

It seems to be a good idea to first state the alias, after all, this is what we are going to refer to this thing later on. Forcing it the other way around like SQL just increases the mental load. In addition, having the aliases last can make them quite hard to find if there are several complex expressions in a query. For example, here are the first few lines of the infamous TPC-H Query 1:

```
SELECT
    l_returnflag,
    l_linestatus,
    sum(l_quantity) sum_qty,
    sum(l_extendedprice) sum_base_price,
    sum(l_extendedprice * (1 - l_discount)) sum_disc_price,
    sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) sum_charge,
    avg(l_quantity) avg_qty,
    avg(l_extendedprice) avg_price,
    avg(l_discount) avg_disc,
    count(*) count_order
    ...
```

It's hard to spot all the aliases here, and this is not even a complex example. What if you could put the aliases first in SQL? Well, wait no more.

Prefix Aliases in DuckDB

In the latest DuckDB release, 1.2.0, we have quietly shipped yet another useful (we think) syntax extension to allow the alias to come before the thing it names using the colon (`:`) syntax. This turned out to be less difficult than we thought, "all" it took was some modification to the Bison parser we inherited from Postgres (but are in the process of replacing). Here is the example from before again:

```
SELECT fortytwo: 42;
```

Prefix aliases also work for table names, e.g.:

```
SELECT *
FROM my_table: some_other_table;
```

Aliases can be quoted if neccessary using *double* quotes:

```
SELECT "forty two": 42;
```

Prefix aliases can be used to name just about everything, for example expressions, function calls and subqueries in the `SELECT` clause:

```
SELECT
    e: 1 + 2,
    f: len('asdf'),
    s: (SELECT 42);
```

They can also apply to function calls and subqueries in the `FROM` clause, e.g.:

```
SELECT *
FROM
    r: range(10),
    v: (VALUES (42)),
    s: (FROM range(10))
```

Note that the `VALUES` clause and the subquery with `FROM` need additional parentheses to work here, this was required to pacify the evil Bison parser generator. Let's look at the Q1 example from earlier, but this time with prefix aliases:

```
SELECT
    l_returnflag,
    l_linestatus,
    sum_qty:        sum(l_quantity),
    sum_base_price: sum(l_extendedprice),
    sum_disc_price: sum(l_extendedprice * (1-l_discount)),
    sum_charge:     sum(l_extendedprice * (1-l_discount) * (1+l_tax)),
    avg_qty:        avg(l_quantity),
    avg_price:      avg(l_extendedprice),
    avg_disc:       avg(l_discount),
    count_order:    count(*)
    ...
```

There is no semantic difference between using `:` or `AS`, they lead to the same query constructs.

Credits

Credit for this idea goes to Looker veteran Michael Toy. We're grateful for his suggestion. We also would like to thank the legendary Lloyd Tabb for recommending Michael's idea to us. Also check out Mark Needham's video on DuckDB's prefix aliases.

Reading and Writing Google Sheets in DuckDB

Publication date: 2025-02-26

Authors: Alex Monahan and Archie Wood

TL;DR: Securely read from and write to Google Sheets directly in DuckDB using the GSheets community extension! For ad hoc querying, authentication is as easy as logging into Google from a browser. Scheduled workflows can use persistent DuckDB Secrets. SQL-on-Sheets has arrived!

Spreadsheets Are Everywhere

Is anything more polarizing for data folks than spreadsheets? Wait, don't answer that, we don't have time to talk about leading and trailing commas again...

The fact is that spreadsheets are everywhere. It is estimated that there are over 750 million spreadsheet users, compared to just 20 to 30 million programmers. That includes all languages put together!

There are a number of ways that using a spreadsheet can improve a data workflow. Blasphemy you say! Well, imagine if your database could actually read and write those spreadsheets. Spreadsheets are often the best place to manually edit data and they also provide highly customizable pivoting for self-serve analytics.

Now, you can use DuckDB to seamlessly bridge the gap between data-y folks and business-y folks! With a simple in-browser authentication flow, or an automateable private key file flow, you can both query from and load into Google Sheets.

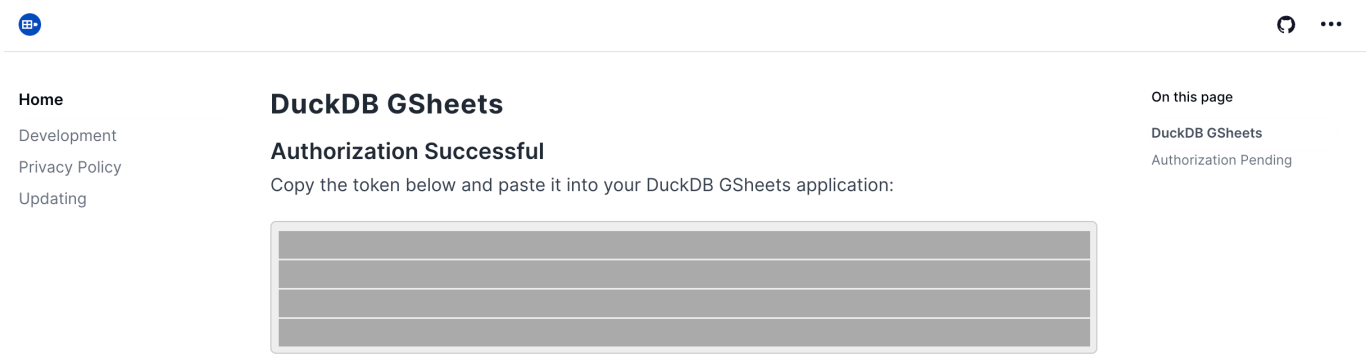
The GSheets extension was originally authored by Archie from the team at Evidence, but has since had significant contributions from Alex and Michael.

Getting Started with the GSheets Extension

The first few steps are to install the gsheets community extension and authenticate with Google.

```
INSTALL gsheets FROM community;
LOAD gsheets;

-- Authenticate with a Google Account in the browser (default)
CREATE SECRET (TYPE gsheet);
```

As a part of the CREATE SECRET command, a browser window will open and allow for a login and copying a temporary token to then paste back into DuckDB.



Examples of Reading from Sheets

Now that you are authenticated, DuckDB can query any Sheet that your Google account has access to. This includes any publicly available sheets like the one below, so give it a run!

```
FROM
'https://docs.google.com/spreadsheets/d/1B4RFuOnZ4ITZ-nR9givZ7vWVOTVddC3VTKuSqgifiyE/edit?gid=0#gid=0';
```

Gotham Wisdom
You either die a hero
or live long enough to query from spreadsheets

Copy the URL of the Sheet to query when viewing the sheet of interest within the workbook. The `gid` query string parameter is the id of that specific sheet.

There are two ways to pass in additional parameters:

1. Add them to the end of the URL as query string parameters or
2. Use the `read_gsheet` table function and specify them as separate SQL parameters.

The repository README has a variety of examples and some are included below!

> Query string parameters must be placed after a `?`. Each parameter is formatted as a `key=value` pair, and multiple are separated with &.

Reading a Specific Sheet and Range

By default, the GSheets extension will read all data on the first sheet in the workbook. The `sheet` and `range` parameters (or their query string equivalents) allow for targeted reads.

For example, to read only the first 3 cells on the `We <3 Ducks` sheet, these two statements are equivalent:

```
-- The sheet with the gid of 0 is named 'We <3 Ducks' (because of course it is!)
FROM read_gsheet(
    'https://docs.google.com/spreadsheets/d/1B4RFuOnZ4ITZ-nR9givZ7vWVOTVddC3VTKuSqgifiyE/edit',
    sheet = 'We <3 Ducks',
    range = 'A1:A3'
    );

FROM 'https://docs.google.com/spreadsheets/d/1B4RFuOnZ4ITZ-
nR9givZ7vWVOTVddC3VTKuSqgifiyE/edit?gid=0#gid=0&range=A1:A3';
```

The Google Sheets API helpfully skips empty rows at the end of a dataset or empty columns to the right. Feel free to specify a slightly bigger `range` if your data may grow! Additionally, the `range` can be specified as a set of columns (e.g., `D:X`) to be friendlier to a variable number of rows.

Data Types

The extension will sample the first row of data in the sheet to attempt to determine the data types of the columns. (We have plans to improve this sampling and are open to contributions!) To skip this step and define the data types within SQL, set the `all_varchar` parameter to `true`. The example below also demonstrates that the full URL is not needed – only the Google Workbook identifier.

```
FROM read_gsheet(
    '1B4RFuOnZ4ITZ-nR9givZ7vWVOTVddC3VTKuSqgifiyE',
    sheet = 'We <3 Ducks',
    range = 'A:A',
    all_varchar = true
    );
```

It is also possible to query data without a header row by setting the `header` parameter to false. Columns will be given default names and can be renamed in SQL.

Examples of Writing to a GSheet

Another key capability of the GSheets extension is to write the results of any DuckDB query to a Google Sheet!

> By default, the entire Sheet will be replaced with the output of the query (including a header row for column names), starting in cell A1 of the first sheet. See below for examples that adjust this behavior!

```
-- Here you will need to specify your own Sheet to experiment with!
-- (We can't predict what folks would write to a public Sheet...
-- Probably just memes, but there is always that one person, you know?)
COPY (FROM range(10))
TO 'https://docs.google.com/spreadsheets/d/...'  (
    FORMAT gsheet
);
```

Writing to a Specific Sheet and Range

As with reading, both query string parameters and SQL parameters can be used to write to a specific `sheet` or `range`. Similarly, the SQL parameters take precedence. These examples are equivalent:

```
COPY (FROM range(10))
TO 'https://docs.google.com/spreadsheets/d/...?' (
    FORMAT gsheet,
    sheet 'The sheet name!',
    range 'A2:Z10000'
);

COPY (FROM range(10))
TO 'https://docs.google.com/spreadsheets/d/...?gid=123#gid=123&range=A2:Z10000' (
    FORMAT gsheet
);
```

The `header` boolean parameter can also be used to determine whether the column names should be written out or not.

Overwriting or Appending

At times, it is helpful to avoid clearing out other data in a Sheet before copying. This is especially handy when writing to specific ranges. Perhaps columns C and D can come from DuckDB and the remainder can be spreadsheet formulas. It would be great to just clear out columns C and D!

To adjust this behavior, pass in these boolean parameters to the COPY function. `OVERWRITE_SHEET` is the default where the entire sheet is cleared out prior to copying. `OVERWRITE_RANGE` will only clear out the specified range.

If both are set to `false`, then data will be appended without any other cells being cleared out. Typically, when appending it is not desirable to include the column headers in the output. Helpfully, the `header` parameter defaults to `false` in the append case, but it can be adjusted if needed.

```
-- To append, set both flags to false.
COPY (FROM range(10))
TO 'https://docs.google.com/spreadsheets/d/...?gid=123#gid=123&range=A2:Z10000' (
    FORMAT gsheet,
    OVERWRITE_SHEET false,
    OVERWRITE_RANGE false
    -- HEADER false is the default in this case!
);
```

Automated Workflows

Working with spreadsheets is great for ad hoc work, but it can also be powerful when ingrained in automated processes. If you want to schedule an interaction with Google Sheets, a key file containing a private key will be needed instead of the in-browser authentication method.

After aquiring this key file, the persistent private key must be converted to a temporary token once every 30 minutes. That process is now automated with the `key_file` secret provider. Create the secret with a command like below, pointing to the JSON file exported from Google.

```
CREATE OR REPLACE PERSISTENT SECRET my_secret (
    TYPE gsheet,
    PROVIDER key_file,
    FILEPATH 'credentials.json'
);
```

As the secret is created, the private key is stored in DuckDB and a temporary token is created. The secret can be stored in memory or optionally persisted to disk (unencrypted) using the `PERSISTENT` keyword. The temporary token is cached within the `SECRET` as well and is recreated if it is over 30 minutes old.

This unlocks the use of the GSheets extension within pipelines, like GitHub Actions (GHA) or other orchestrators like dbt. The best practice is to store the `credentials.json` file as a secret within your orchestrator and write it out to a temporary file. An example GHA workflow is here, which uses this Python script to query a Sheet.

Developing the Extension

The Google Sheets extension is a good example of how DuckDB's extension GitHub template and CI/CD workflows can let even non-C++ experts contribute to the community! Several of the folks who have contributed thus far (thank you!!), including the authors of this post, are not traditional C++ programmers. The combination of a great template, examples from other extensions, and a little help from some LLM-powered "junior devs" made it possible. We encourage you to give your extension idea a shot and reach out on Discord if you need some help!

Roadmap

There are a few more fun features we are thinking about for the extension – we are open to PRs and collaborators!

We would like to use a better heuristic for detecting data types when reading from a Sheet. The DuckDB type system is more advanced than Sheets, so it would be beneficial to be more precise.

Enabling the GSheets extension to work in DuckDB-Wasm would allow in-browser applications to query Sheets directly – no server needed! Several `http` functions need some modification to work in a browser environment.

The OAuth flow that powers the browser-based login may be useful for authenticating to other APIs. We are wondering if maybe it would be possible to have a generic OAuth community extension. There are no concrete plans for this at the moment, but if anyone is interested, please reach out!

Closing Thoughts

At MotherDuck (where Alex works), we have this extension running in production for several internal data pipelines! We have automated exports of forecasts from our warehouse into Sheets and continually load manually collected customer support data into our (MotherDuck-powered) data warehouse. As a result, our KPI dashboards include context from folks talking directly to customers!

Michael Harris has also contributed to the extension (thank you!), and Definite has deployed GSheets scheduled jobs into production for multiple customers!

How do you use Google Sheets in your data analysis workflow and how can DuckDB help? We would love to hear your ideas on BlueSky, LinkedIn, or X / Twitter!

Now go automate that Sheet with some SQL!

Gems of DuckDB 1.2

Publication date: 2025-03-06

Author: The DuckDB team

TL;DR: We highlight a few exciting features that were introduced in DuckDB 1.2.

We published the DuckDB 1.2.1 bugfix release yesterday. As usual, please consult the release notes for the full list of changes and the installation page for instructions on installing or upgrading. In this post, we'll highlight a few features that were recently added to DuckDB and improvements that have been made in its ecosystem.

New Clients Page

DuckDB's client APIs (or "drivers") are distributed through several centralized repositories, such as CRAN for R and Maven for Java. To help users keep track of the rollout of a new DuckDB release, we reworked our "Clients" page to show the latest version for each client. The page also clarifies the support tiers that apply to clients.

Simpler Installation

In line with DuckDB's "low friction" principle, we made sure that you can install the DuckDB command line client more easily.

Installation Script on Linux and macOS

DuckDB can now be installed on UNIX-like systems with an installation script:

```
curl https://install.duckdb.org | sh
```

The script determines your operating system and architecture, fetches the tag of latest release, and if not present downloads the latest available DuckDB binary to `~/.duckdb/cli` (the `~/.duckdb` folder is already used to store extensions).

Running the script does not require root (sudo) privileges, and it only uses the `curl` and `zcat` tools, which are widely available. You can inspect the shell script by visiting `install.duckdb.org` in your browser.

Signed Binaries on Windows

Starting with v1.2.1, we ship digitally signed binaries for the DuckDB Windows command line client. This means that you can run DuckDB in environments where signed binaries are a requirement.

Unsung DuckDB 1.2 Features

Further to the "What's New in 1.2.0" section of the announcement blog post, we collected five new features that were introduced in v1.2.0.

OR / IN Filter Pushdown

Starting with version 1.2.0, DuckDB supports OR and IN expressions for filter pushdown. This optimization comes especially handy when querying remote Parquet files or DuckDB databases.

-f Command Line Flag

The DuckDB CLI client now supports the -f flag to execute SQL script files:

```
duckdb -f script.sql
```

This is equivalent to:

```
duckdb -c ".read script.sql"
```

This feature is documented in the DuckDB tldr page. If you have tldr installed, you can get this page in the CLI via `tldr duckdb`.

allowed_directories / allowed_paths Options

We continue to improve support for operating DuckDB in secure environments. The `allowed_directories` and `allowed_paths` options allow restricting DuckDB's access to certain directories or files (resp.). These options allows fine-grained access control for the file system. For example, you can set DuckDB to only use the `/tmp` directory.

```
SET allowed_directories = ['/tmp'];
SET enable_external_access = false;
FROM read_csv('test.csv');
```

With the setting applied, DuckDB will refuse to read files in the current working directory:

```
Permission Error:
Cannot access file "test.csv" - file system operations are disabled by configuration
```

sum(BOOLEAN)

You can now directly compute the sum of BOOLEAN expressions without wrapping them into a CASE expression:

```
SELECT sum(l_extendedprice > 500) FROM lineitem;
```

This is equivalent to:

```
SELECT sum(CASE WHEN l_extendedprice > 500 THEN 1 END) FROM lineitem;
```

Excel Extension

Prior to DuckDB 1.2, Excel files were only supported by the `spatial` extension, which is a heavyweight extension with several dependencies. Starting with 1.2, the `excel` extension – which was previously limited to computing a few formulas – can read and write Excel sheets. For example:

```
FROM read_xlsx('test.xlsx', header = true);
```

> If you would like to work with *Google Sheets* sheets, take a look at the `gsheets` community extension.

On the Interweb

There are several repositories and pieces of information on DuckDB on the internet. We highlight two important ones:

Awesome DuckDB

The ecosystem around DuckDB keeps growing: many projects are built both with DuckDB and within DuckDB. The community-driven `awesome-duckdb` repository, maintained by David Gasquez, lists these and has recently surpassed 200 entries.

DuckDB File Signature

DuckDB's file signature, DUCK (hex: `44 55 43 4B`), is now listed on Wikipedia.

Parquet Information Sheet

We added DuckDB to the "Implementation status" page of the Parquet documentation. In the process, we also improved DuckDB's Parquet support, e.g., by adding the `FLOAT16` logical type.

Parquet Bloom Filters in DuckDB

Publication date: 2025-03-07

Author: Hannes Mühleisen

TL;DR: DuckDB now supports reading and writing Parquet Bloom filters.

One of the key features of the Parquet file format is the ability for readers to *selectively* read only the data that is relevant to a particular query. To support this, Parquet files contain column *statistics*, most notably the minimum and maximum value for each column in each row group. If a query is filtering with a particular value, and the data is – as it is often – somewhat sorted, a reader can "prove" that a particular row group cannot contain values relevant to the query. DuckDB heavily leverages this, and is able to – even when querying remote endpoints – selectively only read the parts of a Parquet file relevant to a query. For details on how this works, see our by now ancient blog post "Querying Parquet with precision using DuckDB".

However, there are some limitation to this approach. What if a column's data is randomly shuffled? In that case, all values occur in all row groups and the minimum and maximum statistics are less useful because we can only exclude values that are outside the minimum and maximum *of the entire column*. Parquet will use dictionary encoding if there are not too many distinct values in the column. In theory, that dictionary could be used to eliminate row groups but there are two problems with this approach: Parquet (inexplicably) allows *switching* from dictionary to plain encoding *halfway through a row group*. If the dictionary alone were used to eliminate the row group, but the plain part contained the queried value, this would lead to incorrect results. Furthermore, the dictionary is part of the actual column data. If we're reading the column data just to look at the dictionary we have already incurred most of the cost of reading the column in the first place.

Obscure sidenote: Again, in theory the column metadata contains the list of encodings that occur in that column. *If* that list is correct and does *not* include the plain encoding the dictionary *could* – again, in theory – be used for slightly earlier row group elimination. But the usefulness of this approach is more than dubious.

Parquet Bloom Filters

The good people over at the Parquet PMC have recognized that there is room for improvement here and added Bloom filters for Parquet back in 2018. In a nutshell, Bloom filters are compact but approximate index structures for a set of values. For a given value, they can either say with certainty that a value is *not* in the set or that it *may be* in the set, with a false positive ratio depending on the size of the Bloom filter and the amount of distinct values added to it. For now, we can just treat a Bloom filter like an opaque series of bytes with magic properties.

When used, Parquet files can contain a Bloom filter for each column in each row group. Each Bloom filter can be at an arbitrary location in the file (`bloom_filter_offset`). At the offset in the file, we find another Thrift-encoded structure, the `BloomFilterHeader`. This structure has a field for the length of the filter, and some algorithmic settings which are currently redundant because there is only one valid setting for all of them. But decode the header you must to find out where the header ends and where the filter bytes begin. Finally, we have arrived at the precious magic bytes of the Bloom filter. We can now test the filter against any query predicates and see if we can skip the row group entirely.

More obscure sidenotes: while the column metadata *does* contain a field to store the size of the filter (`bloom_filter_length`) it is also optional and some writers (looking at you, Spark) annoyingly only set the offset, not the length. Also, the specification describes two possible filter locations. By also not requiring a length in the metadata, this makes it difficult slash impossible to read all Bloom filters for a Parquet file in a single range request. It is also not clear why the Bloom filters needed to *each* be prefixed with a Thrift metadata blob, when this information could also be contained in the ColumnMetaData. Or perhaps, god forbid, the filters could have become part of the main Parquet metadata itself. In the end we end up with a lot of additional reads to find and read the Bloom filter bytes, in principle requiring a careful trade-off between reading the filters and "just" reading the column brute-force.

DuckDB Bloom Filters

As of the last feature release (1.2.0), DuckDB supports *both reading and writing* of Parquet Bloom filters. This happens completely transparently to users, no additional action or configuration is required.

Writing

Currently, Bloom filters are supported for the following types:

- Integer types (`TINYINT`, `UTINYINT`, `SMALLINT`, `USMALLINT`, `INTEGER`, `UINTEGER`, `BIGINT`, `UBIGINT`)
- Floating point types (`FLOAT`, `DOUBLE`)
- String types (`VARCHAR` and `BLOB`)

Nested types (lists, structs, arrays) are currently *not* supported, but they might be added in a future release. In general, Bloom filters will be written if DuckDB decides to use dictionary encoding for a particular column (chunk) within the row group. There is a `COPY` parameter that controls the maximum dictionary size (`DICTIONARY_SIZE_LIMIT`), but this parameter is by default set to 10% of the row group size (`ROW_GROUP_SIZE`), which is 122,880 rows by default. Those values have been found to be reasonable first approximations for most use cases but users are of course encouraged to experiment with both parameters if they encounter performance issues. As the number of distinct values in a Bloom filter grows, its size needs to be improved to maintain a certain false positive ratio. By default, filters size is chosen using an "acceptable" false positive ratio of 1% or 0.01. The new `BLOOM_FILTER_FALSE_POSITIVE_RATIO` `COPY` parameter controls the acceptable ratio. In general, false positive hurt more the slower the read path is.

Reading

During reading, DuckDB will automatically use constant-comparison filter predicates in the query (e.g., `WHERE a = 42`) to probe the Bloom filter (if present) and skip row groups where the Bloom filter can guarantee there are no matching rows in the group. Again, this happens transparently to users and there is no configuration that needs to be set.

Users can find out if a given Parquet file contains Bloom filters: the `parquet_metadata` function was extended with two new columns, `bloom_filter_offset` and `bloom_filter_length`. Furthermore, to actually find out which row groups would be eliminated by Bloom filters for a given file and column, we have added the `parquet_bloom_probe` function. For example, `parquet_bloom_probe('file.parquet', 'col1', 42)` returns a table for each row group in `file.parquet` that indicates whether the value `42` can guaranteed not to be in each row group or not for column `col1`. Most users will not need to use these functions however, they just help with debugging (and testing).

Example Use Case

Let's showcase the Parquet Bloom filters in DuckDB with an example. First, we create an example file `filter.parquet` that *will* contain Bloom filters:

```
COPY (
    FROM range(10) r1, range(10_000_000) r2
    SELECT r1.range * 100 AS r
    ORDER BY random()
)
TO 'filter.parquet'
(FORMAT parquet, ROW_GROUP_SIZE 10_000_000);
```

```
SELECT r, count(*)
FROM 'filter.parquet'
GROUP BY r
ORDER BY r;
```

The file contains 10 distinct values `(0, 100, 200 ... 900)`, each repeated ten million times. So in total, 100 million rows. The resulting Parquet file weighs in at 88 MB.

We will also create an equivalent file but *without* Bloom filters. We achieve this by setting the `DICTIONARY_SIZE_LIMIT` to 1:

```
COPY 'filter.parquet' to 'nofilter.parquet'
(FORMAT parquet, DICTIONARY_SIZE_LIMIT 1, ROW_GROUP_SIZE 10_000_000);
```

The contents of both files will be equivalent, but `nofilter.parquet` will not use dictionary encoding and thus not use Bloom filters. As a result, the file is also larger, weighing in at 181 MB. However, there is a much larger difference when querying for non-existing values, in our example `501`:

```
.timer on
SELECT sum(r) FROM 'filter.parquet'   WHERE r = 501;
SELECT sum(r) FROM 'nofilter.parquet' WHERE r = 501;
```

The first query completes in ca. than 0.002 s, where the second query takes 0.1 s. This large difference can be explained by Bloom filters! Since `501` does not actually occur in the query, DuckDB can automatically exclude all row groups and not actually read any data besides the Bloom filters. We can explore this further using the `parquet_metadata` function:

```
FROM parquet_metadata('filter.parquet')
SELECT row_group_id, stats_min, stats_max,
    bloom_filter_offset, bloom_filter_length
ORDER BY row_group_id;
```

row_group_id	stats_min	stats_max	bloom_filter_offset	bloom_filter_length
0	0	900	92543967	47
…				
9	0	900	92544390	47

We can see that there are ten row groups, and that there is a quite compact Bloom filter for reach row group with a length of 47 bytes each. That's ca. 500 bytes added to fairly large file, so rather irrelevant for file size.

If we run the query on the other file, we can see the lack of Bloom filters:

```
FROM parquet_metadata('nofilter.parquet')
SELECT row_group_id, stats_min, stats_max,
       bloom_filter_offset, bloom_filter_length
ORDER BY row_group_id;
```

row_group_id	stats_min	stats_max	bloom_filter_offset	bloom_filter_length
0	0	900	NULL	NULL
…				

We can further explore the Bloom filters in the file using the `parquet_bloom_probe` function. For example, for the value of 500 (which exists in the data), the function shows the following:

```
FROM parquet_bloom_probe('filter.parquet', 'r', 500);
```

file_name	row_group_id	bloom_filter_excludes
filter.parquet	0	false
…	…	…

file_name	row_group_id	bloom_filter_excludes
filter.parquet	9	false

So the Bloom filter cannot exclude any row group because the value `500` is contained in all row groups. But if we try a *non-existent* value, the Bloom filter strikes:

```
FROM parquet_bloom_probe('filter.parquet', 'r', 501);
```

file_name	row_group_id	bloom_filter_excludes
filter.parquet	0	true
…	…	…
filter.parquet	9	true

Here, we can confidently skip all row groups because the Bloom filter guarantees that there can be no matching values in those row groups. All that with 47 bytes per row group.

Conclusion

DuckDB's new Bloom filter support for Parquet files can greatly reduce the amount of data to be read in certain scenarios, greatly improving query performance. This is particularly useful if files are read over a slow network connection or if row groups are particularly large with few distinct yet non-clustered values.

The DuckDB Local UI

Publication date: 2025-03-12

Authors: Jeff Raymakers and Gabor Szarnyas

TL;DR: The DuckDB team and MotherDuck are excited to announce the release of a local UI for DuckDB shipped as part of the `ui` extension.

The DuckDB project was built to make it simple to leverage modern database technology. DuckDB can be used from many popular languages and runs on a wide variety of platforms. The included Command Line Interface (CLI) provides a convenient way to interactively run SQL queries from a terminal window, and several third-party tools offer more sophisticated UIs.

The DuckDB CLI provides advanced features like interactive multi-line editing, auto-complete, and progress indicators. However, it can be cumbersome for working with lengthy SQL queries, and its data exploration tools are limited. Many of the available third party UIs are great, but selecting, installing, and configuring one is not straightforward. Using DuckDB through a UI should be as simple as using the CLI. And now it is!

> The DuckDB UI is the result of a collaboration between DuckDB Labs and MotherDuck and is shipped as part of the `ui` extension.

Introducing the DuckDB UI

Starting with DuckDB v1.2.1, a full-featured local web user interface is available out-of-the-box! You can start it from the terminal by launching the DuckDB CLI client with the `-ui` argument:

```
duckdb -ui
```

You can also run the following SQL command from a DuckDB client (e.g., CLI, Python, Java, etc.):

```
CALL start_ui();
```

Both of these approaches install the `ui` extension (if it isn't installed yet), then open the DuckDB UI in your browser:

The DuckDB UI uses interactive notebooks to define SQL scripts and show the results of queries. However, its capabilities go far beyond this. Let's go over its main features.

> The DuckDB UI runs all your queries locally: your queries and data never leave your computer. If you would like to use MotherDuck through the UI, you have to opt-in explicitly.

Features

Databases

Your attached databases are shown on the left. This list includes in-memory databases plus any files and URLs you've loaded. You can explore tables and views by expanding databases and schemas.

Table Summary

Click on a table or view to show a summary below. The UI shows the number of rows, the name and type of each column, and a profile of the data in each column.

Select a column to see a more detailed summary of its data. You can use the *"Preview data"* button near the top right to inspect the first 100 rows. You can also find the SQL definition of the table or view here.

Notebooks

You can organize your work into named notebooks. Each cell of the notebook can execute one or more SQL statements. The UI supports syntax highlighting and autocomplete to assist with writing your queries.

You can run the whole cell, or just a selection, then sort, filter, or further transform the results using the provided controls.

Column Explorer

The right panel contains the Column Explorer, which shows a summary of your results. You can dive into each column to gain insights.

MotherDuck Integration

If you would like to connect to MotherDuck, you can sign into MotherDuck to persist files and tables to a cloud data warehouse crafted for using DuckDB at scale and sharing data with your team.

...And More!

The DuckDB UI is under active development. Expect additions and improvements!

Footprint

Like the DuckDB CLI, the DuckDB UI creates some files in the `.duckdb` directory in your home directory. The UI puts its files in a sub-directory, `extension_data/ui`:

- Your notebooks and some other state are stored in a DuckDB database, `ui.db`.
- When you export data to the clipboard or a file (using the controls below the results), some tiny intermediate files (e.g., `ui_export.csv`) are generated. Your data is cleared from these files after the export is completed, but some near-empty files remain, one per file type.

Internals

Support for the UI is implemented in a DuckDB extension. The extension embeds a localhost HTTP server, which serves the UI browser application, and also exposes an API for communication with DuckDB. In this way, the UI leverages the native DuckDB instance from which it was started, enabling full access to your local memory, compute, and file system.

Results are returned in an efficient binary form closely matching DuckDB's in-memory representation (DataChunk). Server-sent events enable prompt notification of updates such as attaching databases. These techniques and others make for a low-latency experience that keeps you in your flow.

See the UI extension documentation for more details.

Summary

In this blog post, we presented the new DuckDB UI, a powerful web interface for DuckDB.

The DuckDB UI shares many of its design principles with the DuckDB database. It's simple, fast, feature-rich, and portable, and runs locally on your computer. The DuckDB UI extension is also open source: visit the `duckdb/duckdb-ui` repository if you want to dive in deeper into the extension's code.

> The repository does not contain the source code for the frontend, which is currently not available as open-source. Releasing it as open-source is under consideration.

For help or to share feedback, please file an issue, join the `#ui` channel in either the DuckDB Discord or the MotherDuck Community Slack.

Happy quacking!

Update (2025-03-20)

We are very happy to see the overwhelmingly positive response to the announcement of the DuckDB Local UI. In this short update, we'd like to address some of the questions raised by the community:

- Regarding the **licensing of the frontend code** (HTML, JavaScript, etc.), MotherDuck is currently reviewing licensing options for the UI code and assets and will clarify shortly.
- Regarding **offline ("air-gapped") operations,** we are working towards supporting offline usage in a future version, and are currently exploring how to provide version notifications as upgrades become available.
- Regarding the **client-server protocol for the UI,** we're going to work on opening up the client-server protocol so that other user interfaces can use the "back-end" part of the UI extension to run queries in DuckDB.

Preview: Amazon S3 Tables in DuckDB

Publication date: 2025-03-14

Authors: Sam Ansmink, Tom Ebergen, Gabor Szarnyas

TL;DR: We are happy to announce a new preview feature that adds support for Apache Iceberg REST Catalogs, enabling DuckDB users to connect to Amazon S3 Tables and Amazon SageMaker Lakehouse with ease.

> The AWS Storage Blog also published a post on this feature, see Streamlining access to tabular datasets stored in Amazon S3 Tables with DuckDB.

Iceberg Ahead!

In recent years, the Iceberg open table format has become increasingly popular. Major data warehouse platforms such as Databricks, Snowflake, Google BigQuery and AWS have all announced or already implemented support for Iceberg tables. These platforms also support Iceberg catalogs, which are responsible for tracking current metadata for a collection of Iceberg tables grouped by namespaces.

DuckDB has supported reading Iceberg tables since September 2023 via the `iceberg` extension. Today, we are happy to introduce a new preview feature in this extension, which allows attaching to Iceberg REST catalogs. This preview release coincides with two AWS announcements yesterday: support for Iceberg tables in Amazon S3 Tables and the GA release of the integration between S3 Tables and SageMaker Lakehouse (AWS Glue Data Catalog). In practice, these developments mean that DuckDB now provides an end-to-end solution for reading Iceberg tables in S3 Tables and SageMaker Lakehouse.

> DuckDB's support for Iceberg REST Catalog endpoints in Amazon S3 Tables is the result of a collaboration between AWS and DuckDB Labs.

Using Apache Iceberg REST Catalogs in DuckDB

Steps for Installing

To connect to Apache Iceberg REST Catalogs in DuckDB, make sure you are running the **latest stable** DuckDB release (version 1.2.1). For our example steps, we'll use the DuckDB CLI client. You can obtain this client from the installation page and start it with:

```
duckdb
```

Next, we need to install the "bleeding edge" versions of the required extensions from the `core_nightly` repository.

```
FORCE INSTALL aws FROM core_nightly;
FORCE INSTALL httpfs FROM core_nightly;
FORCE INSTALL iceberg FROM core_nightly;
```

> For more information on using the `core_nightly` repository, please see the notes at the end of the post.

