<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>C API - Query</td>
<td>53</td>
</tr>
<tr>
<td>C API - Data Chunks</td>
<td>62</td>
</tr>
<tr>
<td>C API - Values</td>
<td>73</td>
</tr>
<tr>
<td>C API - Types</td>
<td>77</td>
</tr>
<tr>
<td>C API - Prepared Statements</td>
<td>106</td>
</tr>
<tr>
<td>C API - Appender</td>
<td>120</td>
</tr>
<tr>
<td>C API - Table Functions</td>
<td>130</td>
</tr>
<tr>
<td>C API - Replacement Scans</td>
<td>145</td>
</tr>
<tr>
<td>C API - Complete API</td>
<td>147</td>
</tr>
<tr>
<td>C++ API</td>
<td>257</td>
</tr>
<tr>
<td>CLI</td>
<td>262</td>
</tr>
<tr>
<td>CLI API</td>
<td>262</td>
</tr>
<tr>
<td>Command Line Arguments</td>
<td>268</td>
</tr>
<tr>
<td>Dot Commands</td>
<td>269</td>
</tr>
<tr>
<td>Output Formats</td>
<td>274</td>
</tr>
<tr>
<td>Go</td>
<td>275</td>
</tr>
<tr>
<td>Java JDBC API</td>
<td>277</td>
</tr>
<tr>
<td>Julia Package</td>
<td>280</td>
</tr>
<tr>
<td>Node.js</td>
<td>282</td>
</tr>
<tr>
<td>Node.js API</td>
<td>282</td>
</tr>
<tr>
<td>Node.js API</td>
<td>285</td>
</tr>
<tr>
<td>Python</td>
<td>301</td>
</tr>
<tr>
<td>Python API</td>
<td>301</td>
</tr>
<tr>
<td>Data Ingestion</td>
<td>304</td>
</tr>
<tr>
<td>Result Conversion</td>
<td>308</td>
</tr>
<tr>
<td>Python DB API</td>
<td>310</td>
</tr>
<tr>
<td>Relational API</td>
<td>313</td>
</tr>
<tr>
<td>Python Function API</td>
<td>318</td>
</tr>
<tr>
<td>Types API</td>
<td>322</td>
</tr>
<tr>
<td>Expression API</td>
<td>325</td>
</tr>
<tr>
<td>Spark API</td>
<td>329</td>
</tr>
<tr>
<td>Python Client API</td>
<td>330</td>
</tr>
<tr>
<td>Known Python Issues</td>
<td>330</td>
</tr>
<tr>
<td>R API</td>
<td>331</td>
</tr>
<tr>
<td>Rust API</td>
<td>334</td>
</tr>
<tr>
<td>Swift API</td>
<td>335</td>
</tr>
<tr>
<td>Wasm</td>
<td>338</td>
</tr>
<tr>
<td>DuckDB Wasm</td>
<td>338</td>
</tr>
<tr>
<td>Instantiation</td>
<td>338</td>
</tr>
<tr>
<td>Data Ingestion</td>
<td>341</td>
</tr>
<tr>
<td>Query</td>
<td>344</td>
</tr>
<tr>
<td>Extensions</td>
<td>345</td>
</tr>
<tr>
<td>ADBC API</td>
<td>348</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>SQLite Extension</td>
<td>706</td>
</tr>
<tr>
<td>Substrait Extension</td>
<td>711</td>
</tr>
<tr>
<td>TPC-DS Extension</td>
<td>714</td>
</tr>
<tr>
<td>TPC-H Extension</td>
<td>715</td>
</tr>
<tr>
<td>Guides</td>
<td>719</td>
</tr>
<tr>
<td>Data Import &amp; Export</td>
<td>721</td>
</tr>
<tr>
<td>CSV Import</td>
<td>721</td>
</tr>
<tr>
<td>CSV Export</td>
<td>721</td>
</tr>
<tr>
<td>Parquet Import</td>
<td>721</td>
</tr>
<tr>
<td>Parquet Export</td>
<td>722</td>
</tr>
<tr>
<td>Querying Parquet Files</td>
<td>722</td>
</tr>
<tr>
<td>HTTP Parquet Import</td>
<td>722</td>
</tr>
<tr>
<td>S3 Parquet Import</td>
<td>723</td>
</tr>
<tr>
<td>S3 Parquet Export</td>
<td>724</td>
</tr>
<tr>
<td>S3 Iceberg Import</td>
<td>725</td>
</tr>
<tr>
<td>JSON Import</td>
<td>726</td>
</tr>
<tr>
<td>JSON Export</td>
<td>726</td>
</tr>
<tr>
<td>Excel Import</td>
<td>727</td>
</tr>
<tr>
<td>Excel Export</td>
<td>728</td>
</tr>
<tr>
<td>MySQL Import</td>
<td>728</td>
</tr>
<tr>
<td>PostgreSQL Import</td>
<td>729</td>
</tr>
<tr>
<td>SQLite Import</td>
<td>731</td>
</tr>
<tr>
<td>Performance</td>
<td>733</td>
</tr>
<tr>
<td>Performance Guide</td>
<td>733</td>
</tr>
<tr>
<td>Schema</td>
<td>733</td>
</tr>
<tr>
<td>Indexing</td>
<td>734</td>
</tr>
<tr>
<td>Environment</td>
<td>736</td>
</tr>
<tr>
<td>File Formats</td>
<td>737</td>
</tr>
<tr>
<td>Tuning Workloads</td>
<td>739</td>
</tr>
<tr>
<td>My Workload Is Slow</td>
<td>742</td>
</tr>
<tr>
<td>Benchmarks</td>
<td>742</td>
</tr>
<tr>
<td>Meta Queries</td>
<td>745</td>
</tr>
<tr>
<td>Describe</td>
<td>745</td>
</tr>
<tr>
<td>Inspecting Query Plans Using EXPLAIN</td>
<td>745</td>
</tr>
<tr>
<td>List Tables</td>
<td>747</td>
</tr>
<tr>
<td>Profile Queries Using EXPLAIN ANALYZE</td>
<td>748</td>
</tr>
<tr>
<td>Summarize</td>
<td>750</td>
</tr>
<tr>
<td>DuckDB Environment</td>
<td>752</td>
</tr>
</tbody>
</table>
ODBC 755
  ODBC 101: A Duck Themed Guide to ODBC 755

Python 763
  Installing the Python Client 763
  Executing SQL in Python 763
  Jupyter Notebooks 764
  SQL on Pandas 768
  Import from Pandas 769
  Export to Pandas 769
  SQL on Apache Arrow 769
  Import from Apache Arrow 772
  Export to Apache Arrow 772
  Relational API on Pandas 774
  Multiple Python Threads 774
  Integration with Ibis 777
  Integration with Polars 789
  Using fsspec Filesystems 791

SQL Features 793
  AsOf Join 793
  Full-Text Search 796

SQL Editors 799
  DBeaver SQL IDE 799

Data Viewers 801
  Tableau - A Data Visualization Tool 801
  CLI Charting with YouPlot 806

Under the Hood 809

Internals 811
  Overview of DuckDB Internals 811
  Storage 813
  Execution Format 815

Developer Guides 819
  Building DuckDB from Source 819
  Profiling 824
  Testing 827
    sqllogictest 830
    sqllogictest - Debugging 832
    sqllogictest - Result Verification 833
    sqllogictest - Persistent Testing 837
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
Documentation
Connect

Connect or Create a Database

To use DuckDB, you must first create a connection to a database. The exact process varies by client. Most clients take a parameter pointing to a database file to read and write from (the file extension may be anything, e.g., .db, .duckdb, etc.). If the database file does not exist, it will be created. The special value :memory: can be used to create an in-memory database where no data is persisted to disk (i.e., all data is lost when you exit the process).

See the API docs for client-specific details.
Data Import

Importing Data

The first step to using a database system is to insert data into that system. DuckDB provides several data ingestion methods that allow you to easily and efficiently fill up the database. In this section, we provide an overview of these methods so you can select which one is correct for you.

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.

CSV Loading

Data can be efficiently loaded from CSV files using the `read_csv` function or the COPY statement.

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

You can also load data from compressed (e.g., compressed with gzip) CSV files, for example:

```
SELECT * FROM read_csv('test.csv.gz');
```

For more details, see the page on CSV loading.

Parquet Loading

Parquet files can be efficiently loaded and queried using the `read_parquet` function.

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

For more details, see the page on Parquet loading.
JSON Loading

JSON files can be efficiently loaded and queried using the `read_json_auto` function.

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

For more details, see the page on JSON loading.

Appender (C++ and Java)

In C++ and Java, 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 needing to use SQL.

C++:
```
Appender appender(con, "people");
appender.AppendRow(1, "Mark");
appender.Close();
```

Java:
```
con.createAppender("main", "people");
appender.beginRow();
appender.append("Mark");
appender.endRow();
appender.close();
```

For a detailed description, see the pages on the C++ appender and the Java appender.

CSV Files

CSV Import

Examples

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

```sql
-- read a CSV file from disk, auto-infer options
SELECT * FROM 'flights.csv';
-- read_csv 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
```sql
cat flights.csv | duckdb -c "SELECT * FROM read_csv('/dev/stdin')"
```

-- read a CSV file into a table
```sql
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
```sql
CREATE TABLE ontime AS SELECT * FROM 'flights.csv';
```

-- we can use the FROM-first syntax to omit `SELECT *`
```sql
CREATE TABLE ontime AS FROM 'flights.csv';
```

-- write the result of a query to a CSV file
```sql
COPY (SELECT * FROM ontime) TO 'flights.csv' WITH (HEADER true, DELIMITER '|');
```

-- if we serialize the entire table, we can simply refer to it with its name
```sql
COPY ontime TO 'flights.csv' WITH (HEADER true, DELIMITER '|');
```

## 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 CSV reader. These parameters are accepted by both the `COPY` statement and the `read_csv` function.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Type</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>all_vARCHAR</td>
<td>Option to skip type detection for CSV parsing and assume all columns to be of type VARCHAR.</td>
<td>BOOL</td>
<td>false</td>
</tr>
<tr>
<td>auto_detect</td>
<td>Enables auto detection of CSV parameters.</td>
<td>BOOL</td>
<td>true</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Type</td>
<td>Default</td>
</tr>
<tr>
<td>--------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>----------------</td>
<td>----------------------------------------------</td>
</tr>
<tr>
<td>auto_type_candidates</td>
<td>This option allows you to specify the types that the sniffer will use when detecting CSV column types, e.g., <code>SELECT * FROM read_csv('csv_file.csv', auto_type_candidates=[ 'BIGINT', 'DATE'])</code>. The VARCHAR type is always included in the detected types (as a fallback option).</td>
<td>TYPE[]</td>
<td>['SQLNULL', 'BOOLEAN', 'BIGINT', 'DOUBLE', 'DATE', 'TIME', 'TIMESTAMP', 'VARCHAR']</td>
</tr>
<tr>
<td>buffer_size</td>
<td>The buffer size used by the CSV reader, specified in bytes. By default, it is set to 32MB or the size of the CSV file (if smaller). The buffer size must be at least as large as the longest line in the CSV file. Note: this is an advanced option that has a significant impact on performance and memory usage.</td>
<td>BIGINT</td>
<td>min(32000000, CSV file size)</td>
</tr>
<tr>
<td>columns</td>
<td>A struct that specifies the column names and column types contained within the CSV file (e.g., <code>{'col1': 'INTEGER', 'col2': 'VARCHAR'}</code>). Using this option implies that auto detection is not used.</td>
<td>STRUCT</td>
<td>(empty)</td>
</tr>
<tr>
<td>compression</td>
<td>The compression type for the file. By default this will be detected automatically from the file extension (e.g., t.csv.gz will use gzip, t.csv will use none). Options are none, gzip, zstd.</td>
<td>VARCHAR</td>
<td>auto</td>
</tr>
<tr>
<td>dateformat</td>
<td>Specifies the date format to use when parsing dates. See Date Format.</td>
<td>VARCHAR</td>
<td>(empty)</td>
</tr>
<tr>
<td>decimal_separator</td>
<td>The decimal separator of numbers.</td>
<td>VARCHAR</td>
<td>.</td>
</tr>
<tr>
<td>delimorsep</td>
<td>Specifies the string that separates columns within each row (line) of the file.</td>
<td>VARCHAR</td>
<td>,</td>
</tr>
<tr>
<td>escape</td>
<td>Specifies the string that should appear before a data character sequence that matches the quote value.</td>
<td>VARCHAR</td>
<td>&quot;</td>
</tr>
<tr>
<td>filename</td>
<td>Whether or not an extra filename column should be included in the result.</td>
<td>BOOL</td>
<td>false</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Type</td>
<td>Default</td>
</tr>
<tr>
<td>--------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-----------</td>
<td>----------</td>
</tr>
<tr>
<td>force_not_null</td>
<td>Do not match the specified columns' values against the NULL string. In the default case where the NULL string is empty, this means that empty values will be read as zero-length strings rather than NULLs.</td>
<td>VARCHAR[]</td>
<td>[]</td>
</tr>
<tr>
<td>header</td>
<td>Specifies that the file contains a header line with the names of each column in the file.</td>
<td>BOOL</td>
<td>false</td>
</tr>
<tr>
<td>hive_partitioning</td>
<td>Whether or not to interpret the path as a hive partitioned path.</td>
<td>BOOL</td>
<td>false</td>
</tr>
<tr>
<td>ignore_errors</td>
<td>Option to ignore any parsing errors encountered - and instead ignore rows with errors.</td>
<td>BOOL</td>
<td>false</td>
</tr>
<tr>
<td>max_line_size</td>
<td>The maximum line size in bytes.</td>
<td>BIGINT</td>
<td>2097152</td>
</tr>
<tr>
<td>names</td>
<td>The column names as a list, see example.</td>
<td>VARCHAR[]</td>
<td>(empty)</td>
</tr>
<tr>
<td>new_line</td>
<td>Set the new line character(s) in the file. Options are '\r','\n', or '\r\n'.</td>
<td>VARCHAR</td>
<td>(empty)</td>
</tr>
<tr>
<td>normalize_names</td>
<td>Boolean value that specifies whether or not column names should be normalized, removing any non-alphanumeric characters from them.</td>
<td>BOOL</td>
<td>false</td>
</tr>
<tr>
<td>null_padding</td>
<td>If this option is enabled, when a row lacks columns, it will pad the remaining columns on the right with null values.</td>
<td>BOOL</td>
<td>false</td>
</tr>
<tr>
<td>nullstr</td>
<td>Specifies the string that represents a NULL value.</td>
<td>VARCHAR</td>
<td>(empty)</td>
</tr>
<tr>
<td>parallel</td>
<td>Whether or not the parallel CSV reader is used.</td>
<td>BOOL</td>
<td>true</td>
</tr>
<tr>
<td>quote</td>
<td>Specifies the quoting string to be used when a data value is quoted.</td>
<td>VARCHAR</td>
<td>&quot;</td>
</tr>
<tr>
<td>sample_size</td>
<td>The number of sample rows for auto detection of parameters.</td>
<td>BIGINT</td>
<td>20480</td>
</tr>
<tr>
<td>skip</td>
<td>The number of lines at the top of the file to skip.</td>
<td>BIGINT</td>
<td>0</td>
</tr>
<tr>
<td>timestampformat</td>
<td>Specifies the date format to use when parsing timestamps. See Date Format</td>
<td>VARCHAR</td>
<td>(empty)</td>
</tr>
<tr>
<td>types or dtypes</td>
<td>The column types as either a list (by position) or a struct (by name). Example here.</td>
<td>VARCHAR[]</td>
<td>(empty)  or STRUCT</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Type</td>
<td>Default</td>
</tr>
<tr>
<td>---------------</td>
<td>------------------------------------------------------------------------------</td>
<td>------</td>
<td>---------</td>
</tr>
<tr>
<td>union_by_name</td>
<td>Whether the columns of multiple schemas should be unified by name, rather than by position.</td>
<td>BOOL</td>
<td>false</td>
</tr>
</tbody>
</table>

**CSV Functions**

**Note.** DuckDB 0.9.3-dev and the upcoming v0.10.0 versions introduce breaking changes to the `read_csv` function. Namely, The `read_csv` function now attempts auto-detecting the CSV parameters, making its behavior identical to the old `read_csv_auto` function. If you would like to use `read_csv` with its old behavior, turn off the auto-detection manually by using `read_csv(..., auto_detect = false)`.

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:

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

<table>
<thead>
<tr>
<th>FlightDate</th>
<th>UniqueCarrier</th>
<th>OriginCityName</th>
<th>DestCityName</th>
</tr>
</thead>
<tbody>
<tr>
<td>1988-01-01</td>
<td>AA</td>
<td>New York, NY</td>
<td>Los Angeles, CA</td>
</tr>
<tr>
<td>1988-01-02</td>
<td>AA</td>
<td>New York, NY</td>
<td>Los Angeles, CA</td>
</tr>
<tr>
<td>1988-01-03</td>
<td>AA</td>
<td>New York, NY</td>
<td>Los Angeles, CA</td>
</tr>
</tbody>
</table>

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:

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Null</th>
<th>Key</th>
<th>Default</th>
<th>Extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>FlightDate</td>
<td>DATE</td>
<td>YES</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>UniqueCarrier</td>
<td>VARCHAR</td>
<td>YES</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>OriginCityName</td>
<td>VARCHAR</td>
<td>YES</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>DestCityName</td>
<td>VARCHAR</td>
<td>YES</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>

```sql
SELECT * FROM read_csv('flights.csv', sample_size = 20000);
```
If we set `delim/sep`, `quote`, `escape`, or `header` explicitly, we can bypass the automatic detection of this particular parameter:

```sql
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.

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

<table>
<thead>
<tr>
<th>flightdate</th>
<th>uniquecarrier</th>
<th>origincityname</th>
<th>destcityname</th>
</tr>
</thead>
<tbody>
<tr>
<td>1988-01-01</td>
<td>AA</td>
<td>New York, NY</td>
<td>Los Angeles, CA</td>
</tr>
<tr>
<td>1988-01-02</td>
<td>AA</td>
<td>New York, NY</td>
<td>Los Angeles, CA</td>
</tr>
<tr>
<td>1988-01-03</td>
<td>AA</td>
<td>New York, NY</td>
<td>Los Angeles, CA</td>
</tr>
</tbody>
</table>

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

```sql
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.
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 = '|')`).

The 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 20480 rows. Setting the `sample_size` parameter to `-1` means the entire file is read for sampling. 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.

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

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delimiter</td>
<td>delimiter</td>
<td>,</td>
</tr>
<tr>
<td>Quote</td>
<td>quote character</td>
<td>&quot;</td>
</tr>
<tr>
<td>Escape</td>
<td>escape</td>
<td>\</td>
</tr>
<tr>
<td>NewLineDelimiter</td>
<td>new-line delimiter</td>
<td>\r\n</td>
</tr>
<tr>
<td>SkipRow</td>
<td>number of rows skipped</td>
<td>1</td>
</tr>
<tr>
<td>HasHeader</td>
<td>whether the CSV has a header</td>
<td>true</td>
</tr>
<tr>
<td>Columns</td>
<td>column types encoded as a</td>
<td>({'name': 'VARCHAR', 'age': 'BIGINT'})</td>
</tr>
<tr>
<td>DateFormat</td>
<td>date Format</td>
<td>%d/%m/%Y</td>
</tr>
<tr>
<td>TimestampFormat</td>
<td>timestamp Format</td>
<td>%Y-%m-%dT%H%M%S.%f</td>
</tr>
</tbody>
</table>
The Prompt column contains a SQL command with the configurations detected by the sniffer.

```sql
-- 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=',', auto_detect=false, delim=',', quote='', escape='', new_line='
', 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.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Considered values</th>
</tr>
</thead>
<tbody>
<tr>
<td>delim</td>
<td>,</td>
</tr>
<tr>
<td>quote</td>
<td>'' (empty)</td>
</tr>
<tr>
<td>escape</td>
<td>'' \ (empty)</td>
</tr>
</tbody>
</table>

Consider the example file `flights.csv`:

<table>
<thead>
<tr>
<th>FlightDate</th>
<th>UniqueCarrier</th>
<th>OriginCityName</th>
<th>DestCityName</th>
</tr>
</thead>
<tbody>
<tr>
<td>1988-01-01</td>
<td>AA</td>
<td>New York, NY</td>
<td>Los Angeles, CA</td>
</tr>
<tr>
<td>1988-01-02</td>
<td>AA</td>
<td>New York, NY</td>
<td>Los Angeles, CA</td>
</tr>
<tr>
<td>1988-01-03</td>
<td>AA</td>
<td>New York, NY</td>
<td>Los Angeles, CA</td>
</tr>
</tbody>
</table>

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 set of considered candidate types in order of priority is given below:

```
<table>
<thead>
<tr>
<th>Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOLEAN</td>
</tr>
<tr>
<td>BIGINT</td>
</tr>
<tr>
<td>DOUBLE</td>
</tr>
<tr>
<td>TIME</td>
</tr>
<tr>
<td>DATE</td>
</tr>
<tr>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>VARCHAR</td>
</tr>
</tbody>
</table>
```

Note everything can be cast to VARCHAR. This type has the lowest priority - i.e., 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 detected types can be individually overridden using the `types` option. This option takes either a list of types (e.g., `types=[INT, VARCHAR, DATE]`) which 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': INT}`).

The 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).

**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 no header. This can be overridden using the header option.

**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 date format and timestamp format 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 chooses incorrectly, the user can specify the date format and timestamp format options manually.

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

<table>
<thead>
<tr>
<th>date format</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISO 8601</td>
</tr>
<tr>
<td>%y-%m-%d</td>
</tr>
<tr>
<td>%Y-%m-%d</td>
</tr>
<tr>
<td>%d-%m-%y</td>
</tr>
<tr>
<td>%d-%m-%Y</td>
</tr>
<tr>
<td>%m-%d-%y</td>
</tr>
<tr>
<td>%m-%d-%Y</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>timestamp format</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISO 8601</td>
</tr>
<tr>
<td>%y-%m-%d %H:%M:%S</td>
</tr>
<tr>
<td>%Y-%m-%d %H:%M:%S</td>
</tr>
<tr>
<td>%d-%m-%y %H:%M:%S</td>
</tr>
</tbody>
</table>
timestamp format

%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**

Reading erroneous CSV files is possible by utilizing the `ignore_errors` option. With that option set, rows containing data that would otherwise cause the CSV Parser to generate an error will be ignored.

**Using the `ignore_errors` Option**

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.

```sql
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:

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

Outputs:

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pedro</td>
<td>31</td>
</tr>
</tbody>
</table>

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:
**DuckDB Documentation**

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

Outputs:

<table>
<thead>
<tr>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pedro</td>
</tr>
<tr>
<td>Oogie Boogie</td>
</tr>
</tbody>
</table>

**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. It is important to note that the Rejects Table can only be used when `ignore_errors` is set, and currently, only stores casting errors and does not save errors when the number of columns differ.

The CSV Rejects Table returns the following information:

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>file</td>
<td>File path.</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>line</td>
<td>Line number, from the CSV File, where the error occurred.</td>
<td>INTEGER</td>
</tr>
<tr>
<td>column</td>
<td>Column number, from the CSV File, where the error occurred.</td>
<td>INTEGER</td>
</tr>
<tr>
<td>column_name</td>
<td>Column name, from the CSV File, where the error occurred.</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>parsed_value</td>
<td>The value, where the casting error happened, in a string format.</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>recovery_columns</td>
<td>An optional primary key of the CSV File.</td>
<td>STRUCT {NAME: VALUE}</td>
</tr>
<tr>
<td>error</td>
<td>Exact error encountered by the parser.</td>
<td>VARCHAR</td>
</tr>
</tbody>
</table>

**Parameters**

The parameters listed below are used in the `read_csv` function to configure the CSV Rejects Table.
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Type</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>rejects_table</td>
<td>Name of a temporary table where the information of the faulty lines of a CSV file are stored.</td>
<td>VARCHAR</td>
<td>(empty)</td>
</tr>
<tr>
<td>rejects_limit</td>
<td>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.</td>
<td>BIGINT</td>
<td>0</td>
</tr>
<tr>
<td>rejects_recovery_columns</td>
<td>Column values that serve as a primary key to the CSV file. The are stored in the CSV Rejects Table to help identify the faulty tuples.</td>
<td>VARCHAR[]</td>
<td>(empty)</td>
</tr>
</tbody>
</table>

To store the information of the faulty CSV lines in a rejects table, the user must simply provide the rejects table name in the `rejects_table` option. For example:

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

You can then query the `rejects_table` table to retrieve information about the rejected tuples. For example:

```
FROM rejects_table;
```

Outputs:

<table>
<thead>
<tr>
<th>file</th>
<th>line</th>
<th>column</th>
<th>column_name</th>
<th>parsed_value</th>
<th>error</th>
</tr>
</thead>
<tbody>
<tr>
<td>faulty.csv</td>
<td>2</td>
<td>1</td>
<td>age</td>
<td>three</td>
<td>Could not convert string 'three' to 'INTEGER'</td>
</tr>
</tbody>
</table>

Additionally, the `name` column could also be provided as a primary key via the `rejects_recovery_columns` option to provide more information over the faulty lines. For example:

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

Reading from the `rejects_table` will return:

<table>
<thead>
<tr>
<th>file</th>
<th>line</th>
<th>column_ name</th>
<th>parsed_ value</th>
<th>recovery_columns</th>
<th>error</th>
</tr>
</thead>
<tbody>
<tr>
<td>faulty.csv</td>
<td>2</td>
<td>age</td>
<td>three</td>
<td>{'name': 'Oogie Boogie'}</td>
<td>Could not convert string 'three' to 'INTEGER'</td>
</tr>
</tbody>
</table>

**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.

```sql
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.

```sql
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.

```sql
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.

```sql
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.

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

JSON Loading

Examples

-- read a JSON file from disk, auto-infer options
SELECT * FROM 'todos.json';
-- read_json 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_auto('/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';
-- alternatively, create a table without specifying the schema manually
CREATE TABLE todos AS SELECT * FROM 'todos.json';

-- write the result of a query to a JSON file
COPY (SELECT * FROM todos) TO 'todos.json';

JSON Loading

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.

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.

Below are parameters that can be passed in to the JSON reader.

Parameters
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Type</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>maximum_object_size</td>
<td>The maximum size of a JSON object (in bytes)</td>
<td>UINTeger</td>
<td>16777216</td>
</tr>
<tr>
<td>format</td>
<td>Can be one of ['auto', 'unstructured', 'newline_delimited', 'array']</td>
<td>VARCHAR</td>
<td>'array'</td>
</tr>
<tr>
<td>ignore_errors</td>
<td>Whether to ignore parse errors (only possible when format is 'newline_delimited')</td>
<td>BOOL</td>
<td>false</td>
</tr>
<tr>
<td>compression</td>
<td>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'.</td>
<td>VARCHAR</td>
<td>'auto'</td>
</tr>
<tr>
<td>columns</td>
<td>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</td>
<td>STRUCT</td>
<td>(empty)</td>
</tr>
<tr>
<td>records</td>
<td>Can be one of ['auto', 'true', 'false']</td>
<td>VARCHAR</td>
<td>'records'</td>
</tr>
<tr>
<td>auto_detect</td>
<td>Whether to auto-detect detect the names of the keys and data types of the values automatically</td>
<td>BOOL</td>
<td>false</td>
</tr>
<tr>
<td>sample_size</td>
<td>Option to define number of sample objects for automatic JSON type detection. Set to -1 to scan the entire input file</td>
<td>UBIGINT</td>
<td>20480</td>
</tr>
<tr>
<td>maximum_depth</td>
<td>Maximum nesting depth to which the automatic schema detection detects types. Set to -1 to fully detect nested JSON types</td>
<td>BIGINT</td>
<td>-1</td>
</tr>
<tr>
<td>dateformat</td>
<td>Specifies the date format to use when parsing dates. See Date Format</td>
<td>VARCHAR</td>
<td>'iso'</td>
</tr>
<tr>
<td>timestampformat</td>
<td>Specifies the date format to use when parsing timestamps. See Date Format</td>
<td>VARCHAR</td>
<td>'iso'</td>
</tr>
<tr>
<td>filename</td>
<td>Whether or not an extra filename column should be included in the result.</td>
<td>BOOL</td>
<td>false</td>
</tr>
<tr>
<td>hive_partitioning</td>
<td>Whether or not to interpret the path as a hive partitioned path.</td>
<td>BOOL</td>
<td>false</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Type</td>
<td>Default</td>
</tr>
<tr>
<td>------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>------</td>
<td>---------</td>
</tr>
<tr>
<td>union_by_name</td>
<td>Whether the schema's of multiple JSON files should be unified.</td>
<td>BOOL</td>
<td>false</td>
</tr>
</tbody>
</table>

When using `read_json_auto`, every parameter that supports auto-detection is enabled.

**Examples of 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.

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

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

<table>
<thead>
<tr>
<th>key1</th>
<th>key2</th>
</tr>
</thead>
<tbody>
<tr>
<td>value1</td>
<td>value1</td>
</tr>
<tr>
<td>value2</td>
<td>value2</td>
</tr>
<tr>
<td>value3</td>
<td>value3</td>
</tr>
</tbody>
</table>

**Format: array**  If the JSON file contains a JSON array of objects (pretty-printed or not), `array_of_objects` may be used.

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

```
SELECT * FROM read_json_auto('array.json', format = 'array');
```
### Format: unstructured

If the JSON file contains JSON that is not newline-delimited or an array, **unstructured** may be used.

The following JSON is valid:

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

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

<table>
<thead>
<tr>
<th>key1</th>
<th>key2</th>
</tr>
</thead>
<tbody>
<tr>
<td>value1</td>
<td>value1</td>
</tr>
<tr>
<td>value2</td>
<td>value2</td>
</tr>
<tr>
<td>value3</td>
<td>value3</td>
</tr>
</tbody>
</table>

### Examples of 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 from before:

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

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

<table>
<thead>
<tr>
<th>key1</th>
<th>key2</th>
</tr>
</thead>
<tbody>
<tr>
<td>value1</td>
<td>value1</td>
</tr>
<tr>
<td>value2</td>
<td>value2</td>
</tr>
<tr>
<td>value3</td>
<td>value3</td>
</tr>
</tbody>
</table>
When `records = false`, the JSON extension will not unpack the top-level objects, and create STRUCTs instead:

```sql
SELECT * FROM read_json_auto('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:

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

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

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

**Writing**

The contents of tables or the result of queries can be written directly to a JSON file using the COPY statement. See the COPY documentation for more information.

**read_json_auto Function**

The read_json_auto 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.
SELECT * FROM read_json_auto('todos.json') LIMIT 5;

<table>
<thead>
<tr>
<th>user_id</th>
<th>id</th>
<th>title</th>
<th>completed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>delectus aut autem</td>
<td>false</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>quis ut nam facilis et officia qui</td>
<td>false</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>fugiat veniam minus</td>
<td>false</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>et porro tempora</td>
<td>true</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>laboriosam mollitia et enim quasi adipisci quia provident illum</td>
<td>false</td>
</tr>
</tbody>
</table>

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

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

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

<table>
<thead>
<tr>
<th>column_name</th>
<th>column_type</th>
<th>null</th>
<th>key</th>
<th>default</th>
<th>extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>user_id</td>
<td>UBIGINT</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>id</td>
<td>UBIGINT</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>title</td>
<td>VARCHAR</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>completed</td>
<td>BOOLEAN</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If we specify the columns, we can bypass the automatic detection. Note that not all columns need to be specified:

SELECT *
FROM read_json_auto('todos.json',
columns = {user_id: 'UBIGINT',
           completed: 'BOOLEAN'});

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.

**COPY Statement**

The COPY statement can be used to load data from a JSON file into a table. For the COPY statement, we must first create a table with the correct schema to load the data into. We then specify the JSON file to load from plus any configuration options separately.
CREATE TABLE todos (userId BIGINT, id BIGINT, title VARCHAR, completed BOOLEAN);
COPY todos FROM 'todos.json';
SELECT * FROM todos LIMIT 5;

<table>
<thead>
<tr>
<th>userid</th>
<th>id</th>
<th>title</th>
<th>completed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>delectus aut autem</td>
<td>false</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>quis ut nam facilis et officia qui</td>
<td>false</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>fugiat veniam minus</td>
<td>false</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>et porro tempora</td>
<td>true</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>laboriosam mollitia et enim quasi adipisci quia provident illum</td>
<td>false</td>
</tr>
</tbody>
</table>

For more details, see the page on the COPY statement.

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';
DuckDB Documentation

-- read 3 Parquet files and treat them as a single table
SELECT * FROM read_parquet(['file1.parquet', 'file2.parquet', 'file3.parquet']);

-- Read all Parquet files from 2 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 3 Parquet files and treat them as a single table
SELECT * FROM read_parquet(['file1.parquet', 'file2.parquet', 'file3.parquet']);

**Glob Syntax**  Any filename 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.

<table>
<thead>
<tr>
<th>Wildcard</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>matches any number of any characters (including none)</td>
</tr>
<tr>
<td>**</td>
<td>matches any number of subdirectories (including none)</td>
</tr>
<tr>
<td>?</td>
<td>matches any single character</td>
</tr>
<tr>
<td>[abc]</td>
<td>matches one character given in the bracket</td>
</tr>
<tr>
<td>[a-z]</td>
<td>matches one character from the range given in the bracket</td>
</tr>
</tbody>
</table>

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:

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

<table>
<thead>
<tr>
<th>FlightDate</th>
<th>OriginCityName</th>
<th>DestCityName</th>
<th>UniqueCarrier</th>
<th>filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>1988-01-01</td>
<td>New York, NY</td>
<td>Los Angeles, CA</td>
<td>NULL</td>
<td>flights1.csv</td>
</tr>
<tr>
<td>1988-01-02</td>
<td>New York, NY</td>
<td>Los Angeles, CA</td>
<td>NULL</td>
<td>flights1.csv</td>
</tr>
<tr>
<td>1988-01-03</td>
<td>New York, NY</td>
<td>Los Angeles, CA</td>
<td>AA</td>
<td>flights2.csv</td>
</tr>
</tbody>
</table>

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).

```sql
-- Search the current directory for all files
SELECT * FROM glob('*/');
```

<table>
<thead>
<tr>
<th>file</th>
</tr>
</thead>
<tbody>
<tr>
<td>duckdb.exe</td>
</tr>
<tr>
<td>test.csv</td>
</tr>
<tr>
<td>test.json</td>
</tr>
<tr>
<td>test.parquet</td>
</tr>
<tr>
<td>test2.csv</td>
</tr>
<tr>
<td>test2.parquet</td>
</tr>
<tr>
<td>todos.json</td>
</tr>
</tbody>
</table>

Combining Schemas

Examples

-- read a set of CSV files combining columns by position
```sql
SELECT * FROM read_csv('flights*.csv');
```

-- read a set of CSV files combining columns by name
```sql
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:

<table>
<thead>
<tr>
<th>FlightDate</th>
<th>UniqueCarrier</th>
<th>OriginCityName</th>
<th>DestCityName</th>
</tr>
</thead>
<tbody>
<tr>
<td>1988-01-01</td>
<td>AA</td>
<td>New York, NY</td>
<td>Los Angeles, CA</td>
</tr>
<tr>
<td>1988-01-02</td>
<td>AA</td>
<td>New York, NY</td>
<td>Los Angeles, CA</td>
</tr>
<tr>
<td>1988-01-03</td>
<td>AA</td>
<td>New York, NY</td>
<td>Los Angeles, CA</td>
</tr>
</tbody>
</table>

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.

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

<table>
<thead>
<tr>
<th>FlightDate</th>
<th>OriginCityName</th>
<th>DestCityName</th>
<th>UniqueCarrier</th>
</tr>
</thead>
<tbody>
<tr>
<td>1988-01-01</td>
<td>New York, NY</td>
<td>Los Angeles, CA</td>
<td>NULL</td>
</tr>
<tr>
<td>1988-01-02</td>
<td>New York, NY</td>
<td>Los Angeles, CA</td>
<td>NULL</td>
</tr>
<tr>
<td>1988-01-03</td>
<td>New York, NY</td>
<td>Los Angeles, CA</td>
<td>AA</td>
</tr>
</tbody>
</table>

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

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 3 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 a "filename" column that specifies which file each row came from
DuckDB Documentation

```sql
SELECT * FROM read_parquet('test/*.parquet', filename = true);
-- use a list of globs to read all Parquet files from 2 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 schema of a Parquet file
SELECT * FROM parquet_schema('test.parquet');

-- write the results of a query to a Parquet file
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
COPY (FROM generate_series(100000)) TO 'test.parquet' (FORMAT 'parquet', COMPRESSION 'ZSTD', ROW_GROUP_SIZE 100000);
-- export the table contents of the entire database as parquet
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.

**read_parquet Function**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>read_parquet(path(s), *)</td>
<td>Read Parquet file(s)</td>
<td>SELECT * FROM read_parquet('test.parquet');</td>
</tr>
<tr>
<td>parquet_scan(path(s), *)</td>
<td>Alias for read_parquet</td>
<td>SELECT * FROM parquet_scan('test.parquet');</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Type</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary_as_string</td>
<td>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.</td>
<td>BOOL</td>
<td>false</td>
</tr>
<tr>
<td>filename</td>
<td>Whether or not an extra filename column should be included in the result.</td>
<td>BOOL</td>
<td>false</td>
</tr>
<tr>
<td>file_row_number</td>
<td>Whether or not to include the file_row_number column.</td>
<td>BOOL</td>
<td>false</td>
</tr>
<tr>
<td>hive_partitioning</td>
<td>Whether or not to interpret the path as a hive partitioned path.</td>
<td>BOOL</td>
<td>false</td>
</tr>
<tr>
<td>union_by_name</td>
<td>Whether the columns of multiple schemas should be unified by name, rather than by position.</td>
<td>BOOL</td>
<td>false</td>
</tr>
</tbody>
</table>

**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 on this 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.

```sql
-- 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');
```
DuckDB Documentation

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.

```sql
-- 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.

```sql
-- 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', CODEC 'ZSTD');
-- write a CSV file to an uncompressed Parquet file
COPY 'test.csv' TO 'result-uncompressed.parquet' (FORMAT 'PARQUET', CODEC 'UNCOMPRESSED');

-- write a query to a Parquet file with ZSTD compression (same as CODEC) and row_group_size
COPY (FROM generate_series(100000)) TO 'row-groups-zstd.parquet' (FORMAT PARQUET, COMPRESSION ZSTD, ROW_GROUP_SIZE 100000);
```

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

```sql
-- export the table contents of the entire database as parquet
EXPORT DATABASE 'target_directory' (FORMAT PARQUET);
```

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 and loaded separately.

```sql
-- run once
INSTALL parquet;
-- run before usage
LOAD 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.

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

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>file_name</code></td>
<td>VARCHAR</td>
</tr>
<tr>
<td><code>row_group_id</code></td>
<td>BIGINT</td>
</tr>
<tr>
<td><code>row_group_num_rows</code></td>
<td>BIGINT</td>
</tr>
<tr>
<td><code>row_group_num_columns</code></td>
<td>BIGINT</td>
</tr>
<tr>
<td><code>row_group_bytes</code></td>
<td>BIGINT</td>
</tr>
<tr>
<td><code>column_id</code></td>
<td>BIGINT</td>
</tr>
<tr>
<td><code>file_offset</code></td>
<td>BIGINT</td>
</tr>
<tr>
<td><code>num_values</code></td>
<td>BIGINT</td>
</tr>
<tr>
<td><code>path_in_schema</code></td>
<td>VARCHAR</td>
</tr>
<tr>
<td><code>type</code></td>
<td>VARCHAR</td>
</tr>
<tr>
<td><code>stats_min</code></td>
<td>VARCHAR</td>
</tr>
<tr>
<td><code>stats_max</code></td>
<td>VARCHAR</td>
</tr>
<tr>
<td><code>stats_null_count</code></td>
<td>BIGINT</td>
</tr>
<tr>
<td><code>stats_distinct_count</code></td>
<td>BIGINT</td>
</tr>
<tr>
<td><code>stats_min_value</code></td>
<td>VARCHAR</td>
</tr>
<tr>
<td><code>stats_max_value</code></td>
<td>VARCHAR</td>
</tr>
<tr>
<td><code>compression</code></td>
<td>VARCHAR</td>
</tr>
<tr>
<td><code>encodings</code></td>
<td>VARCHAR</td>
</tr>
<tr>
<td><code>index_page_offset</code></td>
<td>BIGINT</td>
</tr>
<tr>
<td><code>dictionary_page_offset</code></td>
<td>BIGINT</td>
</tr>
<tr>
<td><code>data_page_offset</code></td>
<td>BIGINT</td>
</tr>
<tr>
<td><code>total_compressed_size</code></td>
<td>BIGINT</td>
</tr>
<tr>
<td><code>total_uncompressed_size</code></td>
<td>BIGINT</td>
</tr>
</tbody>
</table>
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`.

```sql
-- 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`.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>file_name</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>name</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>type</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>type_length</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>repetition_type</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>num_children</td>
<td>BIGINT</td>
</tr>
<tr>
<td>converted_type</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>scale</td>
<td>BIGINT</td>
</tr>
<tr>
<td>precision</td>
<td>BIGINT</td>
</tr>
<tr>
<td>field_id</td>
<td>BIGINT</td>
</tr>
<tr>
<td>logical_type</td>
<td>VARCHAR</td>
</tr>
</tbody>
</table>

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.

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

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>key_value_metadata</td>
<td>MAP(BLOB, BLOB)</td>
</tr>
</tbody>
</table>
### Field Type

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>file_name</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>created_by</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>num_rows</td>
<td>BIGINT</td>
</tr>
<tr>
<td>num_row_groups</td>
<td>BIGINT</td>
</tr>
<tr>
<td>format_version</td>
<td>BIGINT</td>
</tr>
<tr>
<td>encryption_algorithm</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>footer_signing_key_metadata</td>
<td>VARCHAR</td>
</tr>
</tbody>
</table>

## Parquet Key-Value Metadata

The `parquet_kv_metadata` function can be used to query custom metadata defined as key-value pairs.

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

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>file_name</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>key</td>
<td>BLOB</td>
</tr>
<tr>
<td>value</td>
<td>BLOB</td>
</tr>
</tbody>
</table>

## 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.

```sql
SELECT * FROM read_parquet('flights*.parquet', union_by_name = true);
```

### Tips for Writing Parquet files

**Enabling PER_THREAD_OUTPUT**

If the final number of Parquet files is not important, writing one file per thread can significantly improve performance. Using a glob pattern upon read or a hive partitioning structure are good ways to transparently handle multiple files.
COPY (FROM generate_series(10000000)) TO 'test.parquet' (FORMAT PARQUET, PER_THREAD_OUTPUT true);

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 (currently 2048, but adjustable when compiling DuckDB), 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.

-- write a query to a Parquet file with a different row_group_size
COPY (FROM generate_series(1000000)) TO 'row-groups.parquet' (FORMAT PARQUET, ROW_GROUP_SIZE 100000);

Partitioning

Hive Partitioning

Examples

-- read data from a hive partitioned data set
SELECT * FROM read_parquet('orders/*/*/*.parquet', hive_partitioning = 1);
-- parquet_scan is an alias of read_parquet, so they are equivalent
SELECT * FROM parquet_scan('orders/*/*/*.parquet', hive_partitioning = 1);
-- write a table to a hive partitioned data set
COPY orders TO 'orders' (FORMAT PARQUET, PARTITION_BY (year, month));

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
Files stored in this hierarchy can be read using the `hive_partitioning` flag.

```
SELECT * FROM read_parquet( 'orders/*/*/*.parquet', hive_partitioning = 1);
```

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:

```
SELECT *
FROM read_parquet( 'orders/*/*/*.parquet', hive_partitioning = 1)
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 behaviour can be overridden by setting the `hive_partitioning` flag manually.

**Hive Types**  `hive_types` is a way to specify the logical types of the hive partitions in a struct:

```
FROM read_parquet( 'dir/**/parquet', hive_partitioning = 1, 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**

```sql
-- 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 1);
```

**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 many partitions as many files will be created. The ideal partition count depends on how large your data set is.

**Note.** Writing data into many small partitions is expensive. It is generally recommended to have at least 100MB of data per partition.

**Overwriting** By default the partitioned write will not allow overwriting existing directories. Use the `OVERWRITE_OR_IGNORE` option to allow overwriting an existing directory.

**Filename Pattern** By default, files will be named `data_{i}.parquet` or `data_{i}.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}");

Appender

The C++ Appender can be used to load bulk data into a DuckDB database. 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.

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.
DuckDB Documentation

Below is a short example:

```sql
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
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));

**JDBC Appender**

The appender is available in the **JDBC** driver.

**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.

**Note.** 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.

**Note.** 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:

```sql
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 APIs Overview

There are various client APIs for DuckDB:

- C
- C++
- Java
- Julia
- Node.js
- Python
- R
- Rust
- WebAssembly/Wasm
- ADBC API
- ODBC API

Additionally, there is a standalone Command Line Interface (CLI) client.

There are also contributed third-party DuckDB wrappers for:

- C# by Giorgi
- Common Lisp by ak-coram
- Crystal by amauryt
- Go by marcoeker
- Ruby by suketa
- Zig by karlseguin

C

C API - Overview

DuckDB implements a custom C API modelled 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.

C API - 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

```c
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

```c
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);
const char *duckdb_library_version();

**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 instantiated database should be closed with 'duckdb_close'

**Syntax**

```c
duckdb_state duckdb_open(
    const char *path,
    duckdb_database *out_database
);
```

**Parameters**

• path

Path to the database file on disk, or nullptr or :memory: to open an in-memory database.

• out_database

The result database object.

• returns

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.

**Syntax**

```c
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, or nullptr or :memory: to open an in-memory database.

- out_database
  The result database object.

- config
  (Optional) configuration used to start up the database system.

- out_error
  If set and the function returns DuckDBError, this will contain the reason why the start-up failed. Note that the error must be freed using duckdb_free.

- returns
  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. 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

```c
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

```c
duckdb_state duckdb_connect(
  duckdb_database database,
  duckdb_connection *out_connection
);
```
DuckDB Documentation

Parameters
- database
  The database file to connect to.
- out_connection
  The result connection object.
- returns
  DuckDBSuccess on success or DuckDBError on failure.

**duckdb_interrupt**  Interrupt running query

Syntax
```c
void duckdb_interrupt(
    duckdb_connection connection
);
```

Parameters
- connection
  The connection to interrupt

**duckdb_query_progress**  Get progress of the running query

Syntax
```c
duckdb_query_progress_type duckdb_query_progress(
    duckdb_connection connection
);
```

Parameters
- connection
  The working connection
- returns
  -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

```c
void duckdb_disconnect(
    duckdb_connection *connection
);
```

**Parameters**

- `connection`

  The connection to close.

**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

```c
const char *duckdb_library_version();
```

**C API - 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**

```c
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**

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.
This will always succeed unless there is a malloc failure.

**Syntax**

duckdb_state duckdb_create_config(
    duckdb_config *out_config
);

**Parameters**

• out_config
The result configuration object.

• returns
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.
Syntax

size_t duckdb_config_count();

Parameters

• returns

The amount of config options available.

**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., \( \geq \) 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.

• returns

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.
DuckDB Documentation

Syntax

duckdb_state duckdb_set_config(
    duckdb_config config,
    const char *name,
    const char *option
);

Parameters

• duckdb_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.

• returns
DuckDBSuccess on success or DuckDBError on failure.

duckdb_destroy_config Destroys the specified configuration option and de-allocates all memory allocated for the object.

Syntax

void duckdb_destroy_config(
    duckdb_config *config
);

Parameters

• config
The configuration object to destroy.

C API - 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` and `duckdb_row_count` methods can be used to extract the number of columns and the number of rows, respectively. `duckdb_column_name` and `duckdb_column_type` can be used to extract the names and types of individual columns.

**Example**

```c
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_column_data`/`duckdb_nullmask_data` functions, or using the `duckdb_value` convenience functions. The `duckdb_column_data`/`duckdb_nullmask_data` data functions directly hand you a pointer to the result arrays in columnar 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.

**Note.** For optimal performance, use `duckdb_column_data` and `duckdb_nullmask_data` to extract data from the query result. The `duckdb_value` functions perform internal type-checking, bounds-checking and casting which makes them slower.
**DuckDB Documentation**

**`duckdb_value`**  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.

```c
// 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");
}
```

**`duckdb_column_data`**  Below is an example that prints the above result to CSV format using the `duckdb_column_data` function. Note that the function is NOT generic: we do need to know exactly what the types of the result columns are.

```c
int32_t *i_data = (int32_t *) duckdb_column_data(&result, 0);
int32_t *j_data = (int32_t *) duckdb_column_data(&result, 1);
bool  *i_mask = duckdb_nullmask_data(&result, 0);
bool  *j_mask = duckdb_nullmask_data(&result, 1);
idx_t row_count = duckdb_row_count(&result);
for(idx_t row = 0; row < row_count; row++) {
    if (i_mask[row]) {
        printf("NULL");
    } else {
        printf("%d", i_data[row]);
    }
    printf(",");
    if (j_mask[row]) {
        printf("NULL");
    } else {
        printf("%d", j_data[row]);
    }
    printf("\n");
}
```

**Note.**  When using `duckdb_column_data`, be careful that the type matches exactly what you expect it to be. As the code directly accesses an internal array, there is no type-checking. Accessing a DUCKDB_TYPE_INTEGER column as if it was a DUCKDB_TYPE_BIGINT column will provide unpredictable results!
API Reference

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_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.

• returns

DuckDBSuccess on success or DuckDBError on failure.
**duckdb_destroy_result**  Closes the result and de-allocates all memory allocated for that connection.

**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 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.
- `returns`
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.
- returns
  The column type of the specified column.

**duckdb_result_statement_type** Returns the statement type of the statement that was executed

Syntax

```c
duckdb_statement_type duckdb_result_statement_type(
    duckdb_result result
);
```

Parameters

- result
  The result object to fetch the statement type from.
- returns

  *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

```c
duckdb_logical_type duckdb_column_logical_type(
    duckdb_result *result,
    idx_t col
);
```
**DuckDB Documentation**

**Parameters**

- **result**
  The result object to fetch the column type from.

- **col**
  The column index.

- **returns**
  The logical column type of the specified column.

**duckdb_column_count** Returns the number of columns present in a the result object.

**Syntax**

```c
idx_t duckdb_column_count(
    duckdb_result *result
);
```

**Parameters**

- **result**
  The result object.

- **returns**
  The number of columns present in the result object.

**duckdb_row_count** Returns the number of rows present in a the result object.

**Syntax**

```c
idx_t duckdb_row_count(
    duckdb_result *result
);
```

**Parameters**

- **result**
  The result object.

- **returns**
  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**

```c
idx_t duckdb_rows_changed(
    duckdb_result *result
);
```

**Parameters**

- **result**
The result object.
- **returns**
The number of rows changed.

**duckdb_column_data**  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:

```c
int32_t *data = (int32_t *) duckdb_column_data(&result, 0);
printf("Data for row %d: %d\n", row, data[row]);
```

**Syntax**

```c
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.
- **returns**
The column data of the specified column.
**duckdb_nullmask_data**  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.

```c
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**

```c
bool *duckdb_nullmask_data(
    duckdb_result *result,
    idx_t col
);
```

**Parameters**

- **result**
  The result object to fetch the nullmask from.

- **col**
  The column index.

- **returns**
  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**

```c
const char *duckdb_result_error(
    duckdb_result *result
);
```
Parameters

- result

The result object to fetch the error from.

- returns

The error of the result.

C API - 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.

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, and subsequently using the duckdb_vector_get_data and duckdb_vector_get_validity methods to read the internal data and the validity mask of the vector. For composite types (list and struct vectors), duckdb_list_vector_get_child and duckdb_struct_vector_get_child should be used to read child vectors.

API Reference

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_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);

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_data_chunk Creates an empty DataChunk with the specified set of types.

Syntax

duckdb_data_chunk duckdb_create_data_chunk(
    duckdb_logical_type *types,
    idx_t column_count
);

Parameters

• types
An array of types of the data chunk.

• column_count
The number of columns.

• returns
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.

Syntax

```c
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

```c
idx_t duckdb_data_chunk_get_column_count(
    duckdb_data_chunk chunk
);
```

Parameters

- chunk

The data chunk to get the data from

  - returns

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.
DuckDB Documentation

Syntax

```c
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

- `returns`

The vector

---

```c
idx_t duckdb_data_chunk_get_size(
    duckdb_data_chunk chunk
);
```

Parameters

- `chunk`

The data chunk to get the data from

- `returns`

The number of tuples in the data chunk

---

```c
void duckdb_data_chunk_set_size(
    duckdb_data_chunk chunk,
    idx_t size
);
```

---

**duckdb_data_chunk_get_size**  Retrieves the current number of tuples in a data chunk.

**duckdb_data_chunk_set_size**  Sets the current number of tuples in a data chunk.
**Parameters**

- `chunk`

  The data chunk to set the size in

- `size`

  The number of tuples in the data chunk

**duckdb_vector_get_column_type**  Retrieves the column type of the specified vector.

The result must be destroyed with `duckdb_destroy_logical_type`.

**Syntax**

```c
duckdb_logical_type duckdb_vector_get_column_type(
    duckdb_vector vector
);
```

**Parameters**

- `vector`

  The vector to get the data from

- `returns`

  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**

```c
void *duckdb_vector_get_data(
    duckdb_vector vector
);
```

**Parameters**

- `vector`

  The vector to get the data from

- `returns`

  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:

```c
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**

```c
uint64_t *duckdb_vector_get_validity(  
    duckdb_vector vector
);
```

**Parameters**

- **vector**
  The vector to get the data from

- **returns**
  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**

```c
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**

```c
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.

**Syntax**

```c
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 null-terminated string
DuckDB Documentation

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**

```c
duckdb_vector duckdb_list_vector_get_child(
    duckdb_vector vector
);
```

**Parameters**

• **vector**

The vector

• **returns**

The child vector

**duckdb_list_vector_get_size**  Returns the size of the child vector of the list

**Syntax**

```c
idx_t duckdb_list_vector_get_size(
    duckdb_vector vector
);
```

**Parameters**

• **vector**

The vector

• **returns**

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_list_vector_set_size`

```c
duckdb_list_vector_set_size(duckdb_state 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.

- **returns**
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.

Syntax

```c
duckdb_list_vector_reserve(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**
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.
DuckDB Documentation

**Syntax**

```c
duckdb_vector duckdb_struct_vector_get_child(
    duckdb_vector vector,
    idx_t index
);
```

**Parameters**

- **vector**
The vector
- **index**
The child index
- **returns**
The child vector

**duckdb_validity_row_is_valid** Returns whether or not a row is valid (i.e., not NULL) in the given validity mask.

**Syntax**

```c
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
- **returns**
  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.
**DuckDB Documentation**

**Syntax**

```c
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**

```c
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.
DuckDB Documentation

**Syntax**

```c
void duckdb_validity_set_row_valid(
    uint64_t *validity,
    idx_t row
);
```

**Parameters**

- `validity`
  The validity mask
- `row`
  The row index

**C API - Values**

The value class represents a single value of any type.

**API Reference**

```c
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_int64(int64_t val);
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);
char *duckdb_get_varchar(duckdb_value value);
int64_t duckdb_get_int64(duckdb_value value);
```

**duckdb_destroy_value**  Destroys the value and de-allocates all memory allocated for that type.

**Syntax**

```c
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
   • value
The null-terminated string
   • returns
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
   • value
The text
   • length
The length of the text
   • returns
The value. This must be destroyed with duckdb_destroy_value.

duckdb_create_int64  Creates a value from an int64

Syntax

duckdb_value duckdb_create_int64(
   int64_t val
);
Parameters

• value

The bigint value

• returns

The value. This must be destroyed with duckdb_destroy_value.

duckdb_create_struct_value   Creates a struct value from a type and an array of values

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

• returns

The value. This must be destroyed with duckdb_destroy_value.

duckdb_create_list_value   Creates a list value from a type and an array of values of length value_count

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

- **returns**
  The value. This must be destroyed with `duckdb_destroy_value`.

**duckdb_get_vARCHAR**  Obtains a string representation of the given value. The result must be destroyed with `duckdb_free`.

**Syntax**

```c
char *duckdb_get_vARCHAR(
    duckdb_value value
);
```

**Parameters**

- **value**
  The value

- **returns**
  The string value. This must be destroyed with `duckdb_free`.