With these extensions installed, your DuckDB is now capable of using Apache Iceberg REST Catalogs. Let's find some data.

Setting up an Amazon S3 Table Bucket

(If you already have Iceberg tables in Amazon S3 Tables, you can skip to the "Reading Iceberg Catalogs with DuckDB" section.)

In this post, we demonstrate how to read data from Amazon S3 Tables. To follow along, make sure that your account has `s3tables` permissions and create a new S3 table bucket. Note that Amazon S3 Tables is currently only supported in selected AWS regions.

Populating an Amazon S3 Table Bucket

If you don't have an S3 table bucket with tables already, we found the easiest way to get going is to create a table using Amazon Athena. See their instructions. For our example, we created a simple table with three columns using the Athena query editor:

```
CREATE TABLE duck_species (
    id INT,
    english_name STRING,
    latin_name STRING
) TBLPROPERTIES ('table_type' = 'ICEBERG');
```

Let's insert some data to the table:

```
INSERT INTO duck_species VALUES
    (0, 'Anas nivis', 'Snow duck');
```

Reading Amazon S3 Tables with DuckDB

Querying S3 Tables with DuckDB is really easy. The first step is to get your AWS credentials into DuckDB. You can achieve this in two ways. First, you can let DuckDB detect your AWS credentials and configuration based on the default profile in your `~/.aws` directory by creating the following secret using the Secrets Manager:

```
CREATE SECRET (
    TYPE s3,
    PROVIDER credential_chain
);
```

Alternatively, you can set the AWS key, secret, and region values manually. For example:

```
CREATE SECRET (
    TYPE s3,
    KEY_ID '<AKIAIOSFODNN7EXAMPLE>',
    SECRET '<wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY>',
    REGION '<us-east-1>'
);
```

> **Tip.** To see the secrets in your session, run `FROM duckdb_secrets();`{:.language-sql .highlight}

Next, point DuckDB to your S3 table bucket. You can do so by copy-pasting the S3 Tables ARN value directly from the AWS Management Console and using it in the `ATTACH` command:

```
ATTACH 'arn:aws:s3tables:<us-east-1>:<111122223333>:bucket/<bucket-name>'
    AS s3_tables_db (
        TYPE iceberg,
        ENDPOINT_TYPE s3_tables
    );
```

And that's all! Now, DuckDB is connected to Amazon S3 Tables. To show the available tables, run:

```
SHOW ALL TABLES;
```

database varchar	schema varchar	name varchar	column_names varchar[]	column_types varchar[]	temporary boolean
s3_tables_db	ducks	duck_species	[__]	[INTEGER]	false

You can query tables as if they were ordinary DuckDB tables:

```
FROM s3_tables_db.ducks.duck_species;
```

id int32	english_name varchar	latin_name varchar
0	Anas nivis	Snow duck

You also have an alternative option to connect to S3 Tables using the Amazon SageMaker Lakehouse (AWS Glue Data Catalog) Iceberg REST Catalog endpoint. To do so, run:

```
ATTACH '<account_id>:s3tablescatalog/<namespace_name>'
AS (
    TYPE iceberg,
    ENDPOINT_TYPE glue
);
```

> **Tip.** If you need basic read access to tabular data in a single S3 table bucket, use the `s3_tables` endpoint type. If you want a unified view across all of your tabular data in AWS, use the `glue` endpoint type.

Schema Evolution

A key feature of the Iceberg format is schema evolution, i.e., the ability to follow changes in the table's schema. To demonstrate this, we go back to the Athena query editor and add a new column to the `duck_species` table:

```
ALTER TABLE duck_species
    ADD COLUMNS (conservation_status STRING);
```

Then, we insert a few more duck species:

```
INSERT INTO duck_species VALUES
    (1, 'Anas eatoni', 'Eaton''s pintail', 'Vulnerable'),
    (2, 'Histrionicus histrionicus', 'Harlequin duck', 'Least concern');
```

Let's run the query again from DuckDB:

```
FROM s3_tables_db.ducks.duck_species;
```

The query now returns a table with the additional fourth column, which has a `NULL` value in the row inserted before the change in the schema – as expected.

id int32	english_name varchar	latin_name varchar	conservation_status varchar
1	Anas eatoni	Eaton's pintail	Vulnerable
2	Histrionicus histrionicus	Harlequin duck	Least concern
0	Anas nivis	Snow duck	NULL

Conclusion

The latest preview release of the DuckDB `iceberg` extension enables directly reading tables using Iceberg REST endpoints. This allows you to query Amazon S3 Tables and Amazon SageMaker Lakehouse (AWS Glue Data Catalog) with ease. As of today, the extension is in an experimental state and is under active development. We will publish a stable release later this year.

Footnotes

Cleaning Up

If you created a new S3 table bucket to follow the examples, don't forget to clean up by deleting your S3 table bucket.

Using the `core_nightly` Repository

The extensions used in this blog post are currently experimental, and hence they are distributed through the `core_nightly` repository. If you want to switch back to using extensions from the `core` repository, follow the extension documentation.

Note that DuckDB does not support reloading extensions. Therefore, if you experience any issues, try restarting DuckDB after updating the extensions.

Maximizing Your Delta Scan Performance in DuckDB

Publication date: 2025-03-21

Author: Sam Ansmink

TL;DR: We released a new version of the `delta` extension, which includes several new features and performance improvements. In this blog post, we'll put the `delta` extension through its paces with some benchmarks and take a deep dive into some of the new performance-related features.

Overview

In our previous post, we talked about what the Delta Lake table format is all about, and how DuckDB's `delta` extension leverages the Delta Kernel library to offer native support. In this blog post, we will focus on how to get the most performance out of reading Delta tables from DuckDB. We'll start with a small recap of Delta, then demonstrate the performance gains that were made over the past few releases. Finally, we demonstrate three key features available in the latest Delta release that will ensure you get the most out of your Delta reading performance: metadata caching, file skipping and partition information pushdown.

The Delta Open Table Format

Let's start off with a small recap of Delta to get back up to speed. Delta Lake is an Open Table Format similar to Apache Iceberg and Apache Hudi. Open Table Formats are best understood as "a collection of data and metadata files" that aim to provide the flexibility of data lakes while providing some of the consistency guarantees of traditional data warehouses. In the case of Delta, the format consists of Parquet files for data and a mix of Parquet, JSON and binary files for metadata. Besides providing an improved level of consistency, the additional metadata provided by Open Table Formats allows various types of performance optimizations through things like column statistics and file skipping. For a more in-depth explanation, we refer to the previous Delta blog post.

The `delta` Extension

DuckDB natively supports reading Delta Tables through the `delta` extension. This extension is one of the core DuckDB extensions with >70k weekly downloads. Using this extension to read from a Delta Table is really simple. Since DuckDB v1.2.0, the `delta` extension will be automatically installed upon first use and loaded when invoking the `delta_scan` function.

So for example, to read a local Delta table, simply open any DuckDB client and run:

```
SELECT * FROM delta_scan('./<path_to_your_delta_table>');
```

Is your Delta table on someone else's machine, perhaps in AWS? DuckDB can also query straight from S3! To have DuckDB automatically load your AWS credentials and query a remote Delta table, run:

```
CREATE SECRET (TYPE s3, PROVIDER credential_chain);
SELECT * FROM delta_scan('s3://<your-bucket>/<your_delta_table>');
```

For other cloud providers such as Azure or Google Cloud, check the extension's documentation page.

Performance Improvements between `delta` v0.1.0 and 0.3.0

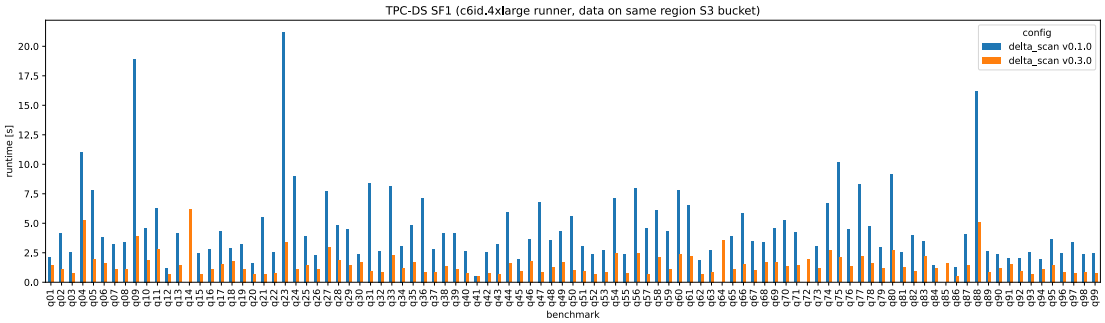
While the first release (v0.1.0) of the `delta` extension already came with various performance-related features such as projection pushdown and constant filter pushdown, the features added since then have massively improved the performance of `delta_scan`. To illustrate this, our first benchmark will use the industry standard TPC-DS benchmark with the scale factor 1 data set (SF1).

Benchmark Setup

For this benchmark, we started up an AWS c6id.4xlarge instance (16 vCPUs, 32 GB RAM) and wrote the TPC-DS SF1 dataset to an S3 bucket in the same region (eu-west-1) using PySpark. Each benchmark is run a total of 6 times with the result being the median runtime of the last 5 runs with the first being considered a cold run. The aggregated results are shown in the following table.

Result	Total runtime	Min runtime	Max runtime	Median runtime	Queries timed out
`delta` extension v0.1.0	444.76	0.48	21.31	3.63	4
`delta` extension v0.3.0	151.06	0.46	6.15	1.22	0

The detailed results of the benchmark are shown in the foldout: Detailed TPC-DS SF1 benchmark results, `delta` extension v0.1.0 vs. v0.3.0



Analysis

We can see from the results that there has been a significant performance improvement all around. In v0.1.0, 4 out of 99 queries hit the benchmark timeout of 30 sec and were excluded from the results. In v0.3.0, all 99 queries completed well within the timeout. Comparing total runtimes (excluding the queries that timed out for v0.1.0) we find a speedup of more than 3×!

Now, without going into too much detail, an important part of the speedup here can be attributed to the **cardinality information propagation** that was added in PR #77. Having accurate cardinality estimation is essential for DuckDB's query optimizer to work well and produce efficient query plans. Specifically, DuckDB's Join Optimizer uses the cardinality estimates to change the order in which joins are performed. The join order can massively impact the cardinality of the intermediate tuples, which has a big impact on query performance. Especially in workloads like the TPC-DS benchmark, which has many queries that contain a lot of joins, the Join Order Optimizer plays a crucial role. For (a lot) more details, check out this thesis.

Further Optimizations

Attaching Delta Tables

Besides the general performance improvements like cardinality information propagation, several performance-related features have been added to the `delta` extension as well. One of those is the ability to attach Delta tables. Using ATTACH to query a Delta table has multiple

advantages. For starters, by using ATTACH, your queries can look a little cleaner when querying the same table multiple times since you don't have to repeat the full Delta table path every time it is queried. More importantly, though, using ATTACH will allow DuckDB to cache/reuse certain parts of the Delta metadata, which can improve query performance. To attach the local Delta table, run:

```
ATTACH './<path_to_your_delta_table>' AS <your_table> (TYPE delta);
```

After attaching the Delta table, you can query the table just by using the alias:

```
SELECT * FROM <your_table>;
```

By default, DuckDB will automatically cache Delta metadata within the same transaction. That means if a Delta table is scanned multiple times in that transaction, DuckDB can reuse parts of the Delta metadata between the different scans. For example, the following query will only read the Delta metadata once:

```
SELECT * FROM t1
UNION ALL
SELECT * FROM t1;
```

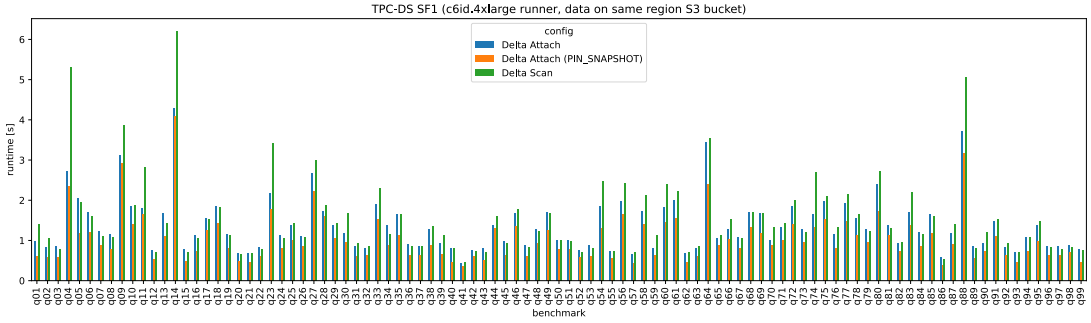
For even more performance improvements, DuckDB also supports **persisting this cached Delta metadata *between* different queries**. To do this, the Delta table can be attached using the PIN_SNAPSHOT option. With this option enabled, subsequent queries can reuse the metadata such as in the following code block:

```
ATTACH 's3://<your-bucket>/<your_delta_table>' AS t2 (
    TYPE delta,
    PIN_SNAPSHOT
);

-- First scan (metadata not yet cached)
SELECT * FROM t1;

-- Second scan (metadata is now cached)
SELECT * FROM t2;
```

Metadata caching can have a significant performance impact, especially in situations where the data is relatively small and the latency high. To illustrate this, we will rerun our TPC-DS experiment to compare three different ways of scanning the Delta table: using `delta_scan`, using ATTACH, and using `ATTACH ... (PIN_SNAPSHOT)`. The rest of the benchmark setup is identical to the one in the previous section.

result	total runtime	min runtime	max runtime	median runtime
`delta_scan`	151.06	0.46	6.15	1.22
ATTACH	134.26	0.43	4.28	1.19
ATTACH (PIN_SNAPSHOT)	102.80	0.36	4.04	0.87

The detailed results of the benchmark are shown in the foldout: Detailed TPC-DS SF1 benchmark results with different configurations



The results show that for many TPC-DS queries, using ATTACH instead of `delta_scan` can already slightly improve performance for several queries, with overall runtime seeing a 1.13× speedup. When the metadata is fully in cache due to `PIN_SNAPSHOT`, we see an even greater speedup of 1.47×. This, however, comes at the tradeoff of missing any updates to the table that occur after the ATTACH statement.

A keen eye looking at the full results will also spot a few cases where the ATTACH results are actually slightly worse than the results with raw `delta_scan`. This is something we will explain in the section on the pushdown / ATTACH interplay.

File Skipping

Another key performance feature of scanning Delta tables is file skipping. As explained in the introduction, Delta tables contain metadata that contains all sorts of statistics of the data files of the table. These statistics can be used by engines like DuckDB to decide which Parquet files need to be scanned and which ones can be skipped altogether. File skipping is something that is done automatically by DuckDB. File skipping will work for both constant filters and dynamic filters (filters that are calculated during query execution):

```
-- constant filter
FROM delta_scan('...')
WHERE col_a > 'some_value';
-- dynamic filter
FROM delta_scan('...')
WHERE col_a > (SELECT max(col_z) FROM other_tbl);
```

In the previous benchmarks, file skipping has only a very limited effect. The overall data is simply not big enough and many queries actually touch large parts of the data anyway. However, when only a relatively small part of the data is touched in a query, file skipping can have a huge impact on performance. To demonstrate this, we will first generate some test data. We'll use the same PySpark-based test data generation script as before.

The table has 100 million rows and has a very basic schema with an incrementing `id` column of `INTEGER` type and a `value` column of VARCHAR type. If we query the data using DuckDB we will see something like:

```
FROM delta_scan('s3://<your-bucket>/<your_delta_table>');
```

```
┌──────────┬──────────────┐
│    id    │    value     │
│  int64   │   varchar    │
├──────────┼──────────────┤
│ 49950000 │ val-49950000 │
│ 49950001 │ val-49950001 │
│ 49950002 │ val-49950002 │
│ 49950003 │ val-49950003 │
│     ·    │      ·       │
│     ·    │      ·       │
│     ·    │      ·       │
│   49996  │ val-49996    │
│   49997  │ val-49997    │
│   49998  │ val-49998    │
│   49999  │ val-49999    │
├──────────┴──────────────┤
│ 100000000 rows          │
│      (8 shown)          │
└─────────────────────────┘
```

Now, let's say we are only interested in a specific range of `id`s: maybe we only want `id`s below 100. We will now construct two queries.

For the first query, we will directly read all the parquet files stored in the table using a glob pattern:

```
FROM parquet_scan('s3://<your-bucket>/<your_delta_table>/*.parquet')
WHERE id < 100;
```

> We are doing this for illustrative purposes to show us the benefits of file skipping, scanning the raw Parquet files in a Delta table like this only works here because we have no updates, deletes or checkpoints in this table.

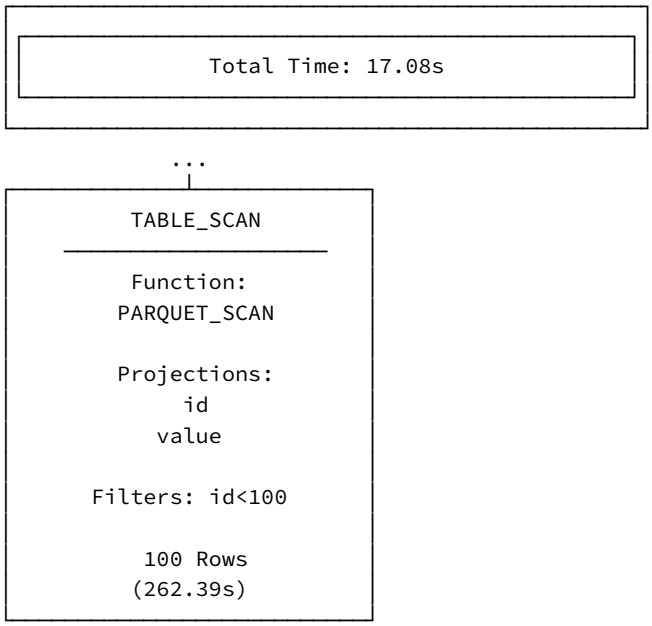
For the second query, we directly scan the table using the `delta_scan` table function, selecting only the `id`s we are interested in using a `WHERE` clause:

```
FROM delta_scan('s3://<your-bucket>/<your_delta_table>')
WHERE id < 100;
```

Now when running these queries from a c6id.4xlarge AWS instance on the S3 bucket in the same region, we can see that they perform wildly differently. The `delta_scan` only requires ≈0.5 seconds to complete, whereas the `parquet_scan` takes ≈17 seconds. So what's going on here exactly?

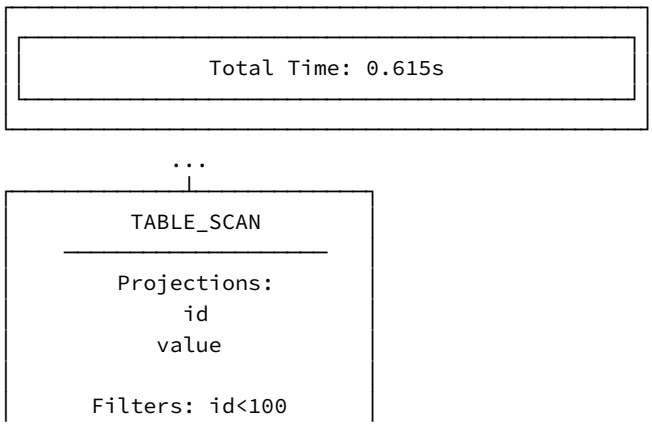
We can use DuckDB's `EXPLAIN ANALYZE` statement to get more details here. Let's start by analyzing the `parquet_scan`:

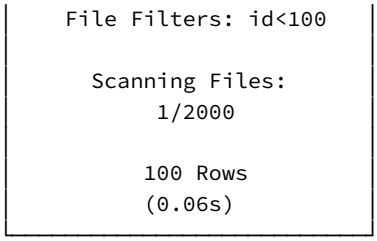
```
EXPLAIN ANALYZE
FROM parquet_scan('s3://<your-bucket>/<your_delta_table>/*.parquet')
WHERE id < 100;
```



We can see in the `EXPLAIN ANALYZE` output that our filter was properly pushed down, and the scan correctly only produced 100 rows. This all seems fine, right? Well, let's compare it with the output of `EXPLAIN ANALYZE` for the `delta_scan`:

```
EXPLAIN ANALYZE
FROM delta_scan('s3://<your-bucket>/<your_delta_table>');
```





For the `delta_scan` function's `EXPLAIN ANALYZE` output, we can see two new fields: `File Filters` and `Scanning Files`. This shows us clearly what's going on. The `id<100` predicate is now used for two things: it's pushed down into the scans on the individual Parquet files, just like the `parquet_scan`, but it also shows up as a file filter, which is used to reduce the list of files to be scanned altogether! This leads to a **2,000× reduction** of the amount of Parquet metadata to be read, which results in a huge performance boost.

Partition Information Pushdown

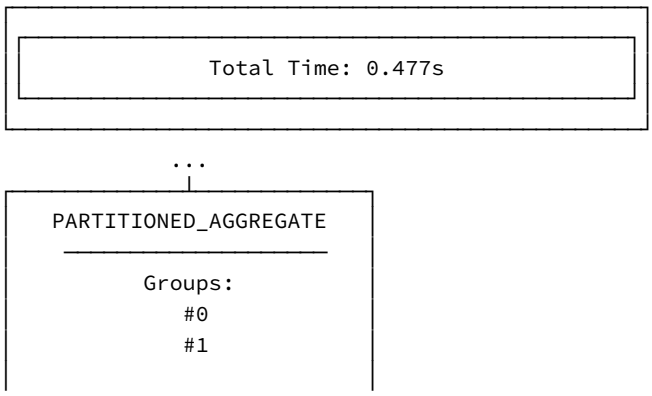
The final DuckDB Delta performance feature is partition information pushdown. Partition information pushdown and the partition-aware aggregation operator are relatively new features introduced in DuckDB v1.2.0. In the v0.3.0 release of the `delta` extension this feature was also added, which means that DuckDB can now use the partitioning information to create query plans that can utilize the fact that the data scanned is already partitioned. To show the performance benefit of partition information, we will, *surprise,* run another benchmark! This time, we chose the TPC-H dataset at scale factor 10 and ran the experiment on a 32 GB MacBook Pro M1 Max. We partitioned the `lineitem` table by the `l_returnflag` and `l_linestatus` columns. We then run Q1 which looks roughly like this:

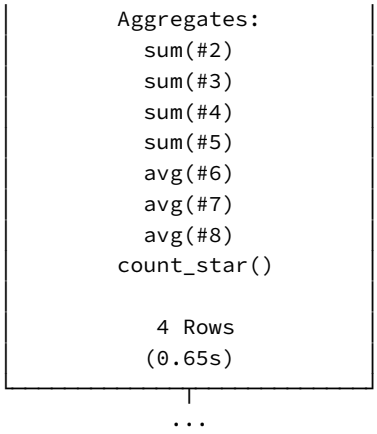
```
SELECT
    l_returnflag,
    l_linestatus,
    sum(l_quantity) AS sum_qty,
    ...
FROM
    lineitem
    ...
GROUP BY
    l_returnflag,
    l_linestatus
    ...;
```

Note that the query contains a GROUP BY statement, which lists the exact columns by which our dataset is already partitioned. Making DuckDB use partitioning-aware operators is done all automatically, so in this case simply running:

```
ATTACH './<path_to_partitioned_directory>/lineitem_sf10' AS lineitem (
    TYPE delta
);
PRAGMA tpch(1);
```

will fire off the TPC-H Q1 on the partitioned Delta dataset. To inspect what's happening we will again use `EXPLAIN ANALYZE`:



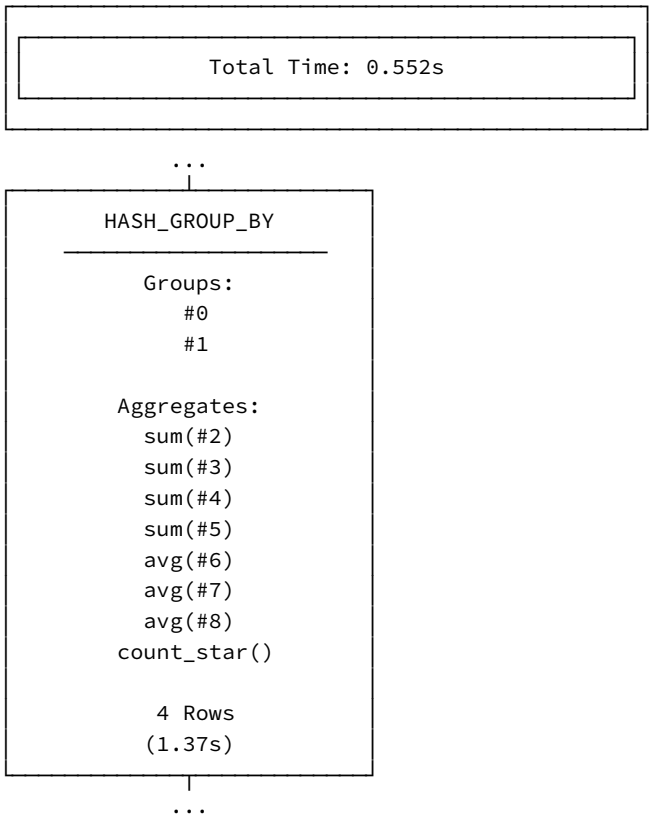


We can see that DuckDB has detected the partitioning information correctly and is using the PARTITIONED_AGGREGATE operator to do the GROUP BY efficiently.

Now, as a baseline, we will rerun the same query, but now with partition information pushdown disabled:

```
ATTACH './<path_to_partitioned_directory>/lineitem_sf10' AS lineitem (
    TYPE delta,
    PUSHDOWN_PARTITION_INFO 0
);
PRAGMA tpch(1);
```

Again, using the EXPLAIN ANALYZE, we can see that DuckDB will now use a regular HASH_GROUP_BY operator, since the partition information from Delta was not available during query planning.



Now looking at the performance differences between these two queries, we can see that the overall speedup is only a modest 1.16×, however the aggregation operation itself was sped up by 2.11×! This means that when queries that regularly do heavy group by operations, partitioning the data by these columns can definitely be a very useful tool to have in your performance tuning toolbox.

A Note on Pushdown / ATTACH Performance Interplay

While features such as filter pushdown and partition information pushdown will improve performance for many workloads, it is useful to be aware that there is a somewhat intricate interplay between the metadata caching mechanism from using ATTACH, and the pushdown of filters and partition information. At the end of the section on the ATTACH feature, we already saw that for some queries, using ATTACH is actually slightly slower than using the raw `delta_scan`. Without diving into too much detail, pushing down filters and partitioning information can negatively affect the effectiveness of metadata caching for some queries. This means that on some queries, you may, somewhat counterintuitively, benefit from partially disabling filter pushdown when using ATTACH:

```
ATTACH './<your_delta_table_directory>' AS dt (
    TYPE delta,
    PIN_SNAPSHOT,
    PUSHDOWN_PARTITION_INFO 0,
    PUSHDOWN_FILTERS 'none'
);
```

This should be considered an advanced use case, though, and only relevant when optimizing for specific queries. The default settings of ATTACH should provide the best overall performance and are recommended for most cases. Furthermore, there is ongoing work in the underlying `delta-kernel-rs` library used by DuckDB Delta that aims to reduce this effect by exposing mechanisms to cleverly refresh metadata objects held by DuckDB. As soon as these mechanisms are available, we will add them to the DuckDB `delta` extension, likely making these flags obsolete for anything but testing.

Conclusion

In this blog post, we've taken a look at the latest version of the DuckDB `delta` extension and put it to the test with some benchmarks. We've run queries from the industry standard TPC benchmarks to demonstrate the large performance improvements that were made over the past releases of the `delta` extension.

Furthermore, we've taken a look at three specific techniques that can be used while working with Delta tables to improve performance even further:

- Using metadata caching with ATTACH
- Using filters and data layout to reduce the number of files that need to be scanned
- Utilizing partitioning information to speed up aggregations

All in all, we think with the v0.3.0 release of the `delta` extension, DuckDB can read Delta tables with excellent performance for many different workloads and highly encourage everyone to give the latest version a shot!

Using DuckDB in Streamlit

Publication date: 2025-03-28

Author: Petrica Leuca

TL;DR: We use a real-world railway dataset to demonstrate the integration of DuckDB and Streamlit, including the database connection management, the DuckDB Python relational API and responsiveness in interactive map charts.

Introduction

In the article Analyzing Railway Traffic in the Netherlands, Gabor has shown how to analyze data with SQL and reading remote files with DuckDB, by using open data from the Rijden de Treinen *(Are the trains running?)* application.

Using the above open data, in this post we will build an application, in which a user can:

- Understand the railway traffic data;
- Visualize the railway network traffic in 2024, by using Plotly;
- Visualize the five closest train stations on any point selected on the map, by using the Streamlit Folium component.

The application will be developed with Streamlit, an open source framework in which one can implement data applications easily with Python. The application code is available on GitHub.

Connecting to DuckDB in Streamlit

To load the railway traffic data, we write a function, which creates the two tables we will be working with, `services` and `stations`:

```
def prepare_duckdb(duckdb_conn):

    duckdb_conn.sql("""
        create table if not exists services as
        from 'https://blobs.duckdb.org/nl-railway/services-2024.csv.gz'
    """)

    duckdb_conn.sql("""
        create table if not exists stations as
        from 'https://blobs.duckdb.org/nl-railway/stations-2023-09.csv'
    """)
```

From a Streamlit application there are three ways to connect to DuckDB:

- In-memory connection;
- Persisted local file connection;
- Attaching an external database.

In-Memory Connection

Connecting to an in-memory database involves loading the data into memory, whenever a DuckDB connection is established. Depending on the resources of the application server and the amount of data, creating a new in-memory DuckDB connection and loading the required data on every request will slow down the application. In Streamlit this behavior is addressed by caching the resource, which can be done as:

- caching the DuckDB connection as a global shared connection in the application;
- caching the DuckDB connection for the user session.

```
@st.cache_resource(ttl=datetime.timedelta(hours=1), max_entries=2)
def get_duckdb_memory(session_id):
    """
    Set a caching resource which will be refreshed
     - either at each hour
     - either at each third call
     - either when the connection is established for a new session_id
    """

    duckdb_conn = duckdb.connect()
    prepare_duckdb(duckdb_conn=duckdb_conn)

    return duckdb_conn
```

Persisted Local File Connection

A DuckDB connection can be established against a persisted database file on the server of the application. The database file can be created:

- during the deployment, before the application starts;
- a standalone process, which creates the database and copies it to the application server. Depending on the freshness requirement of the application, one might schedule a process to refresh the data.

```
duckdb_conn = duckdb.connect(
    "train_stations_and_services.duckdb",
    read_only=True
)
```

Attaching an External Database

Another way to connect to DuckDB is to establish a **read only** connection to an instance over HTTPS or S3 compatible APIs, e.g., DUCKDB_EXTERNAL_LOCATION = "https://blobs.duckdb.org/nl-railway/train_stations_and_services.duckdb":

```
duckdb_conn = duckdb.connect()
duckdb_conn.execute(f"attach '{DUCKDB_EXTERNAL_LOCATION}' as ext_db")
duckdb_conn.execute("use ext_db")
```

More details about attaching an external database can be found in the DuckDB documentation.

Considerations

1. DuckDB works best if you have allocated 1-4GB of memory per thread, therefore configure the appropriate resources when deploying the Streamlit application.

2. The recommendation is to use a new connection for each database interaction or at least one at user session level, and not a global connection at application level. Using a global connection might result in unexpected behaviour, depending on the number of users using the application at the same time.

3. When using a persisted local database, take into account that when there is a database connection established no other process can write to it. One workaround is to have a process which creates and replaces the database file on the server.

4. When using an attached database, take into account that the application can have a performance impact depending on the storage solution chosen.

Starting from this point, we will work with the external database.

Analyzing Dutch Railway Data

To analyze the data within Streamlit, we will make use of the Python relational API, by chaining queries. We start by defining a relation, which will select the needed data from `stations`:

```
stations_selection = duckdb_conn.sql("""
    select name_long as station_name, geo_lat, geo_lng, code
    from stations
""").set_alias("stations_selection")
```

Then we define a relation for the `services` selection:

```
services_selection = (
    duckdb_conn.sql("from services")
    .aggregate("""
        station_code: "Stop:Station code",
        service_date: "Service:Date",
        service_date_format: strftime(service_date, '%d-%b (%A)'),
        num_services: count(*)
    """)
    .set_alias("services")
)
```

We then join the two relations to select the station name, the geolocation and other details we will be using further, bringing the above code together in a function.

```
def get_stations_services_query(duckdb_conn):

    # create a relation for the station selection
    stations_selection = ...

    # create a relation for the services selection
    services_selection = ...

    # return the query and the duckdb_conn
    return (
        stations_selection
        .join(
            services_selection,
            "services.station_code = stations_selection.code"
        )
        .select("""
            service_date,
            service_date_format,
            station_name,
            geo_lat,
```

```
            geo_lng,
            num_services
        """)
    ), duckdb_conn
```

When we run `get_stations_services_query(get_duckdb_conn())`, there is yet no data retrieved from the database. This is because the query evaluation is lazy, which means that the query is not executed against the database until an execution command is encountered.

The query will be executed when one of the following methods is encountered:

- `stations_query.df()` to extract to a Pandas dataframe;
- `stations_query.fetchall()` to extract in a list;
- `stations_query.write_to()` to export the data in a file;
- any other calculation method, such as `.sum`, `.row_number` etc. The relational methods can be found in the DuckDB documentation.

Top 5 Busiest Train Stations per Month

To extract the 5 busiest train stations per month, we start from the above query and add a `row_number` calculation with a final filter of the ranking to be less or equal to 5:

```
stations_query, _ = get_stations_services_query(get_duckdb_conn())

top_5_query = (
    stations_query.aggregate("""
            station_name,
            service_month: monthname(service_date),
            service_month_id: month(service_date),
            num_services: sum(num_services)
        """)
    .select("""
            station_name,
            service_month,
            service_month_id,
            num_services,
            rn: row_number()
                over (
                    partition by service_month
                    order by num_services desc
                )
        """)
    .filter("rn <= 5")
    .order("service_month_id, station_name")
)
```

Streamlit provides a series of charts, such as line and bar, but, unfortunately, they do not offer many personalization possibilities. Therefore, the above chart was generated with Plotly, an open source graphing library, integrated with Streamlit by calling `st.plotly_chart`.

Railway Network Utilization Analysis

The data about the train rides through the train stations of The Netherlands, at day level, is a perfect input for analyzing the utilization of the railway network across the country and during the year.

Railway Network Utilization across the Country

To analyze the network utilization across the country, we use `density_map` chart from Plotly, which will generate a heat map on a map chart. Because heat maps work best with pre-aggregated data, we first aggregate the number of train services and the geolocation of the station:

```
stations_df = stations_query.aggregate(
    "geo_lat, geo_lng, num_services: sum(num_services)"
).df()
```

It is interesting to observe that the north-east of the country is barely covered by railway and that, while Groningen and The Hague appear to be used during the year, neither of their train stations was in the top 5 busiest train stations per month.

Animating the Heat Map From the above static chart, we can easily generate an animation with Plotly, by providing the animation frame parameter. In our case, we generate an animation with the railway network utilization at service day level during July 2024:

Railway Network Utilization during the Year

From a year point of view, we are curious to discover how busy the network is in each month and each day of the week, which is better displayed through a heat map. We prepare the input for the chart, by pivoting the number of services by the month name of the service date and by grouping by the ISO day name of the week. At the moment, there is no pivot method in the Python relational API, but we can use the SQL query generated by the relational API as source in the pivot raw SQL:

```
@st.cache_data(ttl=3600, max_entries=100)

def get_stations_services_data(_duckdb_conn):
    query = _duckdb_conn.sql("from services").aggregate("""
            service_day: dayname("Service:Date"),
            service_day_isodow: isodow("Service:Date"),
            service_month: monthname("Service:Date"),
            num_services: count(distinct "Service:RDT-ID")
        """)

    return (
        _duckdb_conn.sql(f"""
            pivot ({query.sql_query()})
            on service_month
            using sum(num_services)
            group by service_day, service_day_isodow
            order by service_day_isodow
        """)
        .select(
            "January",
            "February",
            "March",
            "April",
            "May",
            "June",
            "July",
            "August",
            "September",
            "October",
            "November",
            "December",
            "service_day",
        )
        .df()
    )
```

We have decided to create a function, in order to highlight `cache_data` functionality in Streamlit. While the above query is executed in 300 ms on a MacBook Pro with 12 GB memory, we want to cache the results because the result of the query does not change often. `cache_data` is used in Streamlit to cache the results of a function and just as `cache_resource`, it has the attributes time to live and max entries. The two caching mechanisms require that the input parameters passed to the function to be hashable. Since the DuckDB connection object is not hashable, we can tell Streamlit to ignore it by adding the prefix `_`.

> It is important to notice that the order of columns and values matters in a heat map chart and that we need to reset the index of the pivoted data before plotting.

The code of implementing interactive maps with Plotly in Streamlit is available on GitHub.

Finding the Closest 5 Stations with Folium

One other use-case for the railway network data is to find the closest train stations to a location selected by the user on the map. While Streamlit and Plotly have the functionality to register click events on the chart, they work only when the user is selecting points which are part of the chart data. Therefore, we will use `streamlit-folium`, a Streamlit component wrapped around Folium, which can register the clicks on any point of the map.

To work with geo data in DuckDB we have to install and load the spatial extension:

```
duckdb_conn.sql("install spatial;")
duckdb_conn.sql("load spatial;")
```

We then define a function to retrieve the query, which returns the closest five train stations to a location, composed of latitude and longitude.

```
def get_closest_stations_query(duckdb_conn, lat, lng):

    stations_selection = duckdb_conn.sql("""
        select name_long as station_name, geo_lat, geo_lng, code
        from stations st
        where exists (
                select count(*)
                from services sv
                where st.code = sv."Stop:Station code"
                having count(*) > 100
            )
    """)

    return (
        stations_selection.project(f"""
            code as station_code,
            station_name,
            geo_lat,
            geo_lng,
            station_geo_point: st_point(geo_lng, geo_lat),
            clicked_geo_point: st_point({lng}, {lat}),
            distance_in_m: st_distance_sphere(
                            st_point(geo_lng, geo_lat),
                            clicked_geo_point
                        ),
            distance_in_km: round(distance_in_m/1000,2)
        """)
        .order("distance_in_km")
        .limit(5)
    )
```

In the above query we use `st_point` to create a point type field and `st_distance_sphere` to get the distance, in meters, between two points.

> It is important to mention that while we usually refer to a geo point as `[latitude, longitude]`, when working with spatial extensions we usually create the point as `[longitude, latitude]`. This applies to PostGIS as well.

When a user clicks on the map, we store the clicks in the session state and rerun Streamlit to display the map with the new user selection. When rerun happens, Streamlit will rerun the entire application for the current session, therefore understanding the session state and caching mechanisms is essential when building Streamlit applications.

The code of implementing interactive maps with Folium in Streamlit is available on GitHub.

Conclusion

In this post, we illustrated how DuckDB can be used in Streamlit, how the Python relational API can make our code cleaner by chaining queries and how to implement interactive maps with Plotly and Folium, while leveraging the Streamlit caching mechanism.

Fully Local Data Transformation with dbt and DuckDB

Publication date: 2025-04-04

Author: Petrica Leuca

TL;DR: In this post, we implement data transformation and reverse ETL pipelines with DuckDB and dbt using the `dbt-duckdb` adapter.

Introduction

The Data Build Tool, `dbt`, is an open-source transformation framework, which enables data teams to adopt software engineering best practices in the code they deliver, such as Git workflow and unit testing. Other notable features of `dbt` are data lineage, documentation and data testing as part of the execution pipeline.

In this article we will demonstrate how to perform fully local data transformations with dbt and DuckDB. To this end, we'll use the `dbt-duckdb` adapter, which integrates the `dbt` standard and with DuckDB's processing power.

Data Model

We use two open datasets in this post: railway services, provided by the team behind the Rijden de Treinen *(Are the trains running?)* application and cartography information about the Netherlands provided by cartomap. The datasets are organized into:

- a persisted DuckDB database, in which we store the railway services data from 2024, hosted on Cloudflare;
- a provinces GeoJSON file, containing the geographic information about the Dutch provinces, hosted on GitHub;
- a municipalities GeoJSON file, containing the geographic information about Dutch municipalities, stored together with the code.

After doing an initial exploration of the above data, we can observe that a province can have one or many municipalities, a municipality can have none or many train stations and a train service record is connected to one and only one train station. Therefore we decide on the following data model:

- a dimension table, `dim_nl_provinces`, to hold information about the Dutch provinces;
- a dimension table, `dim_nl_municipalities`, to hold information about the Dutch municipalities, linked to `dim_nl_provinces`;
- a dimension table, `dim_nl_train_stations`, to hold information about the train stations in the Netherlands, linked to `dim_nl_municipalities`;
- a fact table, `fact_services`, to hold information about the train services, linked to `dim_nl_train_stations`.

The purpose of processing the train services and the Netherlands cartography data is to organize the data in a structure which can easily be used in future railway data analysis use cases. Such curated data structures are often called *data marts*.

> **Tip.** The following code is available on GitHub.

Processing Data with DuckDB and dbt

After we have initialized our project, we configure the connection details for DuckDB in the `profiles.yml` file. Along with specifying if the database should be in memory or persisted to disk, we also specify:

- which extensions are required for the data processing, e.g., spatial;

- external databases, attached from the local disk or other storage solutions.

```
dutch_railway_network:

  outputs:
    dev:
      type: duckdb
      path: data/dutch_railway_network.duckdb
      extensions:
        - spatial
        - httpfs
      threads: 5
      attach:
        - path: 'https://blobs.duckdb.org/nl-railway/train_stations_and_services.duckdb'
          type: duckdb
          alias: external_db
  target: dev
```

We then configure the `sources.yml` file under the `models` directory by specifying external sources (such as files) and table definitions from the attached database(s):

```
version: 2
sources:
  - name: geojson_external
    tables:
      - name: nl_provinces
        config:
          external_location: "https://cartomap.github.io/nl/wgs84/provincie_2025.geojson"
      - name: nl_municipalities
        config:
          external_location: "seeds/gemeente_2025.geojson"
  - name: external_db
    database: external_db
    schema: main
    tables:
      - name: stations
      - name: services
```

> The `external_location` can point to CSV, Parquet or JSON files. Both the local file system and remote endpoints (e.g., HTTP or S3) are supported.

With both the profile and source defined, we can now load the data.

Loading Data to DuckDB

In `dbt` the way of storing the data in the target system is called `materialization`. The `dbt-duckdb` adapter provides the following materialization options:

- `table`, replacing the target table at each run;
- `incremental`, with `append` and `delete+insert` options, modifying the data in the table, but not the table itself (if it exists);
- `snapshot`, implementing a Slowly Changing Dimension Type 2 table with time validity intervals;
- `view`, replacing the target view at each run.

Another feature of using the above adapter is that the SQL dialect used in the data processing scripts has all the friendly SQL extensions DuckDB has to offer.

To refresh the data in the `dim_nl_provinces` table, we use the `st_read` spatial function, which reads and parses automatically the `nl_provinces` GeoJSON file, defined in `sources.yml`.

{% raw %}

```
{{ config(materialized='table') }}

SELECT
    {{ dbt_utils.generate_surrogate_key(['id']) }} AS province_sk,
    id AS province_id,
    statnaam AS province_name,
    geom AS province_geometry,
    {{ common_columns() }}
FROM st_read({{ source("geojson_external", "nl_provinces") }}) AS src;
```

{% endraw %}

Similarly, we fully refresh the data for `dim_nl_municipalities` and `fact_services`.

To build the relation between a train station location and a municipality, we use the `st_contains` spatial function, which returns `true` when a geometry contains another geometry:

{% raw %}

```
{{ config(materialized='table') }}

SELECT
    {{ dbt_utils.generate_surrogate_key(['tr_st.code']) }} AS station_sk,
    tr_st.id AS station_id,
    tr_st.code AS station_code,
    tr_st.name_long AS station_name,
    tr_st.type AS station_type,
    st_point(tr_st.geo_lng, tr_st.geo_lat) AS station_geo_location,
    coalesce(dim_mun.municipality_sk, 'unknown') AS municipality_sk,
    {{ common_columns() }}
FROM {{ source("external_db", "stations") }} AS tr_st
LEFT JOIN {{ ref ("dim_nl_municipalities") }} AS dim_mun
       ON st_contains(
           dim_mun.municipality_geometry,
           st_point(tr_st.geo_lng, tr_st.geo_lat)
       )
WHERE tr_st.country = 'NL';
```

{% endraw %}

> To read from the external sources, we reference the source by providing the source and table names.

Exporting Data from DuckDB

One major benefit of using DuckDB for data processing is the ability to export data to files (such as CSV, JSON and Parquet) and to refresh data directly in PostgreSQL or MySQL databases.

External Files

The feature of exporting data to files is enabled by the `dbt-duckdb` adapter with the `external` materialization. With the `external` materialization, we are able to export data to CSV, JSON and Parquet file types to a specified storage location (local or external). The load type is `full refresh`, therefore existing files are overwritten.

In the following processing step, we export aggregated train service data at month level, to a Parquet file, partitioned by year and month:

{% raw %}

```
{{
    config(
```

```
        materialized='external',
        location="data/exports/nl_train_services_aggregate",
        options={
            "partition_by": "service_year, service_month",
            "overwrite": True
        }
    )
}}

SELECT
    year(service_date) AS service_year,
    month(service_date) AS service_month,
    service_type,
    service_company,
    tr_st.station_sk,
    tr_st.station_name,
    m.municipality_sk,
    m.municipality_name,
    p.province_sk,
    p.province_name,
    count(*) AS number_of_rides
FROM {{ ref ("fact_services") }} AS srv
INNER JOIN {{ ref("dim_nl_train_stations") }} AS tr_st
        ON srv.station_sk = tr_st.station_sk
INNER JOIN {{ ref("dim_nl_municipalities") }} AS m
        ON tr_st.municipality_sk = m.municipality_sk
INNER JOIN {{ ref("dim_nl_provinces") }} AS p
        ON m.province_sk = p.province_sk
WHERE service_year = {{ var('execution_year') }}
GROUP BY ALL
```

{% endraw %}

The exported files are placed in Hive partitioned directory structure.

```
./service_year=2024/service_month=1:
49255 Apr  2 14:54 data_0.parquet

...

./service_year=2024/service_month=12:
48031 Apr  2 14:54 data_0.parquet
```

PostgreSQL

After we process the data into our railway services data mart, we can generate from it a daily aggregate at train station level, organized into a star schema model, by having the dimension keys part of the data:

{% raw %}

```
{{
    config(
        materialized='incremental',
        incremental_strategy='delete+insert',
        unique_key="""
            service_date,
            service_type,
            service_company,
            station_sk
        """
```

```
    )
}}

SELECT
    service_date,
    service_type,
    service_company,
    srv.station_sk,
    mn.municipality_sk,
    province_sk,
    count(*) AS number_of_rides,
    {{ common_columns() }}
FROM {{ ref ("fact_services") }} AS srv
INNER JOIN {{ ref("rep_dim_nl_train_stations") }} AS tr_st
        ON srv.station_sk = tr_st.station_sk
INNER JOIN {{ ref("rep_dim_nl_municipalities") }} AS mn
        ON tr_st.municipality_sk = mn.municipality_sk
WHERE NOT service_arrival_cancelled

  {% if is_incremental() %}
    AND srv.invocation_id = (
        SELECT invocation_id
        FROM {{ ref("fact_services") }}
        ORDER BY last_updated_dt DESC
        LIMIT 1
    )
  {% endif %}
GROUP BY ALL
```

{% endraw %}

Thanks to DuckDB's ability to connect and write to a PostgreSQL database, we can add the above processing step to our `dbt` project, under the directory `models/reverse_etl`.

To connect to a PostgreSQL database, we need to specify in `profiles.yml`:

- the `postgres` extension;
- the PostgreSQL connection string in the `attach` section.

{% raw %}

```
dutch_railway_network:

  outputs:
    dev:
      type: duckdb
      path: data/dutch_railway_network.duckdb
      extensions:
        - ...
        - postgres
      threads: 5
      attach:
        - ...
        - path: "postgresql://postgres:{{ env_var('DBT_DUCKDB_PG_PWD') }}@localhost:5466/postgres"
          type: postgres
          alias: postgres_db
  target: dev
```

{% endraw %}

We also have to configure the model's database details in `dbt_project.yml`:

```
models:
  dutch_railway_network:
    transformation:
      schema: main
      +docs:
        node_color: 'silver'
    reverse_etl:
      database: postgres_db
      schema: public
      +docs:
        node_color: '#d5b85a'
```

With this configuration, all the models from the `transformation` directory will be executed on the `main_main` schema, while the models from `reverse_etl` will be executed on the `main_public` schema.

After executing the models with `dbt run --model +reverse_etl`, the data is available to query from PostgreSQL:

```
psql -U postgres
```

```
SELECT count(*), sum(number_of_rides)
FROM main_public.rep_fact_train_services_daily_agg;
```

```
 count  |   sum
--------+----------
 240826 | 17438151
```

> It is important to mention that while PostgreSQL is the target database and `dbt` provides `merge` as an incremental strategy for it; the execution of the above pipeline is happening in DuckDB, therefore the incremental load can be done only with `append` or `delete+insert` strategies.

Execution Details

The above implementation consists of 10 models and 20 data tests, processing 400 MB of data from the attached DuckDB database, together with small data organized in GeoJSON files. The total execution time, on a single thread and on a MacBook Pro of 12 GB, is between 40 and 45 seconds. From the total execution time, approximatively 30 seconds are spent to process the train services data and 4 seconds to write the aggregated data to PostgreSQL:

```
05:48:07  Running with dbt=1.9.3
05:48:08  Registered adapter: duckdb=1.9.2
05:48:08  Found 10 models, 20 data tests, 4 sources, 565 macros
05:48:08
05:48:08  Concurrency: 1 threads (target='dev')
...
05:48:45  19 of 30 OK created sql table model main_main.fact_services ..................... [OK in 32.60s]
...
05:48:50  26 of 30 OK created sql incremental model postgres_db.main_public.rep_fact_train_services_
daily_agg  [OK in 3.74s]
...
05:48:51  Finished running 2 external models, 1 incremental model, 7 table models, 20 data tests in 0
hours 0 minutes and 42.63 seconds (42.63s).
05:48:51
05:48:51  Completed successfully
05:48:51
05:48:51  Done. PASS=30 WARN=0 ERROR=0 SKIP=0 TOTAL=30
```

Conclusion

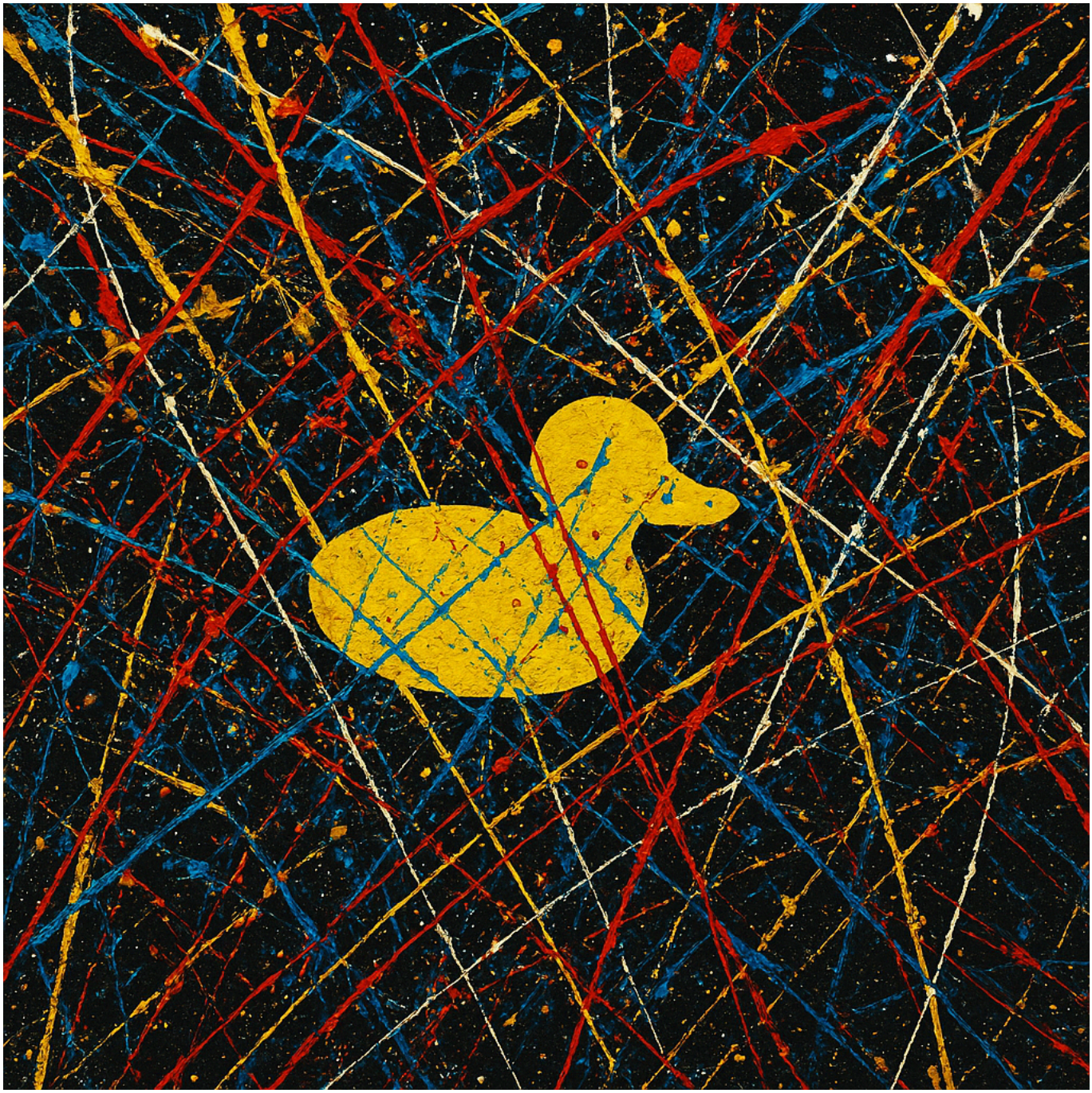
In this post we have demonstrated how DuckDB integrates with `dbt` and how it is part of the data processing ecosystem by showcasing data mart creation, file exports and reverse ETL to a PostgreSQL database.

DuckDB's CSV Reader and the Pollock Robustness Benchmark: Into the CSV Abyss

Publication date: 2025-04-16

Authors: Pedro Holanda, Gabor Szarnyas (DuckDB Labs); Gerardo Vitagliano (MIT)

TL;DR: DuckDB ships with a fast and robust CSV reader, which – we believe – can consume most CSV files found in the wild. To empirically evaluate this, we used the Pollock Benchmark, a state-of-the-art test suite designed to measure how well CSV readers can operate on non-standard files, and found that DuckDB ranks #1.



The Sorry State of the CSV Landscape

It is a well-known fact that CSV files come in all shapes and forms. While there is a well-defined standard, it is common for systems to export data without following even the basic rules of CSV construction. For example, Rabobank, one of the biggest banks in the Netherlands, exports the financial data of its clients with unescaped quotes in quoted values.

If a big financial institute cannot write proper CSV files, imagine the errors occurring in CSV files exported by decades-old legacy software, in hand-written CSV files, files exported from spreadsheet software, and so on! These sorts of errors are so prevalent that spreadsheet software, dataframe libraries and specialized CSV tools go above and beyond to salvage these files and hammer them into tables that users can work with.

DuckDB's CSV Parser

When we designed the DuckDB CSV parser, we strived to ensure that it's not only easy to use and fast, but also extremely reliable – to the point it can read most CSVs thrown at it. In this blog post, we study this from two angles. First, we explore the main options that can help you read non-standard CSV files and show their effects on parsing with some simple examples. Second, we evaluate how robust DuckDB's CSV parser is by putting it to the test with the Pollock Benchmark.

> There is a lot of discourse on the internet about whether a system should go above and beyond to read files that do not follow the standard. And yes, non-standard files are frustrating. We can all complain about them on the internet or decide to ignore them. But we strongly believe that ignoring reality comes at a cost. Being able to read non-standard files – even if it might result in ambiguity – should be the user's decision.

Running Example

As our running example, we'll use a CSV file called `cafes.csv`:

```
ZIP,Name,Country
1014,"Cafe ""Gedoogt""",Netherlands
```

```
1015,"Cafe "De Tuin" Amsterdam",Netherlands
1095,Joost
1872,Cafe Gezellig,Netherlands,5
```

Despite its small size, the file contains many different errors. Let's inspect it line by line:

Line	Row	Comment
1	`ZIP,Name,Country`	The header of the file.
2	`1014,"Cafe ""Gedoogt""",Netherlands`	A well-defined line, with a quoted value that contains escaped quotes ("").
3	`1015,"Cafe "De Tuin" Amsterdam",Netherlands`	The value for `Name` contains unescaped quotes.
4	`1095,Joost`	This line is missing the value for the `Country`.
5	`1872,Cafe Gezellig,Netherlands,5`	Has an extra column with the value `5`.

Loading the Example Naïvely

If we naïvely try to read our example, we get a table with only a single line:

```
FROM read_csv('cafes.csv');
```

column0 int64	column1 varchar	column2 varchar	column3 int64
1872	Cafe Gezellig	Netherlands	5

We can see that the reader ignored all lines except the last one. Let's query the CSV sniffer to see the dialect it detected:

```
SELECT Delimiter, Quote, Escape, SkipRows
FROM sniff_csv('cafes.csv');
```

Delimiter varchar	Quote int16	Escape varchar	SkipRows uint16
,	\0	\0	4

We can see that the sniffer decided that quotes and escape characters are not present in the file (represented as `\0` values). Consequently, it decided to skip the first 4 rows (including the header) in order to read the file according to the dialect.

> **Tip.** DuckDB allows users to retrieve all errors that occur when reading a CSV file using the `reject_errors` table. This is beyond the scope of this blog post, but you can find more details in the "Reading Faulty CSV Files" documentation page.

Loading the Example with a User-Defined CSV Dialect

Let's attempt to read the faulty CSV files with a user-defined CSV dialect. To this end, we'll set the `auto_detect` option to `false` and manually specify the presence of the `header`, the values of the `delim`(iter), `quote`, and `escape` options, as well as the schema with the `columns` option.

```
FROM read_csv('cafes.csv',
    auto_detect = false,
    header = true,
```

```
    delim = ',',
    quote = '"',
    escape = '"',
    columns = {'ZIP': 'INT16', 'Name': 'VARCHAR', 'Country': 'VARCHAR'}
);
```

Running this SQL statement will result in an error due to unescaped quotes in line 3: the parser can't determine whether the first quote in `"De Tuin"` marks the end of the quoted value or not:

```
Invalid Input Error:
CSV Error on Line: 3
Original Line: 1015,"Cafe "De Tuin" Amsterdam",Netherlands
Value with unterminated quote found.

Possible fixes:
* Disable the parser's strict mode (strict_mode=false) to allow reading rows that do not comply with the
CSV standard.
* Enable ignore errors (ignore_errors=true) to skip this row
* Set quote to empty or to a different value (e.g., quote='')
```

The error message informs us that there was a value with an *unterminated quote,* caused by the unescaped quote character. It also suggests some possible fixes, which will guide our attempts in the next sections.

Disabling Strict Mode

As the error message informed us, one of the ways to relax the constraints of the DuckDB CSV reader to read this file is by utilizing the `strict_mode` option. Namely, disabling the `strict_mode` option will allow the CSV Reader to go over the following common errors:

1. Unescaped quoted values, such as the one in line 3.
2. Rows that have too many columns. For example, for the row `1872,Cafe Gezellig,Netherlands,5`, the last column is ignored to fit our schema of three columns.
3. A mix of newline delimiters, e.g., files having both `\n` and `\r\n` as newline delimiters. This error does not occur in our running example.

By default, `strict_mode` is set to `true`, which implies that DuckDB will not attempt to parse rows that do not fit the dialect. However, setting it to `false` allows the CSV reader to attempt to read the file even if it cannot be correctly read under the given configuration. Let's go back to our example and turn off strict mode:

```
FROM read_csv('cafes.csv',
    auto_detect = false,
    header = true,
    strict_mode = false,
    delim = ',',
    quote = '"',
    escape = '"',
    columns = {'ZIP': 'INT16', 'Name': 'VARCHAR', 'Country': 'VARCHAR'}
);
```

We can see that the reader now only errors at line 4, indicating the previous lines were correctly read up to that point:

```
Invalid Input Error:
CSV Error on Line: 4
Original Line: 1095,Joost
Expected Number of Columns: 3 Found: 2
Possible fixes:
* Enable null padding (null_padding=true) to replace missing values with NULL
* Enable ignore errors (ignore_errors=true) to skip this row
```

> It is important to note two things for the `strict_mode = false` option:
>
> - When using this option, there is no guarantee that the result is correct. This is because it's impossible to define what constitutes a "correct result" when parsing a non-standard CSV file.
> - DuckDB runs a best-effort parser, as it will try only fail if it encounters an error to which it can't find any reasonable guess of what the correct data should look like (e.g., imagine a column of integers, where some integers are represented as a spelled-out string, this will cause a casting error). While it is possible that, with a given combination of errors, our parser may still fail, we are striving to make it even more robust as we face the abyss.

Ignoring Errors

The other option suggested by the error message was to set `ignore_errors = true`. This option simply means that any rows which do not fit the chosen dialect or schema will be skipped from the result. In practice, if we add it to our query:

```
FROM read_csv('cafes.csv',
    auto_detect = false,
    header = true,
    strict_mode = false,
    delim = ',',
    quote = '"',
    escape = '"',
    columns = {'ZIP': 'INT16', 'Name': 'VARCHAR', 'Country': 'VARCHAR'},
    ignore_errors = true
);
```

We get an almost complete result! Only missing line number 4 `1095,Joost` since it is missing a column.

ZIP int16	Name varchar	Country varchar
1014	Cafe "Gedoogt"	Netherlands
1015	Cafe De Tuin Amsterdam	Netherlands
1872	Cafe Gezellig	Netherlands

> The `ignore_errors` option also affects the sniffer's behavior, as it will ignore lines that produce errors during sniffing. In general, the configuration that results in the fewest errors will be preferred.

Null Padding for Missing Values

The last option that can help when reading non-standard CSV files is the `null_padding` option. This option is useful when lines in the CSV file have an inconsistent number of values, with some missing values for given columns. Let's read the file with the `null_padding = true` option.

```
FROM read_csv('cafes.csv',
    auto_detect = false,
    header = true,
    strict_mode = false,
    delim = ',',
    quote = '"',
    escape = '"',
    columns = {'ZIP': 'INT16', 'Name': 'VARCHAR', 'Country': 'VARCHAR'},
    null_padding = true
);
```

This will produce the following result:

ZIP int16	Name varchar	Country varchar
1014	Cafe "Gedoogt"	Netherlands
1015	Cafe De Tuin Amsterdam	Netherlands
1095	Joost	NULL
1872	Cafe Gezellig	Netherlands

Minimal Configuration to Read the Example File

Whew. We loaded the file but the `read_csv` call is about 200 characters long! Luckily, we can do better than that: if we set both `strict_mode = false` and `null_padding = true`, we only need to specify the `quote` and `escape` values. This call only requires about 80 characters and gets a valid result from this CSV file:

```
FROM read_csv('cafes.csv',
    strict_mode = false,
    null_padding = true,
    quote = '"',
    escape = '"'
);
```

ZIP int64	Name varchar	Country varchar	column3 int64
1014	Cafe "Gedoogt"	Netherlands	NULL
1015	Cafe De Tuin Amsterdam	Netherlands	NULL
1095	Joost	NULL	NULL
1872	Cafe Gezellig	Netherlands	5

With this configuration, the sniffer actually creates a new column (`column3`) to accommodate the extra value in the last line. This example shows that even on a CSV file exhibiting several issues, DuckDB's CSV sniffer can still make reasonably good decisions on the file's dialect and schema.

> **Tip.** You can drop the last column by e.g. using `SELECT #1, #2, #3 FROM ...` in the query.

The Pollock Benchmark

The Pollock Benchmark is a CSV data loading benchmark designed to evaluate how robust a CSV system is when reading non-standard CSV files. The benchmark was published at VLDB 2023 and is completely open-source, with a DuckDB entry recently added to its repository.

The authors of the Pollock benchmark analyzed over 245,000 public CSV datasets to understand the most common ways the RFC-4180 standard is violated in CSV files in the wild. After identifying the most common error types, they created a CSV file generator with a pollution mechanism, which inserts these errors. In total, the benchmark tool generates over 2,200 polluted files. It also generates the correct dialect configuration that should be used to read these files, along with a clean version of the file.

To evaluate the robustness of systems, the benchmark tool reads the polluted file with the configuration set for a given system-under-test. It then uses the system to write a new file with the answer and compares it with the clean version. This comparison yields a score that reflects how accurately the system has read the file. There are mainly two scores that the benchmark produces: the simple score and the weighted score, with the weighted score accounting for how commonly that type of error appears in real life.

In this section, we will describe the most common errors depicted in the paper and share the results of the benchmark after adding DuckDB.

Common Errors

The paper depicts many of the common errors that appear in CSV files. In this section, we will briefly discuss some of the most common errors, but refer to the paper for a complete overview:

1. **Inconsistent Number of Cells in Rows:** RFC-4180 requires that all rows, including the header, have a consistent number of columns, but many files have too many or too few delimiters per row.
2. **Non-Standard Newline Sequences:** RFC-4180 requires that files use a carriage return followed by a line feed (`\r\n`), but many files use only line feeds, carriage returns, or a mix of them.
3. **Multiple Header Lines:** The standard allows for an optional single header line but many files have multiple headers.
4. **Incorrectly Quoted or Unescaped Cells:** This error occurs when quoted values have unescaped quotes in them. See the `file_escape_char_0x00.csv` for an example. Note how GitHub's table renderer, which can read the clean version of this file fails to parse it.
5. **Files with Multibyte Delimiters:** The standard requires files to be comma-separated. Although single-byte delimiters are widely accepted, multibyte delimiters are common.
6. **Files with Multiple Tables:** When a CSV file stores multiple tables with different schemas in the same file.

Methodology

The benchmark includes a wide variety of systems being tested, such as CSV parsing frameworks (e.g., CleverCSV), relational database systems (e.g., PostgreSQL), spreadsheet software (e.g., LibreOffice Calc), data visualization tools (e.g., Dataviz), and dataframe libraries (e.g., Pandas).

An important aspect of the benchmark is that it provides the dialect and schema for each file. Without this, systems that do not have a sniffer (e.g., PostgreSQL) would not be able to read the file. There is no specific rule or differentiation in how much each system utilizes that information. For example, Pandas only takes partial advantage of these settings. To incorporate these differences in our evaluation of DuckDB, we decided to add two different configurations:

1. **DuckDB (benchmark config).** Under this configuration, all the options in the configuration file that are relevant to DuckDB – such as CSV dialect and schema – are passed to the reader. In addition, we also set all the options described in the previous section (i.e., `null_padding = true, strict_mode = false, ignore_errors = true`). This essentially tells us how much we can read from these files if the users manually set the necessary options.
2. **DuckDB (auto-detect only).** Under this configuration, we do not take advantage of providing custom configuration files. The only options set are those that allow for non-standard file reading (i.e., `null_padding = true, strict_mode = false, ignore_errors = true`). Hence, this option also evaluates the full power of our sniffer in scenarios of uncertainty.

Pollock Scores

The following table presents the results with DuckDB included. We also limited the original table to show only the systems with the best scores in each category (e.g., parsing frameworks, relational systems, and so on). For both the simple and weighted scores, DuckDB with all options set (the default configuration for the Pollock benchmark) is the clear winner. In terms of the simple score, this means that DuckDB correctly read `99.61%` of the data from all the files. It also correctly handles the most common errors, as reflected in the weighted score.