**duckdb_get_int64**  Obtains an int64 of the given value.

**Syntax**

```c
int64_t duckdb_get_int64(
    duckdb_value value
);
```
Parameters

- value

The value

- returns

The int64 value, or 0 if no conversion is possible

C API - 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 (INT []). 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.

```c
typedef enum DUCKDB_TYPE {
    DUCKDB_Type_Invalid,
    DUCKDB_Type_Boolean,
    DUCKDB_Type_Tinyint,
    DUCKDB_Type_Smallint,
    DUCKDB_Type_Integer,
    DUCKDB_Type_Bigint,
    DUCKDB_Type_Utinyint,
    DUCKDB_Type_Usmallint,
    DUCKDB_Type_Uinteger,
    DUCKDB_Type_UBigint,
    DUCKDB_Type_Float,
    DUCKDB_Type_Double,
    DUCKDB_Type_Timestamp,
    DUCKDB_Type_Date,
    DUCKDB_Type_Time,
    DUCKDB_Type_Interval,
    DUCKDB_Type_Hugeint,
    DUCKDB_Type_VARCHAR,
    DUCKDB_Type_BLOB,
    DUCKDB_Type_Decimal,
    DUCKDB_Type_TIMESTAMP_S,
    DUCKDB_Type_TIMESTAMP_MS,
    DUCKDB_Type_TIMESTAMP_NS,
    DUCKDB_Type_ENUM,
```
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.

Note. Note that `duckdb_value_varchar` and `duckdb_value_blob` require the result to be deallocated using `duckdb_free`.

`duckdb_result_get_chunk` The `duckdb_result_get_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

- `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);`
- `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);
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);

Date/Time/Timestamp Helpers

duckdb_date_struct duckdb_from_date(duckdb_date date);
duckdb_date duckdb_to_date(duckdb_date_struct date);
duckdb_time_struct duckdb_from_time(duckdb_time time);
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);

Hugeint Helpers

double duckdb_hugeint_to_double(duckdb_hugeint val);
duckdb_hugeint duckdb_double_to_hugeint(double val);
duckdb_decimal duckdb_double_to_decimal(double val, uint8_t width, uint8_t scale);

Decimal Helpers

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);
duckdb_logical_type duckdb_create_list_type(duckdb_logical_type type);
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);
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_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_result_get_chunk_  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,
duckdb_result_is_streaming  Checks if the type of the internal result is StreamQueryResult.

Syntax

```c
bool duckdb_result_is_streaming(
    duckdb_result result
);
```

Parameters

- `result`
  The result object to check.

  Returns

  Whether or not the result object is of the type StreamQueryResult

duckdb_result_chunk_count  Returns the number of data chunks present in the result.

Syntax

```c
idx_t duckdb_result_chunk_count(
    duckdb_result result
);
```
Parameters

• result
The result object

• returns
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

• returns
The return_type

duckdb_value_boolean

Syntax

bool duckdb_value_boolean(
   duckdb_result *result,
   idx_t col,
   idx_t row
);

Parameters

• returns
The boolean value at the specified location, or false if the value cannot be converted.

duckdb_value_int8

82
DuckDB Documentation

Syntax

```c
int8_t duckdb_value_int8(
    duckdb_result *result,
    idx_t col,
    idx_t row
);
```

Parameters

- returns

The int8_t value at the specified location, or 0 if the value cannot be converted.

**duckdb_value_int16**

Syntax

```c
int16_t duckdb_value_int16(
    duckdb_result *result,
    idx_t col,
    idx_t row
);
```

Parameters

- returns

The int16_t value at the specified location, or 0 if the value cannot be converted.

**duckdb_value_int32**

Syntax

```c
int32_t duckdb_value_int32(
    duckdb_result *result,
    idx_t col,
    idx_t row
);
```

Parameters

- returns

The int32_t value at the specified location, or 0 if the value cannot be converted.
duckdb_value_int64

**Syntax**

```c
int64_t duckdb_value_int64(
    duckdb_result *result,
    idx_t col,
    idx_t row
);
```

**Parameters**

- **returns**

  The int64_t value at the specified location, or 0 if the value cannot be converted.

---

duckdb_value_hugeint

**Syntax**

```c
duckdb_hugeint duckdb_value_hugeint(
    duckdb_result *result,
    idx_t col,
    idx_t row
);
```

**Parameters**

- **returns**

  The duckdb_hugeint value at the specified location, or 0 if the value cannot be converted.

---

duckdb_value_uhugeint

**Syntax**

```c
duckdb_uhugeint duckdb_value_uhugeint(
    duckdb_result *result,
    idx_t col,
    idx_t row
);
```

**Parameters**

- **returns**

  The duckdb_uhugeint value at the specified location, or 0 if the value cannot be converted.
**duckdb_value_decimal**

**Syntax**

```c
duckdb_decimal duckdb_value_decimal(
    duckdb_result *result,
    idx_t col,
    idx_t row
);
```

**Parameters**

- **returns**

The `duckdb_decimal` value at the specified location, or 0 if the value cannot be converted.

**duckdb_value_uint8**

**Syntax**

```c
uint8_t duckdb_value_uint8(
    duckdb_result *result,
    idx_t col,
    idx_t row
);
```

**Parameters**

- **returns**

The `uint8_t` value at the specified location, or 0 if the value cannot be converted.

**duckdb_value_uint16**

**Syntax**

```c
uint16_t duckdb_value_uint16(
    duckdb_result *result,
    idx_t col,
    idx_t row
);
```

**Parameters**

- **returns**

The `uint16_t` value at the specified location, or 0 if the value cannot be converted.
**duckdb_value_uint32**

**Syntax**

```c
uint32_t duckdb_value_uint32(
    duckdb_result *result,
    idx_t col,
    idx_t row
);
```

**Parameters**
- returns

The uint32_t value at the specified location, or 0 if the value cannot be converted.

**duckdb_value_uint64**

**Syntax**

```c
uint64_t duckdb_value_uint64(
    duckdb_result *result,
    idx_t col,
    idx_t row
);
```

**Parameters**
- returns

The uint64_t value at the specified location, or 0 if the value cannot be converted.

**duckdb_value_float**

**Syntax**

```c
float duckdb_value_float(
    duckdb_result *result,
    idx_t col,
    idx_t row
);
```

**Parameters**
- returns

The float value at the specified location, or 0 if the value cannot be converted.
**duckdb_value_double**

**Syntax**

double duckdb_value_double(
    duckdb_result *result,
    idx_t col,
    idx_t row
);

**Parameters**

- returns

The double value at the specified location, or 0 if the value cannot be converted.

**duckdb_value_date**

**Syntax**

duckdb_date duckdb_value_date(
    duckdb_result *result,
    idx_t col,
    idx_t row
);

**Parameters**

- returns

The duckdb_date value at the specified location, or 0 if the value cannot be converted.

**duckdb_value_time**

**Syntax**

duckdb_time duckdb_value_time(
    duckdb_result *result,
    idx_t col,
    idx_t row
);

**Parameters**

- returns

The duckdb_time value at the specified location, or 0 if the value cannot be converted.
**duckdb_value_timestamp**

**Syntax**

```c
duckdb_timestamp duckdb_value_timestamp(
    duckdb_result *result,
    idx_t col,
    idx_t row
);
```

**Parameters**

- **returns**

The `duckdb_timestamp` value at the specified location, or 0 if the value cannot be converted.

**duckdb_value_interval**

**Syntax**

```c
duckdb_interval duckdb_value_interval(
    duckdb_result *result,
    idx_t col,
    idx_t row
);
```

**Parameters**

- **returns**

The `duckdb_interval` value at the specified location, or 0 if the value cannot be converted.

**duckdb_value_varchar**

**Syntax**

```c
char *duckdb_value_varchar(
    duckdb_result *result,
    idx_t col,
    idx_t row
);
```
Parameters

• DEPRECATED

use duckdb_value_string instead. This function does not work correctly if the string contains null bytes.

• returns

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.

duckdb_value_varchar_internal

Syntax

char *duckdb_value_varchar_internal(
   duckdb_result *result,
   idx_t col,
   idx_t row
);

Parameters

• DEPRECATED

use duckdb_value_string_internal instead. This function does not work correctly if the string contains null bytes.

• returns

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.

duckdb_value_string_internal

Syntax

duckdb_string duckdb_value_string_internal(
   duckdb_result *result,
   idx_t col,
   idx_t row
);
Parameters

• DEPRECATED
use duckdb_value_string_internal instead. This function does not work correctly if the string contains null bytes.

• returns
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.

**duckdb_value_blob**

Syntax

```c
duckdb_blob duckdb_value_blob(  
  duckdb_result *result,  
  idx_t col,  
  idx_t row
);
```

Parameters

• returns
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 "blob.data" must be freed with duckdb_free.

**duckdb_value_is_null**

Syntax

```c
bool duckdb_value_is_null(  
  duckdb_result *result,  
  idx_t col,  
  idx_t row
);
```

Parameters

• returns
Returns true if the value at the specified index is NULL, and false otherwise.
**duckdb_from_date**  Decompose a `duckdb_date` object into year, month and date (stored as `duckdb_date_struct`).

**Syntax**

```c
duckdb_date_struct duckdb_from_date(
    duckdb_date date
);
```

**Parameters**

- **date**
  
  The date object, as obtained from a DUCKDB_TYPE_DATE column.

- **returns**
  
  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**

```c
duckdb_date duckdb_to_date(
    duckdb_date_struct date
);
```

**Parameters**

- **date**
  
  The year, month and date stored in a `duckdb_date_struct`.

- **returns**
  
  The `duckdb_date` element.

**duckdb_from_time**  Decompose a `duckdb_time` object into hour, minute, second and microsecond (stored as `duckdb_time_struct`).

**Syntax**

```c
duckdb_time_struct duckdb_from_time(
    duckdb_time time
);
```
**Parameters**

- `time`

The time object, as obtained from a DUCKDB_TYPE_TIME column.

- `returns`

The `duckdb_time_struct` with the decomposed elements.

---

**`duckdb_to_time`**  Re-compose a `duckdb_time` from hour, minute, second and microsecond (`duckdb_time_struct`).

**Syntax**

```c
duckdb_time duckdb_to_time(
    duckdb_time_struct time
);
```

**Parameters**

- `time`

The hour, minute, second and microsecond in a `duckdb_time_struct`.

- `returns`

The `duckdb_time` element.

---

**`duckdb_from_timestamp`**  Decompose a `duckdb_timestamp` object into a `duckdb_timestamp_struct`.

**Syntax**

```c
duckdb_timestamp_struct duckdb_from_timestamp(
    duckdb_timestamp ts
);
```

**Parameters**

- `ts`

The `ts` object, as obtained from a DUCKDB_TYPE_TIMESTAMP column.

- `returns`

The `duckdb_timestamp_struct` with the decomposed elements.
**duckdb_to_timestamp**  Re-compose a `duckdb_timestamp` from a `duckdb_timestamp_struct`.

**Syntax**

```c
duckdb_timestamp duckdb_to_timestamp(
    duckdb_timestamp_struct ts
);
```

**Parameters**

- *ts*
  The de-composed elements in a `duckdb_timestamp_struct`.
- *returns*
  The `duckdb_timestamp` element.

**duckdb_hugeint_to_double**  Converts a `duckdb_hugeint` object (as obtained from a `DUCKDB_TYPE_HUGEINT` column) into a double.

**Syntax**

```c
double duckdb_hugeint_to_double(
    duckdb_hugeint val
);
```

**Parameters**

- *val*
  The hugeint value.
- *returns*
  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**

```c
duckdb_hugeint duckdb_double_to_hugeint(
    double val
);
```
Parameters

• val
The double value.

• returns
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**

```c
duckdb_decimal duckdb_double_to_decimal(
    double val,
    uint8_t width,
    uint8_t scale
);
```

Parameters

• val
The double value.

• returns
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**

```c
double duckdb_decimal_to_double(
    duckdb_decimal val
);
```

Parameters

• val
The decimal value.

• returns
The converted double element.
**duckdb_create_logical_type**  Creates a `duckdb_logical_type` from a standard primitive type. The resulting type should be destroyed with `duckdb_destroy_logical_type`.

This should not be used with `DUCKDB_TYPE_DECIMAL`.

**Syntax**

```c
duckdb_logical_type duckdb_create_logical_type(    
    duckdb_type type
);
```

**Parameters**

- **type**
  The primitive type to create.
- **returns**
  The logical type.

**duckdb_logical_type_get_alias**  Returns the alias of a `duckdb_logical_type`, if one is set, else NULL. You must free the result.

**Syntax**

```c
char *duckdb_logical_type_get_alias(    
    duckdb_logical_type type
);
```

**Parameters**

- **type**
  The logical type to return the alias of
- **returns**
  The alias or NULL

**duckdb_create_list_type**  Creates a list type from its child type. The resulting type should be destroyed with `duckdb_destroy_logical_type`.

**Syntax**

```c
duckdb.logical_type duckdb_create_list_type(    
    duckdb.logical_type type
);
```
Parameters

• type
The child type of list type to create.

• returns
The logical type.

duckdb_create_map_type  Creates a map type from its key type and value type. The resulting type should 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

• type
The key type and value type of map type to create.

• returns
The logical type.

duckdb_create_union_type  Creates a UNION type from the passed types array The resulting type should 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

• types
The array of types that the union should consist of.

• type_amount
The size of the types array.

- **returns**

The logical type.

**duckdb_create_struct_type**  Creates a STRUCT type from the passed member name and type arrays. The resulting type should be destroyed with duckdb_destroy_logical_type.

**Syntax**

```c
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 that the struct should consist of.

- **member_names**
  The array of names that the struct should consist of.

- **member_count**
  The number of members that were specified for both arrays.

- **returns**
  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**

```c
duckdb_logical_type duckdb_create_enum_type(
    const char **member_names,
    idx_t member_count
);
```
Parameters

- enum_name
  The name of the enum.
- member_names
  The array of names that the enum should consist of.
- member_count
  The number of elements that were specified in the array.
- returns
  The logical type.

**duckdb_create_decimal_type**  Creates a duckdb_logical_type of type decimal with the specified width and scale. The resulting type should be destroyed with duckdb_destroy_logical_type.

**Syntax**

```c
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
- returns
  The logical type.

**duckdb_get_type_id**  Retrieves the type class of a duckdb_logical_type.

**Syntax**

```c
duckdb_type duckdb_get_type_id(
    duckdb_logical_type type
);
```
DuckDB Documentation

**Parameters**

- **type**
  The logical type object

- **returns**
  The type id

**duckdb_decimal_width**  Retrieves the width of a decimal type.

**Syntax**

```c
uint8_t duckdb_decimal_width(
    duckdb_logical_type type
);
```

**Parameters**

- **type**
  The logical type object

- **returns**
  The width of the decimal type

**duckdb_decimal_scale**  Retrieves the scale of a decimal type.

**Syntax**

```c
uint8_t duckdb_decimal_scale(
    duckdb_logical_type type
);
```

**Parameters**

- **type**
  The logical type object

- **returns**
  The scale of the decimal type

**duckdb_decimal_internal_type**  Retrieves the internal storage type of a decimal type.
**Syntax**

```c
duckdb_type duckdb_decimal_internal_type(
    duckdb_logical_type type
);
```

**Parameters**

- **type**
  The logical type object

- **returns**
  The internal type of the decimal type

**duckdb_enum_internal_type**  Retrieves the internal storage type of an enum type.

**Syntax**

```c
duckdb_type duckdb_enum_internal_type(
    duckdb_logical_type type
);
```

**Parameters**

- **type**
  The logical type object

- **returns**
  The internal type of the enum type

**duckdb_enum_dictionary_size**  Retrieves the dictionary size of the enum type

**Syntax**

```c
uint32_t duckdb_enum_dictionary_size(
    duckdb_logical_type type
);
```

**Parameters**

- **type**
  The logical type object

- **returns**
  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**

```c
char *duckdb_enum_dictionary_value(
    duckdb_logical_type type,
    idx_t index
);
```

**Parameters**

- `type`
  The logical type object

- `index`
  The index in the dictionary

- `returns`
  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.

The result must be freed with duckdb_destroy_logical_type

**Syntax**

```c
duckdb_logical_type duckdb_list_type_child_type(
    duckdb_logical_type type
);
```

**Parameters**

- `type`
  The logical type object

- `returns`
  The child type of the list type. Must be destroyed with duckdb_destroy_logical_type.

**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**

```c
duckdb_logical_type duckdb_map_type_key_type(
    duckdb_logical_type type
);
```

**Parameters**

- `type`  

The logical type object

- `returns`  

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**

```c
duckdb_logical_type duckdb_map_type_value_type(
    duckdb_logical_type type
);
```

**Parameters**

- `type`  

The logical type object

- `returns`  

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**

```c
idx_t duckdb_struct_type_child_count(
    duckdb_logical_type type
);
```

---

102
Parameters

• type

The logical type object

• returns

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

• returns

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

- returns
  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

```c
idx_t duckdb_union_type_member_count(
    duckdb_logical_type type
);
```

Parameters

- type
  The logical type (union) object

- returns
  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

```c
char *duckdb_union_type_member_name(
    duckdb_logical_type type,
    idx_t index
);
```

Parameters

- type
  The logical type object

- index
The child index

- returns

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**

```c
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

  - returns

  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**

```c
void duckdb_destroy_logical_type(
    duckdb_logical_type *type
);
```

**Parameters**

- type

  The logical type to destroy.
C API - 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

```c
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.

**Note.** Do not use prepared statements to insert large amounts of data into DuckDB. Instead it is recommended to use the Appender.

**API Reference**

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_state duckdb_clear_bindings(duckdb_prepared_statement prepared_statement);
duckdb_statement_type duckdb_prepared_statement_type(duckdb_prepared_statement);
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_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);
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);
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);

**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
• returns
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.

• returns
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.
DuckDB Documentation

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.

• returns
  The parameter type

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.

• returns
  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.
DuckDB Documentation

Syntax

duckdb_state duckdb_bind_int16(
    duckdb_prepared_statement prepared_statement,
    idx_t param_idx,
    int16_t val
);

duckdb_bind_int16  Binds an int16_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_int32  Binds an int32_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_int64  Binds an int64_t 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_hugeint  Binds a duckdb_hugeint 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_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.
DuckDB Documentation

Syntax

```c
duckdb_state duckdb_bind_uint32(
    duckdb_prepared_statement prepared_statement,
    idx_t param_idx,
    uint32_t val
);
```

**duckdb_bind_uint64**  Binds an uint64_t value to the prepared statement at the specified index.

Syntax

```c
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

```c
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

```c
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_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.
DuckDB Documentation

**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.
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.
- **returns**
DuckDBSuccess on success or DuckDBError on failure.

**duckdb_execute_prepared_streaming** 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.

**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.
- **returns**
DuckDBSuccess on success or DuckDBError on failure.
duckdb_execute_prepared_arrow  Executes the prepared statement with the given bound parameters, and returns an arrow query result.

**Syntax**

```c
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.

- **returns**
  DuckDBSuccess on success or DuckDBError on failure.

duckdb_arrow_scan  Scans the Arrow stream and creates a view with the given name.

**Syntax**

```c
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.

- **returns**
  DuckDBSuccess on success or DuckDBError on failure.
**duckdb_arrow_array_scan**  Scans the Arrow array and creates a view with the given name.

**Syntax**

```c
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.

**Returns**

DuckDBSuccess on success or DuckDBError on failure.

**C API - 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**

```c
duckdb_query(con, "CREATE TABLE people (id INTEGER, name VARCHAR)");

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**

```c
duckdb_state duckdb_appender_create(duckdb_connection connection, const char *schema, const char *table, duckdb_appender *out_appender);
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_begin_row(duckdb_appender appender);
duckdb_state duckdb_appender_end_row(duckdb_appender appender);
duckdb_state duckdb_appender_append_bool(duckdb_appender appender, bool value);
duckdb_state duckdb_appender_append_int8(duckdb_appender appender, int8_t value);
duckdb_state duckdb_appender_append_int16(duckdb_appender appender, int16_t value);
duckdb_state duckdb_appender_append_int32(duckdb_appender appender, int32_t value);
duckdb_state duckdb_appender_append_int64(duckdb_appender appender, int64_t value);
duckdb_state duckdb_appender_append_hugeint(duckdb_appender appender, duckdb_hugeint value);
duckdb_state duckdb_appender_append_uint8(duckdb_appender appender, uint8_t value);
duckdb_state duckdb_appender_append_uint16(duckdb_appender appender, uint16_t value);
duckdb_state duckdb_appender_append_uint32(duckdb_appender appender, uint32_t value);
duckdb_state duckdb_appender_append_uint64(duckdb_appender appender, uint64_t value);
duckdb_state duckdb_appender_append_uhugeint(duckdb_appender appender, duckdb_uhugeint value);
duckdb_state duckdb_appender_append_float(duckdb_appender appender, float value);
duckdb_state duckdb_appender_append_double(duckdb_appender appender, double value);
duckdb_state duckdb_appender_append_date(duckdb_appender appender, duckdb_date value);
duckdb_state duckdb_appender_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_data_chunk(duckdb_appender appender, duckdb_data_chunk chunk);

**duckdb_appender_create**  
Creates an appender object.

**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.

• returns
  DuckDBSuccess on success or DuckDBError on failure.

**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.
DuckDB Documentation

Syntax

const char *duckdb_appender_error(
    duckdb_appender appender
);

Parameters

• appender

The appender to get the error from.

• returns

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 and the data to be appended to the base table.

This should generally not be used unless you know what you are doing. Instead, call duckdb_appender_destroy when you are done with the appender.

Syntax

duckdb_state duckdb_appender_flush(
    duckdb_appender appender
);

Parameters

• appender

The appender to flush.

• returns

DuckDBSuccess on success or DuckDBError on failure.

duckdb_appender_close Close the appender, flushing all intermediate state in the appender to the table and closing it for further appends.

This is generally not necessary. Call duckdb_appender_destroy instead.

Syntax

duckdb_state duckdb_appender_close(
    duckdb_appender appender
);
**Parameters**

- **appender**
  The appender to flush and close.

  - **returns**
    DuckDBSuccess on success or DuckDBError on failure.

**duckdb_appender_destroy**  Close the appender and destroy it. Flushing all intermediate state in the appender to the table, and de-allocating all memory associated with the appender.

**Syntax**

```c
duckdb_state duckdb_appender_destroy(
    duckdb_appender *appender
);
```

**Parameters**

- **appender**
  The appender to flush, close and destroy.

  - **returns**
    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**

```c
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**

```c
duckdb_state duckdb_appender_end_row(
    duckdb_appender appender
);
```
**Parameters**

- appender

The appender.

- returns

DuckDBSuccess on success or DuckDBError on failure.

**duckdb_append_bool**  
Append a bool value to the appender.

**Syntax**

```c
duckdb_state duckdb_append_bool(
    duckdb_appender appender,
    bool value
);
```

**duckdb_append_int8**  
Append an int8_t value to the appender.

**Syntax**

```c
duckdb_state duckdb_append_int8(
    duckdb_appender appender,
    int8_t value
);
```

**duckdb_append_int16**  
Append an int16_t value to the appender.

**Syntax**

```c
duckdb_state duckdb_append_int16(
    duckdb_appender appender,
    int16_t value
);
```

**duckdb_append_int32**  
Append an int32_t value to the appender.

**Syntax**

```c
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**
```c
duckdb_state duckdb_append_uint64(
    duckdb_appender appender,
    uint64_t value
);
```

**duckdb_append_uhugeint**  Append a duckdb_uhugeint value to the appender.

**Syntax**
```c
duckdb_state duckdb_append_uhugeint(
    duckdb_appender appender,
    duckdb_uhugeint value
);
```

**duckdb_append_float**  Append a float value to the appender.

**Syntax**
```c
duckdb_state duckdb_append_float(
    duckdb_appender appender,
    float value
);
```

**duckdb_append_double**  Append a double value to the appender.

**Syntax**
```c
duckdb_state duckdb_append_double(
    duckdb_appender appender,
    double value
);
```

**duckdb_append_date**  Append a duckdb_date value to the appender.

**Syntax**
```c
duckdb_state duckdb_append_date(
    duckdb_appender appender,
    duckdb_date value
);
```
**duckdb_append_time**  Append a duckdb_time value to the appender.

**Syntax**

```c
duckdb_state duckdb_append_time(
    duckdb_appender appender,
    duckdb_time value
);
```

**duckdb_append_timestamp**  Append a duckdb_timestamp value to the appender.

**Syntax**

```c
duckdb_state duckdb_append_timestamp(
    duckdb_appender appender,
    duckdb_timestamp value
);
```

**duckdb_append_interval**  Append a duckdb_interval value to the appender.

**Syntax**

```c
duckdb_state duckdb_append_interval(
    duckdb_appender appender,
    duckdb_interval value
);
```

**duckdb_append_varchar**  Append a varchar value to the appender.

**Syntax**

```c
duckdb_state duckdb_append_varchar(
    duckdb_appender appender,
    const char *val
);
```

**duckdb_append_varchar_length**  Append a varchar value to the appender.

**Syntax**

```c
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**

```c
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**

```c
duckdb_state duckdb_append_null(
   duckdb_appender appender
);
```

**duckdb_append_data_chunk**  Appends a pre-filled data chunk to the specified appender.

The types of the data chunk must exactly match the types of the table, no casting is performed. If the types do not match or the appender is in an invalid state, DuckDBError is returned. If the append is successful, DuckDBSuccess is returned.

**Syntax**

```c
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.

- **returns**
  The return state.
C API - 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

```c
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_bound(duckdb_table_function table_function, duckdb_table_function_bound_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

```c
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

```c
```
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.

Syntax
duckdb_table_function duckdb_create_table_function()

Parameters
  • returns
The table function object.

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**

```c
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**

```c
void duckdb_table_function_add_parameter(
    duckdb_table_function table_function,
    duckdb_logical_type type
);
```

**Parameters**

- **table_function**
The table function
- **type**
The type of the parameter to add.

---

duckdb_table_function_add_named_parameter  Adds a named parameter to the table function.

**Syntax**

```c
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 name of the parameter

- type
  The type of the parameter to add.

*duckdb_table_function_set_extra_info*  Assigns extra information to the table function that can be fetched during binding, etc.

Syntax

```c
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 bind data (if any)

*duckdb_table_function_set_bind*  Sets the bind function of the table function

Syntax

```c
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 Documentation

duckdb_table_function_set_init  Sets the init function of the table function

Syntax

```c
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 Documentation

duckdb_table_function_set_local_init  Sets the thread-local init function of the table function

Syntax

```c
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**

```c
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**

```c
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.
**duckdb_register_table_function**

```c
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

- **returns**
  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`.

```c
void *duckdb_bind_get_extra_info(
    duckdb_bind_info info
);
```

**Parameters**

- **info**
  The info object

- **returns**
  The extra info

**duckdb_bind_add_result_column**

Adds a result column to the output of the table function.

```c
void duckdb_bind_add_result_column(
    duckdb_bind_info info,
    const char *name,
    duckdb_logical_type type
);
```
**Parameters**

- **info**
  The info object

- **name**
  The name of the column

- **type**
  The logical type of the column

---

**duckdb_bind_get_parameter_count** Retrieves the number of regular (non-named) parameters to the function.

**Syntax**

```c
idx_t duckdb_bind_get_parameter_count(
    duckdb_bind_info info
);
```

**Parameters**

- **info**
  The info object

- **returns**
  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**

```c
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
  - returns
  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
  - returns
  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. 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 info object

- `extra_data`

  The bind data object.

- `destroy`

  The callback that will be called to destroy the bind data (if any)

---

**duckdb_bind_set_cardinality**  
Sets the cardinality estimate for the table function, used for optimization.

**Syntax**

```c
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.

**Syntax**

```c
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**

```c
void *duckdb_init_get_extra_info(
    duckdb_init_info info
);
```

**Parameters**

- **info**
  The info object
- **returns**
  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**

```c
void *duckdb_init_get_bind_data(
    duckdb_init_info info
);
```

**Parameters**

- **info**
  The info object
- **returns**
  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**

```c
void duckdb_init_set_init_data(
    duckdb_init_info info,
    void *init_data,
    duckdb_delete_callback_t destroy
);
```

**Parameters**

- **info**
  The info object

- **extra_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**

```c
idx_t duckdb_init_get_column_count(
    duckdb_init_info info
);
```

**Parameters**

- **info**
  The info object

- **returns**
  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**

```c
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)

- **returns**
  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**

```c
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**

```c
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

```c
void *duckdb_function_get_extra_info(
    duckdb_function_info info
);
```

Parameters

• info

The info object

• returns

The extra info

`duckdb_function_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

```c
void *duckdb_function_get_bind_data(
    duckdb_function_info info
);
```

Parameters

• info

The info object

• returns

The bind data object
**duckdb_function_get_init_data**  Gets the init data set by duckdb_init_set_init_data during the init.

**Syntax**

```c
void *duckdb_function_get_init_data(
    duckdb_function_info info
);
```

**Parameters**

- `info`  
  The info object

- `returns`  
  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**

```c
void *duckdb_function_get_local_init_data(
    duckdb_function_info info
);
```

**Parameters**

- `info`  
  The info object

- `returns`  
  The init data object

**duckdb_function_set_error**  Report that an error has occurred while executing the function.

**Syntax**

```c
void duckdb_function_set_error(
    duckdb_function_info info,  
    const char *error  
);
```
Parameters

• info
The info object

• error
The error message

C API - 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

```c
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);
```

```c
duckdb_add_replacement_scan    Add a replacement scan definition to the specified database
```

Syntax

```c
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 to use. 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**

```c
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**

```c
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**

```c
void duckdb_replacement_scan_set_error(
    duckdb_replacement_scan_info info,
    const char *error
);
```

**Parameters**

- **info**
  The info object
- **error**
  The error message

### C API - Complete API

**API Reference**

**Open/Connect**

```c
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_query_progress_type duckdb_query_progress);duckdb_connection *connection);
const char *duckdb_library_version();
```

**Configuration**

```c
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_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);

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);
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);
tint16_t duckdb_value_int16(duckdb_result *result, idx_t col, idx_t row);
tint32_t duckdb_value_int32(duckdb_result *result, idx_t col, idx_t row);
tint64_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_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);
DuckDB Documentation

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);

Date/Time/Timestamp Helpers

duckdb_date_struct duckdb_from_date(duckdb_date date);
duckdb_date duckdb_to_date(duckdb_date_struct date);
duckdb_time_struct duckdb_from_time(duckdb_time time);
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);

Hugeint Helpers

double duckdb_hugeint_to_double(duckdb_hugeint val);
duckdb_hugeint duckdb_double_to_hugeint(double val);
duckdb_decimal duckdb_double_to_decimal(double val, uint8_t width, uint8_t scale);

Unsigned Hugeint Helpers

double duckdb_uhugeint_to_double(duckdb_uhugeint val);
duckdb_uhugeint duckdb_double_to_uhugeint(double val);

Decimal Helpers

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_state duckdb_clear_bindings(duckdb_prepared_statement prepared_statement);
duckdb_statement_type duckdb_prepared_statement_type(duckdb_prepared_statement);
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_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);
DuckDB Documentation

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);
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);

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_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_int64(int64_t val);
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);
char *duckdb_get_varchar(duckdb_value value);
int64_t duckdb_get_int64(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);
duckdb_logical_type duckdb_create_list_type(duckdb_logical_type type);
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_create_list_type_child_type(duckdb_logical_type type);
duckdb_logical_type duckdb_create_map_type_key_type(duckdb_logical_type type);
duckdb_logical_type duckdb_create_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_create_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);

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_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_vectorEnsure_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);
duckdb_state duckdb_list_vector_get_size(duckdb_vector vector);
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);

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);

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 table_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);

Appender
DuckDB Documentation

```c
duckdb_state duckdb_appender_create(duckdb_connection connection, const char *schema, const char *table, duckdb_appender *out_appender);
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_begin_row(duckdb_appender appender);
duckdb_state duckdb_appender_end_row(duckdb_appender appender);
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_data_chunk(duckdb_appender appender, duckdb_data_chunk chunk);
```

**Arrow Interface**

```c
duckdb_state duckdb_query_arrow(duckdb_connection connection, const char *query, duckdb_arrow *out_result);
duckdb_state duckdb_query_arrow_schema(duckdb_result result, duckdb_arrow_schema *out_schema);
```

Arrow Interface
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);

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_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 instantiated database should 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, or nullptr or :memory: to open an in-memory database.
  • out_database
The result database object.
  • returns
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.

**Syntax**

```c
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, or nullptr or :memory: to open an in-memory database.

- **out_database**
  
  The result database object.

- **config**
  
  (Optional) configuration used to start up the database system.

- **out_error**
  
  If set and the function returns DuckDBError, this will contain the reason why the start-up failed. Note that the error must be freed using duckdb_free.

**returns**

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. 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**

```c
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**

```c
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.

**returns**

DuckDBSuccess on success or DuckDBError on failure.

**duckdb_interrupt**  
Interrupt running query

**Syntax**

```c
void duckdb_interrupt(
    duckdb_connection connection
);
```

**Parameters**

- `connection`

The connection to interrupt.

**duckdb_query_progress**  
Get progress of the running query
DuckDB Documentation

**Syntax**

```c
duckdb_query_progress_type duckdb_query_progress(
    duckdb_connection connection
);
```

**Parameters**

• `connection`

The working connection

• `returns`

-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**

```c
void duckdb_disconnect(
    duckdb_connection *connection
);
```

**Parameters**

• `connection`

The connection to close.

**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**

```c
const char *duckdb_library_version(
);
```

**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`.

This will always succeed unless there is a malloc failure.
**Syntax**

```c
duckdb_state duckdb_create_config(
    duckdb_config *out_config
);
```

**Parameters**

- `out_config`
The result configuration object.

- `returns`
  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.

**Syntax**

```c
size_t duckdb_config_count(
);
```

**Parameters**

- `returns`
The amount of config options available.

**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**

```c
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.

- **returns**
  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**

```c
duckdb_state duckdb_set_config(
    duckdb_config config,
    const char *name,
    const char *option
);
```

**Parameters**

- **duckdb_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.

- **returns**
  DuckDBSuccess on success or DuckDBError on failure.
**duckdb_destroy_config**  Destroys the specified configuration option and de-allocates all memory allocated for the object.

**Syntax**

```c
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**

```c
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.

- **returns**
  `DuckDBSuccess` on success or `DuckDBError` on failure.

**duckdb_destroy_result**  Closes the result and de-allocates all memory allocated for that connection.
DuckDB Documentation

Syntax

```c
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 be
freed; the column names will automatically be destroyed when the result is destroyed.

Returns NULL if the column is out of range.

Syntax

```c
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.

- **returns**
  
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

```c
duckdb_type duckdb_column_type(
    duckdb_result *result, 
    idx_t col
);
```
**duckdb_result_statement_type**  Returns the statement type of the statement that was executed

**Syntax**

```c
duckdb_statement_type duckdb_result_statement_type(
    duckdb_result result
);
```

**Parameters**

- `result`  The result object to fetch the statement type from.

- `returns`  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**

```c
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.

- **returns**
  The logical column type of the specified column.

**duckdb_column_count**  Returns the number of columns present in a the result object.

**Syntax**

```c
idx_t duckdb_column_count(
    duckdb_result *result
);
```

**Parameters**

- **result**
  The result object.

- **returns**
  The number of columns present in the result object.

**duckdb_row_count**  Returns the number of rows present in a the result object.

**Syntax**

```c
idx_t duckdb_row_count(
    duckdb_result *result
);
```

**Parameters**

- **result**
  The result object.

- **returns**
  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**

```c
idx_t duckdb_rows_changed(
    duckdb_result *result
);
```

**Parameters**

- **result**
The result object.
- **returns**
The number of rows changed.

**duckdb_column_data**  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:

```c
int32_t *data = (int32_t *) duckdb_column_data(&result, 0);
printf("Data for row %d: %d\n", row, data[row]);
```

**Syntax**

```c
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.
- **returns**
The column data of the specified column.
**duckdb_nullmask_data**  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.

```c
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**

```c
bool *duckdb_nullmask_data(
    duckdb_result *result,
    idx_t col
);
```

**Parameters**

- **result**
  The result object to fetch the nullmask from.

- **col**
  The column index.

- **returns**
  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**

```c
const char *duckdb_result_error(
    duckdb_result *result
);
```
**Parameters**

- **result**
  The result object to fetch the error from.

- **returns**
  The error of the result.

**duckdb_result_get_chunk**  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**

```c
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.

- **returns**
  The resulting data chunk. Returns NULL if the chunk index is out of bounds.

**duckdb_result_is_streaming**  Checks if the type of the internal result is StreamQueryResult.

**Syntax**

```c
bool duckdb_result_is_streaming(
    duckdb_result result
);
```
**Parameters**

- **result**
  The result object to check.
  - **returns**
  Whether or not the result object is of the type StreamQueryResult

**duckdb_result_chunk_count**  Returns the number of data chunks present in the result.

**Syntax**

```c
idx_t duckdb_result_chunk_count(
  duckdb_result result
);
```

**Parameters**

- **result**
  The result object
  - **returns**
  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**

```c
duckdb_result_type duckdb_result_return_type(
  duckdb_result result
);
```

**Parameters**

- **result**
  The result object
  - **returns**
  The return_type

**duckdb_value_boolean**
**Syntax**

`bool duckdb_value_boolean(
    duckdb_result *result,
    idx_t col,
    idx_t row
);`

**Parameters**

- **returns**
The boolean value at the specified location, or false if the value cannot be converted.

**duckdb_value_int8**

**Syntax**

`int8_t duckdb_value_int8(
    duckdb_result *result,
    idx_t col,
    idx_t row
);`

**Parameters**

- **returns**
The int8_t value at the specified location, or 0 if the value cannot be converted.

**duckdb_value_int16**

**Syntax**

`int16_t duckdb_value_int16(
    duckdb_result *result,
    idx_t col,
    idx_t row
);`

**Parameters**

- **returns**
The int16_t value at the specified location, or 0 if the value cannot be converted.


duckdb_value_int32

Syntax

```c
int32_t duckdb_value_int32(*result, idx_t col, idx_t row);
```

Parameters

- returns
The int32_t value at the specified location, or 0 if the value cannot be converted.

duckdb_value_int64

Syntax

```c
int64_t duckdb_value_int64(*result, idx_t col, idx_t row);
```

Parameters

- returns
The int64_t value at the specified location, or 0 if the value cannot be converted.

duckdb_value_hugeint

Syntax

```c
duckdb_hugeint duckdb_value_hugeint(*result, idx_t col, idx_t row);
```

Parameters

- returns
The duckdb_hugeint value at the specified location, or 0 if the value cannot be converted.
duckdb_value_uhugeint

Syntax

```c
duckdb_uhugeint duckdb_value_uhugeint(
    duckdb_result *result,
    idx_t col,
    idx_t row
);
```

Parameters

- returns

The duckdb_uhugeint value at the specified location, or 0 if the value cannot be converted.

duckdb_value_decimal

Syntax

```c
duckdb_decimal duckdb_value_decimal(
    duckdb_result *result,
    idx_t col,
    idx_t row
);
```

Parameters

- returns

The duckdb_decimal value at the specified location, or 0 if the value cannot be converted.

duckdb_value_uint8

Syntax

```c
uint8_t duckdb_value_uint8(
    duckdb_result *result,
    idx_t col,
    idx_t row
);
```

Parameters

- returns

The uint8_t value at the specified location, or 0 if the value cannot be converted.
**duckdb_value_uint16**

**Syntax**

```c
uint16_t duckdb_value_uint16(
    duckdb_result *result,
    idx_t col,
    idx_t row
);
```

**Parameters**

- returns

The `uint16_t` value at the specified location, or 0 if the value cannot be converted.

**duckdb_value_uint32**

**Syntax**

```c
uint32_t duckdb_value_uint32(
    duckdb_result *result,
    idx_t col,
    idx_t row
);
```

**Parameters**

- returns

The `uint32_t` value at the specified location, or 0 if the value cannot be converted.

**duckdb_value_uint64**

**Syntax**

```c
uint64_t duckdb_value_uint64(
    duckdb_result *result,
    idx_t col,
    idx_t row
);
```

**Parameters**

- returns

The `uint64_t` value at the specified location, or 0 if the value cannot be converted.
duckdb_value_float

Syntax

```c
float duckdb_value_float(
    duckdb_result *result,
    idx_t col,
    idx_t row
);
```

Parameters

• returns

The float value at the specified location, or 0 if the value cannot be converted.

duckdb_value_double

Syntax

```c
double duckdb_value_double(
    duckdb_result *result,
    idx_t col,
    idx_t row
);
```

Parameters

• returns

The double value at the specified location, or 0 if the value cannot be converted.

duckdb_value_date

Syntax

```c
duckdb_date duckdb_value_date(
    duckdb_result *result,
    idx_t col,
    idx_t row
);
```

Parameters

• returns

The duckdb_date value at the specified location, or 0 if the value cannot be converted.
duckdb_value_time

Syntax

duckdb_value_time(duckdb_result *result, idx_t col, idx_t row);

Parameters

• returns

The duckdb_time value at the specified location, or 0 if the value cannot be converted.

duckdb_value_timestamp

Syntax

duckdb_value_timestamp(duckdb_result *result, idx_t col, idx_t row);

Parameters

• returns

The duckdb_timestamp value at the specified location, or 0 if the value cannot be converted.

duckdb_value_interval

Syntax

duckdb_value_interval(duckdb_result *result, idx_t col, idx_t row);

Parameters

• returns

The duckdb_interval value at the specified location, or 0 if the value cannot be converted.
duckdb_value_varchar

Syntax

char *duckdb_value_varchar(
    duckdb_result *result,
    idx_t col,
    idx_t row
);

Parameters

• DEPRECATED
use duckdb_value_string instead. This function does not work correctly if the string contains null bytes.

• returns
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.

duckdb_value_varchar_internal

Syntax

char *duckdb_value_varchar_internal(
    duckdb_result *result,
    idx_t col,
    idx_t row
);

Parameters

• DEPRECATED
use duckdb_value_string_internal instead. This function does not work correctly if the string contains null bytes.

• returns
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.

duckdb_value_string_internal
DuckDB Documentation

**Syntax**

`duckdb_string duckdb_value_string_internal(`
`    duckdb_result *result,`
`    idx_t col,`
`    idx_t row`
`);`

**Parameters**

- **DEPRECATED**
  
  use `duckdb_value_string_internal` instead. This function does not work correctly if the string contains null bytes.
  
  - **returns**
  
  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.

**duckdb_value_blob**

**Syntax**

`duckdb_blob duckdb_value_blob(`
`    duckdb_result *result,`
`    idx_t col,`
`    idx_t row`
`);`

**Parameters**

- **returns**

  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 "blob.data" must be freed with `duckdb_free`.

**duckdb_value_is_null**

**Syntax**

`bool duckdb_value_is_null(`
`    duckdb_result *result,`
`    idx_t col,`
`    idx_t row`
`);`
Parameters

- returns

Returns true if the value at the specified index is NULL, and false otherwise.

**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**

```c
void *duckdb_malloc(
    size_t size
);
```

**Parameters**

- size

  The number of bytes to allocate.

- returns

  A pointer to the allocated memory region.

**duckdb_free** Free a value returned from duckdb_malloc, duckdb_value_varchar or duckdb_value_blob.

**Syntax**

```c
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.

**Syntax**

```c
idx_t duckdb_vector_size(
);
```
Parameters

- returns

The 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**

```c
bool duckdb_string_is_inlined(
    duckdb_string_t string
);
```

**duckdb_from_date** Decompose a *duckdb_date* object into year, month and date (stored as *duckdb_date_struct*).

**Syntax**

```c
duckdb_date_struct duckdb_from_date(
    duckdb_date date
);
```

**Parameters**

- date

The date object, as obtained from a DUCKDB_TYPE_DATE column.

- returns

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**

```c
duckdb_date duckdb_to_date(
    duckdb_date_struct date
);
```
Parameters

- **date**
  The year, month and date stored in a duckdb_date_struct.
  - **returns**
    The duckdb_date element.

**duckdb_from_time**  Decompose a duckdb_time object into hour, minute, second and microsecond (stored as duckdb_time_struct).

Syntax

```c
duckdb_time_struct duckdb_from_time(
    duckdb_time time
);
```

Parameters

- **time**
  The time object, as obtained from a DUCKDB_TYPE_TIME column.
  - **returns**
    The duckdb_time_struct with the decomposed elements.

**duckdb_to_time**  Re-compose a duckdb_time from hour, minute, second and microsecond (duckdb_time_struct).

Syntax

```c
duckdb_time duckdb_to_time(
    duckdb_time_struct time
);
```

Parameters

- **time**
  The hour, minute, second and microsecond in a duckdb_time_struct.
  - **returns**
    The duckdb_time element.
**duckdb_from_timestamp**  Decompose a `duckdb_timestamp` object into a `duckdb_timestamp_struct`.

**Syntax**

```c
duckdb_timestamp_struct duckdb_from_timestamp(
    duckdb_timestamp ts
);
```

**Parameters**

- `ts`
  The `ts` object, as obtained from a `DUCKDB_TYPE_TIMESTAMP` column.

- `returns`
  The `duckdb_timestamp_struct` with the decomposed elements.

**duckdb_to_timestamp**  Re-compose a `duckdb_timestamp` from a `duckdb_timestamp_struct`.

**Syntax**

```c
duckdb_timestamp duckdb_to_timestamp(
    duckdb_timestamp_struct ts
);
```

**Parameters**

- `ts`
  The decomposed elements in a `duckdb_timestamp_struct`.

- `returns`
  The `duckdb_timestamp` element.

**duckdb_hugeint_to_double**  Converts a `duckdb_hugeint` object (as obtained from a `DUCKDB_TYPE_HUGEINT` column) into a double.

**Syntax**

```c
double duckdb_hugeint_to_double(
    duckdb_hugeint val
);
```
**Parameters**

- **val**  
The hugeint value.

- **returns**  
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**

```c
duckdb_hugeint duckdb_double_to_hugeint(
    double val
);
```

**Parameters**

- **val**  
The double value.

- **returns**  
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**

```c
duckdb_decimal duckdb_double_to_decimal(
    double val,
    uint8_t width,
    uint8_t scale
);
```

**Parameters**

- **val**  
The double value.

- **returns**  
The converted duckdb_decimal element.
**duckdb_uhugeint_to_double**  Converts a duckdb_uhugeint object (as obtained from a DUCKDB_TYPE_UHUGEINT column) into a double.

**Syntax**

```c
double duckdb_uhugeint_to_double(
    duckdb_uhugeint val
);
```

**Parameters**

- **val**
  The uhugeint value.

- **returns**
  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**

```c
duckdb_uhugeint duckdb_double_to_uhugeint(
    double val
);
```

**Parameters**

- **val**
  The double value.

- **returns**
  The converted duckdb_uhugeint element.

**duckdb_decimal_to_double**  Converts a duckdb_decimal object (as obtained from a DUCKDB_TYPE_DECIMAL column) into a double.

**Syntax**

```c
double duckdb_decimal_to_double(
    duckdb_decimal val
);
```
Parameters

• val
The decimal value.

• returns
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

• returns
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

```c
const char *duckdb_prepare_error(
    duckdb_prepared_statement prepared_statement
);
```

Parameters

- prepared_statement
  The prepared statement to obtain the error from.
- returns
  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

```c
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**

```c
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**

```c
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.
- `returns`
  The parameter type

**duckdb_clear_bindings** Clear the params bind to the prepared statement.

**Syntax**

```c
duckdb_state duckdb_clear_bindings(
    duckdb_prepared_statement prepared_statement
);
```

**duckdb_prepared_statement_type** Returns the statement type of the statement to be executed
DuckDB Documentation

Syntax

duckdb_statement_type duckdb_prepared_statement_type(
    duckdb_prepared_statement statement
);

Parameters

• statement

The prepared statement.

• returns

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**

```c
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**

```c
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**

```c
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**

```c
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**

```c
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**

```c
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**

```c
duckdb_state duckdb_bind_decimal(
    duckdb_prepared_statement prepared_statement,
    idx_t param_idx,
    duckdb_decimal val
);
```

**duckdb_bind_uint8** Binds an `uint8_t` value to the prepared statement at the specified index.

**Syntax**

```c
duckdb_state duckdb_bind_uint8(
    duckdb_prepared_statement prepared_statement,
    idx_t param_idx,
    uint8_t val
);
```

**duckdb_bind_uint16** Binds an `uint16_t` value to the prepared statement at the specified index.
**Syntax**

```c
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**

```c
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**

```c
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**

```c
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.
DuckDB Documentation

**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_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.
DuckDB Documentation

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.

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.

- **returns**
  DuckDBSuccess on success or DuckDBError on failure.

**duckdb_execute_prepared_streaming** 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.

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.
- *returns*
DuckDBSuccess on success or DuckDBError on failure.

**duckdb_execute_prepared_arrow**  Executes the prepared statement with the given bound parameters, and returns an arrow query result.

**Syntax**

```c
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.
- *returns*
DuckDBSuccess on success or DuckDBError on failure.

**duckdb_arrow_scan**  Scans the Arrow stream and creates a view with the given name.

**Syntax**

```c
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.
- returns
  DuckDBSuccess on success or DuckDBError on failure.

**duckdb-arrow-array-scan**  Scans the Arrow array and creates a view with the given name.

Syntax

```c
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.
- returns
  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**

```c
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

**Returns**

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**

```c
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
        - returns
          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

```c
const char *duckdb_extract_statements_error(
    duckdb_extracted_statements extracted_statements
);
```

Parameters

- result
  The extracted statements to fetch the error from.
  - returns
    The error of the extracted statements.

duckdb_destroy_extracted  De-allocates all memory allocated for the extracted statements.

Syntax

```c
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.

**returns**

`DuckDBSuccess` on success or `DuckDBError` on failure.

**duckdb_pending_prepared_streaming**  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.

- `returns`
  
  DuckDBSuccess on success or DuckDBError on failure.

**duckdb_destroy_pending**  
Closes the pending result and de-allocates all memory allocated for the result.

**Syntax**

```c
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**

```c
const char *duckdb_pending_error(
    duckdb_pending_result pending_result
);
```

**Parameters**

- `result`
  
  The pending result to fetch the error from.

- `returns`
  
  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**

```c
duckdb_pending_state duckdb_pending_execute_task(
    duckdb_pending_result pending_result
);
```

**Parameters**

- **pending_result**
  The pending result to execute a task within.

- **returns**
  The state of the pending result after the execution.

**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.

**Syntax**

```c
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.

- **returns**
  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**

```c
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.

- *returns*
  
  Boolean indicating pending execution should be considered finished.

**duckdb_destroy_value**  Destroys the value and de-allocates all memory allocated for that type.

**Syntax**

```c
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**

```c
duckdb_value duckdb_create_varchar(
    const char *text
);
```
**Parameters**

- **value**
  The null-terminated string

- **returns**
  The value. This must be destroyed with duckdb_destroy_value.

### `duckdb_create_varchar_length`

**Creates a value from a string**

**Syntax**

```c
duckdb_value duckdb_create_varchar_length(
    const char *text,
    idx_t length
);
```

**Parameters**

- **value**
  The text

- **length**
  The length of the text

- **returns**
  The value. This must be destroyed with duckdb_destroy_value.

### `duckdb_create_int64`

**Creates a value from an int64**

**Syntax**

```c
duckdb_value duckdb_create_int64(
    int64_t val
);
```

**Parameters**

- **value**
  The bigint value

- **returns**
  The value. This must be destroyed with duckdb_destroy_value.
duckdb_create_struct_value  Creates a struct value from a type and an array of values

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

• returns
The value. This must be destroyed with duckdb_destroy_value.

duckdb_create_list_value  Creates a list value from a type and an array of values of length value_count

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

• returns
The value. This must be destroyed with duckdb_destroy_value.
**duckdb_get_vchar**  Obtains a string representation of the given value. The result must be destroyed with **duckdb_free**.

**Syntax**

```c
char *duckdb_get_vchar(
    duckdb_value value
);
```

**Parameters**

• **value**
  The value
  • **returns**
  The string value. This must be destroyed with **duckdb_free**.

**duckdb_get_int64**  Obtains an int64 of the given value.

**Syntax**

```c
int64_t duckdb_get_int64(
    duckdb_value value
);
```

**Parameters**

• **value**
  The value
  • **returns**
  The int64 value, or 0 if no conversion is possible

**duckdb_create_logical_type**  Creates a duckdb_logical_type from a standard primitive type. The resulting type should be destroyed with **duckdb_destroy_logical_type**.

This should not be used with **DUCKDB_TYPE_DECIMAL**.

**Syntax**

```c
duckdb_logical_type duckdb_create_logical_type(
    duckdb_type type
);```
**Parameters**

- **type**

  The primitive type to create.

- **returns**

  The logical type.

**duckdb_logical_type_get_alias**  Returns the alias of a duckdb_logical_type, if one is set, else NULL

You must free the result.

**Syntax**

```c
char *duckdb_logical_type_get_alias(
    duckdb_logical_type type
);
```

**Parameters**

- **type**

  The logical type to return the alias of

- **returns**

  The alias or NULL

**duckdb_create_list_type**  Creates a list type from its child type. The resulting type should be destroyed with duckdb_destroy_logical_type.

**Syntax**

```c
duckdb_logical_type duckdb_create_list_type(
    duckdb_logical_type type
);
```

**Parameters**

- **type**

  The child type of list type to create.

- **returns**

  The logical type.
duckdb_create_map_type  Creates a map type from its key type and value type. The resulting type should 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

• type
  The key type and value type of map type to create.

• returns
  The logical type.

duckdb_create_union_type  Creates a UNION type from the passed types array. The resulting type should 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

• types
  The array of types that the union should consist of.

• type_amount
  The size of the types array.

• returns
  The logical type.

duckdb_create_struct_type  Creates a STRUCT type from the passed member name and type arrays. The resulting type should be destroyed with duckdb_destroy_logical_type.
DuckDB Documentation

**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 that the struct should consist of.

- **member_names**
  The array of names that the struct should consist of.

- **member_count**
  The number of members that were specified for both arrays.

- **returns**
  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**

- **enum_name**
  The name of the enum.

- **member_names**
  The array of names that the enum should consist of.

- **member_count**
  The number of elements that were specified in the array.

- **returns**
  The logical type.
**duckdb_create_decimal_type**  Creates a `duckdb_logical_type` of type decimal with the specified width and scale. The resulting type should be destroyed with `duckdb_destroy_logical_type`.

**Syntax**

```c
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
- **returns**  The logical type.

**duckdb_get_type_id**  Retrieves the type class of a `duckdb_logical_type`.

**Syntax**

```c
duckdb_type duckdb_get_type_id(
    duckdb_logical_type type
);
```

**Parameters**

- **type**  The logical type object
- **returns**  The type id

**duckdb_decimal_width**  Retrieves the width of a decimal type.

**Syntax**

```c
uint8_t duckdb_decimal_width(
    duckdb_logical_type 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
- `returns`  The width 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
- `returns`  The scale 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
- `returns`  The internal type of the decimal type
**Syntax**

duckdb_type duckdb_enum_internal_type(
    duckdb_logical_type type
);

**Parameters**

- **type**
  The logical type object

- **returns**
  The internal type of the enum type

**duckdb_enum_dictionary_size**  Retrieves the dictionary size of the enum type

**Syntax**

```c
uint32_t duckdb_enum_dictionary_size(
    duckdb_logical_type type
);
```

**Parameters**

- **type**
  The logical type object

- **returns**
  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**

```c
char *duckdb_enum_dictionary_value(
    duckdb_logical_type type,
    idx_t index
);
```
Parameters

- type
  The logical type object
  - index
    The index in the dictionary
    - returns
      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.
The result must be freed with duckdb_destroy_logical_type

Syntax

```c
duckdb_logical_type duckdb_list_type_child_type(
    duckdb_logical_type type
);
```

Parameters

- type
  The logical type object
  - returns
    The child type of the list type. Must be destroyed with duckdb_destroy_logical_type.

`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

```c
duckdb_logical_type duckdb_map_type_key_type(
    duckdbLogicalType type
);
```

Parameters

- type
  The logical type object
  - returns
    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**

```c
duckdb_logical_type duckdb_map_type_value_type(
    duckdb_logical_type type
);
```

**Parameters**

- **type**
  The logical type object

- **returns**
  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**

```c
idx_t duckdb_struct_type_child_count(
    duckdb_logical_type type
);
```

**Parameters**

- **type**
  The logical type object

- **returns**
  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**

```c
char *duckdb_struct_type_child_name(
    duckdb_logical_type type,
    idx_t index
);
```
Parameters

- **type**
  The logical type object

- **index**
  The child index

- **returns**
  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

```c
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

- **returns**
  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

```c
idx_t duckdb_union_type_member_count(
    duckdb_logical_type type
);
```
Parameters

- type
  The logical type (union) object

- returns
  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

```c
char *duckdb_union_type_member_name(
    duckdb_logical_type type,
    idx_t index
);
```

Parameters

- type
  The logical type object

- index
  The child index

- returns
  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

```c
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
  - **returns**
  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**

```c
void duckdb_destroy_logical_type(
    duckdb_logical_type *type
);
```

**Parameters**

- **type**
  The logical type to destroy.

**duckdb_create_data_chunk**  Creates an empty DataChunk with the specified set of types.

**Syntax**

```c
duckdb_data_chunk duckdb_create_data_chunk(
    duckdb_logical_type *types,
    idx_t column_count
);
```

**Parameters**

- **types**
  An array of types of the data chunk.
  - **column_count**
  The number of columns.
  - **returns**
  The data chunk.
**duckdb_destroy_data_chunk**  Destroys the data chunk and de-allocates all memory allocated for that chunk.

**Syntax**

```c
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.