Pollock scores sorted by weighted score (out of 10):

System under test	Pollock score (weighted)	Pollock score (simple)
DuckDB 1.2 (benchmark config)	9.599	9.961
“SpreadDesktop”	9.597	9.929
Pandas 1.4.3	9.431	9.895

System under test	Pollock score (weighted)	Pollock score (simple)
“SpreadWeb”	9.431	9.721
SQLite 3.39.0	9.375	9.955
DuckDB 1.2 (auto-detect only)	8.439	9.075
UniVocity 2.9.1	7.936	9.939
LibreOffice Calc 7.3.6	7.833	9.925
Dataviz	5.152	5.003

> According to the benchmark's website, “SpreadDesktop” is a desktop-based commercial spreadsheet software and “SpreadWeb” is a web-based spreadsheet software. Their real names were omitted due to licensing.

As expected, running DuckDB in full auto mode scores lower, since the sniffer must detect dialects and schemas by itself. Some files even have a multibyte delimiter, which is supported by DuckDB, but is not part of the sniffer's search space, yielding a lower score. In these cases, DuckDB still managed to read about `90.75%` of the data correctly, reaching in 9.075 as the total score and 8.439 as the total weighted score. Once again, this result comes from simply calling `read_csv('file_path', null_padding = true, strict_mode = false, ignore_errors = true)` with zero user input regarding the actual data configuration, demonstrating that DuckDB's CSV reader can indeed read most non-standard CSV files, even with minimal configuration!

Contributing to the Benchmark

It's entirely possible that your favorite CSV reader system is not yet included in the benchmark results. DuckDB, for example, was initially not included, but we found that adding it was quite easy! We hope that the DuckDB pull request serves as inspiration for those who wish to add their favorite systems to it. The reproduction of the results is quite simple too. Unlike most benchmarks that focus on performance and correctness, the Pollock Benchmark measures *accuracy,* making it easy to reproduce independently of the machine used.

Happy hacking!

Temporal Analysis with Stream Windowing Functions in DuckDB

Publication date: 2025-05-02

Authors: Petrica Leuca, Gabor Szarnyas

TL;DR: DuckDB can perform time-based analytics using windows with different semantics (e.g., tumbling, hopping and sliding windows). In this post, we demonstrate these by detecting trends and anomalies in the railway service at Amsterdam Centraal Station.

Introduction

In data platforms, we usually categorize the data into dimension and fact data. While dimensions contain information about entities (name, address, serial number, etc.), facts contain events related to such entities (clicks, sales, bank transactions, readings from IoT devices, etc.). In general, fact data includes a timestamp attribute, denoting the moment when the event happened (or was observed).

When the timestamped data is processed on a streaming platform, it is often processed with *stream windowing functions* in order to organize the data into time windows. In this post, we will show how to apply stream windows on static timestamped fact data in DuckDB, as part of a data analysis task to compute train service summaries, trends and interruptions at Amsterdam Centraal Station.

> In a future post, we'll cover streaming design patterns with DuckDB.

For the current implementation, we will use the DuckDB database created in the dbt project detailed in the article "Fully Local Data Transformation with dbt and DuckDB", based on the open data from the Rijden de Treinen *(Are the trains running?)* application. We start by attaching (in any DuckDB session) the database from our storage location.

```
ATTACH 'http://blobs.duckdb.org/data/dutch_railway_network.duckdb';
USE dutch_railway_network.main_main;
```

> **Warning.** The database is rather big (approx. 1.2 GB), therefore make sure to have a stable internet connection. Instead of attaching the database, you can also download the database file and connect to it from the command line:
>
> ```
> duckdb dutch_railway_network.duckdb -cmd 'USE main_main'
> ```

Tumbling Windows

Tumbling windows are fixed-size **[left-closed, right-open)** time intervals, used to calculate summaries at a certain time unit level (year, day, hour, etc.). Tumbling windows are also used to transform (irregular) fact data into time series data, by aggregating it at a regular time interval.

One way of implementing tumbling windows is to use the `date_trunc` function, which will truncate the timestamp to the specified precision. For example, in the following, we retrieve the number of services for each hour and each day in 2024:

```
SELECT
    date_trunc('hour', station_service_time) AS window_start,
    window_start + INTERVAL 1 HOUR AS window_end,
    count(*) AS number_of_services
FROM ams_traffic_v
```

```
WHERE year(station_service_time) = 2024
GROUP BY ALL
ORDER BY 1;
```

window_start timestamp	window_end timestamp	number_of_services int64
2024-01-01 01:00:00	2024-01-01 02:00:00	2
2024-01-01 02:00:00	2024-01-01 03:00:00	3
2024-01-01 03:00:00	2024-01-01 04:00:00	4
.	.	.
.	.	.
.	.	.
2024-12-31 20:00:00	2024-12-31 21:00:00	9
2024-12-31 21:00:00	2024-12-31 22:00:00	1
2024-12-31 23:00:00	2025-01-01 00:00:00	2
8781 rows (6 shown)		3 columns

Another approach is to use the `time_bucket` function, which will truncate the timestamp to the bucket width provided, starting from the specified offset. For example, we calculate the number of services each quarter of an hour, starting with `00`:

```
SELECT
    time_bucket(
        INTERVAL 15 MINUTE, -- bucket width
        station_service_time,
        INTERVAL 0 MINUTE -- offset
    ) AS window_start,
    window_start + INTERVAL 15 MINUTE as window_end,
    count(*) AS number_of_services
FROM ams_traffic_v
WHERE year(station_service_time) = 2024
GROUP BY ALL
ORDER BY 1;
```

window_start timestamp	window_end timestamp	number_of_services int64
2024-01-01 01:30:00	2024-01-01 01:45:00	1
2024-01-01 01:45:00	2024-01-01 02:00:00	1
2024-01-01 02:15:00	2024-01-01 02:30:00	2
.	.	.
.	.	.
.	.	.
2024-12-31 20:45:00	2024-12-31 21:00:00	2
2024-12-31 21:00:00	2024-12-31 21:15:00	1
2024-12-31 23:45:00	2025-01-01 00:00:00	2
32932 rows (6 shown)		3 columns

> The time bucket function is generating the buckets from the timestamp column itself, there could be gaps in the time series data. As seen in the above result, the first record is `2024-01-01 01:30:00`, because there are no records before that timestamp.

Given that tumbling windows are non-overlapping intervals, we can calculate summaries, such as the average number of services during a 15 minute interval. It is interesting to observe that the number of train services is quite stable during the day but it's much lower during the night – even in Amsterdam.

Hopping Windows

Hopping windows are fixed-size time intervals, but, contrary to tumbling windows, are overlapping. A hopping window is defined by:

- how much time should elapse between the window start time, called **hopping size**;
- how much time should a window contain, called **window size**.

One use case for hopping windows is to identify the five busiest 15-minute periods (window size) during 2024, starting every 5 minutes (hopping size). We start by generating artificial hopping windows for all the dates we are interested in:

```
WITH time_range AS (
    SELECT
        range AS window_start,
        window_start + INTERVAL 15 MINUTE AS window_end
    FROM range(
        '2024-01-01 00:00:00'::TIMESTAMP,
        '2025-01-01 00:00:00'::TIMESTAMP,
        INTERVAL 5 MINUTE -- hopping size
    )
)
```

window_start timestamp	window_end timestamp
2024-01-01 00:00:00	2024-01-01 00:15:00
2024-01-01 00:05:00	2024-01-01 00:20:00
2024-01-01 00:10:00	2024-01-01 00:25:00
.	.
.	.
.	.
2024-12-31 23:45:00	2025-01-01 00:00:00
2024-12-31 23:50:00	2025-01-01 00:05:00
2024-12-31 23:55:00	2025-01-01 00:10:00
105408 rows (6 shown)	2 columns

We then join the above intervals with the train service data in order to calculate the number of services for each **[left-closed, right-open)** interval:

```
SELECT
    window_start,
    window_end,
    count(service_sk) AS number_of_services
FROM ams_traffic_v
INNER JOIN time_range AS ts
    ON station_service_time >= ts.window_start
        AND station_service_time < ts.window_end
GROUP BY ALL
ORDER BY 3 DESC, 1 ASC
LIMIT 5;
```

resulting in:

window_start timestamp	window_end timestamp	number_of_services int64
2024-02-17 10:25:00	2024-02-17 10:40:00	28
2024-02-17 11:25:00	2024-02-17 11:40:00	28

2024-02-17 16:25:00	2024-02-17 16:40:00	28
2024-02-17 09:25:00	2024-02-17 09:40:00	27
2024-02-17 12:25:00	2024-02-17 12:40:00	27

Can you imagine how it must have been like in the control room when within 15 minutes, 28 trains were arriving or departing in a station with 15 tracks?

> By applying a `RIGHT OUTER JOIN` in the above query, gaps are filled with 0 number of services.

Sliding Windows

Sliding windows are overlapping intervals, but, compared to hopping windows, they are dynamically generated from the time column analyzed, therefore changing when new records are inserted. Sliding windows can be implemented by using the RANGE window framing:

```
SELECT
    station_service_time - INTERVAL 15 MINUTE AS window_start, -- window size
    station_service_time                      AS window_end,
    count(service_sk) OVER (
        ORDER BY station_service_time
            RANGE
                BETWEEN INTERVAL 15 MINUTE PRECEDING -- window size
                AND CURRENT ROW
    ) AS number_of_services
FROM ams_traffic_v
ORDER BY 3 DESC, 1
LIMIT 5;
```

window_start timestamp	window_end timestamp	number_of_services int64
2024-02-17 11:25:00	2024-02-17 11:40:00	29
2024-02-17 10:24:00	2024-02-17 10:39:00	28
2024-02-17 11:18:00	2024-02-17 11:33:00	28
2024-02-17 11:18:00	2024-02-17 11:33:00	28
2024-02-17 11:23:00	2024-02-17 11:38:00	28

> Because the current row is included in the calculation, the sliding windows are **[left-closed, right-closed]**.

Session Windows

A *session window* groups events that happen close together in time, separated by inactivity gaps. A new session starts when the time between two events exceeds a defined timeout. The most common use case of session windows is to detect gaps in the timestamped data.

We continue the data analysis by identifying the days in which there were periods of time larger than 10 minutes in which no train was arriving/departing in/from the Amsterdam Centraal Station. In this context, a session window is the period of time in which train services run without a service inactivity gap longer than 10 minutes.

We start by calculating, for each record, the previous service time, by using the `lag` window function. We observed above that there is almost no traffic during the night, therefore we include only services between 6 AM and 11 PM:

```
SELECT
    service_sk,
    station_service_time,
```

```
    lag(station_service_time) OVER (
        PARTITION BY station_service_time::DATE
        ORDER BY station_service_time
    ) AS previous_service_time,
    date_diff('minute', previous_service_time, station_service_time) AS gap_minutes
FROM ams_traffic_v
WHERE hour(station_service_time) BETWEEN 6 AND 23
```

In the above query we also calculate the gap, in minutes, between the current service and the previous service, with `date_diff`. If there is no previous service, the column will be `NULL`, depicting the first service session in the day:

station_service_time timestamp	previous_service_time timestamp	gap_minutes int64
2024-01-09 06:00:00	NULL	NULL
2024-01-16 06:00:00	NULL	NULL
2024-01-22 06:00:00	NULL	NULL
·	·	·
·	·	·
·	·	·
2024-11-28 06:01:00	NULL	NULL
2024-12-05 06:01:00	NULL	NULL
2024-12-23 06:00:00	NULL	NULL
366 rows (6 shown)		3 columns

> **Tip.** Because `gap_minutes` is computed based on a window function, we can filter on it with `QUALIFY`, e.g.: `QUALIFY gap_minutes IS NULL`

We then mark if the current record is in the same session as the previous one, by comparing the minutes elapsed to a timeout, in our case 10 minutes:

```
IF(gap_minutes >= 10 OR gap_minutes IS NULL, 1, 0) AS new_session
```

By applying a moving sum, at day level, over the `new_session` attribute, we assign an identifier to the session:

```
sum(new_session) OVER (
    PARTITION BY station_service_date
    ORDER BY station_service_time ROWS UNBOUNDED PRECEDING
) AS session_id_in_day
```

Bringing it all together, we can now retrieve the dates which had at least one inactivity gap of 10 minutes during the 18 hours day service time (the number of hours between 6 AM and 11 PM):

```
WITH ams_daily_traffic AS (
    SELECT
        service_sk,
        station_service_time,
        lag(station_service_time) OVER (
            PARTITION BY station_service_time::DATE
            ORDER BY station_service_time
        ) AS previous_service_time,
        date_diff('minute', previous_service_time, station_service_time) AS gap_minutes
    FROM ams_traffic_v
    WHERE hour(station_service_time) BETWEEN 6 AND 23
), window_calculation AS (
        SELECT
            service_sk,
            station_service_time,
```

```
            station_service_time::DATE AS station_service_date,
            gap_minutes,
            IF(gap_minutes >= 10 OR gap_minutes IS NULL, 1, 0) new_session,
            sum(new_session) OVER (
                PARTITION BY station_service_date
                ORDER BY station_service_time ROWS UNBOUNDED PRECEDING
            ) AS session_id_in_day
        FROM ams_daily_traffic
), session_window AS (
    SELECT
        station_service_date,
        session_id_in_day,
        max(gap_minutes)          AS gap_minutes,
        min(station_service_time) AS window_start,
        max(station_service_time) AS window_end,
        count(service_sk)         AS number_of_services
    FROM window_calculation
    GROUP BY ALL
)
SELECT
    station_service_date,
    max(ceil(date_diff('minute', window_start, window_end) / 60)) AS number_of_hours_without_gap,
    count(*) AS number_of_sessions,
    sum(number_of_services) as number_of_services,
FROM session_window
GROUP BY ALL
HAVING number_of_hours_without_gap < 18
ORDER BY 2, 1;
```

station_service_date date	number_of_hours_without_gap double	number_of_sessions int64	number_of_services int128
2024-04-29	7.0	12	521
2024-12-31	14.0	6	946
2024-01-01	16.0	6	847
2024-04-30	16.0	7	645
2024-04-14	17.0	3	1289
2024-05-01	17.0	5	788
2024-05-02	17.0	3	729
2024-05-03	17.0	5	699
2024-05-04	17.0	3	907
2024-05-19	17.0	3	837
2024-10-28	17.0	2	748
2024-10-29	17.0	2	785
2024-10-30	17.0	2	783
2024-11-02	17.0	2	654
14 rows			4 columns

Something must have happened on 29 April 2024! We observe that, during the 18 hours of service, there were 12 session windows, which means that, for at least 10 times, no train arrived or departed during a period of 10 minutes. A reason for this could be that a regular train service was not running on that day. And indeed maintenance work started between Amsterdam and Utrecht.

> **Tip.** Time windows are visualized with Plotly timeline charts, a type of Gantt charts.

Conclusion

In this post we have demonstrated how stream windowing functions can be implemented on historical timestamped data in DuckDB, offering a starting point in time (series) data analysis. We also recommend "Catching up with Windowing", a post about DuckDB's windowing features, which can be adopted in the functions presented herein.

Sorting on Insert for Fast Selective Queries

Publication date: 2025-05-14

Author: Alex Monahan

TL;DR: Sorting data when loading can speed up selective read queries by an order of magnitude, thanks to DuckDB's automatic min-max indexes (also known as zone maps). This approach applies to most columnar file formats and databases as well. This post unpacks the DuckDB file structure as an example of a columnar data format and gives practical advice for using sorting to improve the speed of queries.

The fastest way to read data is to not read data. It's as simple as that! This post is all about how to read as little data as possible to answer selective read queries.

> Feel free to skip straight to the best practices!

Use Cases

The cases where these techniques are most useful are when read performance is more critical than write performance. In these cases, we can deliberately add ordering steps that will slow down our writes, but will greatly speed up our reads. This is often the case when data is pre-processed in the background, but read queries are customer facing in a dashboard or app.

Using sorting will be helpful in any of these situations:

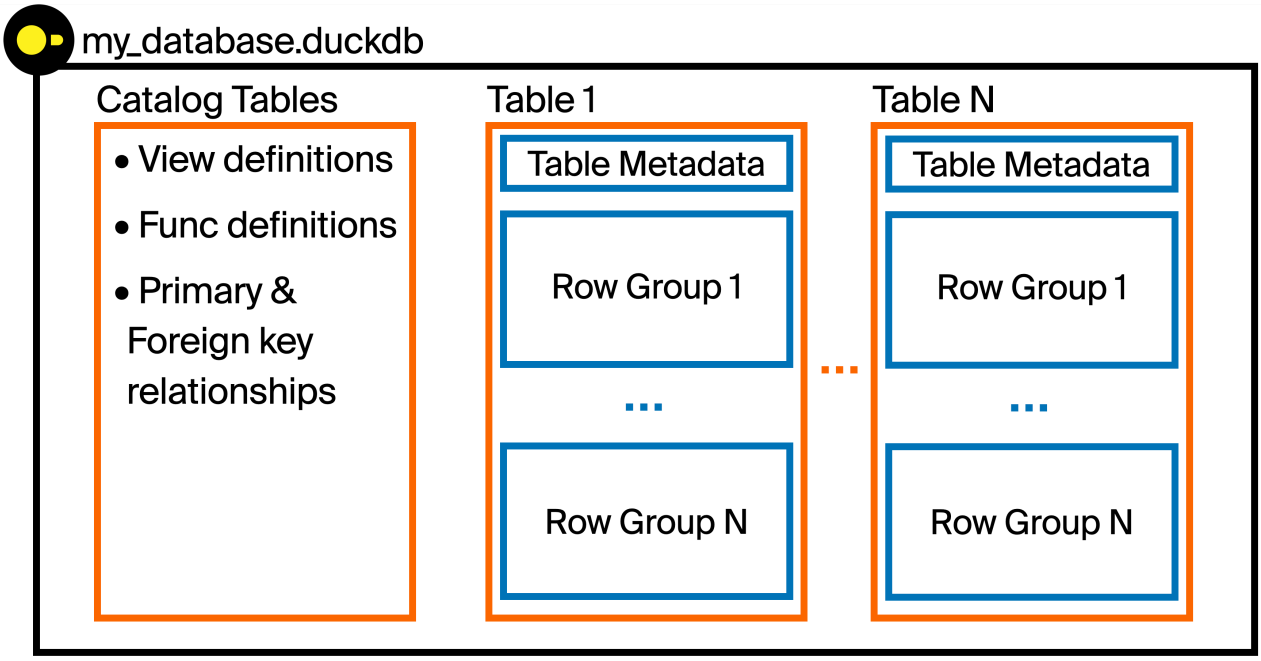
- Your dataset is large and doesn't fit entirely in memory
- You only want to read a portion of your dataset for each query
- You access your data via HTTP(S)
- Your data lives in the cloud on object storage such as AWS S3

This overview describes the DuckDB file format, but thanks to DuckDB's partial reading support, these techniques can be generally applied to nearly any columnar file format or database. This is a great way to speed up querying Apache Parquet files on remote endpoints – including data lakes!

> Stay tuned for a subsequent post covering advanced multi-column sorting!

How Does the DuckDB Format Help?

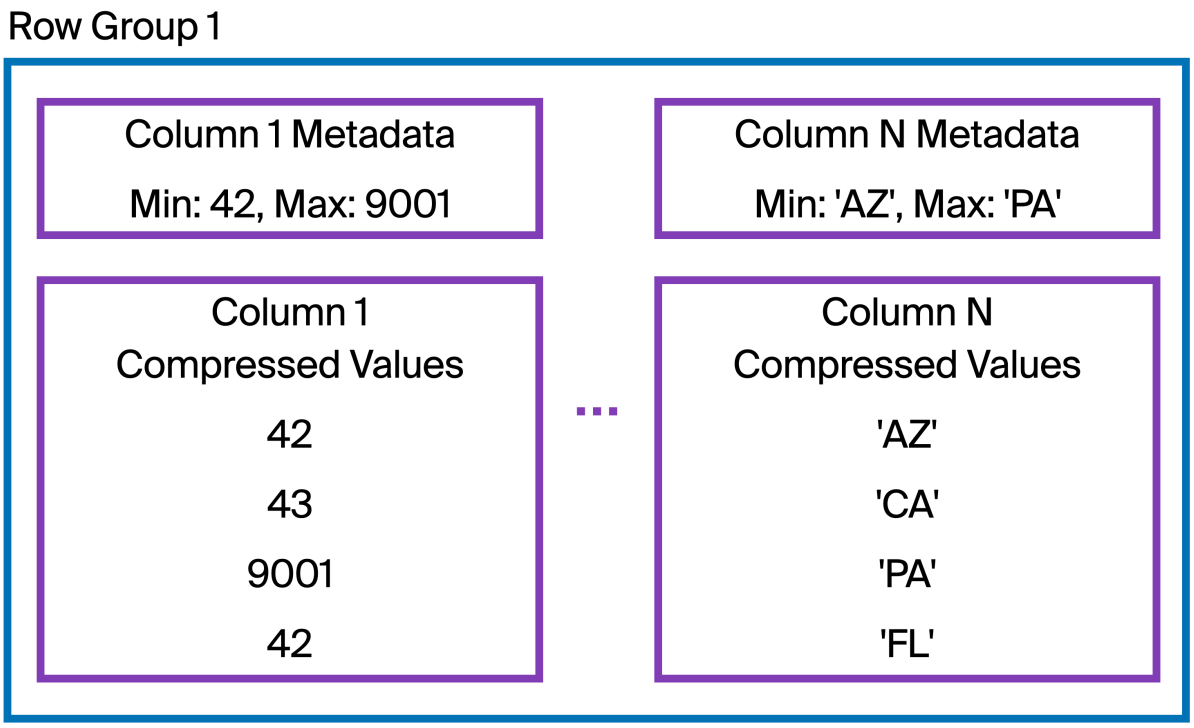
Let's build up some intuition around the structure of the DuckDB file format! DuckDB stores data in a single file. Each file is called a *database* (the DuckDB library is the *database engine*).



Each database file can store multiple tables, views, functions, indexes, and primary/foreign key relationships all in the same file. This has some key advantages for portability, but it also allows for the storage of more than just data – metadata can be stored as well. This metadata allows for the DuckDB engine to selectively read just portions of the DuckDB file as they are requested, which is key for handling larger than memory datasets and for the performance optimizations in this post!

DuckDB stores data in a columnar fashion (meaning that values within a column are stored together in the same set of blocks). However, columnar does not mean storing the entire column contiguously! Before storing data, DuckDB breaks tables up into chunks of rows called row groups. Each row group is 122 880 rows by default.

Next, we zoom in on `Row Group 1` from `Table 1` in the first diagram:



Within each row group, the data related to a single column is stored contiguously together on disk in one or more blocks. DuckDB com-

presses this data to reduce file size.

Using Zone Maps to Skip Reading Data

So, how does storing data in columns help with selective read queries? By itself, it does not! However, at the start of the column data stored within each row group, DuckDB also stores metadata about the column data being stored. This includes the minimum and maximum values of that column within that row group. We call these *zone maps* or *min-max indexes.*

When DuckDB receives a SQL query that contains a filter, before reading the column segments off of disk, it checks the metadata first. Could this filter value possibly fall within the minimum/maximum range of this column segment? If it is not possible, then DuckDB can skip reading the data in that entire row group.

For example, `Column N` in this example is storing the abbreviations for US states. If we are querying data for the state of VA (Virginia), could this row group contain any data for VA? Our query could be:

```
FROM "Table 1"
WHERE "Column N" = 'VA';
```

> DuckDB's friendly SQL makes the traditional SQL `SELECT *` optional!

DuckDB first checks the zone map (labeled `Column N Metadata` in the diagram). Does VA fall within the range (alphabetically) of AZ to PA? No it does not! We can skip this entire row group.

If instead we were filtering for the state of NM (New Mexico), the zone map indicates that it is possible that NM data exists within this row group. As a result, DuckDB will retrieve the entire column of data (possibly from a remote location) and check each row to see if any NM data exists. According to the diagram, there is no NM data in this row group, so retrieving it and checking each row for a match would be unnecessary effort. All 122 880 rows would need to be checked! If only the zone map index did not cover such a large range of state abbreviations...

Strategically Skipping Data

Our goal is to sort our data so that the zone maps are as selective as possible for the columns we are interested in filtering. If the min-max index of this row group only extended from AZ to CA, then the data would be far more likely to be skipped. Said another way, each subset of the data we want to retrieve should only be stored in a few row groups. We don't want to pull every row group and check every row! Hopefully states that are early in the alphabet are all grouped together in the same row group, for example.

When executing selective queries, you could aim for pulling just a single row group! However, since DuckDB's multithreading model is based on row groups, you should still see high performance as long as the number of row groups is less than the number of threads (≈CPU cores) DuckDB is using.

Sorting Best Practices

There are many rules of thumb for how to best utilize min-max indexes, since their effectiveness is highly data and workload dependent. These are useful both in DuckDB as well as in other columnar formats like Apache Parquet or other columnar databases. Here are several approaches to consider!

When deciding which columns to sort, it is critical to examine the `WHERE` clauses of any read workloads. Often the most important factor is which columns are used to filter most often. A basic approach would be to sort by all columns that are used as filters, beginning with the columns used most often.

Another option if the workload involves filtering by multiple different columns is to consider sorting first by columns with the lowest cardinality (the fewest unique values). For example, it may be helpful to sort by a broad `customer_type` prior to a unique `customer_id`.

It is often useful to only read recent data, so ensuring recent data is not spread throughout the table can improve performance. However, when ordering by timestamps, be mindful that timestamps are frequently very high cardinality. As a result, sorting by lower cardinality

columns first may be helpful. You may be tempted to sort by timestamp first – **resist that temptation!** (or at least test out alternatives). It may be more beneficial to sort by a timestamp rounded to the nearest week, month, or year and then by other columns.

> To benefit from min-max indexes, a `WHERE` clause must filter directly on a specific column, not on a calculated expression. If an expression is used, it must be evaluated for each row, so no row groups can be skipped!

Additional Techniques

There are more ways to take full advantage of the min-max indexes in DuckDB!

Avoid Small Inserts

If a workload is inserting data in small batches or a single row at a time, there is not an opportunity to effectively sort the data when it is being inserted. Instead, the data will be sorted largely by insertion time, which will only provide effective pruning for time-based filters. If possible, bulk inserts or batching will allow the sorting to work more effectively for other columns. As an alternative, there can be a periodic re-sorting job, which is analogous to a re-indexing task in transactional systems.

Sort in Chunks

Sorting can be a computationally expensive operation for large tables. One way to reduce the amount of memory (or disk spill) required when sorting is to process the table in pieces by looping through multiple SQL statements, each filtered to a specific chunk. Since SQL does not have a looping construct, this would be handled by a host language (Python, Jinja templating, etc.). The pseudocode would be to:

```
CREATE OR REPLACE TABLE sorted_table AS
    FROM unsorted_table
    WITH NO DATA;

for chunk in chunks:
    INSERT INTO sorted_table
        FROM unsorted_table
        WHERE chunking_column = chunk
        ORDER BY other_columns...;
```

This will have the effect of sorting initially by the chunking column, and then by the `other_columns`. It may also take longer to run (since the data must be scanned once per chunk), but memory use is likely to be much lower.

Sort the First Few Characters of Strings

Approximate sorting works well for improving read performance. In the zone maps of VARCHAR columns, DuckDB stores just the first 8 bytes of the min and max string values. As a result, there is no need to sort more than the first 8 bytes (8 ASCII characters)!

This has an added benefit of faster sorting, as the runtime of DuckDB's radix sort algorithm is sensitive to the length of strings (by design!). The time complexity of the algorithm is $O(nk)$, where n is the number of rows, and k is the width of the sorting key. Sorting by just the first few characters of a VARCHAR can be quicker and less compute intensive while achieving similar read performance. DuckDB's VARCHAR data type also inlines strings when they are under 12 bytes, so working with short strings is faster for that reason as well. For example:

```
CREATE OR REPLACE TABLE sorted_table AS
    FROM unsorted_table
    ORDER BY varchar_column_to_sort[:8];
```

> DuckDB's friendly SQL allows bracket notation for string slicing!

Filter by More Columns

Adding filters to a `WHERE` clause can be helpful if those columns being filtered have any kind of approximate order. For example, instead of just filtering by `customer_id`, if the table is sorted by `customer_type`, include that in the query also. Often, if the `customer_id` is known at query time, it is possible to know other metadata as well.

Adjust the Row Group Size

One parameter that can be tuned for specific workloads is the number of rows within a row group (the `ROW_GROUP_SIZE`). If there are many unique values within a column being filtered on, then a smaller number of rows per row group could reduce the total number of rows that must be scanned. However, there is an overhead of checking metadata more often when row groups are smaller, so there is a tradeoff.

A larger row group size may actually be preferable if a table is particularly large and queries are very selective. For example, if querying a large fact table with years of history, but filtering to only the last week of data. Larger row group sizes reduce the number of metadata checks that are necessary to reach the recent data. However, each row group is larger, so there is a tradeoff there as well.

To adjust the row group size, pass in a parameter when attaching a database. Note that a row group size should be a power of 2. The minimum row group size is 2 048, the vector size of DuckDB.

```
ATTACH './smaller_row_groups.duckdb' (ROW_GROUP_SIZE 8192);
```

Sneak Peak of the Next Post in the Series

A subsequent post in this series will cover advanced sorting techniques. These can boost performance if multiple reader workloads with different filters are querying the same dataset. For example, what if I need to filter by US state for some queries, but by customer last name for others? Additionally, sorting by rounded time buckets (day, month, or year) and then by other columns can be helpful when only recent data is needed.

Microbenchmarks will show the 10x performance benefits that were mentioned at the start of this post! The post will also provide ways to measure how effectively a table is sorted along various columns.

Conclusion

Ordering data upon insert can significantly speed up read queries that include filters. Once your dataset becomes large or you are storing it remotely, consider applying these techniques. Plus, these approaches can be used in nearly any columnar file format or database!

Happy analyzing!

Machine Learning Prototyping with DuckDB and scikit-learn

Publication date: 2025-05-16

Author: Petrica Leuca

TL;DR: In this post, we prototype a machine learning workflow using DuckDB for data handling and scikit-learn for modeling.

Introduction

Machine learning prototyping often involves juggling datasets, preprocessing steps, and performance constraints, which can make the process both complex and time-consuming. scikit-learn is one of Python's most popular and powerful libraries for machine learning, offering a large set of utilities for building and evaluating models. In this post, we will explore how DuckDB complements `scikit-learn` in the model development life cycle, by implementing a penguin species prediction model on Penguins observations in the Palmer Archipelago.

> The following implementation is executed in a marimo Python notebook, which is available on GitHub, in our examples repository.

Data Preparation

We start by loading the Palmer Penguins dataset and use DuckDB's `COLUMNS(*)` expression to filter out any record which contains `NA` or `NULL`:

```
duckdb_conn.read_csv(
    "http://blobs.duckdb.org/data/penguins.csv"
).filter(
    "columns(*)::text != 'NA'"
).filter(
    "columns(*) is not null"
).select(
    "*, row_number() over () as observation_id"
).to_table(
    "penguins_data"
)
```

Even if the `NA` values are removed from the dataset, DuckDB has inferred the column type as `VARCHAR` at the moment of `read_csv`. Therefore, we modify the column type to a numeric type, `DECIMAL(5, 2)`, using the `ALTER TABLE` statement:

```
duckdb_conn.sql(
    "alter table penguins_data alter bill_length_mm set data type decimal(5, 2)"
)
```

> **Tip.** One can also define the schema at import time.

We can now plot the data and check for species specific clusters. For example, using a scatter plot we identify clusters at the combination between bill depth and bill length:

We also observe that there are a few descriptive columns in the dataset, such as `species` and `island`. In a machine learning workflow, one common preprocessing step is to transform such values into numerical values, assigning them a unique identifier; a process called *label encoding*. While scikit-learn offers a LabelEncoder utility, this process is very similar to how we work with reference tables in a data warehouse. Therefore, we define a function which will create, for each column (feature), a reference table, leveraging the DuckDB Python relational API:

```
def process_reference_data(duckdb_conn):
    for feature in ["species", "island"]:
        duckdb_conn.sql(f"drop table if exists {feature}_ref")
        (
            duckdb_conn.table("penguins_data")
            .select(feature)
            .unique(feature)
            .row_number(
                window_spec=f"over (order by {feature})",
                projected_columns=feature
            )
            .select(f"{feature}, #2 - 1 as {feature}_id")
            .to_table(f"{feature}_ref")
        )
        duckdb_conn.table(f"{feature}_ref").show()
```

After the execution of the above function, two tables which will contain the distinct values of the category and a unique identifier are created, e.g., `species_ref`:

species varchar	species_id int64
Adelie	0
Chinstrap	1
Gentoo	2

The last step in our data preparation step is to define a selection query variable, which selects the data from the initial dataset and the reference tables:

```
selection_query = (
    conn.table("penguins_data")
    .join(conn.table("island_ref"), condition="island")
    .join(conn.table("species_ref"), condition="species")
)
```

> `selection_query` is lazy evaluated, therefore, no data is retrieved at the definition time.

Model Training

Our scope is to predict a penguin's specie based on its characteristics (features), such as bill length and depth, flipper length, body mass and island. Such a model is called a classification model because it predicts the category to which the input data belongs.`scikit-learn` offers multiple classification models. For our dataset, we have chosen to implement a random forest classifier, based on decision trees:

We start by splitting the data into train and test data, with the `train_test_split` utility from `scikit-learn`:

```
def train_split_data(selection_query):
    X_df = selection_query.select("""
        bill_length_mm,
        bill_depth_mm,
        flipper_length_mm,
        body_mass_g,
        island_id,
        observation_id,
        species_id
    """).order("observation_id").df()
    y_df = [
        x[0]
```

```
        for x in selection_query.order("observation_id").select("species_id").fetchall()
    ]

    num_test = 0.30
    return train_test_split(X_df, y_df, test_size=num_test)

X_train, X_test, y_train, y_test = train_split_data(selection_query)
```

Splitting the data is a common step in a machine learning workflow and it returns:

- `X_train`, containing the input data based on which we want to assign a species category;
- `y_train`, containing the species for each record in `X_train`;
- `X_test`, containing the input data based on which we will test the model;
- `y_test`, containing the species for each record in `X_test`.

We then define the RandomForestClassifier and fit the `X_train` and `y_train` data to it. We save the model in a pickle file, such that we can use the model without retraining:

```
model = RandomForestClassifier(n_estimators=1, max_depth=2, random_state=5)

model.fit(X_train.drop(["observation_id", "species_id"], axis=1).values, y_train)

pickle.dump(model, open("./model/penguin_model.sav", "wb"))
```

We can now check the model accuracy score:

```
model.score(
    X_test.drop(["observation_id", "species_id"], axis=1).values, y_test
)
```

```
0.98
```

> We are working with `pickle` due to the small size of our model. Other persistence methods are detailed on the `scikit-learn` documentation page.

Inference with DuckDB

Inference is the process of using the model to get predictions on (new) data:

```
model = pickle.load(open("./model/penguin_model.sav", "rb"))

model.predict(...)
```

When working with DuckDB there are three approaches to retrieve the predicted category from the data. We can use Pandas, or a DuckDB Python UDF with or without batching. In the following, we present these approaches.

With Pandas

The most common method of retrieving predictions is to load the predictions into a new column in a Pandas dataframe:

```
predicted_df = selection_query.select("""
    bill_length_mm,
    bill_depth_mm,
    flipper_length_mm,
    body_mass_g,
    island_id,
    observation_id,
    species_id
```

```
""").df()

predicted_df["predicted_species_id"] = model.predict(
    predicted_df.drop(["observation_id", "species_id"], axis=1).values
)
```

With DuckDB, the dataframe can be queried with SQL:

```
(
    duckdb_conn.table("predicted_df")
    .select("observation_id", "species_id", "predicted_species_id")
    .filter("species_id != predicted_species_id")
)
```

resulting in:

observation_id int64	species_id int64	predicted_species_id int64
13	1	2
15	1	2
39	1	2
44	1	2
68	1	2
70	1	2
76	1	2
90	1	2
94	1	2
104	1	2
106	1	2
110	1	2
124	1	2
126	1	2
243	3	2
296	2	1
16 rows		3 columns

> **Warning.** If there is a table which has the same name as the dataframe, then `register` can be used to give another table name to the dataframe, e.g., `duckdb_conn.register("table_name", predicted_df)`

DuckDB Python UDF, Row by Row

DuckDB offers the possibility to register User Defined Functions (UDFs) from Python functions. Because the UDF is executed row by row, our prediction function will return the predicted species id for each row:

```
def get_prediction_per_row(
    bill_length_mm: Decimal,
    bill_depth_mm: Decimal,
    flipper_length_mm: int,
    body_mass_g: int,
    island_id: int
) -> int:
    model = pickle.load(open("./model/penguin_model.sav", "rb"))
    return int(
        model.predict(
            [
                [
```

```
                    bill_length_mm,
                    bill_depth_mm,
                    flipper_length_mm,
                    body_mass_g,
                    island_id,
                ]
            ]
        )[0]
    )
```

In the above Python function we provide the required features as input for the model and return the predicted value (an `integer`). With this information, we create the DuckDB function:

```
duckdb_conn.create_function(
    "predict_species_per_row", get_prediction_per_row, return_type=int
)
```

We can now use the UDF in the SQL `SELECT` clause:

```
selection_query.select("""
    observation_id,
    species_id,
    predict_species_per_row(
        bill_length_mm,
        bill_depth_mm,
        flipper_length_mm,
        body_mass_g,
        island_id
    ) as predicted_species_id
""").filter("species_id != predicted_species_id")
```

This returns the same results as above.

DuckDB Python UDF, Batch Style

While the row-by-row prediction can be useful when the scope is to retrieve the prediction for a limited number of rows, it will be less performant when working with more data. Therefore, by aggregating the data into arrays (which preserve order), we can simulate a mass retrieval (or batch style) of the prediction.

We first create a Python function to get the predictions for a JSON object, which contains the columnar representation of the features needed as model input:

```
def get_prediction_per_batch(input_data: dict[str, list[Decimal | int ]]) -> np.ndarray:
    """
    input_data example:
        {
            "bill_length_mm": [40.5],
            "bill_depth_mm": [41.5],
            "flipper_length_mm: [250],
            "body_mass_g": [3000],
            "island_id": [1]
        }
    """
    model = pickle.load(open("./model/penguin_model.sav", "rb"))

    input_data_parsed = orjson.loads(input_data)

    input_data_converted_to_numpy = np.stack(tuple(input_data_parsed.values()), axis=1)

    return model.predict(input_data_converted_to_numpy)
```

```
duckdb_conn.create_function(
    "predict_species_per_batch",
    get_prediction_per_batch,
    return_type=duckdb.typing.DuckDBPyType(list[int]),
)
```

> While DuckDB has MAP and STRUCT data types, which convert automatically to a dict, they have a slower execution time compared to `json_object` (including the `orjson` deserialization time).

With `json_object` we extract the columnar representation of the features, in the format `'feature_name': array[feature]`:

```
json_object(
    'bill_length_mm', array_agg(bill_length_mm),
    'bill_depth_mm', array_agg(bill_depth_mm),
    'flipper_length_mm', array_agg(flipper_length_mm),
    'body_mass_g', array_agg(body_mass_g),
    'island_id', array_agg(island_id)
) as input_data,
```

We then pack in a STRUCT, the predictions together with other columns we are interested in:

```
struct_pack(
    observation_id := array_agg(observation_id),
    species_id := array_agg(species_id),
    predicted_species_id := predict_species_per_batch(input_data)
) as output_data
```

Last, we unnest the results, in order to flatten the lists into tables:

```
.select("""
    unnest(output_data.observation_id) as observation_id,
    unnest(output_data.species_id) as species_id,
    unnest(output_data.predicted_species_id) as predicted_species_id
""")
```

The above query is wrapped into a Python function, named `get_selection_query_for_batch`, with which we can chain queries, e.g., for mass retrieval of incorrect predictions:

```
get_selection_query_for_batch(selection_query).filter("species_id != predicted_species_id")
```

A batch approach can be implemented by using LIMIT and OFFSET to loop through the data:

```
for i in range(4):
    (
        get_selection_query_for_batch(
            selection_query
            .order("observation_id")
            .limit(100, offset=100*i)
            .select("*")
        )
        .filter("species_id != predicted_species_id")
    ).show()
```

> LIMIT and OFFSET are executed last, therefore, they should be applied before the prediction selection.

Performance Considerations

To obtain performance data on a larger scale dataset, we have generated a dummy dataset with approximately 59 million records. On a sample of 10%, on a 16 GB MacBook Pro, the batch processing ranges between 3 and 4 seconds, while the Pandas implementation is executed under 1 second. This is because the Python UDF includes multiple conversion steps which affect performance:

- parsing the input data as JSON;
- converting the JSON to numpy array;
- unnesting the array to rows.

We have still chosen to show-case the Python UDFs, because they are a powerful asset when working within a Python environment and the performance difference is negligible when we speak of small data.

Conclusion

In this post, we have demonstrated how DuckDB complements `scikit-learn` in the machine learning development life cycle, with a focus on the data preparation and inference phases. The inference results on the dummy data are poor; is this because of the model or the dummy data? We leave this question open to our readers, as a means to explore how DuckDB can be used in the model evaluation and model optimization phases.

The Lost Decade of Small Data?

Publication date: 2025-05-19

Author: Hannes Mühleisen

TL;DR: We benchmark DuckDB on a 2012 MacBook Pro to decide: did we lose a decade chasing distributed architectures for data analytics?

Much has been said, not in the very least by ourselves, about how data is actually not that "Big" and how the speed of hardware innovation is outpacing the growth of useful datasets. We may have gone so far to predict a data singularity in the near future, where 99% of useful datasets can be comfortably queried on a single node. As recently shown, the median scan in Amazon Redshift and Snowflake reads a doable 100 MB of data, and the 99.9-percentile reads less than 300 GB. So the singularity might be closer than we think.

But we started wondering, when did this development really start? When did personal computers like the ubiquitous MacBook Pro, usually condemned to running Chrome, become the data processing powerhouses that they really are today?

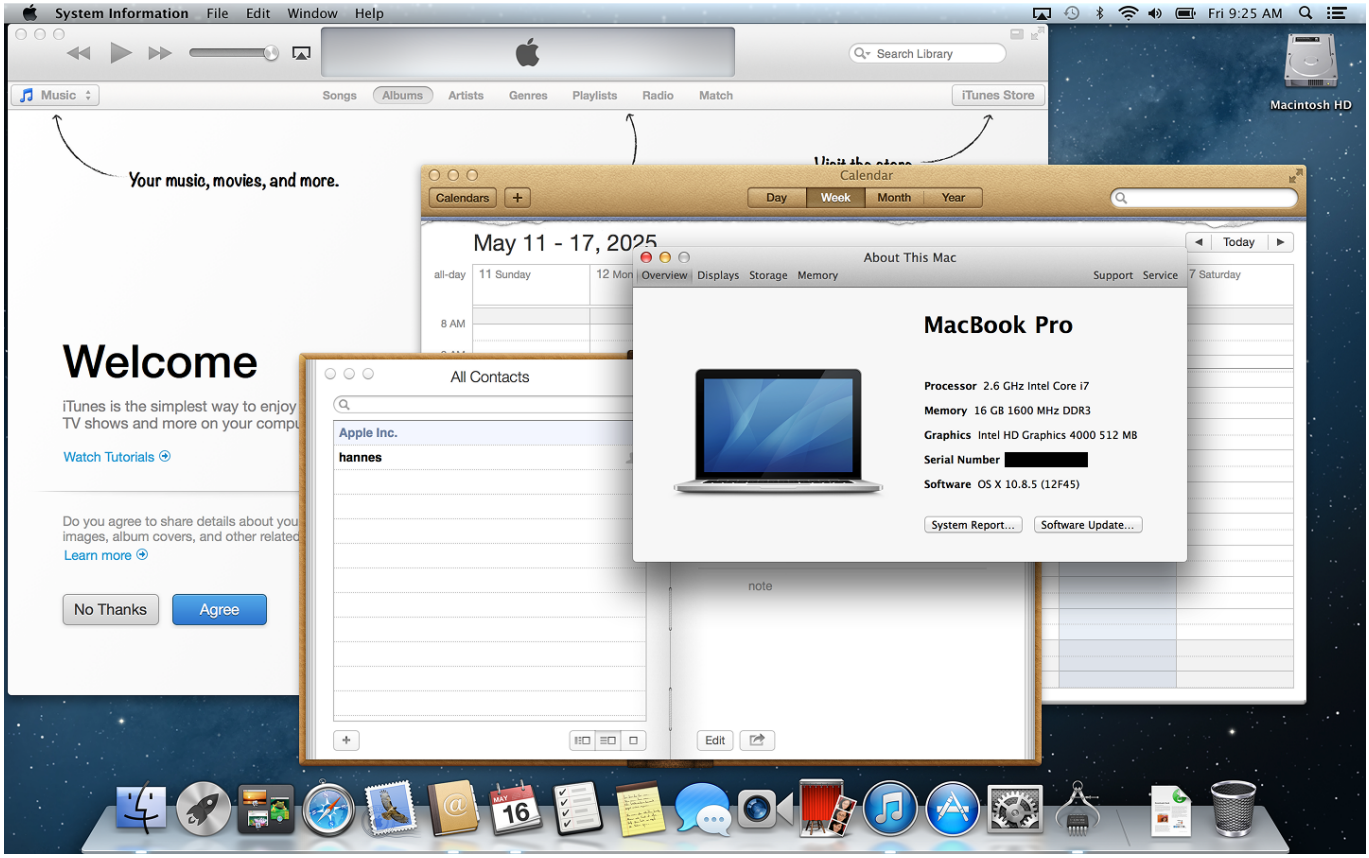
Let's turn our attention to the **2012** Retina MacBook Pro, a computer many people (myself included) bought at the time because of its gorgeous "Retina" display. Millions were sold. Despite being unemployed at the time, I had even splurged for the 16 GB RAM upgrade. But there was another often-forgotten revolutionary change in this machine: it was the first MacBook with a built-in Solid-State Disk (SSD) *and* a competitive 4-core 2.6 GHz "Core i7" CPU. It's funny to watch the announcement again, where they *do* stress the performance aspect of the "all-flash architecture" as well.

Side note: the MacBook Air was actually the first MacBook with an (optional) built-in SSD already back in 2008. But it did not have the CPU firepower of the Pro, sadly.

Coincidentally, I still have this laptop in the DuckDB Labs office, currently used by my kids to type their names in a massive font size or watch Bluey on YouTube when they're around. But can this relic still run modern-day DuckDB? How will its performance compare to modern MacBooks? And could we have had the data revolution that we are seeing now already back in 2012? Let's find out!

Software

First, what about the operating system? In order to make the comparison fair(er) to the decades, we actually *downgraded* the operating system on the Retina to OS X 10.8.5 "Mountain Lion", the operating system version that shipped just a few weeks after the laptop itself in July 2012. Even though the Retina can actually run 10.15 (Catalina), we felt a true 2012 comparison should also use an operating system from the era. Below is a screenshot of the user interface for those of us who sometimes feel a little old.



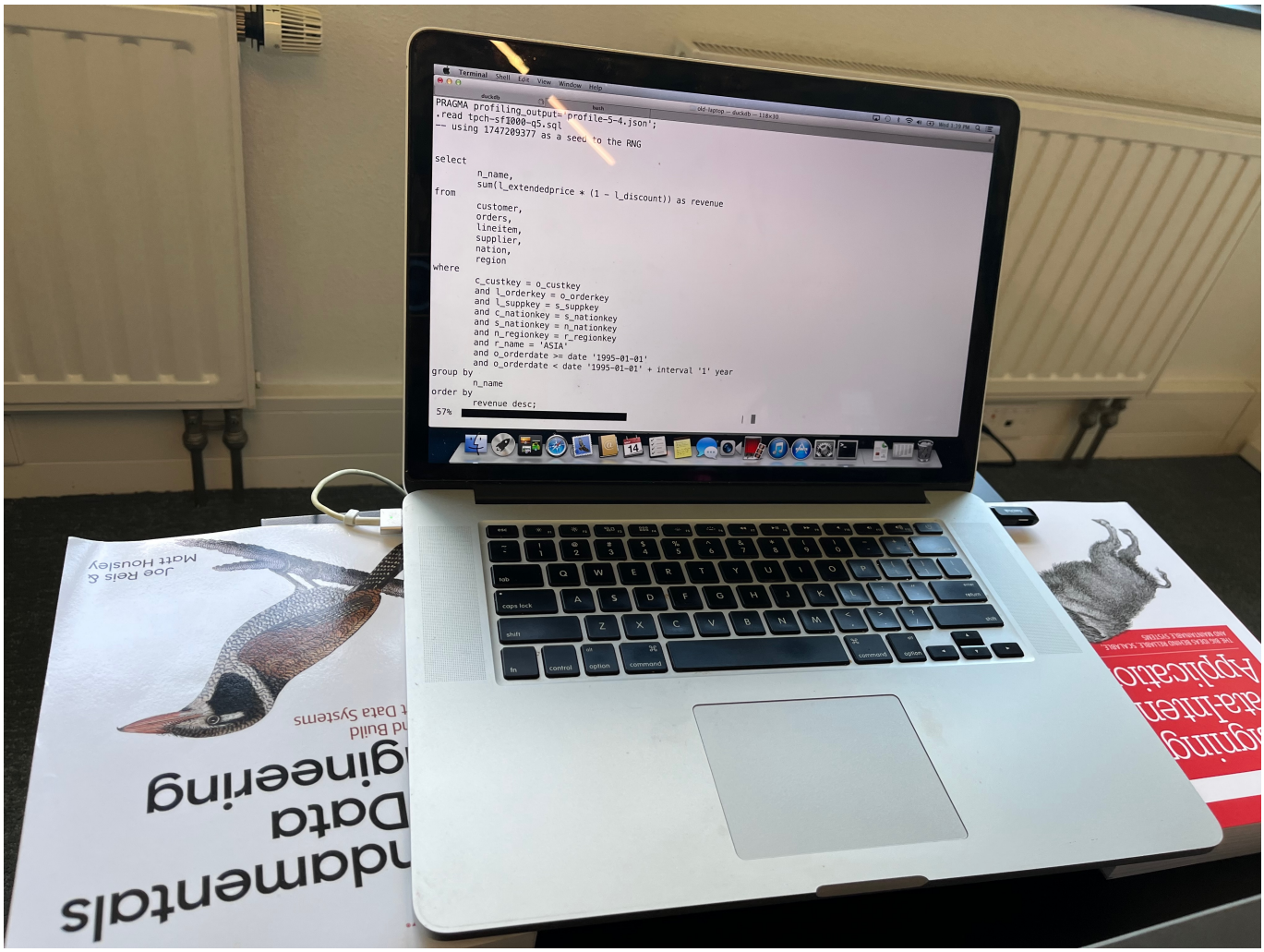
Moving on to DuckDB itself: here at DuckDB we are more than a little religious about portability and dependencies – or rather the lack thereof. This means that very little had to happen to make DuckDB run on the ancient Mountain Lion: the stock DuckDB binary is built with by default with backwards-compatibility to OS X 11.0 (Big Sur), but simply changing the flag and recompiling turned out to be enough to make DuckDB 1.2.2 run on Mountain Lion. We would have loved to also use a 2012 compiler to build DuckDB, but, alas, C++ 11 was unsurprisingly simply *too new* in 2012 to be fully supported by compilers. Either way, the binary runs fine and could have been also produced by working around the compiler bugs. Or we could have just hand-coded Assembly like others have done.

Benchmarks

But we're not interested in synthetic CPU scores, we're interested in synthetic SQL scores instead! To see how the old machine is holding up when performing *serious* data crunching, we used the *at this point rather tired but well-known TPC-H benchmark* at scale factor 1000. This means that the two main tables, `lineitem` and `orders` contain 6 and 1.5 Billion rows, respectively. When stored as a DuckDB database, the database has a size of ca. 265 GB.

From the audited results on the TPC website, we can see that running the benchmark on this scale factor on a single node seems to require hardware costing hundreds of thousands of Dollars.

We ran each of the 22 benchmark queries five times, and took the median runtime to remove noise. However, because the amount of RAM (16 GB) is very much less than the database size (256 GB), no significant amount of the input data can be cached in the buffer manager, so those are not really what people sometimes call “hot” runs.



Below are the per-query results in seconds:

query	latency
1	142.2
2	23.2
3	262.7
4	167.5
5	185.9
6	127.7
7	278.3
8	248.4
9	675.0
10	1266.1
11	33.4
12	161.7

query	latency
13	384.7
14	215.9
15	197.6
16	100.7
17	243.7
18	2076.1
19	283.9
20	200.1
21	1011.9
22	57.7

But what do those cold numbers actually mean? The hidden sensation is that we actually *have* numbers, this old computer could actually complete all benchmark queries using DuckDB! If we look at the time a bit closer, we see the queries take anywhere between a minute and half an hour. Those are not unreasonable waiting times for analytical queries on that sort of data in any way. Heck, you would have been waiting way longer back in 2012 for Hadoop YARN to pick up your job in the first place only to spew stack traces at you at some point.

2023 Improvements

But how do those results stack up against a modern MacBook? As a comparison point, we used a modern ARM-based M3 Max MacBook Pro, which happened to be sitting on the same desk. But between them, the two MacBooks represent more than a decade of hardware development.

Looking at GeekBench 5 benchmark scores alone, we see a ca. 7× difference in raw CPU speed when using all cores, and ca. factor 3 difference in single-core speed. Of course there are also big differences in RAM and SSD speeds. Funnily, the display size and resolution are almost unchanged.

Here are the results side-by-side:

query	latency_old	latency_new	speedup
1	142.2	19.6	7.26
2	23.2	2.0	11.60
3	262.7	21.8	12.05
4	167.5	11.1	15.09
5	185.9	15.5	11.99
6	127.7	6.6	19.35
7	278.3	14.9	18.68
8	248.4	14.5	17.13
9	675.0	33.3	20.27
10	1266.1	23.6	53.65
11	33.4	2.2	15.18
12	161.7	10.1	16.01
13	384.7	24.4	15.77

query	latency_old	latency_new	speedup
14	215.9	9.2	23.47
15	197.6	8.2	24.10
16	100.7	4.1	24.56
17	243.7	15.3	15.93
18	2076.1	47.6	43.62
19	283.9	23.1	12.29
20	200.1	10.9	18.36
21	1011.9	47.8	21.17
22	57.7	4.3	13.42

We do see significant speedups, from 7 up to as much as 53. The geometric mean of the timings improved from 218 to 12, a ca. 20× improvement.

Reproducibility

The binary, scripts, queries, and results are available on GitHub for inspection. We also made the TPC-H SF1000 database file available for download so you don't have to generate it. But be warned, it's a large file.

Discussion

We have seen how the decade-old MacBook Pro Retina has been able to complete a complex analytical benchmark. A newer laptop was able to significantly improve on those times. But absolute speedup numbers are a bit pointless here. The difference is purely *quantitative*, not *qualitative*.

From a user perspective, it matters much more that those queries complete in somewhat reasonable time, not if it took 10 or 100 seconds to do so. We can tackle almost the same kind of data problems with both laptops, we just have to be willing to wait a little longer. This is especially true given DuckDB's out-of-core capability, which allows it to spill query intermediates to disks if required.

What is perhaps more interesting is that back in 2012, it would have been completely feasible to have a single-node SQL engine like DuckDB that could run complex analytical SQL queries against a database of 6 billion rows in manageable time – and we did not even have to immerse it in dry ice this time.

History is full of "what if"s, what if something like DuckDB had existed in 2012? The main ingredients were there, vectorized query processing had already been invented in 2005. Would the now somewhat-silly-looking move to distributed systems for data analysis have ever happened? The dataset size of our benchmark database was awfully close to that 99.9% percentile of input data volume for analytical queries in 2024. And while the retina MacBook Pro was a high-end machine in 2012, by 2014 many other vendors shifted to offering laptops with built-in SSD storage and larger amounts of memory became more widespread.

So, yes, we really did lose a full decade.

Announcing DuckDB 1.3.0

Publication date: 2025-05-21

Author: The DuckDB team

TL;DR: The DuckDB team is happy to announce that today we're releasing DuckDB version 1.3.0, codenamed “Ossivalis”.

> To install the new version, please visit the installation guide. Note that it can take a few hours to days to release some client libraries (e.g., Go, R, Java) and extensions (e.g., the UI) due to the extra changes and review rounds required.

We are proud to release DuckDB 1.3.0. This release of DuckDB is named “Ossivalis” after Bucephala Ossivalis, an ancestor of the Goldeneye duck that lived millions of years ago.

In this blog post, we cover the most important features of the new release. DuckDB is moving rather quickly, and we could cover only a small fraction of the changes in this release. For the complete release notes, see the release page on GitHub.

Breaking Changes and Deprecations

Old Linux glibc Deprecation

Now that all mainstream Linux distributions use glibc 2.28 or newer, DuckDB's official Linux binaries require at least glibc 2.28 or newer. The release is built using the `manylinux_2_28` image from Python, which combines an older glibc with a newer compiler. This change also implies that extensions are no longer distributed for the `linux_amd64_gcc4` platform.

We highly value portability, so it is of course still possible to build DuckDB from source for older versions of glibc.

Lambda Function Syntax

Previously, lambda functions in DuckDB could be specified using the single arrow syntax: `x -> x + 1`. The single arrow operator is also used by the JSON extension to express JSON extraction using the syntax `->'field'`. The two meanings of the single arrow operator are treated the same by the binder, thus they share the same (low) precedence, necessitating extra parentheses in JSON expression with equality checks:

```
SELECT (JSON '{"field": 42}')->'field' = 42;
-- throws a Binder Error:
-- No function matches the given name and argument types 'json_extract(JSON, BOOLEAN)

SELECT ((JSON '{"field": 42}')->'field') = 42;
-- return true
```

This often caused confusion among users, therefore, the new release deprecates the old arrow lambda syntax and replaces it with Python-style lambda syntax:

```
SELECT list_transform([1, 2, 3], lambda x: x + 1);
```

To make the transition smoother, the deprecation will happen in several steps over the next year. First, DuckDB 1.3.0 introduces a new setting to configure the lambda syntax:

```
SET lambda_syntax = 'DEFAULT';
SET lambda_syntax = 'ENABLE_SINGLE_ARROW';
SET lambda_syntax = 'DISABLE_SINGLE_ARROW';
```

Currently, `DEFAULT` enables both syntax styles, i.e., the old single arrow syntax and the Python-style syntax. DuckDB 1.4.0 will be the last release supporting the single arrow syntax without explicitly enabling it. DuckDB 1.5.0 will disable the single arrow syntax by default. DuckDB 1.6.0 will remove the `lambda_syntax` flag and fully deprecate the single arrow syntax, so the old behavior will no longer be accessible.

Serializing Strings in List with Escapes

Starting with the new version, DuckDB escapes characters such as `'` in strings serialized in nested data structures to allow round-tripping between the serialized string and the nested representation. For example:

```
SELECT ['hello ''my'' world'] AS s;
```

DuckDB version 1.2.2 returns `[hello 'my' world]` while DuckDB 1.3.0 returns `['hello \'my\' world']`.

To serialize a list of strings with the old behavior, use the `array_to_string` function:

```
SELECT printf('[%s]', array_to_string(
        ['hello ''my'' world', 'hello ''cruel'' world'], ', '
    )) AS s;
```

```
┌──────────────────────────────────────────┐
│                    s                     │
│                 varchar                  │
├──────────────────────────────────────────┤
│ [hello 'my' world, hello 'cruel' world]  │
└──────────────────────────────────────────┘
```

Minor SQL Parser Changes

- The term AT now needs quotes to be used as an identifier as it is used for time travel in Iceberg.
- LAMBDA is now a reserved keyword due to the change in lambda syntax.
- GRANT is no longer a reserved keyword.

New Features

The new DuckDB release again contains a lot of exciting new features:

External File Cache

DuckDB is used a lot to read from remote files, e.g., Parquet files stored on HTTP servers or blob storage. Previous versions would always fully re-read file data. With this release, we added a cache for data from external files. This cache is subject to the overall DuckDB memory limit. If space is available, it will be used to dynamically cache data from external files. This should greatly improve re-running queries on remote data. For example:

```
.timer on
.mode trash -- do not show query result
FROM 's3://duckdb-blobs/data/shakespeare.parquet';
Run Time (s): real <1.456> user 0.037920 sys 0.028510
FROM 's3://duckdb-blobs/data/shakespeare.parquet';
Run Time (s): real <0.360> user 0.029188 sys 0.007620
```

We can see that the query is much faster the second time around due to the cache. In previous versions, the runtime would have been the same.

The cache contents can be queried using the `duckdb_external_file_cache()` table function like so:

```
.mode duckbox -- re-enable output
FROM duckdb_external_file_cache();
```

path varchar	nr_bytes int64	location int64	loaded boolean
s3://duckdb-blobs/data/shakespeare.parquet	1697483	4	true
s3://duckdb-blobs/data/shakespeare.parquet	16384	1681808	true

The cache is enabled by default but can be disabled with:

```
SET enable_external_file_cache = false;
```

Directly Query Data Files with the CLI

DuckDB's command line interface (CLI) gained the capability to directly query Parquet, CSV or JSON files. This works by just using e.g. a Parquet file instead of the database file. This will expose a view that can be queried. For example, say we have a Parquet file called `region.parquet`, this will work:

```
duckdb region.parquet -c 'FROM region;'
```

r_name varchar
AFRICA
AMERICA
ASIA
EUROPE
MIDDLE EAST

When using the CLI like this, what actually happens is that we launch a temporary in-memory DuckDB database, and create two views over the given file:

- `file` – this view is always named the same, regardless of the name of the file.
- `[base_file_name]` – this view depends on the name of the file, e.g., for `region.parquet` this is `region`.

Both views can be queried and will give the same result.

The main advantage of this feature is usability: we can use the regular shell to navigate to a file, and then use DuckDB to open that file without having to refer to the path of the file at the SQL level.

TRY Expression

DuckDB already supported TRY_CAST, which was trying to cast a value but did not fail the query if this was not possible. For example:

```
SELECT TRY_CAST('asdf' AS INTEGER);
```

returns NULL. This release generalizes this functionality beyond casting to arbitrary expressions that can error using TRY. For example, the logarithm of 0 is undefined, and `log(0)` will throw an exception and tell you that it "cannot take logarithm of zero". With the new TRY, this will return NULL instead, e.g.:

```
SELECT TRY(log(0));
```

```
NULL
```

Again, this will work for arbitrary expressions. We recommend to use TRY sparingly however if an error is expected often because there is going to be a performance impact. If any *batch* of rows causes an error, we switch to row-by-row execution of the expression to figure out exactly which row had an error and which did not. This is slower.

Updatings Structs

Starting with the new release, it's possible to update the sub-schema of structs using the `ALTER TABLE` clause. You can add, drop, and rename fields:

```
CREATE TABLE test(s STRUCT(i INTEGER, j INTEGER));
INSERT INTO test VALUES (ROW(1, 1)), (ROW(2, 2));
ALTER TABLE test DROP COLUMN s.i;
ALTER TABLE test ADD COLUMN s.k INTEGER;
ALTER TABLE test RENAME COLUMN s.j TO l;
```

```
┌──────────────────────────────┐
│              s               │
│ struct(l integer, k integer) │
├──────────────────────────────┤
│ {'l': 1, 'k': NULL}          │
│ {'l': 2, 'k': NULL}          │
└──────────────────────────────┘
```

Altering structs is also supported inside `LIST` and `MAP` columns.

Swapping in New Databases

The `ATTACH OR REPLACE` clause allows you to replace a database, so you can swap a database on the fly. For example:

```
ATTACH 'taxi_v1.duckdb' AS taxi;
USE taxi;
ATTACH OR REPLACE 'taxi_v2.duckdb' AS taxi;
```

This feature was implemented by external contributor `xevix`.

UUID v7 Support

> **Warning. Update (2025-05-23).** The UUID v7 implementation in DuckDB v1.3.0 is not consistent with the UUID standard, causing the timestamp values to be off. This means that the timestamps in the generated UUID v7 values will only be correct if they are used exclusively within DuckDB. Importing or exporting UUID v7 values from/to systems will yield incorrect timestamps. We patched this bug and the fix will soon be available in the preview builds and the upcoming 1.3.1 patch release.

DuckDB now supports UUID v7, which is a newer version of UUIDs. `UUIDv7` combines a Unix timestamp in milliseconds and random bits, offering both uniqueness and sortability. This is useful to e.g. order UUIDs by age or to combine the ubiquitous `ID` and `TIMESTAMP` columns in many tables into a single `UUIDv7` column.

New UUIDs can be created using the `uuidv7()` scalar function. For example:

```
SELECT uuidv7();
```

```
┌──────────────────────────────────────┐
│               uuidv7()               │
│                 uuid                 │
├──────────────────────────────────────┤
│ 8196f1f6-e3cf-7a74-bc0e-c89ac1ea1e19 │
└──────────────────────────────────────┘
```

There are also additional functions to determine the UUID version (`uuid_extract_version()`) and to extract the internal timestamp (`uuid_extract_timestamp()`), for example:

```
SELECT uuid_extract_version(uuidv7());
```

```
┌──────────────────────────────────────┐
│ uuid_extract_version(uuidv7())       │
│                uint32                │
├──────────────────────────────────────┤
│                  7                   │
└──────────────────────────────────────┘
```

```
SELECT uuid_extract_timestamp(uuidv7());
```

```
┌──────────────────────────────────────────┐
│ uuid_extract_timestamp(uuidv7())         │
│    timestamp with time zone              │
├──────────────────────────────────────────┤
│ 2025-05-21 08:32:14.61+00                │
└──────────────────────────────────────────┘
```

This feature was implemented by external contributor dentiny.

Expression Support in CREATE SECRET

DuckDB has an internal "secret" management facility for things like S3 credentials. With this release, it is possible to use scalar expressions in the creation of the secret. This allows for secret contents to not be specified in query text, which makes them easier to keep out of log files, etc. For example:

```
SET VARIABLE my_bearer_token = 'hocus pocus this token is bogus';
CREATE SECRET http (
    TYPE http,
    BEARER_TOKEN getvariable('my_bearer_token')
);
```

You can see that the BEARER_TOKEN field in the secret is set from the `getvariable` function in CREATE SECRET. In the CLI, this is also possible through *environment variables* using `getenv()`. For example, this is now possible:

```
MY_SECRET=asdf duckdb -c \
    "CREATE SECRET http (TYPE http, BEARER_TOKEN getenv('MY_SECRET'))"
```

Unpacking Columns

DuckDB v1.3.0 brings a further boost to the popular `COLUMNS(*)` expression. Previously, *unpacking* the entities into a list was possible by adding a leading `*` character:

```
CREATE TABLE tbl AS SELECT 21 AS a, 1.234 AS b;
SELECT [*COLUMNS(*)] AS col_exp FROM tbl;
```

```
┌──────────────────┐
│     col_exp      │
│ decimal(13,3)[]  │
├──────────────────┤
│ [21.000, 1.234]  │
└──────────────────┘
```

However, this syntax could not be used in tandem with other expressions such as casting:

```
SELECT [*COLUMNS(*)::VARCHAR] AS col_exp FROM tbl;
```

```
Binder Error:
*COLUMNS() can not be used in this place
```

The new UNPACK keyword removes this limitation. The following expression

```
SELECT [UNPACK(COLUMNS(*)::VARCHAR)] AS col_exp FROM tbl;
```

is equivalent to:

```
SELECT [a::VARCHAR, b::VARCHAR] AS col_exp FROM tbl;
```

```
┌──────────────┐
│   col_exp    │
│  varchar[]   │
├──────────────┤
│ [21, 1.234]  │
└──────────────┘
```

Spatial `JOIN` Operator

We added a new specialized join operator as part of the `spatial` extension, which greatly improves the efficiency of *spatial joins*, that is, queries that `JOIN` two geometry columns using specific spatial predicate functions, such as `ST_Intersects` and `ST_Contains`.