**Syntax**

```c
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**

```c
idx_t duckdb_data_chunk_get_column_count(
    duckdb_data_chunk chunk
);
```

**Parameters**

- `chunk`  
  The data chunk to get the data from

- `returns`  
  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**

```c
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
- **returns**
  The vector

**duckdb_data_chunk_get_size**  Retrieves the current number of tuples in a data chunk.

**Syntax**

```c
idx_t duckdb_data_chunk_get_size(  
    duckdb_data_chunk chunk  
);
```

**Parameters**

- **chunk**
  The data chunk to get the data from
- **returns**
  The number of tuples in the data chunk

**duckdb_data_chunk_set_size**  Sets the current number of tuples in a data chunk.

**Syntax**

```c
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_vector_get_column_type**  
Retrieves the column type of the specified vector.  
The result must be destroyed with `duckdb_destroy_logical_type`.

**Syntax**

```c
duckdb_logical_type duckdb_vector_get_column_type(
    duckdb_vector vector
);
```

**Parameters**

- `vector`

The vector get the data from

- `returns`

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**

```c
void *duckdb_vector_get_data(
    duckdb_vector vector
);
```

**Parameters**

- `vector`

The vector to get the data from

- `returns`

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:

```c
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**

```c
uint64_t *duckdb_vector_get_validity(
    duckdb_vector vector
);
```

**Parameters**

- **vector**
  The vector to get the data from

- **returns**
  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**

```c
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**

```c
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.

**Syntax**

```c
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 null-terminated string
The string

- \textit{str\_len}

The length of the string (in bytes)

\textbf{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.

\textbf{Syntax}

\begin{verbatim}
duckdb\_vector duckdb\_list\_vector\_get\_child(
    duckdb\_vector vector
);
\end{verbatim}

\textbf{Parameters}

- \textit{vector}

The vector

- \textit{returns}

The child vector

\textbf{duckdb\_list\_vector\_get\_size}  Returns the size of the child vector of the list

\textbf{Syntax}

\begin{verbatim}
duckdb\_vector duckdb\_list\_vector\_get\_size(
    duckdb\_vector vector
);
\end{verbatim}

\textbf{Parameters}

- \textit{vector}

The vector

- \textit{returns}

The size of the child list

\textbf{duckdb\_list\_vector\_set\_size}  Sets the total size of the underlying child-vector of a list vector.
**duckdb_list_vector_set_size**

Sets the size of the child list.

**Syntax**

```c
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.
- **returns**
  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.

**Syntax**

```c
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**
  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.
DuckDB Documentation

**Syntax**

duckdb_vector duckdb_struct_vector_get_child(
    duckdb_vector vector,
    idx_t index
);

**Parameters**

- **vector**
The vector
- **index**
The child index
- **returns**
The child 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
- **returns**
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

```c
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_validity`  In a validity mask, sets a specific row to valid.
Equivalent to `duckdb_validity_set_row_validity` with valid set to true.

Syntax

```c
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_invalid`  In a validity mask, sets a specific row to invalid.
Equivalent to `duckdb_validity_set_row_validity` with valid set to false.
DuckDB Documentation

Syntax

```c
void duckdb_validity_set_row_valid(
    uint64_t *validity,
    idx_t row
);
```

Parameters

- **validity**
The validity mask
- **row**
The row index

**duckdb_create_table_function**  Creates a new empty table function.
The return value should be destroyed with duckdb_destroy_table_function.

Syntax

```c
duckdb_table_function duckdb_create_table_function(
);
```

Parameters

- **returns**
The table function object.

**duckdb_destroy_table_function**  Destroys the given table function object.

Syntax

```c
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.
DuckDB Documentation

Syntax

```c
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

```c
void duckdb_table_function_add_parameter(
    duckdb_table_function table_function,
    duckdb_logical_type type
);
```

Parameters

- `table_function`
  The table function

- `type`
  The type of the parameter to add.

`duckdb_table_function_add_named_parameter`  Adds a named parameter to the table function.

Syntax

```c
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 name of the parameter

- **type**
  The type of the parameter to add.

**duckdb_table_function_set_extra_info**  Assigns extra information to the table function that can be fetched during binding, etc.

Syntax

```c
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 bind data (if any)

**duckdb_table_function_set_bind**  Sets the bind function of the table function

Syntax

```c
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**

```c
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**

```c
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**

```c
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**

```c
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.
**DuckDB Documentation**

**Syntax**

```c
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
- **returns**
  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**

```c
void *duckdb_bind_get_extra_info(
    duckdb_bind_info info
);
```

**Parameters**

- **info**
  The info object
- **returns**
  The extra info

**duckdb_bind_add_result_column**  Adds a result column to the output of the table function.

**Syntax**

```c
void duckdb_bind_add_result_column(
    duckdb_bind_info info,
    const char *name,
    duckdb_logical_type type
);
```
Parameters

• info

The info object

• name

The name of the column

• type

The logical type of the column

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

• returns

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
- returns
  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

```c
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
- returns
  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. This object can be retrieved again during execution.

Syntax

```c
void duckdb_bind_set_bind_data(
    duckdb_bind_info info,
    void *bind_data,
    duckdb_delete_callback_t destroy
);
```
DuckDB Documentation

Parameters

- **info**
The info object

- **extra_data**
The bind data object.

- **destroy**
The callback that will be called to destroy the bind data (if any)

**duckdb_bind_set_cardinality**  Sets the cardinality estimate for the table function, used for optimization.

**Syntax**

```c
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.

**Syntax**

```c
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

• returns
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

• returns
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**

```c
void duckdb_init_set_init_data(
    duckdb_init_info info,
    void *init_data,
    duckdb_delete_callback_t destroy
);
```

**Parameters**

- **info**
The info object
- **extra_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**

```c
idx_t duckdb_init_get_column_count(
    duckdb_init_info info
);
```

**Parameters**

- **info**
The info object
- **returns**
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**

```c
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)

- **returns**
  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**

```c
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**

```c
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**

```c
void *duckdb_function_get_extra_info(  
duckdb_function_info info  
);
```

**Parameters**
- **info**
The info object
- **returns**
The extra info

**duckdb_function_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**

```c
void *duckdb_function_get_bind_data(  
duckdb_function_info info  
);
```

**Parameters**
- **info**
The info object
- **returns**
The bind data object
duckdb_function_get_init_data  Gets the init data set by duckdb_init_set_init_data during the init.

Syntax

```c
void *duckdb_function_get_init_data(
    duckdb_function_info info
);
```

Parameters

- `info`
  The info object

- `returns`
  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

```c
void *duckdb_function_get_local_init_data(
    duckdb_function_info info
);
```

Parameters

- `info`
  The info object

- `returns`
  The init data object

duckdb_function_set_error  Report that an error has occurred while executing the function.

Syntax

```c
void duckdb_function_set_error(
    duckdb_function_info info,
    const char *error
);
```
DuckDB Documentation

Parameters

• info
  The info object

• error
  The error message

**duckdb_add_replacement_scan**  Add a replacement scan definition to the specified database

**Syntax**

```c
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 to use. 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**

```c
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**

```c
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**

```c
void duckdb_replacement_scan_set_error(
    duckdb_replacement_scan_info info,
    const char *error
);
```

**Parameters**

- info

The info object

- error

The error message
**duckdb_appender_create**  Creates an appender object.

**Syntax**

```c
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.

**returns**

DuckDBSuccess on success or DuckDBError on failure.

**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**

```c
const char *duckdb_appender_error(
    duckdb_appender appender
);
```
Parameters

- **appender**
The appender to get the error from.

- **returns**
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 and the data to be appended to the base table.
This should generally not be used unless you know what you are doing. Instead, call `duckdb_appender_destroy` when you are done with the appender.

Syntax

```c
duckdb_state duckdb_appender_flush(
    duckdb_appender appender
);
```

Parameters

- **appender**
The appender to flush.

- **returns**
DuckDBSuccess on success or DuckDBError on failure.

**duckdb_appender_close**  Close the appender, flushing all intermediate state in the appender to the table and closing it for further appends.
This is generally not necessary. Call `duckdb_appender_destroy` instead.

Syntax

```c
duckdb_state duckdb_appender_close(
    duckdb_appender appender
);
```

Parameters

- **appender**
The appender to flush and close.

- **returns**
DuckDBSuccess on success or DuckDBError on failure.
**duckdb_appender_destroy**  Close the appender and destroy it. Flushing all intermediate state in the appender to the table, and de-allocating all memory associated with the appender.

**Syntax**

```c
duckdb_state duckdb_appender_destroy(
    duckdb_appender *appender
);
```

**Parameters**

- **appender**
  The appender to flush, close and destroy.

- **returns**
  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**

```c
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**

```c
duckdb_state duckdb_appender_end_row(
    duckdb_appender appender
);
```

**Parameters**

- **appender**
  The appender.

- **returns**
  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**

```c
duckdb_state duckdb_append_hugeint(
    duckdb_appender appender,
    duckdb_hugeint value
);
```

**duckdb_append_uint8**  Append a uint8_t value to the appender.

**Syntax**

```c
duckdb_state duckdb_append_uint8(
    duckdb_appender appender,
    uint8_t value
);
```

**duckdb_append_uint16**  Append a uint16_t value to the appender.

**Syntax**

```c
duckdb_state duckdb_append_uint16(
    duckdb_appender appender,
    uint16_t value
);
```

**duckdb_append_uint32**  Append a uint32_t value to the appender.

**Syntax**

```c
duckdb_state duckdb_append_uint32(
    duckdb_appender appender,
    uint32_t value
);
```

**duckdb_append_uint64**  Append a uint64_t value to the appender.

**Syntax**

```c
duckdb_state duckdb_append_uint64(
    duckdb_appender appender,
    uint64_t value
);
```
**duckdb_append_uhugeint**  Append a duckdb_uhugeint value to the appender.

**Syntax**

```c
duckdb_state duckdb_append_uhugeint(
    duckdb_appender appender,
    duckdb_uhugeint value
);
```

**duckdb_append_float**  Append a float value to the appender.

**Syntax**

```c
duckdb_state duckdb_append_float(
    duckdb_appender appender,
    float value
);
```

**duckdb_append_double**  Append a double value to the appender.

**Syntax**

```c
duckdb_state duckdb_append_double(
    duckdb_appender appender,
    double value
);
```

**duckdb_append_date**  Append a duckdb_date value to the appender.

**Syntax**

```c
duckdb_state duckdb_append_date(
    duckdb_appender appender,
    duckdb_date value
);
```

**duckdb_append_time**  Append a duckdb_time value to the appender.

**Syntax**

```c
duckdb_state duckdb_append_time(
    duckdb_appender appender,
    duckdb_time value
);
```
**duckdb_append_timestamp**  Append a duckdb_timestamp value to the appender.

**Syntax**

```c
duckdb_state duckdb_append_timestamp(
    duckdb_appender appender,
    duckdb_timestamp value
);
```

**duckdb_append_interval**  Append a duckdb_interval value to the appender.

**Syntax**

```c
duckdb_state duckdb_append_interval(
    duckdb_appender appender,
    duckdb_interval value
);
```

**duckdb_append_varchar**  Append a varchar value to the appender.

**Syntax**

```c
duckdb_state duckdb_append_varchar(
    duckdb_appender appender,
    const char *val
);
```

**duckdb_append_varchar_length**  Append a varchar value to the appender.

**Syntax**

```c
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

```c
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

```c
duckdb_state duckdb_append_null(
    duckdb_appender appender
);
```

---

**duckdb_append_data_chunk**  Appends a pre-filled data chunk to the specified appender.

The types of the data chunk must exactly match the types of the table, no casting is performed. If the types do not match or the appender is in an invalid state, DuckDBError is returned. If the append is successful, DuckDB-Success is returned.

Syntax

```c
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.

- **returns**
  The return state.

---

**duckdb_query_arrow**  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.

• returns
  DuckDBSuccess on success or DuckDBError on failure.


duckdb_query_arrow_schema
  Fetch the internal arrow schema from the arrow result.

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.

• returns
  DuckDBSuccess on success or DuckDBError on failure.


duckdb_prepared_arrow_schema
  Fetch the internal arrow schema from the prepared statement.
### Syntax

```c
duckdb_state duckdb_prepared_arrow_schema(
    duckdb_prepared_statement prepared,
    duckdb_arrow_schema *out_schema
);
```

**Parameters**

- **result**
The prepared statement to fetch the schema from.
  - **out_schema**
The output schema.

**Returns**

DuckDBSuccess on success or DuckDBError on failure.

---

### Syntax

```c
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.

---

### Syntax

```c
void duckdb_query_arrow_array
```

**Parameters**

- **out_array**

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.
DuckDB Documentation

**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.

  *returns*
  DuckDBSuccess on success or DuckDBError on failure.

---

**duckdb_arrow_column_count**  Returns the number of columns present in a the arrow result object.

**Syntax**

idx_t duckdb_arrow_column_count(
   duckdb_arrow result
);

**Parameters**

- **result**
  The result object.

  *returns*
  The number of columns present in the result object.

---

**duckdb_arrow_row_count**  Returns the number of rows present in a the arrow result object.

**Syntax**

idx_t duckdb_arrow_row_count(
   duckdb_arrow result
);
Parameters

• result
  The result object.

  • returns
  The number of rows present in the result object.

duckdb_arrow_rows_changed

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.

  • returns
  The number of rows changed.

duckdb_query_arrow_error

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 nullmask from.

  • returns
  The error of the result.
duckdb_destroy_arrow  Closes the result and de-allocates all memory allocated for the arrow result.

Syntax

```c
void duckdb_destroy_arrow(
    duckdb_arrow *result
);
```

Parameters

- `result`
  
The result to destroy.

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

```c
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 should be called on the result in order to free memory.

Syntax

```c
duckdb_task_state duckdb_create_task_state(
    duckdb_database database
);
```
Parameters

- database
The database object to create the task state for
  - returns
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

```c
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

```c
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
returns
The amount of tasks that have actually been executed

duckdb_finish_execution Finish execution on a specific task.

Syntax

```c
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

```c
bool duckdb_task_state_is_finished(
    duckdb_task_state state
);
```

Parameters

- `state` The task state to inspect
  - `returns` 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

```c
void duckdb_destroy_task_state(
    duckdb_task_state state
);
```
**Parameters**

- **state**

The task state to clean up

**duckdb_execution_is_finished** Returns true if execution of the current query is finished.

**Syntax**

```c
bool duckdb_execution_is_finished(
    duckdb_connection con
);
```

**Parameters**

- **con**

The connection on which to check

**duckdb_stream_fetch_chunk** 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_result` 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**

```c
duckdb_data_chunk duckdb_stream_fetch_chunk(
    duckdb_result result
);
```

**Parameters**

- **result**

The result object to fetch the data chunk from.

- **returns**

The resulting data chunk. Returns NULL if the result has an error.
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.

Note. The standard source distribution of libduckdb contains an "amalgamation" of the DuckDB sources, which combine all sources into two files duckdb.hpp and duckdb.cpp. The duckdb.hpp header is much larger in this case. Regardless of whether you are using the amalgamation or not, just include 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)");

MaterializedQueryResult result = con.Query("SELECT * FROM integers");
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:

```cpp
std::unique_ptr<PreparedStatement> prepare = con.Prepare("SELECT count(*) FROM a WHERE i = $1");
std::unique_ptr<QueryResult> result = prepare->Execute(12);
```

**Note.** 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 create 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:

```cpp
bool bigger_than_four(int value) {
    return value > 4;
}
```

```cpp
connection.CreateScalarFunction<bool, int>("bigger_than_four", &bigger_than_four);
```

```cpp
connection.Query("SELECT bigger_than_four(i) FROM (VALUES(3), (5)) tbl(i)")->Print();
```

The `CreateScalarFunction()` methods automatically create 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...))`
DuckDB Documentation

- **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.

```cpp
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:

```cpp
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<
tf32_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 `TRtype`;
• **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:

- `LogicalTypeld::BOOLEAN` → `bool`
- `LogicalTypeld::TINYINT` → `int8_t`
- `LogicalTypeld::SMALLINT` → `int16_t`
- `LogicalTypeld::DATE`, `LogicalTypeld::TIME`, `LogicalTypeld::INTEGER` → `int32_t`
- `LogicalTypeld::BIGINT`, `LogicalTypeld::TIMESTAMP` → `int64_t`
- `LogicalTypeld::FLOAT`, `LogicalTypeld::DOUBLE`, `LogicalTypeld::DECIMAL` → `double`
- `LogicalTypeld::VARCHAR`, `LogicalTypeld::CHAR`, `LogicalTypeld::BLOB` → `string_t`
- `LogicalTypeld::VARBINARY` → `blob_t`

**CreateVectorizedFunction**  The `CreateVectorizedFunction()` methods register a vectorized UDF such as:

```cpp
/*
 * 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.CreateInstance<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 \textit{scalar\_function\_t}:

\begin{verbatim}
typedef std::function<void(DataChunk &args, ExpressionState &expr, Vector &result)>
    scalar_function_t;
\end{verbatim}

\begin{itemize}
\item \textbf{args} is a \textit{DataChunk} that holds a set of input vectors for the UDF that all have the same length;
\item \textbf{expr} is an \textit{ExpressionState} that provides information to the query’s expression state;
\item \textbf{result} is a \textit{Vector} to store the result values.
\end{itemize}

There are different vector types to handle in a Vectorized UDF:

\begin{itemize}
\item ConstantVector;
\item DictionaryVector;
\item FlatVector;
\item ListVector;
\item StringVector;
\item StructVector;
\item SequenceVector.
\end{itemize}

The general API of the \texttt{CreateVectorizedFunction()} method is as follows:

1. \texttt{template<typename TR, typename... Args>
   void CreateVectorizedFunction(string name, scalar\_function\_t udf\_func, LogicalType
   varargs = LogicalType::INVALID)}

\begin{itemize}
\item template parameters:
  \begin{itemize}
  \item \textbf{TR} is the return type of the UDF function;
  \item \textbf{Args} are the arguments up to 3 for the UDF function.
  \end{itemize}
\item \textbf{name} is the name to register the UDF function;
\item \textbf{udf\_func} is a \textit{vectorized} UDF function;
\item \textbf{varargs} The type of varargs to support, or LogicalType::INVALID (default value) if the function does not accept variable length arguments.
\end{itemize}

This method automatically discovers from the template typenames the corresponding LogicalTypes:

\begin{itemize}
\item \texttt{bool} \rightarrow LogicalType::BOOLEAN;
\item \texttt{int8\_t} \rightarrow LogicalType::TINYINT;
\item \texttt{int16\_t} \rightarrow LogicalType::SMALLINT
\item \texttt{int32\_t} \rightarrow LogicalType::INTEGER
\item \texttt{int64\_t} \rightarrow LogicalType::BIGINT
\end{itemize}
• float → LogicalType::FLOAT
• double → LogicalType::DOUBLE
• string_t → LogicalType::VARCHAR

2.

```c++
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 API**

**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).

Note. DuckDB has a tldr page that 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

v0.9.2 3c695d7ba9
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.

To open or create a persistent database, simply include a path as a command line argument like duckdb path/to/my_database.duckdb. 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 any arbitrary extension, but .db or .duckdb are two common choices. Running on a persistent database allows spilling to disk, thus facilitating larger-than-memory workloads (i.e. out-of-core-processing).

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;
```

<table>
<thead>
<tr>
<th>my_column</th>
</tr>
</thead>
<tbody>
<tr>
<td>varchar</td>
</tr>
<tr>
<td>quack</td>
</tr>
</tbody>
</table>

The CLI supports all of DuckDB's rich SQL syntax including SELECT, CREATE, and ALTER statements.

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 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.
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
```

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.

<table>
<thead>
<tr>
<th>generate_series</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>generate_series</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

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).
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, pipe 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 :memory: "SELECT * FROM read_csv('/dev/stdin')"
To write back to stdout, the copy command can be used with the /dev/stdout file location.

$ cat test.csv | duckdb :memory: "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:

```sql
SELECT getenv('HOME') AS home;
```

<table>
<thead>
<tr>
<th>home</th>
</tr>
</thead>
<tbody>
<tr>
<td>varchar</td>
</tr>
<tr>
<td>/Users/user_name</td>
</tr>
</tbody>
</table>

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:

```sql
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` 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.
The table below summarizes DuckDB's command line options. To list all command line options, use the command `duckdb -help`.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-append</td>
<td>Append the database to the end of the file</td>
</tr>
<tr>
<td>-ascii</td>
<td>Set output mode to ascii</td>
</tr>
<tr>
<td>-bail</td>
<td>Stop after hitting an error</td>
</tr>
<tr>
<td>-batch</td>
<td>Force batch I/O</td>
</tr>
<tr>
<td>-box</td>
<td>Set output mode to box</td>
</tr>
<tr>
<td>-column</td>
<td>Set output mode to column</td>
</tr>
<tr>
<td>-cmd COMMAND</td>
<td>Run COMMAND before reading stdin</td>
</tr>
<tr>
<td>-c COMMAND</td>
<td>Run COMMAND and exit</td>
</tr>
<tr>
<td>-csv</td>
<td>Set output mode to csv</td>
</tr>
<tr>
<td>-echo</td>
<td>Print commands before execution</td>
</tr>
<tr>
<td>-init FILENAME</td>
<td>Run the script in FILENAME upon startup (instead of ~/.duckdbrc)</td>
</tr>
<tr>
<td>-header</td>
<td>Turn headers on</td>
</tr>
<tr>
<td>-help</td>
<td>Show this message</td>
</tr>
<tr>
<td>-html</td>
<td>Set output mode to HTML</td>
</tr>
<tr>
<td>-interactive</td>
<td>Force interactive I/O</td>
</tr>
<tr>
<td>-json</td>
<td>Set output mode to json</td>
</tr>
<tr>
<td>-line</td>
<td>Set output mode to line</td>
</tr>
<tr>
<td>-list</td>
<td>Set output mode to list</td>
</tr>
<tr>
<td>-markdown</td>
<td>Set output mode to markdown</td>
</tr>
<tr>
<td>-newline SEP</td>
<td>Set output row separator. Default: \n</td>
</tr>
<tr>
<td>-nofollow</td>
<td>Refuse to open symbolic links to database files</td>
</tr>
<tr>
<td>-noheader</td>
<td>Turn headers off</td>
</tr>
<tr>
<td>-no-stdin</td>
<td>Exit after processing options instead of reading stdin</td>
</tr>
<tr>
<td>-nullvalue TEXT</td>
<td>Set text string for NULL values. Default: empty string</td>
</tr>
<tr>
<td>-quote</td>
<td>Set output mode to quote</td>
</tr>
<tr>
<td>-readonly</td>
<td>Open the database read-only</td>
</tr>
<tr>
<td>-s COMMAND</td>
<td>Run COMMAND and exit</td>
</tr>
<tr>
<td>-separator SEP</td>
<td>Set output column separator to SEP. Default:</td>
</tr>
</tbody>
</table>
DuckDB Documentation

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-stats</td>
<td>Print memory stats before each finalize</td>
</tr>
<tr>
<td>-table</td>
<td>Set output mode to table</td>
</tr>
<tr>
<td>-unsigned</td>
<td>Allow loading of unsigned extensions</td>
</tr>
<tr>
<td>-version</td>
<td>Show DuckDB version</td>
</tr>
</tbody>
</table>

**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.

**Dot Commands**

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.bail on</td>
<td>off</td>
</tr>
<tr>
<td>.binary on</td>
<td>off</td>
</tr>
<tr>
<td>.cd DIRECTORY</td>
<td>Change the working directory to DIRECTORY</td>
</tr>
<tr>
<td>.changes on</td>
<td>off</td>
</tr>
<tr>
<td>.check GLOB</td>
<td>Fail if output since .testcase does not match</td>
</tr>
<tr>
<td>.columns</td>
<td>Column-wise rendering of query results</td>
</tr>
<tr>
<td>.constant ?COLOR?</td>
<td>Sets the syntax highlighting color used for constant values</td>
</tr>
<tr>
<td>.constantcode ?CODE?</td>
<td>Sets the syntax highlighting terminal code used for constant values</td>
</tr>
<tr>
<td>.databases</td>
<td>List names and files of attached databases</td>
</tr>
<tr>
<td>.echo on</td>
<td>off</td>
</tr>
<tr>
<td>.excel</td>
<td>Display the output of next command in spreadsheet</td>
</tr>
<tr>
<td>.exit ?CODE?</td>
<td>Exit this program with return-code CODE</td>
</tr>
<tr>
<td>.explain ?on</td>
<td>off</td>
</tr>
<tr>
<td>.fullschema ?--indent?</td>
<td>Show schema and the content of sqlite_stat tables</td>
</tr>
<tr>
<td>.headers on</td>
<td>off</td>
</tr>
<tr>
<td>.help ?-all? ?PATTERN?</td>
<td>Show help text for PATTERN</td>
</tr>
<tr>
<td>Command</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>.highlight [on</td>
<td>off]</td>
</tr>
<tr>
<td>.import FILE TABLE</td>
<td>Import data from FILE into TABLE</td>
</tr>
<tr>
<td>.indexes ?TABLE?</td>
<td>Show names of indexes</td>
</tr>
<tr>
<td>.keyword ?COLOR?</td>
<td>Sets the syntax highlighting color used for keywords</td>
</tr>
<tr>
<td>.keywordcode ?CODE?</td>
<td>Sets the syntax highlighting terminal code used for keywords</td>
</tr>
<tr>
<td>.lint OPTIONS</td>
<td>Report potential schema issues.</td>
</tr>
<tr>
<td>.log FILE</td>
<td>off</td>
</tr>
<tr>
<td>.maxrows COUNT</td>
<td>Sets the maximum number of rows for display. Only for duckbox mode</td>
</tr>
<tr>
<td>.maxwidth COUNT</td>
<td>Sets the maximum width in characters. 0 defaults to terminal width. Only for duckbox mode</td>
</tr>
<tr>
<td>.mode MODE ?TABLE?</td>
<td>Set output mode</td>
</tr>
<tr>
<td>.nullvalue STRING</td>
<td>Use STRING in place of NULL values</td>
</tr>
<tr>
<td>.once ?OPTIONS? ?FILE?</td>
<td>Output for the next SQL command only to FILE</td>
</tr>
<tr>
<td>.output ?FILE?</td>
<td>Send output to FILE or stdout if FILE is omitted</td>
</tr>
<tr>
<td>.parameter CMD ...</td>
<td>Manage SQL parameter bindings</td>
</tr>
<tr>
<td>.print STRING...</td>
<td>Print literal STRING</td>
</tr>
<tr>
<td>.prompt MAIN CONTINUE</td>
<td>Replace the standard prompts</td>
</tr>
<tr>
<td>.quit</td>
<td>Exit this program</td>
</tr>
<tr>
<td>.read FILE</td>
<td>Read input from FILE</td>
</tr>
<tr>
<td>.rows</td>
<td>Row-wise rendering of query results (default)</td>
</tr>
<tr>
<td>.schema ?PATTERN?</td>
<td>Show the CREATE statements matching PATTERN</td>
</tr>
<tr>
<td>.separator COL ?ROW?</td>
<td>Change the column and row separators</td>
</tr>
<tr>
<td>.sha3sum ...</td>
<td>Compute a SHA3 hash of database content</td>
</tr>
<tr>
<td>.shell CMD ARGS...</td>
<td>Run CMD ARGS... in a system shell</td>
</tr>
<tr>
<td>.show</td>
<td>Show the current values for various settings</td>
</tr>
<tr>
<td>.system CMD ARGS...</td>
<td>Run CMD ARGS... in a system shell</td>
</tr>
<tr>
<td>.tables ?TABLE?</td>
<td>List names of tables matching LIKE pattern TABLE</td>
</tr>
<tr>
<td>.testcase NAME</td>
<td>Begin redirecting output to NAME</td>
</tr>
<tr>
<td>.timer on</td>
<td>off</td>
</tr>
<tr>
<td>.width NUM1 NUM2 ...</td>
<td>Set minimum column widths for columnar output</td>
</tr>
</tbody>
</table>
DuckDB Documentation

Using the .help Command

The .help text may be filtered by passing in a text string as the second 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;
```

```
SELECT 'back to the terminal' AS displayed_column;
```

The file `my_results.md` will then contain:

```
<table>
<thead>
<tr>
<th>output_column</th>
</tr>
</thead>
<tbody>
<tr>
<td>taking flight</td>
</tr>
</tbody>
</table>
```

The terminal will then display:

```
<table>
<thead>
<tr>
<th>displayed_column</th>
</tr>
</thead>
<tbody>
<tr>
<td>back to the terminal</td>
</tr>
</tbody>
</table>
```

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:

### 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);
```

### Configuring the 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:
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 INT, col_2 INT);
.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.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ascii</td>
<td>Columns/rows delimited by 0x1F and 0x1E</td>
</tr>
<tr>
<td>box</td>
<td>Tables using unicode box-drawing characters</td>
</tr>
<tr>
<td>csv</td>
<td>Comma-separated values</td>
</tr>
<tr>
<td>column</td>
<td>Output in columns. (See <code>.width</code>)</td>
</tr>
<tr>
<td>duckbox</td>
<td>Tables with extensive features</td>
</tr>
<tr>
<td>html</td>
<td>HTML <code>&lt;table&gt;</code> code</td>
</tr>
<tr>
<td>insert</td>
<td>SQL insert statements for TABLE</td>
</tr>
<tr>
<td>json</td>
<td>Results in a JSON array</td>
</tr>
<tr>
<td>jsonlines</td>
<td>Results in a NDJSON</td>
</tr>
<tr>
<td>latex</td>
<td>LaTeX <code>\text{tabular}</code> environment code</td>
</tr>
<tr>
<td>line</td>
<td>One value per line</td>
</tr>
<tr>
<td>list</td>
<td>Values delimited by &quot;</td>
</tr>
<tr>
<td>markdown</td>
<td>Markdown table format</td>
</tr>
<tr>
<td>quote</td>
<td>Escape answers as for SQL</td>
</tr>
<tr>
<td>table</td>
<td>ASCII-art table</td>
</tr>
<tr>
<td>tabs</td>
<td>Tab-separated values</td>
</tr>
<tr>
<td>tcl</td>
<td>TCL list elements</td>
</tr>
<tr>
<td>trash</td>
<td>No output</td>
</tr>
</tbody>
</table>

```
.mode markdown
SELECT 'quacking intensifies' AS incoming_ducks;

<table>
<thead>
<tr>
<th>incoming_ducks</th>
<th>quacking intensifies</th>
</tr>
</thead>
<tbody>
<tr>
<td>-----------------</td>
<td>----------------------</td>
</tr>
</tbody>
</table>
```

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.
Go

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.

Note. The Go client is provided as a third-party library.

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"
)
```

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 API

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 like so:

```java
Class.forName("org.duckdb.DuckDBDriver");
```

To create a DuckDB connection, call `DriverManager` with the `jdbc:duckdb:` JDBC URL prefix, like so:

```java
Connection conn = DriverManager.getConnection("jdbc:duckdb:");
```

To use DuckDB-specific features such as the `appender`, cast the object to a `DuckDBConnection`:

```java
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:

```java
Properties ro_prop = new Properties();
ro_prop.setProperty("duckdb.read_only", "true");
Connection conn_ro = DriverManager.getConnection("jdbc:duckdb:/tmp/my_database", ro_prop);
```
Additional connections can be created using the `DriverManager`. A more efficient mechanism is to call the `DuckDBConnection#duplicate()` method like so:

```java
Connection conn2 = ((DuckDBConnection) conn).duplicate();
```

Multiple connections are allowed, but mixing read-write and read-only connections is unsupported.

**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.

```java
// create a table
Statement stmt = conn.createStatement();
stmt.execute("CREATE TABLE items (item VARCHAR, value DECIMAL(10, 2), count INTEGER)"lobals are prepared statements as per the JDBC API:

```java
try (PreparedStatement p_stmt = conn.prepareStatement("INSERT INTO items VALUES (?,?,?)");)
    p_stmt.setString(1, "chainsaw");
    p_stmt.setDouble(2, 500.0);
    p_stmt.setInt(3, 42);
    p_stmt.executeUpdate();

    // more calls to execute() possible
```

**Note.** 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

```java
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 p_stmt = conn.prepareStatement("SELECT * FROM generate_series(2000)");
    var resultset = (DuckDBResultSet) p_stmt.executeQuery();
    var allocator = new RootAllocator();
    try {
        ArrowReader reader = resultset.arrowExportStream(allocator, 256); {
            while (reader.loadNextBatch()) {
                System.out.println(reader.getVectorSchemaRoot().getVector("generate_series");
            }
        }
    }
}
```

**Arrow Import**  The following demonstrates consuming an arrow stream from the java arrow bindings

```java
import org.apache.arrow.memory.RootAllocator;
import org.apache.arrow.vector.ipc.ArrowReader;
import org.duckdb.DuckDBConnection;

// arrow stuff
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 stuff
    try {
        var conn = (DuckDBConnection) DriverManager.getConnection("jdbc:duckdb:"); {
            conn.registerArrowStream("adsf", arrow_array_stream);

            // run a query
            try {
                var stmt = conn.createStatement();
                var rs = (DuckDBResultSet) stmt.executeQuery("SELECT count(*) FROM adsf"); {
                    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 org.duckdb.DuckDBConnection;

DuckDBConnection conn = (DuckDBConnection)
DriverManager.getConnection("jdbc:duckdb:");
Statement 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();
}

Julia Package

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:
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 using 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)

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.

using DuckDB
using DataFrames

# create a new in-memory database
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)
Original Julia Connector

Credits to kimmolinna for the original DuckDB Julia connector.

Node.js

Node.js API

This 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:

```javascript
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.

```javascript
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.

```javascript
db.all('SELECT 42 AS fortytwo', function(err, res) {
  if (err) {
    throw err;
  }
  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:
DuckDB Documentation

db.all('SELECT ?:INTEGER AS fortytwo, ?:STRING AS hello', 42, 'Hello, World',
  function(err, res) {
    if (err) {
      throw err;
    }
    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) {
    throw err;
  }
  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) {
    throw err;
  }
  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) {
        throw err;
    }
    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) {
            throw err;
        }
        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) {
        throw err;
    }

    const arrowTable = arrow.tableFromJSON(jsonData);
    db.register_buffer("jsonDataTable", [arrow.tableToIPC(arrowTable)], true, (err, res) => {
        if (err) {
            throw err;
        }

        // `SELECT * FROM jsonDataTable` would return the entries in `jsonData`
    });
});
Loading Unsigned Extensions

To load unsigned extensions, instantiate the database as follows:

```javascript
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
  * `.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
  * `.arrowPCAll(sql, ...params, callback)` ⇒ void
  * `.arrowPCStream(sql, ...params, callback)` ⇒ void
  * `.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`

<table>
<thead>
<tr>
<th>Param</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sql</td>
<td></td>
</tr>
<tr>
<td>...params</td>
<td>*</td>
</tr>
<tr>
<td>callback</td>
<td></td>
</tr>
</tbody>
</table>

**connection.all(sql, ...params, callback) => void**  
Run a SQL query and triggers the callback once for all result rows  
**Kind:** instance method of `Connection`

<table>
<thead>
<tr>
<th>Param</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sql</td>
<td></td>
</tr>
<tr>
<td>...params</td>
<td>*</td>
</tr>
<tr>
<td>callback</td>
<td></td>
</tr>
</tbody>
</table>

**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`

<table>
<thead>
<tr>
<th>Param</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sql</td>
<td></td>
</tr>
<tr>
<td>...params</td>
<td>*</td>
</tr>
<tr>
<td>callback</td>
<td></td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Param</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sql</td>
<td></td>
</tr>
<tr>
<td>...params</td>
<td>*</td>
</tr>
<tr>
<td>callback</td>
<td></td>
</tr>
</tbody>
</table>

connection.each(sql, ...params, callback) ⇒ void Runs a SQL query and triggers the callback for each result row

**Kind**: instance method of Connection

<table>
<thead>
<tr>
<th>Param</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sql</td>
<td></td>
</tr>
<tr>
<td>...params</td>
<td>*</td>
</tr>
<tr>
<td>callback</td>
<td></td>
</tr>
</tbody>
</table>

connection.stream(sql, ...params) **Kind**: instance method of Connection

<table>
<thead>
<tr>
<th>Param</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sql</td>
<td></td>
</tr>
<tr>
<td>...params</td>
<td>*</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Param</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
</tr>
<tr>
<td>return_type</td>
</tr>
<tr>
<td>fun</td>
</tr>
</tbody>
</table>
**DuckDB Documentation**

**connection.prepare(sql, ...params, callback) ⇒ Statement**  
Prepare a SQL query for execution  

**Kind:** instance method of **Connection**

---

<table>
<thead>
<tr>
<th>Param</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sql</td>
<td></td>
</tr>
<tr>
<td>...params</td>
<td>*</td>
</tr>
<tr>
<td>callback</td>
<td></td>
</tr>
</tbody>
</table>

**connection.exec(sql, ...params, callback) ⇒ void**  
Execute a SQL query  

**Kind:** instance method of **Connection**

---

<table>
<thead>
<tr>
<th>Param</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sql</td>
<td></td>
</tr>
<tr>
<td>...params</td>
<td>*</td>
</tr>
<tr>
<td>callback</td>
<td></td>
</tr>
</tbody>
</table>

**connection.register_udf_bulk(name, return_type, callback) ⇒ void**  
Register a User Defined Function  

**Kind:** instance method of **Connection**

---

<table>
<thead>
<tr>
<th>Param</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td></td>
</tr>
<tr>
<td>return_type</td>
<td></td>
</tr>
<tr>
<td>callback</td>
<td></td>
</tr>
</tbody>
</table>

**connection.unregister_udf(name, return_type, callback) ⇒ void**  
Unregister a User Defined Function  

**Kind:** instance method of **Connection**

---

<table>
<thead>
<tr>
<th>Param</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td></td>
</tr>
<tr>
<td>return_type</td>
<td></td>
</tr>
<tr>
<td>callback</td>
<td></td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Param</th>
<th>name</th>
<th>array</th>
<th>force</th>
<th>callback</th>
</tr>
</thead>
</table>

connection.unregister_buffer(name, callback) ⇒ void
Unregister the Buffer

**Kind:** instance method of Connection

<table>
<thead>
<tr>
<th>Param</th>
<th>name</th>
<th>callback</th>
</tr>
</thead>
</table>

connection.close(callback) ⇒ void
Closes connection

**Kind:** instance method of Connection

<table>
<thead>
<tr>
<th>Param</th>
<th>callback</th>
</tr>
</thead>
</table>

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`  

<table>
<thead>
<tr>
<th>Param</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sql</td>
<td></td>
</tr>
<tr>
<td>...params</td>
<td>*</td>
</tr>
<tr>
<td>callback</td>
<td></td>
</tr>
</tbody>
</table>

**statement.all(sql, ...params, callback) ⇒ void**  
*Kind*: instance method of `Statement`  

<table>
<thead>
<tr>
<th>Param</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sql</td>
<td></td>
</tr>
<tr>
<td>...params</td>
<td>*</td>
</tr>
<tr>
<td>callback</td>
<td></td>
</tr>
</tbody>
</table>

**statement.arrowIPCAll(sql, ...params, callback) ⇒ void**  
*Kind*: instance method of `Statement`  

<table>
<thead>
<tr>
<th>Param</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sql</td>
<td></td>
</tr>
<tr>
<td>...params</td>
<td>*</td>
</tr>
<tr>
<td>callback</td>
<td></td>
</tr>
</tbody>
</table>

**statement.each(sql, ...params, callback) ⇒ void**  
*Kind*: instance method of `Statement`
<table>
<thead>
<tr>
<th>Param</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sql</td>
<td></td>
</tr>
<tr>
<td>...params *</td>
<td></td>
</tr>
<tr>
<td>callback</td>
<td></td>
</tr>
</tbody>
</table>

**statement.finalize(sql, ...params, callback) ⇒ void**  
*Kind:* instance method of `Statement`

<table>
<thead>
<tr>
<th>Param</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sql</td>
<td></td>
</tr>
<tr>
<td>...params *</td>
<td></td>
</tr>
<tr>
<td>callback</td>
<td></td>
</tr>
</tbody>
</table>

**statement.stream(sql, ...params)**  
*Kind:* instance method of `Statement`

<table>
<thead>
<tr>
<th>Param</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sql</td>
<td></td>
</tr>
<tr>
<td>...params *</td>
<td></td>
</tr>
</tbody>
</table>

**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`

<table>
<thead>
<tr>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>path</td>
<td>path to database file or :memory: for in-memory database</td>
</tr>
<tr>
<td>access_mode</td>
<td>access mode</td>
</tr>
<tr>
<td>config</td>
<td>the configuration object</td>
</tr>
<tr>
<td>callback</td>
<td>callback function</td>
</tr>
</tbody>
</table>

---

- `~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`
  - `.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

<table>
<thead>
<tr>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>path</td>
<td>the database to connect to, either a file path, or <code>:memory:</code></td>
</tr>
</tbody>
</table>

**database.interrupt(callback) ⇒ void**  Supposedly interrupt queries, but currently does not do anything.

**Kind:** instance method of Database

<table>
<thead>
<tr>
<th>Param</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>callback</td>
<td></td>
</tr>
</tbody>
</table>

**database.prepare(sql) ⇒ Statement**  Prepare a SQL query for execution

**Kind:** instance method of Database

<table>
<thead>
<tr>
<th>Param</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>sql</td>
<td></td>
</tr>
</tbody>
</table>

**database.run(sql, ...params, callback) ⇒ void**  Convenience method for Connection#run using a built-in default connection

**Kind:** instance method of Database

<table>
<thead>
<tr>
<th>Param</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sql</td>
<td></td>
</tr>
<tr>
<td>...params</td>
<td>*</td>
</tr>
<tr>
<td>callback</td>
<td></td>
</tr>
</tbody>
</table>

**database.scanArrowIpc(sql, ...params, callback) ⇒ void**  Convenience method for Connection#scanArrowIpc using a built-in default connection

**Kind:** instance method of Database
**database.each(sql, ...params, callback) ⇒ void**  
Kind: instance method of `Database`

<table>
<thead>
<tr>
<th>Param</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sql</td>
<td></td>
</tr>
<tr>
<td>...params</td>
<td>*</td>
</tr>
<tr>
<td>callback</td>
<td></td>
</tr>
</tbody>
</table>

**database.all(sql, ...params, callback) ⇒ void**  
Convenience method for Connection#apply using a built-in default connection

Kind: instance method of `Database`

<table>
<thead>
<tr>
<th>Param</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sql</td>
<td></td>
</tr>
<tr>
<td>...params</td>
<td>*</td>
</tr>
<tr>
<td>callback</td>
<td></td>
</tr>
</tbody>
</table>

**database.arrowIPCAll(sql, ...params, callback) ⇒ void**  
Convenience method for Connection#arrowIPCAll using a built-in default connection

Kind: instance method of `Database`

<table>
<thead>
<tr>
<th>Param</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sql</td>
<td></td>
</tr>
<tr>
<td>...params</td>
<td>*</td>
</tr>
<tr>
<td>callback</td>
<td></td>
</tr>
</tbody>
</table>

**database.arrowIPCStream(sql, ...params, callback) ⇒ void**  
Convenience method for Connection#arrowIPCStream using a built-in default connection

<table>
<thead>
<tr>
<th>Param</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sql</td>
<td></td>
</tr>
<tr>
<td>...params</td>
<td>*</td>
</tr>
<tr>
<td>callback</td>
<td></td>
</tr>
</tbody>
</table>
**DuckDB Documentation**

**Kind**: instance method of `Database`

<table>
<thead>
<tr>
<th>Param</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sql</td>
<td></td>
</tr>
<tr>
<td>...params</td>
<td>*</td>
</tr>
<tr>
<td>callback</td>
<td></td>
</tr>
</tbody>
</table>

`database.exec(sql, ...params, callback) ⇒ void`  
**Kind**: instance method of `Database`

<table>
<thead>
<tr>
<th>Param</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sql</td>
<td></td>
</tr>
<tr>
<td>...params</td>
<td>*</td>
</tr>
<tr>
<td>callback</td>
<td></td>
</tr>
</tbody>
</table>

`database.register_udf(name, return_type, fun) ⇒ this`  
Register a User Defined Function

Convenience method for `Connection#register_udf`

**Kind**: instance method of `Database`

<table>
<thead>
<tr>
<th>Param</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
</tr>
<tr>
<td>return_type</td>
</tr>
<tr>
<td>fun</td>
</tr>
</tbody>
</table>

`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`

<table>
<thead>
<tr>
<th>Param</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
</tr>
</tbody>
</table>
**database.unregister_buffer(name)** ⇒ this  
Unregister a Buffer  
Convenience method for Connection#unregister_buffer  
**Kind:** instance method of `Database`

<table>
<thead>
<tr>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td></td>
</tr>
</tbody>
</table>

**database.unregister_udf(name)** ⇒ this  
Unregister a UDF  
Convenience method for Connection#unregister_udf  
**Kind:** instance method of `Database`

<table>
<thead>
<tr>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td></td>
</tr>
</tbody>
</table>

**database.registerReplacementScan(fun)** ⇒ this  
Register a table replace scan function  
**Kind:** instance method of `Database`

<table>
<thead>
<tr>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fun</td>
<td>Replacement scan function</td>
</tr>
</tbody>
</table>

**database.tokenize(text)** ⇒ `ScriptTokens`  
Return positions and types of tokens in given text  
**Kind:** instance method of `Database`

<table>
<thead>
<tr>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>text</td>
<td></td>
</tr>
</tbody>
</table>

**database.get()**  
Not implemented  
**Kind:** instance method of `Database`

**duckdb~TokenType**  
Types of tokens return by `tokenize`.  
**Kind:** inner property of `duckdb`
**DuckDB Documentation**

**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**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>string</td>
<td>Column name</td>
</tr>
<tr>
<td>type</td>
<td>TypeInfo</td>
<td>Column type</td>
</tr>
</tbody>
</table>
### TypeInfo: object

**Kind:** global typedef  

**Properties**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>string</td>
<td>Type ID</td>
</tr>
<tr>
<td>[alias]</td>
<td>string</td>
<td>SQL type alias</td>
</tr>
<tr>
<td>sql_type</td>
<td>string</td>
<td>SQL type name</td>
</tr>
</tbody>
</table>

### DuckDbError: object

**Kind:** global typedef  

**Properties**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>errno</td>
<td>number</td>
<td>-1 for DuckDB errors</td>
</tr>
<tr>
<td>message</td>
<td>string</td>
<td>Error message</td>
</tr>
<tr>
<td>code</td>
<td>string</td>
<td>'DUCKDB_NODEJS_ERROR' for DuckDB errors</td>
</tr>
<tr>
<td>errorType</td>
<td>string</td>
<td>DuckDB error type code (eg, HTTP, IO, Catalog)</td>
</tr>
</tbody>
</table>

### HTTPError: object

**Kind:** global typedef  

**Extends:** DuckDbError  

**Properties**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>statusCode</td>
<td>number</td>
<td>HTTP response status code</td>
</tr>
<tr>
<td>reason</td>
<td>string</td>
<td>HTTP response reason</td>
</tr>
<tr>
<td>response</td>
<td>string</td>
<td>HTTP response body</td>
</tr>
<tr>
<td>headers</td>
<td>object</td>
<td>HTTP headers</td>
</tr>
</tbody>
</table>
### Python

### Python API

#### Installation

The DuckDB Python API can be installed using `pip`:

```bash
pip install duckdb
```

Please see the [installation page](...) for details. It is also possible to install DuckDB using `conda`:

```bash
conda install python-duckdb -c conda-forge
```

**Python version:** DuckDB requires Python 3.7 or newer. DuckDB v0.9 does not yet support Python 3.12. The next version, v0.10, will support Python 3.12.

#### Basic API Usage

The most straight-forward manner of running SQL queries using DuckDB is using the `duckdb.sql` command.

```python
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.

```python
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.

```python
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 also directly query Pandas DataFrames, Polars DataFrames and Arrow tables.

```python
import duckdb

duckdb.sql("SELECT * FROM pandas_df")
```

```python
import pandas as pd
pandas_df = pd.DataFrame({"a": [42]})
duckdb.sql("SELECT * FROM pandas_df")
```

```python
import polars as pl
polars_df = pl.DataFrame({"a": [42]})
duckdb.sql("SELECT * FROM polars_df")
```

```python
import pyarrow as pa
arrow_table = pa.Table.from_pydict({"a": [42]})
duckdb.sql("SELECT * FROM arrow_table")
```

Result Conversion

DuckDB supports converting query results efficiently to a variety of formats. See the [result conversion page](#) for more information.

```python
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.

```python
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
```

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()` function creates a connection to a persistent database. Any data written to that connection will be persisted, and can be reloaded by re-connecting to the same file, both from Python and from other DuckDB clients.

```python
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:

```python
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
```

**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.

Note that 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 DuckDB PyConnection 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:

```python
import duckdb
con = duckdb.connect()
con.install_extension("spatial")
con.load_extension("spatial")
```

Note. To load unsigned extensions, add the config={"allow_unsigned_extensions": "true"} argument to the duckdb.connect() method.

Data Ingestion

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.

```python
import duckdb
# 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"])
# use the (experimental) parallel CSV reader
duckdb.read_csv("example.csv", parallel = True)
# 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.

```python
import duckdb

# 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 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.

```python
import duckdb

# 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')")
```

DataFrames & Arrow Tables

DuckDB is automatically able to query a Pandas DataFrame, Polars DataFrame, or Arrow object that is stored in a Python variable by name. Accessing these is made possible by replacement scans.

DuckDB supports querying multiple types of Apache Arrow objects including tables, datasets, RecordBatchReaders, and scanners. See the Python guides for more examples.

```python
import duckdb
import pandas as pd
```
test_df = pd.DataFrame.from_dict({"i": [1, 2, 3, 4], "j": ["one", "two", "three", "four"]})
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:

```python
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"])
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):

```python
# 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.

```python
# example setting the sample size to 100000
duckdb.default_connection.execute("SET GLOBAL pandas_analyze_sample = 100000")
```

**Object Conversion**

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:

```python
my_list_value = [
    12345,
    "test"
]
```
Will become VARCHAR[] because 12345 can convert to VARCHAR but test cannot 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:

```python
default = {
    "key": [1, 2, 3],
    "value": ["one", "two", "three"
```

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):

```python
{1=one, 2=two, 3=three}
```

Note. The name of the fields matters and the two lists need to have the same size.

Otherwise we'll try to convert it to a STRUCT.

```python
my_struct_dict = {
    1: "one",
    "2": 2,
    "three": [1, 2, 3],
    False: True
```

Becomes:

```python
{1: one, 2: two, 3: three}
```

Note. 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'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

Below are some examples using this functionality. See the Python guides for more examples.

```python
# fetch as Pandas DataFrame
df = con.execute("SELECT * FROM items").fetchdf()
print(df)
#     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

# 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],
```
DuckDB Documentation

```python
# 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 the database. The connection object takes as a 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 Python process). If you would like to connect to an existing database in read-only mode, you can set the read_only flag to True. Read-only mode is required if multiple Python processes want to access the same database file at the same time.

By default we create an 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. You can also get a reference to this connection by providing the special value :default: to connect.

```python
import duckdb
duckdb.execute("CREATE TABLE tbl AS SELECT 42 a")
con = duckdb.connect(":default:"
con.sql("SELECT * FROM tbl")

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.
DuckDB also supports **prepared statements** in the API with the `execute` and `execute_many` 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.

Here are some examples:

```python
# insert a row using prepared statements
con.execute("INSERT INTO items VALUES (?, ?, ?)", ["laptop", 2000, 1])

# insert several rows using prepared statements
con.execute_many("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 $ notation for prepared statement and reused values
con.execute("SELECT $1, $1, $2", ["duck", "goose"])
print(con.fetchall())
# [('duck', 'duck', 'goose')]  
```

**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 string to value in the parameters argument.

An example use:

```python
import duckdb

res = duckdb.execute(""
SELECT
    $my_param,
    $other_param,
    $also_param
""",
    {
        "my_param": 5,
        "other_param": "DuckDB",
        "also_param": [42]
    }
).fetchall()
```
Note. Do not use `executemany` to insert large amounts of data into DuckDB. See the data ingestion page for better options.

### 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. They do not hold any data - and nothing is executed - until a method that triggers execution is called.

#### Constructing Relations

Relations can be created from SQL queries using the `duckdb.sql` method. Alternatively, they can be created from the various data ingestion methods (read_parquet, read_csv, read_json).

For example, here we create a relation from a SQL query:

```python
import duckdb
rel = duckdb.sql("SELECT * FROM range(10000000000) tbl(id)")
rel.show()
```

```
   id
  ---
  0
  1
  2
  3
  4
  5
  6
  7
  8
  .
  .
  .
9990
9991
9992
9993
9994
9995
9996
```
Note how we are constructing a relation that computes an immense amount of data (10B rows, or 74GB of data). The relation is constructed instantly - and we can even print the relation instantly.

When printing a relation using show or displaying it in 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).

### Data Ingestion

Outside of SQL queries, the following methods are provided to construct relation objects from external data.

- `from_arrow`
- `from_df`
- `read_csv`
- `read_json`
- `read_parquet`

### SQL Queries

Relation objects can be queried through SQL through so-called replacement scans. If you have a relation object stored in a variable, you can refer to that variable as if it was a SQL table (in the FROM clause). This allows you to incrementally build queries using relation objects.

```python
import duckdb
col = duckdb.sql("SELECT * FROM range(1000000) tbl(id)"
duckdb.sql("SELECT sum(id) FROM col").show()
```

<table>
<thead>
<tr>
<th>sum(id)</th>
</tr>
</thead>
<tbody>
<tr>
<td>int128</td>
</tr>
<tr>
<td>499999500000</td>
</tr>
</tbody>
</table>

### Operations

There are a number of operations that can be performed on relations. These are all short-hand for running the SQL queries - and will return relations again themselves.
**aggregate(expr, groups = {})**  Apply an (optionally grouped) aggregate over the relation. The system will automatically group by any columns that are not aggregates.

```python
import duckdb
rel = duckdb.sql("SELECT * FROM range(1000000) tbl(id)"
rel.aggregate("id % 2 AS g, sum(id), min(id), max(id)"
```

<table>
<thead>
<tr>
<th>g</th>
<th>sum(id)</th>
<th>min(id)</th>
<th>max(id)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>249999500000</td>
<td>0</td>
<td>999998</td>
</tr>
<tr>
<td>1</td>
<td>250000000000</td>
<td>1</td>
<td>999999</td>
</tr>
</tbody>
</table>

**except_(rel)**  Select all rows in the first relation, that do not occur in the second relation. The relations must have the same number of columns.

```python
import duckdb
r1 = duckdb.sql("SELECT * FROM range(10) tbl(id)"
r2 = duckdb.sql("SELECT * FROM range(5) tbl(id)"
r1.except_(r2).show()
```

<table>
<thead>
<tr>
<th>id</th>
<th>int64</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

**filter(condition)**  Apply the given condition to the relation, filtering any rows that do not satisfy the condition.

```python
import duckdb
rel = duckdb.sql("SELECT * FROM range(1000000) tbl(id)"
rel.filter("id > 5")\ limit(3).show()
```

<table>
<thead>
<tr>
<th>id</th>
<th>int64</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>
**intersect(rel)**  Select the intersection of two relations - returning all rows that occur in both relations. The relations must have the same number of columns.

```python
import duckdb
r1 = duckdb.sql("SELECT * FROM range(10) tbl(id)")
r2 = duckdb.sql("SELECT * FROM range(5) tbl(id)")
r1.intersect(r2).show()
```

```
+-----+
<table>
<thead>
<tr>
<th>id</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>
```

**join(rel, condition, type = "inner")**  Combine two relations, joining them based on the provided condition.

```python
import duckdb
r1 = duckdb.sql("SELECT * FROM range(5) tbl(id)" ).set_alias("r1")
r2 = duckdb.sql("SELECT * FROM range(10, 15) tbl(id)" ).set_alias("r2")
r1.join(r2, "r1.id + 10 = r2.id").show()
```

```
+-----+-----+
<table>
<thead>
<tr>
<th>id</th>
<th>id</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>14</td>
</tr>
</tbody>
</table>
```

**limit(n, offset = 0)**  Select the first n rows, optionally offset by offset.

```python
import duckdb
rel = duckdb.sql("SELECT * FROM range(1000000) tbl(id)")
rel.limit(3).show()
```

```
+-----+
<table>
<thead>
<tr>
<th>id</th>
</tr>
</thead>
</table>
```

316
order(expr)  Sort the relation by the given set of expressions.

```python
import duckdb
rel = duckdb.sql("SELECT * FROM range(1000000) tbl(id)")
rel.order("id DESC").limit(3).show()
```

```
<table>
<thead>
<tr>
<th>id</th>
</tr>
</thead>
<tbody>
<tr>
<td>999999</td>
</tr>
<tr>
<td>999998</td>
</tr>
<tr>
<td>999997</td>
</tr>
</tbody>
</table>
```

project(expr)  Apply the given expression to each row in the relation.

```python
import duckdb
rel = duckdb.sql("SELECT * FROM range(1000000) tbl(id)")
rel.project("id + 10 AS id_plus_ten").limit(3).show()
```

```
<table>
<thead>
<tr>
<th>id_plus_ten</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>12</td>
</tr>
</tbody>
</table>
```

union(rel)  Combine two relations, returning all rows in r1 followed by all rows in r2. The relations must have the same number of columns.

```python
import duckdb
r1 = duckdb.sql("SELECT * FROM range(5) tbl(id)")
r2 = duckdb.sql("SELECT * FROM range(10, 15) tbl(id)")
r1.union(r2).show()
```

```
<table>
<thead>
<tr>
<th>id</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>
```
The result of relations can be converted to various types of Python structures, see the result conversion page for more information.

The result of relations can also be directly written to files using the below methods.

- write_csv
- write_parquet

Python Function API

You can create a DuckDB user-defined function (UDF) out of 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.

```python
import duckdb
from duckdb.typing import *
from faker import Faker

def random_name():
    fake = Faker()
    return fake.name()

duckdb.create_function("random_name", random_name, [], VARCHAR)
res = duckdb.sql("SELECT random_name()").fetchall()
print(res)
# [("Gerald Ashley",)]
```

Creating Functions

To register a Python UDF, simply use the create_function method from a DuckDB connection. Here is the syntax:
import duckdb
con = duckdb.connect()
con.create_function(name, function, argument_type_list, return_type, type, null_handling)

The `create_function` method requires 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. **return_type**: Scalar functions return one element per row. This parameter specifies the return type of the function.
4. **parameters**: Scalar functions can operate on one or more columns. This parameter takes a list of column types used as input.
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 set 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:

```python
con.remove_function(name)
```

**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:

```python
import duckdb

def my_function(x: int) -> str:
    return x

duckdb.create_function("my_func", my_function)
duckdb.sql("SELECT my_func(42)")
```

If only the parameter list types can be inferred, you'll need to pass in None as `argument_type_list`.

# my_func(42)  
# varchar  
# 42  

If only the parameter list types can be inferred, you'll need to pass in None as `argument_type_list`.  

---

319
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".

```python
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,)]

duckdb.remove_function("dont_intercept")
duckdb.create_function("dont_intercept", dont_intercept_null, [BIGINT], BIGINT, null_handling="special")
res = duckdb.sql("SELECT dont_intercept(NULL)").fetchall()
print(res)
# [(5,)]
```

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)
# [(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
```

```python
count.counter = 0
```

If we create this function without marking it as having side effects, the result will be the following:

```python
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:

```python
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 Builtins  The table below shows the mapping of Python Builtin type to DuckDB type.

<table>
<thead>
<tr>
<th>Type</th>
<th>DuckDB Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>str</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>int</td>
<td>BIGINT</td>
</tr>
<tr>
<td>bytearray</td>
<td>BLOB</td>
</tr>
<tr>
<td>bytes</td>
<td>BLOB</td>
</tr>
<tr>
<td>float</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>bool</td>
<td>BOOLEAN</td>
</tr>
</tbody>
</table>

Numpy DTYPES  The table below shows the mapping of Numpy DType to DuckDB type.

<table>
<thead>
<tr>
<th>Type</th>
<th>DuckDB Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>BOOLEAN</td>
</tr>
</tbody>
</table>
## DuckDB Documentation

<table>
<thead>
<tr>
<th>Type</th>
<th>DuckDB Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>int8</td>
<td>TINYINT</td>
</tr>
<tr>
<td>int16</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>int32</td>
<td>INTEGER</td>
</tr>
<tr>
<td>int64</td>
<td>BIGINT</td>
</tr>
<tr>
<td>uint8</td>
<td>UTINYINT</td>
</tr>
<tr>
<td>uint16</td>
<td>USMALLINT</td>
</tr>
<tr>
<td>uint32</td>
<td>UINTINTEGER</td>
</tr>
<tr>
<td>uint64</td>
<td>UBIGINT</td>
</tr>
<tr>
<td>float32</td>
<td>FLOAT</td>
</tr>
<tr>
<td>float64</td>
<td>DOUBLE</td>
</tr>
</tbody>
</table>

### Nested Types

**list[child_type]**  
list type objects map to a LIST type of the child type. Which can also be arbitrarily nested.

```python
import duckdb
from typing import Union

duckdb.typing.DuckDBPyType([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.

```python
import duckdb
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.

```python
import duckdb
duckdb.typing.DuckDBPyType({'a': str, 'b': int})
# STRUCT(a VARCHAR, b BIGINT)
```
Union[<type_one>, ... <type_n>] typing.Union objects map to a UNION type of the provided types.

```python
import duckdb
from typing import Union

duckdb.typing.DuckDBPyType(Union[int, str, bool, bytearray])
# UNION(u1 BIGINT, u2 VARCHAR, u3 BOOLEAN, u4 BLOB)
```

**Creation Functions**  For the builtin types, you can use the constants defined in `duckdb.typing`

<table>
<thead>
<tr>
<th>DuckDB Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLNULL</td>
</tr>
<tr>
<td>BOOLEAN</td>
</tr>
<tr>
<td>TINYINT</td>
</tr>
<tr>
<td>UTINYINT</td>
</tr>
<tr>
<td>SMALLINT</td>
</tr>
<tr>
<td>USMALLINT</td>
</tr>
<tr>
<td>INTEGER</td>
</tr>
<tr>
<td>UINTeger</td>
</tr>
<tr>
<td>BIGINT</td>
</tr>
<tr>
<td>UBIGINT</td>
</tr>
<tr>
<td>HUGEINT</td>
</tr>
<tr>
<td>UUID</td>
</tr>
<tr>
<td>FLOAT</td>
</tr>
<tr>
<td>DOUBLE</td>
</tr>
<tr>
<td>DATE</td>
</tr>
<tr>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>TIMESTAMP_MS</td>
</tr>
<tr>
<td>TIMESTAMP_NS</td>
</tr>
<tr>
<td>TIMESTAMP_S</td>
</tr>
<tr>
<td>TIME</td>
</tr>
<tr>
<td>TIME_TZ</td>
</tr>
<tr>
<td>TIMESTAMP_TZ</td>
</tr>
<tr>
<td>VARCHAR</td>
</tr>
<tr>
<td>BLOB</td>
</tr>
</tbody>
</table>
DuckDB Type

BIT
INTERVAL

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.

```python
import duckdb
import pandas as pd

df = pd.DataFrame({'a': [1, 2, 3, 4]})

col = duckdb.ColumnExpression('a')
res = duckdb.df(df).select(col).fetchall()
print(res)
# [(1,), (2,), (3,), (4,)]
```

**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.

```python
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'])
res = duckdb.df(df).select(star).fetchall()
print(res)
# [(1, 42), (2, 21), (3, 13), (4, 14)]
```

**Constant Expression**

This expression contains a single value.
**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 = ...)`. 

```python
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)

res = duckdb.df(df).select(case).fetchall()
print(res)
# [('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.

```python
import duckdb
import pandas as pd
from duckdb import (ConstantExpression,
                     ColumnExpression,
                     FunctionExpression)

df = pd.DataFrame(
    {'a': ['test',
           'pest',
           'text',
           'rest',
           ]})

ends_with = FunctionExpression('ends_with', ColumnExpression('a'), ConstantExpression('est'))
res = duckdb.df(df).select(ends_with).fetchall()
print(res)
# [(True,), (True,), (False,), (True,)]
```

Common Operations

The Expression class also contains many operations that can be applied to any Expression type.

- `.cast(type: DuckDBPyType)`
  Applies a cast to the provided type on the expression.

- `.alias(name: str)`
  Apply an alias to the expression.

- `.isin(*exprs: Expression)`
  Create a IN expression against the provided expressions as the list.

- `.isnotin(*exprs: Expression)`
  Create a NOT IN expression against the provided expressions as the list.

Order Operations  When expressions are provided to DuckDBPyRelation.order() these take effect:

- `.asc()`
  Indicates that this expression should be sorted in ascending order.
DuckDB Documentation

**.desc()**
Indicates that this expression should be sorted in descending order.

**.nulls_first()**
Indicates that the nulls in this expression should preceed 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.

*Note.* 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**

```python
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'),
]
```
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

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.

Running EXPLAIN Renders Newlines in Jupyter and IPython

When DuckDB is run in Jupyter notebooks or in the IPython shell, the output of the EXPLAIN statement contains hard line breaks (\n):

In [1]: import duckdb
   ...: duckdb.sql("EXPLAIN SELECT 42 AS x")

Out[1]:

<table>
<thead>
<tr>
<th>explain_key</th>
<th>explain_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>varchar</td>
<td>varchar</td>
</tr>
<tr>
<td>physical_plan</td>
<td>\n</td>
</tr>
</tbody>
</table>

330
To work around this, print the output of the explain() function:

```r
In [2]: print(duckdb.sql("SELECT 42 AS x").explain())
```

```
+--------+--------+
| PROJECTION     |
| ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ |
|   x        |
| └─────────────┐
|   DUMMY_SCAN |
| └─────────────┘
```

Please also check out the Jupyter guide for tips on using Jupyter with Jupyter SQL.

**Error When Importing the DuckDB Python Package on Windows**

When importing DuckDB on Windows, the Python runtime may return the following error:

```r
import duckdb
```

```python
ImportError: DLL load failed while importing duckdb: The specified module could not be found.
```

The solution is to install the Microsoft Visual C++ Redistributable package.

**R API**

**Installation**

The DuckDB R API can be installed using `install.packages("duckdb")`. Please see the [installation page](#) for details.

**Basic API Usage**

The standard DuckDB R API implements the [DBI interface](http://) for R. If you are not familiar with DBI yet, see [here](http://) 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.

```r
# 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 API with the `dbExecute` and `dbGetQuery` methods. Here is an example:

```r
# 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 (?, ?, ?)"
```

```
# query the database using a prepared statement
res <- dbGetQuery(con, "SELECT item FROM items WHERE value > ?", list(400))
print(res)
```
DuckDB Documentation

# item
# 1 laptop

Note. Do not use prepared statements to insert large amounts of data into DuckDB. See below for better options.

Efficient Transfer

To write a R dataframe into DuckDB, use the standard DBI function `dbWriteTable()`. This creates a table in DuckDB and populates it with the dataframe contents. For example:

```r
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 dataframe as a virtual table, comparable to a SQL VIEW. This *does not actually transfer data* into DuckDB yet. Below is an example:

```r
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
```

Note. DuckDB keeps a reference to the R dataframe after registration. This prevents the dataframe 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()
```

333
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
df <- mtcars
write.csv(df, "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 = 1)") |> 
  filter(month == "3") |> 
  summarise(delay = mean(dep_time, na.rm = TRUE)) |> 
  collect()
```

**GitHub Repository**

[GitHub](https://github.com/duckdb)

**Rust API**

**Installation**

The DuckDB Rust API 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).

```rust
use duckdb::{params, Connection, Result};
let conn = Connection::open_in_memory();
```

You can `conn.close()` the Connection manually, or just leave it out of scope, we had implement the Drop trait which will automatically close the underlining db connection for you.
DuckDB Documentation

**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.

```rust
#[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());
}
```

**Swift API**

DuckDB offers a Swift API. See the announcement post for details.

**Instantiating DuckDB**

DuckDB supports both in-memory and persistent databases. To work with an in-memory database, run:

```swift
let database = try Database(store: .inMemory)
```

To work with a persistent database, run:

```swift
let database = try Database(store: .file(at: "test.db"))
```

Queries can be issued through a database connection.

```swift
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.

```swift
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:

```swift
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(  
```
DuckDB Documentation

```sql
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 queries 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.

...
// 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 repo](https://github.com/DuckDB/DuckDB-Swift) and open up the runnable app project located in `Examples/SwiftUI/ExoplanetExplorer.xcodeproj`.

**GitHub Repository**

[GitHub](https://github.com/DuckDB/DuckDB-Swift)

**Wasm**

**DuckDB Wasm**

DuckDB has been compiled to WebAssembly, so it can run inside any browser on any device.

{% include iframe.html src="https://github.com/DuckDB/DuckDB-Wasm" %}

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](https://duckdb.org/blog/duckdb-wasm-launch).

Another great resource is the [GitHub repository](https://github.com/DuckDB/DuckDB-Wasm).

For details, see the full [DuckDB-Wasm API Documentation](https://duckdb.org/docs/wasm/api).

**Instantiation**

DuckDB-Wasm has multiple ways to be instantiated depending on the use case.
DuckDB Documentation

**Instantiation**

cdn(jsdelivr)

```javascript
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 asynchronous 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**

```javascript
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 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);
```

**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 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);

--

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(JSDELIVR_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);

--

340
**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**

```javascript
const c = await db.connect();

// 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/
```javascript
import { tableFromArrays } from 'apache-arrow';

// EOS signal according to Arrow IPC streaming format
// See https://arrow.apache.org/docs/format/Columnar.html#ipc-streaming-format
const EOS = new Uint8Array([255, 255, 255, 0, 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
2|bar';
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: 342,
    col2: 343,
};
"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 = 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')
`);

https (Wasm-flavored)

// ..., by specifying URLs in the SQL text
await c.query(`
    CREATE TABLE direct AS
    SELECT * FROM 'https://origin/remote.parquet'
`);
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, 10000) AS 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 Em-
scripten'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.

**Note.** 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

<table>
<thead>
<tr>
<th>Extension name</th>
<th>Description</th>
<th>Aliases</th>
</tr>
</thead>
<tbody>
<tr>
<td>autocomplete</td>
<td>Adds support for autocomplete in the shell</td>
<td></td>
</tr>
<tr>
<td>excel</td>
<td>Adds support for Excel-like format strings</td>
<td></td>
</tr>
<tr>
<td>fts</td>
<td>Adds support for Full-Text Search Indexes</td>
<td></td>
</tr>
<tr>
<td>icu</td>
<td>Adds support for time zones and collations using the ICU library</td>
<td></td>
</tr>
<tr>
<td>inet</td>
<td>Adds support for IP-related data types and functions</td>
<td></td>
</tr>
<tr>
<td>json</td>
<td>Adds support for JSON operations</td>
<td></td>
</tr>
<tr>
<td>parquet</td>
<td>Adds support for reading and writing Parquet files</td>
<td></td>
</tr>
<tr>
<td>sqlite GitHub</td>
<td>Adds support for reading SQLite database files</td>
<td>sqlite, sqlite3</td>
</tr>
<tr>
<td>sqlsmith</td>
<td></td>
<td></td>
</tr>
<tr>
<td>substrait GitHub</td>
<td>Adds support for the Substrait integration</td>
<td></td>
</tr>
<tr>
<td>tpcds</td>
<td>Adds TPC-DS data generation and query support</td>
<td></td>
</tr>
<tr>
<td>tpch</td>
<td>Adds TPC-H data generation and query support</td>
<td></td>
</tr>
</tbody>
</table>
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 need 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 interchangable, 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](https://developer.mozilla.org) 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](http://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.

{include iframe.html src="https://shell.duckdb.org"}%

ADBC API

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.

DuckDB's ADBC driver currently supports version 0.5.1 of ADBC.

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 Connection-ReadPartition 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.

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
<th>Arguments</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>DatabaseNew</td>
<td>Allocate a new (but uninitialized) database.</td>
<td>(AdbcDatabase *database, AdbcError *error)</td>
<td>AdbcDatabaseNew(&amp;adbc_database, &amp;adbc_error)</td>
</tr>
<tr>
<td>DatabaseSetOption</td>
<td>Set a char* option.</td>
<td>(AdbcDatabase *database, const char *key, const char *value, AdbcError *error)</td>
<td>AdbcDatabaseSetOption(&amp;adbc_database, &quot;path&quot;, &quot;test.db&quot;, &amp;adbc_error)</td>
</tr>
<tr>
<td>Function Name</td>
<td>Description</td>
<td>Arguments</td>
<td>Example</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>---------------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>DatabaseInit</td>
<td>Finish setting options and initialize the database.</td>
<td>(AdbcDatabase<em>database, AdbcError</em>error)</td>
<td>AdbcDatabaseInit(&amp;adbc_database, &amp;adbc_error)</td>
</tr>
<tr>
<td>DatabaseRelease</td>
<td>Destroy the database.</td>
<td>(AdbcDatabase<em>database, AdbcError</em>error)</td>
<td>AdbcDatabaseRelease(&amp;adbc_database, &amp;adbc_error)</td>
</tr>
</tbody>
</table>

**Connection**  
A set of functions that create and destroy a connection to interact with a database.

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
<th>Arguments</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConnectionNew</td>
<td>Allocate a new (but uninitialized) connection.</td>
<td>(AdbcConnection*, AdbcError*)</td>
<td>AdbcConnectionNew(&amp;adbc_connection, &amp;adbc_error)</td>
</tr>
<tr>
<td>ConnectionSetOption</td>
<td>Options may be set before ConnectionInit.</td>
<td>(AdbcConnection*, const char*, const char*, AdbcError*)</td>
<td>AdbcConnectionSetOption(&amp;adbc_connection, ADBC_CONNECTION_OPTION_AUTOCOMMIT, ADBC_OPTION_VALUE_DISABLED, &amp;adbc_error)</td>
</tr>
<tr>
<td>ConnectionInit</td>
<td>Finish setting options and initialize the connection.</td>
<td>(AdbcConnection*, AdbcDatabase*, AdbcError*)</td>
<td>AdbcConnectionInit(&amp;adbc_connection, &amp;adbc_database, &amp;adbc_error)</td>
</tr>
<tr>
<td>ConnectionRelease</td>
<td>Destroy this connection.</td>
<td>(AdbcConnection*, AdbcError*)</td>
<td>AdbcConnectionRelease(&amp;adbc_connection, &amp;adbc_error)</td>
</tr>
</tbody>
</table>

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*, 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)
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:
**DuckDB Documentation**

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
<th>Arguments</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>StatementNew</strong></td>
<td>Create a new statement for a given connection.</td>
<td>(AdbcConnection*, AdbcStatement*, AdbcError*)</td>
<td>AdbcStatementNew(&amp;adbc_connection, &amp;adbc_statement, &amp;adbc_error)</td>
</tr>
<tr>
<td><strong>StatementRelease</strong></td>
<td>Destroy a statement.</td>
<td>(AdbcStatement*, AdbcError*)</td>
<td>AdbcStatementRelease(&amp;adbc_statement, &amp;adbc_error)</td>
</tr>
<tr>
<td><strong>StatementSetOption</strong></td>
<td>Set a string option on a statement.</td>
<td>(AdbcStatement*, const char*, const char*, AdbcError*)</td>
<td>StatementSetOption(&amp;adbc_statement, ADBC_INGEST_OPTION_TARGET_TABLE, &quot;TABLE_NAME&quot;, &amp;adbc_error)</td>
</tr>
</tbody>
</table>

**Functions related to query execution:**

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
<th>Arguments</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>StatementSetSqlQuery</strong></td>
<td>Set the SQL query to execute. The query can then be executed with StatementExecuteQuery.</td>
<td>(AdbcStatement*, const char*, AdbcError*)</td>
<td>AdbcStatementSetSqlQuery(&amp;adbc_statement, &quot;SELECT * FROM TABLE&quot;, &amp;adbc_error)</td>
</tr>
<tr>
<td><strong>StatementSetSubstraitPlan</strong></td>
<td>Set a substrait plan to execute. The query can then be executed with StatementExecuteQuery.</td>
<td>(AdbcStatement*, const uint8_t*, size_t, AdbcError*)</td>
<td>AdbcStatementSetSubstraitPlan(&amp;adbc_statement, substrait_plan, length, &amp;adbc_error)</td>
</tr>
<tr>
<td><strong>StatementExecuteQuery</strong></td>
<td>Execute a statement and get the results.</td>
<td>(AdbcStatement*, ArrowArrayStream*, int64_t*, AdbcError*)</td>
<td>AdbcStatementExecuteQuery(&amp;adbc_statement, &amp;arrow_stream, &amp;rows_affected, &amp;adbc_error)</td>
</tr>
</tbody>
</table>
### Functions

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
<th>Arguments</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>StatementPrepare</td>
<td>Turn this statement into a prepared statement to be executed multiple times.</td>
<td>(AdbcStatement*, AdbcError*)</td>
<td>AdbcStatementPrepare(&amp;adbc_statement, &amp;adbc_error)</td>
</tr>
</tbody>
</table>

**Example**

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.

```c++
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.

```c
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.

```c
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.

```c
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.

```c
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
```

**Note.** 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.
```python
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.

```python
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)
```

**ODBC**

**ODBC API - Overview**

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 DM 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.

We release the ODBC driver as assets for Linux and Windows. Users can download them from the Latest Release of DuckDB.
Operating System

ODBC API - Linux

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:

Debian SO Flavors

```
sudo apt get install unixodbc
```

Fedora SO Flavors

```
sudo yum install unixodbc
```
# or
```
sudo dnf install unixodbc
```

Step 1: Download ODBC Driver

DuckDB releases the ODBC driver as asset. For Linux, download it from ODBC Linux Asset that contains the following artifacts:

- `libduckdb_odbc.so`: the DuckDB driver compiled to Ubuntu 16.04.
- `unixodbc_setup.sh`: a setup script to aid the configuration on Linux.

Step 2: Extracting ODBC Artifacts

Run `unzip` to extract the files to a permanent directory:

```
mkdir duckdb_odbc
unzip duckdb_odbc-linux-amd64.zip -d duckdb_odbc
```

Step 3: Configuring with unixODBC

The `unixodbc_setup.sh` script aids 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`.

In a terminal window, change to the `duckdb_odbc` permanent directory, and run the following commands with level options `-u` or `-s` either to configure DuckDB ODBC.
**User-Level ODBC Setup (-u)**  The `-u` option based on the user home directory to setup the ODBC init files.

`unixodbc_setup.sh -u`

P.S.: The default configuration consists of a database :memory:.

**System-Level ODBC setup (-s)**  The `-s` changes the system level files that will be visible for all users, because of that it requires root privileges.

`sudo unixodbc_setup.sh -s`

P.S.: The default configuration consists of a database :memory:.

**Show Usage (--help)**  The option `--help` shows the usage of `unixodbc_setup.sh` that provides alternative options for a customer configuration, like `-db` and `-D`.

`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

**Step 4 (Optional): Configure the ODBC Driver**

The ODBC setup on Linux is based on files, the well-known `.odbc.ini` and `.odbcinst.ini`. These files can be placed at the system `/etc` directory or at the user home directory `/home/<user>` (shortcut as `~`). The DM prioritizes the user configuration files and then the system files.

**The `.odbc.ini` File**  The `.odbc.ini` contains the DSNs for the drivers, which can have specific knobs.

An example of `.odbc.ini` with DuckDB would be:

```
[DuckDB]
Driver = DuckDB Driver
Database = :memory:
```
DuckDB Documentation

[DuckDB]: between the brackets is a DSN for the DuckDB.

Driver: it describes the driver's name, and other configurations will be placed at the .odbcinst.ini.

Database: it describes the database name used by DuckDB, and it can also be a file path to a .db in the system.

The .odbcinst.ini File  The .odbcinst.ini 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.

An example of .odbcinst.ini with the DuckDB driver would be:

[ODBC]
Trace = yes
TraceFile = /tmp/odbctrace

[DuckDB Driver]
Driver = /home/<user>/duckdb_odbc/libduckdb_odbc.so

[ODBC]: it is the DM configuration section.

Trace: it 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.

ODBC API - Windows

The Microsoft Windows requires an ODBC Driver Manager to manage communication between applications and the ODBC drivers. The DM on Windows is provided in a DLL file odbc32.dll, and other files and tools. For detailed information checkout out the Common ODBC Component Files.

Step 1: Download ODBC Driver

DuckDB releases the ODBC driver as asset. For Windows, download it from Windows Asset that 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.
**Step 2: Extracting ODBC Artifacts**

Unzip the file to a permanent directory (e.g., duckdb_odbc).

An example with PowerShell and `unzip` command would be:

```bash
mkdir duckdb_odbc
unzip duckdb_odbc-linux-amd64.zip -d duckdb_odbc
```

**Step 3: ODBC Windows Installer**

The `odbc_install.exe` aids the configuration of the DuckDB ODBC Driver on Windows. It depends on the `OdbcP32.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 is required, in case of a non-administrator a User Account Control shall display:

**Step 4: Configure the ODBC Driver**

The `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**

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.

**ODBC API - MacOS**

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 MacOS (and Linux). Users can install it from the command line:

**Brew**

```bash
brew install unixodbc
```

**Step 1: Download ODBC Driver**

DuckDB releases the ODBC driver as asset. For MacOS, download it from ODBC Linux Asset that contains the following artifacts:

- `libduckdb_odbc.dylib`: the DuckDB driver compiled to MacOS (with Intel and Apple M1 support).

**Step 2: Extracting ODBC Artifacts**

Run unzip to extract the files to a permanent directory:

```bash
mkdir duckdb_odbc
unzip duckdb_odbc-osx-universal.zip -d duckdb_odbc
```
Step 3: Configure the ODBC Driver

The \textbf{odbc.ini} or \textbf{.odbc.ini} File  The \texttt{.odbc.ini} contains the DSNs for the drivers, which can have specific knobs.

An example of \texttt{.odbc.ini} with DuckDB would be:

\begin{verbatim}
[DuckDB]
Driver = DuckDB Driver
Database = :memory:
\end{verbatim}

\textbf{[DuckDB]}: between the brackets is a DSN for the DuckDB.

\textbf{Driver}: it describes the driver's name, and other configurations will be placed at the \texttt{.odbcinst.ini}.

\textbf{Database}: it describes the database name used by DuckDB, and it can also be a file path to a \texttt{.db} in the system.

The \textbf{.odbcinst.ini} File  The \texttt{.odbcinst.ini} 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.

An example of \texttt{.odbcinst.ini} with the DuckDB driver would be:

\begin{verbatim}
[ODBC]
Trace = yes
TraceFile = /tmp/odbctrace

[DuckDB Driver]
Driver = /User/<user>/duckdb_odbc/libduckdb_odbc.dylib

[ODBC]: it is the DM configuration section.

Trace: it enables the ODBC trace file using the option \texttt{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.
\end{verbatim}

Step 4 (Optional): Test the ODBC Driver

After the configuration, for validate the installation, it is possible to use an 	exttt{odbc} client. \texttt{unixODBC} use a command line tool called \texttt{isql}.

Use the DSN defined in \texttt{odbc.ini} as a parameter of \texttt{isql}.

\begin{verbatim}
isql DuckDB
+---------------------------------------+
| Connected! |                           |
\end{verbatim}
<table>
<thead>
<tr>
<th>sql-statement</th>
<th>help [tablename]</th>
<th>echo [string]</th>
<th>quit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
+-----------------+-----------------+--------------+------+

SQL> SELECT 42;
+------------+
| 42         |
+------------+
| 42         |
+------------+

SQLRowCount returns -1
1 rows fetched
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.

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:

```sql
CREATE TABLE weather (  
city VARCHAR,  
temp_lo INTEGER, -- minimum temperature on a day  
temp_hi INTEGER, -- maximum temperature on a day  
prcp REAL,  
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 key words and identifiers.
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). REAL 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, REAL, 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, 0.8, '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);
```
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.

You could also have used COPY to load large amounts of data from CSV files. This is usually faster because the COPY command is optimized for this application while allowing less flexibility than INSERT. An example with weather.csv would be:

```sql
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:

```sql
SELECT *
FROM weather;
```

Here * is a shorthand for “all columns”. So the same result would be had with:

```sql
SELECT city, temp_lo, temp_hi, prcp, date
FROM weather;
```

The output should be:

<table>
<thead>
<tr>
<th>city</th>
<th>temp_lo</th>
<th>temp_hi</th>
<th>prcp</th>
<th>date</th>
</tr>
</thead>
<tbody>
<tr>
<td>San Francisco</td>
<td>46</td>
<td>50</td>
<td>0.25</td>
<td>1994-11-27</td>
</tr>
<tr>
<td>San Francisco</td>
<td>43</td>
<td>57</td>
<td>0.0</td>
<td>1994-11-29</td>
</tr>
<tr>
<td>Hayward</td>
<td>37</td>
<td>54</td>
<td></td>
<td>1994-11-29</td>
</tr>
</tbody>
</table>

You can write expressions, not just simple column references, in the select list. For example, you can do:

```sql
SELECT city, (temp_hi+temp_lo)/2 AS temp_avg, date
FROM weather;
```

This should give:

<table>
<thead>
<tr>
<th>city</th>
<th>temp_avg</th>
<th>date</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>double</td>
<td>date</td>
</tr>
</tbody>
</table>
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:

```sql
SELECT *
FROM weather
WHERE city = 'San Francisco' AND prcp > 0.0;
```

Result:

<table>
<thead>
<tr>
<th>city</th>
<th>temp_lo</th>
<th>temp_hi</th>
<th>prcp</th>
<th>date</th>
</tr>
</thead>
<tbody>
<tr>
<td>San Francisco</td>
<td>46</td>
<td>50</td>
<td>0.25</td>
<td>1994-11-27</td>
</tr>
</tbody>
</table>

You can request that the results of a query be returned in sorted order:

```sql
SELECT *
FROM weather
ORDER BY city;
```

Result:

<table>
<thead>
<tr>
<th>city</th>
<th>temp_lo</th>
<th>temp_hi</th>
<th>prcp</th>
<th>date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hayward</td>
<td>37</td>
<td>54</td>
<td></td>
<td>1994-11-29</td>
</tr>
<tr>
<td>San Francisco</td>
<td>46</td>
<td>50</td>
<td>0.25</td>
<td>1994-11-27</td>
</tr>
<tr>
<td>San Francisco</td>
<td>43</td>
<td>57</td>
<td>0.0</td>
<td>1994-11-29</td>
</tr>
</tbody>
</table>

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:

```sql
SELECT *
FROM weather
ORDER BY city, temp_lo;
```

You can request that duplicate rows be removed from the result of a query:

```sql
SELECT DISTINCT city
FROM weather;
```
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;
```

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 *:

```sql
SELECT city, temp_lo, temp_hi, prcp, date, lon, lat
FROM weather, cities
WHERE city = name;
```

<table>
<thead>
<tr>
<th>city</th>
<th>temp_lo</th>
<th>temp_hi</th>
<th>prcp</th>
<th>date</th>
<th>lon</th>
<th>lat</th>
</tr>
</thead>
<tbody>
<tr>
<td>varchar</td>
<td>int32</td>
<td>int32</td>
<td>float</td>
<td>date</td>
<td>decimal(18,3)</td>
<td>decimal(18,3)</td>
</tr>
<tr>
<td>San Francisco</td>
<td>46</td>
<td>50</td>
<td>0.25</td>
<td>1994-11-27</td>
<td>53.000</td>
<td>-194.000</td>
</tr>
<tr>
<td>San Francisco</td>
<td>43</td>
<td>57</td>
<td>0.0</td>
<td>1994-11-29</td>
<td>53.000</td>
<td>-194.000</td>
</tr>
</tbody>
</table>

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:

```sql
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:

```sql
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:

```sql
SELECT *
FROM weather
LEFT OUTER JOIN cities ON weather.city = cities.name;
```

<table>
<thead>
<tr>
<th>city</th>
<th>temp_lo</th>
<th>temp_hi</th>
<th>prcp</th>
<th>date</th>
<th>name</th>
<th>lat</th>
</tr>
</thead>
<tbody>
<tr>
<td>lon</td>
<td>int32</td>
<td>int32</td>
<td>float</td>
<td>date</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

370
DuckDB Documentation

<table>
<thead>
<tr>
<th>varchar</th>
<th>int32</th>
<th>int32</th>
<th>float</th>
<th>date</th>
<th>varchar</th>
</tr>
</thead>
<tbody>
<tr>
<td>decimal(18,3)</td>
<td>decimal(18,3)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>San Francisco</td>
<td>46</td>
<td>50</td>
<td>0.25</td>
<td>1994-11-27</td>
<td>San Francisco</td>
</tr>
<tr>
<td></td>
<td>-194.000</td>
<td>53.000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>San Francisco</td>
<td>43</td>
<td>57</td>
<td>0.0</td>
<td>1994-11-29</td>
<td>San Francisco</td>
</tr>
<tr>
<td></td>
<td>-194.000</td>
<td>53.000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hayward</td>
<td>37</td>
<td>54</td>
<td></td>
<td>1994-11-29</td>
<td></td>
</tr>
</tbody>
</table>

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:

```sql
SELECT max(temp_lo)
FROM weather;
```

```
max(temp_lo)
<table>
<thead>
<tr>
<th>int32</th>
</tr>
</thead>
<tbody>
<tr>
<td>46</td>
</tr>
</tbody>
</table>
```

If we wanted to know what city (or cities) that reading occurred in, we might try:

```sql
SELECT city
FROM weather
WHERE temp_lo = max(temp_lo);  -- WRONG
```

but this will not work since the aggregate max cannot be used in the WHERE clause. (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:

```sql
SELECT city
FROM weather
WHERE temp_lo = (SELECT max(temp_lo) FROM weather);
```
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) │
│ varchar │ int32 │
├───────────────┼──────────────┤
│ 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) │
│ varchar │ int32 │
├─────────┼──────────────┤
│ 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%'
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;
```

<table>
<thead>
<tr>
<th>city</th>
<th>temp_lo</th>
<th>temp_hi</th>
<th>prcp</th>
<th>date</th>
</tr>
</thead>
<tbody>
<tr>
<td>San Francisco</td>
<td>46</td>
<td>50</td>
<td>0.25</td>
<td>1994-11-27</td>
</tr>
<tr>
<td>San Francisco</td>
<td>41</td>
<td>55</td>
<td>0.0</td>
<td>1994-11-29</td>
</tr>
<tr>
<td>Hayward</td>
<td>35</td>
<td>52</td>
<td></td>
<td>1994-11-29</td>
</tr>
</tbody>
</table>

**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;
```
One should be wary of statements of the form

```
DELETE FROM tablename;
```

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

**ALTER TABLE Statement**

The `ALTER TABLE` statement changes the schema of an existing table in the catalog.

#### Examples

```
-- 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 t ALTER COLUMN x SET NOT NULL;
-- drop the not null constraint
ALTER TABLE t ALTER COLUMN x 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 j;

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

-- rename a column of a table
ALTER TABLE integers RENAME i TO j;
ALTER TABLE integers RENAME COLUMN j TO k;

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

-- 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;
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**

```sql
-- drop the column "k" from the table integers
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.

**ALTER TYPE**

```sql
-- 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);
```

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.

**SET/DROP DEFAULT**

```sql
-- 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.

**Note.** 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/DROP CONSTRAINT**

**Note.** The **ADD CONSTRAINT** and **DROP CONSTRAINT** clauses are not yet supported in DuckDB.
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/DETACH Statement

The ATTACH statement adds a new database file to the catalog that can be read from and written to.

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 a SQLite database for reading and writing (see sqlite extension for more information)
ATTACH 'sqlite_file.db' AS sqlite (TYPE SQLITE);
-- attach the database "file.db" if inferred database alias "file_db" 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;
-- 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;
**Syntax**

ATTACH allows DuckDB to operate on multiple database files, and allows for transfer of data between different database files.

**Detach**

The DETACH statement allows previously attached database files to be closed and detached, releasing any locks held on the database file. 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.

**Note.** 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).

**Name Qualification**

The fully qualified name of catalog objects contains the catalog, the schema and the name of the object. For example:

```sql
-- 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.

**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.

```sql
-- 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.

```sql
-- 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);
-- Error: 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**

```sql
-- Invoke the 'duckdb_functions' table function.
CALL duckdb_functions();

-- Invoke the 'pragma_table_info' table function.
CALL pragma_table_info('pg_am');
```

**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**

```sql
-- 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.

**Vacuuming Deletes**  As part of performing a checkpoint (automatic or otherwise), vacuuming deleted rows is triggered. Note that this does not remove all deletes, 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.

**Note.** The VACUUM statement does not trigger vacuuming deletes.

**COPY Statement**

**Examples**

```sql
-- 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);

-- write a table to a CSV file
COPY lineitem TO 'lineitem.csv' (FORMAT CSV, DELIMITER '|', HEADER);

-- 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);
```

**COPY Statements**

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.

```sql
-- 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**

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.

```sql
-- 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 '|', HEADER);
```
DuckDB Documentation

-- 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' WITH (HEADER 1, 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);

COPY FROM DATABASE ... TO

Note. This statement is currently only available in nightly builds (DuckDB 0.9.3-dev) and will be released in the upcoming v0.10.0 version.

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;

<table>
<thead>
<tr>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>87</td>
</tr>
</tbody>
</table>

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

**COPY 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.
The below options are applicable to all formats written with COPY.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Type</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>allow_overwrite</td>
<td>Whether or not to allow overwriting a directory if one already exists. Only has an effect when used with partition_by.</td>
<td>BOOL</td>
<td>false</td>
</tr>
<tr>
<td>format</td>
<td>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. Available options are CSV, PARQUET and JSON.</td>
<td>VARCHAR</td>
<td>auto</td>
</tr>
<tr>
<td>partition_by</td>
<td>The columns to partition by using a hive partitioning scheme, seethe partitioned writes section.</td>
<td>VARCHAR[]</td>
<td>(empty)</td>
</tr>
<tr>
<td>per_thread_output</td>
<td>Generate one file per thread, rather than one file in total. This allows for faster parallel writing.</td>
<td>BOOL</td>
<td>false</td>
</tr>
<tr>
<td>use_tmp_file</td>
<td>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.</td>
<td>BOOL</td>
<td>auto</td>
</tr>
</tbody>
</table>

**CSV Options**  The below options are applicable when writing CSV files.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Type</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>compression</td>
<td>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 will use none). Options are none, gzip, zstd.</td>
<td>VARCHAR</td>
<td>auto</td>
</tr>
<tr>
<td>force_quote</td>
<td>The list of columns to always add quotes to, even if not required.</td>
<td>VARCHAR[]</td>
<td>[]</td>
</tr>
<tr>
<td>dateformat</td>
<td>Specifies the date format to use when writing dates. See Date Format</td>
<td>VARCHAR</td>
<td>(empty)</td>
</tr>
<tr>
<td>delim or sep</td>
<td>The character that is written to separate columns within each row.</td>
<td>VARCHAR</td>
<td>,</td>
</tr>
</tbody>
</table>
DuckDB Documentation

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Type</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>escape</td>
<td>The character that should appear before a character that matches the quote value.</td>
<td>VARCHAR</td>
<td>&quot;</td>
</tr>
<tr>
<td>header</td>
<td>Whether or not to write a header for the CSV file.</td>
<td>BOOL</td>
<td>true</td>
</tr>
<tr>
<td>nullstr</td>
<td>The string that is written to represent a NULL value.</td>
<td>VARCHAR</td>
<td>(empty)</td>
</tr>
<tr>
<td>quote</td>
<td>The quoting character to be used when a data value is quoted.</td>
<td>VARCHAR</td>
<td>&quot;</td>
</tr>
<tr>
<td>timestampformat</td>
<td>Specifies the date format to use when writing timestamps. See Date Format</td>
<td>VARCHAR</td>
<td>(empty)</td>
</tr>
</tbody>
</table>

Parquet Options  The below options are applicable when writing Parquet files.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Type</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>compression</td>
<td>The compression format to use (uncompressed, snappy, gzip or zstd).</td>
<td>VARCHAR</td>
<td>snappy</td>
</tr>
<tr>
<td>row_group_size</td>
<td>The target size, i.e., number of rows, of each row group.</td>
<td>BIGINT</td>
<td>122880</td>
</tr>
<tr>
<td>row_group_size_bytes</td>
<td>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.</td>
<td>BIGINT</td>
<td>row_group_size * 1024</td>
</tr>
<tr>
<td>field_ids</td>
<td>The field_id for each column. Pass auto to attempt to infer automatically.</td>
<td>STRUCT</td>
<td>(empty)</td>
</tr>
</tbody>
</table>

Some examples of FIELD_IDS are:

```sql
-- 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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Type</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>compression</td>
<td>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 will use none). Options are none, gzip, zstd.</td>
<td>VARCHAR</td>
<td>auto</td>
</tr>
<tr>
<td>dateformat</td>
<td>Specifies the date format to use when writing dates. See Date Format</td>
<td>VARCHAR</td>
<td>(empty)</td>
</tr>
<tr>
<td>timestampformat</td>
<td>Specifies the date format to use when writing timestamps. See Date Format</td>
<td>VARCHAR</td>
<td>(empty)</td>
</tr>
<tr>
<td>array</td>
<td>Whether to write a JSON array. If true, a JSON array of records is written, if false, newline-delimited JSON is written</td>
<td>BOOL</td>
<td>false</td>
</tr>
</tbody>
</table>

**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 for a case expression
CREATE MACRO elseif(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);
-- create a macro with a common table expression
-- (parameter names get priority over column names: disambiguate using the table name)
CREATE MACRO plus_one(a) AS (WITH cte AS (SELECT 1 AS a) SELECT cte.a + a FROM cte);
-- macros are schema-dependent, and have an alias: FUNCTION
CREATE FUNCTION main.myavg(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: SELECT * FROM get_users([1, 5])
CREATE MACRO get_users(i) AS TABLE SELECT * FROM users WHERE uid IN (SELECT unnest(i));

**Syntax**

Macros allow you to create shortcuts for combinations of expressions.

-- failure! cannot find column "b"
CREATE MACRO add(a) AS a + b;
-- this works
CREATE MACRO add(a, b) AS a + b;
-- error! cannot bind +(VARCHAR, INTEGER)
Macros can have default parameters. Unlike some languages, default parameters must be named when the macro is invoked.

```sql
-- 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);
-- error! add_default only has one positional parameter
SELECT add_default(40, 2);
-- success! default parameters are used by assigning them like so
SELECT add_default(40, b := 2);
-- error! default parameters must come after positional parameters
SELECT add_default(b = 2, 40);
-- the order of default parameters does not matter
CREATE MACRO triple_add(a, b := 5, c := 10) AS a + b + c;
-- success!
SELECT triple_add(40, c := 1, b := 1);
-- 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:

```sql
-- the 'add' macro we defined above is used in a query
SELECT add(40, 2);
-- internally, add is replaced with its definition of a + b
SELECT a + b;
-- then, the parameters are replaced by the supplied arguments
SELECT 40 + 2;
-- 42
```

### CREATE SCHEMA Statement

The `CREATE SCHEMA` statement creates a schema in the catalog. The default schema is `main`.

#### Examples

```sql
-- create a schema
CREATE SCHEMA s1;
-- create a schema if it does not exist yet
CREATE SCHEMA IF NOT EXISTS 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 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 a Serialization Error, e.g.:
-- 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 INT DEFAULT nextval('id_sequence'), s VARCHAR);
INSERT INTO tbl (s) VALUES ('hello'), ('world');
SELECT * FROM tbl;
The script results in the following table:

<table>
<thead>
<tr>
<th>id</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>int32</td>
<td>varchar</td>
</tr>
<tr>
<td>1</td>
<td>hello</td>
</tr>
<tr>
<td>2</td>
<td>world</td>
</tr>
</tbody>
</table>

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 INT DEFAULT nextval('id_sequence');
SELECT * FROM tbl;
```

This script results in the same table as the previous example.

**Selecting the Next Value** Select the next number from a sequence:

```
SELECT nextval('serial') AS nextval;
```

<table>
<thead>
<tr>
<th>nextval</th>
<th>int64</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

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.

```
SELECT currval('serial') AS currval;
```

<table>
<thead>
<tr>
<th>currval</th>
<th>int64</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
**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**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEMPORARY or TEMP</td>
<td>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.</td>
</tr>
<tr>
<td>name</td>
<td>The name (optionally schema-qualified) of the sequence to be created.</td>
</tr>
<tr>
<td>increment</td>
<td>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.</td>
</tr>
<tr>
<td>minvalue</td>
<td>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.</td>
</tr>
<tr>
<td>maxvalue</td>
<td>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.</td>
</tr>
<tr>
<td>start</td>
<td>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.</td>
</tr>
<tr>
<td>CYCLE or NO CYCLE</td>
<td>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.</td>
</tr>
</tbody>
</table>

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.
Sequences are based on BIGINT arithmetic, so the range cannot exceed the range of an eight-byte integer (-9223372036854775808 to 9223372036854775807).

**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 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;
```

**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);

**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);
-- Error: Constraint Error: CHECK constraint failed: t1
INSERT INTO t1 VALUES (3, 101);
-- Error: 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);
-- Error: 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);
-- Error: 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);
-- Error: 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');
-- Error: 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)
);
Note. Foreign keys with cascading deletes (FOREIGN KEY ... REFERENCES ... ON DELETE CASCADE) are not supported.

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.

```sql
-- 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

```sql
-- 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:

```sql
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

CREATE TYPE defines a new data type available to this duckdb instance. These new types can then be inspected in the duckdb_types table.

Extending these custom types to support custom operators (such as the PostgreSQL && operator) would require C++ development. To do this, create an extension.

DELETE Statement

The DELETE statement removes rows from the table identified by the table-name.

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;
```
Syntax

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.

The `USING` clause allows deleting based on the content of other tables or subqueries.

Limitations on Reclaiming Memory and Disk Space

Running `DELETE` does not mean space is reclaimed. In general, rows are only marked as deleted. DuckDB’s support for `VACUUM` is limited to vacuuming entire row groups.

DROP Statement

The `DROP` statement removes a catalog entry added previously with the `CREATE` command.

Examples

```sql
-- 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 type
DROP TYPE type_name;
```

Syntax

The optional `IF EXISTS` clause suppresses the error that would normally result if the table does not exist.

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.

```sql
CREATE SCHEMA myschema;
CREATE TABLE myschema.t1(i INTEGER);
-- ERROR: Cannot drop myschema because the table myschema.t1 depends on it.
DROP SCHEMA myschema;
-- Cascade drops both myschema and myschema.t1
DROP SCHEMA myschema CASCADE;
```
Limitations on Reclaiming Disk Space

Running \texttt{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 \texttt{PRAGMA} and check the number of \texttt{free_blocks} in the output:

\begin{verbatim}
PRAGMA database_size;
\end{verbatim}

**EXPORT/IMPORT DATABASE Statements**

The \texttt{EXPORT DATABASE} command allows you to export the contents of the database to a specific directory. The \texttt{IMPORT DATABASE} command allows you to then read the contents again.

**Examples**

\begin{verbatim}
-- export the database to the target directory 'db_name' as CSV files
EXPORT DATABASE 'db_name';
-- export to directory 'db_name', using the given options for the CSV serialization
EXPORT DATABASE 'db_name' (FORMAT CSV, DELIMITER '|');
-- export to directory 'db_name', tables serialized as Parquet
EXPORT DATABASE 'db_name' (FORMAT PARQUET);
-- export to directory 'db_name', tables serialized as Parquet, compressed with ZSTD, with a row_group_size of 100000
EXPORT DATABASE 'db_name' (FORMAT PARQUET, COMPRESSION ZSTD, ROW_GROUP_SIZE 100000);
-- reload the database again
IMPORT DATABASE 'db_name';
-- alternatively, use a PRAGMA
PRAGMA import_database('db_name');
\end{verbatim}

For details regarding the writing of Parquet files, see the Parquet Files page in the Data Import section, and the COPY Statement page.

**Syntax**

The \texttt{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:

\begin{verbatim}
target_directory/schema.sql
target_directory/load.sql
target_directory/t_1.csv
...
target_directory/t_n.csv
\end{verbatim}
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.

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.

**INSERT Statement**

The INSERT statement inserts new data into a table.

**Examples**

```sql
-- 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);

This will insert 5 into b and 42 into a.

**INSERT INTO ... BY NAME**  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, even if the order of the columns in the table differs from the order of the values in the SELECT statement. For example:

CREATE TABLE tbl (a INTEGER, b INTEGER);
INSERT INTO tbl BY NAME (SELECT 42 AS b);

This will insert 42 into b and insert NULL (or its default value) into a.

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 INT PRIMARY KEY, j INT);
INSERT INTO tbl VALUES (1, 42);
INSERT INTO tbl VALUES (1, 84);

This raises as an error and leaves the table with a single row <i: 1, j: 42>.

**Error: Constraint Error: Duplicate key "i: 1" violates primary key constraint.**

There are two supported actions: **DO NOTHING** and **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 INT PRIMARY KEY, j INT);
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>.
**Shorthand** 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** 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.

An example using `DO UPDATE` is the following:

```
CREATE TABLE tbl (i INT PRIMARY KEY, j INT);
INSERT INTO tbl VALUES (1, 42);
INSERT INTO tbl VALUES (1, 84) ON CONFLICT DO UPDATE SET j = EXCLUDED.j;
-- rearranging columns and BY NAME also work:
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;
```

These statements finish successfully and leaves the table with a single row `<i: 1, j: 84>`.

**Shorthand** 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 INT PRIMARY KEY, j INT);
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;
```

**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.

```sql
CREATE TABLE tbl (i INT PRIMARY KEY, j INT UNIQUE, k INT);
INSERT INTO tbl VALUES (1, 20, 300);
INSERT INTO tbl VALUES (1, 40, 700) ON CONFLICT (i) DO UPDATE SET k = 2 * EXCLUDED.k;
-- tbl will contain <1, 20, 1400>
INSERT INTO tbl VALUES (1, 20, 900) ON CONFLICT (j) DO UPDATE SET k = 5 * EXCLUDED.k;
-- tbl will contain <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.

```sql
INSERT INTO tbl VALUES (1, 40, 700) ON CONFLICT (i) DO UPDATE SET k = 2 * EXCLUDED.k WHERE k < 100;
```

### Multiple Tuples Conflicting on the Same Key

Having multiple tuples conflicting on the same key is not supported. For example:

```sql
CREATE TABLE tbl (i INT PRIMARY KEY, j INT);
INSERT INTO tbl VALUES (1, 42);
INSERT INTO tbl VALUES (1, 84), (1, 168) ON CONFLICT DO NOTHING;
```

Running this returns the following message.

```
Error: Invalid Input Error: ON CONFLICT DO UPDATE can not update the same row twice in the same command.
```

Ensure that no rows proposed for insertion within the same command have duplicate constrained values

### 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:

```sql
CREATE TABLE t1 (i INT);
INSERT INTO t1
SELECT 42
RETURNING *;
```
A more complex example that includes an expression in the RETURNING clause:

```sql
CREATE TABLE t2 (i INT, j INT);
INSERT INTO t2
    SELECT 2 AS i, 3 AS j
    RETURNING *, i * j AS i_times_j;

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>i_times_j</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>
```

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.

```sql
CREATE TABLE t3 (i INT PRIMARY KEY, j INT);
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 *;

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>42</td>
</tr>
<tr>
<td>2</td>
<td>43</td>
</tr>
</tbody>
</table>
```

**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.
Note. 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 INT, Population INT);
INSERT INTO Cities VALUES ('NL', 'Amsterdam', 2000, 1005);
INSERT INTO Cities VALUES ('NL', 'Amsterdam', 2010, 1065);
INSERT INTO Cities VALUES ('NL', 'Amsterdam', 2020, 1158);
INSERT INTO Cities VALUES ('US', 'Seattle', 2000, 564);
INSERT INTO Cities VALUES ('US', 'Seattle', 2010, 608);
INSERT INTO Cities VALUES ('US', 'Seattle', 2020, 738);
INSERT INTO Cities VALUES ('US', 'New York City', 2000, 8015);
INSERT INTO Cities VALUES ('US', 'New York City', 2010, 8175);
INSERT INTO Cities VALUES ('US', 'New York City', 2020, 8772);
FROM Cities;
```
**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);
```

<table>
<thead>
<tr>
<th>Country</th>
<th>Name</th>
<th>2000</th>
<th>2010</th>
<th>2020</th>
</tr>
</thead>
<tbody>
<tr>
<td>NL</td>
<td>Amsterdam</td>
<td>1005</td>
<td>1065</td>
<td>1158</td>
</tr>
<tr>
<td>US</td>
<td>Seattle</td>
<td>564</td>
<td>608</td>
<td>738</td>
</tr>
<tr>
<td>US</td>
<td>New York City</td>
<td>8015</td>
<td>8175</td>
<td>8772</td>
</tr>
</tbody>
</table>

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);
```

**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;
```

<table>
<thead>
<tr>
<th>Country</th>
<th>2000</th>
<th>2010</th>
<th>2020</th>
</tr>
</thead>
<tbody>
<tr>
<td>NL</td>
<td>1005</td>
<td>1065</td>
<td>1158</td>
</tr>
<tr>
<td>US</td>
<td>8579</td>
<td>8783</td>
<td>9510</td>
</tr>
</tbody>
</table>

**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;
```
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.

```sql
PIVOT Cities ON Country, Name USING sum(Population);
```

<table>
<thead>
<tr>
<th>Year</th>
<th>NL_Amsterdam</th>
<th>NL_New_York_City</th>
<th>NL_Seattle</th>
<th>US_Amsterdam</th>
<th>US_New_York_City</th>
<th>US_Seattle</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>1005</td>
<td>NULL</td>
<td>NULL</td>
<td>8015</td>
<td>564</td>
<td></td>
</tr>
<tr>
<td>2010</td>
<td>1065</td>
<td>NULL</td>
<td>NULL</td>
<td>8175</td>
<td>608</td>
<td></td>
</tr>
<tr>
<td>2020</td>
<td>1158</td>
<td>NULL</td>
<td>NULL</td>
<td>8772</td>
<td>738</td>
<td></td>
</tr>
</tbody>
</table>

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).

```sql
PIVOT Cities ON Country || '_' || Name USING sum(Population);
```

<table>
<thead>
<tr>
<th>Year</th>
<th>NL_Amsterdam</th>
<th>US_New_York_City</th>
<th>US_Seattle</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>1005</td>
<td>8015</td>
<td>564</td>
</tr>
<tr>
<td>2010</td>
<td>1065</td>
<td>8175</td>
<td>608</td>
</tr>
<tr>
<td>2020</td>
<td>1158</td>
<td>8772</td>
<td>738</td>
</tr>
</tbody>
</table>
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.

```sql
PIVOT Cities ON Year USING sum(Population) AS total, max(Population) AS max GROUP BY Country;
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>NL</td>
<td>1005</td>
<td>1005</td>
<td>1065</td>
<td>1065</td>
<td>1158</td>
<td>1158</td>
</tr>
<tr>
<td>US</td>
<td>8579</td>
<td>8015</td>
<td>8783</td>
<td>8175</td>
<td>9510</td>
<td>8772</td>
</tr>
</tbody>
</table>

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.

```sql
PIVOT Cities ON Year USING sum(Population) GROUP BY Country, Name;
```

<table>
<thead>
<tr>
<th>Country</th>
<th>Name</th>
<th>2000</th>
<th>2010</th>
<th>2020</th>
</tr>
</thead>
<tbody>
<tr>
<td>NL</td>
<td>Amsterdam</td>
<td>1005</td>
<td>1065</td>
<td>1158</td>
</tr>
<tr>
<td>US</td>
<td>Seattle</td>
<td>564</td>
<td>608</td>
<td>738</td>
</tr>
<tr>
<td>US</td>
<td>New York City</td>
<td>8015</td>
<td>8175</td>
<td>8772</td>
</tr>
</tbody>
</table>

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.

```sql
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.

```sql
SELECT *  
FROM (  
    PIVOT Cities ON Year USING sum(Population) GROUP BY Country
  ) pivot_alias;
```
**Multiple Pivots**  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.

```sql
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);
```

<table>
<thead>
<tr>
<th>Country</th>
<th>2000</th>
<th>2010</th>
<th>2020</th>
<th>Amsterdam</th>
<th>New York City</th>
<th>Seattle</th>
</tr>
</thead>
<tbody>
<tr>
<td>NL</td>
<td>1005</td>
<td>1065</td>
<td>1158</td>
<td>3228</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>US</td>
<td>8579</td>
<td>8783</td>
<td>9510</td>
<td>NULL</td>
<td>24962</td>
<td>1910</td>
</tr>
</tbody>
</table>

**Internals**

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 ENUMs, the query is re-written again into a set of aggregations into lists.