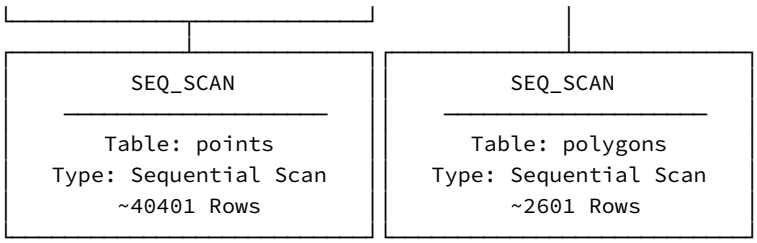
Similarly to a `HASH_JOIN`, the `SPATIAL_JOIN` builds a temporary lookup data-structure for the smaller side of the join, except it's an R-Tree, instead of a hash table. What this means for you is that you don't need to create an index first, or do any other pre-processing to optimize your spatial joins. It's all handled by the join operator internally.

While the query optimizer will try to instantiate this new operator for `LEFT`, `OUTER`, `INNER` and `RIGHT` spatial joins, one limitation currently is that the join can only contain a single join condition, or the optimizer will fall back to use a less efficient join strategy.

The following example illustrates how the `SPATIAL_JOIN` operator becomes part of the query plan. It's a relatively small query, but on my machine it executes almost 100× faster than it used to do in DuckDB v1.2.2!

```
LOAD spatial;

-- generate random points
CREATE TABLE points AS
  SELECT
      ST_Point(x, y) AS geom,
      (y * 50) + x // 10 AS id
  FROM
      generate_series(0, 1000, 5) r1(x),
      generate_series(0, 1000, 5) r2(y);

-- generate random polygons
CREATE TABLE polygons AS
  SELECT
      ST_Buffer(ST_Point(x, y), 5) AS geom,
      (y * 50) + x // 10 AS id
  FROM
      generate_series(0, 500, 10) r1(x),
      generate_series(0, 500, 10) r2(y);

-- inspect the join plan
EXPLAIN
    SELECT *
    FROM polygons
    JOIN points ON ST_Intersects(points.geom, polygons.geom);
```

```
             ...
┌─────────────┴─────────────┐
│       SPATIAL_JOIN        │
│    ──────────────────     │
│      Join Type: INNER     │
│        Conditions:        │
│ ST_Intersects(geom, geom) │
│       ~40401 Rows         │
└───────────────────────────┘
```



For the curious, there are more details in the PR.

Internal Changes

For this release there have also been a large number of internal changes.

We have completed an almost complete re-implementation of DuckDB's *Parquet reader and writer*. This should greatly improve Parquet performance and reliability, and also expanded Parquet feature support for obscure logical types like `UNKNOWN` and `FLOAT16`.

We have also done a large amount of internal changes around the reading of multiple files (e.g., a folder of Parquet files) in an API called the `MultiFileReader`. We have unified the handling of multiple files across many of our file readers, e.g., Parquet, CSV, JSON, Avro, etc. This allows DuckDB to handle e.g. schema differences between multiple files in a unified way.

We have also added a new string compression method, `DICT_FSST`. Before, DuckDB supported *either* dictionary encoding *or* FSST compression ("Fast Static Symbol Table") for strings. Those compression methods could not be mixed within a storage block (265 kB by default). However, we observed a lot of real-world data where part of the block would benefit from dictionary encoding, and another part would benefit from FSST. FSST does not by default duplicate-eliminate strings. This release combines both methods into a new compression method, `DICT_FFST`. This *first* runs dictionary encoding and *then* compresses the dictionary using FSST. Dictionary encoding and FSST-only encoding are also still available. We have also optimized storing validity masks ("which rows are NULL?") in this release, some compression methods (like the new `DICT_FSST`) can handle NULLs internally and this obviates the need for a separate validity mask. Combined, those new features should greatly reduce the storage space required especially for strings. Note that the compression method is automatically picked by DuckDB based on actually observed compression ratios so users will not have to explicitly set this.

Final Thoughts

These were a few highlights – but there are many more features and improvements in this release. There have been over 3,000 commits by over 75 contributors since we released v1.2.2. The full release notes can be found on GitHub. We would like to thank our community for providing detailed issue reports and feedback. And our special thanks goes to external contributors, who directly landed features in this release!

Arrow IPC Support in DuckDB

Publication date: 2025-05-23

Authors: Pedro Holanda (DuckDB Labs), Ian Cook (Columnar), Dewey Dunnington (Wherobots), Bryce Mecum (Voltron Data)

TL;DR: DuckDB now supports consuming and producing the Arrow IPC Serialization Format through the `arrow` community extension.

DuckDB and Arrow

In recent years, the Apache Arrow project has gained a lot of traction in the data world, thanks to its columnar format that allows for easy interchange of data between different systems – mostly with a zero-copy approach. Apache Arrow powers, among others, the integration between DuckDB and Polars. In practice, when DuckDB produces or consumes a Polars DataFrame, it is actually using the Arrow columnar format underneath.

The importance of having such a format is also one of the main reasons DuckDB was among the pioneers in integrating with Arrow and implementing an Arrow Database Connectivity (ADBC) interface – particularly because Arrow makes this possible with no additional dependencies, thanks to its C data interface.

But one limitation of Arrow's C data interface is that it exchanges data using pointers (memory addresses). This limits the possibilities if you want to exchange Arrow data between different processes or systems. To overcome this limitation, the Arrow project also specifies the Arrow IPC format, which allows users to efficiently serialize Arrow columnar data and pass it between processes or over a network. This data can be consumed as a stream, either directly from a memory buffer or from a file.

We're thrilled to announce that DuckDB is now able to consume and produce these Arrow streams via the new `arrow` community extension. In this post, we will describe the Arrow IPC serialization format in more detail, show how to install the new `arrow` community extension for DuckDB, and give a demo showing how to use it.

Arrow Interprocess Communication (Arrow IPC)

The Arrow IPC format provides a way of serializing (and optionally compressing) Arrow-formatted data, enabling you to transfer data over a network or store it on disk while keeping it in Arrow format, avoiding the overhead of converting it to a different format. Arrow IPC supports LZ4 and ZSTD compression, and when stored as a file, it also supports a file footer that can be used to speed up retrieval and processing by allowing parts of the data to be skipped (similar to the approach used by the Parquet format). When compared to Parquet, the Arrow IPC format has two main benefits:

1. **Ease of implementation:** Writing a low-level Arrow IPC consumer/producer is less complex than writing a Parquet one, especially if the system already integrates with the Arrow format.
2. **Faster encoding and decoding:** The process of encoding and decoding (serializing and deserializing) Arrow data is much simpler and faster than with Parquet. This can yield faster processing times—especially if you are streaming data that does not need to be stored on disk afterwards.

Arrow and Parquet are complementary formats. Parquet's sophisticated system of encoding and compression options typically yields much smaller files, making Parquet a better choice for archival storage. Arrow's ability to eliminate encoding and decoding overheads typically yields faster and more efficient data interchange, making Arrow a better choice for query result transfer and ephemeral caching.

To give you an illustration of just how simple the Arrow IPC format is, consider the following illustration. In the Arrow IPC format, a table is serialized as a sequence of record batches (a collection of records organized in a columnar layout), preceded by their shared schema:

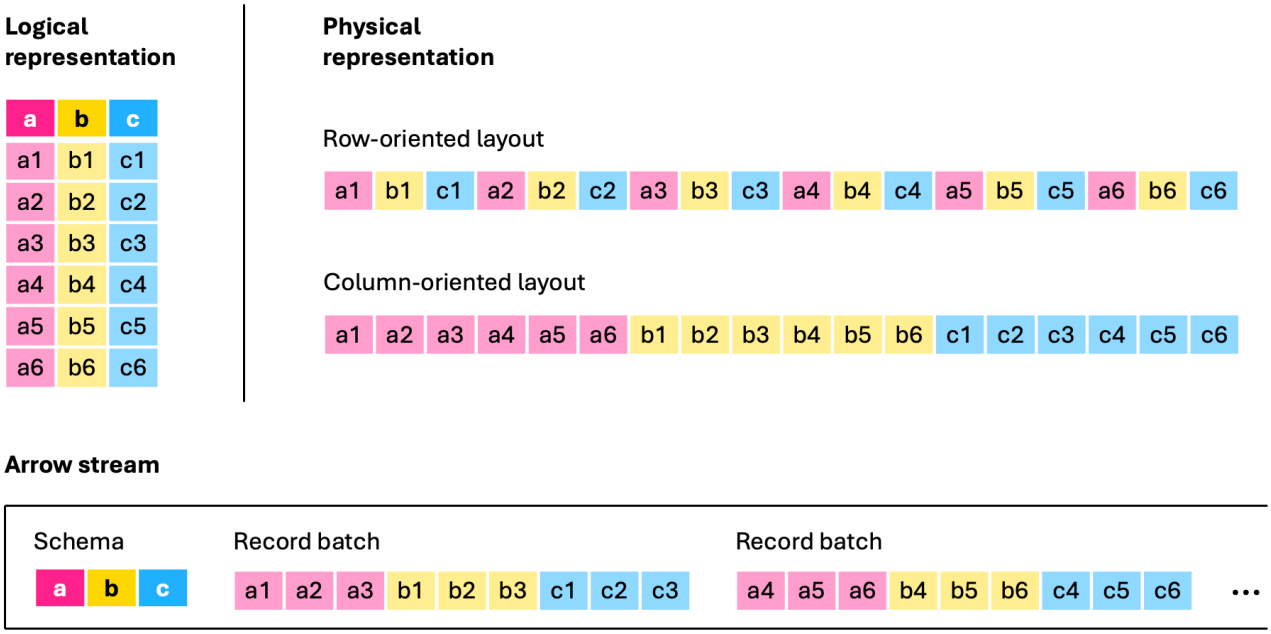


Figure from Apache Arrow Blog: How the Apache Arrow Format Accelerates Query Result Transfer.

Note that in realistic scenarios, record batches are much larger and the figure above is simplified for illustrative purposes.

The Arrow Community Extension

DuckDB has included an integration with the Arrow IPC format for many years, via the (now-deprecated) Arrow core extension. However, the main purpose of this support was to allow DuckDB interoperability with JavaScript, hence it was designed only to read in-memory serialized buffers, and not Arrow IPC files. The extension's code complexity and maintainability were very high, because working with Arrow IPC required having the entire Arrow C++ library as a dependency, as we did not want to write our own serialization and deserialization code for the Arrow IPC format.

More recently, a much smaller Arrow C++ implementation started to gain popularity as a way to interact with Arrow IPC data: the nanoarrow library. Using nanoarrow, we completely redesigned the old DuckDB Arrow extension to have a much smaller dependency, a cleaner codebase, and the ability to scan Arrow IPC files. We also took the opportunity to shift the Arrow DuckDB extension from a core extension to a community extension. This change was made for two main reasons. The first is to enable the Arrow developer and user community to be more involved in building and supporting the extension. The second is to have a release schedule that is not tied to the DuckDB release schedule. In practice, this means that members of the core Arrow developer community can decide when a new version of the extension will be released.

Installing and loading the new Arrow extension is very simple:

```
INSTALL arrow FROM community;
LOAD arrow;
```

Demo

In this demo, we will use the new Arrow DuckDB extension to generate the `lineitem` TPC-H table with scale factor 10 as an Arrow IPC file. While our demo will focus on Arrow IPC data stored in a file, the extension itself also allows you to consume and produce the Arrow IPC format directly as buffers. You can find detailed examples of usage and accepted parameters in the extension's README.

We start off by loading the `arrow` extension and generating our TPC-H tables.

```
LOAD arrow;

CALL dbgen(sf = 10);
```

To generate the Arrow IPC files we can simply use the `COPY ... TO ...` clause, as follows. We use the recommended file extension `.arrows` since this file is in the Arrow IPC stream format.

```
COPY lineitem TO 'lineitem.arrows';
```

> In this demo, for simplicity, we wrote our table in a single file. However, our Arrow `COPY ... TO ...` clause allows us to set the `chunk_size` and the number of `row_groups` per file. These options allow us to produce data optimized for the best possible performance for your use case. For example, a smaller `chunk_size` may reduce overall performance but benefit streaming scenarios.

We can now run TPC-H query 6 directly on our file using the `read_arrow` function:

```
SELECT
    sum(l_extendedprice * l_discount) AS revenue
FROM
    read_arrow('lineitem.arrows')
WHERE
    l_shipdate >= CAST('1994-01-01' AS date)
    AND l_shipdate < CAST('1995-01-01' AS date)
    AND l_discount BETWEEN 0.05
    AND 0.07
    AND l_quantity < 24;
```

which prints:

```
┌──────────────────┐
│     revenue      │
│  decimal(38,4)   │
├──────────────────┤
│ 1230113636.0101  │
│ (1.23 billion)   │
└──────────────────┘
```

Thanks to replacement scans, you can omit the function `read_arrow` if the filename ends with `.arrow` or `.arrows`. For example:

```
SELECT count(*) FROM 'lineitem.arrows';
```

which prints:

```
┌──────────────────┐
│  count_star()    │
│      int64       │
├──────────────────┤
│     59986052     │
│ (59.99 million)  │
└──────────────────┘
```

> For simplicity, we focus on the scenario of reading a single file, but our reader supports multi-file reading, with functionality on par with the DuckDB Parquet reader.

What if you want to fetch an Arrow IPC stream directly from a server into DuckDB? To demonstrate this, we can start an HTTP file server in the same directory where we saved `lineitem.arrows`. We use the Node.js package serve to do this (instead of Python's built-in `http.server`) because it supports HTTP range requests:

```
npx serve -l 8008
```

Then you can use DuckDB's `httpfs` extension to query the Arrow data over the HTTP(S) protocol:

```
INSTALL httpfs;
LOAD httpfs;
LOAD arrow;

SELECT count(*) FROM read_arrow('http://localhost:8008/lineitem.arrows');
```

which prints the same result:

```
┌──────────────────┐
│  count_star()    │
│      int64       │
├──────────────────┤
│    59986052      │
│ (59.99 million)  │
└──────────────────┘
```

Alternatively, you can use a tool like `curl` to fetch Arrow IPC data from a server and pipe it to DuckDB in the terminal:

```
URL="http://localhost:8008/lineitem.arrows"
SQL="LOAD arrow; FROM read_arrow('/dev/stdin') SELECT count(*);"

curl -s "$URL" | duckdb -c "$SQL"
```

which prints the same result. For other demos of the `arrow` extension, see our arrow-ipc demo repository.

Bonus: Cool Use Cases for Arrow IPC

Running a DuckDB query against data in Arrow IPC format like in the demo above is a pretty neat trick, and it works so well because DuckDB and Arrow are a natural pair due to both using a columnar data layout. However, you may be wondering what else you can do with Arrow IPC data. One of Arrow's main goals is interoperability, and by saving our data in Arrow IPC format, we've opened up many options for connecting with other tools.

For example, we can now work with our data with PyArrow:

```
import pyarrow as pa

with open('lineitem.arrows', 'rb') as source:
   stream = pa.ipc.open_stream(source)
   tbl = stream.read_all()
```

or Polars:

```
import polars as pl

tbl = pl.read_ipc_stream("lineitem.arrows")
```

or ClickHouse:

```
CREATE TABLE
    lineitem
ENGINE MergeTree()
ORDER BY tuple()
AS
    SELECT * FROM file('lineitem.arrows', 'ArrowStream');
```

or any of the numerous other Arrow libraries (available in a dozen different languages) or Arrow-compatible systems.

The benefits of Arrow IPC don't stop there: Arrow IPC is also ideal for larger-than-memory use cases. Using PyArrow, we can memory-map our `lineitem.arrows` file and work with it without reading the entire thing into memory:

```
import pyarrow as pa

with pa.memory_map('lineitem.arrows', 'rb') as source:
    stream = pa.ipc.open_stream(source)
    tbl = stream.read_all()

tbl.num_rows
# => 59986052
```

Then, we can check that PyArrow didn't have to allocate any buffers to hold the data because it all lives on disk:

```
pa.total_allocated_bytes()
# => 0
```

Now we can perform the same query we did in the demo above and show we get the same result:

```
import datetime
import pyarrow.compute as pc

subset = tbl.filter(
    (pc.field("l_shipdate") >= datetime.datetime(1994, 1, 1)) &
    (pc.field("l_shipdate") < datetime.datetime(1995, 1, 1)) &
    (pc.field("l_discount") >= 0.05) &
    (pc.field("l_discount") <= 0.07) &
    (pc.field("l_quantity") < 24.)
)
pc.sum(pc.multiply(subset.column("l_extendedprice"), subset.column("l_discount")))
# => <pyarrow.Decimal128Scalar: Decimal('1230113636.0101')>
```

And, despite `lineitem.arrows` being over 10 GB, PyArrow only had to allocate a fraction of the memory:

```
pa.total_allocated_bytes()
# => 201594240 (192MB)
```

Conclusion & What's Next

In this blog post, we presented the new Arrow community extension, which enables DuckDB users to interact with Arrow IPC streaming buffers and files. Special thanks to Voltron Data for enabling this extension by working with DuckDB Labs. Below we list our future plans for this extension:

- Support for both ZSTD and LZ4 compression when writing Arrow IPC. DuckDB currently only supports writing uncompressed buffers.
- Support for LZ4 compression when reading Arrow buffers. The reader currently only supports ZSTD or uncompressed buffers.
- Support for writing the Arrow IPC file format containing the file footer, and using the footer to speed up reads.
- Implementation of C API DuckDB functions to produce and consume Arrow IPC data.

If you'd like to work on any of these planned features or suggest other features, or if you find any bugs, feel free to log them in our issue tracker. Happy hacking!

USING KEY in Recursive CTEs

Publication date: 2025-05-23

Authors: Björn Bamberg and Torsten Grust

TL;DR: Recursive CTEs in SQL allow for powerful iterative queries like graph traversals but can be memory-intensive and slow due to repeated row accumulation. DuckDB's new `USING KEY` feature addresses this by treating intermediate results as keyed dictionaries instead of an ever growing set: existing entries can be updated. This can lead to dramatically better performance and memory efficiency, especially in graph algorithms like shortest path and distance-vector routing. It also simplifies query logic by providing direct access to the entire dictionary.

Assembling SQL Queries from Pieces: CTEs

As SQL queries become more complex, managing their readability, modularity and reusability increasingly becomes a challenge. Common Table Expressions (CTEs) were introduced to address these issues by allowing developers to define temporary, named result sets within a query. Similar to functions in programming, CTEs allow a large query to be broken down into logical building blocks, making it easier to understand, maintain and debug.

CTEs are particularly useful for structuring multi-step transformations that might otherwise require deeply nested subqueries or complex joins. By improving both the clarity and structure of SQL code, CTEs have become an essential tool in modern query writing – enabling the clear, declarative expression of even the most sophisticated logic.

Iterate Like It's 1999: Recursive CTEs

To enhance the expressive power of SQL, **recursive CTEs** were introduced in the SQL:1999 standard. These allow a query to reference the results from previous iterations within the same expression, enabling SQL to solve more complex problems such as graph traversal and other iterative computations.

This capability pushes SQL beyond basic data retrieval, allowing for the formulation of complex, iterative logic directly in SQL. In fact, recursive CTEs make SQL Turing-complete, meaning it can theoretically express any computation (given sufficient time and memory).

But how do recursive CTEs work in DuckDB?

Let's look at a simple example to break down the mechanism. Suppose we want to calculate the largest power of 2 that is smaller than 100. We can use a recursive CTE `power` to generate powers of 2 iteratively until we reach that limit. For any row `(a, b, c)` in `power` we will have that `a^b = c`:

```
WITH RECURSIVE power(a, b, c) AS (
    SELECT 2, 0, 1        -- 2^0 = 1
        UNION
    SELECT a, b+1, a * c -- a^(b+1) = a * a^b
    FROM power           -- reads the working table (contains a single row)
    WHERE a * c < 100
)
FROM power;          -- reads the union table (contains all intermediate results)
```

We can divide a recursive CTE into two parts, separated by the `UNION` keyword. The part above the `UNION` is the **non-recursive part** (`SELECT 2, 0, 1` in our example), and the part below is the **recursive part**.

In the recursive part, CTE `power` references itself. This self-reference points to what we call the ***working table***. The working table always holds the rows produced in the **immediately preceding iteration**. And only those.

Here's how it works step by step:

- First, the **non-recursive part** is executed, producing initial rows – in our example, just the row `(2, 0, 1)`. These rows are stored in the ***working table***.
- Then, the **recursive part** is executed using the rows from the working table. Every new row the recursive part produces is stored in the ***intermediate table***, which holds results from the current iteration.
- If the intermediate table is **empty**, the iteration ends.
- Otherwise, we clear the working table and replace it with the contents of the intermediate table – preparing for the next iteration.
- We additionally add the contents of the intermediate table to the ***union table***, which accumulates all intermediate results across iterations.

Here you can see the entries of the three involved tables in each iteration of the CTE:

Iteration	Output of Recursive Step	Working Table	Intermediate Table	Union Table
0	SELECT 2, 0, 1	∅ (no rows)	(2, 0, 1)	(2, 0, 1)
1	2 * 1 = 2	(2, 0, 1)	(2, 1, 2)	(2, 0, 1) (2, 1, 2)
2	2 * 2 = 4	(2, 1, 2)	(2, 2, 4)	(2, 0, 1) (2, 1, 2) (2, 2, 4)
3	2 * 4 = 8	(2, 2, 4)	(2, 3, 8)	(2, 0, 1) ... (2, 3, 8)
4	2 * 8 = 16	(2, 3, 8)	(2, 4, 16)	(2, 0, 1) ... (2, 4, 16)
5	2 * 16 = 32	(2, 4, 16)	(2, 5, 32)	(2, 0, 1) ... (2, 5, 32)
6	2 * 32 = 64	(2, 5, 32)	(2, 6, 64)	(2, 0, 1) ... (2, 6, 64)
7	2 * 64 = 128	(2, 6, 64)	∅ *stop!*	(2, 0, 1) ... (2, 6, 64)

When the recursive CTE completes, the union table holds the entire result set, including all intermediate rows from every iteration:

```
-- a^b = c
```

a int32	b int32	c int32
2	0	1
2	1	2
2	2	4
2	3	8
2	4	16
2	5	32
2	6	64

The union table provides a complete record of the history of the power computation. This can lead to unnecessary overhead, especially if all we need is the last row – the final result of the recursion. Storing every intermediate value can be inefficient, in particular when the intermediate results aren't required or when working with large datasets (that involve many rows or wide rows, e.g., in the presence of array-typed columns).

Recursive CTEs Suffer from Amnesia

While the union table holds the just mentioned history *once the computation is complete,* recursive CTEs suffer from a case of ”amnesia” *while the iteration is going on:* the recursive part only ever sees the intermediate results of the *immediately preceding* iteration. This can be limiting and many of us will have seen how query authors work around this limitation in terms of manually maintained arrays (or similar container structures) that hold information about previous iterations. This may be costly. Then again, enabling the recursive part to access

the union table with all its accumulated – potentially sizable – intermediate results may easily incur performance problems while we iterate. A real conundrum.

USING a KEY to Cure Amnesia

Can we afford to allow the recursive part to access the union table? Yes, but we need a means to control its size. Operating the union table in *append-only mode* thus is a no-go. Instead, let the recursive part optionally *overwrite* existing rows in the union table with fresh information that has been computed in the current iteration. This can reduce the union table's size significantly (see our experiments below).

Starting with version 1.3, DuckDB features a `USING KEY` variant of recursive CTEs that incorporates just this idea.

> If you would like to read more about its origins, check out CIDR 2023. For further information about implementation, see SIGMOD 2025 (to be published).

The variant introduces two major differences compared to traditional recursive CTEs:

1. It provides access to the working table (as always) as well as **access to the union table** – which we now call the ***recurring table***.
2. Instead of simply appending new rows to the recurring table, the table now functions more like a dictionary (similar to a Python `dict`), allowing **key-based updates**.

To use this new feature, add the `USING KEY (...)` clause to your recursive CTE:

```
WITH RECURSIVE power(a, b, c) USING KEY (a)
    -- key: a, payload: b, c
AS (
    ...
)
```

With `USING KEY`, the schema of the recursive CTE is divided into **key columns** and **payload columns**. The key columns are specified using the `USING KEY (<column names>)`{:.language-sql .highlight} clause, while the remaining columns are treated as payload.

This division affects how the recurring table behaves. Instead of stubbornly appending new rows on each iteration, it acts more like a dictionary: if the recursive part returns a row that hasn't been seen before, it is added to the recurring table as usual. But if a row shares a key with an existing entry in the recurring table, the payload is updated – replacing the previous values for that key in the recurring table.

> If multiple rows with the same key are produced in a single iteration, only the last one is retained. Therefore, you may wish to use the `ORDER BY` clause in the recursive part to control which row is kept.

This approach allows recursive queries to maintain and update state more efficiently, especially for algorithms in which keeping the latest (or "best") value for a given key is crucial.

Overwriting Old Intermediate Results

Let us pick up our recursive example query from above. Now we compute the powers of bases 2 and 3, as long as these are less than 100:

```
WITH RECURSIVE power(a, b, c) USING KEY (a) AS (
    FROM (VALUES (2, 0, 1), (3, 0, 1))   -- 2^0 = 1, 3^0 = 1
        UNION
    SELECT a, b+1, a * c                 -- a^(b+1) = a * a^b
    FROM power                           -- reads the working table (contains two rows)
    WHERE a * c < 100
)
FROM power;                              -- reads the recurring table (contains two rows)
```

We start with two rows in the **non-recursive part**: one with a key (base) of 2 and another with a key of 3. In the **recursive part**, we multiply the intermediate power by the base. This produces two rows, again with the keys 2 and 3.

Unlike traditional recursive CTEs, we **do not append** these two new rows to the recurring table. Instead, we **update the existing rows** with keys 2 and 3 in the recurring table, overwriting their payload values. We thus keep only two rows throughout the entire computation, each holding the current power value for its key (or base) in column a.

Iteration	Output of Recursive Step	Working Table	Intermediate Table	Recurring Table
0	SELECT 2, 0, 1 SELECT 3, 0, 1	∅ (no rows)	(2, 0, 1) (3, 0, 1)	(2, 0, 1) (3, 0, 1)
1	2 * 1 = 2 3 * 1 = 3	(2, 0, 1) (3, 0, 1)	(2, 1, 2) (3, 1, 3)	(2, 1, 2) (3, 1, 3)
2	2 * 2 = 4 3 * 3 = 9	(2, 1, 2) (3, 1, 3)	(2, 2, 4) (3, 2, 9)	(2, 2, 4) (3, 2, 9)
3	2 * 4 = 8 3 * 9 = 27	(2, 2, 4) (3, 2, 9)	(2, 3, 8) (3, 3, 27)	(2, 3, 8) (3, 3, 27)
4	2 * 8 = 16 3 * 27 = 81	(2, 3, 8) (3, 3, 27)	(2, 4, 16) (3, 4, 81)	(2, 4, 16) (3, 4, 81)
5	2 * 16 = 32 3 * 81 = 243	(2, 4, 16) (3, 4, 81)	(2, 5, 32)	(2, 5, 32) (3, 4, 81)
6	2 * 32 = 64	(2, 5, 32)	(2, 6, 64)	(2, 6, 64) (3, 4, 81)
7	2 * 64 = 128	(2, 6, 64)	∅ *stop!*	(2, 6, 64) (3, 4, 81)

As we can see, the size of the recurring table remains constant. Irrelevant computation history is overwritten which leads to **reduced memory usage**. The final recurring table reads:

```
┌─────────┬─────────┬─────────┐
│    a    │    b    │    c    │  -- a^b = c
│  int32  │  int32  │  int32  │
├─────────┼─────────┼─────────┤
│       2 │       6 │      64 │
│       3 │       4 │      81 │
└─────────┴─────────┴─────────┘
```

This behavior is especially useful with algorithms in which we are interested in the **latest**, **best**, or **smallest** value for a given key. The maximum number of rows in the recurring table is now **bounded by the number of unique keys**. Since the number of distinct keys in use is under control of the recursive part, this can be a powerful advantage when working with sizable datasets.

A Change of Key

Now, *should* you happen to be interested in the history of the computation and are willing to invest the memory space, all that is required is a change of key. With

```
WITH RECURSIVE power(a, b, c)
    USING KEY (a, b)
AS (  -- formerly: USING KEY (a)
    ...
)
FROM power
ORDER BY a, b;
```

the iteration counter (or exponent) in column b is considered to be part of the key, too. The recurring table will thus track unique `(a, b)` combinations (i.e., base, exponent) and we are able to trace what was going on during iteration:

```
┌─────────┬─────────┬─────────┐
│    a    │    b    │    c    │
│  int32  │  int32  │  int32  │
├─────────┼─────────┼─────────┤
│       2 │       0 │       1 │
│       2 │       1 │       2 │
│       2 │       2 │       4 │
│       2 │       3 │       8 │
└─────────┴─────────┴─────────┘
```

```
┌─────────┬─────────┬─────────┐
│    2    │    4    │   16    │
│    2    │    5    │   32    │
│    2    │    6    │   64    │
│    3    │    0    │    1    │
│    3    │    1    │    3    │
│    3    │    2    │    9    │
│    3    │    3    │   27    │
│    3    │    4    │   81    │
└─────────┴─────────┴─────────┘
```

Accessing Relevant History

Another major difference to vanilla recursive CTEs: now that the size of the recurring table is under control, we can afford to reference it directly in the recursive part of the CTE. This allows us to access *any intermediate result* that has not been overwritten yet – regardless of which iteration computed these results. No amnesia anymore! To access the recurring table, simply prefix the CTE name with the pseudo-schema name `recurring`:

```
WITH RECURSIVE t(...) USING KEY (...) AS (
    ...
    FROM recurring.t  -- reads the recurring table while we iterate
)
...
```

`USING KEY` Can Unlock Performance Advantages

To further highlight the differences between vanilla and key-based CTEs, let's consider a more complex example using a graph dataset – a social network graph.

> The graphs were derived from the LDBC Social Network Benchmark (SNB), which provides a generator for synthetic social network data. To make the graphs more manageable for use with the `WITH RECURSIVE` query, we further filtered them by person name, reducing their density.

In this dataset, nodes represent people while edges correspond to relationships between them. The tables we're working with are `Person(id)`, which contains all existing ids in the network and `knows(person1id, person2id)` in which each row holds a pair of people knowing each other.

> If you would like to try this out for yourself, begin by attaching the database to any DuckDB session.

```
ATTACH 'https://blobs.duckdb.org/data/using-key-graph.duckdb';
USE 'using-key-graph';
```

Our goal is to compute the *shortest path between all pairs of people* in the social network. Since this is a problem with inherent quadratic complexity, we need to closely eye runtime and memory requirements. For each pair, we start by adding a row that has one person as the start node and the other as the target node. We then iteratively explore all individuals known by the start node, continuing the traversal until the target person is reached. Paths through the network are encoded by `via` nodes: to reach the target from the start node, first proceed to the `via` node (an immediate neighbor of the start node) – once there, use that node's `via` entry to continue your traversal.

A recursive CTE that implements this approach reads as follows:

```
WITH RECURSIVE paths(here, current, via, len, there, completed, found) AS (
  SELECT n1.id AS here, n1.id AS current, NULL::BIGINT AS via,
         0 AS len, n2.id AS there, false AS completed, false AS found
    FROM Person AS n1 JOIN Person AS n2 ON (n2.id <> n1.id)
    UNION ALL
  SELECT paths.here,
         person2id AS current,
```

```
        coalesce(paths.via, knows.person2id) AS via,
        paths.len+1 AS len,
        paths.there,
        bool_or(knows.person2id = paths.there)
                OVER (PARTITION BY paths.here, paths.there
                      ROWS BETWEEN UNBOUNDED PRECEDING
                               AND UNBOUNDED FOLLOWING) AS completed,
        knows.person2id = paths.there AS found
    FROM paths
    JOIN knows ON (paths.current = knows.person1id AND NOT paths.completed)
)
SELECT here, there, via, len
FROM paths WHERE found
ORDER BY len, here, there;
```

> The working table of recursive CTE `paths` has seven columns:
>
> - `here` refers to the person where the traversal starts,
> - `there` refers to the target person we aim to reach,
> - `via` indicates the immediate neighbour of the start node of the current path,
> - `len` is the length of the current path, which is incremented by one for each step taken in the traversal.
>
> The remaining three columns control the traversal logic and optimize the computation.
>
> - `current` refers to the node that is currently being explored during the traversal,
> - `found` indicates whether the current path has successfully reached the target person,
> - `completed` tracks whether any path with the same `here` and `there` has already reached the target. This prevents further traversal for that pair once a shortest path has been found, thus avoiding the exploration of longer paths.

If we search in a large graph with many edges, the **union table** in a vanilla recursive CTE can grow large, potentially exceeding memory limits. This not only causes significant **performance issues** but can even lead to **query crashes** in extreme cases.

The new key-based CTEs can avoid this problem by changing how the state of the search is maintained. This enables a range of new algorithms to be expressed efficiently in SQL, including algorithms for finding the shortest paths in large graphs.

One such algorithm is **Distance Vector Routing (DVR)**, a method for computing the shortest paths in a network based on node-local routing table that indicate where to "hop next".

- In DVR, each node maintains a **routing table** that records the length of the paths that reach other nodes.
- We use the **recurring table** to store these routing tables for *all* nodes in the network: a row `(here, there, via, len)` indicates that the first hop on the path from node `here` to node `there` is node `via`. The length of the overall path is `len`.
- When a shorter path to a target node is found, the corresponding routing table entry is updated.
- Routing updates found in the last iteration are distributed to neighboring nodes through the **working table**.