For example:

```sql
PIVOT Cities ON Year USING sum(Population);
```

is initially translated into:

```sql
CREATE TEMPORARY TYPE __pivot_enum_0_0 AS ENUM (SELECT DISTINCT Year::VARCHAR
FROM Cities
ORDER BY Year);
```

```sql
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:

<table>
<thead>
<tr>
<th>Country</th>
<th>Name</th>
<th>list(&quot;YEAR&quot;)</th>
<th>list(population_sum)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NL</td>
<td>Amsterdam</td>
<td>[2000, 2010, 2020]</td>
<td>[1005, 1065, 1158]</td>
</tr>
</tbody>
</table>

The PhysicalPivot operator converts those lists into column names and values to return this result:

<table>
<thead>
<tr>
<th>Country</th>
<th>Name</th>
<th>2000</th>
<th>2010</th>
<th>2020</th>
</tr>
</thead>
<tbody>
<tr>
<td>NL</td>
<td>Amsterdam</td>
<td>1005</td>
<td>1065</td>
<td>1158</td>
</tr>
<tr>
<td>US</td>
<td>Seattle</td>
<td>564</td>
<td>608</td>
<td>738</td>
</tr>
<tr>
<td>US</td>
<td>New York City</td>
<td>8015</td>
<td>8175</td>
<td>8772</td>
</tr>
</tbody>
</table>

**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:

```sql
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:

```sql
FROM Cities
PIVOT (sum(Population) FOR Year IN (2000, 2010, 2020)
       GROUP BY Country)
```

<table>
<thead>
<tr>
<th>Country</th>
<th>2000</th>
<th>2010</th>
<th>2020</th>
</tr>
</thead>
<tbody>
<tr>
<td>NL</td>
<td>1005</td>
<td>1065</td>
<td>1158</td>
</tr>
<tr>
<td>US</td>
<td>8579</td>
<td>8783</td>
<td>9510</td>
</tr>
</tbody>
</table>

This example is somewhat contrived, but serves as an example of using multiple value expressions and multiple columns in the FOR clause.

```sql
FROM Cities
PIVOT (sum(Population) AS total,
       count(Population) AS count
       FOR Year IN (2000, 2010)
       Country in ('NL', 'US'))
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>NL_total</td>
<td>US_total</td>
<td>NL_count</td>
<td>US_count</td>
<td>NL_total</td>
<td>US_total</td>
</tr>
<tr>
<td>Amsterdam</td>
<td>1005</td>
<td>1</td>
<td>NULL</td>
<td>1065</td>
<td>1</td>
<td>NULL</td>
</tr>
<tr>
<td>Seattle</td>
<td>NULL</td>
<td>0</td>
<td>564</td>
<td>1</td>
<td>NULL</td>
<td>0</td>
</tr>
<tr>
<td>New York City</td>
<td>NULL</td>
<td>0</td>
<td>8015</td>
<td>1</td>
<td>NULL</td>
<td>0</td>
</tr>
</tbody>
</table>

**SQL Standard PIVOT Full Syntax Diagram**  Below is the full syntax diagram of the SQL Standard version of the PIVOT statement.
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;
```
The SELECT statement retrieves rows from the database. The canonical order of a select statement is as follows, with less common clauses being indented:

```sql
SELECT select_list
FROM tables
    USING SAMPLE sample_expr
WHERE condition
GROUP BY groups
HAVING group_filter
    WINDOW window_expr
    QUALIFY qualify_filter
ORDER BY order_expr
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.

---

| Note. | 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 sample 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 and LIMIT Clauses

`ORDER BY` and `LIMIT` are output modifiers. Logically they are applied at the very end of the query. The `LIMIT` clause restricts the amount of rows fetched, and the `ORDER BY` clause sorts the rows on the sorting criteria in either ascending or descending order.

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.

```sql
CREATE TABLE t (id INT, content STRING);
INSERT INTO t VALUES (42, 'hello'), (43, 'world');
SELECT rowid, id, content FROM t;
```

<table>
<thead>
<tr>
<th>rowid</th>
<th>id</th>
<th>content</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>42</td>
<td>hello</td>
</tr>
<tr>
<td>1</td>
<td>43</td>
<td>world</td>
</tr>
</tbody>
</table>

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. Therefore, it is strongly recommended not to use rowids as identifiers.
Note. The rowid values are stable within a transaction.

Note. 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/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 catalog search path globally
SET GLOBAL search_path = 'db1,db2'
-- 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.

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.

```sql
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 "Bob") 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.

Note. 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:

```sql
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 INT, dept TEXT, Jan INT, Feb INT, Mar INT, Apr INT, May INT, Jun INT);

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;

<table>
<thead>
<tr>
<th>empid</th>
<th>dept</th>
<th>Jan</th>
<th>Feb</th>
<th>Mar</th>
<th>Apr</th>
<th>May</th>
<th>Jun</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>electronics</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>clothes</td>
<td>10</td>
<td>20</td>
<td>30</td>
<td>40</td>
<td>50</td>
<td>60</td>
</tr>
<tr>
<td>3</td>
<td>cars</td>
<td>100</td>
<td>200</td>
<td>300</td>
<td>400</td>
<td>500</td>
<td>600</td>
</tr>
</tbody>
</table>

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;

<table>
<thead>
<tr>
<th>empid</th>
<th>dept</th>
<th>month</th>
<th>sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>electronics</td>
<td>Jan</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>electronics</td>
<td>Feb</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>electronics</td>
<td>Mar</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>electronics</td>
<td>Apr</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>electronics</td>
<td>May</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>electronics</td>
<td>Jun</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>clothes</td>
<td>Jan</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>clothes</td>
<td>Feb</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>clothes</td>
<td>Mar</td>
<td>30</td>
</tr>
<tr>
<td>2</td>
<td>clothes</td>
<td>Apr</td>
<td>40</td>
</tr>
<tr>
<td>2</td>
<td>clothes</td>
<td>May</td>
<td>50</td>
</tr>
<tr>
<td>2</td>
<td>clothes</td>
<td>Jun</td>
<td>60</td>
</tr>
<tr>
<td>3</td>
<td>cars</td>
<td>Jan</td>
<td>100</td>
</tr>
</tbody>
</table>
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.

```sql
UNPIVOT monthly_sales
ON COLUMNS(* EXCLUDE (empid, dept))
INTO
  NAME month
  VALUE sales;
```

<table>
<thead>
<tr>
<th>empid</th>
<th>dept</th>
<th>month</th>
<th>sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>electronics</td>
<td>Jan</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>electronics</td>
<td>Feb</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>electronics</td>
<td>Mar</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>electronics</td>
<td>Apr</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>electronics</td>
<td>May</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>electronics</td>
<td>Jun</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>clothes</td>
<td>Jan</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>clothes</td>
<td>Feb</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>clothes</td>
<td>Mar</td>
<td>30</td>
</tr>
<tr>
<td>2</td>
<td>clothes</td>
<td>Apr</td>
<td>40</td>
</tr>
<tr>
<td>2</td>
<td>clothes</td>
<td>May</td>
<td>50</td>
</tr>
<tr>
<td>2</td>
<td>clothes</td>
<td>Jun</td>
<td>60</td>
</tr>
<tr>
<td>3</td>
<td>cars</td>
<td>Jan</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>cars</td>
<td>Feb</td>
<td>200</td>
</tr>
<tr>
<td>3</td>
<td>cars</td>
<td>Mar</td>
<td>300</td>
</tr>
</tbody>
</table>
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;
```

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
```

---

<table>
<thead>
<tr>
<th>empid</th>
<th>dept</th>
<th>month</th>
<th>sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>cars</td>
<td>Apr</td>
<td>400</td>
</tr>
<tr>
<td>3</td>
<td>cars</td>
<td>May</td>
<td>500</td>
</tr>
<tr>
<td>3</td>
<td>cars</td>
<td>Jun</td>
<td>600</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>empid</th>
<th>dept</th>
<th>quarter</th>
<th>month_1_sales</th>
<th>month_2_sales</th>
<th>month_3_sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>electronics</td>
<td>q1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>electronics</td>
<td>q2</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>clothes</td>
<td>q1</td>
<td>10</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>2</td>
<td>clothes</td>
<td>q2</td>
<td>40</td>
<td>50</td>
<td>60</td>
</tr>
<tr>
<td>3</td>
<td>cars</td>
<td>q1</td>
<td>100</td>
<td>200</td>
<td>300</td>
</tr>
<tr>
<td>3</td>
<td>cars</td>
<td>q2</td>
<td>400</td>
<td>500</td>
<td>600</td>
</tr>
</tbody>
</table>
```sql
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.

```sql
SELECT *
FROM (UNPIVOT monthly_sales
    ON COLUMNS(* EXCLUDE (empid, dept))
    INTO
        NAME  month
        VALUE sales
) unpivot_alias;
```
DuckDB Documentation

<table>
<thead>
<tr>
<th>empid</th>
<th>dept</th>
<th>month</th>
<th>sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>electronics</td>
<td>apr</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>electronics</td>
<td>may</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>electronics</td>
<td>jun</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>clothes</td>
<td>jan</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>clothes</td>
<td>feb</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>clothes</td>
<td>mar</td>
<td>30</td>
</tr>
<tr>
<td>2</td>
<td>clothes</td>
<td>apr</td>
<td>40</td>
</tr>
<tr>
<td>2</td>
<td>clothes</td>
<td>may</td>
<td>50</td>
</tr>
<tr>
<td>2</td>
<td>clothes</td>
<td>jun</td>
<td>60</td>
</tr>
<tr>
<td>3</td>
<td>cars</td>
<td>jan</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>cars</td>
<td>feb</td>
<td>200</td>
</tr>
<tr>
<td>3</td>
<td>cars</td>
<td>mar</td>
<td>300</td>
</tr>
<tr>
<td>3</td>
<td>cars</td>
<td>apr</td>
<td>400</td>
</tr>
<tr>
<td>3</td>
<td>cars</td>
<td>may</td>
<td>500</td>
</tr>
<tr>
<td>3</td>
<td>cars</td>
<td>jun</td>
<td>600</td>
</tr>
</tbody>
</table>

**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)  
);
```
### 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.

```sql
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
  )
)
```

<table>
<thead>
<tr>
<th>empid</th>
<th>dept</th>
<th>quarter</th>
<th>month_1_sales</th>
<th>month_2_sales</th>
<th>month_3_sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>electronics</td>
<td>q1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>electronics</td>
<td>q2</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>clothes</td>
<td>q1</td>
<td>10</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>2</td>
<td>clothes</td>
<td>q2</td>
<td>40</td>
<td>50</td>
<td>60</td>
</tr>
<tr>
<td>3</td>
<td>cars</td>
<td>q1</td>
<td>100</td>
<td>200</td>
<td>300</td>
</tr>
<tr>
<td>3</td>
<td>cars</td>
<td>q2</td>
<td>400</td>
<td>500</td>
<td>600</td>
</tr>
</tbody>
</table>

**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.

```sql
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;

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>original</td>
</tr>
<tr>
<td>2</td>
<td>original 2</td>
</tr>
</tbody>
</table>

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;

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>new</td>
</tr>
<tr>
<td>2</td>
<td>new 2</td>
</tr>
</tbody>
</table>
```

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 `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 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;

<table>
<thead>
<tr>
<th>name</th>
<th>revenue</th>
<th>country_code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paris</td>
<td>800</td>
<td>FR</td>
</tr>
<tr>
<td>Lyon</td>
<td>300</td>
<td>FR</td>
</tr>
<tr>
<td>Brussels</td>
<td>400</td>
<td>BE</td>
</tr>
</tbody>
</table>

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
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 alone does nothing and is at present provided for PostgreSQL-compatibility. The VACUUM ANALYZE statement recomputes table statistics if they have become stale due to table updates or deletions.

Examples

-- No-op
VACUUM;
-- Rebuild database statistics
VACUUM ANALYZE;
-- Rebuild statistics for the table & column
VACUUM ANALYZE memory.main.my_table(my_column);
-- Not supported
VACUUM FULL; -- error

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 columns in a table, and provide an alias
SELECT col1 + col2 AS res, sqrt(col1) AS root 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 regex 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.

Note. 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 regex 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.

Note. 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.

Note. 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 guarantees 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, INT[] is transformed into INT. STRUCT(a INT, b INT) is transformed into a INT, b INT. The unnest function can be used to transform nested types into regular scalar types, which makes them easier to operate on.

FROM & 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 "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
**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**  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.

```sql
-- return all pairs of rows
SELECT a.*, b.* FROM a CROSS JOIN 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:

```sql
CREATE TABLE l_regions (  
    r_regionkey INTEGER NOT NULL PRIMARY KEY,  
    r_name CHAR(25) NOT NULL,  
    r_comment VARCHAR(152)
);
```

```sql
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 to 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;

**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 and anti joins provide the same logic as (NOT) IN statements.

-- return a list of cars that have a valid region.
SELECT cars.name, cars.manufacturer
FROM cars SEMI JOIN region
ON cars.region = region.id;

-- return a list of cars with no recorded safety data.
SELECT cars.name, cars.manufacturer
FROM cars ANTI JOIN safety_data
ON cars.safety_report_id = safety_data.report_id;

**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);

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

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);

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>1</td>
<td>101</td>
</tr>
</tbody>
</table>

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);

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>
Note. 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:

```sql
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 (like 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:

```sql
-- treat two data frames as a single table
SELECT df1.*, df2.*
FROM df1 POSITIONAL JOIN df2;
```

Positional joins are always FULL OUTER joins.

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:

```sql
-- 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.)

```sql
-- 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 (>=):

```sql
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:

```sql
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");
```

**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**

```sql
-- select all rows that have id equal to 3
SELECT *
FROM table_name
WHERE id = 3;

-- select all rows that match the given case-insensitive LIKE expression
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 (Ex: Prevents any 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;

-- GROUP BY city, street_name
;

-- 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
;
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
-- () signifies 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, ());
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 UNION statement:

```sql
-- 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.

GROUPING (alias GROUPING_ID) is a special aggregate function that can be used in combination with grouping sets. The GROUPING function takes as parameters a group, and returns 0 if the group is included in the grouping for that row, or 1 otherwise. This is primarily useful because the grouping columns by which we do not aggregate return NULL, which is ambiguous with groups that are actually the value NULL. The GROUPING (or GROUPING_ID) function can be used to distinguish these two cases.
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);

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 (Ex: 1. Note that it is 1-indexed), 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 if no modifiers are provided, DuckDB sorts ASC 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.

**Note.** Using ASC NULLS LAST as default the default sorting order was a breaking change in version 0.8.0. Prior to 0.8.0, DuckDB sorted using ASC NULLS FIRST.

```sql
-- change the default null sorting order to either NULLS FIRST and NULLS LAST
SET default_null_order = 'NULLS FIRST';
-- change 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 UTF8 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:

```sql
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', '1111-0001';
```

-- select the addresses, ordered by city name using the default null order and default order
```sql
SELECT *
FROM addresses
ORDER BY city;
```

-- select the addresses, ordered by city name in descending order with nulls at the end
```sql
SELECT *
FROM addresses
ORDER BY city DESC;
```
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
-- Order from left to right (by address, then by city, then by zip) in ascending order
SELECT *
FROM addresses
ORDER BY ALL;

<table>
<thead>
<tr>
<th>address</th>
<th>city</th>
<th>zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>111 Duck Duck Goose Ln</td>
<td>Duck Town</td>
<td>11111</td>
</tr>
<tr>
<td>111 Duck Duck Goose Ln</td>
<td>Duck Town</td>
<td>11111-0001</td>
</tr>
<tr>
<td>111 Duck Duck Goose Ln</td>
<td>DuckTown</td>
<td>11111</td>
</tr>
<tr>
<td>123 Quack Blvd</td>
<td>DuckTown</td>
<td>11111</td>
</tr>
</tbody>
</table>

-- Order from left to right (by address, then by city, then by zip) in descending order
SELECT *
FROM addresses
ORDER BY ALL DESC;

<table>
<thead>
<tr>
<th>address</th>
<th>city</th>
<th>zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>123 Quack Blvd</td>
<td>Duck Town</td>
<td>11111</td>
</tr>
<tr>
<td>111 Duck Duck Goose Ln</td>
<td>DuckTown</td>
<td>11111</td>
</tr>
<tr>
<td>111 Duck Duck Goose Ln</td>
<td>Duck Town</td>
<td>11111-0001</td>
</tr>
<tr>
<td>111 Duck Duck Goose Ln</td>
<td>Duck Town</td>
<td>11111</td>
</tr>
</tbody>
</table>

Syntax

LIMIT Clause

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**

```sql
-- 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;
```

**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**

```sql
-- 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);

The unnest 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.

Unnesting Lists

-- unnest a list, generating 3 rows (1, 2, 3)
SELECT unnest([1, 2, 3]);
-- unnest a scalar list, generating 3 rows ((1, 10), (2, 11), (3, NULL))
SELECT unnest([1, 2, 3]), unnest([10, 11]);
-- unnest a scalar list, generating 3 rows ((1, 10), (2, 10), (3, 10))
SELECT unnest([1, 2, 3]), 10;
-- unnest a list column generated 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 will emit one tuple per entry in the list. When unnest is combined with regular scalar expressions, those expressions are repeated for every entry in the list. 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 will be padded with NULL values.

An empty list and a NULL list will both unnest to zero elements.

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.

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.

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;

<table>
<thead>
<tr>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
</tr>
</tbody>
</table>

-- create two CTEs, where the second CTE references the first CTE
WITH cte AS (SELECT 42 AS i),
cte2 AS (SELECT i*100 AS x FROM cte)
SELECT * FROM cte2;

<table>
<thead>
<tr>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>4200</td>
</tr>
</tbody>
</table>
Materialized CTEs

By default, CTEs are inlined into the main query. Inlining can result in duplicate work, because the definition is copied for each reference. Take this query for example:

```sql
WITH t(x) AS (Q_t)
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:

```sql
SELECT * FROM (Q_t) AS t1(x),
    (Q_t) AS t2(x),
    (Q_t) AS t3(x);
```

If `Q_t` is expensive, materializing it with the `MATERIALIZED` keyword can improve performance. In this case, `Q_t` is evaluated only once.

```sql
WITH t(x) AS MATERIALIZED (Q_t)
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.

Tree Traversal

`WITH RECURSIVE` can be used to traverse trees. For example, take a hierarchy of tags:
CREATE TABLE tag (id INT, name VARCHAR, subclassof INT);

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 t1.id, t1.name, t1.[name] || t2.path AS path
    FROM tag t1, tag_hierarchy t2
    WHERE t1.id = t2.source AND t1.subclassof = t2.id
)

SELECT * FROM tag_hierarchy WHERE path LIKE '%Oasis%';
```sql
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';
```

**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 INT, node2id INT);
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, 2, and 5.

**Enumerate All Paths from a Node** The following query returns all paths starting in node 1:

```sql
WITH RECURSIVE paths(startNode, endNode, path) AS (
    -- define the path as the first edge of the traversal
    SELECT 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 node2id != ALL(paths.path)
)
SELECT startNode, endNode, path
FROM paths
ORDER BY length(path), path;
```

<table>
<thead>
<tr>
<th>startNode</th>
<th>endNode</th>
<th>path</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>[1, 3]</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>[1, 5]</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>[1, 3, 5]</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>[1, 3, 8]</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>[1, 3, 10]</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>[1, 5, 3]</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>[1, 5, 4]</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>[1, 5, 8]</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>[1, 3, 5, 4]</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>[1, 3, 5, 8]</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>[1, 5, 3, 8]</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>[1, 5, 3, 10]</td>
</tr>
</tbody>
</table>

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 \textbf{(unweighted) shortest paths} are of interest. To find these, the second half of the \texttt{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:

\begin{verbatim}
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 (SELECT 1
    FROM paths previous_paths
    WHERE list_contains(previous_paths.path, node2id))
)
SELECT startNode, endNode, path
FROM paths
ORDER BY length(path), path;
\end{verbatim}

\begin{center}
\begin{tabular}{|c|c|c|}
\hline
startNode & endNode & path \\
\hline
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] \\
\hline
\end{tabular}
\end{center}

\textbf{Enumerate Unweighted Shortest Paths between Two Nodes}  \texttt{WITH RECURSIVE} can also be used to find all \textbf{(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 \texttt{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):

\begin{verbatim}
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 (SELECT 1
    FROM paths previous_paths
    WHERE list_contains(previous_paths.path, node2id))
)
SELECT startNode, endNode, path
FROM paths
ORDER BY length(path), path;
\end{verbatim}
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 (SELECT 1  
                         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;

<table>
<thead>
<tr>
<th>startNode</th>
<th>endNode</th>
<th>path</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
<td>[1, 3, 8]</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>[1, 5, 8]</td>
</tr>
</tbody>
</table>

Common Table Expressions

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 \textit{WINDOW functions}. This filtering of results is similar to how a \textit{HAVING} clause filters the results of aggregate functions applied based on the \textit{GROUP BY} clause.

The QUALIFY clause avoids the need for a subquery or \textit{WITH} clause to perform this filtering (much like \textit{HAVING} avoids a subquery). An example using a \textit{WITH} clause instead of QUALIFY is included below the QUALIFY examples.

Note that this is filtering based on \textit{WINDOW functions}, not necessarily based on the \textit{WINDOW clause}. The \textit{WINDOW} clause is optional and can be used to simplify the creation of multiple \textit{WINDOW} function expressions.

The position of where to specify a QUALIFY clause is following the \textit{WINDOW clause} in a \textit{SELECT} statement (\textit{WINDOW} does not need to be specified), and before the \textit{ORDER BY}.

Examples

Each of the following examples produce the same output, located below.

```sql
-- 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 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;

<table>
<thead>
<tr>
<th>schema_name</th>
<th>function_name</th>
<th>function_rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>__postfix</td>
<td>1</td>
</tr>
<tr>
<td>main</td>
<td>!~~</td>
<td>2</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>col_description</td>
<td>1</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>format_pg_type</td>
<td>2</td>
</tr>
</tbody>
</table>
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

```sql
-- 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);
```

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. FILTERs are not currently able to be used when the aggregate function is in a windowing context.

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

```sql
-- Compare total row count to:
-- The number of rows where i <= 5
-- The number of rows where i is odd
SELECT 452
```
count(*) AS total_rows,
count(*) FILTER (WHERE i <= 5) AS lte_five,
count(*) FILTER (WHERE i % 2 = 1) AS odds
FROM generate_series(1, 10) tbl(i);

<table>
<thead>
<tr>
<th></th>
<th>total_rows</th>
<th>lte_five</th>
<th>odds</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>5</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

-- Different aggregate functions may be used, and multiple WHERE expressions are also permitted
-- The sum of i for rows where i <= 5
-- The median of i where i is odd

SELECT
  sum(i) FILTER (WHERE i <= 5) AS lte_five_sum,
  median(i) FILTER (WHERE i % 2 = 1) AS odds_median,
  median(i) FILTER (WHERE i % 2 = 1 AND i <= 5) AS odds_lte_five_median
FROM generate_series(1, 10) tbl(i);

<table>
<thead>
<tr>
<th></th>
<th>lte_five_sum</th>
<th>odds_median</th>
<th>odds_lte_five_median</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>15</td>
<td>5.0</td>
<td>3.0</td>
</tr>
</tbody>
</table>

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, 100000000) tbl(i)
  ) tbl;

-- "Pivot" the data out by year (move each year out to a separate column)
SELECT
  count(i) FILTER (WHERE year = 2022) AS "2022",
  count(i) FILTER (WHERE year = 2023) AS "2023",
  count(i) FILTER (WHERE year = 2024) AS "2024",
  count(i) FILTER (WHERE year = 2025) AS "2025",
  count(i) FILTER (WHERE 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;

<table>
<thead>
<tr>
<th></th>
<th>2022</th>
<th>2023</th>
<th>2024</th>
<th>2025</th>
<th>NULLs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2500000</td>
<td>2500000</td>
<td>2500000</td>
<td>1250000</td>
<td>1250000</td>
</tr>
</tbody>
</table>

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 (WHERE year = 2022) AS "2022",
  first(i) FILTER (WHERE year = 2023) AS "2023",
  first(i) FILTER (WHERE year = 2024) AS "2024",
  first(i) FILTER (WHERE year = 2025) AS "2025",
  first(i) FILTER (WHERE year IS NULL) AS "NULLs"
FROM stacked_data;

<table>
<thead>
<tr>
<th></th>
<th>2022</th>
<th>2023</th>
<th>2024</th>
<th>2025</th>
<th>NULLs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1474561</td>
<td>2580480</td>
<td>5074944</td>
<td>7643136</td>
<td>8750000</td>
</tr>
</tbody>
</table>

-- 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;
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 and EXCEPT clauses.

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 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.

Examples

-- the values [0..10) and [0..5)
SELECT * FROM range(10) t1 UNION ALL SELECT * FROM range(5) t2;
-- the values [0..10) (' UNION' eliminates duplicates)
SELECT * FROM range(10) t1 UNION SELECT * FROM range(5) t2;
-- the values [0..5] (all values that are both in t1 and t2)
SELECT * FROM range(10) t1 INTERSECT SELECT * FROM range(5) t2;
-- the values [5..10]
SELECT * FROM range(10) t1 EXCEPT SELECT * FROM range(5) t2;
-- two rows, (24, NULL) and (NULL, Amsterdam)
SELECT 24 AS id UNION ALL BY NAME SELECT 'Amsterdam' AS City;

Syntax

Example Tables

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');
UNION (ALL)

The UNION clause can be used to combine rows from multiple queries. The queries are required to have the same number of columns and the same column types.

The UNION clause performs duplicate elimination by default - only unique rows will be included in the result.

UNION ALL returns all rows of both queries without duplicate elimination.

```
SELECT city FROM capitals UNION SELECT city FROM weather;
-- Amsterdam, Berlin, Seattle

SELECT city FROM capitals UNION ALL SELECT city FROM weather;
-- Amsterdam, Amsterdam, Berlin, Seattle
```

INTERSECT

The INTERSECT clause can be used to select all rows that occur in the result of both queries. Note that INTERSECT performs duplicate elimination, so only unique rows are returned.

```
SELECT city FROM capitals INTERSECT SELECT city FROM weather;
-- Amsterdam
```

EXCEPT

The EXCEPT clause can be used to select all rows that only occur in the left query. Note that EXCEPT performs duplicate elimination, so only unique rows are returned.

```
SELECT city FROM capitals EXCEPT SELECT city FROM weather;
-- Berlin
```

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.

```
SELECT * FROM capitals UNION BY NAME SELECT * FROM weather;
```

<table>
<thead>
<tr>
<th>city</th>
<th>country</th>
<th>degrees</th>
<th>date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amsterdam</td>
<td>NULL</td>
<td>10</td>
<td>2022-10-14</td>
</tr>
<tr>
<td>Seattle</td>
<td>NULL</td>
<td>8</td>
<td>2022-10-12</td>
</tr>
<tr>
<td>Amsterdam</td>
<td>NL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>Berlin</td>
<td>Germany</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>
UNION BY NAME performs duplicate elimination, whereas UNION ALL BY NAME does not.

**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.

```sql
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:

```sql
PREPARE query_person AS
SELECT *
FROM person
WHERE starts_with(name, ?)
    AND age >= ?;
```

Using the CLI client, the statement is executed as follows.

```sql
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:

```sql
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 named 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);
```

## 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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Aliases</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIGINT</td>
<td>INT8, LONG</td>
<td>signed eight-byte integer</td>
</tr>
<tr>
<td>BIT</td>
<td>BITSTRING</td>
<td>string of 1s and 0s</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>BOOL, LOGICAL</td>
<td>logical boolean (true/false)</td>
</tr>
<tr>
<td>BLOB</td>
<td>BYTEA, BINARY, VARBINARY</td>
<td>variable-length binary data</td>
</tr>
<tr>
<td>DATE</td>
<td></td>
<td>calendar date (year, month day)</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>FLOAT8, NUMERIC, DECIMAL</td>
<td>double precision floating-point number (8 bytes)</td>
</tr>
<tr>
<td>DECIMAL(prec, scale)</td>
<td></td>
<td>fixed-precision number with the given width (precision) and scale</td>
</tr>
<tr>
<td>HUGEINT</td>
<td>INT4, INT, SIGNED</td>
<td>signed sixteen-byte integer</td>
</tr>
<tr>
<td>INTEGER</td>
<td></td>
<td>signed four-byte integer</td>
</tr>
</tbody>
</table>
DuckDB Documentation

<table>
<thead>
<tr>
<th>Name</th>
<th>Aliases</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTERVAL</td>
<td></td>
<td>date / time delta</td>
</tr>
<tr>
<td>REAL</td>
<td>FLOAT4, FLOAT</td>
<td>single precision floating-point number (4 bytes)</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>INT2, SHORT</td>
<td>signed two-byte integer</td>
</tr>
<tr>
<td>TIME</td>
<td></td>
<td>time of day (no time zone)</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>DATETIME</td>
<td>combination of time and date</td>
</tr>
<tr>
<td>TIMESTAMP WITH TIME ZONE</td>
<td>TIMESTAMPTZ</td>
<td>combination of time and date that uses the current time zone</td>
</tr>
<tr>
<td>TINYINT</td>
<td>INT1</td>
<td>signed one-byte integer</td>
</tr>
<tr>
<td>UBIGINT</td>
<td></td>
<td>unsigned eight-byte integer</td>
</tr>
<tr>
<td>UINTGER</td>
<td></td>
<td>unsigned four-byte integer</td>
</tr>
<tr>
<td>USMALLINT</td>
<td></td>
<td>unsigned two-byte integer</td>
</tr>
<tr>
<td>UTINYINT</td>
<td></td>
<td>unsigned one-byte integer</td>
</tr>
<tr>
<td>UUID</td>
<td></td>
<td>UUID data type</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>CHAR, BPCHAR, TEXT, STRING</td>
<td>variable-length character string</td>
</tr>
</tbody>
</table>

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, STRUCT, MAP, and UNION. Each supports different use cases and has a different structure.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Rules when used in a column</th>
<th>Build from values</th>
<th>Define in DDL/CREATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARRAY</td>
<td>An ordered, fixed-length sequence of data values of the same type.</td>
<td>Each row must have the same data type within each instance of the ARRAY and the same number of elements of elements.</td>
<td>[1, 2, 3]</td>
<td>INT[3]</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Rules when used in a column</td>
<td>Build from values</td>
<td>Define in DDL/CREATE</td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
<td>-----------------------------</td>
<td>-------------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>LIST</td>
<td>An ordered sequence of data values of the same type.</td>
<td>Each row must have the same data type within each instance of the LIST, but can have any number of elements.</td>
<td>[1, 2, 3]</td>
<td>INT[]</td>
</tr>
<tr>
<td>MAP</td>
<td>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.</td>
<td>Rows may have different keys.</td>
<td>map([1, 2], ['a', 'b'])</td>
<td>MAP(INT, VARCHAR)</td>
</tr>
<tr>
<td>STRUCT</td>
<td>A dictionary of multiple named values, where each key is a string, but the value can be a different type for each key.</td>
<td>Each row must have the same keys.</td>
<td>{'i': 42, 'j': 'a'}</td>
<td>STRUCT(i INT, j VARCHAR)</td>
</tr>
<tr>
<td>UNION</td>
<td>A union of multiple alternative data types, storing one of them in each value at a time. A union also contains a discriminator &quot;tag&quot; value to inspect and access the currently set member type.</td>
<td>Rows may be set to different member types of the union.</td>
<td>union_value(num := 2)</td>
<td>UNION(num INT, text VARCHAR)</td>
</tr>
</tbody>
</table>

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 lists
SELECT {'birds': ['duck', 'goose', 'heron'], 'aliens': NULL, 'amphibians': ['frog', 'toad']};

-- Struct with list of maps
SELECT {'test': [map([1, 5], [42.1, 45]), map([1, 5], [42.1, 45])]};

-- A list of unions
**ArrayType**

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.

**Note.** 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.

```sql
-- 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, but the dimensions have to match up!
SELECT [3, 2, 1]::INT[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>]` syntax. For example, to create an array field for 3 integers, run:

```sql
CREATE TABLE array_table (id INT, arr INT[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:

```sql
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;
```

```
<table>
<thead>
<tr>
<th>id</th>
<th>element</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>4</td>
</tr>
</tbody>
</table>
```

Using the slicing notation returns a LIST:

```sql
SELECT id, arr[1:2] AS elements FROM array_table;
```

```
<table>
<thead>
<tr>
<th>id</th>
<th>elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>[1, 2]</td>
</tr>
<tr>
<td>20</td>
<td>[4, 5]</td>
</tr>
</tbody>
</table>
```

### Functions

All LIST functions work with the ARRAY type. Additionally, several ARRAY-native functions are also supported. In the following, `l1` stands for the 3-element list created by `array_value(1.0::FLOAT, 2.0::FLOAT, 3.0::FLOAT)` and `l2` stands for `array_value(2.0::FLOAT, 3.0::FLOAT, 4.0::FLOAT)`.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>array_value(index)</code></td>
<td>Create an ARRAY containing the argument values.</td>
<td><code>array_value(1.0::FLOAT, 2.0::FLOAT, 3.0::FLOAT)</code></td>
<td><code>[1.0, 2.0, 3.0]</code></td>
</tr>
<tr>
<td><code>array_cross_product(array1, array2)</code></td>
<td>Compute the cross product of two arrays of size 3. The array elements can not be NULL.</td>
<td><code>array_cross_product(l1, l2)</code></td>
<td><code>[-1.0, 2.0, -1.0]</code></td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>Example</td>
<td>Result</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>----------------------------------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td><code>array_cosine_similarity(array1, array2)</code></td>
<td>Computes the cosine similarity between two arrays of the same size.</td>
<td><code>array_cosine_similarity(l1, l2)</code></td>
<td>0.9925833</td>
</tr>
<tr>
<td><code>array_distance(array1, array2)</code></td>
<td>Computes the distance between two arrays of the same size.</td>
<td><code>array_distance(l1, l2)</code></td>
<td>1.7320508</td>
</tr>
<tr>
<td><code>array_inner_product(array1, array2)</code></td>
<td>Computes the inner product between two arrays of the same size.</td>
<td><code>array_inner_product(l1, l2)</code></td>
<td>20.0</td>
</tr>
<tr>
<td><code>array_dot_product(array1, array2)</code></td>
<td>Alias for <code>array_inner_product(array1, array2)</code></td>
<td><code>array_dot_product(l1, l2)</code></td>
<td>20.0</td>
</tr>
</tbody>
</table>

**Examples**

```sql
-- create sample data
CREATE TABLE x (i INT, v FLOAT[3]);
CREATE TABLE y (i INT, 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.

### Functions

See [Nested Functions](#).

### Bitstring Type

<table>
<thead>
<tr>
<th>Name</th>
<th>Aliases</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIT</td>
<td>BITSTRING</td>
<td>variable-length strings of 1s and 0s</td>
</tr>
</tbody>
</table>

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.

```sql
-- create a bitstring
SELECT '101010'::BIT
-- create a bitstring with predefined length
-- the resulting bitstring will be left-padded with zeroes. This returns 000000101011
SELECT bitstring('0101011', 12);
```

### Functions

See [Bitstring Functions](#).

### Blob Type

<table>
<thead>
<tr>
<th>Name</th>
<th>Aliases</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLOB</td>
<td>BYTEA, BINARY, VARBINARY</td>
<td>variable-length binary data</td>
</tr>
</tbody>
</table>

464
The blob (Binary Large Object) 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 4GB 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

<table>
<thead>
<tr>
<th>Name</th>
<th>Aliases</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOLEAN</td>
<td>BOOL</td>
<td>logical boolean(true/false)</td>
</tr>
</tbody>
</table>

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 value (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:
```
Conjunctions

The AND/OR conjunctions can be used to combine boolean values.

Below is the truth table for the AND conjunction (i.e., \(x \land y\)).

<table>
<thead>
<tr>
<th>X</th>
<th>X AND true</th>
<th>X AND false</th>
<th>X AND NULL</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>false</td>
<td>NULL</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>false</td>
<td>NULL</td>
</tr>
</tbody>
</table>

Below is the truth table for the OR conjunction (i.e., \(x \lor y\)).

<table>
<thead>
<tr>
<th>X</th>
<th>X OR true</th>
<th>X OR false</th>
<th>X OR NULL</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
<td>NULL</td>
</tr>
<tr>
<td>NULL</td>
<td>true</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>

Expressions

See Logical Operators and Comparison Operators.

Date Types

<table>
<thead>
<tr>
<th>Name</th>
<th>Aliases</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE</td>
<td></td>
<td>calendar date (year, month day)</td>
</tr>
</tbody>
</table>

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).

```sql
-- 20 September, 1992
SELECT DATE '1992-09-20';
```
There are also three special date values that can be used on input:

<table>
<thead>
<tr>
<th>Input String</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>epoch</td>
<td>1970-01-01 (Unix system day zero)</td>
</tr>
<tr>
<td>infinity</td>
<td>later than all other dates</td>
</tr>
<tr>
<td>-infinity</td>
<td>earlier than all other dates</td>
</tr>
</tbody>
</table>

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.

```sql
SELECT '-infinity '::DATE, 'epoch '::DATE, 'infinity '::DATE;
```

<table>
<thead>
<tr>
<th>Negative</th>
<th>Epoch</th>
<th>Positive</th>
</tr>
</thead>
<tbody>
<tr>
<td>-infinity</td>
<td>1970-01-01</td>
<td>infinity</td>
</tr>
</tbody>
</table>

**Functions**

See Date Functions.

**Enum Data Type**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENUM</td>
<td>Dictionary Encoding representing all possible string values of a column.</td>
</tr>
</tbody>
</table>

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});

For example:

-- Creates new user defined type 'mood' as an Enum
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');

-- This will fail since the mood type already exists
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy', 'anxious');

-- This will fail since Enums cannot hold null values
CREATE TYPE breed AS ENUM ('maltese', NULL);

-- This will fail since Enum values must be unique
CREATE TYPE breed AS ENUM ('maltese', 'maltese');

-- Create an enum from a select statement
-- First create an example table of values
CREATE TABLE my_inputs AS
SELECT 'duck' AS my_varchar UNION ALL
SELECT 'duck' AS my_varchar UNION ALL
SELECT 'goose' AS my_varchar;

-- 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;

<table>
<thead>
<tr>
<th>my_enum_range</th>
</tr>
</thead>
<tbody>
<tr>
<td>[duck, goose]</td>
</tr>
</tbody>
</table>

**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
);
Insert tuples in the `person` table:

```sql
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 a 'quackity-quack' value.

```sql
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.

```sql
SELECT * FROM person WHERE current_mood = 'sad';
```

<table>
<thead>
<tr>
<th>name</th>
<th>current_mood</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pagliacci</td>
<td>sad</td>
</tr>
</tbody>
</table>

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:

```sql
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

```sql
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:

```sql
-- 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;
```

<table>
<thead>
<tr>
<th>contains_a</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
</tr>
<tr>
<td>NULL</td>
</tr>
<tr>
<td>true</td>
</tr>
<tr>
<td>false</td>
</tr>
</tbody>
</table>
Create a new mood and table:

```sql
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:

```sql
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:

```sql
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:

```sql
DROP TYPE ${enum_name};
```

Currently, it is possible to drop Enums that are used in tables without affecting the tables.

**Note.** This feature is subject to change in future releases.

For example, this will fail since `person` has a catalog dependency to the `mood` type:

**Interval Type**

Intervals represent a period of time. This period can be measured in a specific unit or combination of units, for example years, days, or seconds. Intervals are generally used to modify timestamps or dates by either adding or subtracting them.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTERVAL</td>
<td>Period of time</td>
</tr>
</tbody>
</table>

An INTERVAL can be constructed by providing an amount together with a unit. Intervals can be added or subtracted from DATE or TIMESTAMP values.
-- 1 year
SELECT INTERVAL 1 YEAR;
-- add 1 year to a specific date
SELECT DATE '2000-01-01' + INTERVAL 1 YEAR;
-- subtract 1 year from a specific date
SELECT DATE '2000-01-01' - INTERVAL 1 YEAR;
-- construct an interval from a column, instead of a constant
SELECT INTERVAL (i) YEAR FROM range(1, 5) t(i);
-- construct an interval with mixed units
SELECT INTERVAL '1 month 1 day';
-- intervals greater than 24 hours/12 months/etc. are supported
SELECT '540:58:47.210'::INTERVAL;
SELECT INTERVAL '16 MONTHS';

-- WARNING:
-- If a decimal value is specified, it will be automatically rounded to an integer
-- To use more precise values, simply use a more granular date part
-- (In this example use 18 MONTHS instead of 1.5 YEARS)
-- The statement below is equivalent to to_years(CAST(1.5 AS INTEGER))
SELECT INTERVAL '1.5' YEARS; -- WARNING! This returns 2 years!

Details

The interval class represents a period of time using three distinct components: the month, day and microsecond. These three components are required because there is no direct translation between them. For example, a month does not correspond to a fixed amount of days. That depends on which month is referenced. February has fewer days than March.

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);
```

Difference between Dates

If we subtract two timestamps from one another, we obtain an interval describing the difference between the timestamps with the days and microseconds components. For example:

```
SELECT TIMESTAMP '2000-02-01 12:00:00' - TIMESTAMP '2000-01-01 11:00:00' AS diff;
```

```
   ┌──────────────────┐
   │ diff │
   ├──────────────────┤
   │ 31 days 01:00:00 │
   └──────────────────┘
```

The datediff function can be used to obtain the difference between two dates for a specific unit.
```sql
SELECT datediff('month', TIMESTAMP '2000-01-01 11:00:00', TIMESTAMP '2000-02-01 12:00:00') AS diff;
```

<table>
<thead>
<tr>
<th>diff</th>
<th>int64</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

**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. LISTS are typically used to store arrays of numbers, but can contain any uniform data type, including other LISTS and STRUCTs.

LISTs 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.

**Note.** 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.

```sql
-- 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 INT[], 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.

-- Note that we wrap the list creation in parenthesis so that it happens first.
-- This is only needed in our basic examples here, not when working with a list column
-- For example, this can't be parsed: SELECT ['a', 'b', 'c'][1]

<table>
<thead>
<tr>
<th>example</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT (['a', 'b', 'c'])[3]</td>
<td>'c'</td>
</tr>
<tr>
<td>SELECT (['a', 'b', 'c'])[-1]</td>
<td>'c'</td>
</tr>
<tr>
<td>SELECT (['a', 'b', 'c'])[2 + 1]</td>
<td>'c'</td>
</tr>
<tr>
<td>SELECT list_extract(['a', 'b', 'c'], 3)</td>
<td>'c'</td>
</tr>
<tr>
<td>SELECT (['a', 'b', 'c'])[1:2]</td>
<td>['a','b']</td>
</tr>
<tr>
<td>SELECT (['a', 'b', 'c'][:2])</td>
<td>['a','b']</td>
</tr>
<tr>
<td>SELECT (['a', 'b', 'c'])[-2:]</td>
<td>['b','c']</td>
</tr>
<tr>
<td>SELECT list_slice(['a', 'b', 'c'], 2, 3)</td>
<td>['b','c']</td>
</tr>
</tbody>
</table>

Ordering

The ordering is defined positionally. NULL values compare greater than all other values and are considered equal to each other.

Null Comparisons

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 Nested Functions.

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 (Ex: 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=1, key2=5}
SELECT MAP {'key1': 1, 'key2': 5};
-- Alternatively use the map_from_entries function. This returns {key1=1, key2=5}
SELECT map_from_entries([(key1, 1), (key2, 5)]);
-- A map with 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 map_table (map_col MAP(INT, 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 a list containing the value at a key's location. This returns [5]
-- Note that the expression in bracket notation must match the type of the map's key
SELECT MAP {'key1': 5, 'key2': 43}['key1'];
-- To retrieve the underlying value, use list selection syntax to grab the first element.
-- This returns 5
SELECT MAP {'key1': 5, 'key2': 43}['key1'][1];
-- If the element is not in the map, an empty list will be returned. Returns []
-- Note that the expression in bracket notation must match the type of the map's key else an error is returned
SELECT MAP {'key1': 5, 'key2': 43}['key3'];
-- The element_at function can also be used to retrieve a map value. This returns [5]
SELECT element_at(MAP {'key1': 5, 'key2': 43}, 'key1');
**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 *NULL*s, and a *NULL* nested value member will compare above a non-*NULL* nested value member.

**Functions**

See *Nested 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".

```sql
-- insert a null value into a table
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:

---

**Note.** 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*.

```sql
SELECT NULL = NULL;
-- returns NULL
SELECT NULL IS NOT DISTINCT FROM NULL;
-- returns true
SELECT NULL IS NULL;
-- returns true
```

**NULL and Functions**

A function that has input argument as *NULL* *usually* returns *NULL*.

```sql
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.

```sql
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.

```sql
SELECT ifnull(NULL, 'default_string');
-- default_string
SELECT ifnull(1, 'default_string');
-- 1
```

### NULL and Conjunctions

NULL values have special semantics in AND/OR conjunctions. For the ternary logic truth tables, see the Boolean Type 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.

```sql
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, UINTEGE, UBIGINT store whole unsigned numbers. Attempts to store negative numbers or values outside of the allowed range will result in an error.
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.

Fixed-Point Decimals

The data type DECIMAL(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.

Internally, decimals are represented as integers depending on their specified width.

<table>
<thead>
<tr>
<th>Width</th>
<th>Internal</th>
<th>Size (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-4</td>
<td>INT16</td>
<td>2</td>
</tr>
<tr>
<td>5-9</td>
<td>INT32</td>
<td>4</td>
</tr>
<tr>
<td>10-18</td>
<td>INT64</td>
<td>8</td>
</tr>
<tr>
<td>19-38</td>
<td>INT128</td>
<td>16</td>
</tr>
</tbody>
</table>

Performance can be impacted by using too large decimals when not required. In particular decimal values with a width above 19 are very 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 REAL and DOUBLE precision are inexact, 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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Aliases</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>REAL</td>
<td>FLOAT4, FLOAT</td>
<td>single precision floating-point number (4 bytes)</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>FLOAT8</td>
<td>double precision floating-point number (8 bytes)</td>
</tr>
</tbody>
</table>

Inexact means that some values cannot be converted exactly to the internal format and are stored as approximations, so that storing and retrieving a value might show slight discrepancies. Managing these errors and how they propagate through calculations is the subject of an entire branch of mathematics and computer science and will not be discussed here, except for the following points:

- If you require exact storage and calculations (such as for monetary amounts), use the numeric type instead.
- If you want to do complicated calculations with these types for anything important, especially if you rely on certain behavior in boundary cases (infinity, underflow), you should evaluate the implementation carefully.
- Comparing two floating-point values for equality might not always work as expected.

On most platforms, the REAL type has a range of at least $1 \times 10^{-37}$ to $1 \times 10^{+37}$ with a precision of at least 6 decimal digits. The DOUBLE type typically has a range of around $1 \times 10^{-307}$ to $1 \times 10^{+308}$ with a precision of at least 15 digits. Values that are too large or too small will cause an error. Rounding might take place if the precision of an input number is too high. Numbers too close to zero that are not representable as distinct from zero will cause an underflow error.

In addition to ordinary numeric values, the floating-point types have several special values:

- Infinity
- -Infinity
- NaN

These represent the IEEE 754 special values "infinity", "negative infinity", and "not-a-number", respectively. (On a machine whose floating-point arithmetic does not follow IEEE 754, these values will probably not work as expected.) When writing these values as constants in an SQL command, you must put quotes around them, for example: UPDATE table SET x = '-Infinity'. On input, these strings are recognized in a case-insensitive manner.

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 LISTs.

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 nested functions docs 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 or the equivalent array notation `{ 'name' : expr, ... }` notation. The expressions can be constants or arbitrary expressions.

-- Struct of integers
`SELECT { 'x' : 1, 'y' : 2, 'z' : 3 };`
-- Struct of strings with a NULL value
`SELECT { 'yes' : 'duck', 'maybe' : 'goose', 'huh' : NULL, 'no' : 'heron' };`
-- Struct with a different type for each key
`SELECT { 'key1' : 'string', 'key2' : 1, 'key3' : 12.345 };`
-- Note the lack of single quotes around the keys and the use of the := operator
`SELECT struct_pack(key1 := 'value1', key2 := 42);`
-- Struct of structs with NULL values
`SELECT { 'birds':
    'aliens':
        NULL,
    'amphibians':
        { 'yes' : 'frog', 'maybe' : 'salamander', 'huh' : 'dragon', 'no' : 'toad' }
};`
-- Create a struct from columns and/or expressions using the row function.
-- This returns { '' : 1, '' : 2, '' : a }
`SELECT row(x, x + 1, y) FROM (SELECT 1 AS x, 'a' AS y);`
-- If using multiple expressions when creating a struct, the row function is optional
-- This also returns { '' : 1, '' : 2, '' : a }
`SELECT (x, x + 1, y) FROM (SELECT 1 AS x, 'a' AS y);`

Adding Field(s)/Value(s) to Structs
-- Add to a Struct of integers
SELECT struct_insert({'a': 1, 'b': 2, 'c': 3}, d := 4);

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. This returns 1
-- 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 key contains a space, simply wrap it in double quotes. This returns 1
-- Note: Use double quotes not single quotes
-- This is because this action is most similar to selecting a column from within the struct
SELECT a."x space" FROM (SELECT {'x space': 1, 'y': 2, 'z': 3} AS a);

-- Bracket notation may also be used. This returns 1
-- Only constant expressions may be used inside the brackets (no columns)
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');

Struct.*  Rather than retrieving a single key from a struct, star notation (*) 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.

-- All keys within a struct can be returned as separate columns using *
SELECT a.* FROM (SELECT {'x': 1, 'y': 2, 'z': 3} AS a);

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

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

```sql
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

```sql
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:

```sql
-- 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));
-- The table will contain a single entry:
-- {'v': a, 'i': 42}
```

```sql
-- The following produces the same result as above
CREATE TABLE t1 AS (SELECT ROW('a', 42)::STRUCT(v VARCHAR, i INTEGER));
```

```sql
-- Initializing a struct column with the row function will fail
CREATE TABLE t2 AS SELECT ROW('a');
-- The following error is thrown:
-- "Error: Invalid Input Error: A table cannot be created from an unnamed struct"
```

When casting structs, the names of fields have to match. Therefore, the following query will fail:

```sql
SELECT a::STRUCT(y INTEGER) AS b
FROM
 (SELECT {'x': 42} AS a);
```

```sql
Error: Mismatch Type Error: Type STRUCT(x INTEGER) does not match with STRUCT(y INTEGER). Cannot cast STRUCTs with different names
```

A workaround for this would be to use `struct_pack` instead:
SELECT struct_pack(y := a.x) AS b
FROM
  (SELECT {'x': 42} AS a);

Note. This behavior was introduced in DuckDB v0.9.0. Previously, this query ran successfully and returned struct {'y': 42} as column b.

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 Nested Functions.

Text Types

In DuckDB, strings can be stored in the VARCHAR field.

<table>
<thead>
<tr>
<th>Name</th>
<th>Aliases</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VARCHAR</td>
<td>CHAR, BPCHAR, TEXT, STRING</td>
<td>variable-length character string</td>
</tr>
<tr>
<td>VARCHAR(n)</td>
<td></td>
<td>variable-length character string with maximum length n</td>
</tr>
</tbody>
</table>

It is possible to supply a number along with the type by initializing a type as VARCHAR(n), where n is a positive integer. Note that specifying this length is not required and has no effect on the system. Specifying this length will not improve performance or reduce storage space of the strings in the database. This 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:

```sql
CREATE TABLE strings (
    val VARCHAR CHECK (length(val) <= 10) -- val has a maximum length of 10 characters
);
```
The VARCHAR field allows storage of Unicode characters. Internally, the data is encoded as UTF-8.

**Formatting Strings**

Strings in DuckDB are surrounded by single quote (apostrophe) characters ('):

```sql
SELECT 'Hello world' AS msg;
```

```
msg  varchar
Hello world
```

To include a single quote character in a string, use '':

```sql
SELECT 'Hello ''world''' AS msg;
```

```
msg  varchar
Hello 'world'
```

To include special characters such as newline, use the `chr` character function:

```sql
SELECT 'Hello' || chr(10) || 'world' AS msg;
```

```
msg  varchar
Hello\nworld
```

**Double Quote Characters**

Double quote characters (") are used to denote table and column names. Surrounding their names allows the use of keywords, e.g.:

```sql
CREATE TABLE "table" ("order" BIGINT);
```

While DuckDB occasionally accepts both single quote and double quotes for strings (e.g., both FROM "filename.csv" and FROM 'filename.csv' work), their use is not recommended.

**Functions**

See Character Functions and Pattern Matching.
**Time Types**

The TIME and TIMETZ types specify the hour, minute, second, microsecond of a day.

<table>
<thead>
<tr>
<th>Name</th>
<th>Aliases</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIME</td>
<td>TIME WITHOUT TIME ZONE</td>
<td>time of day (ignores time zone)</td>
</tr>
<tr>
<td>TIMETZ</td>
<td>TIME WITH TIME ZONE</td>
<td>time of day (uses time zone)</td>
</tr>
</tbody>
</table>

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]]).

```sql
SELECT TIME '1992-09-20 11:30:00.123456'; -- 11:30:00.123456
SELECT TIMETZ '1992-09-20 11:30:00.123456+00'; -- 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
```

**Note.** 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 absolute time, usually called *instants*. DuckDB represents instants as the number of microseconds (μs) since 1970-01-01 00:00:00+00.

**Timestamp types**

<table>
<thead>
<tr>
<th>Name</th>
<th>Aliases</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIMESTAMP_NS</td>
<td>TIMESTAMP,DATETIME</td>
<td>timestamp with nanosecond precision (ignores time zone)</td>
</tr>
<tr>
<td>TIMESTAMP_MS</td>
<td></td>
<td>timestamp with millisecond precision (ignores time zone)</td>
</tr>
<tr>
<td>TIMESTAMP_S</td>
<td></td>
<td>timestamp with second precision (ignores time zone)</td>
</tr>
<tr>
<td>TIMESTAMPTZ</td>
<td>TIMESTAMP WITH TIME ZONE</td>
<td>timestamp (uses time zone)</td>
</tr>
</tbody>
</table>

A timestamp specifies a combination of DATE (year, month, day) and a TIME (hour, minute, second, microsecond). Timestamps can be created using the `TIMESTAMP` keyword, where the data must be formatted according to the ISO 8601 format (`YYYY-MM-DD  hh:mm:ss[.zzzzzz][+-TT[tt]]`).

484
Special Values

There are also three special date values that can be used on input:

<table>
<thead>
<tr>
<th>Input String</th>
<th>Valid Types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>epoch</td>
<td>TIMESTAMP, TIMESTAMPTZ</td>
<td>1970-01-01 00:00:00+00 (Unix system time zero)</td>
</tr>
<tr>
<td>infinity</td>
<td>TIMESTAMP, TIMESTAMPTZ</td>
<td>later than all other time stamps</td>
</tr>
<tr>
<td>-infinity</td>
<td>TIMESTAMP, TIMESTAMPTZ</td>
<td>earlier than all other time stamps</td>
</tr>
</tbody>
</table>

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 time stamp value when read.

```
SELECT '-infinity'::TIMESTAMP, 'epoch'::TIMESTAMP, 'infinity'::TIMESTAMP;
```

<table>
<thead>
<tr>
<th>Negative</th>
<th>Epoch</th>
<th>Positive</th>
</tr>
</thead>
<tbody>
<tr>
<td>-infinity</td>
<td>1970-01-01 00:00:00</td>
<td>infinity</td>
</tr>
</tbody>
</table>

Functions

See Timestamp Functions.

Time Zones

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.

Calendars

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 looked from in the duckdb_settings() table function:

SELECT * FROM duckdb_settings() WHERE name = 'TimeZone';
-- America/Los_Angeles
SELECT * FROM duckdb_settings() WHERE name = 'Calendar';
-- gregorian
Time Zone Reference List

An up-to-date version of this list can be pulled from the `pg_timezone_names()` table function:

```sql
SELECT name, abbrev
FROM pg_timezone_names()
ORDER BY name;
```

<table>
<thead>
<tr>
<th>name</th>
<th>abbrev</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACT</td>
<td>ACT</td>
</tr>
<tr>
<td>AET</td>
<td>AET</td>
</tr>
<tr>
<td>AGT</td>
<td>AGT</td>
</tr>
<tr>
<td>ART</td>
<td>ART</td>
</tr>
<tr>
<td>AST</td>
<td>AST</td>
</tr>
<tr>
<td>Africa/Abidjan</td>
<td>Iceland</td>
</tr>
<tr>
<td>Africa/Accra</td>
<td>Iceland</td>
</tr>
<tr>
<td>Africa/Addis_Ababa</td>
<td>EAT</td>
</tr>
<tr>
<td>Africa/Algiers</td>
<td>Africa/Algiers</td>
</tr>
<tr>
<td>Africa/Asmara</td>
<td>EAT</td>
</tr>
<tr>
<td>Africa/Asmera</td>
<td>EAT</td>
</tr>
<tr>
<td>Africa/Bamako</td>
<td>Iceland</td>
</tr>
<tr>
<td>Africa/Bangui</td>
<td>Africa/Bangui</td>
</tr>
<tr>
<td>Africa/Banjul</td>
<td>Iceland</td>
</tr>
<tr>
<td>Africa/Bissau</td>
<td>Africa/Bissau</td>
</tr>
<tr>
<td>Africa/Blantyre</td>
<td>CAT</td>
</tr>
<tr>
<td>Africa/Brazzaville</td>
<td>Africa/Brazzaville</td>
</tr>
<tr>
<td>Africa/Bujumbura</td>
<td>CAT</td>
</tr>
<tr>
<td>Africa/Cairo</td>
<td>ART</td>
</tr>
<tr>
<td>Africa/Casablanca</td>
<td>Africa/Casablanca</td>
</tr>
<tr>
<td>Africa/Ceuta</td>
<td>Africa/Ceuta</td>
</tr>
<tr>
<td>Africa/Conakry</td>
<td>Iceland</td>
</tr>
<tr>
<td>Africa/Dakar</td>
<td>Iceland</td>
</tr>
<tr>
<td>Africa/Dar-es-Salaam</td>
<td>EAT</td>
</tr>
<tr>
<td>Africa/Djibouti</td>
<td>EAT</td>
</tr>
<tr>
<td>Africa/Douala</td>
<td>Africa/Douala</td>
</tr>
<tr>
<td>name</td>
<td>abbrev</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>Africa/El_Aaiun</td>
<td>Africa/El_Aaiun</td>
</tr>
<tr>
<td>Africa/Freetown</td>
<td>Iceland</td>
</tr>
<tr>
<td>Africa/Gaborone</td>
<td>CAT</td>
</tr>
<tr>
<td>Africa/Harare</td>
<td>CAT</td>
</tr>
<tr>
<td>Africa/Johannesburg</td>
<td>Africa/Johannesburg</td>
</tr>
<tr>
<td>Africa/Juba</td>
<td>Africa/Juba</td>
</tr>
<tr>
<td>Africa/Kampala</td>
<td>EAT</td>
</tr>
<tr>
<td>Africa/Khartoum</td>
<td>Africa/Khartoum</td>
</tr>
<tr>
<td>Africa/Kigali</td>
<td>CAT</td>
</tr>
<tr>
<td>Africa/Kinshasa</td>
<td>Africa/Kinshasa</td>
</tr>
<tr>
<td>Africa/Lagos</td>
<td>Africa/Lagos</td>
</tr>
<tr>
<td>Africa/Libreville</td>
<td>Africa/Libreville</td>
</tr>
<tr>
<td>Africa/Lome</td>
<td>Iceland</td>
</tr>
<tr>
<td>Africa/Luanda</td>
<td>Africa/Luanda</td>
</tr>
<tr>
<td>Africa/Lubumbashi</td>
<td>CAT</td>
</tr>
<tr>
<td>Africa/Lusaka</td>
<td>CAT</td>
</tr>
<tr>
<td>Africa/Malabo</td>
<td>Africa/Malabo</td>
</tr>
<tr>
<td>Africa/Maputo</td>
<td>CAT</td>
</tr>
<tr>
<td>Africa/Maseru</td>
<td>Africa/Maseru</td>
</tr>
<tr>
<td>Africa/Mbabane</td>
<td>Africa/Mbabane</td>
</tr>
<tr>
<td>Africa/Mogadishu</td>
<td>EAT</td>
</tr>
<tr>
<td>Africa/Monrovia</td>
<td>Africa/Monrovia</td>
</tr>
<tr>
<td>Africa/Nairobi</td>
<td>EAT</td>
</tr>
<tr>
<td>Africa/Ndjamena</td>
<td>Africa/Ndjamena</td>
</tr>
<tr>
<td>Africa/Niamey</td>
<td>Africa/Niamey</td>
</tr>
<tr>
<td>Africa/Nouakchott</td>
<td>Iceland</td>
</tr>
<tr>
<td>Africa/Ouagadougou</td>
<td>Iceland</td>
</tr>
<tr>
<td>Africa/Porto-Novo</td>
<td>Africa/Porto-Novo</td>
</tr>
<tr>
<td>Africa/Sao_Tome</td>
<td>Africa/Sao_Tome</td>
</tr>
<tr>
<td>Africa/Timbuktu</td>
<td>Iceland</td>
</tr>
<tr>
<td>Africa/Tripoli</td>
<td>Libya</td>
</tr>
<tr>
<td>Africa/Tunis</td>
<td>Africa/Tunis</td>
</tr>
<tr>
<td>name</td>
<td>abbrev</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>Africa/Windhoek</td>
<td>Africa/Windhoek</td>
</tr>
<tr>
<td>America/Adak</td>
<td>America/Adak</td>
</tr>
<tr>
<td>America/Anchorage</td>
<td>AST</td>
</tr>
<tr>
<td>America/Anguilla</td>
<td>PRT</td>
</tr>
<tr>
<td>America/Antigua</td>
<td>PRT</td>
</tr>
<tr>
<td>America/Araguaina</td>
<td>America/Araguaina</td>
</tr>
<tr>
<td>America/Argentina/Buenos_Aires</td>
<td>America/Argentina/Buenos_Aires</td>
</tr>
<tr>
<td>America/Argentina/Catamarca</td>
<td>America/Argentina/Catamarca</td>
</tr>
<tr>
<td>America/Argentina/ComodRivadavia</td>
<td>America/Argentina/ComodRivadavia</td>
</tr>
<tr>
<td>America/Argentina/Cordoba</td>
<td>America/Argentina/Cordoba</td>
</tr>
<tr>
<td>America/Argentina/Jujuy</td>
<td>America/Argentina/Jujuy</td>
</tr>
<tr>
<td>America/Argentina/La_Rioja</td>
<td>America/Argentina/La_Rioja</td>
</tr>
<tr>
<td>America/Argentina/Mendoza</td>
<td>America/Argentina/Mendoza</td>
</tr>
<tr>
<td>America/Argentina/Rio_Gallegos</td>
<td>America/Argentina/Rio_Gallegos</td>
</tr>
<tr>
<td>America/Argentina/Salta</td>
<td>America/Argentina/Salta</td>
</tr>
<tr>
<td>America/Argentina/San_Juan</td>
<td>America/Argentina/San_Juan</td>
</tr>
<tr>
<td>America/Argentina/San_Luis</td>
<td>America/Argentina/San_Luis</td>
</tr>
<tr>
<td>America/Argentina/Tucuman</td>
<td>America/Argentina/Tucuman</td>
</tr>
<tr>
<td>America/Argentina/Ushuaia</td>
<td>America/Argentina/Ushuaia</td>
</tr>
<tr>
<td>America/Aruba</td>
<td>PRT</td>
</tr>
<tr>
<td>America/Asuncion</td>
<td>America/Asuncion</td>
</tr>
<tr>
<td>America/Atikokan</td>
<td>America/Atikokan</td>
</tr>
<tr>
<td>America/Atka</td>
<td>America/Atka</td>
</tr>
<tr>
<td>America/Bahia</td>
<td>America/Bahia</td>
</tr>
<tr>
<td>America/Bahia_Banderas</td>
<td>America/Bahia_Banderas</td>
</tr>
<tr>
<td>America/Barbados</td>
<td>America/Barbados</td>
</tr>
<tr>
<td>America/Belem</td>
<td>America/Belem</td>
</tr>
<tr>
<td>America/Belize</td>
<td>America/Belize</td>
</tr>
<tr>
<td>America/Blanc-Sablon</td>
<td>PRT</td>
</tr>
<tr>
<td>America/Boa_Vista</td>
<td>America/Boa_Vista</td>
</tr>
<tr>
<td>America/Bogota</td>
<td>America/Bogota</td>
</tr>
<tr>
<td>America/Boise</td>
<td>America/Boise</td>
</tr>
<tr>
<td>name</td>
<td>abbrev</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>America/Buenos_Aires</td>
<td>AGT</td>
</tr>
<tr>
<td>America/Cambridge_Bay</td>
<td>America/Cambridge_Bay</td>
</tr>
<tr>
<td>America/Campo_Grande</td>
<td>America/Campo_Grande</td>
</tr>
<tr>
<td>America/Cancun</td>
<td>America/Cancun</td>
</tr>
<tr>
<td>America/Caracas</td>
<td>America/Caracas</td>
</tr>
<tr>
<td>America/Catamarca</td>
<td>America/Catamarca</td>
</tr>
<tr>
<td>America/Cayenne</td>
<td>America/Cayenne</td>
</tr>
<tr>
<td>America/Cayman</td>
<td>America/Cayman</td>
</tr>
<tr>
<td>America/Chicago</td>
<td>CST</td>
</tr>
<tr>
<td>America/Chihuahua</td>
<td>America/Chihuahua</td>
</tr>
<tr>
<td>America/Ciudad_Juarez</td>
<td>America/Ciudad_Juarez</td>
</tr>
<tr>
<td>America/Coral_Harbour</td>
<td>America/Coral_Harbour</td>
</tr>
<tr>
<td>America/Cordoba</td>
<td>America/Cordoba</td>
</tr>
<tr>
<td>America/Costa_Rica</td>
<td>America/Costa_Rica</td>
</tr>
<tr>
<td>America/Creston</td>
<td>PNT</td>
</tr>
<tr>
<td>America/Cuiaba</td>
<td>America/Cuiaba</td>
</tr>
<tr>
<td>America/Curacao</td>
<td>PRT</td>
</tr>
<tr>
<td>America/Danmarkshavn</td>
<td>America/Danmarkshavn</td>
</tr>
<tr>
<td>America/Dawson</td>
<td>America/Dawson</td>
</tr>
<tr>
<td>America/Dawson_Creek</td>
<td>America/Dawson_Creek</td>
</tr>
<tr>
<td>America/Denver</td>
<td>Navajo</td>
</tr>
<tr>
<td>America/Detroit</td>
<td>America/Detroit</td>
</tr>
<tr>
<td>America/Dominica</td>
<td>PRT</td>
</tr>
<tr>
<td>America/Edmonton</td>
<td>America/Edmonton</td>
</tr>
<tr>
<td>America/Eirunepe</td>
<td>America/Eirunepe</td>
</tr>
<tr>
<td>America/El_Salvador</td>
<td>America/El_Salvador</td>
</tr>
<tr>
<td>America/Ensenada</td>
<td>America/Ensenada</td>
</tr>
<tr>
<td>America/Fort_Nelson</td>
<td>America/Fort_Nelson</td>
</tr>
<tr>
<td>America/Fort_Wayne</td>
<td>IET</td>
</tr>
<tr>
<td>America/Fortaleza</td>
<td>America/Fortaleza</td>
</tr>
<tr>
<td>America/Glace_Bay</td>
<td>America/Glace_Bay</td>
</tr>
<tr>
<td>America/Godthab</td>
<td>America/Godthab</td>
</tr>
</tbody>
</table>

490
<table>
<thead>
<tr>
<th>name</th>
<th>abbrev</th>
</tr>
</thead>
<tbody>
<tr>
<td>America/Goose_Bay</td>
<td>America/Goose_Bay</td>
</tr>
<tr>
<td>America/Grand_Turk</td>
<td>America/Grand_Turk</td>
</tr>
<tr>
<td>America/Grenada</td>
<td>PRT</td>
</tr>
<tr>
<td>America/Guadeloupe</td>
<td>PRT</td>
</tr>
<tr>
<td>America/Guatemala</td>
<td>America/Guatemala</td>
</tr>
<tr>
<td>America/Guayaquil</td>
<td>America/Guayaquil</td>
</tr>
<tr>
<td>America/Guyana</td>
<td>America/Guyana</td>
</tr>
<tr>
<td>America/Halifax</td>
<td>America/Halifax</td>
</tr>
<tr>
<td>America/Havana</td>
<td>Cuba</td>
</tr>
<tr>
<td>America/Hermosillo</td>
<td>America/Hermosillo</td>
</tr>
<tr>
<td>America/Indiana/Indianapolis</td>
<td>IET</td>
</tr>
<tr>
<td>America/Indiana/Knox</td>
<td>America/Indiana/Knox</td>
</tr>
<tr>
<td>America/Indiana/Marengo</td>
<td>America/Indiana/Marengo</td>
</tr>
<tr>
<td>America/Indiana/Petersburg</td>
<td>America/Indiana/Petersburg</td>
</tr>
<tr>
<td>America/Indiana/Tell_City</td>
<td>America/Indiana/Tell_City</td>
</tr>
<tr>
<td>America/Indiana/Vevay</td>
<td>America/Indiana/Vevay</td>
</tr>
<tr>
<td>America/Indiana/Vincennes</td>
<td>America/Indiana/Vincennes</td>
</tr>
<tr>
<td>America/Indiana/Winamac</td>
<td>America/Indiana/Winamac</td>
</tr>
<tr>
<td>America/Indianapolis</td>
<td>IET</td>
</tr>
<tr>
<td>America/Inuvik</td>
<td>America/Inuvik</td>
</tr>
<tr>
<td>America/Iqaluit</td>
<td>America/Iqaluit</td>
</tr>
<tr>
<td>America/Jamaica</td>
<td>Jamaica</td>
</tr>
<tr>
<td>America/Jujuy</td>
<td>America/Jujuy</td>
</tr>
<tr>
<td>America/Juneau</td>
<td>America/Juneau</td>
</tr>
<tr>
<td>America/Kentucky/Louisville</td>
<td>America/Kentucky/Louisville</td>
</tr>
<tr>
<td>America/Kentucky/Monticello</td>
<td>America/Kentucky/Monticello</td>
</tr>
<tr>
<td>America/Knox_IN</td>
<td>America/Knox_IN</td>
</tr>
<tr>
<td>America/Kralendijk</td>
<td>PRT</td>
</tr>
<tr>
<td>America/La_Paz</td>
<td>America/La_Paz</td>
</tr>
<tr>
<td>America/Lima</td>
<td>America/Lima</td>
</tr>
<tr>
<td>America/Los_Angeles</td>
<td>PST</td>
</tr>
<tr>
<td>America/Louisville</td>
<td>America/Louisville</td>
</tr>
<tr>
<td>name</td>
<td>abbrev</td>
</tr>
<tr>
<td>------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>America/Lower_Princes</td>
<td>PRT</td>
</tr>
<tr>
<td>America/Maceio</td>
<td>America/Maceio</td>
</tr>
<tr>
<td>America/Managua</td>
<td>America/Managua</td>
</tr>
<tr>
<td>America/Managua</td>
<td>America/Managua</td>
</tr>
<tr>
<td>America/Marigot</td>
<td>PRT</td>
</tr>
<tr>
<td>America/Martinique</td>
<td>America/Martinique</td>
</tr>
<tr>
<td>America/Matamoros</td>
<td>America/Matamoros</td>
</tr>
<tr>
<td>America/Mazatlan</td>
<td>America/Mazatlan</td>
</tr>
<tr>
<td>America/Mendoza</td>
<td>America/Mendoza</td>
</tr>
<tr>
<td>America/Menominee</td>
<td>America/Menominee</td>
</tr>
<tr>
<td>America/Merida</td>
<td>America/Merida</td>
</tr>
<tr>
<td>America/Metlakatla</td>
<td>America/Metlakatla</td>
</tr>
<tr>
<td>America/Mexico_City</td>
<td>America/Mexico_City</td>
</tr>
<tr>
<td>America/Miquelon</td>
<td>America/Miquelon</td>
</tr>
<tr>
<td>America/Moncton</td>
<td>America/Moncton</td>
</tr>
<tr>
<td>America/ Monterrey</td>
<td>America/ Monterrey</td>
</tr>
<tr>
<td>America/Montevideo</td>
<td>America/Montevideo</td>
</tr>
<tr>
<td>America/Montréal</td>
<td>America/Montréal</td>
</tr>
<tr>
<td>America/Montserrat</td>
<td>PRT</td>
</tr>
<tr>
<td>America/Nassau</td>
<td>America/Nassau</td>
</tr>
<tr>
<td>America/New_York</td>
<td>America/New_York</td>
</tr>
<tr>
<td>America/Nipigon</td>
<td>America/Nipigon</td>
</tr>
<tr>
<td>America/Nome</td>
<td>America/Nome</td>
</tr>
<tr>
<td>America/Noronha</td>
<td>America/Noronha</td>
</tr>
<tr>
<td>America/North_Dakota/Beulah</td>
<td>America/North_Dakota/Beulah</td>
</tr>
<tr>
<td>America/North_Dakota/Center</td>
<td>America/North_Dakota/Center</td>
</tr>
<tr>
<td>America/North_Dakota/New_Salem</td>
<td>America/North_Dakota/New_Salem</td>
</tr>
<tr>
<td>America/Nuuk</td>
<td>America/Nuuk</td>
</tr>
<tr>
<td>America/Ojinaga</td>
<td>America/Ojinaga</td>
</tr>
<tr>
<td>America/Panama</td>
<td>America/Panama</td>
</tr>
<tr>
<td>America/Pangnirtung</td>
<td>America/Pangnirtung</td>
</tr>
<tr>
<td>America/Paramaribo</td>
<td>America/Paramaribo</td>
</tr>
<tr>
<td>name</td>
<td>abbrev</td>
</tr>
<tr>
<td>--------------------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>America/Phoenix</td>
<td>PNT</td>
</tr>
<tr>
<td>America/Port-au-Prince</td>
<td>America/Port-au-Prince</td>
</tr>
<tr>
<td>America/Port_of_Spain</td>
<td>PRT</td>
</tr>
<tr>
<td>America/Porto_Acre</td>
<td>America/Porto_Acre</td>
</tr>
<tr>
<td>America/Porto_Velho</td>
<td>America/Porto_Velho</td>
</tr>
<tr>
<td>America/Puerto_Rico</td>
<td>PRT</td>
</tr>
<tr>
<td>America/Punta_Arenas</td>
<td>America/Punta_Arenas</td>
</tr>
<tr>
<td>America/Rainy_River</td>
<td>America/Rainy_River</td>
</tr>
<tr>
<td>America/Rankin_Inlet</td>
<td>America/Rankin_Inlet</td>
</tr>
<tr>
<td>America/Recife</td>
<td>America/Recife</td>
</tr>
<tr>
<td>America/Regina</td>
<td>America/Regina</td>
</tr>
<tr>
<td>America/Resolute</td>
<td>America/Resolute</td>
</tr>
<tr>
<td>America/Rio_Branco</td>
<td>America/Rio_Branco</td>
</tr>
<tr>
<td>America/Rosario</td>
<td>America/Rosario</td>
</tr>
<tr>
<td>America/Santa_Isabel</td>
<td>America/Santa_Isabel</td>
</tr>
<tr>
<td>America/Santarem</td>
<td>America/Santarem</td>
</tr>
<tr>
<td>America/Santiago</td>
<td>America/Santiago</td>
</tr>
<tr>
<td>America/Santo_Domingo</td>
<td>America/Santo_Domingo</td>
</tr>
<tr>
<td>America/Sao_Paulo</td>
<td>BET</td>
</tr>
<tr>
<td>America/Scoresbysund</td>
<td>America/Scoresbysund</td>
</tr>
<tr>
<td>America/Shiprock</td>
<td>Navajo</td>
</tr>
<tr>
<td>America/Sitka</td>
<td>America/Sitka</td>
</tr>
<tr>
<td>America/St_Barthelemy</td>
<td>PRT</td>
</tr>
<tr>
<td>America/St_Johns</td>
<td>CNT</td>
</tr>
<tr>
<td>America/St_Kitts</td>
<td>PRT</td>
</tr>
<tr>
<td>America/St_Lucia</td>
<td>PRT</td>
</tr>
<tr>
<td>America/St_Thomas</td>
<td>PRT</td>
</tr>
<tr>
<td>America/St_Vincent</td>
<td>PRT</td>
</tr>
<tr>
<td>America/Swift_Current</td>
<td>America/Swift_Current</td>
</tr>
<tr>
<td>America/Tegucigalpa</td>
<td>America/Tegucigalpa</td>
</tr>
<tr>
<td>America/Thule</td>
<td>America/Thule</td>
</tr>
<tr>
<td>America/Thunder_Bay</td>
<td>America/Thunder_Bay</td>
</tr>
<tr>
<td>name</td>
<td>abbrev</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>America/Tijuana</td>
<td>America/Tijuana</td>
</tr>
<tr>
<td>America/Toronto</td>
<td>America/Toronto</td>
</tr>
<tr>
<td>America/Tortola</td>
<td>PRT</td>
</tr>
<tr>
<td>America/Vancouver</td>
<td>America/Vancouver</td>
</tr>
<tr>
<td>America/Virgin</td>
<td>PRT</td>
</tr>
<tr>
<td>America/Whitehorse</td>
<td>America/Whitehorse</td>
</tr>
<tr>
<td>America/Winnipeg</td>
<td>America/Winnipeg</td>
</tr>
<tr>
<td>America/Yakutat</td>
<td>America/Yakutat</td>
</tr>
<tr>
<td>America/Yellowknife</td>
<td>America/Yellowknife</td>
</tr>
<tr>
<td>Antarctica/Casey</td>
<td>Antarctica/Casey</td>
</tr>
<tr>
<td>Antarctica/Davis</td>
<td>Antarctica/Davis</td>
</tr>
<tr>
<td>Antarctica/DumontDUrville</td>
<td>Antarctica/DumontDUrville</td>
</tr>
<tr>
<td>Antarctica/Macquarie</td>
<td>Antarctica/Macquarie</td>
</tr>
<tr>
<td>Antarctica/Mawson</td>
<td>Antarctica/Mawson</td>
</tr>
<tr>
<td>Antarctica/McMurdo</td>
<td>NZ</td>
</tr>
<tr>
<td>Antarctica/Palmer</td>
<td>Antarctica/Palmer</td>
</tr>
<tr>
<td>Antarctica/Rothera</td>
<td>Antarctica/Rothera</td>
</tr>
<tr>
<td>Antarctica/South_Pole</td>
<td>NZ</td>
</tr>
<tr>
<td>Antarctica/Syowa</td>
<td>Antarctica/Syowa</td>
</tr>
<tr>
<td>Antarctica/Troll</td>
<td>Antarctica/Troll</td>
</tr>
<tr>
<td>Antarctica/Vostok</td>
<td>Antarctica/Vostok</td>
</tr>
<tr>
<td>Arctic/Longyearbyen</td>
<td>Arctic/Longyearbyen</td>
</tr>
<tr>
<td>Asia/Aden</td>
<td>Asia/Aden</td>
</tr>
<tr>
<td>Asia/Almaty</td>
<td>Asia/Almaty</td>
</tr>
<tr>
<td>Asia/Amman</td>
<td>Asia/Amman</td>
</tr>
<tr>
<td>Asia/Anadyr</td>
<td>Asia/Anadyr</td>
</tr>
<tr>
<td>Asia/Aqtau</td>
<td>Asia/Aqtau</td>
</tr>
<tr>
<td>Asia/Aqtobe</td>
<td>Asia/Aqtobe</td>
</tr>
<tr>
<td>Asia/Ashgabat</td>
<td>Asia/Ashgabat</td>
</tr>
<tr>
<td>Asia/Ashkhabad</td>
<td>Asia/Ashkhabad</td>
</tr>
<tr>
<td>Asia/Atyrau</td>
<td>Asia/Atyrau</td>
</tr>
<tr>
<td>Asia/Baghdad</td>
<td>Asia/Baghdad</td>
</tr>
<tr>
<td>name</td>
<td>abbrev</td>
</tr>
<tr>
<td>-----------------------</td>
<td>------------</td>
</tr>
<tr>
<td>Asia/Bahrain</td>
<td>Asia/Bahrain</td>
</tr>
<tr>
<td>Asia/Baku</td>
<td>Asia/Baku</td>
</tr>
<tr>
<td>Asia/Bangkok</td>
<td>Asia/Bangkok</td>
</tr>
<tr>
<td>Asia/Barnaul</td>
<td>Asia/Barnaul</td>
</tr>
<tr>
<td>Asia/Beirut</td>
<td>Asia/Beirut</td>
</tr>
<tr>
<td>Asia/Bishkek</td>
<td>Asia/Bishkek</td>
</tr>
<tr>
<td>Asia/Brunei</td>
<td>Asia/Brunei</td>
</tr>
<tr>
<td>Asia/Calcutta</td>
<td>IST</td>
</tr>
<tr>
<td>Asia/Chita</td>
<td>Asia/Chita</td>
</tr>
<tr>
<td>Asia/Choibalsan</td>
<td>Asia/Choibalsan</td>
</tr>
<tr>
<td>Asia/Chongqing</td>
<td>CTT</td>
</tr>
<tr>
<td>Asia/Chungking</td>
<td>CTT</td>
</tr>
<tr>
<td>Asia/Colombo</td>
<td>Asia/Colombo</td>
</tr>
<tr>
<td>Asia/Dacca</td>
<td>BST</td>
</tr>
<tr>
<td>Asia/Damascus</td>
<td>Asia/Damascus</td>
</tr>
<tr>
<td>Asia/Dhaka</td>
<td>BST</td>
</tr>
<tr>
<td>Asia/Dili</td>
<td>Asia/Dili</td>
</tr>
<tr>
<td>Asia/Dubai</td>
<td>Asia/Dubai</td>
</tr>
<tr>
<td>Asia/Dushanbe</td>
<td>Asia/Dushanbe</td>
</tr>
<tr>
<td>Asia/Famagusta</td>
<td>Asia/Famagusta</td>
</tr>
<tr>
<td>Asia/Gaza</td>
<td>Asia/Gaza</td>
</tr>
<tr>
<td>Asia/Harbin</td>
<td>CTT</td>
</tr>
<tr>
<td>Asia/Hebron</td>
<td>Asia/Hebron</td>
</tr>
<tr>
<td>Asia/Ho_Chi_Minh</td>
<td>VST</td>
</tr>
<tr>
<td>Asia/Hong_Kong</td>
<td>Hongkong</td>
</tr>
<tr>
<td>Asia/Hovd</td>
<td>Asia/Hovd</td>
</tr>
<tr>
<td>Asia/Irkutsk</td>
<td>Asia/Irkutsk</td>
</tr>
<tr>
<td>Asia/Istanbul</td>
<td>Turkey</td>
</tr>
<tr>
<td>Asia/Jakarta</td>
<td>Asia/Jakarta</td>
</tr>
<tr>
<td>Asia/Jayapura</td>
<td>Asia/Jayapura</td>
</tr>
<tr>
<td>Asia/Jerusalem</td>
<td>Israel</td>
</tr>
<tr>
<td>Asia/Kabul</td>
<td>Asia/Kabul</td>
</tr>
<tr>
<td>name</td>
<td>abbrev</td>
</tr>
<tr>
<td>-----------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>Asia/Kamchatka</td>
<td>Asia/Kamchatka</td>
</tr>
<tr>
<td>Asia/Karachi</td>
<td>PLT</td>
</tr>
<tr>
<td>Asia/Kashgar</td>
<td>Asia/Kashgar</td>
</tr>
<tr>
<td>Asia/Kathmandu</td>
<td>Asia/Kathmandu</td>
</tr>
<tr>
<td>Asia/Katmandu</td>
<td>Asia/Katmandu</td>
</tr>
<tr>
<td>Asia/Khandyga</td>
<td>Asia/Khandyga</td>
</tr>
<tr>
<td>Asia/Kolkata</td>
<td>IST</td>
</tr>
<tr>
<td>Asia/Krasnoyarsk</td>
<td>Asia/Krasnoyarsk</td>
</tr>
<tr>
<td>Asia/Kuala_Lumpur</td>
<td>Singapore</td>
</tr>
<tr>
<td>Asia/Kuching</td>
<td>Asia/Kuching</td>
</tr>
<tr>
<td>Asia/Kuwait</td>
<td>Asia/Kuwait</td>
</tr>
<tr>
<td>Asia/Macao</td>
<td>Asia/Macao</td>
</tr>
<tr>
<td>Asia/Macau</td>
<td>Asia/Macau</td>
</tr>
<tr>
<td>Asia/Magadan</td>
<td>Asia/Magadan</td>
</tr>
<tr>
<td>Asia/Makassar</td>
<td>Asia/Makassar</td>
</tr>
<tr>
<td>Asia/Manila</td>
<td>Asia/Manila</td>
</tr>
<tr>
<td>Asia/Muscat</td>
<td>Asia/Muscat</td>
</tr>
<tr>
<td>Asia/Nicosia</td>
<td>Asia/Nicosia</td>
</tr>
<tr>
<td>Asia/Novokuenzetsk</td>
<td>Asia/Novokuenzetsk</td>
</tr>
<tr>
<td>Asia/Novosibirsk</td>
<td>Asia/Novosibirsk</td>
</tr>
<tr>
<td>Asia/Omsk</td>
<td>Asia/Omsk</td>
</tr>
<tr>
<td>Asia/Oral</td>
<td>Asia/Oral</td>
</tr>
<tr>
<td>Asia/Phnom_Penh</td>
<td>Asia/Phnom_Penh</td>
</tr>
<tr>
<td>Asia/Pontianak</td>
<td>Asia/Pontianak</td>
</tr>
<tr>
<td>Asia/Pyongyang</td>
<td>Asia/Pyongyang</td>
</tr>
<tr>
<td>Asia/Qatar</td>
<td>Asia/Qatar</td>
</tr>
<tr>
<td>Asia/Qostanay</td>
<td>Asia/Qostanay</td>
</tr>
<tr>
<td>Asia/Qyzylorda</td>
<td>Asia/Qyzylorda</td>
</tr>
<tr>
<td>Asia/Rangoon</td>
<td>Asia/Rangoon</td>
</tr>
<tr>
<td>Asia/Riyadh</td>
<td>Asia/Riyadh</td>
</tr>
<tr>
<td>Asia/Saigon</td>
<td>VST</td>
</tr>
<tr>
<td>Asia/Sakhalin</td>
<td>Asia/Sakhalin</td>
</tr>
<tr>
<td>name</td>
<td>abbrev</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>Asia/Samarkand</td>
<td>Asia/Samarkand</td>
</tr>
<tr>
<td>Asia/Seoul</td>
<td>ROK</td>
</tr>
<tr>
<td>Asia/Shanghai</td>
<td>CTT</td>
</tr>
<tr>
<td>Asia/Singapore</td>
<td>Singapore</td>
</tr>
<tr>
<td>Asia/Srednekolymsk</td>
<td>Asia/Srednekolymsk</td>
</tr>
<tr>
<td>Asia/Taipei</td>
<td>ROC</td>
</tr>
<tr>
<td>Asia/Tashkent</td>
<td>Asia/Tashkent</td>
</tr>
<tr>
<td>Asia/Tbilisi</td>
<td>Asia/Tbilisi</td>
</tr>
<tr>
<td>Asia/Tehran</td>
<td>Iran</td>
</tr>
<tr>
<td>Asia/Tel_Aviv</td>
<td>Israel</td>
</tr>
<tr>
<td>Asia/Thimbu</td>
<td>Asia/Thimbu</td>
</tr>
<tr>
<td>Asia/Thimphu</td>
<td>Asia/Thimphu</td>
</tr>
<tr>
<td>Asia/Tokyo</td>
<td>JST</td>
</tr>
<tr>
<td>Asia/Tomsk</td>
<td>Asia/Tomsk</td>
</tr>
<tr>
<td>Asia/Ujong_Pandang</td>
<td>Asia/Ujong_Pandang</td>
</tr>
<tr>
<td>Asia/Ulaanbaatar</td>
<td>Asia/Ulaanbaatar</td>
</tr>
<tr>
<td>Asia/Ulan_Bator</td>
<td>Asia/Ulan_Bator</td>
</tr>
<tr>
<td>Asia/Urumqi</td>
<td>Asia/Urumqi</td>
</tr>
<tr>
<td>Asia/Ust-Nera</td>
<td>Asia/Ust-Nera</td>
</tr>
<tr>
<td>Asia/Vientiane</td>
<td>Asia/Vientiane</td>
</tr>
<tr>
<td>Asia/Vladivostok</td>
<td>Asia/Vladivostok</td>
</tr>
<tr>
<td>Asia/Yakutsk</td>
<td>Asia/Yakutsk</td>
</tr>
<tr>
<td>Asia/Yangon</td>
<td>Asia/Yangon</td>
</tr>
<tr>
<td>Asia/Yekaterinburg</td>
<td>Asia/Yekaterinburg</td>
</tr>
<tr>
<td>Asia/Yerevan</td>
<td>NET</td>
</tr>
<tr>
<td>Atlantic/Azores</td>
<td>Atlantic/Azores</td>
</tr>
<tr>
<td>Atlantic/Bermuda</td>
<td>Atlantic/Bermuda</td>
</tr>
<tr>
<td>Atlantic/Canary</td>
<td>Atlantic/Canary</td>
</tr>
<tr>
<td>Atlantic/Cape_Verde</td>
<td>Atlantic/Cape_Verde</td>
</tr>
<tr>
<td>Atlantic/Faero</td>
<td>Atlantic/Faero</td>
</tr>
<tr>
<td>Atlantic/Faroe</td>
<td>Atlantic/Faroe</td>
</tr>
<tr>
<td>Atlantic/Jan_Mayen</td>
<td>Atlantic/Jan_Mayen</td>
</tr>
<tr>
<td>name</td>
<td>abbrev</td>
</tr>
<tr>
<td>---------------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>Atlantic/Madeira</td>
<td>Atlantic/Madeira</td>
</tr>
<tr>
<td>Atlantic/Reykjavik</td>
<td>Iceland</td>
</tr>
<tr>
<td>Atlantic/South_Georgia</td>
<td>Atlantic/South_Georgia</td>
</tr>
<tr>
<td>Atlantic/St_Helena</td>
<td>Iceland</td>
</tr>
<tr>
<td>Atlantic/Stanley</td>
<td>Atlantic/Stanley</td>
</tr>
<tr>
<td>Australia/ACT</td>
<td>AET</td>
</tr>
<tr>
<td>Australia/Adelaide</td>
<td>Australia/Adelaide</td>
</tr>
<tr>
<td>Australia/Brisbane</td>
<td>Australia/Brisbane</td>
</tr>
<tr>
<td>Australia/Broken_Hill</td>
<td>Australia/Broken_Hill</td>
</tr>
<tr>
<td>Australia/Canberra</td>
<td>AET</td>
</tr>
<tr>
<td>Australia/Currie</td>
<td>Australia/Currie</td>
</tr>
<tr>
<td>Australia/Darwin</td>
<td>ACT</td>
</tr>
<tr>
<td>Australia/Eucla</td>
<td>Australia/Eucla</td>
</tr>
<tr>
<td>Australia/Hobart</td>
<td>Australia/Hobart</td>
</tr>
<tr>
<td>Australia/LHI</td>
<td>Australia/LHI</td>
</tr>
<tr>
<td>Australia/Lindeman</td>
<td>Australia/Lindeman</td>
</tr>
<tr>
<td>Australia/Lord_Howe</td>
<td>Australia/Lord_Howe</td>
</tr>
<tr>
<td>Australia/Melbourne</td>
<td>Australia/Melbourne</td>
</tr>
<tr>
<td>Australia/NSW</td>
<td>AET</td>
</tr>
<tr>
<td>Australia/North</td>
<td>ACT</td>
</tr>
<tr>
<td>Australia/Perth</td>
<td>Australia/Perth</td>
</tr>
<tr>
<td>Australia/Queensland</td>
<td>Australia/Queensland</td>
</tr>
<tr>
<td>Australia/South</td>
<td>Australia/South</td>
</tr>
<tr>
<td>Australia/Sydney</td>
<td>AET</td>
</tr>
<tr>
<td>Australia/Tasmania</td>
<td>Australia/Tasmania</td>
</tr>
<tr>
<td>Australia/Victoria</td>
<td>Australia/Victoria</td>
</tr>
<tr>
<td>Australia/West</td>
<td>Australia/West</td>
</tr>
<tr>
<td>Australia/Yancowinna</td>
<td>Australia/Yancowinna</td>
</tr>
<tr>
<td>BET</td>
<td>BET</td>
</tr>
<tr>
<td>BST</td>
<td>BST</td>
</tr>
<tr>
<td>Brazil/Acre</td>
<td>Brazil/Acre</td>
</tr>
<tr>
<td>Brazil/DeNoronha</td>
<td>Brazil/DeNoronha</td>
</tr>
<tr>
<td>name</td>
<td>abbrev</td>
</tr>
<tr>
<td>------------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>Brazil/East</td>
<td>BET</td>
</tr>
<tr>
<td>Brazil/West</td>
<td>Brazil/West</td>
</tr>
<tr>
<td>CAT</td>
<td>CAT</td>
</tr>
<tr>
<td>CET</td>
<td>CET</td>
</tr>
<tr>
<td>CNT</td>
<td>CNT</td>
</tr>
<tr>
<td>CST</td>
<td>CST</td>
</tr>
<tr>
<td>CST6CDT</td>
<td>CST6CDT</td>
</tr>
<tr>
<td>CTT</td>
<td>CTT</td>
</tr>
<tr>
<td>Canada/Atlantic</td>
<td>Canada/Atlantic</td>
</tr>
<tr>
<td>Canada/Central</td>
<td>Canada/Central</td>
</tr>
<tr>
<td>Canada/East-Saskatchewan</td>
<td>Canada/East-Saskatchewan</td>
</tr>
<tr>
<td>Canada/Eastern</td>
<td>Canada/Eastern</td>
</tr>
<tr>
<td>Canada/Mountain</td>
<td>Canada/Mountain</td>
</tr>
<tr>
<td>Canada/Newfoundland</td>
<td>CNT</td>
</tr>
<tr>
<td>Canada/Pacific</td>
<td>Canada/Pacific</td>
</tr>
<tr>
<td>Canada/Saskatchewan</td>
<td>Canada/Saskatchewan</td>
</tr>
<tr>
<td>Canada/Yukon</td>
<td>Canada/Yukon</td>
</tr>
<tr>
<td>Chile/Continental</td>
<td>Chile/Continental</td>
</tr>
<tr>
<td>Chile/EasterIsland</td>
<td>Chile/EasterIsland</td>
</tr>
<tr>
<td>Cuba</td>
<td>Cuba</td>
</tr>
<tr>
<td>EAT</td>
<td>EAT</td>
</tr>
<tr>
<td>ECT</td>
<td>ECT</td>
</tr>
<tr>
<td>EET</td>
<td>EET</td>
</tr>
<tr>
<td>EST</td>
<td>EST</td>
</tr>
<tr>
<td>EST5EDT</td>
<td>EST5EDT</td>
</tr>
<tr>
<td>Egypt</td>
<td>ART</td>
</tr>
<tr>
<td>Eire</td>
<td>Eire</td>
</tr>
<tr>
<td>Etc/GMT</td>
<td>GMT</td>
</tr>
<tr>
<td>Etc/GMT+0</td>
<td>GMT</td>
</tr>
<tr>
<td>Etc/GMT+1</td>
<td>Etc/GMT+1</td>
</tr>
<tr>
<td>Etc/GMT+10</td>
<td>Etc/GMT+10</td>
</tr>
<tr>
<td>Etc/GMT+11</td>
<td>Etc/GMT+11</td>
</tr>
<tr>
<td>name</td>
<td>abbrev</td>
</tr>
<tr>
<td>--------------------</td>
<td>------------</td>
</tr>
<tr>
<td>Etc/GMT+12</td>
<td>Etc/GMT+12</td>
</tr>
<tr>
<td>Etc/GMT+2</td>
<td>Etc/GMT+2</td>
</tr>
<tr>
<td>Etc/GMT+3</td>
<td>Etc/GMT+3</td>
</tr>
<tr>
<td>Etc/GMT+4</td>
<td>Etc/GMT+4</td>
</tr>
<tr>
<td>Etc/GMT+5</td>
<td>Etc/GMT+5</td>
</tr>
<tr>
<td>Etc/GMT+6</td>
<td>Etc/GMT+6</td>
</tr>
<tr>
<td>Etc/GMT+7</td>
<td>Etc/GMT+7</td>
</tr>
<tr>
<td>Etc/GMT+8</td>
<td>Etc/GMT+8</td>
</tr>
<tr>
<td>Etc/GMT+9</td>
<td>Etc/GMT+9</td>
</tr>
<tr>
<td>Etc/GMT-0</td>
<td>GMT</td>
</tr>
<tr>
<td>Etc/GMT-1</td>
<td>Etc/GMT-1</td>
</tr>
<tr>
<td>Etc/GMT-10</td>
<td>Etc/GMT-10</td>
</tr>
<tr>
<td>Etc/GMT-11</td>
<td>Etc/GMT-11</td>
</tr>
<tr>
<td>Etc/GMT-12</td>
<td>Etc/GMT-12</td>
</tr>
<tr>
<td>Etc/GMT-13</td>
<td>Etc/GMT-13</td>
</tr>
<tr>
<td>Etc/GMT-14</td>
<td>Etc/GMT-14</td>
</tr>
<tr>
<td>Etc/GMT-2</td>
<td>Etc/GMT-2</td>
</tr>
<tr>
<td>Etc/GMT-3</td>
<td>Etc/GMT-3</td>
</tr>
<tr>
<td>Etc/GMT-4</td>
<td>Etc/GMT-4</td>
</tr>
<tr>
<td>Etc/GMT-5</td>
<td>Etc/GMT-5</td>
</tr>
<tr>
<td>Etc/GMT-6</td>
<td>Etc/GMT-6</td>
</tr>
<tr>
<td>Etc/GMT-7</td>
<td>Etc/GMT-7</td>
</tr>
<tr>
<td>Etc/GMT-8</td>
<td>Etc/GMT-8</td>
</tr>
<tr>
<td>Etc/GMT-9</td>
<td>Etc/GMT-9</td>
</tr>
<tr>
<td>Etc/GMT0</td>
<td>GMT</td>
</tr>
<tr>
<td>Etc/Greenwich</td>
<td>GMT</td>
</tr>
<tr>
<td>Etc/UCT</td>
<td>UCT</td>
</tr>
<tr>
<td>Etc/UTC</td>
<td>UCT</td>
</tr>
<tr>
<td>Etc/Universal</td>
<td>UCT</td>
</tr>
<tr>
<td>Etc/Zulu</td>
<td>UCT</td>
</tr>
<tr>
<td>Europe/Amsterdam</td>
<td>Europe/Amsterdam</td>
</tr>
<tr>
<td>Europe/Andorra</td>
<td>Europe/Andorra</td>
</tr>
<tr>
<td>name</td>
<td>abbrev</td>
</tr>
<tr>
<td>---------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Europe/Astrakhan</td>
<td>Europe/Astrakhan</td>
</tr>
<tr>
<td>Europe/Athens</td>
<td>Europe/Athens</td>
</tr>
<tr>
<td>Europe/Belfast</td>
<td>GB</td>
</tr>
<tr>
<td>Europe/Belgrade</td>
<td>Europe/Belgrade</td>
</tr>
<tr>
<td>Europe/Berlin</td>
<td>Europe/Berlin</td>
</tr>
<tr>
<td>Europe/Bratislava</td>
<td>Europe/Bratislava</td>
</tr>
<tr>
<td>Europe/Brussels</td>
<td>Europe/Brussels</td>
</tr>
<tr>
<td>Europe/Bucharest</td>
<td>Europe/Bucharest</td>
</tr>
<tr>
<td>Europe/Budapest</td>
<td>Europe/Budapest</td>
</tr>
<tr>
<td>Europe/Busingen</td>
<td>Europe/Busingen</td>
</tr>
<tr>
<td>Europe/Chisinau</td>
<td>Europe/Chisinau</td>
</tr>
<tr>
<td>Europe/Copenhagen</td>
<td>Europe/Copenhagen</td>
</tr>
<tr>
<td>Europe/Dublin</td>
<td>Eire</td>
</tr>
<tr>
<td>Europe/Gibraltar</td>
<td>Europe/Gibraltar</td>
</tr>
<tr>
<td>Europe/Guernsey</td>
<td>GB</td>
</tr>
<tr>
<td>Europe/Helsinki</td>
<td>Europe/Helsinki</td>
</tr>
<tr>
<td>Europe/Isle_of_Man</td>
<td>GB</td>
</tr>
<tr>
<td>Europe/Istanbul</td>
<td>Turkey</td>
</tr>
<tr>
<td>Europe/Jersey</td>
<td>GB</td>
</tr>
<tr>
<td>Europe/Kaliningrad</td>
<td>Europe/Kaliningrad</td>
</tr>
<tr>
<td>Europe/Kiev</td>
<td>Europe/Kiev</td>
</tr>
<tr>
<td>Europe/Kirov</td>
<td>Europe/Kirov</td>
</tr>
<tr>
<td>Europe/Kyiv</td>
<td>Europe/Kyiv</td>
</tr>
<tr>
<td>Europe/Lisbon</td>
<td>Portugal</td>
</tr>
<tr>
<td>Europe/Ljublana</td>
<td>Europe/Ljublana</td>
</tr>
<tr>
<td>Europe/London</td>
<td>GB</td>
</tr>
<tr>
<td>Europe/Luxembourg</td>
<td>Europe/Luxembourg</td>
</tr>
<tr>
<td>Europe/Madrid</td>
<td>Europe/Madrid</td>
</tr>
<tr>
<td>Europe/Malta</td>
<td>Europe/Malta</td>
</tr>
<tr>
<td>Europe/Mariehamn</td>
<td>Europe/Mariehamn</td>
</tr>
<tr>
<td>Europe/Minsk</td>
<td>Europe/Minsk</td>
</tr>
<tr>
<td>Europe/Monaco</td>
<td>ECT</td>
</tr>
<tr>
<td>name</td>
<td>abbrev</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>Europe/Moscow</td>
<td>W-SU</td>
</tr>
<tr>
<td>Europe/Nicosia</td>
<td>Europe/Nicosia</td>
</tr>
<tr>
<td>Europe/Oslo</td>
<td>Europe/Oslo</td>
</tr>
<tr>
<td>Europe/Paris</td>
<td>ECT</td>
</tr>
<tr>
<td>Europe/Podgorica</td>
<td>Europe/Podgorica</td>
</tr>
<tr>
<td>Europe/Prague</td>
<td>Europe/Prague</td>
</tr>
<tr>
<td>Europe/Riga</td>
<td>Europe/Riga</td>
</tr>
<tr>
<td>Europe/Rome</td>
<td>Europe/Rome</td>
</tr>
<tr>
<td>Europe/Samara</td>
<td>Europe/Samara</td>
</tr>
<tr>
<td>Europe/San_Marino</td>
<td>Europe/San_Marino</td>
</tr>
<tr>
<td>Europe/Sarajevo</td>
<td>Europe/Sarajevo</td>
</tr>
<tr>
<td>Europe/Saratov</td>
<td>Europe/Saratov</td>
</tr>
<tr>
<td>Europe/Simferopol</td>
<td>Europe/Simferopol</td>
</tr>
<tr>
<td>Europe/Skopje</td>
<td>Europe/Skopje</td>
</tr>
<tr>
<td>Europe/Sofia</td>
<td>Europe/Sofia</td>
</tr>
<tr>
<td>Europe/Stockholm</td>
<td>Europe/Stockholm</td>
</tr>
<tr>
<td>Europe/Tallinn</td>
<td>Europe/Tallinn</td>
</tr>
<tr>
<td>Europe/Tirane</td>
<td>Europe/Tirane</td>
</tr>
<tr>
<td>Europe/Tiraspol</td>
<td>Europe/Tiraspol</td>
</tr>
<tr>
<td>Europe/Ulyanovsk</td>
<td>Europe/Ulyanovsk</td>
</tr>
<tr>
<td>Europe/Uzhgorod</td>
<td>Europe/Uzhgorod</td>
</tr>
<tr>
<td>Europe/Vaduz</td>
<td>Europe/Vaduz</td>
</tr>
<tr>
<td>Europe/Vatican</td>
<td>Europe/Vatican</td>
</tr>
<tr>
<td>Europe/Vienna</td>
<td>Europe/Vienna</td>
</tr>
<tr>
<td>Europe/Vilnius</td>
<td>Europe/Vilnius</td>
</tr>
<tr>
<td>Europe/Volgograd</td>
<td>Europe/Volgograd</td>
</tr>
<tr>
<td>Europe/Warsaw</td>
<td>Poland</td>
</tr>
<tr>
<td>Europe/Zagreb</td>
<td>Europe/Zagreb</td>
</tr>
<tr>
<td>Europe/Zaporozhye</td>
<td>Europe/Zaporozhye</td>
</tr>
<tr>
<td>Europe/Zurich</td>
<td>Europe/Zurich</td>
</tr>
<tr>
<td>Factory</td>
<td>Factory</td>
</tr>
<tr>
<td>GB</td>
<td>GB</td>
</tr>
<tr>
<td>name</td>
<td>abbrev</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>GB-Eire</td>
<td>GB</td>
</tr>
<tr>
<td>GMT</td>
<td>GMT</td>
</tr>
<tr>
<td>GMT+0</td>
<td>GMT</td>
</tr>
<tr>
<td>GMT-0</td>
<td>GMT</td>
</tr>
<tr>
<td>GMT0</td>
<td>GMT</td>
</tr>
<tr>
<td>Greenwich</td>
<td>GMT</td>
</tr>
<tr>
<td>HST</td>
<td>HST</td>
</tr>
<tr>
<td>Hongkong</td>
<td>Hongkong</td>
</tr>
<tr>
<td>IET</td>
<td>IET</td>
</tr>
<tr>
<td>IST</td>
<td>IST</td>
</tr>
<tr>
<td>Iceland</td>
<td>Iceland</td>
</tr>
<tr>
<td>Indian/Antananarivo</td>
<td>EAT</td>
</tr>
<tr>
<td>Indian/Chagos</td>
<td>Indian/Chagos</td>
</tr>
<tr>
<td>Indian/Christmas</td>
<td>Indian/Christmas</td>
</tr>
<tr>
<td>Indian/Cocos</td>
<td>Indian/Cocos</td>
</tr>
<tr>
<td>Indian/Comoro</td>
<td>EAT</td>
</tr>
<tr>
<td>Indian/Kerguelen</td>
<td>Indian/Kerguelen</td>
</tr>
<tr>
<td>Indian/Mahe</td>
<td>Indian/Mahe</td>
</tr>
<tr>
<td>Indian/Maldives</td>
<td>Indian/Maldives</td>
</tr>
<tr>
<td>Indian/Mauritius</td>
<td>Indian/Mauritius</td>
</tr>
<tr>
<td>Indian/Mayotte</td>
<td>EAT</td>
</tr>
<tr>
<td>Indian/Reunion</td>
<td>Indian/Reunion</td>
</tr>
<tr>
<td>Iran</td>
<td>Iran</td>
</tr>
<tr>
<td>Israel</td>
<td>Israel</td>
</tr>
<tr>
<td>JST</td>
<td>JST</td>
</tr>
<tr>
<td>Jamaica</td>
<td>Jamaica</td>
</tr>
<tr>
<td>Japan</td>
<td>JST</td>
</tr>
<tr>
<td>Kwajalein</td>
<td>Kwajalein</td>
</tr>
<tr>
<td>Libya</td>
<td>Libya</td>
</tr>
<tr>
<td>MET</td>
<td>MET</td>
</tr>
<tr>
<td>MIT</td>
<td>MIT</td>
</tr>
<tr>
<td>MST</td>
<td>MST</td>
</tr>
<tr>
<td>name</td>
<td>abbrev</td>
</tr>
<tr>
<td>---------------------</td>
<td>------------</td>
</tr>
<tr>
<td>MST7MDT</td>
<td>MST7MDT</td>
</tr>
<tr>
<td>Mexico/BajaNorte</td>
<td>Mexico/BajaNorte</td>
</tr>
<tr>
<td>Mexico/BajaSur</td>
<td>Mexico/BajaSur</td>
</tr>
<tr>
<td>Mexico/General</td>
<td>Mexico/General</td>
</tr>
<tr>
<td>NET</td>
<td>NET</td>
</tr>
<tr>
<td>NST</td>
<td>NZ</td>
</tr>
<tr>
<td>NZ</td>
<td>NZ</td>
</tr>
<tr>
<td>NZ-CHAT</td>
<td>NZ-CHAT</td>
</tr>
<tr>
<td>Navajo</td>
<td>Navajo</td>
</tr>
<tr>
<td>PLT</td>
<td>PLT</td>
</tr>
<tr>
<td>PNT</td>
<td>PNT</td>
</tr>
<tr>
<td>PRC</td>
<td>CTT</td>
</tr>
<tr>
<td>PRT</td>
<td>PRT</td>
</tr>
<tr>
<td>PST</td>
<td>PST</td>
</tr>
<tr>
<td>PST8PDT</td>
<td>PST8PDT</td>
</tr>
<tr>
<td>Pacific/Apia</td>
<td>MIT</td>
</tr>
<tr>
<td>Pacific/Auckland</td>
<td>NZ</td>
</tr>
<tr>
<td>Pacific/Bougainville</td>
<td>Pacific/Bougainville</td>
</tr>
<tr>
<td>Pacific/Chatham</td>
<td>NZ-CHAT</td>
</tr>
<tr>
<td>Pacific/Chuuk</td>
<td>Pacific/Chuuk</td>
</tr>
<tr>
<td>Pacific/Easter</td>
<td>Pacific/Easter</td>
</tr>
<tr>
<td>Pacific/Efate</td>
<td>Pacific/Efate</td>
</tr>
<tr>
<td>Pacific/Enderbury</td>
<td>Pacific/Enderbury</td>
</tr>
<tr>
<td>Pacific/Fakaofo</td>
<td>Pacific/Fakaofo</td>
</tr>
<tr>
<td>Pacific/Fiji</td>
<td>Pacific/Fiji</td>
</tr>
<tr>
<td>Pacific/Funafuti</td>
<td>Pacific/Funafuti</td>
</tr>
<tr>
<td>Pacific/Galapagos</td>
<td>Pacific/Galapagos</td>
</tr>
<tr>
<td>Pacific/Gambier</td>
<td>Pacific/Gambier</td>
</tr>
<tr>
<td>Pacific/Guadalcanal</td>
<td>SST</td>
</tr>
<tr>
<td>Pacific/Guam</td>
<td>Pacific/Guam</td>
</tr>
<tr>
<td>Pacific/Honolulu</td>
<td>Pacific/Honolulu</td>
</tr>
<tr>
<td>Pacific/Johnston</td>
<td>Pacific/Johnston</td>
</tr>
<tr>
<td>name</td>
<td>abbrev</td>
</tr>
<tr>
<td>---------------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>Pacific/Kanton</td>
<td>Pacific/Kanton</td>
</tr>
<tr>
<td>Pacific/Kirimitati</td>
<td>Pacific/Kirimitati</td>
</tr>
<tr>
<td>Pacific/Kosrae</td>
<td>Pacific/Kosrae</td>
</tr>
<tr>
<td>Pacific/Kwajalein</td>
<td>Kwajalein</td>
</tr>
<tr>
<td>Pacific/Majuro</td>
<td>Pacific/Majuro</td>
</tr>
<tr>
<td>Pacific/Marquesas</td>
<td>Pacific/Marquesas</td>
</tr>
<tr>
<td>Pacific/Midway</td>
<td>Pacific/Midway</td>
</tr>
<tr>
<td>Pacific/Nauru</td>
<td>Pacific/Nauru</td>
</tr>
<tr>
<td>Pacific/Niue</td>
<td>Pacific/Niue</td>
</tr>
<tr>
<td>Pacific/Norfolk</td>
<td>Pacific/Norfolk</td>
</tr>
<tr>
<td>Pacific/Noumea</td>
<td>Pacific/Noumea</td>
</tr>
<tr>
<td>Pacific/Pago_Pago</td>
<td>Pacific/Pago_Pago</td>
</tr>
<tr>
<td>Pacific/Palau</td>
<td>Pacific/Palau</td>
</tr>
<tr>
<td>Pacific/Pitcairn</td>
<td>Pacific/Pitcairn</td>
</tr>
<tr>
<td>Pacific/Pohnpei</td>
<td>SST</td>
</tr>
<tr>
<td>Pacific/Ponape</td>
<td>SST</td>
</tr>
<tr>
<td>Pacific/Port_Moresby</td>
<td>Pacific/Port_Moresby</td>
</tr>
<tr>
<td>Pacific/Rarotonga</td>
<td>Pacific/Rarotonga</td>
</tr>
<tr>
<td>Pacific/Saipan</td>
<td>Pacific/Saipan</td>
</tr>
<tr>
<td>Pacific/Samoa</td>
<td>Pacific/Samoa</td>
</tr>
<tr>
<td>Pacific/Tahiti</td>
<td>Pacific/Tahiti</td>
</tr>
<tr>
<td>Pacific/Tarawa</td>
<td>Pacific/Tarawa</td>
</tr>
<tr>
<td>Pacific/Tongatapu</td>
<td>Pacific/Tongatapu</td>
</tr>
<tr>
<td>Pacific/Truk</td>
<td>Pacific/Truk</td>
</tr>
<tr>
<td>Pacific/Wake</td>
<td>Pacific/Wake</td>
</tr>
<tr>
<td>Pacific/Wallis</td>
<td>Pacific/Wallis</td>
</tr>
<tr>
<td>Pacific/Yap</td>
<td>Pacific/Yap</td>
</tr>
<tr>
<td>Poland</td>
<td>Poland</td>
</tr>
<tr>
<td>Portugal</td>
<td>Portugal</td>
</tr>
<tr>
<td>ROC</td>
<td>ROC</td>
</tr>
<tr>
<td>ROK</td>
<td>ROK</td>
</tr>
<tr>
<td>SST</td>
<td>SST</td>
</tr>
<tr>
<td>name</td>
<td>abbrev</td>
</tr>
<tr>
<td>-------------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>Singapore</td>
<td>Singapore</td>
</tr>
<tr>
<td>SystemV/AST4</td>
<td>SystemV/AST4</td>
</tr>
<tr>
<td>SystemV/AST4ADT</td>
<td>SystemV/AST4ADT</td>
</tr>
<tr>
<td>SystemV/CST6</td>
<td>SystemV/CST6</td>
</tr>
<tr>
<td>SystemV/CST6CDT</td>
<td>SystemV/CST6CDT</td>
</tr>
<tr>
<td>SystemV/EST5</td>
<td>SystemV/EST5</td>
</tr>
<tr>
<td>SystemV/EST5EDT</td>
<td>SystemV/EST5EDT</td>
</tr>
<tr>
<td>SystemV/HST10</td>
<td>SystemV/HST10</td>
</tr>
<tr>
<td>SystemV/MST7</td>
<td>SystemV/MST7</td>
</tr>
<tr>
<td>SystemV/MST7MDT</td>
<td>SystemV/MST7MDT</td>
</tr>
<tr>
<td>SystemV/PST8</td>
<td>SystemV/PST8</td>
</tr>
<tr>
<td>SystemV/PST8PDT</td>
<td>SystemV/PST8PDT</td>
</tr>
<tr>
<td>SystemV/YST9</td>
<td>SystemV/YST9</td>
</tr>
<tr>
<td>SystemV/YST9YDT</td>
<td>SystemV/YST9YDT</td>
</tr>
<tr>
<td>Turkey</td>
<td>Turkey</td>
</tr>
<tr>
<td>UCT</td>
<td>UCT</td>
</tr>
<tr>
<td>US/Alaska</td>
<td>AST</td>
</tr>
<tr>
<td>US/Aleutian</td>
<td>US/Aleutian</td>
</tr>
<tr>
<td>US/Arizona</td>
<td>PNT</td>
</tr>
<tr>
<td>US/Central</td>
<td>CST</td>
</tr>
<tr>
<td>US/East‑Indiana</td>
<td>IET</td>
</tr>
<tr>
<td>US/Eastern</td>
<td>US/Eastern</td>
</tr>
<tr>
<td>US/Hawaii</td>
<td>US/Hawaii</td>
</tr>
<tr>
<td>US/Indiana‑Starke</td>
<td>US/Indiana‑Starke</td>
</tr>
<tr>
<td>US/Michigan</td>
<td>US/Michigan</td>
</tr>
<tr>
<td>US/Mountain</td>
<td>Navajo</td>
</tr>
<tr>
<td>US/Pacific</td>
<td>PST</td>
</tr>
<tr>
<td>US/Pacific‑New</td>
<td>PST</td>
</tr>
<tr>
<td>US/Samoa</td>
<td>US/Samoa</td>
</tr>
<tr>
<td>UTC</td>
<td>UCT</td>
</tr>
<tr>
<td>Universal</td>
<td>UCT</td>
</tr>
<tr>
<td>VST</td>
<td>VST</td>
</tr>
</tbody>
</table>
### 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
```sql
CREATE TABLE tbl1(u UNION(num INT, str VARCHAR));
```

-- 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 targets.
-- Note: only if the cast is unambiguous!
-- More details in the 'Union casts' section below.
```sql
INSERT INTO tbl1 values (1), ('two'), (union_value(str := 'three'));
```

-- Union use the member types varchar cast functions when casting to varchar.
```sql
SELECT u FROM tbl1;
```
-- returns:
-- 1
-- two
-- three

-- Select all the 'str' members
```sql
SELECT union_extract(u, 'str') FROM tbl1;
```
-- Alternatively, you can use 'dot syntax' like with structs
```sql
SELECT u.str FROM tbl1;
```
-- returns:
-- NULL
-- two
-- three

-- Select the currently active tag from the union as an enum.
SELECT union_tag(u) FROM tbl1;
-- returns:
-- num
-- str
-- str

### Union Casts

Compared to other nested types, UNIONs 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 UNIONs 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 INT, v VARCHAR) cast will always cast the FLOAT to the INT 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 INT, num INT) 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::INT) will create a UNION with a single member of type INT 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 INT, b INT)).

#### 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.

<table>
<thead>
<tr>
<th>Ok</th>
<th>Source</th>
<th>Target</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>UNION(a A, b B)</td>
<td>UNION(a A, b B, c C)</td>
<td></td>
</tr>
<tr>
<td>✓</td>
<td>UNION(a A, b B)</td>
<td>UNION(a A, b C)</td>
<td>if B can be implicitly cast to C</td>
</tr>
<tr>
<td>✗</td>
<td>UNION(a A, b B, c C)</td>
<td>UNION(a A, b B)</td>
<td></td>
</tr>
<tr>
<td>✗</td>
<td>UNION(a A, b B)</td>
<td>UNION(a A, b C)</td>
<td>if B can’t be implicitly cast to C</td>
</tr>
<tr>
<td>✗</td>
<td>UNION(A, B, D)</td>
<td>UNION(A, B, C)</td>
<td></td>
</tr>
</tbody>
</table>

**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 STRUCTs. 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 Nested 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.

The following matrix describes which conversions are supported. When implicit casting is allowed, it implies that explicit casting is also possible.
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.

Casting operations that result in loss of precision are typically allowed. For example, it is possible to cast a numeric type with fractional digits like DECIMAL, FLOAT or DOUBLE to an integral type like INTEGER:

```sql
SELECT cast(pi() AS INTEGER);
```

Casting operations that would result in a value overflow are typically not allowed. 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 would fail.
DuckDB Documentation

results in a runtime error:

```
SELECT CAST(999 AS TINYINT);
```

So even though the cast operation from INTEGER to TINYINT is supported, it is not possible for this particular value.

The VARCHAR type (also available under the aliases TEXT and STRING) acts like an universal target: any arbitrary value of any arbitrary type can always be cast to the VARCHAR type. Casting from VARCHAR to another data type is generally supported, but may fail at runtime if DuckDB cannot figure out how to parse and convert the provided value to the target data type.

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 Statement

The CASE statement 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;
```

```
-- 1, 2, 3
-- 0, 0, 1

-- this is equivalent to:
SELECT i, IF(i > 2, 1, 0) AS test FROM integers;
```

```
-- 1, 2, 3
-- 0, 0, 1
```

The WHEN cond THEN expr part of the CASE statement 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;
```

```
-- 1, 2, 3
-- 10, 20, 0
```

The ELSE part of the CASE statement is optional. If no else statement is provided and none of the conditions match, the CASE statement 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;
-- 1, 2, 3
-- 10, NULL, NULL

After the CASE but before the WHEN an individual expression can also be provided. When this is done, the CASE statement is essentially 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;
-- 1, 2, 3
-- 10, 20, 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.

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 easier to type shorthand expr::TYPENAME, which is also present in PostgreSQL.

SELECT CAST(i AS VARCHAR) FROM generate_series(1, 3) tbl(i);
-- "1", "2", "3"
SELECT i::DOUBLE FROM generate_series(1, 3) tbl(i);
-- 1.0, 2.0, 3.0

SELECT CAST('hello' AS INTEGER);
-- Conversion Error: Could not convert string 'hello' to INT32
SELECT TRY_CAST('hello' AS INTEGER);
-- NULL

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.

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.

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.
Implicit Casting

In many situations, the system will add casts by itself. This is called *implicit* casting. This happens for example when a function is called with an argument that does not match the type of the function, but can be casted to the desired type.

Consider the function \( \sin(\text{DOUBLE}) \). This function takes as input argument a column of type \( \text{DOUBLE} \), 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.

Generally, implicit casts only cast upwards. That is to say, we can implicitly cast an \( \text{INTEGER} \) to a \( \text{BIGINT} \), but not the other way around.

Allowed Casting Operations

Values of a particular data type can typically not be casted to any arbitrary target data type. The supported cast operations are described in the *typecasting page* as part of the data types documentation.

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 \( \text{BINARY} \) 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 \( \text{BINARY} \) collation unless required otherwise.

Using Collations

In the stand-alone installation of DuckDB three collations are included: \( \text{NOCASE} \), \( \text{NOACCENT} \) and \( \text{NFC} \). The \( \text{NOCASE} \) collation compares characters as equal regardless of their casing. The \( \text{NOACCENT} \) collation compares characters as equal regardless of their accents. The \( \text{NFC} \) collation performs NFC-normalized comparisons, see *Unicode normalization* for more information.

```
SELECT 'hello' = 'hEllO';
-- false
SELECT 'hello' COLLATE NOCASE = 'hEllO';
-- true
```
SELECT 'hello' = 'hêllö';
-- false
SELECT 'hello' COLLATE NOACCENT = 'hêllö';
-- 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 * FROM names, other_names WHERE names.name = other_names.name;
-- ERROR: Cannot combine types with different collation!
```

-- need to manually overwrite the collation!
**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.

If you are using the C++ API, you may find the extension in the extension/icu folder of the DuckDB project. Using the C++ API, the extension can be loaded as follows:

```cpp
DuckDB db;
db.LoadExtension<ICUExtension>();
```

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;

```sql
SELECT * FROM pragma_collations();
```

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:

```sql
CREATE TABLE strings (s VARCHAR COLLATE DE);
INSERT INTO strings VALUES ('Gabel'), ('Göbel'), ('Goethe'), ('Goldmann'), ('Göthe'), ('Götz');
SELECT * FROM strings ORDER BY s;
```

**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.

```sql
SELECT * FROM names, other_names WHERE names.name COLLATE NOACCENT.NOCASE = other_names.name COLLATE NOACCENT.NOCASE;
```

-- hannes|HANNES

```sql
SELECT * FROM names, other_names WHERE names.name COLLATE NOACCENT.NOCASE = other_names.name COLLATE NOACCENT.NOCASE;
```

-- hannes|HANNES
The table below shows the standard distinction operators. These operators treat NULL values as equal.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>less than</td>
<td>2 &lt; 3</td>
<td>true</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
<td>2 &gt; 3</td>
<td>false</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal to</td>
<td>2 &lt;= 3</td>
<td>true</td>
</tr>
<tr>
<td>=&gt;</td>
<td>greater than or equal to</td>
<td>4 &gt;= NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>=</td>
<td>equal</td>
<td>NULL = NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>&lt;&gt; or !=</td>
<td>not equal</td>
<td>2 &lt;&gt; 2</td>
<td>false</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a BETWEEN x AND y</td>
<td>equivalent to a &gt;= x AND a &lt;= y</td>
</tr>
<tr>
<td>a NOT BETWEEN x AND y</td>
<td>equivalent to a &lt; x OR a &gt; y</td>
</tr>
<tr>
<td>expression IS NULL</td>
<td>true if expression is NULL, false otherwise</td>
</tr>
<tr>
<td>expression ISNULL</td>
<td>alias for IS NULL (non-standard)</td>
</tr>
<tr>
<td>expression IS NOT NULL</td>
<td>false if expression is NULL, true otherwise</td>
</tr>
<tr>
<td>expression NOTNULL</td>
<td>alias for IS NOT NULL (non-standard)</td>
</tr>
</tbody>
</table>
**IN Operator**

The IN operator checks containment of the left expression inside 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.

```sql
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
```

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).

The IN operator can also be used with a subquery that returns a single column. See the subqueries page for more information.

**Logical Operators**

The following logical operators are available: AND, OR and NOT. SQL uses a three-valued 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:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>a AND b</th>
<th>a OR b</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>NULL</td>
<td>NULL</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>NULL</td>
<td>false</td>
<td>NULL</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a</th>
<th>NOT a</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
</tr>
</tbody>
</table>
The operators AND and OR are commutative, that is, you can switch the left and right operand without affecting the result.

**Star Expression**

**Examples**

```
-- select all columns present in the FROM clause
SELECT * FROM table_name;

-- select all columns from the table called "table_name"
SELECT table_name.* FROM table_name JOIN other_table_name USING (id);

-- 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 expression
SELECT COLUMNS(c -> c LIKE '%num%') FROM addresses;

-- select all columns matching the given regex from the table
SELECT COLUMNS('number\d+') FROM addresses;
```

**Syntax**

**Star Expression**

The * expression can be used in a SELECT statement to select all columns that are projected in the FROM clause.

```
SELECT * FROM tbl;
```

The * expression can be modified using the EXCLUDE and REPLACE.

**EXCLUDE Clause**  EXCLUDE allows us to exclude specific columns from the * expression.

```
SELECT * EXCLUDE (col) FROM tbl;
```

**REPLACE Clause**  REPLACE allows us to replace specific columns with different expressions.

```
SELECT * REPLACE (col / 1000 AS col) FROM tbl;
```
The COLUMNS expression can be used to execute the same expression on multiple columns. Like the * expression, it can only be used in the SELECT clause.

```sql
CREATE TABLE numbers (id INT, number INT);
INSERT INTO numbers VALUES (1, 10), (2, 20), (3, NULL);
SELECT min(COLUMNS(*)), count(COLUMNS(*)) FROM numbers;
```

<table>
<thead>
<tr>
<th>min(numbers.id)</th>
<th>min(numbers.number)</th>
<th>count(numbers.id)</th>
<th>count(numbers.number)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

The * expression in the COLUMNS statement can also contain EXCLUDE or REPLACE, similar to regular star expressions.

```sql
SELECT min(COLUMNS(* REPLACE (number + id AS number))), count(COLUMNS(* EXCLUDE (number))) FROM numbers;
```

<table>
<thead>
<tr>
<th>min(numbers.id)</th>
<th>min(number := (number + id))</th>
<th>count(numbers.id)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11</td>
<td>3</td>
</tr>
</tbody>
</table>

COLUMNS expressions can also be combined, as long as the COLUMNS contains the same (star) expression:

```sql
SELECT COLUMNS(*) + COLUMNS(*) FROM numbers;
```

<table>
<thead>
<tr>
<th>(numbers.id + numbers.id)</th>
<th>(numbers.number + numbers.number)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>40</td>
</tr>
<tr>
<td>6</td>
<td>NULL</td>
</tr>
</tbody>
</table>

**COLUMNS Regular Expression**

COLUMNS supports passing a regex in as a string constant:

```sql
SELECT COLUMNS('^(id|numbers?)$') FROM numbers;
```

<table>
<thead>
<tr>
<th>id</th>
<th>number</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>
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 columns.

```sql
SELECT COLUMNS(c -> c LIKE '%num%') FROM numbers;
```

<table>
<thead>
<tr>
<th>number</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
</tr>
<tr>
<td>20</td>
</tr>
<tr>
<td>NULL</td>
</tr>
</tbody>
</table>

`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 nested functions pages for more details on working with structs.

```sql
-- All keys within a struct can be returned as separate columns using *
SELECT st.* FROM (SELECT {'x': 1, 'y': 2, 'z': 3} AS st);
```

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Subqueries

Scalar Subquery

Scalar subqueries are subqueries that return a single value. They can be used anywhere where a regular expression can be used. If a scalar subquery returns more than a single value, the first value returned will be used.

Consider the following table:
Grades

<table>
<thead>
<tr>
<th>grade</th>
<th>course</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Math</td>
</tr>
<tr>
<td>9</td>
<td>Math</td>
</tr>
<tr>
<td>8</td>
<td>CS</td>
</tr>
</tbody>
</table>

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;
-- {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);
-- {Math}

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 (SELECT * FROM grades WHERE course = 'Math');
-- true

SELECT EXISTS (SELECT * FROM grades WHERE course = 'History');
-- false

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 (SELECT * FROM interest WHERE interest.PersonId = Person.id);

```
<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Jane</td>
</tr>
</tbody>
</table>
```

**Note.** 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);
-- 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);
-- {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;
```

```
<table>
<thead>
<tr>
<th>t</th>
<th>struct(x bigint, y varchar)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>{'x': 41, 'y': hello}</td>
</tr>
<tr>
<td></td>
<td>{'x': 42, 'y': hello}</td>
</tr>
<tr>
<td></td>
<td>{'x': 43, 'y': hello}</td>
</tr>
</tbody>
</table>
```

**Functions**

**Function Syntax**

**Query Functions**  
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;
```

```
<table>
<thead>
<tr>
<th>function_name</th>
<th>function_type</th>
<th>return_type</th>
<th>parameters</th>
<th>parameter_types</th>
</tr>
</thead>
<tbody>
<tr>
<td>varchar</td>
<td>varchar</td>
<td>varchar</td>
<td>varchar[]</td>
<td>varchar[]</td>
</tr>
<tr>
<td>varchar</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Bitstring Functions

This section describes functions and operators for examining and manipulating bit 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.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>bitwise AND</td>
<td>'10101':BIT &amp; '10001':BIT</td>
<td>10001</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bitwise OR</td>
<td>'1011':BIT</td>
</tr>
<tr>
<td>xor</td>
<td>bitwise XOR</td>
<td>xor('101':BIT, '001':BIT)</td>
<td>100</td>
</tr>
<tr>
<td>~</td>
<td>bitwise NOT</td>
<td>~( '101':BIT)</td>
<td>010</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>bitwise shift left</td>
<td>'1001011':BIT &lt;&lt; 3</td>
<td>1011000</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>bitwise shift right</td>
<td>'1001011':BIT &gt;&gt; 3</td>
<td>0001001</td>
</tr>
</tbody>
</table>

Bitstring Functions

The table below shows the available scalar functions for BIT type.
**Function**  
**Description**  
**Example**  
**Result**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit_count(bitstring)</td>
<td>Returns the number of set bits in the bitstring.</td>
<td>bit_count('1101011':BIT)</td>
<td>5</td>
</tr>
<tr>
<td>bit_length(bitstring)</td>
<td>Returns the number of bits in the bitstring.</td>
<td>bit_length('1101011':BIT)</td>
<td>7</td>
</tr>
<tr>
<td>bit_position(substring, bitstring)</td>
<td>Returns first starting index of the specified substring within bits, or zero if it's not present. The first (leftmost) bit is indexed 1.</td>
<td>bit_position('010':BIT, '1110101':BIT)</td>
<td>4</td>
</tr>
<tr>
<td>bitstring(bitstring, length)</td>
<td>Returns a bitstring of determined length.</td>
<td>bitstring('1010':BIT, 7)</td>
<td>0001010</td>
</tr>
<tr>
<td>get_bit(bitstring, index)</td>
<td>Extracts the nth bit from bitstring; the first (leftmost) bit is indexed 0.</td>
<td>get_bit('0110010':BIT, 2)</td>
<td>1</td>
</tr>
<tr>
<td>length(bitstring)</td>
<td>Alias for bit_length.</td>
<td>length('1101011':BIT)</td>
<td>7</td>
</tr>
<tr>
<td>octet_length(bitstring)</td>
<td>Returns the number of bytes in the bitstring.</td>
<td>octet_length('1101011':BIT)</td>
<td>1</td>
</tr>
<tr>
<td>set_bit(bitstring, index, new_value)</td>
<td>Sets the nth bit in bitstring to newvalue; the first (leftmost) bit is indexed 0. Returns a new bitstring.</td>
<td>set_bit('0110010':BIT, 2, 0)</td>
<td>0100100</td>
</tr>
</tbody>
</table>

**Bitstring Aggregate Functions**

These aggregate functions are available for BIT type.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit_and(arg)</td>
<td>Returns the bitwise AND operation performed on all bitstrings in a given expression.</td>
<td>bit_and(A)</td>
</tr>
<tr>
<td>bit_or(arg)</td>
<td>Returns the bitwise OR operation performed on all bitstrings in a given expression.</td>
<td>bit_or(A)</td>
</tr>
<tr>
<td>bit_xor(arg)</td>
<td>Returns the bitwise XOR operation performed on all bitstrings in a given expression.</td>
<td>bit_xor(A)</td>
</tr>
</tbody>
</table>
Function Description Example

| bitstring_agg(arg) | Returns a bitstring with bits set for each distinct value. | bitstring_agg(A) |
| bitstring_agg(arg, min, max) | Returns a bitstring with bits set for each distinct value. | bitstring_agg(A, 1, 42) |

**Bitstring Aggregation**  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.

The combination of BIT_COUNT and BITSTRING_AGG could be used as an alternative to COUNT DISTINCT, with possible performance improvements in cases of low cardinality and dense values.

**Blob Functions**

This section describes functions and operators for examining and manipulating blob values.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>blob</td>
<td></td>
<td>blob</td>
<td>Blob concatenation</td>
</tr>
<tr>
<td>decode(blob)</td>
<td>Convert blob to varchar. Fails if blob is not valid utf-8.</td>
<td>decode('\xC3\xBC'::BLOB)</td>
<td>ü</td>
</tr>
<tr>
<td>encode(string)</td>
<td>Convert varchar to blob. Converts utf-8 characters into literal encoding.</td>
<td>encode('my_string_with_ ü')</td>
<td>my_string_with_ \xC3\xBC</td>
</tr>
<tr>
<td>octet_length(blob)</td>
<td>Number of bytes in blob</td>
<td>octet_length('\xAA\xBB'::BLOB)</td>
<td>2</td>
</tr>
</tbody>
</table>

**Date Format Functions**

The strftime and strptime functions can be used to convert between dates/timestamps 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

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

strptime(string, format) 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
```

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 COPY statement
COPY dates FROM 'test.csv' (DATEFORMAT '%d/%m/%Y', TIMESTAMPFORMAT '%A, %-d %B %Y - %I:%M:%S %p');

-- in read_csv function
SELECT * FROM read_csv('test.csv', dateformat = '%m/%d/%Y');
```

Format Specifiers

Below is a full list of all available format specifiers.

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>%a</td>
<td>Abbreviated weekday name.</td>
<td>Sun, Mon,...</td>
</tr>
<tr>
<td>%A</td>
<td>Full weekday name.</td>
<td>Sunday, Monday,...</td>
</tr>
<tr>
<td>%b</td>
<td>Abbreviated month name.</td>
<td>Jan, Feb,..., Dec</td>
</tr>
<tr>
<td>%B</td>
<td>Full month name.</td>
<td>January, February,...</td>
</tr>
<tr>
<td>%c</td>
<td>ISO date and time representation</td>
<td>1992-03-02 10:30:20</td>
</tr>
<tr>
<td>Specifier</td>
<td>Description</td>
<td>Example</td>
</tr>
<tr>
<td>-----------</td>
<td>------------------------------------------------------------------------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>%d</td>
<td>Day of the month as a zero-padded decimal.</td>
<td>01, 02, ..., 31</td>
</tr>
<tr>
<td>%-d</td>
<td>Day of the month as a decimal number.</td>
<td>1, 2, ..., 30</td>
</tr>
<tr>
<td>%f</td>
<td>Microsecond as a decimal number, zero-padded on the left.</td>
<td>0000000 - 9999999</td>
</tr>
<tr>
<td>%g</td>
<td>Millisecond as a decimal number, zero-padded on the left.</td>
<td>000 - 999</td>
</tr>
<tr>
<td>%G</td>
<td>ISO 8601 year with century representing the year that contains the greater part of the ISO week (see %V).</td>
<td>0001, 0002, ..., 2013, 2014, ..., 9998, 9999</td>
</tr>
<tr>
<td>%H</td>
<td>Hour (24-hour clock) as a zero-padded decimal number.</td>
<td>00, 01, ..., 23</td>
</tr>
<tr>
<td>%-H</td>
<td>Hour (24-hour clock) as a decimal number.</td>
<td>0, 1, ..., 23</td>
</tr>
<tr>
<td>%I</td>
<td>Hour (12-hour clock) as a zero-padded decimal number.</td>
<td>01, 02, ..., 12</td>
</tr>
<tr>
<td>%-I</td>
<td>Hour (12-hour clock) as a decimal number.</td>
<td>1, 2, ..., 12</td>
</tr>
<tr>
<td>%j</td>
<td>Day of the year as a zero-padded decimal number.</td>
<td>001, 002, ..., 366</td>
</tr>
<tr>
<td>%-j</td>
<td>Day of the year as a decimal number.</td>
<td>1, 2, ..., 366</td>
</tr>
<tr>
<td>%m</td>
<td>Month as a zero-padded decimal number.</td>
<td>01, 02, ..., 12</td>
</tr>
<tr>
<td>%-m</td>
<td>Month as a decimal number.</td>
<td>1, 2, ..., 12</td>
</tr>
<tr>
<td>%M</td>
<td>Minute as a zero-padded decimal number.</td>
<td>00, 01, ..., 59</td>
</tr>
<tr>
<td>%-M</td>
<td>Minute as a decimal number.</td>
<td>0, 1, ..., 59</td>
</tr>
<tr>
<td>%n</td>
<td>Nanosecond as a decimal number, zero-padded on the left.</td>
<td>0000000000 - 9999999999</td>
</tr>
<tr>
<td>%p</td>
<td>Locale's AM or PM.</td>
<td>AM, PM</td>
</tr>
<tr>
<td>%S</td>
<td>Second as a zero-padded decimal number.</td>
<td>00, 01, ..., 59</td>
</tr>
<tr>
<td>%-S</td>
<td>Second as a decimal number.</td>
<td>0, 1, ..., 59</td>
</tr>
<tr>
<td>%u</td>
<td>ISO 8601 weekday as a decimal number where 1 is Monday.</td>
<td>1, 2, ..., 7</td>
</tr>
<tr>
<td>%U</td>
<td>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.</td>
<td>00, 01, ..., 53</td>
</tr>
<tr>
<td>%V</td>
<td>ISO 8601 week as a decimal number with Monday as the first day of the week. Week 01 is the week containing Jan 4.</td>
<td>01, ..., 53</td>
</tr>
<tr>
<td>%w</td>
<td>Weekday as a decimal number.</td>
<td>0, 1, ..., 6</td>
</tr>
</tbody>
</table>
DuckDB Documentation

Specifier | Description | Example
---|---|---
%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.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>addition of days (integers)</td>
<td>DATE '1992-03-22' + 5</td>
<td>1992-03-27</td>
</tr>
<tr>
<td>+</td>
<td>addition of an INTERVAL</td>
<td>DATE '1992-03-22' + INTERVAL 5 DAY</td>
<td>1992-03-27</td>
</tr>
<tr>
<td>+</td>
<td>addition of a variable INTERVAL</td>
<td>SELECT DATE '1992-03-22' + INTERVAL 1 DAY * d.days FROM (VALUES (5), (11)) AS d(days)</td>
<td>1992-03-27 and 1992-04-02</td>
</tr>
</tbody>
</table>
Operator | Description | Example | Result
--- | --- | --- | ---
- | subtraction of a variable INTERVAL | `SELECT DATE '1992-03-27' - INTERVAL 1 DAY * d.days FROM (VALUES (5), (11)) AS d(days)` | 1992-03-22 and 1992-03-16

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.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>current_date</code></td>
<td>Current date (at start of current transaction)</td>
<td><code>current_date</code></td>
<td>2022-10-08</td>
</tr>
<tr>
<td><code>date_add(date, interval)</code></td>
<td>Add the interval to the date</td>
<td><code>date_add(DATE '1992-09-15', INTERVAL 2 MONTH)</code></td>
<td>1992-11-15</td>
</tr>
<tr>
<td><code>date_diff(part, startdate, enddate)</code></td>
<td>The number of partition boundaries between the dates</td>
<td><code>date_diff('month', DATE '1992-09-15', DATE '1992-11-14')</code></td>
<td>2</td>
</tr>
<tr>
<td><code>datediff(part, startdate, enddate)</code></td>
<td>Alias of date_diff. The number of partition boundaries between the dates</td>
<td><code>datediff('month', DATE '1992-09-15', DATE '1992-11-14')</code></td>
<td>2</td>
</tr>
<tr>
<td><code>date_part(part, date)</code></td>
<td>Get the subfield (equivalent to extract)</td>
<td><code>date_part('year', DATE '1992-09-20')</code></td>
<td>1992</td>
</tr>
<tr>
<td><code>datepart(part, date)</code></td>
<td>Alias of date_part. Get the subfield (equivalent to extract)</td>
<td><code>datepart('year', DATE '1992-09-20')</code></td>
<td>1992</td>
</tr>
<tr>
<td><code>date_sub(part, startdate, enddate)</code></td>
<td>The number of complete partitions between the dates</td>
<td><code>date_sub('month', DATE '1992-09-15', DATE '1992-11-14')</code></td>
<td>1</td>
</tr>
</tbody>
</table>
### Function Descriptions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>datesub(part, startdate, enddate)</code></td>
<td>Alias of date_sub. The number of complete partitions between the dates</td>
<td><code>datesub('month', DATE '1992-09-15', DATE '1992-11-14')</code></td>
<td>1</td>
</tr>
<tr>
<td><code>date_trunc(part, date)</code></td>
<td>Truncate to specified precision</td>
<td><code>date_trunc('month', DATE '1992-03-07')</code></td>
<td>1992-03-01</td>
</tr>
<tr>
<td><code>datetrunc(part, date)</code></td>
<td>Alias of date_trunc. Truncate to specified precision</td>
<td><code>datetrunc('month', DATE '1992-03-07')</code></td>
<td>1992-03-01</td>
</tr>
<tr>
<td><code>dayname(date)</code></td>
<td>The (English) name of the weekday</td>
<td><code>dayname(DATE '1992-09-20')</code></td>
<td>Sunday</td>
</tr>
<tr>
<td><code>isfinite(date)</code></td>
<td>Returns true if the date is finite, false otherwise</td>
<td><code>isfinite(DATE '1992-03-07')</code></td>
<td>true</td>
</tr>
<tr>
<td><code>isinf(date)</code></td>
<td>Returns true if the date is infinite, false otherwise</td>
<td><code>isinf(DATE '-infinity')</code></td>
<td>true</td>
</tr>
<tr>
<td><code>extract(part from date)</code></td>
<td>Get subfield from a date</td>
<td><code>extract('year' FROM DATE '1992-09-20')</code></td>
<td>1992</td>
</tr>
<tr>
<td><code>greatest(date, date)</code></td>
<td>The later of two dates</td>
<td><code>greatest(DATE '1992-09-20', DATE '1992-03-07')</code></td>
<td>1992-09-20</td>
</tr>
<tr>
<td><code>last_day(date)</code></td>
<td>The last day of the corresponding month in the date</td>
<td><code>last_day(DATE '1992-09-20')</code></td>
<td>1992-09-30</td>
</tr>
<tr>
<td><code>least(date, date)</code></td>
<td>The earlier of two dates</td>
<td><code>least(DATE '1992-09-20', DATE '1992-03-07')</code></td>
<td>1992-03-07</td>
</tr>
<tr>
<td><code>make_date(bigint, bigint, bigint)</code></td>
<td>The date for the given parts</td>
<td><code>make_date(1992, 9, 20)</code></td>
<td>1992-09-20</td>
</tr>
<tr>
<td><code>monthname(date)</code></td>
<td>The (English) name of the month</td>
<td><code>monthname(DATE '1992-09-20')</code></td>
<td>September</td>
</tr>
<tr>
<td><code>strftime(date, format)</code></td>
<td>Converts a date to a string according to the format string</td>
<td><code>strftime(date '1992-01-01', '%a, %d %B %Y')</code></td>
<td>Wed, 1 January 1992</td>
</tr>
</tbody>
</table>
Function | Description | Example | Result
--- | --- | --- | ---
time_bucket(
  bucket_width,
  date[,origin]) | Truncate date by the specified interval bucket_width. Buckets are aligned relative to origin date. origin defaults to 2000-01-03 for buckets that don't include a month or year interval, and to 2000-01-01 for month and year buckets. | time_bucket(INTERVAL '2 weeks', DATE '1992-04-20', DATE '1992-04-01') | 1992-04-15

time_bucket(
  bucket_width,
  date[,offset]) | Truncate date by the specified interval bucket_width. Buckets are offset by offset interval. | time_bucket(INTERVAL '2 months', DATE '1992-04-20', INTERVAL '1 month') | 1992-04-01

today() | Current date (start of current transaction) | today() | 2022-10-08

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. The fields are specified as strings that contain the part name of the field.

**Part Specifiers**

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.

**Usable as Date Part Specifiers and in Intervals**
DuckDB Documentation

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Description</th>
<th>Synonyms</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>'century'</td>
<td>Gregorian century</td>
<td>'cent', 'centuries', 'c'</td>
<td>21</td>
</tr>
<tr>
<td>'day'</td>
<td>Gregorian day</td>
<td>'days', 'd', 'dayofmonth'</td>
<td>3</td>
</tr>
<tr>
<td>'decade'</td>
<td>Gregorian decade</td>
<td>'dec', 'decades', 'decs'</td>
<td>202</td>
</tr>
<tr>
<td>'hour'</td>
<td>Hours</td>
<td>'hr', 'hours', 'hrs', 'h'</td>
<td>11</td>
</tr>
<tr>
<td>'microseconds'</td>
<td>Sub-minute microseconds</td>
<td>'microsecond', 'us', 'usec', 'usecs', 'usec', 'usecs'</td>
<td>44123456</td>
</tr>
<tr>
<td>'millennium'</td>
<td>Gregorian millennium</td>
<td>'mil', 'millenniums', 'milenia', 'mils', 'millenium'</td>
<td>3</td>
</tr>
<tr>
<td>'millisecond'</td>
<td>Sub-minute millisecond</td>
<td>'millisecond', 'ms', 'msec', 'msecs', 'msec', 'msecs'</td>
<td>44123</td>
</tr>
<tr>
<td>'minute'</td>
<td>Minutes</td>
<td>'min', 'minutes', 'mins', 'm'</td>
<td>59</td>
</tr>
<tr>
<td>'month'</td>
<td>Gregorian month</td>
<td>'mon', 'months', 'mons'</td>
<td>8</td>
</tr>
<tr>
<td>'quarter'</td>
<td>Quarter of the year (1-4)</td>
<td>'quarters'</td>
<td>3</td>
</tr>
<tr>
<td>'second'</td>
<td>Seconds</td>
<td>'sec', 'seconds', 'secs', 's'</td>
<td>44</td>
</tr>
<tr>
<td>'year'</td>
<td>Gregorian year</td>
<td>'yr', 'y', 'years', 'yrs'</td>
<td>2021</td>
</tr>
</tbody>
</table>

**Usable in Date Part Specifiers Only**

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Description</th>
<th>Synonyms</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>'dayofweek'</td>
<td>Day of the week (Sunday = 0, Saturday = 6)</td>
<td>'weekday', 'dow'</td>
<td>2</td>
</tr>
<tr>
<td>'dayofyear'</td>
<td>Day of the year (1-365/366)</td>
<td>'doy'</td>
<td>215</td>
</tr>
<tr>
<td>'epoch'</td>
<td>Seconds since 1970-01-01</td>
<td></td>
<td>1627991984</td>
</tr>
<tr>
<td>'era'</td>
<td>Gregorian era (CE/AD, BCE/BC)</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>'isodow'</td>
<td>ISO day of the week (Monday = 1, Sunday = 7)</td>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>

533
<table>
<thead>
<tr>
<th>Specifier</th>
<th>Description</th>
<th>Synonyms</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>'isoyear'</td>
<td>ISO Year number (Starts on Monday of week containing Jan 4th)</td>
<td></td>
<td>2021</td>
</tr>
<tr>
<td>'timezone'</td>
<td>Time zone offset in seconds</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>'timezone_hour'</td>
<td>Time zone offset hour portion</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>'timezone_minute'</td>
<td>Time zone offset minute portion</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>'week'</td>
<td>Week number</td>
<td>'weeks', 'w'</td>
<td>31</td>
</tr>
<tr>
<td>'yearweek'</td>
<td>ISO year and week number in YYYYWW format</td>
<td></td>
<td>202131</td>
</tr>
</tbody>
</table>

Note that the time zone parts are all zero unless a time zone plugin such as ICU has been installed to support `TIMESTAMP WITH TIME ZONE`.

**Part Functions** There are dedicated extraction functions to get certain subfields:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>century(date)</td>
<td>Century</td>
<td>century(date '1992-02-15')</td>
<td>20</td>
</tr>
<tr>
<td>day(date)</td>
<td>Day</td>
<td>day(date '1992-02-15')</td>
<td>15</td>
</tr>
<tr>
<td>dayofmonth(date)</td>
<td>Day (synonym)</td>
<td>dayofmonth(date '1992-02-15')</td>
<td>15</td>
</tr>
<tr>
<td>dayofweek(date)</td>
<td>Numeric weekday</td>
<td>dayofweek(date '1992-02-15')</td>
<td>6</td>
</tr>
<tr>
<td>dayofyear(date)</td>
<td>Day of the year (starts from 1, i.e., January 1 = 1)</td>
<td>dayofyear(date '1992-02-15')</td>
<td>46</td>
</tr>
<tr>
<td>decade(date)</td>
<td>Decade (year / 10)</td>
<td>decade(date '1992-02-15')</td>
<td>199</td>
</tr>
<tr>
<td>epoch(date)</td>
<td>Seconds since 1970-01-01</td>
<td>epoch(date '1992-02-15')</td>
<td>698112080</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>Example</td>
<td>Result</td>
</tr>
<tr>
<td>------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>----------------------------------------</td>
<td>--------</td>
</tr>
<tr>
<td><code>era(date)</code></td>
<td>Calendar era</td>
<td><code>era(date '0044-03-15 (BC)')</code></td>
<td>0</td>
</tr>
<tr>
<td><code>hour(date)</code></td>
<td>Hours</td>
<td><code>hour(timestamp '2021-08-03 11:59:44.123456')</code></td>
<td>11</td>
</tr>
<tr>
<td><code>isodow(date)</code></td>
<td>Numeric ISO weekday (Monday = 1, Sunday = 7)</td>
<td><code>isodow(date '1992-02-15')</code></td>
<td>6</td>
</tr>
<tr>
<td><code>isoyear(date)</code></td>
<td>ISO Year number (Starts on Monday of week containing Jan 4th)</td>
<td><code>isoyear(date '2022-01-01')</code></td>
<td>2021</td>
</tr>
<tr>
<td><code>microsecond(date)</code></td>
<td>Sub-minute microseconds</td>
<td><code>microsecond(timestamp '2021-08-03 11:59:44.123456')</code></td>
<td>44123456</td>
</tr>
<tr>
<td><code>millennium(date)</code></td>
<td>Millennium</td>
<td><code>millennium(date '1992-02-15')</code></td>
<td>2</td>
</tr>
<tr>
<td><code>millisecond(date)</code></td>
<td>Sub-minute milliseconds</td>
<td><code>millisecond(timestamp '2021-08-03 11:59:44.123456')</code></td>
<td>44123</td>
</tr>
<tr>
<td><code>minute(date)</code></td>
<td>Minutes</td>
<td><code>minute(timestamp '2021-08-03 11:59:44.123456')</code></td>
<td>59</td>
</tr>
<tr>
<td><code>month(date)</code></td>
<td>Month</td>
<td><code>month(date '1992-02-15')</code></td>
<td>2</td>
</tr>
<tr>
<td><code>quarter(date)</code></td>
<td>Quarter</td>
<td><code>quarter(date '1992-02-15')</code></td>
<td>1</td>
</tr>
<tr>
<td><code>second(date)</code></td>
<td>Seconds</td>
<td><code>second(timestamp '2021-08-03 11:59:44.123456')</code></td>
<td>44</td>
</tr>
<tr>
<td><code>timezone(date)</code></td>
<td>Time Zone offset in minutes</td>
<td><code>timezone(date '1992-02-15')</code></td>
<td>0</td>
</tr>
<tr>
<td><code>timezone_hour(date)</code></td>
<td>Time zone offset hour portion</td>
<td><code>timezone_hour(date '1992-02-15')</code></td>
<td>0</td>
</tr>
<tr>
<td><code>timezone_minute(date)</code></td>
<td>Time zone offset minutes portion</td>
<td><code>timezone_minute(date '1992-02-15')</code></td>
<td>0</td>
</tr>
<tr>
<td><code>week(date)</code></td>
<td>ISO Week</td>
<td><code>week(date '1992-02-15')</code></td>
<td>7</td>
</tr>
</tbody>
</table>
Function | Description | Example | Result
--- | --- | --- | ---
weekday(date) | Numeric weekday synonym (Sunday = 0, Saturday = 6) | weekday(date '1992-02-15') | 6
weekofyear(date) | ISO Week (synonym) | weekofyear(date '1992-02-15') | 7
year(date) | Year | year(date '1992-02-15') | 1992
yearweek(date) | BIGINT of combined ISO Year number and 2-digit version of ISO Week number | yearweek(date '1992-02-15') | 199207

**Enum Functions**

This section describes functions and operators for examining and manipulating ENUM values. The examples assume an enum type created as:

```sql
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.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>enum_code(enum_value)</code></td>
<td>Returns the numeric value backing the given enum value</td>
<td><code>enum_code('happy':mood)</code></td>
<td>2</td>
</tr>
<tr>
<td><code>enum_first(enum)</code></td>
<td>Returns the first value of the input enum type.</td>
<td><code>enum_first(null::mood)</code></td>
<td>sad</td>
</tr>
<tr>
<td><code>enum_last(enum)</code></td>
<td>Returns the last value of the input enum type.</td>
<td><code>enum_last(null::mood)</code></td>
<td>anxious</td>
</tr>
<tr>
<td><code>enum_range(enum)</code></td>
<td>Returns all values of the input enum type as an array.</td>
<td><code>enum_range(null::mood)</code></td>
<td>[sad, ok, happy, anxious]</td>
</tr>
</tbody>
</table>
Function Description Example Result

enum_range_boundary(enum, enum) 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. enum_range_boundary(NULL, 'happy'::mood) [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.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>addition of an INTERVAL</td>
<td>INTERVAL 1 HOUR + INTERVAL 5 HOUR</td>
<td>INTERVAL 6 HOUR</td>
</tr>
<tr>
<td>+</td>
<td>addition to a DATE</td>
<td>DATE '1992-03-22' + INTERVAL 5 DAY</td>
<td>1992-03-27</td>
</tr>
<tr>
<td>+</td>
<td>addition to a TIMESTAMP</td>
<td>TIMESTAMP '1992-03-22 01:02:03' + INTERVAL 5 DAY</td>
<td>1992-03-27 01:02:03</td>
</tr>
<tr>
<td>+</td>
<td>addition to a TIME</td>
<td>TIME '01:02:03' + INTERVAL 5 HOUR</td>
<td>06:02:03</td>
</tr>
<tr>
<td>-</td>
<td>subtraction of an INTERVAL</td>
<td>INTERVAL 5 HOUR - INTERVAL 1 HOUR</td>
<td>INTERVAL 4 HOUR</td>
</tr>
<tr>
<td>-</td>
<td>subtraction from a TIMESTAMP</td>
<td>TIMESTAMP '1992-03-27 01:02:03' - INTERVAL 5 DAY</td>
<td>1992-03-22 01:02:03</td>
</tr>
</tbody>
</table>
### Interval Functions

The table below shows the available scalar functions for INTERVAL types.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>date_part(part, interval)</code></td>
<td>Get subfield (equivalent to <code>extract</code>)</td>
<td><code>date_part('year', INTERVAL '14 months')</code></td>
<td>1</td>
</tr>
<tr>
<td><code>datepart(part, interval)</code></td>
<td>Alias of <code>date_part</code>. Get subfield (equivalent to <code>extract</code>)</td>
<td><code>datepart('year', INTERVAL '14 months')</code></td>
<td>1</td>
</tr>
<tr>
<td><code>extract(part from interval)</code></td>
<td>Get subfield from a date</td>
<td><code>extract('month' FROM INTERVAL '14 months')</code></td>
<td>2</td>
</tr>
<tr>
<td><code>to_centuries(integer)</code></td>
<td>Construct a century interval</td>
<td><code>to_centuries(5)</code></td>
<td>INTERVAL 500 YEAR</td>
</tr>
<tr>
<td><code>to_days(integer)</code></td>
<td>Construct a day interval</td>
<td><code>to_days(5)</code></td>
<td>INTERVAL 5 DAY</td>
</tr>
<tr>
<td><code>to_decades(integer)</code></td>
<td>Construct a decade interval</td>
<td><code>to_decades(5)</code></td>
<td>INTERVAL 50 YEAR</td>
</tr>
<tr>
<td><code>to_hours(integer)</code></td>
<td>Construct a hour interval</td>
<td><code>to_hours(5)</code></td>
<td>INTERVAL 5 HOUR</td>
</tr>
<tr>
<td><code>to_microseconds(integer)</code></td>
<td>Construct a microsecond interval</td>
<td><code>to_microseconds(5)</code></td>
<td>INTERVAL 5 MICROSECOND</td>
</tr>
<tr>
<td><code>to_millennia(integer)</code></td>
<td>Construct a millenium interval</td>
<td><code>to_millennia(5)</code></td>
<td>INTERVAL 5000 YEAR</td>
</tr>
<tr>
<td><code>to_milliseconds(integer)</code></td>
<td>Construct a millisecond interval</td>
<td><code>to_milliseconds(5)</code></td>
<td>INTERVAL 5 MILLISECOND</td>
</tr>
<tr>
<td><code>to_minutes(integer)</code></td>
<td>Construct a minute interval</td>
<td><code>to_minutes(5)</code></td>
<td>INTERVAL 5 MINUTE</td>
</tr>
<tr>
<td><code>to_months(integer)</code></td>
<td>Construct a month interval</td>
<td><code>to_months(5)</code></td>
<td>INTERVAL 5 MONTH</td>
</tr>
<tr>
<td><code>to_seconds(integer)</code></td>
<td>Construct a second interval</td>
<td><code>to_seconds(5)</code></td>
<td>INTERVAL 5 SECOND</td>
</tr>
</tbody>
</table>
DuckDB Documentation

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>to_weeks(integer)</td>
<td>Construct a week interval</td>
<td>to_weeks(5)</td>
<td>INTERVAL 35 DAY</td>
</tr>
<tr>
<td>to_years(integer)</td>
<td>Construct a year interval</td>
<td>to_years(5)</td>
<td>INTERVAL 5 YEAR</td>
</tr>
</tbody>
</table>

Only the documented date parts are defined for intervals.

**Lambda Functions**

Lambda functions enable the use of more complex and flexible expressions in queries. DuckDB supports several scalar functions that accept lambda functions as parameters in the form (parameter1, parameter2, ...) -> expression. 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:

- `param -> param > 1`
- `s -> contains(concat(s, 'DB'), 'duck')`
- `(x, y) -> x + y`

**Scalar Functions That Accept Lambda Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Aliases</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>list_transform(list, lambda)</td>
<td>array_transform, apply, list_apply, array_apply</td>
<td>Returns a list that is the result of applying the lambda function to each element of the input list.</td>
<td>list_transform([4, 5, 6], x -&gt; x + 1)</td>
<td>[5, 6, 7]</td>
</tr>
<tr>
<td>list_filter(list, lambda)</td>
<td>array_filter, filter</td>
<td>Constructs a list from those elements of the input list for which the lambda function returns true.</td>
<td>list_filter([4, 5, 6], x -&gt; x &gt; 4)</td>
<td>[5, 6]</td>
</tr>
<tr>
<td>list_reduce(list, lambda)</td>
<td>array_reduce, reduce</td>
<td>Returns a single value that is the result of applying the lambda function to each element of the input list.</td>
<td>list_reduce([4, 5, 6], (x, y) -&gt; x + y)</td>
<td>15</td>
</tr>
</tbody>
</table>

**Nesting**  All scalar functions can be arbitrarily nested.

*Nested lambda functions to get all squares of even list elements:*
**SELECT** `list_transform`

- `list_filter([0, 1, 2, 3, 4, 5], x -> x % 2 = 0),
  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`

- `list_reduce([4, 5, 6], (a, b) -> a + b + x)`

  [17, 19, 21]

**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, 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], (x, i) -> x > i)`;

  [3, 5]

**Transform**

**Signature:** `list_transform(list, lambda)`

**Description:**

`list_transform` returns a list that is the result of applying the lambda function to each element of the input list.

**Aliases:**

- `array_transform`
- `apply`
- `list_apply`
- `array_apply`

**Number of parameters (excluding indexes): 1**

**Return type:** Defined by the Return type of the lambda function

**Examples:**

*Incrementing each list element by one:*

**SELECT** `list_transform([1, 2, NULL, 3], x -> x + 1);`

  [2, 3, NULL, 4]
Transforming strings:

SELECT list_transform(['duck', 'a', 'b'], s -> concat(s, 'DB'));
[duckDB, aDB, bDB]

Combining lambda functions with other functions:

SELECT list_transform([5, NULL, 6], x -> coalesce(x, 0) + 1);
[6, 1, 7]

Filter

Signature: list_filter(list, lambda)

Description:
Constructs a list from those elements of the input list for which the lambda function returns true. The lambda function must have the Return type of BOOLEAN.

Aliases:
• array_filter
• filter

Number of parameters (excluding indexes): 1

Return type: The same type as the input list

Examples:
Filter out negative values:

SELECT list_filter([5, -6, NULL, 7], x -> x > 0);
[5, 7]

Divisible by 2 and 5:

SELECT list_filter(list_filter([2, 4, 3, 1, 20, 10, 3, 30], x -> x % 2 == 0), y -> y % 5 == 0);
[20, 10, 30]

In combination with range(...) to construct lists:

SELECT list_filter([1, 2, 3, 4], x -> x > #1) FROM range(4);
[1, 2, 3, 4]
[2, 3, 4]
[3, 4]
[4]
[]
Reduce

**Signature:** `list_reduce(list, lambda)`

**Description:**
The scalar function returns a single value that is the result of applying the lambda function to each element of the input list. Starting with the first element and then repeatedly applying the lambda function to the result of the previous application and the next element of the list. The list must have at least one element.

**Aliases:**
- `array_reduce`
- `reduce`

**Number of parameters (excluding indexes):** 2

**Return type:** The underlying list type

**Examples:**

*Sum of all list elements:*

```sql
SELECT list_reduce([1, 2, 3, 4], (x, y) -> x + y);
```

10

*Only add up list elements if they are greater than 2:*

```sql
SELECT list_reduce(list_filter([1, 2, 3, 4], x -> x > 2), (x, y) -> x + y);
```

7

*Concat all list elements:*

```sql
SELECT list_reduce(['DuckDB', 'is', 'awesome'], (x, y) -> concat(x, ' ', y));
```

DuckDB is awesome

**Nested Functions**

This section describes functions and operators for examining and manipulating nested values. There are five nested data types: ARRAY, LIST, MAP, STRUCT, and UNION.