To check if a newly incoming routing update improves a currently known path, we perform a lookup in the recurring table. If the new length is smaller than that of the known path, we update the entry and propagate the update to our immediate neighbors. This mechanism allows for efficient path finding even in very large graphs – see below how DVR outperforms the above approach based on a vanilla recursive CTEs.

```
WITH RECURSIVE dvr(here, there, via, len) USING KEY (here, there) AS (
  -- initialize routing tables for all nodes, only the routes to
  -- immediate neighbors are known at this time
  SELECT n.person1id AS here, n.person2id AS there, n.person2id AS via, 1::DOUBLE AS len
  FROM   knows AS n
    UNION
  (SELECT n.person1id AS here, dvr.there, dvr.here AS via, 1 + dvr.len AS len
    FROM dvr -- working table - routing updates shared by neighbors
    JOIN knows AS n ON
        (n.person2id =  dvr.here AND    -- send update only to immediate neighbors
        n.person1id <> dvr.there)       -- no need to store a route to myself
```

```
    LEFT JOIN recurring.dvr AS rec ON   -- recurring table (current routing tables)
        (rec.here = n.person1id AND rec.there = dvr.there) -- identify affected routing table entry
    WHERE 1 + dvr.len < coalesce(rec.len, 'Infinity'::DOUBLE) --  does the routing update improve the
    entry in the routing table?
    ORDER BY len -- shortest path first
  )
)
FROM dvr
ORDER BY len, here, there;
```

The larger the social network graphs (A to G), the more pronounced is the performance gap between vanilla recursive CTEs (REC) and the new `USING KEY` variant (KEY). The table below reports **the number of rows processed during each iteration**:

graph	nodes	edges	KEY	REC
A	184	233	744	352,906
B	322	903	8,232	40,732,577
C	424	1,446	19,213	605,859,791
D	484	2,049	30,871	❌
E	1,119	8,809	255,425	❌
F	1,481	14,256	491,880	❌
G	1,618	16,619	607,926	❌

Even for the smallest graph, Graph A, the difference is significant: the REC CTE produces around 350,000 rows, while KEY generates only 744 rows. As the graph size increases, the gap becomes even more striking. In Graph C, with 424 nodes and 1,446 edges, the REC approach processes nearly 1 billion rows, while the KEY method handles less than 20,000 rows. Although this isn't the largest graph in our benchmark, the REC approach is already approaching out-of-memory conditions (OOM, ❌).

This substantial difference in memory usage is only part of the story. The performance of the REC approach also degrades quickly. While both CTEs perform similarly on small graphs, REC becomes significantly slower as the graph grows – eventually crashing – whereas KEY continues to scale smoothly:

And that's the beauty of the new `USING KEY` CTEs. It enables more efficient expression of complex iterative algorithms by providing key-based control over the size of intermediate results that are passed from iteration to iteration – memory pressure lessens, run time performance can go up significantly. If you're working with recursive CTEs in DuckDB, be sure to take advantage of this powerful addition – it may make a significant difference in your queries.

DuckLake: SQL as a Lakehouse Format

Publication date: 2025-05-27

Authors: Mark Raasveldt and Hannes Mühleisen

TL;DR: DuckLake simplifies lakehouses by using a standard SQL database for all metadata, instead of complex file-based systems, while still storing data in open formats like Parquet. This makes it more reliable, faster, and easier to manage.

> The first part of the blog post is shared with the DuckLake manifesto. Jump to the DuckLake extension section to read the rest.

Background

Innovative data systems like BigQuery and Snowflake have shown that *disconnecting storage and compute* is a great idea in a time where storage is a virtualized commodity. That way, both storage and compute can scale independently and we don't have to buy expensive database machines just to store tables we will never read.

At the same time, market forces have pushed people to insist that data systems *use open formats* like Parquet to avoid the all-too-common hostage taking of data by a single vendor. In this new world, lots of data systems happily frolic around a pristine "data lake" built on Parquet and S3 and all was well. Who needs those old school databases anyway!

But quickly it emerged that – shockingly – people would like to *make changes* to their dataset. Simple appends worked pretty well by just dropping more files into a folder, but anything beyond that required complex and error-prone custom scripts without any notion of correctness or – Codd beware – transactional guarantees.

An actual lakehouse *An actual lakehouse. Maybe more like a cabin on a lake.*{: .caption }

Iceberg and Delta

To address the basic task of changing data in the lake, various new open standards emerged, most prominently Apache Iceberg and Linux Foundation Delta Lake. Both formats were designed to essentially recover some sanity of making changes to tables *without* giving up the basic premise: use open formats on blob storage. For example, Iceberg uses a maze of JSON and Avro files to define schemas, snapshots and which Parquet files are part of the table at a point in time. The result was christened the "Lakehouse", effectively an addition of database features to data lakes that enabled a lot of new exciting use cases for data management, e.g., cross-engine data sharing.

Iceberg table architecture{: .lightmode-img } Iceberg table architecture{: .darkmode-img } *Iceberg table architecture*{: .caption }

But both formats hit a snag in the road: finding the latest version of a table is tricky in blob stores with their mercurial consistency guarantees. It's tricky to atomically (the "A" in ACID) swap a pointer to make sure everyone sees the latest version. Iceberg and Delta Lake also only really know about a single table, but people – again, shockingly – wanted to manage multiple tables.

Catalogs

The solution was another layer of technology: we added a catalog service on top of the various files. That catalog service in turn talks to a *database* that manages all the table folder names. It also manages the saddest table of all time that only contains a single row for each table with the current version number. We can now borrow the database's transactional guarantees around updating that number and everyone's happy.

Iceberg catalog architecture{: .lightmode-img } Iceberg catalog architecture{: .darkmode-img } *Iceberg catalog architecture*{: .caption }

A Database You Say?

But here's the problem: Iceberg and Delta Lake were specifically designed to *not* require a database. Their designers went to great lengths to encode all information needed to efficiently read and update tables into files on the blob store. They make many compromises to achieve this. For example, every single root file in Iceberg contains *all* existing snapshots complete with schema information, etc. For every single change, a new file is written that contains the complete history. A lot of other metadata had to be batched together, e.g., in the two-layer manifest files to avoid writing or reading too many small files, something that would not be efficient on blob stores. Making small changes to data is also a largely unsolved problem that requires complex cleanup procedures that are still not very well understood nor supported by open-source implementations. Entire companies exist and are still being started to solve this problem of managing fast-changing data. Almost as if a specialized data management system of sorts would be a good idea.

But as pointed out above, the Iceberg and Delta Lake designs *already* had to compromise and add a database as part of the catalog for consistency. However, they never revisited the rest of their design constraints and tech stack to adjust for this fundamental design change.

DuckLake

Here at DuckDB, we actually like databases. They are amazing tools to safely and efficiently manage fairly large datasets. Once a database has entered the Lakehouse stack anyway, it makes an insane amount of sense to also use it for managing the rest of the table metadata! We can still take advantage of the "endless" capacity and "infinite" scalability of blob stores for storing the actual table data in open formats like Parquet, but we can much more efficiently and effectively manage the metadata needed to support changes in a database! Coincidentally, this is also what Google BigQuery (with Spanner) and Snowflake (with FoundationDB) have chosen, just without the open formats at the bottom.

DuckLake's architecture{: .lightmode-img } DuckLake's architecture{: .darkmode-img } *DuckLake's architecture: Just a database and some Parquet files*{: .caption }

To resolve the fundamental problems of the existing Lakehouse architecture, we have created a new open table format called DuckLake. DuckLake re-imagines what a "Lakehouse" format should look like by acknowledging two simple truths:

1. Storing *data* files in open formats on blob storage is a great idea for scalability and to prevent lock-in.
2. Managing metadata is a complex and interconnected data management task best left to a database management system.

The basic design of DuckLake is to *move all metadata structures into a SQL database*, both for catalog and table data. The format is defined as a set of relational tables and pure-SQL transactions on them that describe data operations like schema creation, modification, and addition, deletion and updating of data. The DuckLake format can manage an *arbitrary number* of tables with cross-table transactions. It also supports "advanced" database concepts like views, nested types, transactional schema changes etc.; see below for a list. One major advantage of this design is by leveraging referential consistency (the "C" in ACID), the schema makes sure there are e.g. no duplicate snapshot ids.

DuckLake schema{: .lightmode-img } DuckLake schema{: .darkmode-img } *DuckLake schema*{: .caption }

Which exact SQL database to use is up to the user, the only requirements are that the system supports ACID operations and primary keys along with standard SQL support. The DuckLake-internal table schema is intentionally kept simple in order to maximize compatibility with different SQL databases. Here is the core schema through an example.

Let's follow the sequence of queries that occur in DuckLake when running the following query on a new, empty table:

```
INSERT INTO demo VALUES (42), (43);
```

```
BEGIN TRANSACTION;
  -- some metadata reads skipped here
  INSERT INTO ducklake_data_file VALUES (0, 1, 2, NULL, NULL, 'data_files/ducklake-8196...13a.parquet',
  'parquet', 2, 279, 164, 0, NULL, NULL);
  INSERT INTO ducklake_table_stats VALUES (1, 2, 2, 279);
  INSERT INTO ducklake_table_column_stats VALUES (1, 1, false, NULL, '42', '43');
  INSERT INTO ducklake_file_column_statistics VALUES (0, 1, 1, NULL, 2, 0, 56, '42', '43', NULL)
  INSERT INTO ducklake_snapshot VALUES (2, now(), 1, 2, 1);
  INSERT INTO ducklake_snapshot_changes VALUES (2, 'inserted_into_table:1');
COMMIT;
```

We see a single coherent SQL transaction that:

- Inserts the **new Parquet** file path
- Updates the global **table** statistics (now has more rows)
- Updates the global **column** statistics (now has a different minimum and maximum value)
- Updates the **file** column statistics (also record min/max among other things)
- Creates a new schema **snapshot** (#2)
- Logs the **changes** that happened in the snapshot

Note that the actual write to Parquet is not part of this sequence, it happens beforehand. But no matter how many values are added, this sequence has the same (low) cost.

Let's discuss the three principles of DuckLake: **Simplicity**, **Scalability** and **Speed**.

Simplicity

DuckLake follows the DuckDB design principles of keeping things *simple and incremental*. In order to run DuckLake on a laptop, it is enough to just install DuckDB with the `ducklake` extension. This is great for testing purposes, development and prototyping. In this case, the catalog store is just a local DuckDB file.

The next step is making use of *external storage systems*. DuckLake data files are immutable, it never requires modifying files in place or re-using file names. This allows use with almost any storage system. DuckLake supports integration with any storage system like local disk, local NAS, S3, Azure Blob Store, GCS, etc. The storage prefix for data files (e.g., `s3://mybucket/mylake/`) is specified when the metadata tables are created.

Finally, the SQL database that hosts the *catalog server* can be any halfway competent SQL database that supports ACID and primary key constraints. Most organizations will already have a lot of experience operating a system like that. This greatly simplifies deployment as no additional software stack is needed beyond the SQL database. Also, SQL databases have been heavily commoditized in recent years, there are innumerable hosted PostgreSQL services or even hosted DuckDB that can be used as the catalog store! Again, the lock-in here is very limited because transitioning does not require any table data movement, and the schema is simple and standardized.

There are no Avro or JSON files. There is no additional catalog server or additional API to integrate with. **It's all just SQL.** We all know SQL.

Scalability

DuckLake actually increases separation of concerns within a data architecture into *three parts*. Storage, compute and metadata management. Storage remains on purpose-built file storage (e.g., blob storage), DuckLake can scale infinitely in storage.

An arbitrary number of compute nodes are querying and updating the catalog database and then independently reading and writing from storage. DuckLake can scale infinitely regarding compute.

Finally, the catalog database needs to be able to run *only* the metadata transactions requested by the compute nodes. Their volume is several orders of magnitude smaller than the actual data changes. But DuckLake is not bound to a single catalog database, making it possible to migrate e.g. from PostgreSQL to something else as demand grows. In the end, DuckLake uses simple tables and basic, portable SQL. But don't worry, a PostgreSQL-backed DuckLake will already be able to scale to hundreds of terabytes and thousands of compute nodes.

Again, this is the exact design used by BigQuery and Snowflake that successfully manage immense datasets already. And hey, nothing keeps you from using Spanner as the DuckLake catalog database if required.

Speed

Just like DuckDB itself, DuckLake is very much about speed. One of the biggest pain points of Iceberg and Delta Lake is the involved sequence of file IO that is required to run the smallest query. Following the catalog and file metadata path requires many separate sequential HTTP requests. As a result, there is a lower bound to how fast reads or transactions can run. There is a lot of time spent in the critical path of

transaction commits, leading to frequent conflicts and expensive conflict resolution. While caching can be used to alleviate some of these problems, this adds additional complexity and is only effective for "hot" data.

The unified metadata within a SQL database also allows for low-latency query planning. In order to read from a DuckLake table, a single query is sent to the catalog database, which performs the schema-based, partition-based and statistics-based pruning to essentially retrieve a list of files to be read from blob storage. There are no multiple round trips to storage to retrieve and reconstruct metadata state. There is also less that can go wrong, no S3 throttling, no failing requests, no retries, no not-yet consistent views on storage that lead to files being invisible, etc.

DuckLake is also able to improve the two biggest performance problems of data lakes: small changes and many concurrent changes.

For small changes, DuckLake will *dramatically reduce the number of small files* written to storage. There is no new snapshot file with a tiny change compared to the previous one, there is no new manifest file or manifest list. DuckLake even optionally allows *transparent inlining* of small changes to tables into actual tables directly in the metadata store! Turns out, a database system can be used to manage data, too. This allows for sub-millisecond writes and for improved overall query performance by reducing the number of files that have to be read. By writing many fewer files, DuckLake also greatly simplifies cleanup and compaction operations.

In DuckLake, table changes consist of two steps: staging the data files (if any) to storage, and then running a single SQL transaction in the catalog database. This greatly reduces the time spent in the critical path of transaction commits, there is only a single transaction to run. SQL databases are pretty good at de-conflicting transactions. This means that the compute nodes spend a much smaller amount of time in the critical path where conflicts can occur. This allows for much faster conflict resolution and for many more concurrent transactions. Essentially, DuckLake supports as many table changes as the catalog database can commit. Even the venerable Postgres can run thousands of transactions *per second*. One could run a thousand compute nodes running appends to a table at a one-second interval and it would work fine.

In addition, DuckLake snapshots are just a few rows added to the metadata store, allowing for many snapshots to exist at the same time. There is no need to proactively prune snapshots. Snapshots can also refer to *parts of a Parquet file*, allowing many more snapshots to exist than there are files on disk. Combined, this allows DuckLake to manage *millions of snapshots!*

Features

DuckLake has all of your favorite Lakehouse features:

- **Arbitrary SQL:** DuckLake supports all the same vastness of SQL features that e.g. DuckDB supports.
- **Data Changes:** DuckLake supports efficient appends, updates and deletes to data.
- **Multi-Schema, Multi-Table:** DuckLake can manage an arbitrary number of schemas that each contain an arbitrary number of tables in the same metadata table structure.
- **Multi-Table Transactions:** DuckLake supports fully ACID-compliant transactions over all of the managed schemas, tables and their content.
- **Complex Types:** DuckLake supports all your favorite complex types like lists, arbitrarily nested.
- **Full Schema Evolution:** Table schemas can be changed arbitrarily, e.g., columns can be added, removed, or have their data types changed.
- **Schema-Level Time Travel and Rollback:** DuckLake supports full snapshot isolation and time travel, allowing to query tables as of a specific point in time.
- **Incremental Scans:** DuckLake supports retrieval of only the changes that occurred between specified snapshots.
- **SQL Views:** DuckLake supports the definition of lazily evaluated SQL-level views.
- **Hidden Partitioning and Pruning:** DuckLake is aware of data partitioning and table- and file-level statistics, allowing for early pruning of scans for maximum efficiency.
- **Transactional DDL:** Schema and table and view creation, evolution and removal are fully transactional.
- **Data Compaction Avoidance:** DuckLake requires far fewer compaction operations than comparable formats. DuckLake supports efficient compaction of snapshots.
- **Inlining:** When making small changes to the data, DuckLake can optionally use the catalog database to store those small changes directly to avoid writing many small files.
- **Encryption:** DuckLake can optionally encrypt all data files written to data store, allowing for *zero-trust data hosting*. Keys are managed by the catalog database.

- **Compatibility:** The data and (positional) deletion files that DuckLake writes to storage are fully compatible with Apache Iceberg allowing for metadata-only migrations.

The `ducklake` DuckDB Extension

Specifying data formats into thin air is easy, making things work is hard. This is why today we're also releasing a compute node implementation for DuckLake, in the form of the `ducklake` DuckDB extension. The extension implements the DuckLake format as described above and supports all the features described. The extension is free and open-source software under the MIT license with all IP resting in the non-profit DuckDB Foundation.

Conceptually, the `ducklake` extension massively elevates DuckDB from its single-node beginnings to where it can *support centralized client-server data warehouse use cases* without requiring extra supporting infrastructure. With DuckLake, an organization can set up a centralized catalog database and file storage (e.g., AWS RDS and S3 or self-hosted), and then run a DuckDB instance with the `ducklake` extension on a large number of participating devices, e.g., employee workstations, phones, application servers, or serverless compute code.

The extension is able to run DuckLake independently using a local DuckDB file as its catalog database. It is also able to use any third-party database that DuckDB can talk to. Currently this includes PostgreSQL, SQLite, MySQL and MotherDuck as external centralized catalog databases. The extension can make use of any file systems that DuckDB supports, currently local files, S3, Azure Blob Store, GCS, etc.

Of course, the availability of the DuckLake extension does augment, not replace DuckDB's existing and continuing support for Iceberg and Delta and the associated catalogs. DuckLake is also well-positioned to serve as a local cache or acceleration feature for those formats.

DuckLake is available starting from DuckDB release v1.3.0 (codename "Ossivalis").

Installation

- Step 1: Install DuckDB
- Step 2: Start DuckDB
- Step 3: Type `INSTALL ducklake;`

Checklist complete.

Usage

A DuckLake can be initialized through the ATTACH command in DuckDB. For example:

```
ATTACH 'ducklake:metadata.ducklake' AS my_ducklake;
```

This will create a new attached database `metadata.ducklake` in DuckDB and aliased as `my_ducklake`. In this fully local case, the metadata tables are stored in the `metadata.ducklake` file, and the Parquet files with the data are stored in the `metadata.ducklake.files` folder in the current working directory. Of course, absolute paths will work as well.

Let's next create a table and insert some data:

```
CREATE TABLE my_ducklake.demo (i INTEGER);
INSERT INTO my_ducklake.demo VALUES (42), (43);
```

> You can switch the default database with the USE command, e.g., `USE my_ducklake`.

Let's query the table again:

```
FROM my_ducklake.demo;
```

i
int32
42
43

So far, so good.

```
FROM glob('metadata.ducklake.files/*');
```

file
varchar
metadata.ducklake.files/ducklake-019711dd-6f55-7f41-ab99-6ac7d9de6ef3.parquet

We can see that a single Parquet file has been created that contains the two rows. Now, let's delete a row again:

```
DELETE FROM my_ducklake.demo WHERE i = 43;
FROM my_ducklake.demo;
```

i
int32
42

We can see the row being gone. If we inspect the folder again, we see a new file appearing

```
FROM glob('metadata.ducklake.files/*');
```

file
varchar
metadata.ducklake.files/ducklake-019711dd-6f55-7f41-ab99-6ac7d9de6ef3.parquet
metadata.ducklake.files/ducklake-019711e0-16f7-7261-9d08-563a48529955-delete.parquet

A second file with `-delete` in the name has appeared, this is also a Parquet file that contains the identifiers of the deleted rows.

Of course, DuckLake supports *time travel,* and we can query the available snapshots using the `ducklake_snapshots()` function.

```
FROM ducklake_snapshots('my_ducklake');
```

snapshot_id	snapshot_time	schema_version	changes
int64	timestamp with time zone	int64	map(varchar, varchar[])
0	2025-05-27 15:10:04.953+02	0	{schemas_created=[main]}
1	2025-05-27 15:10:14.079+02	1	{tables_created=[main.demo]}
2	2025-05-27 15:10:14.092+02	1	{tables_inserted_into=[1]}
3	2025-05-27 15:13:08.08+02	1	{tables_deleted_from=[1]}

Let's say we want to read the table before the row with 43 in it was deleted, we can use the new AT syntax in DuckDB:

```
FROM my_ducklake.demo AT (VERSION => 2);
```

i
int32
42
43

Version 2 still had this row, so there it is. This also works with snapshot timestamps instead of version numbers: just use `TIMESTAMP` instead of `VERSION`.

We can also look at what has changed between versions using the `ducklake_table_changes()` function, for example

```
FROM ducklake_table_changes('my_ducklake', 'main', 'demo', 2, 3);
```

snapshot_id	rowid	change_type	i
int64	int64	varchar	int32
2	0	insert	42
2	1	insert	43
3	1	delete	43

We can see how in version two we added the two values 42 and 43 in snapshot 2 and then deleted 43 in snapshot 3 again.

Changes in DuckLake are of course transactional, previously we were running in "autocommit" mode, where every command is its own transaction. But we can change that using `BEGIN TRANSACTION` and `COMMIT` or `ROLLBACK`.

```
BEGIN TRANSACTION;
DELETE FROM my_ducklake.demo;
FROM my_ducklake.demo;
```

i
int32
0 rows

```
ROLLBACK;
FROM my_ducklake.demo;
```

i
int32
42

Here, we start a transaction, and then delete all the rows from `my_table`, which then indeed shows as empty. However, because we then `ROLLBACK`, the non-committed delete is reverted.

Summary In this post, we presented the rationale and the design of a new lakehouse format built on pure SQL. If you would like to use data in a simple, scalable and fast lakehouse format, give it a try! We are looking forward to seeing your use cases!

> The `ducklake` extension is currently experimental. If you encounter any bugs, please submit an issue in the `ducklake` extension repository.

Press Inquiries

For press inquiries, please reach out to Gabor Szarnyas.

Faster Dashboards with Multi-Column Approximate Sorting

Publication date: 2025-06-06

Author: Alex Monahan

TL;DR: With any columnar data format, using advanced multi-column sorting when loading data can improve performance on a wide variety of selective read queries. Space filling curve encodings like Morton (Z-Order) and Hilbert approximately sort multiple columns together. Sorting by rounded timestamps adds additional benefits when filtering on recent data.

An animated Hilbert space filling curve by TimSauder – Own work, CC BY-SA 4.0

It is rare to have a dashboard with a single chart. It is even more rare to only query your data one way. It is rarer still to query your data with no filters at all!

When queries read a subset of the total rows, sorting data while loading provides significant benefits. Please have a look at the prior post in this series to understand how this approach works and for practical tips!

In order to see these benefits in a wider variety of real world cases, we have to get a bit more creative. These advanced techniques help when one of these situations occur:

- Queries filter on different columns
- Query patterns are not perfectly predictable
- Queries filter by time and at least one other column

In this post, we describe several advanced sorting strategies, compare them with some experiments (microbenchmarks), and then calculate a metric to measure their effectiveness.

The Strategy

Instead of sorting precisely by one or a small number of columns, we want to sort approximately by a larger number of columns. That will allow queries with different `WHERE` clauses to all benefit from DuckDB's min-max indexes (zone maps). This post introduces two high-level approaches with several examples of each: sorting by space filling curves, and sorting by truncated timestamps.

Space Filling Curves

Both Morton and Hilbert are space filling curve algorithms that are designed to combine multiple columns into an order that preserves some approximate ordering for both columns. One application of space filling curves is in geospatial analytics and it is a helpful illustration.

If a dataset contained the latitude and longitude coordinates of every café on earth (one row per café), but we wanted to sort so that cafés that are physically close to one another are near each other in the list, we could use a space filling curve. Cafés that are somewhat close in both latitude and longitude will receive a similar Morton or Hilbert encoding value. This will allow us to quickly execute queries like "Find all cafés within this rectangular region on a map". (A rectangular region like that is called a bounding box in geospatial-land!) The GIF at the top of this post shows various levels of granularity of a Hilbert encoding – imagine if the x axis were longitude and the y axis were latitude. The "zig-zag" line of the Hilbert algorithm is the list of cafés, sorted approximately.

Both Morton and Hilbert operate on integers or floats, but this post outlines a way to use them to sort VARCHAR columns as well. A SQL macro is used to convert the first few characters of a VARCHAR column into an integer as a pre-processing step.

Truncated Timestamps

More recent data tends to be more useful data, so frequently queries filter on a time column. However, often queries filter on time and on other columns as well.

Don't just sort on the timestamp column! You will miss out on performance benefits. This is because timestamps tend to be so granular, that in practical terms the data is only sorted by timestamp. How many rows of your data were inserted at exactly `2025-01-01 01:02:03.456789`? Probably just one!

To sort on multiple columns as well as a time column, first sort by a truncated timestamp (truncated to the start of the day, week, year, etc.) and then on the other columns.

The "On-Time Flights" Dataset

Ever wondered how likely you will be delayed when you are getting ready to fly? If you are traveling in the US, you can use a government dataset to see how on-time each route is! You can get more details on the dataset at its official site, but I grabbed a Parquet version of the

data from Kaggle. It includes almost 5 years of data on US flights. Combined, the Parquet files are about 1.1 GB. This is fairly small when accessed locally but can be more of a challenge over a slow remote connection. In the rest of the post, we'll run different benchmarks to demonstrate how pruning can help reducing the amount of network traffic required to answer certain queries.

Our first experiment is a hypothetical use case to serve a dashboard where people can explore a single `origin`, a single `dest` (destination), or an `origin-dest` pair. The `origin` and `dest` columns are three-letter airport codes (like `LAX` or `JFK`). We assume once you have picked one of those filters, you want all the rows that match and a few arbitrary columns.

Our second experiment will add in a time component where our hypothetical users could additionally filter to the latest 1, 13, or 52 weeks of data in the dataset.

Experimental Design

We will show a few approaches to sort data approximately to speed up retrieval.

The control groups are:

- `random`: Sort by a hash – the worst case scenario!
- `origin`: Single-column sort on `origin`
- `origin-dest`: Sort on `origin`, then on `destination`

Our alternative sorting approaches are:

- `zipped_varchar`: Sort by one letter at a time, alternating between `origin` and `destination`. – For example, an `origin` of ABC and a `destination` of XYZ would become AXBYCZ, which would then be sorted.
- `morton`: Convert `origin` and `destination` into integers, then order by Morton encoding (Z-order)
- `hilbert`: Convert `origin` and `destination` into integers, then order by Hilbert encoding

> The `zipped_varchar` algorithm is implemented using a SQL macro so no extensions are needed. It also handles arbitrary length strings.

> The Morton and Hilbert encoding functions come from the `lindel` DuckDB community extension, contributed by Rusty Conover. Thank you to Rusty and the folks who have built the `lindel` Rust crate upon which the DuckDB extension is based! The `spatial` extension also contains an `ST_Hilbert` function that works similarly. Thanks to Max Gabrielsson and the GDAL community!

These plots display query runtime when pulling from a DuckDB file hosted on S3. These same techniques can also be successfully applied to the DuckLake integrated data lake and catalog format! DuckLake is the modern evolution of the cloud data lakehouse – take a minute to check out the launch post if you haven't yet! DuckLake has an additional concept of a partition, which enables entire files to be skipped. To take full advantage of DuckLake, first partition your data (by time or otherwise) and then apply the techniques in this post when loading your data.

All experiments were run on an M1 MacBook Pro with DuckDB v1.2.2. The tests were conducted with the DuckDB Python client with results returned as Apache Arrow tables. In between each query, the DuckDB connection is closed and recreated (this time is not measured as a part of the results). This is to better simulate a single user's experience accessing the dashboard in our hypothetical use case.

Experimental Results

We can see that when filtering **by `origin`**, the `origin` single column sort is a full order of magnitude faster than randomly distributed data, requiring only 1.6 seconds instead of 16. Our advanced techniques are nearly as fast as the dedicated sort on `origin`.

However, if you look at the plot **by `destination`**, we see the value of the more balanced techniques. The `origin_dest` approach of just appending `destination` to the list of sorted columns misses out on much of the benefits.

When querying for a **specific `origin` and `destination` pair,** all approaches are significantly faster than random. However, zooming in on the non-random techniques, the more advanced sorting approaches are as fast or slightly faster than sorting by `origin` or by `origin`

and then `destination`. Performance is also faster than the other experiments, as less data needs to be read (since the filter is more selective).

The Hilbert encoding method provides the most consistent performance across the three query patterns. It would be the best choice to support all 3 workloads.

Approximate Time Sorting

Sorting by an "approximate time" involves truncating the time to the nearest value of a certain time granularity. In this experiment, we load the data with 3 different sorting approaches. We first sort on a truncation of the `flightdate` column, truncated to either day, month, or year. We then use our most effective multi-column approach and sort by the Hilbert encoding of `origin` and `dest` next.

When measuring performance, we test each of the query patterns tested previously (filters on `origin`, `destination`, and `origin` / `destination`), but additionally filter on three different time ranges: the latest 1 week, 13 weeks, and 52 weeks.

When querying a week of data for a specific `origin`, sorting at the daily level performs the best. However, sorting by a more approximate time (month or year) performs better when analyzing the latest 13 or 52 weeks of data. This is because the more approximate time buckets allow the Hilbert encoding to separate `origin`s into different row groups more effectively.

Querying by time and destination follows a very similar pattern, with the ideal sort order being highly dependent on how far back in time is analyzed.

Filtering by `origin` and `destination` shows a very different outcome, with a yearly granularity being superior across the board! This is because the `origin` and `destination` filters are much more effective at skipping row groups when the timestamp ordering is very approximate.

As a result, the best compromise across those three workloads is likely to be the very approximate yearly granularity. See! Don't just sort by timestamp!

Table Creation Time

The upfront investment required to see these benefits is the time required to sort the data when inserting. Typically this is still a good trade off – data inserts tend to happen in the background, but nobody wants to be staring at dashboard loading spinners!

Creating a DuckDB table from Parquet files without sorting took slightly over 21 seconds. Each other approach copied from the unsorted DuckDB table and created a new table. The various methods of sorting required similar amounts of time (between 48 and 61 seconds), so we are free to choose the one with the most effective results without considering relative insert performance. However, it is worth noting that overall insert performance slows down by nearly 3× with any sorting.

Table name	Creation time (s)
`from_parquet`	21.4
`random`	60.2
`origin`	51.9
`origin_dest`	48.7
`zipped_varchar`	58.2
`morton`	54.6
`hilbert`	58.5
`hilbert_day`	58.7
`hilbert_month`	53.8
`hilbert_year`	60.2

Measuring Sortedness

The most effective way to choose a sort order is to simulate your production workload, like in the experiments above. However, this is not always feasible or easy to do. Instead, we can measure how well sorted the dataset is on the columns of interest. The metric we will use is "Number of Row Groups per Value". The way to interpret this is that for selective queries to work effectively, each value being filtered on should only be present in a small number of row groups. Smaller is better! However, there are likely diminishing returns when this metric is below the number of threads DuckDB is using.

> Other database systems measure a similar metric called "Clustering Depth".

What can be interpreted from this graph? The random ordering spreads nearly every value across 100 or more row groups (the visualization is truncated at 100 row groups for readability). This highlights how a random ordering will be very slow for selective queries! Sorting by `origin` greatly reduces the number of row groups that each `origin` is spread across, but `destination`s are still spread widely. Sorting by `origin` and `destination` retains the tight distribution of `origin`s and slightly improves the metric for `destination`s.

The three advanced approaches (`zipped_varchar`, `morton`, and `hilbert`) are more balanced, with both `origin`s and `destination`s only occuring in a moderate number of row groups. While they score worse in the `origin` metric than when sorting directly by `origin`, most `origin`s are spread across fewer row groups than a modern laptop processor's core count, so high performance is retained. The Hilbert encoding is the most balanced, so by this metric it would be declared the victor as well!

To calculate this metric, we define several SQL macros using dynamic SQL and the `query` table function.

We can then call the `rowgroup_counts` function on any table and any columns!

```
FROM rowgroup_counts('flights_hilbert', ['origin', 'dest']);
```

table_name	column_name	column_value	total_row_groups	rowgroup_id_count
flights_hilbert	dest	PSG	238	2
flights_hilbert	dest	ESC	238	2
flights_hilbert	origin	YUM	238	2
…	…	…	…	…

The `rowgroup_id_count` column is a measurement of how many distinct row groups that a specific column value is present in, so it is an indicator of how much work DuckDB would need to do to pull all data associated with that value.

> This calculation uses the pseudo-column `rowid`, and it requires data to have been inserted in a single batch to be perfectly accurate. It is only an approximate metric when data is inserted in batches.

Conclusion

When storing data in a columnar file format like a DuckDB database or a Parquet file, approximately sorting by multiple columns can lead to fast read queries across a variety of query patterns. Sorting using the Hilbert encoding provided high performance across multiple workloads, and sorting by year and then by Hilbert performed well when also filtering by time.

Thanks to the "Number of Row Groups per Value" calculation, we can measure the sortedness of any table by any column, and it was predictive of the experimental performance we observed. This way we can experiment with different sorting approaches and quickly forecast their effectiveness, without having to benchmark read workloads each time.

Using these approaches can greatly speed up dashboard interactivity when users are unpredictable (which we always are!). They can also be used in combination with the the additional techniques outlined in the prior post for further benefits. Data is always different, so we get to be creative!

Happy analyzing, folks!

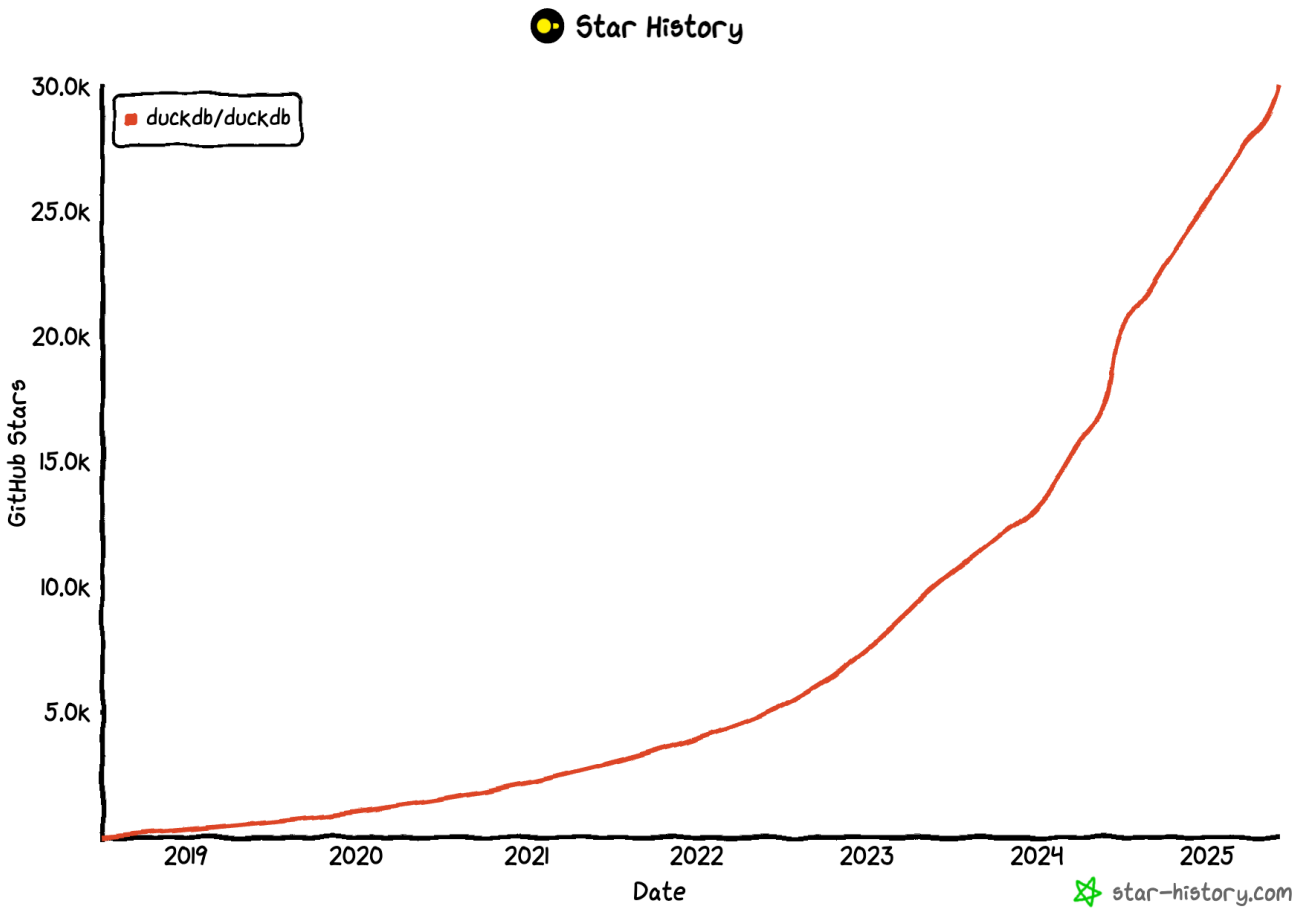
30 000 Stars on GitHub

Publication date: 2025-06-06

Author: The DuckDB team

TL;DR: We have reached 30 000 stars on GitHub! Here's what happened while we accumulated the last 5 000 stars.

The duckdb/duckdb GitHub repository has just passed 30 000 stars!



We would like to use this event to stop for a moment and revisit recent developments in the DuckDB ecosystem since last December, not even six months ago, when we reached 25 000 stars.

Recent Updates

- We released DuckDB 1.2.0 and DuckDB 1.3.0, both packing many new features, performance optimizations and bugfixes.
- We published the DuckLake specification, an open format for SQL-as-a-lakehouse. For details, read the announcement blog post and its coverage in the Register.
- DuckDB now has a built-in UI.

- DuckDB now supports Iceberg REST Catalog, which also allows it to connect to Amazon S3 Tables
- We published the much-requested roadmap.
- Coincidentally, this is the 100th post on our blog.

Metrics

Besides the GitHub stars we have also observed a lot of growth in other metrics:

- Each month, our website now welcomes over 3 million unique visitors – more than double than in December. We also see over 700 TB in traffic from millions of extension downloads. Thanks again to Cloudflare for sponsoring the project.
- Just in this calendar year, we rose 10 positions in the DB Engines ranking from position 55 to 45.
- We now count 20M+ monthly downloads in PyPI.

We would like to reiterate that while we're happy to see these numbers grow, we are not glorifying them and they are not a target per se in accordance with Goodhart's law.

Events

We hosted the first livestreamed DuckCon in Amsterdam and organized another DuckDB Meetup.

Closing Thoughts

That's it for today. We would like to thank our community of users and contributors! We are looking forward to keep growing our community and hope the GitHub stars follow. We'll report back when we reach 40,000 stars.

Lightweight Text Analytics Workflows with DuckDB

Publication date: 2025-06-13

Author: Petrica Leuca

TL;DR: In this post, we demonstrate how to use DuckDB for keyword, full-text, and semantic similarity search with embeddings.

Introduction

Text analytics is a central component of many modern data workflows, covering tasks such as keyword matching, full-text search, and semantic comparison. Conventional tools frequently require complex pipelines and substantial infrastructure, which can pose significant challenges. DuckDB offers a high-performance SQL engine that simplifies and streamlines text analytics. In this post, we will demonstrate how to leverage DuckDB to efficiently perform advanced text analytics in Python.

> The following implementation is executed in a marimo Python notebook, which is available on GitHub, in our examples repository.

Data Preparation

We will be working with a public dataset, available on Hugging Face, which contains English Twitter messages and their classification to one of the following emotions: anger, fear, joy, love, sadness, and surprise.

With DuckDB we are able to access Hugging Face datasets via the `hf://` prefix:

```
from_hf_rel = conn.read_parquet(
        "hf://datasets/dair-ai/emotion/unsplit/train-00000-of-00001.parquet",
        file_row_number=True
    )
from_hf_rel = from_hf_rel.select("""
    text,
    label as emotion_id,
    file_row_number as text_id
""")
from_hf_rel.to_table("text_emotions")
```

> How to access Hugging Face datasets with DuckDB is detailed in the post "Access 150k+ Datasets from Hugging Face with DuckDB".

In the above data we have available only the identifier of an emotion (`emotion_id`), without its descriptive information. Therefore, from the list provided in the dataset description, we create a reference table by unnesting the Python list and retrieving the index for each value with the `generate_subscripts` function:

```
emotion_labels = ["sadness", "joy", "love", "anger", "fear", "surprise"]

from_labels_rel = conn.values([emotion_labels])
from_labels_rel = from_labels_rel.select("""
    unnest(col0) as emotion,
    generate_subscripts(col0, 1) - 1 as emotion_id
""")
from_labels_rel.to_table("emotion_ref")
```

Last we define a relation, by joining the two tables:

```
text_emotions_rel = conn.table("text_emotions").join(
    conn.table("emotion_ref"), condition="emotion_id"
)
```

> By executing `text_emotions_rel.to_view("text_emotions_v", replace=True)` a view with the name `text_emotions_v` will be created, which can be used in SQL cells.

We plot on a bar chart the emotion distribution to have an initial understanding of our data:

Keyword Search

Keyword search is the most basic form of text retrieval, matching exact words or phrases in text fields using SQL conditions such as `CONTAINS`, `ILIKE`, or other DuckDB text functions. It is fast, requires no preprocessing, and works well for structured queries like filtering logs, matching tags, or finding product names.

For example, getting the texts and their emotion label containing the phrase `excited to learn` is a matter of applying `filter` on the relation defined above:

```
text_emotions_rel.filter("text ilike '%excited to learn%'").select("""
    emotion,
    substring(
        text,
        position('excited to learn' in text),
        len('excited to learn')
    ) as substring_text
""")
```

emotion varchar	substring_text varchar
sadness	excited to learn
joy	excited to learn
joy	excited to learn
fear	excited to learn
fear	excited to learn
joy	excited to learn
joy	excited to learn
sadness	excited to learn

One common step in text processing is to split text into tokens (keywords), where raw text is broken down into smaller units (typically words), that can be analyzed or indexed. This process, known as *tokenization*, helps convert unstructured text into a structured form suitable for keyword search. In DuckDB this process can be implemented with the `regexp_split_to_table` function, which will split the text based on the provided regex and return each keyword on a row.

> This step is case sensitive, therefore it is important to convert all text to a consistent case (by applying `lcase` or `ucase`) before processing.

In the below code snippet we select all the keywords by splitting the text on one or more non-word characters (anything except `[a-zA-Z0-9_]`):

```
text_emotions_tokenized_rel = text_emotions_rel.select("""
    text_id,
    emotion,
    regexp_split_to_table(text, '\\W+') as token
""")
```

In the tokenization step, we usually exclude common words (such as `and`, `the`), called *stopwords*. In DuckDB we implement the exclusion by applying an `ANTI JOIN` on a curated CSV file hosted on GitHub:

```
english_stopwords_rel = duckdb_conn.read_csv(
    "https://raw.githubusercontent.com/stopwords-iso/stopwords-en/refs/heads/master/stopwords-en.txt",
    header=False,
).select("column0 as token")

text_emotions_tokenized_rel.join(
    english_stopwords_rel,
    condition="token",
    how="anti",
).to_table("text_emotion_tokens")
```

Now that we have tokenized and cleaned the text, we can implement keyword search by ranking the match with similarity functions, such as Jaccard:

```
text_token_rel = conn.table(
    "text_emotion_tokens"
).select("token, emotion, jaccard(token, 'learn') as jaccard_score")

text_token_rel = text_token_rel.max(
    "jaccard_score",
    groups="emotion, token",
    projected_columns="emotion, token"
)

text_token_rel.order("3 desc").limit(10)
```

emotion varchar	token varchar	max(jaccard_score) double
fear	learn	1.0
surprise	learn	1.0
love	learn	1.0
joy	lerna	1.0
sadness	learn	1.0
fear	learner	1.0
anger	learn	1.0
joy	leaner	1.0
fear	allaner	1.0
anger	learner	1.0
10 rows		3 columns

We can also visualize the data to gain insights. One simple and effective approach is to plot the most frequently used words. By counting token occurrences across the dataset and displaying them in bubble plots, we can quickly identify dominant themes, repeated keywords, or unusual patterns in the text. For example, we plot the data by using a scatter facet plot per emotion:

From the above plot, we observe repeated keywords, such as `feel - feeling`, `love - loved - loving`. In order to de-duplicate such data, we need to look at the word stem rather than at the word itself. This brings us to full-text search.

Full-Text Search

The Full-Text Search (FTS) DuckDB extension is an experimental extension, which implements two main full-text search functionalities:

- the `stem` function, to retrieve the word stem;

- the `match_bm25` function, to calculate the Best Match score.

By applying `stem` to the token column, we can now visualize the most frequently used word stem in our data:

We observe that `feel` and `love` appear only once and new word stems are plotted, such as `support`, `surpris`.

While the `stem` function can be used standalone, the `match_bm25` one requires the build of a FTS index, a special index that allows fast and efficient searching of text by indexing the words (tokens) in a column:

```
conn.sql("""
    PRAGMA create_fts_index(
        "text_emotions",
        text_id,
        "text",
        stemmer = 'english',
        stopwords = 'english_stopwords',
        ignore = '(\\.|[^a-z])+',
        strip_accents = 1,
        lower = 1,
        overwrite = 1
    )
""")
```

In the FTS index creation we are using the same list of English stopwords as in the tokenization process, by saving it into a table, named `english_stopwords`. The index is case insensitive due to the `lower` parameter, which will lowercase the text automatically.

> **Warning.** The index can be created only on tables and it requires a unique identifier of the text. It also needs to be rebuild when the underlying data has been modified.

Once the index has been created, we can rank the match between the `text` column and the phrase `excited to learn`:

```
text_emotions_rel
.select("""
    emotion,
    text,
    emotion_color,
    fts_main_text_emotions.match_bm25(
        text_id,
        'excited to learn'
    )::decimal(3, 2) as bm25_score
""")
.order("bm25_score desc")
.limit(10)
```

Out of the 10 returned texts, displayed above in a table plot, 2 are poor matches to our search input; likely due to BM25 scoring being skewed by common terms or differences in document length.

Semantic Search

Compared to keyword and full-text search, *semantic search* takes into account the meaning and context of the text. Instead of just looking for exact words, it uses techniques like vector embeddings to capture the underlying concepts. Semantic search, which is case insensitive, can be implemented in DuckDB, by making use of the (also experimental) Vector Similarity Search extension.

The vector embeddings of a (list of) text can be calculated with the `sentence-transformers` library and the pre-trained model `all-MiniLM-L6-v2`:

```
from sentence_transformers import SentenceTransformer

model = SentenceTransformer('all-MiniLM-L6-v2')
```

```
def get_text_embedding_list(list_text: list[str]):
    """
    Return the list of normalized vector embeddings for list_text.
    """
    return model.encode(list_text, normalize_embeddings=True)
```

For example, `get_text_embedding_list(['excited to learn'])` will return:

```
array([[ 3.14795598e-02, -6.66208193e-02,  1.05058309e-02,
         4.12571728e-02, -8.67664907e-03, -1.79746319e-02,
        ...
        -2.50727013e-02, -3.00881546e-03,  1.55055271e-02]], dtype=float32)
```

We register the model inference function as a Python User Defined Function and create a table with a column of type `FLOAT[384]` to load the embeddings into:

```
conn.create_function(
    "get_text_embedding_list",
    get_text_embedding_list,
    return_type='FLOAT[384][]'
)

conn.sql("""
    create table text_emotion_embeddings (
        text_id integer,
        text_embedding FLOAT[384]
    )
""")
```

With the Python UDF, we save, in batches, the model output in `text_emotion_embeddings`:

```
for i in range(num_batches):
    selection_query = (
        duckdb_conn.table("text_emotions")
        .order("text_id")
        .limit(batch_size, offset=batch_size*i)
        .select("*")
    )

    (
        selection_query.aggregate("""
            array_agg(text) as text_list,
            array_agg(text_id) as id_list,
            get_text_embedding_list(text_list) as text_emb_list
        """).select("""
            unnest(id_list) as text_id,
            unnest(text_emb_list) as text_embedding
        """)
    ).insert_into("text_emotion_embeddings")
```

> We wrote about model inference in DuckDB in the post Machine Learning Prototyping with DuckDB and scikit-learn.

We can now perform semantic search by using the cosine distance between the vector embedding of our search text, `excited to learn`, and the embedding of the `text` field:

```
input_text_emb_rel = conn.sql("""
    select get_text_embedding_list(['excited to learn'])[1] as input_text_embedding
""")

text_emotions_rel
.join(conn.table("text_emotion_embeddings"), condition="text_id")
```

```
.join(input_text_emb_rel, condition="1=1")
.select("""
        text,
        emotion,
        emotion_color,
        array_cosine_distance(
            text_embedding,
            input_text_embedding
        )::decimal(3, 2) as cosine_distance_score
    """)
.order("cosine_distance_score asc")
.limit(10)
```

Interesting to observe that the phrase `i am excited to learn and feel privileged to be here` didn't make it in the top 10 results in our semantic search!

Similarity Joins

Vector embeddings are most known for their usability in search engines, but they can be used in a variety of text analytics use cases, such as topic grouping, classification or semantic matching between documents. The VSS extension provides vector similarity joins, which can be used to conduct these types of analytics.

For example, we show in the below heatmap chart the number of texts for each combination of emotion labels, where the x-axis corresponds to the semantic matching between the text and the emotion, the y-axis to the classified emotion, and the color indicates the count of texts assigned to each pair:

It is particularly noticeable that from the 6 emotions, only `sadness` has a strong semantic match with the text classified with the same label. As with full-text search, semantic search is affected by differences in document length (in this case the emotion keyword versus a text).

Hybrid Search

While each type of search has its own applicability, we observed that some results are not as expected:

- keyword search and full-text search do not take word meaning into account;
- semantic search scored synonyms higher than the search text.

In practice, the three search methods are combined and used to perform a "hybrid search", in order to improve the search relevance and accuracy. We start by calculating the score for each type of search, by implementing custom logic, such as a check on the emotion:

```
if(
    emotion = 'joy' and contains(text, 'excited to learn'),
    1,
    0
) exact_match_score,

fts_main_text_emotions.match_bm25(
        text_id,
        'excited to learn'
)::decimal(3, 2) as bm25_score,

array_cosine_similarity(
    text_embedding,
    input_text_embedding
)::decimal(3, 2) as cosine_similarity_score
```

The BM25 score is ranked in descending order and the cosine distance in ascending order. In hybrid search, we use the `array_cosine_similarity` score to ensure the same sort order (in this case descending).

> cosine similarity = 1 - cosine distance

Because the BM25 score can be, in theory, unbounded, we need to scale the score to the interval `[0, 1]` by implementing the min-max normalization:

```
max(bm25_score) over () as max_bm25_score,
min(bm25_score) over () as min_bm25_score,
(bm25_score - min_bm25_score) / nullif((max_bm25_score - min_bm25_score), 0) as norm_bm25_score
```

The hybrid search score is calculated by applying a weight to the BM25 and cosine similarity scores:

```
if(
    exact_match_score = 1,
    exact_match_score,
    cast(
        0.3 * coalesce(norm_bm25_score, 0) +
        0.7 * coalesce(cosine_similarity_score, 0)
        as
        decimal(3, 2)
    )
) as hybrid_score
```

And here are the results! Much better, don't you think?

Conclusion

In this post, we showed how DuckDB can be used for text analytics by combining keyword, full-text, and semantic search techniques. Using the experimental `fts` and `vss` extensions with the `sentence-transformers` library, we demonstrated how DuckDB can support both traditional and modern text analytics workflows.

Discovering DuckDB Use Cases via GitHub

Publication date: 2025-06-27

Author: Petrica Leuca

TL;DR: In this post, we use the GitHub API to find repositories that mention DuckDB, then use DuckDB itself to parse and query the results efficiently with SQL.

Introduction

One of the persistent challenges of maintaining an open-source library is the lack of visibility into how, where, and by whom it's being used. At DuckDB we look at the awesome-duckdb repository, which contains a curated list of DuckDB libraries, tools and resources. To supplement this, we wanted to automate insights from GitHub in order to uncover new and emerging projects using DuckDB. In this post we write about how we use DuckDB to query the GitHub API, export the data into a Markdown file and visualize the historical data, generated from Git commits.

Data Retrieval from GitHub

GitHub provides a series of REST API which can be used to retrieve data from GitHub. Among them, the search repositories API returns a list of public repositories whose name, description or readme matches the search text.

> GitHub also has a search code API in order to match on code dependency, but it does not return all the information the search repository provides. One could use the search code API and for each result to retrieve the repository information from the repository API.

In order to use the GitHub API, we create an access token with read access on public repositories. With the access token, saved as an environment variable, we create a secret in DuckDB to be used by the API request:

```
import os

import duckdb


def get_duckdb_conn():
    conn = duckdb.connect()
    conn.sql(f"""
        CREATE SECRET http_auth (
            TYPE http,
            BEARER_TOKEN '{os.getenv("READ_PUBLIC_REPO_TOKEN")}',
            SCOPE 'https://api.github.com/search'
        );
    """)
    return conn
```

With the DuckDB Python client's `read_json` function we are able to query the API response with SQL. The above secret is passed automatically by DuckDB to the API call:

```
duckdb_conn = get_duckdb_conn()

api_response = duckdb_conn.read_json(
    "https://api.github.com/search/repositories?q=duckdb"
)
```

The API call returns the following response:

total_count int64	incomplete_results boolean	items struct(id bigint, node_id varchar,...
3837	false	[{'id': 138754790, 'node_id': ...

We can analyze the response by:

- getting the list of columns with `api_response.columns`: `['total_count', 'incomplete_results', 'items']`
- getting the column types `api_response.types`: `[BIGINT, BOOLEAN, STRUCT(id BIGINT, node_id VARCHAR, "name" VARCHAR, full_name VARCHAR, private BOOLEAN, "owner" STRUCT(login VARCHAR, id BIGINT, node_id VARCHAR, avatar_url VARCHAR, gravatar_id VARCHAR, url VARCHAR, html_url VARCHAR, followers_url VARCHAR, following_url VARCHAR, gists_url VARCHAR, starred_url VARCHAR, subscriptions_url VARCHAR, organizations_url VARCHAR, repos_url VARCHAR, events_url VARCHAR, received_events_url VARCHAR, "type" VARCHAR, user_view_type VARCHAR, site_admin BOOLEAN), html_url VARCHAR, description VARCHAR, fork BOOLEAN, url VARCHAR, forks_url VARCHAR, keys_url VARCHAR, collaborators_url VARCHAR, teams_url VARCHAR, hooks_url VARCHAR, issue_events_url VARCHAR, events_url VARCHAR, assignees_url VARCHAR, branches_url VARCHAR, tags_url VARCHAR, blobs_url VARCHAR, git_tags_url VARCHAR, git_refs_url VARCHAR, trees_url VARCHAR, statuses_url VARCHAR, languages_url VARCHAR, stargazers_url VARCHAR, contributors_url VARCHAR, subscribers_url VARCHAR, subscription_url VARCHAR, commits_url VARCHAR, git_commits_url VARCHAR, comments_url VARCHAR, issue_comment_url VARCHAR, contents_url VARCHAR, compare_url VARCHAR, merges_url VARCHAR, archive_url VARCHAR, downloads_url VARCHAR, issues_url VARCHAR, pulls_url VARCHAR, milestones_url VARCHAR, notifications_url VARCHAR, labels_url VARCHAR, releases_url VARCHAR, deployments_url VARCHAR, created_at TIMESTAMP, updated_at TIMESTAMP, pushed_at TIMESTAMP, git_url VARCHAR, ssh_url VARCHAR, clone_url VARCHAR, svn_url VARCHAR, homepage VARCHAR, size BIGINT, stargazers_count BIGINT, watchers_count BIGINT, "language" VARCHAR, has_issues BOOLEAN, has_projects BOOLEAN, has_downloads BOOLEAN, has_wiki BOOLEAN, has_pages BOOLEAN, has_discussions BOOLEAN, forks_count BIGINT, mirror_url JSON, archived BOOLEAN, disabled BOOLEAN, open_issues_count BIGINT, license STRUCT("key" VARCHAR, "name" VARCHAR, spdx_id VARCHAR, url VARCHAR, node_id VARCHAR), allow_forking BOOLEAN, is_template BOOLEAN, web_commit_signoff_required BOOLEAN, topics VARCHAR[], visibility VARCHAR, forks BIGINT, open_issues BIGINT, watchers BIGINT, default_branch VARCHAR, permissions STRUCT("admin" BOOLEAN, maintain BOOLEAN, push BOOLEAN, triage BOOLEAN, pull BOOLEAN), score DOUBLE)[]]`
- getting the number of items with `api_response.select("len(items)")`:

```
┌──────────────┐
│ len(items)   │
│    int64     │
├──────────────┤
│           30 │
└──────────────┘
```

The total number of repositories mentioning DuckDB is 3837, but the `items` object contains only 30. This is due to pagination that is set to 30 records per page by default. We can retrieve a maximum of 100 records by using the `per_page` query parameter. Additionaly, we use the `page` query parameter in order to retrieve the results for each page:

```
(
    duckdb_conn
    .read_json("https://api.github.com/search/repositories?q=duckdb&per_page=100&page=2")
    .select("len(items)")
)
```

```
┌──────────────┐
│ len(items)   │
│   int64      │
├──────────────┤
│          100 │
└──────────────┘
```

Because GitHub has a limit of returning maximum 1000 results in the search API and we retrieve the data daily, we added a filter to retrieve only the repositories which had a push in the last 7 days:

```
api_url = "https://api.github.com/search/repositories?q=duckdb"

last_pushed_date = (datetime.now() - timedelta(days=7)).strftime("%Y-%m-%d")

api_url = f"{api_url}+pushed:>={last_pushed_date}"

duckdb_conn.read_json(f"{api_url}&per_page=100&page=1").to_table("github_raw_data")
```

We store the first page in a table called `github_raw_data` and calculate how many pages we need to retrieve based on the total count returned by the first call. Then, with the `insert_into` method, we append the data from each page into `github_raw_data`:

```
for page in range(2, number_pages + 1):
    logger.info(f"Fetching {page} out of {number_pages}")
    (
        duckdb_conn
        .read_json(f"{api_url}&per_page=100&page={page}")
        .insert_into("github_raw_data")
    )
```

Saving Data to a Markdown File

With the data available in a table, we can continue our data processing, by using the `unnest` function, which will flatten the `items` returned by the API:

```
(
    duckdb_conn.table("github_raw_data")
    .select("unnest(items, recursive := true)")
)
```

By using `recursive` unnesting, the STRUCT objects within `items` will be flattened too; below is a sample of the columns derived from `items` after unnesting:

```
['id',
 'node_id',
 'name',
 'full_name',
 'private',
 'login',
 'id',
 ...
 'pull',
 'score']
```

Our scope is to create a Markdown file containing a table, with the following format:

- **name**, containing the name, link, description, license and owner of the repository;
- **topics**, the list of topics of the repository;
- **stars**, the number of stars;
- **open issues**, the number of open issues;

- **forks**, the number of forks;
- **created at**, when the repository was created;
- **updated at**, when the repository was last updated.

To retrieve the `name` field we use `concat_ws` and `concat` functions, in order to generate the hyperlink and text with the Markdown newline character(<br>):

```
selection_query = (
    duckdb_conn.table("github_raw_data")
    .select("unnest(items, recursive := true)")
    .select("""
        concat_ws(
            '<br>',
            concat('[', name, '](#', concat('https:::github.com:', full_name),')'),
            coalesce(description, ' '),
            concat('**License** ', coalesce(name_1, 'unknown')),
            concat('**Owner** ', login)
        ) as repo_details
    """)
)
```

The above returns:

```
repo_details = [duckdb-web](https://github.com/duckdb/duckdb-web)<br>DuckDB website and
documentation<br>**License** MIT License<br>**Owner** duckdb
repo_details = [duckdb](https://github.com/duckdb/duckdb)<br>DuckDB is an analytical in-process SQL
database management system<br>**License** MIT License<br>**Owner** duckdb
...
```

We also calculate metrics such as stars, open issues count and forks, which we sum into a field `activity_count`. A repository will end up in the list only if it has an `activity_count` greater than 3 and if it is not a fork. From above we also see that repositories owned by DuckDB are returned, therefore we filter them out too:

```
selection_query.filter("""
    login != 'duckdb'
    and not fork
    and activity_count >= 3
""")
```

A Markdown table is similar to a CSV file, separated by pipe (|), but it must:

- contain the separator at the beginning and end of each row;
- between the table header and the first table row there needs to be a row with dashes (called delimiter row).

In order to export the data as a Markdown file we are applying a few tricks. The first one is to select the header by adding dummy columns, at the beginning and end, containing `NULL`:

```
duckdb_conn.sql("""
    select
        NULL,
        'Name',
        'Topics',
        'Stars',
        'Open Issues',
        'Forks',
        'Created At',
        'Updated At',
        NULL
""")
```

We then union the above header with the delimiter row:

```
.union(
    duckdb_conn.sql("""
    select
        NULL as '',
        '--' as "Name",
        '--' as "Topics",
        '--' as "Stars",
        '--' as "Open Issues",
        '--' as "Forks",
        '--' as "Created At",
        '--' as "Updated At",
        NULL as ''
""")
)
```

Returning:

NULL int32	'Name' varchar	'Topics' varchar	'Stars' varchar	'Open Issues' varchar	'Forks' varchar	'Created At' varchar	'Updated At' varchar	NULL int32
NULL NULL	Name --	Topics --	Stars --	Open Issues --	Forks --	Created At --	Updated At --	NULL NULL

And finally we union with the initial GitHub selection query and export the data to CSV, by disabling the header export:

```
    .union(selection_query)
).to_csv("./exported_records.md", sep="|", header=False)
```

Below is a sample of the `exported_records.md` file, rendered as HTML:

Name	Topics	Stars	Open Issues	Forks	Created At	Updated At
tailpipeselect * from logs;**License** GNU Affero General Public License v3.0**Owner** turbot	[aws, azure, detections, devops, duckdb, forensics, gcp, incident-response, log-analysis, mitre-attack, open-source, parquet, siem, tailpipe, threat-detection]	438	41	9	2024-04-18 02:44:35	2025-06-17 11:57:42

Why a Markdown file you may wonder. Because it is rendered automatically by GitHub and there is no need to host our tiny data application somewhere else. In fact, we copy the above table to README, such that it is displayed automatically on the first page of the repository:

```
echo '# Repositories using `duckdb`' > README.md && \
cat exported_records.md >> README.md
```

Another way to create the README file is by using the `string_agg` function:

```
selected_data = (
    selection_query
    .select("""
        concat(
```

```
            '|',
            concat_ws(
                '|',
                repo_details,
                topics, stars,
                open_issues,
                forks,
                created_at,
                updated_at
            )
        ) as line
    """)
    .string_agg(
        'line',
        sep='|\n'
    )
).fetchone()[0]

with open('README.md', 'w') as readme_file:
    readme_file.write("# Repositories using `duckdb`\n")
    readme_file.write("|Name|Topics|Stars|Open Issues|Forks|Created At|Updated At|\n")
    readme_file.write(f"{duckdb_conn.sql("select concat(repeat('|--', 7),'|')").fetchone()[0]}\n")
    readme_file.write(f"{selected_data}|")
```

In the above code snippet we concatenate the columns with the pipe character and then we aggregate the records into a string, separated by `|\n` in order to add a pipe and newline at the end of each line. We then use Python to write to README the title of the page, the Markdown table header, the delimiter row (by using the `repeat` function) and the data itself.

> The data size is very small, each commit having a README of approx. 35 KB.

Automating with GitHub Workflow

With GitHub workflows we automated the above data processing steps. We first define a Makefile in our project to configure the steps needed to be executed in the workflow:

```
search-repos:
    uv run using_duckdb/search_repositories.py

readme:
    echo '# Repositories using `duckdb`' > README.md && \
    cat exported_records.md >> README.md
```

In the settings of the GitHub repository we create a repository secret which contains the value of the API token generated above. We can then configure an environment variable in our workflow and define the processing steps:

```
name: Search

on:
  workflow_dispatch:
  schedule:
    - cron: '37 5 * * *'

env:
  READ_PUBLIC_REPO_TOKEN : ${{ secrets.READ_PUBLIC_REPO_TOKEN }}

jobs:
  make-readme-md:
    runs-on: ubuntu-latest
```

```
    permissions:
      contents: write

    steps:
      - uses: actions/checkout@v4
        with:
          ref: ${{ github.head_ref }}
          fetch-depth: 0

      - name: Install uv
        uses: astral-sh/setup-uv@v6

      - name: Install the project
        run: uv sync

      - name: Get search results
        run: make search-repos

      - name: Export current results
        run: make readme

      - name: Get datetime
        id: datetime
        run: echo "datetime=$(date -u +'%Y-%m-%dT%H:%M:%SZ')" >> $GITHUB_OUTPUT

      - uses: stefanzweifel/git-auto-commit-action@v5
        with:
          commit_message: 'Update as of ${{ steps.datetime.outputs.datetime }}'
          file_pattern: 'README.md'
```

With git-auto-commit-action we delegate to GitHub the right to create a commit with the updated `README.md` file:

Visualizing Historical Data

Because we store the search results into the README file and we update it daily with Git, we can retrieve the history of the search by getting the Git difference for each commit:

```
git-log:
    echo '|Name|Topics|Stars|Open Issues|Forks|Created At|Updated At|' > git_log.md && \
    echo '|--|--|--|--|--|--|--|' >> git_log.md && \
    git log --follow -p --pretty=format:"" -- README.md | grep '^+|\[' | sed 's/^+//' >> git_log.md && \
    git diff -- README.md | grep '^+|\[' | sed 's/^+//' >> git_log.md
```

In the above Makefile command we:

- create a file, `git_log.md`, which has on the first line the header of the Markdown table;
- add to `git_log.md` the delimiter row;
- for each commit, append to `git_log.md` the new lines of the README differences;
- append to `git_log.md` the new lines from the current change of README.

The `git_log.md` contains now the entire history of appended records to the README file and it can be used to display a bubble chart with the repositories mentioning DuckDB:

The above plot is generated with Plotly by providing as data source a DuckDB Python relation:

```
px.scatter(
    duckdb_conn.read_csv(
        "./git_log.md"
        skiprows=2
```

```
)
    .select("""
    #2 as repo,
    #3 as topics,
    #4 as stars,
    #5 as open_issues,
    #6 as forks,
    strftime(#7, '%B %d, %Y, %H:%m' ) as created_at,
    #8 as updated_at,
    if(stars + forks + open_issues = 0, 0, log(stars + forks + open_issues)) as log_activity_count,
    substr(repo, position('[' in repo) + 1 , position(']' in repo) - 2) as repo_name,
    count(distinct updated_at::date) over (partition by repo_name) as number_of_updates,
    row_number() over (partition by repo_name order by updated_at desc) as rn
""")
.filter("rn = 1")
.order("updated_at"),
x="updated_at",
y="log_activity_count",
labels={
    "updated_at": "Updated Date",
    "log_activity_count": "Activity count, based on stars, open issues and forks"
},
...
```

> Using Markdown as a file format for our historical data might not be the best solution, because – among others – it does not support schema evolution. While we cannot do changes in the middle of the file (except column renames), we can remove the last column or add new ones at the end, due to the way DuckDB reads malformed CSV files.

Conclusion

In this post we showed how DuckDB can be used to process API requests and historical data from git commits. Some of the repositories returned by the GitHub search API made it to awesome-duckdb, which is our go-to list for curated DuckDB related projects; for example tailpipe, an open source Security Information and Event Management for instant log insights, and preswald, a Wasm packager for Python-based interactive data apps.

Acknowledgments

This document is built with Pandoc using the Eisvogel template. The scripts to build the document are available in the DuckDB-Web repository.

The emojis used in this document are provided by Twemoji under the CC-BY 4.0 license.

The syntax highlighter uses the Bluloco Light theme by Umut Topuzoğlu.