**List Functions**

In the descriptions, `l` is the three element list `[4, 5, 6].`
<table>
<thead>
<tr>
<th>Function</th>
<th>Aliases</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>list[index]</code></td>
<td></td>
<td>Bracket notation serves as an alias for <code>list_extract</code>.</td>
<td><code>l[3]</code></td>
<td>6</td>
</tr>
<tr>
<td><code>list[begin:end]</code></td>
<td></td>
<td>Bracket notation with colon is an alias for <code>list_slice</code>.</td>
<td><code>l[2:3]</code></td>
<td><code>[5, 6]</code></td>
</tr>
<tr>
<td><code>list[begin:end:step]</code></td>
<td></td>
<td><code>list_slice</code> in bracket notation with an added step feature.</td>
<td><code>l[::2]</code></td>
<td><code>[4, 6]</code></td>
</tr>
<tr>
<td><code>array_pop_back(list)</code></td>
<td></td>
<td>Returns the list without the last element.</td>
<td><code>array_pop_back(l)</code></td>
<td><code>[4, 5]</code></td>
</tr>
<tr>
<td><code>array_pop_front(list)</code></td>
<td></td>
<td>Returns the list without the first element.</td>
<td><code>array_pop_front(l)</code></td>
<td><code>[5, 6]</code></td>
</tr>
<tr>
<td><code>flatten(list_of_lists)</code></td>
<td></td>
<td>Concatenate a list of lists into a single list. This only flattens one level of the list (see examples).</td>
<td><code>flatten([[1, 2], [3, 4]])</code></td>
<td><code>[1, 2, 3, 4]</code></td>
</tr>
<tr>
<td><code>len(list)</code></td>
<td><code>array_length</code></td>
<td>Return the length of the list.</td>
<td><code>len([1, 2, 3])</code></td>
<td>3</td>
</tr>
<tr>
<td><code>list_aggregate(list, name)</code></td>
<td><code>list_aggr</code>, <code>aggregate</code>, <code>array_aggregate</code>, <code>array_aggr</code></td>
<td>Executes the aggregate function name on the elements of <code>list</code>. See the List Aggregates section for more details.</td>
<td><code>list_aggregate([1, 2, NULL], 'min')</code></td>
<td>1</td>
</tr>
<tr>
<td><code>list_any_value(list)</code></td>
<td></td>
<td>Returns the first non-null value in the list</td>
<td><code>list_any_value([NULL, -3])</code></td>
<td>-3</td>
</tr>
<tr>
<td><code>list_append(list, element)</code></td>
<td><code>array_append</code>, <code>array_push_back</code></td>
<td>Appends element to <code>list</code>.</td>
<td><code>list_append([2, 3, 4])</code></td>
<td><code>[2, 3, 4]</code></td>
</tr>
<tr>
<td><code>list_concat(list1, list2)</code></td>
<td><code>list_cat</code>, <code>array_concat</code>, <code>array_cat</code></td>
<td>Concatenates two lists.</td>
<td><code>list_concat([2, 3, [4, 5, 6]])</code></td>
<td><code>[2, 3, 4, 5, 6]</code></td>
</tr>
<tr>
<td><code>list_contains(list, element)</code></td>
<td><code>list_has</code>, <code>array_contains</code>, <code>array_has</code></td>
<td>Returns true if the list contains the element.</td>
<td><code>list_contains([1, 2, NULL, 1])</code></td>
<td><code>true</code></td>
</tr>
<tr>
<td>Function</td>
<td>Aliases</td>
<td>Description</td>
<td>Example</td>
<td>Result</td>
</tr>
<tr>
<td>----------</td>
<td>---------</td>
<td>-------------</td>
<td>---------</td>
<td>--------</td>
</tr>
<tr>
<td>list_cosine_similarity(list1, list2)</td>
<td></td>
<td>Compute the cosine similarity between two lists</td>
<td>list_cosine_similarity([1, 2, 3], [1, 2, 5])</td>
<td>0.9759000729485332</td>
</tr>
<tr>
<td>list_distance(list1, list2)</td>
<td>array_distance(distinct(list))</td>
<td>Calculates the Euclidean distance between two points with coordinates given in two inputs lists of equal length.</td>
<td>list_distance([1, 2, 3], [1, 2, 5])</td>
<td>2.0</td>
</tr>
<tr>
<td>list_distinct(list)</td>
<td>list distinct</td>
<td>Removes all duplicates and NULLs from a list. Does not preserve the original order.</td>
<td>list_distinct([1, 1, NULL, -3, 1, 5])</td>
<td>[1, 5, -3]</td>
</tr>
<tr>
<td>list_dot_product(list1, list2)</td>
<td>list_inner_product</td>
<td>Computes the dot product of two same-sized lists of numbers.</td>
<td>list_dot_product([1, 2, 3], [1, 2, 5])</td>
<td>20.0</td>
</tr>
<tr>
<td>list_extract(list, index)</td>
<td>list_element, array_extract</td>
<td>Extract the indexth (1-based) value from the list.</td>
<td>list_extract(l, 3)</td>
<td>6</td>
</tr>
<tr>
<td>list_filter(list, lambda)</td>
<td>list filter, array_filter</td>
<td>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.</td>
<td>list_filter(l, x -&gt; x &gt; 4)</td>
<td>[5, 6]</td>
</tr>
<tr>
<td>list_grade_up(list)</td>
<td>array_grade_up</td>
<td>Works like sort, but the results are the indexes that correspond to the position in the original list instead of the actual values.</td>
<td>list_grade_up([30, 10, 40, 20])</td>
<td>[2, 4, 1, 3]</td>
</tr>
<tr>
<td>list_has_all(list, sub-list)</td>
<td>array_has_all</td>
<td>Returns true if all elements of sub-list exist in list.</td>
<td>list_has_all(l, [4, 6])</td>
<td>true</td>
</tr>
<tr>
<td>list_has_any(list1, list2)</td>
<td>array_has_any</td>
<td>Returns true if any elements exist is both lists.</td>
<td>list_has_any([1, 2, 3], [2, 3, 4])</td>
<td>true</td>
</tr>
<tr>
<td>Function</td>
<td>Aliases</td>
<td>Description</td>
<td>Example</td>
<td>Result</td>
</tr>
<tr>
<td>-------------------</td>
<td>--------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>--------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>list_intersect</td>
<td>array_intersect</td>
<td>Returns a list of all the elements that exist in both l1 and l2, without duplicates.</td>
<td>list_intersect([1, 2, 3], [2, 3, 4])</td>
<td>[2, 3]</td>
</tr>
<tr>
<td>list_position</td>
<td>list_indexof, array_indexof</td>
<td>Returns the index of the element if the list contains the element.</td>
<td>list_contains([1, 2, NULL], 2)</td>
<td>2</td>
</tr>
<tr>
<td>list_prepend</td>
<td>array_prepend, array_push_front</td>
<td>Prepends element to list.</td>
<td>list_prepend(3, [4, 5, 6])</td>
<td>[3, 4, 5, 6]</td>
</tr>
<tr>
<td>list_reduce</td>
<td>array_reduce, reduce</td>
<td>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.</td>
<td>list_reduce(l, (x, y) -&gt; x + y)</td>
<td>15</td>
</tr>
<tr>
<td>list_resize</td>
<td>array_resize</td>
<td>Resizes the list to contain size elements. Initializes new elements with value or NULL if value is not set.</td>
<td>list_resize([1, 2, 3, 5, 0], 0)</td>
<td>[1, 2, 3, 5, 0]</td>
</tr>
<tr>
<td>list_reverse_sort</td>
<td>array_reverse_sort</td>
<td>Sorts the elements of the list in reverse order. See the Sorting Lists section for more details about the null sorting order.</td>
<td>list_reverse_sort([3, 6, 1, 2])</td>
<td>[6, 3, 2, 1]</td>
</tr>
<tr>
<td>list_reverse</td>
<td>array_reverse</td>
<td>Reverses the list.</td>
<td>list_reverse(l)</td>
<td>[6, 5, 4]</td>
</tr>
<tr>
<td>list_select</td>
<td>array_select</td>
<td>Returns a list based on the elements selected by the index_list.</td>
<td>list_select([10, 20, 30, 40], [1, 4])</td>
<td>[10, 40]</td>
</tr>
<tr>
<td>list_slice</td>
<td>array_slice</td>
<td>list_slice with added step feature.</td>
<td>list_slice(l, 1, 3, 2)</td>
<td>[4, 6]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Extract a sublist using slice conventions. Negative values are accepted. See slicing.</td>
<td>list_slice(l, [2, 3])</td>
<td>[5, 6]</td>
</tr>
<tr>
<td>Function</td>
<td>Aliases</td>
<td>Description</td>
<td>Example</td>
<td>Result</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------</td>
<td>----------------------------------------------------------------------------</td>
<td>-----------------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td><code>list_sort(list)</code></td>
<td><code>array_sort</code></td>
<td>Sorts the elements of the list. See the Sorting Lists section for more details about the sorting order and the null sorting order.</td>
<td><code>list_sort([3, 6, 1, 2])</code></td>
<td><code>[1, 2, 3, 6]</code></td>
</tr>
<tr>
<td><code>list_transform(list, lambda)</code></td>
<td><code>array_transform</code>, <code>list_apply</code>, <code>array_apply</code></td>
<td>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.</td>
<td><code>list_transform(l, x -&gt; x + 1)</code></td>
<td><code>[5, 6, 7]</code></td>
</tr>
<tr>
<td><code>list_unique(list)</code></td>
<td><code>array_unique</code></td>
<td>Counts the unique elements of a list.</td>
<td><code>list_unique([1, 1, NULL, -3, 1, 5])</code></td>
<td><code>3</code></td>
</tr>
<tr>
<td><code>list_value(any, ...)</code></td>
<td><code>list_pack</code></td>
<td>Create a LIST containing the argument values.</td>
<td><code>list_value(4, 5, 6)</code></td>
<td><code>[4, 5, 6]</code></td>
</tr>
<tr>
<td><code>list_where(value_list, mask_list)</code></td>
<td><code>array_where</code></td>
<td>Returns a list with theBOOLEANS in mask_list applied as a mask to the value_list.</td>
<td><code>list_where([10, 20, 30, 40], [true, false, false, true])</code></td>
<td><code>[10, 40]</code></td>
</tr>
<tr>
<td><code>list_zip(list1, list2, ...)</code></td>
<td><code>array_zip</code></td>
<td>Zips k LISTs to a new LIST whose length will be that of the longest list. Its elements are structs of k elements list_1,...,list_k. Elements missing will be replaced with NULL.</td>
<td><code>list_zip([1, 2], [3, 4], [5, 6])</code></td>
<td><code>[['list_1': 1, 'list_2': 1], ['list_1': 2, 'list_2': 3], ['list_1': 3], ['list_2': 4]]</code></td>
</tr>
</tbody>
</table>
### Function Aliases Description Example Result

<table>
<thead>
<tr>
<th>Function</th>
<th>Aliases</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>unnest(list)</td>
<td></td>
<td>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.</td>
<td>unnest([1, 2, 3])</td>
<td>1, 2, 3</td>
</tr>
</tbody>
</table>

---

### List Operators

The following operators are supported for lists:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp;</td>
<td>Alias for list_intersect</td>
<td>[1, 2, 3, 4, 5] &amp;&amp; [2, 5, 5, 6]</td>
<td>[2, 5]</td>
</tr>
<tr>
<td>@&gt;</td>
<td>Alias for list_has_all, where the list on the right of the operator is the sublist.</td>
<td>[1, 2, 3, 4] @&gt; [3, 4, 3]</td>
<td>true</td>
</tr>
<tr>
<td>&lt;@</td>
<td>Alias for list_has_all, where the list on the left of the operator is the sublist.</td>
<td>[1, 4] &lt;@ [1, 2, 3, 4]</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Alias for list_concat</td>
</tr>
<tr>
<td>&lt;=&gt;</td>
<td>Alias for list_cosine_similarity</td>
<td>[1, 2, 3] &lt;=&gt; [1, 2, 5]</td>
<td>0.9759000729485332</td>
</tr>
<tr>
<td>&lt;-&gt;</td>
<td>Alias for list_distance</td>
<td>[1, 2, 3] &lt;-&gt; [1, 2, 5]</td>
<td>2.0</td>
</tr>
</tbody>
</table>

---

### List Comprehension

Python-style list comprehension can be used to compute expressions over elements in a list. For example:

```sql
SELECT [lower(x) for x in strings] FROM VALUES (['Hello', '', 'World']) t(strings);  -- ['hello', '', 'world']
SELECT [upper(x) for x in strings if len(x) > 0] FROM VALUES (['Hello', '', 'World']) t(strings);  -- [HELLO, WORLD]
```

---

### Struct Functions
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>struct.entry</code></td>
<td>Dot notation serves as an alias for <code>struct_extract</code>.</td>
<td><code>({'i': 3, 's': 'string'}).s</code></td>
<td><code>string</code></td>
</tr>
<tr>
<td><code>struct[entry]</code></td>
<td>Bracket notation serves as an alias for <code>struct_extract</code>.</td>
<td><code>({'i': 3, 's': 'string'})['s']</code></td>
<td><code>string</code></td>
</tr>
<tr>
<td><code>row(any, ...)</code></td>
<td>Create a STRUCT containing the argument values. If the values are column references, the entry name will be the column name; otherwise it will be the string 'vN' where N is the (1-based) position of the argument.</td>
<td><code>row(i, i % 4, i / 4) {'i': 3, 'v2': 3, 'v3': 0}</code></td>
<td><code>{'i': 3, 'v2': 3, 'v3': 0}</code></td>
</tr>
<tr>
<td><code>struct_extract(struct, 'entry')</code></td>
<td>Extract the named entry from the struct.</td>
<td><code>struct_extract({'i': 3, 'v2': 3, 'v3': 0}, 'i')</code></td>
<td><code>3</code></td>
</tr>
<tr>
<td><code>struct_pack(name := any, ...)</code></td>
<td>Create a STRUCT containing the argument values. The entry name will be the bound variable name.</td>
<td><code>struct_pack(i := 4, s := 'string')</code></td>
<td><code>{'i': 4, 's': string}</code></td>
</tr>
<tr>
<td><code>struct_insert(struct, name := any, ...)</code></td>
<td>Add field(s)/value(s) to an existing STRUCT with the argument values. The entry name(s) will be the bound variable name(s).</td>
<td><code>struct_insert({'a': 1}, b := 2)</code></td>
<td><code>{'a': 1, 'b': 2}</code></td>
</tr>
</tbody>
</table>

### Map Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>map[entry]</code></td>
<td>Alias for <code>element_at</code></td>
<td><code>map([100, 5], ['a', 'b'])[100]</code></td>
<td><code>[a]</code></td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>Example</td>
<td>Result</td>
</tr>
<tr>
<td>----------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>------------------------------------------------------------------------</td>
<td>------------</td>
</tr>
<tr>
<td>element_at(map, key)</td>
<td>Return a list containing the value for a given key or an empty list 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.</td>
<td>element_at(map([100, 5], [42, 43]), 100)</td>
<td>[42]</td>
</tr>
<tr>
<td>map_extract(map, key)</td>
<td>Alias of element_at. Return a list containing the value for a given key or an empty list 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.</td>
<td>map_extract(map([100, 5], [42, 43]), 100)</td>
<td>[42]</td>
</tr>
<tr>
<td>cardinality(map)</td>
<td>Return the size of the map (or the number of entries in the map).</td>
<td>cardinality(map([4, 2], ['a', 'b']))</td>
<td>2</td>
</tr>
<tr>
<td>map_from_entries(STRUCT(k, v)[])</td>
<td>Returns a map created from the entries of the array</td>
<td>map_from_entries([{'k': 5, 'v': 'val1'}, {'k': 3, 'v': 'val2'}])</td>
<td>{5=val1, 3=val2}</td>
</tr>
<tr>
<td>map()</td>
<td>Returns an empty map.</td>
<td>map()</td>
<td>{}</td>
</tr>
<tr>
<td>map_keys(map)</td>
<td>Return a list of all keys in the map.</td>
<td>map_keys(map([100, 5], [42, 43]))</td>
<td>[100, 5]</td>
</tr>
<tr>
<td>map_values(map)</td>
<td>Return a list of all values in the map.</td>
<td>map_values(map([100, 5], [42, 43]))</td>
<td>[42, 43]</td>
</tr>
<tr>
<td>map_entries(map)</td>
<td>Return a list of struct(k, v) for each key-value pair in the map.</td>
<td>map_entries(map([100, 5], [42, 43]))</td>
<td>[{'key': 100, 'value': 42}, {'key': 5, 'value': 43}]</td>
</tr>
</tbody>
</table>

**Union Functions**
# DuckDB Documentation

## Function Description

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>union.tag</code></td>
<td>Dot notation serves as an alias for <code>union_extract</code>.</td>
<td><code>(union_value(k := 'hello')).k</code></td>
<td><code>string</code></td>
</tr>
<tr>
<td><code>union_extract(union, 'tag')</code></td>
<td>Extract the value with the named tags from the union. NULL if the tag is not currently selected</td>
<td><code>union_extract(s, 'k')</code></td>
<td><code>hello</code></td>
</tr>
<tr>
<td><code>union_value(tag := any)</code></td>
<td>Create a single member UNION containing the argument value. The tag of the value will be the bound variable name.</td>
<td><code>union_value(k := 'hello')</code></td>
<td><code>'hello'::UNION(kVARCHAR)</code></td>
</tr>
<tr>
<td><code>union_tag(union)</code></td>
<td>Retrieve the currently selected tag of the union as an <code>Enum</code>.</td>
<td><code>union_tag(union_value(k := 'foo'))</code></td>
<td><code>'k'</code></td>
</tr>
</tbody>
</table>

## Range Functions

The functions `range` and `generate_series` create a list of values in the range between `start` and `stop`. The `start` parameter is inclusive. For the `range` function, the `stop` parameter is exclusive, while for `generate_series`, it is inclusive.

Based on the number of arguments, the following variants exist:

- `range(start, stop, step)`
- `range(start, stop)`
- `range(stop)`
- `generate_series(start, stop, step)`
- `generate_series(start, stop)`
- `generate_series(stop)`

The default value of `start` is 0 and the default value of `step` is 1.

```sql
SELECT range(5);
-- [0, 1, 2, 3, 4]

SELECT range(2, 5);
-- [2, 3, 4]

SELECT range(2, 5, 3);
-- [2]

SELECT generate_series(5);
-- [0, 1, 2, 3, 4, 5]
```
DuckDB Documentation

```sql
SELECT generate_series(2, 5);
-- [2, 3, 4, 5]

SELECT generate_series(2, 5, 3);
-- [2, 5]
```

Date ranges are also supported:

```sql
SELECT * FROM range(date '1992-01-01', date '1992-03-01', interval '1' month);
```

<table>
<thead>
<tr>
<th>range</th>
</tr>
</thead>
<tbody>
<tr>
<td>1992-01-01 00:00:00</td>
</tr>
<tr>
<td>1992-02-01 00:00:00</td>
</tr>
</tbody>
</table>

**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)`
- `array_slice(list, begin, end, step)`
- `array_slice(list, begin, end, step)`
- `list[begin:end]`
- `list[begin:end:step]`

**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

```sql
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.

```sql
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
```

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_min, list_max, list_product, list_skewness, list_sum,
list_string_agg, list_mode, list_median, list_mad and list_histogram.

```sql
SELECT list_min([1, 2, -4, NULL]);
```

-- -4

```sql
SELECT list_sum([2, 4, 8, 42]);
```

-- 56

```sql
SELECT list_last([[1, 2], [NULL], [2, 10, 3]]);
```

-- [2, 10, 3]

**array_to_string**  Concatenates list/array elements using an optional delimiter.

```sql
SELECT array_to_string([1, 2, 3], '-' AS str;
```

-- 1-2-3

```sql
-- 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.

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 these 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 sort order and can be either NULLS FIRST or NULLS LAST.

```sql
-- default sort order and default NULL sort order
SELECT list_sort([1, 3, NULL, 5, NULL, -5]);
```

----

[NULL, NULL, -5, 1, 3, 5]

```sql
-- only providing the sort order
SELECT list_sort([1, 3, NULL, 2], 'ASC');
```

----

[NULL, 1, 2, 3]

```sql
-- providing the sort order and the NULL sort order
```

```sql
SELECT list_sort([1, 3, NULL, 2], 'ASC', 'NULLS LAST');
```

----

[1, 2, 3, NULL]
```
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.

-- default NULL sort order
SELECT list_sort([1, 3, NULL, 5, NULL, -5]);
----
[NULL, NULL, -5, 1, 3, 5]

-- providing the NULL sort order
SELECT list_reverse_sort([1, 3, NULL, 2], 'NULLS LAST');
----
[3, 2, 1, NULL]
```

**Lambda Functions**

DuckDB supports lambda functions in the form (parameter1, parameter2, ...) -> expression. For details, see the [lambda functions page](#).

**Flatten**

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],
    ]]);
```

554
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]

generate_subscripts

The generate_subscript( arr, dim) function generates indexes along the d'imth dimension of array arr.

SELECT generate_subscripts([4, 5, 6], 1) AS i;

   i
   ────
     1
Related Functions

There are also aggregate functions `list` and `histogram` that produces lists and lists of structs. The `unnest` function is used to unnest a list by one level.

Numeric Functions

Numeric Operators

The table below shows the available mathematical operators for numeric types.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>addition</td>
<td>2 + 3</td>
<td>5</td>
</tr>
<tr>
<td>-</td>
<td>subtraction</td>
<td>2 - 3</td>
<td>-1</td>
</tr>
<tr>
<td>*</td>
<td>multiplication</td>
<td>2 * 3</td>
<td>6</td>
</tr>
<tr>
<td>/</td>
<td>float division</td>
<td>5 / 2</td>
<td>2.5</td>
</tr>
<tr>
<td>//</td>
<td>division</td>
<td>5 // 2</td>
<td>2</td>
</tr>
<tr>
<td>%</td>
<td>modulo (remainder)</td>
<td>5 % 4</td>
<td>1</td>
</tr>
<tr>
<td>**</td>
<td>exponent</td>
<td>3 ** 4</td>
<td>81</td>
</tr>
<tr>
<td>^</td>
<td>exponent (alias for **)</td>
<td>3 ^ 4</td>
<td>81</td>
</tr>
<tr>
<td>&amp;</td>
<td>bitwise AND</td>
<td>91 &amp; 15</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bitwise OR</td>
<td>32</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>bitwise shift left</td>
<td>1 &lt;&lt; 4</td>
<td>16</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>bitwise shift right</td>
<td>8 &gt;&gt; 2</td>
<td>2</td>
</tr>
<tr>
<td>~</td>
<td>bitwise negation</td>
<td>~15</td>
<td>-16</td>
</tr>
<tr>
<td>!</td>
<td>factorial of x</td>
<td>4 !</td>
<td>24</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs(x)</td>
<td>Absolute value</td>
<td>abs(-17.4)</td>
<td>17.4</td>
</tr>
<tr>
<td>acos(x)</td>
<td>Computes the arccosine of x</td>
<td>acos(0.5)</td>
<td>1.0471975511965976</td>
</tr>
<tr>
<td>add(x, y)</td>
<td>Alias for x + y</td>
<td>add(2, 3)</td>
<td>5</td>
</tr>
<tr>
<td>asin(x)</td>
<td>Computes the arcsine of x</td>
<td>asin(0.5)</td>
<td>0.5235987755982989</td>
</tr>
<tr>
<td>atan(x)</td>
<td>Computes the arctangent of x</td>
<td>atan(0.5)</td>
<td>0.4636476090008061</td>
</tr>
<tr>
<td>atan2(y, x)</td>
<td>Computes the arctangent of (y, x)</td>
<td>atan2(0.5, 0.5)</td>
<td>0.7853981633974483</td>
</tr>
<tr>
<td>bit_count(x)</td>
<td>Returns the number of bits that are set</td>
<td>bit_count(31)</td>
<td>5</td>
</tr>
<tr>
<td>cbrt(x)</td>
<td>Returns the cube root of the number</td>
<td>cbrt(8)</td>
<td>2</td>
</tr>
<tr>
<td>ceil(x)</td>
<td>Rounds the number up</td>
<td>ceil(17.4)</td>
<td>18</td>
</tr>
<tr>
<td>ceiling(x)</td>
<td>Rounds the number up. Alias of ceil</td>
<td>ceiling(17.4)</td>
<td>18</td>
</tr>
<tr>
<td>cos(x)</td>
<td>Computes the cosine of x</td>
<td>cos(90)</td>
<td>-0.4480736161291701</td>
</tr>
<tr>
<td>cot(x)</td>
<td>Computes the cotangent of x</td>
<td>cot(0.5)</td>
<td>1.830487721712452</td>
</tr>
<tr>
<td>degrees(x)</td>
<td>Converts radians to degrees</td>
<td>degrees(pi())</td>
<td>180</td>
</tr>
<tr>
<td>divide(x, y)</td>
<td>Alias for x // y</td>
<td>divide(5, 2)</td>
<td>2</td>
</tr>
<tr>
<td>even(x)</td>
<td>Round to next even number by rounding away from zero</td>
<td>even(2.9)</td>
<td>4</td>
</tr>
<tr>
<td>exp(x)</td>
<td>Computes e ** x</td>
<td>exp(0.693)</td>
<td>2</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>Example</td>
<td>Result</td>
</tr>
<tr>
<td>--------------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>--------------</td>
<td>------------</td>
</tr>
<tr>
<td>factorial(x)</td>
<td>See <code>!</code> operator. Computes the product of the current integer and all integers below it</td>
<td>factorial(4)</td>
<td>24</td>
</tr>
<tr>
<td>fdiv(x, y)</td>
<td>Performs integer division ((x \div y)) but returns a DOUBLE value</td>
<td>fdiv(5, 2)</td>
<td>2.0</td>
</tr>
<tr>
<td>floor(x)</td>
<td>Rounds the number down</td>
<td>floor(17.4)</td>
<td>17</td>
</tr>
<tr>
<td>fmod(x, y)</td>
<td>Calculates the modulo value. Always returns a DOUBLE value</td>
<td>fmod(5, 2)</td>
<td>1.0</td>
</tr>
<tr>
<td>gamma(x)</td>
<td>Interpolation of ((x-1)) factorial (so decimal inputs are allowed)</td>
<td>gamma(5.5)</td>
<td>52.34277778455352</td>
</tr>
<tr>
<td>gcd(x, y)</td>
<td>Computes the greatest common divisor of (x) and (y)</td>
<td>gcd(42, 57)</td>
<td>3</td>
</tr>
<tr>
<td>greatest_common_divisor(x, y)</td>
<td>Computes the greatest common divisor of (x) and (y)</td>
<td>greatest_common_divisor(42, 57)</td>
<td>3</td>
</tr>
<tr>
<td>greatest(x1, x2, ...)</td>
<td>Selects the largest value</td>
<td>greatest(3, 2, 4, 4)</td>
<td>4</td>
</tr>
<tr>
<td>isfinite(x)</td>
<td>Returns true if the floating point value is finite, false otherwise</td>
<td>isfinite(5.5)</td>
<td>true</td>
</tr>
<tr>
<td>isinf(x)</td>
<td>Returns true if the floating point value is infinite, false otherwise</td>
<td>isinf('Infinity':float)</td>
<td></td>
</tr>
<tr>
<td>isnan(x)</td>
<td>Returns true if the floating point value is not a number, false otherwise</td>
<td>isnan('NaN':float)</td>
<td>true</td>
</tr>
<tr>
<td>lcm(x, y)</td>
<td>Computes the least common multiple of (x) and (y)</td>
<td>lcm(42, 57)</td>
<td>798</td>
</tr>
<tr>
<td>least_common_multiple(x, y)</td>
<td>Computes the least common multiple of (x) and (y)</td>
<td>least_common_multiple(42, 57)</td>
<td>798</td>
</tr>
<tr>
<td>least(x1, x2, ...)</td>
<td>Selects the smallest value</td>
<td>least(3, 2, 4, 4)</td>
<td>2</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>Example</td>
<td>Result</td>
</tr>
<tr>
<td>---------------</td>
<td>--------------------------------------------------</td>
<td>--------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>lgamma(x)</td>
<td>Computes the log of the gamma function</td>
<td>lgamma(2)</td>
<td>0</td>
</tr>
<tr>
<td>ln(x)</td>
<td>Computes the natural logarithm of x</td>
<td>ln(2)</td>
<td>0.693</td>
</tr>
<tr>
<td>log(x)</td>
<td>Computes the 10-log of x</td>
<td>log(100)</td>
<td>2</td>
</tr>
<tr>
<td>log10(x)</td>
<td>Alias of log. computes the 10-log of x</td>
<td>log10(1000)</td>
<td>3</td>
</tr>
<tr>
<td>log2(x)</td>
<td>Computes the 2-log of x</td>
<td>log2(8)</td>
<td>3</td>
</tr>
<tr>
<td>multiply(x, y)</td>
<td>Alias for x * y</td>
<td>multiply(2, 3)</td>
<td>6</td>
</tr>
<tr>
<td>nextafter(x, y)</td>
<td>Return the next floating point value after x in the direction of y</td>
<td>nextafter(1::float, 2::float)</td>
<td>1.0000001</td>
</tr>
<tr>
<td>pi()</td>
<td>Returns the value of pi</td>
<td>pi()</td>
<td>3.141592653589793</td>
</tr>
<tr>
<td>pow(x, y)</td>
<td>Computes x to the power of y</td>
<td>pow(2, 3)</td>
<td>8</td>
</tr>
<tr>
<td>power(x, y)</td>
<td>Alias of pow. computes x to the power of y</td>
<td>power(2, 3)</td>
<td>8</td>
</tr>
<tr>
<td>radians(x)</td>
<td>Converts degrees to radians</td>
<td>radians(90)</td>
<td>1.5707963267948966</td>
</tr>
<tr>
<td>random()</td>
<td>Returns a random number between 0 and 1</td>
<td>random()</td>
<td>various</td>
</tr>
<tr>
<td>round_even(v NUMERIC, s INT)</td>
<td>Alias of roundbankers(v, s). Round to s decimal places using the rounding half to even rule. Values s &lt; 0 are allowed</td>
<td>round_even(24.5, 0)</td>
<td>24.0</td>
</tr>
<tr>
<td>round(v NUMERIC, s INT)</td>
<td>Round to s decimal places. Values s &lt; 0 are allowed</td>
<td>round(42.4332, 2)</td>
<td>42.43</td>
</tr>
<tr>
<td>setseed(x)</td>
<td>Sets the seed to be used for the random function</td>
<td>setseed(0.42)</td>
<td></td>
</tr>
<tr>
<td>sign(x)</td>
<td>Returns the sign of x as -1, 0 or 1</td>
<td>sign(-349)</td>
<td>-1</td>
</tr>
<tr>
<td>signbit(x)</td>
<td>Returns whether the signbit is set or not</td>
<td>signbit(-0.0)</td>
<td>true</td>
</tr>
<tr>
<td>sin(x)</td>
<td>Computes the sin of x</td>
<td>sin(90)</td>
<td>0.8939966636005579</td>
</tr>
</tbody>
</table>
### Function Description Example Result

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>sqrt(x)</td>
<td>Returns the square root of the number</td>
<td>sqrt(9)</td>
<td>3</td>
</tr>
<tr>
<td>subtract(x, y)</td>
<td>Alias for x - y</td>
<td>subtract(2, 3)</td>
<td>-1</td>
</tr>
<tr>
<td>tan(x)</td>
<td>Computes the tangent of x</td>
<td>tan(90)</td>
<td>-1.995200412208242</td>
</tr>
<tr>
<td>trunc(x)</td>
<td>Truncates the number</td>
<td>trunc(17.4)</td>
<td>17</td>
</tr>
<tr>
<td>xor(x)</td>
<td>Bitwise XOR</td>
<td>xor(17, 5)</td>
<td>20</td>
</tr>
<tr>
<td>@</td>
<td>Absolute value (parentheses optional if operating on a column)</td>
<td>@(-2)</td>
<td>2</td>
</tr>
</tbody>
</table>

### 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:

- `'abc' LIKE 'abc'` -- true
- `'abc' LIKE 'a%'` -- true
- `'abc' LIKE '_b_'` -- true
- `'abc' LIKE 'c'` -- false
- `'abc' LIKE 'c%'` -- false
- `'abc' LIKE '%c'` -- true
- `'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.
DuckDB Documentation

'abc' **LIKE** '%C' -- true
'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'
'a%c' **LIKE** 'a$%c' **ESCAPE** '$' -- true
'azc' **LIKE** 'a$%c' **ESCAPE** '$' -- false
```

```
-- Case insensitive ILIKE with ESCAPE
'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.

<table>
<thead>
<tr>
<th>LIKE-style</th>
<th>PostgreSQL-style</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIKE</td>
<td>~~</td>
</tr>
<tr>
<td>NOT LIKE</td>
<td>!~~</td>
</tr>
<tr>
<td>ILIKE</td>
<td>~~~*</td>
</tr>
<tr>
<td>NOT ILIKE</td>
<td>!~~*</td>
</tr>
</tbody>
</table>

**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:

```
'abc' **SIMILAR TO** 'abc' -- true
'abc' **SIMILAR TO** 'a' -- false
'abc' **SIMILAR TO** '.*(b|d).*' -- true
'abc' **SIMILAR TO** '(b|c).*' -- false
'abc' **NOT SIMILAR TO** 'abc' -- false
```
There are also alternative characters that can be used as keywords in place of SIMILAR TO expressions. These follow POSIX syntax.

<table>
<thead>
<tr>
<th>SIMILAR TO-style</th>
<th>POSIX-style</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIMILAR TO</td>
<td>~</td>
</tr>
<tr>
<td>NOT SIMILAR TO</td>
<td>!~</td>
</tr>
</tbody>
</table>

**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). Use the question mark (?) wildcard to match any single character, and use the asterisk (*) to match zero or more characters. In addition, use 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](https://en.wikipedia.org/wiki/Glob_syntax).

Some examples:

```
'best.txt' GLOB '*.txt' -- true
'best.txt' GLOB '????.txt' -- true
'best.txt' GLOB '??.txt' -- false
'best.txt' GLOB '[abc]est.txt' -- true
'best.txt' GLOB '[a-z]est.txt' -- true

-- The bracket syntax is case sensitive
'Best.txt' GLOB '[a-z]est.txt' -- false
'Best.txt' GLOB '[a-zA-Z]est.txt' -- true

-- The ! applies to all characters within the brackets
'Best.txt' GLOB '![a-zA-Z]est.txt' -- false

-- To negate a GLOB operator, negate the entire expression
NOT 'best.txt' GLOB '*.txt' -- false
```

Three tildes (~~~~) may also be used in place of the GLOB keyword.
Glob Function to Find File Names

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('*');
```

---

Regular Expressions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>regexp_full_match</code> (<code>string</code>, <code>regex</code>)</td>
<td>Returns true if the entire string matches the regex</td>
<td><code>regexp_full_match('anabanana', '(an)*')</code></td>
<td>false</td>
</tr>
<tr>
<td><code>regexp_matches</code> (<code>string</code>, <code>pattern</code>)</td>
<td>Returns true if string contains the regex pattern, false otherwise</td>
<td><code>regexp_matches('anabanana', '(an)*')</code></td>
<td>true</td>
</tr>
<tr>
<td><code>regexp_replace</code> (<code>string</code>, <code>pattern</code>, <code>replacement</code>)</td>
<td>If string contains the regex pattern, replaces the matching part with replacement</td>
<td><code>regexp_replace('hello', '[lo]', '-')</code></td>
<td>he-lo</td>
</tr>
<tr>
<td><code>regexp_split_to_array</code> (<code>string</code>, <code>regex</code>)</td>
<td>Alias of string_split_regex. Splits the string along the regex</td>
<td><code>regexp_split_to_array('hello@world; 42', ';?@')</code></td>
<td></td>
</tr>
<tr>
<td><code>regexp_extract</code> (<code>string</code>, <code>pattern</code>, <code>idx</code>)</td>
<td>If string contains the regex pattern, returns the capturing group specified by optional parameter idx</td>
<td><code>regexp_extract('hello_world', '([a-z]_*)', 1)</code></td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>Example</td>
<td>Result</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td><code>regexp_extract(string, pattern, name_list);</code></td>
<td>If string contains the regexp pattern, returns the capturing groups as a struct with corresponding names from name_list</td>
<td><code>regexp_extract('2023-04-15', '(\d+)-(\d+)-(\d+)', ['y', 'm', 'd'])</code></td>
<td><code>{'y':'2023', 'm':'04', 'd':'15'}</code></td>
</tr>
<tr>
<td><code>regexp_extract_all(string, regex[,group=0])</code></td>
<td>Split the string along the regex and extract all occurrences of group</td>
<td><code>regexp_extract_all('hello_world', '(a-z)+_?', 1)</code></td>
<td><code>[hello, world]</code></td>
</tr>
</tbody>
</table>

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:

- `regexp_matches('abc', 'abc')` -- `true`
- `regexp_matches('abc', '^abc$')` -- `true`
- `regexp_matches('abc', 'a')` -- `true`
- `regexp_matches('abc', '^a$')` -- `false`
- `regexp_matches('abc', '.*(b|d).*')` -- `true`
- `regexp_matches('abc', '(b|c).*')` -- `true`
- `regexp_matches('abc', '^ (b|c).*')` -- `false`
- `regexp_matches('abc', '(?i)A')` -- `true`

**Options for Regular Expression Functions**  The `regexp_matches` and `regexp_replace` functions also support the following options.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'c'</td>
<td>case-sensitive matching</td>
</tr>
<tr>
<td>'i'</td>
<td>case-insensitive matching</td>
</tr>
<tr>
<td>'l'</td>
<td>match literals instead of regular expression tokens</td>
</tr>
<tr>
<td>'m', 'n', 'p'</td>
<td>newline sensitive matching</td>
</tr>
<tr>
<td>'s'</td>
<td>non-newline sensitive matching</td>
</tr>
<tr>
<td>'g'</td>
<td>global replace, only available for <code>regexp_replace</code></td>
</tr>
</tbody>
</table>

- `regexp_matches('abcd', 'ABC', 'c')` -- `false`
- `regexp_matches('abcd', 'ABC', 'i')` -- `true`
- `regexp_matches('ab^/$cd', '^/$', 'l')` -- `true`
Using `regexp_matches` The `regexp_matches` operator will be optimized to the LIKE operator when possible. To achieve best performance, the 's' option (case-sensitive matching) should be passed if applicable. Note that by default the RE2 library doesn't match '!' to newline.

<table>
<thead>
<tr>
<th>Original</th>
<th>Optimized equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>regexp_matches('hello world', 'hello', 's')</code></td>
<td>prefix('hello world', 'hello')</td>
</tr>
<tr>
<td><code>regexp_matches('hello world', 'world$', 's')</code></td>
<td>suffix('hello world', 'world')</td>
</tr>
<tr>
<td><code>regexp_matches('hello world', 'hello.world', 's')</code></td>
<td>LIKE 'hello_world'</td>
</tr>
<tr>
<td><code>regexp_matches('hello world', 'he.*rld', 's')</code></td>
<td>LIKE '%he%rld'</td>
</tr>
</tbody>
</table>

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`:

```sql
regexp_replace('abc', '(b|c)', 'X')  -- aXc
regexp_replace('abc', '(b|c)', 'X', 'g')  -- aXX
regexp_replace('abc', '(b|c)', '\1\1\1\1')  -- abbbbc
regexp_replace('abc', '(.*c)', '\1e')  -- a be
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 `idx` parameter. If `idx` is not specified, it defaults to 0, extracting the first match with the whole pattern.

```sql
regexp_extract('abc', '.b.')  -- abc
regexp_extract('abc', '.b.', @)  -- abc
regexp_extract('abc', '.b.', 1)  -- (empty)
regexp_extract('abc', '([a-z])(b)', 1)  -- a
regexp_extract('abc', '([a-z])(b)', 2)  -- b
```

If `idx` is a LIST of strings, then `regexp_extract` will return the corresponding capture groups as fields of a STRUCT:
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.

The RE2 Library

DuckDB uses RE2 as its regular expression engine. For more information see the RE2 docs

Text Functions

This section describes functions and operators for examining and manipulating string values. The symbol denotes a space character.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
<th>Alias</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>string ^@ string</code></td>
<td>Alias for <code>starts_with</code>.</td>
<td>'abc' ^@ 'a'</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>`string</td>
<td></td>
<td>string`</td>
<td>String concatenation</td>
<td>'Duck'</td>
</tr>
<tr>
<td><code>string[index]</code></td>
<td>Alias for <code>array_extract</code>.</td>
<td>'DuckDB'[4]</td>
<td>'k'</td>
<td></td>
</tr>
<tr>
<td><code>array_extract(list, index)</code></td>
<td>Extract a single character using a (1-based) index.</td>
<td>array_extract('DuckDB', 2)</td>
<td>'u'</td>
<td>list_element, list_extract</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>Example</td>
<td>Result</td>
<td>Alias</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>--------------------------------</td>
<td>------------</td>
<td>--------</td>
</tr>
<tr>
<td><code>array_slice(list, begin, end)</code></td>
<td>Extract a string using slice conventions. Negative values are accepted.</td>
<td><code>array_slice('DuckDB', 5, NULL)</code></td>
<td>'DB'</td>
<td></td>
</tr>
<tr>
<td><code>ascii(string)</code></td>
<td>Returns an integer that represents the Unicode code point of the first character of the string</td>
<td><code>ascii('Ω')</code></td>
<td>937</td>
<td></td>
</tr>
<tr>
<td><code>bar(x, min, max[, width])</code></td>
<td>Draw a band whose width is proportional to <code>(x - min)</code> and equal to <code>width</code> characters when <code>x = max</code>. <code>width</code> defaults to 80.</td>
<td><code>bar(5, 0, 20, 10)</code></td>
<td>☐▌</td>
<td></td>
</tr>
<tr>
<td><code>bit_length(string)</code></td>
<td>Number of bits in a string.</td>
<td><code>bit_length('abc')</code></td>
<td>24</td>
<td></td>
</tr>
<tr>
<td><code>chr(x)</code></td>
<td>returns a character which is corresponding the ASCII code value or Unicode code point</td>
<td><code>chr(65)</code></td>
<td>A</td>
<td></td>
</tr>
<tr>
<td><code>concat(string, ...)</code></td>
<td>Concatenate many strings together</td>
<td><code>concat('Hello', ' ', 'World')</code></td>
<td>Hello World</td>
<td></td>
</tr>
<tr>
<td><code>concat_ws(separator, string, ...)</code></td>
<td>Concatenate strings together separated by the specified separator</td>
<td><code>concat_ws(' ', 'Banana', 'Apple', 'Melon')</code></td>
<td>Banana, Apple, Melon</td>
<td></td>
</tr>
<tr>
<td><code>contains(string, search_string)</code></td>
<td>Return true if <code>search_string</code> is found within <code>string</code></td>
<td><code>contains('abc', 'a')</code></td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>Example</td>
<td>Result</td>
<td>Alias</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>----------------------------------------------</td>
<td>-------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>ends_with(string, search_string)</td>
<td>Return true if string ends with search_string</td>
<td>ends_with('abc', 'c')</td>
<td>true</td>
<td>suffix</td>
</tr>
<tr>
<td>format(format, parameters...)</td>
<td>Formats a string using the fmt syntax</td>
<td>format('Benchmark &quot;{}&quot; took {} seconds', 'CSV', 42)</td>
<td>Benchmark &quot;CSV&quot; took 42 seconds</td>
<td></td>
</tr>
<tr>
<td>format_bytes(bytes)</td>
<td>Converts bytes to a human-readable representation using units based on powers of 2 (KiB, MiB, GiB, etc.).</td>
<td>format_bytes(16384)</td>
<td>16.0 KiB</td>
<td></td>
</tr>
<tr>
<td>from_base64(string)</td>
<td>Convert a base64 encoded string to a character string.</td>
<td>from_base64('QQ==')</td>
<td>'A'</td>
<td></td>
</tr>
<tr>
<td>hash(value)</td>
<td>Returns an integer with the hash of the value</td>
<td>hash('fred')</td>
<td>2595805878642663834</td>
<td></td>
</tr>
<tr>
<td>ilike_escape(string, like_specifier, escape_character)</td>
<td>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.</td>
<td>ilike_escape('A%c', 'a$%C', '$')</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>Example</td>
<td>Result</td>
<td>Alias</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>--------------------------------</td>
<td>--------</td>
<td>-------</td>
</tr>
<tr>
<td>instr(&lt;string&gt;, &lt;search_string&gt;)</td>
<td>Return location of first occurrence of &lt;search_string&gt; in &lt;string&gt;, counting from 1. Returns 0 if no match found.</td>
<td>instr('test test', 'es')</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>left(&lt;string&gt;, &lt;count&gt;)</td>
<td>Extract the left-most count characters</td>
<td>left('Hello&gt;Lorem', 'Hello')</td>
<td>He 2</td>
<td></td>
</tr>
<tr>
<td>left_grapheme(&lt;string&gt;, &lt;count&gt;)</td>
<td>Extract the left-most grapheme clusters</td>
<td>left_grapheme('♂♀', 1)</td>
<td>♂♀</td>
<td></td>
</tr>
<tr>
<td>length(&lt;string&gt;)</td>
<td>Number of characters in &lt;string&gt;</td>
<td>length('HelloLorem')</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>length_grapheme(&lt;string&gt;)</td>
<td>Number of grapheme clusters in &lt;string&gt;</td>
<td>length_grapheme('♂♀')</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>string LIKE target</td>
<td>Returns true if the &lt;string&gt; matches the like specifier (see Pattern Matching)</td>
<td>'hello' LIKE '%lo'</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>like_escape(&lt;string&gt;, &lt;like specifier&gt;, &lt;escape character&gt;)</td>
<td>Returns true if the &lt;string&gt; matches the &lt;like specifier&gt; (see Pattern Matching) using case-sensitive matching. &lt;escape character&gt; is used to search for wildcard characters in the &lt;string&gt;.</td>
<td>like_escape('a%c', 'a$%c', '$')</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>Example</td>
<td>Result</td>
<td>Alias</td>
</tr>
<tr>
<td>---------------------</td>
<td>--------------------------------------------------</td>
<td>----------------</td>
<td>------------------</td>
<td>-------</td>
</tr>
<tr>
<td>lower(string)</td>
<td>Convert string to lowercase</td>
<td>lower('Hello')</td>
<td>hello</td>
<td>lcase</td>
</tr>
<tr>
<td>lpad(string, count, character)</td>
<td>Pads the string with the character from the left until it has count characters</td>
<td>lpad('hello', 10, '&gt;')</td>
<td>&gt;&gt;&gt;&gt;hello</td>
<td></td>
</tr>
<tr>
<td>ltrim(string)</td>
<td>Removes any spaces from the left side of the string</td>
<td>ltrim('test')</td>
<td>test</td>
<td></td>
</tr>
<tr>
<td>ltrim(string, characters)</td>
<td>Removes any occurrences of any of the characters from the left side of the string</td>
<td>ltrim('test&lt;', ')'</td>
<td>test&lt;</td>
<td></td>
</tr>
<tr>
<td>md5(value)</td>
<td>Returns the MD5 hash of the value</td>
<td>md5('123')</td>
<td>'202cb962ac59075b964b07152d234b70'</td>
<td></td>
</tr>
<tr>
<td>nfc_normalize(string)</td>
<td>Convert string to Unicode NFC normalized string. Useful for comparisons and ordering if text data is mixed between NFC normalized and not.</td>
<td>nfc_normalize('ardèch')</td>
<td>arde`ch</td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>Example</td>
<td>Result</td>
<td>Alias</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>---------</td>
<td>--------</td>
<td>-------</td>
</tr>
<tr>
<td>not_ilike__escape(\text{string}, \text{like}_\text{specifier}, \text{escape}_\text{character})</td>
<td>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.</td>
<td>not_ilike__escape('A%c', 'a$%C', '$')</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>not_like__escape(\text{string}, \text{like}_\text{specifier}, \text{escape}_\text{character})</td>
<td>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.</td>
<td>not_like__escape('a%c', 'a$%c', '$')</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>ord(\text{string})</td>
<td>Return ASCII character code of the leftmost character in a string.</td>
<td>ord('ü')</td>
<td>252</td>
<td></td>
</tr>
<tr>
<td>position(\text{search}_\text{stringinstring})</td>
<td>Return location of first occurrence of search_string in string, counting from 1. Returns 0 if no match found.</td>
<td>position('b' in 'abc')</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>Example</td>
<td>Result</td>
<td>Alias</td>
</tr>
<tr>
<td>----------------------</td>
<td>------------------------------------</td>
<td>----------------------------------------------</td>
<td>---------------------------------------------</td>
<td>-------</td>
</tr>
<tr>
<td>printf(</td>
<td>Formats a string using printf syntax</td>
<td>printf('Benchmark Took %s took %d seconds', 'CSV', 42)</td>
<td>Benchmark &quot;CSV&quot; took 42 seconds</td>
<td></td>
</tr>
<tr>
<td>format, parameters...)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>regexp_full_match(</td>
<td>Returns true if the entire string matches the regex</td>
<td>regexp_full_match('anabanana', '(an)*')</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>string, regex</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>regexp_matches(</td>
<td>Returns true if the string contains the regex pattern, false otherwise</td>
<td>regexp_matches('anabanana', '(an)*')</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>string, pattern</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>regexp_replace(</td>
<td>If string contains the regex pattern, replaces the matching part with replacement</td>
<td>regexp_replace('hello', '[lo]', '-')</td>
<td>he-lo</td>
<td></td>
</tr>
<tr>
<td>string, pattern, replacement</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>regexp_split_to_array(</td>
<td>Alias of string_split_regex. Splits the string along the regex</td>
<td>regexp_split_to_array('hello;world', ';???')</td>
<td>['hello', 'world', '42']</td>
<td></td>
</tr>
<tr>
<td>string, regex</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>regexp_extract(</td>
<td>If string contains the regex pattern, returns the capturing group specified by optional parameter idx</td>
<td>regexp_extract('hello_world', '([a-z ]+)_?', 1)</td>
<td>hello</td>
<td></td>
</tr>
<tr>
<td>string, pattern[,idx]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

DuckDB Documentation
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
<th>Alias</th>
</tr>
</thead>
<tbody>
<tr>
<td>regexp_extract()</td>
<td>If string contains the regex pattern, returns the capturing groups as a struct with corresponding names from name_list (see Pattern Matching)</td>
<td>regexp_extract('2023-04-15', '(\d+)-(\d+)-(\d+)', ['y', 'm', 'd'])</td>
<td>{'y':'2023', 'm':'04', 'd':'15'}</td>
<td></td>
</tr>
<tr>
<td>regexp_extract_all()</td>
<td>Split the string along the regex and extract all occurrences of group</td>
<td>regexp_extract_all('hello_world', '([a-z]+)_?', 1)</td>
<td>[hello, world]</td>
<td></td>
</tr>
<tr>
<td>regexp_escape()</td>
<td>Escapes special patterns to turn string into a regular expression similarly to Python's re.escape function</td>
<td>regexp_escape('<a href="https://duckdb.org">https://duckdb.org</a>')</td>
<td>https://duckdb.org</td>
<td></td>
</tr>
<tr>
<td>repeat()</td>
<td>Repeats the string count number of times</td>
<td>repeat('A', 5)</td>
<td>AAAAA</td>
<td></td>
</tr>
<tr>
<td>replace()</td>
<td>Replaces any occurrences of the source with target in string</td>
<td>replace('hello', 'l', 't')</td>
<td>he--o</td>
<td></td>
</tr>
<tr>
<td>reverse()</td>
<td>Reverses the string</td>
<td>reverse('hello')</td>
<td>olleh</td>
<td></td>
</tr>
<tr>
<td>right()</td>
<td>Extract the right-most count characters</td>
<td>right('Hello', 3)</td>
<td>lo$</td>
<td></td>
</tr>
</tbody>
</table><p>ightarrow$        |       |</p>
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
<th>Alias</th>
</tr>
</thead>
<tbody>
<tr>
<td>right_grapheme(string, count)</td>
<td>Extract the right-most count grapheme clusters</td>
<td>right_grapheme('♀♂', 1)</td>
<td>2♀♂</td>
<td></td>
</tr>
<tr>
<td>rpad(string, count, character)</td>
<td>Pads the string with the character from the right until it has count characters</td>
<td>rpad('hello', 10, '&lt;')</td>
<td>hello&lt;&lt;&lt;&lt;&lt;&lt;</td>
<td></td>
</tr>
<tr>
<td>rtrim(string)</td>
<td>Removes any spaces from the right side of the string</td>
<td>rtrim('test'.test')</td>
<td>test</td>
<td></td>
</tr>
<tr>
<td>rtrim(string, characters)</td>
<td>Removes any occurrences of any of the characters from the right side of the string</td>
<td>rtrim('test'.test', '&lt;&gt;')</td>
<td>test</td>
<td></td>
</tr>
<tr>
<td>split_part(string, separator, index)</td>
<td>Split the string along the separator and return 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).</td>
<td>split_part('a</td>
<td>b</td>
<td>c', '</td>
</tr>
<tr>
<td>starts_with(string, search_string)</td>
<td>Return true if string begins with search_string</td>
<td>starts_with('abc', 'a')</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>Example</td>
<td>Result</td>
<td>Alias</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------------------------------</td>
<td>--------</td>
<td>----------</td>
</tr>
<tr>
<td><code>string SIMILAR TO regex</code></td>
<td>Returns true if the string matches the regex; identical to <code>regexp_full_match</code> (see Pattern Matching)</td>
<td>'hello' SIMILAR TO 'l+'</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td><code>strlen(string)</code></td>
<td>Number of bytes in string</td>
<td><code>strlen('θ')</code></td>
<td>4</td>
<td></td>
</tr>
<tr>
<td><code>strpos(string, search_string)</code></td>
<td>Return location of first occurrence of search_string in string, counting from 1. Returns 0 if no match found.</td>
<td><code>strpos('test test', 'es')</code></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td><code>strip_accents(string)</code></td>
<td>Strips accents from string</td>
<td><code>strip_accents('mühleisen')</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>string_split(string, separator)</code></td>
<td>Splits the string along the separator</td>
<td><code>string_split('hello@world';world')</code></td>
<td>['hello', 'world']</td>
<td>str_split, string_to_array</td>
</tr>
<tr>
<td><code>string_split_regex(string, regex)</code></td>
<td>Splits the string along the regex</td>
<td><code>regex('hello@world;world', 42', ';?@')</code></td>
<td>['hello', '42']</td>
<td>regexp_split_to_array, str_split_regex</td>
</tr>
<tr>
<td><code>substring(string, start, length)</code></td>
<td>Extract substring of length characters starting from character start. Note that a start value of 1 refers to the first character of the string.</td>
<td><code>substring('Hello', el 2, 2)</code></td>
<td></td>
<td>substr</td>
</tr>
</tbody>
</table>
### DuckDB Documentation

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
<th>Alias</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>substring_grapheme(string, start, length)</code></td>
<td>Extract substring of length grapheme clusters starting from character start. Note that a start value of 1 refers to the first character of the string.</td>
<td><code>substring_grapheme('♂♀', 3, 2)</code></td>
<td>♂♀</td>
<td></td>
</tr>
<tr>
<td><code>to_base64(blob)</code></td>
<td>Convert a blob to a base64 encoded string.</td>
<td><code>to_base64('A': blob)</code></td>
<td>QQ==</td>
<td>base64</td>
</tr>
<tr>
<td><code>trim(string)</code></td>
<td>Removes any spaces from either side of the string.</td>
<td><code>trim('&gt;&gt;&gt;test&lt;&lt;&lt;')</code></td>
<td>test</td>
<td></td>
</tr>
<tr>
<td><code>trim(string, characters)</code></td>
<td>Removes any occurrences of any of the characters from either side of the string</td>
<td><code>trim('&gt;&gt;&gt;&gt;test&lt;&lt;', '&gt;&lt;')</code></td>
<td>test '&lt;'</td>
<td></td>
</tr>
<tr>
<td><code>unicode(string)</code></td>
<td>Returns the unicode code of the first character of the string</td>
<td><code>unicode('ü')</code></td>
<td>252</td>
<td></td>
</tr>
<tr>
<td><code>upper(string)</code></td>
<td>Convert string to upper case</td>
<td><code>upper('Hello')</code></td>
<td>HELLO</td>
<td>ucase</td>
</tr>
</tbody>
</table>

### Text Similarity Functions

These functions are used to measure the similarity of two strings using various similarity measures.
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>damerau_levenshtein(s1, s2)</code></td>
<td>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. Different case is considered different.</td>
<td><code>damerau_levenshtein('duckdb', 'udckbd')</code></td>
<td>2</td>
</tr>
<tr>
<td><code>editdist3(s1, s2)</code></td>
<td>Alias of <code>levenshtein</code> for SQLite compatibility. The minimum number of single-character edits (insertions, deletions or substitutions) required to change one string to the other. Different case is considered different.</td>
<td><code>editdist3('duck', 'db')</code></td>
<td>3</td>
</tr>
<tr>
<td><code>jaccard(s1, s2)</code></td>
<td>The Jaccard similarity between two strings. Different case is considered different. Returns a number between 0 and 1.</td>
<td><code>jaccard('duck', 'luck')</code></td>
<td>0.6</td>
</tr>
<tr>
<td><code>jaro_similarity(s1, s2)</code></td>
<td>The Jaro similarity between two strings. Different case is considered different. Returns a number between 0 and 1.</td>
<td><code>jaro_similarity('duck', 'duckdb')</code></td>
<td>0.88</td>
</tr>
</tbody>
</table>
### Function Description

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>jaro_winkler_similarity(s1, s2)</td>
<td>The Jaro-Winkler similarity between two strings. Different case is considered different. Returns a number between 0 and 1.</td>
<td>jaro_winkler_similarity('duck', 'duckdb')</td>
<td>0.93</td>
</tr>
<tr>
<td>levenshtein(s1, s2)</td>
<td>The minimum number of single-character edits (insertions, deletions or substitutions) required to change one string to the other. Different case is considered different.</td>
<td>levenshtein('duck', 'db')</td>
<td>3</td>
</tr>
<tr>
<td>mismatches(s1, s2)</td>
<td>Alias for hamming(s1, s2). The number of positions with different characters for two strings of equal length. Different case is considered different.</td>
<td>mismatches('duck', 'luck')</td>
<td>1</td>
</tr>
</tbody>
</table>

## 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
// s == "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

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>{:d}</td>
<td>integer</td>
<td>123456</td>
</tr>
<tr>
<td>Specifier</td>
<td>Description</td>
<td>Example</td>
</tr>
<tr>
<td>-----------</td>
<td>--------------------------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>{:E}</td>
<td>scientific notation</td>
<td>3.141593E+00</td>
</tr>
<tr>
<td>{:f}</td>
<td>float</td>
<td>4.560000</td>
</tr>
<tr>
<td>{:o}</td>
<td>octal</td>
<td>361100</td>
</tr>
<tr>
<td>{:s}</td>
<td>string</td>
<td>asd</td>
</tr>
<tr>
<td>{:x}</td>
<td>hexadecimal</td>
<td>1e240</td>
</tr>
<tr>
<td>{:tX}</td>
<td>integer, X is the thousand separator</td>
<td>123 456</td>
</tr>
</tbody>
</table>

**Formatting Types**

--- Integers

```
SELECT format('{} + {} = {}', 3, 5, 3 + 5); -- 3 + 5 = 8
```

--- Booleans

```
SELECT format('{} != {}', true, false); -- true != false
```

--- Format datetime values

```
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}', 123456789); -- 75bcd15
```

--- Convert to binary

```
SELECT format('{:b}', 123456789); -- 11101011011110011001010101
```

**Print Numbers with Thousand Separators**

```
SELECT format('{:,}', 123456789); -- 123,456,789
SELECT format('{:t}', 123456789); -- 123.456.789
SELECT format('{:t}', 123456789); -- 123'456'789
SELECT format('{:t}', 123456789); -- 123 456 789
SELECT format('{:tX}', 123456789); -- 123X456X789
```

**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 {}
SELECT printf('The answer is %d', 42); -- The answer is 42
// s == "The answer is 42."
-- Format a string using positional arguments '%position$formatter',
-- e.g., the second parameter as a string is encoded as '%2$s'
SELECT printf('I'd rather be %2$s than %1$s.', 'right', 'happy'); -- I'd rather be happy than right.

### Format Specifiers

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>%c</td>
<td>character code to character</td>
<td>a</td>
</tr>
<tr>
<td>%d</td>
<td>integer</td>
<td>123456</td>
</tr>
<tr>
<td>%Xd</td>
<td>integer with thousand seperator X from ',,'</td>
<td>123_456</td>
</tr>
<tr>
<td>%E</td>
<td>scientific notation</td>
<td>3.141593E+00</td>
</tr>
<tr>
<td>%f</td>
<td>float</td>
<td>4.560000</td>
</tr>
<tr>
<td>%hd</td>
<td>integer</td>
<td>123456</td>
</tr>
<tr>
<td>%hhd</td>
<td>integer</td>
<td>123456</td>
</tr>
<tr>
<td>%lld</td>
<td>integer</td>
<td>123456</td>
</tr>
<tr>
<td>%o</td>
<td>octal</td>
<td>361100</td>
</tr>
<tr>
<td>%s</td>
<td>string</td>
<td>asd</td>
</tr>
<tr>
<td>%x</td>
<td>hexadecimal</td>
<td>1e240</td>
</tr>
</tbody>
</table>

### Formatting Types

-- Integers
SELECT printf('%d + %d = %d', 3, 5, 3 + 5); -- 3 + 5 = 8
-- Booleans
SELECT printf('%s != %s', true, false); -- true != false
-- Format datetime values
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
SELECT printf('%s', BLOB '\x00hello'); -- \x00hello
-- Pad integers with 0s
SELECT printf('%04d', 33); -- 0033
-- Create timestamps from integers
SELECT printf('%02d:%02d:%02d %s', 12, 3, 16, 'AM'); -- 12:03:16 AM
-- Convert to hexadecimal
SELECT printf('%x', 123456789); -- 75bcd15
-- Convert to binary
SELECT printf('%b', 123456789);  -- 11101011011100110100010101

**Thousand Separators**

SELECT printf('%,d', 123456789);  -- 123,456,789
SELECT printf('%.%d', 123456789);  -- 123.456.789
SELECT printf('%''d', 123456789);  -- 123'456'789
SELECT printf('%_d', 123456789);  -- 123_456_789

**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.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>addition of an INTERVAL</td>
<td>TIME '01:02:03' + INTERVAL 5 HOUR</td>
<td>06:02:03</td>
</tr>
<tr>
<td>-</td>
<td>subtraction of an INTERVAL</td>
<td>TIME '06:02:03' - INTERVAL 5 HOUR</td>
<td>01:02:03</td>
</tr>
</tbody>
</table>

**Time Functions**

The table below shows the available scalar functions for TIME types.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>current_time/get_current_time()</td>
<td>Current time (start of current transaction)</td>
<td>date_diff('hour', TIME '01:02:03', TIME '06:01:03')</td>
<td>5</td>
</tr>
<tr>
<td>date_diff(part, starttime, endtime)</td>
<td>The number of partition boundaries between the times</td>
<td>date_diff('hour', TIME '01:02:03', TIME '06:01:03')</td>
<td>5</td>
</tr>
<tr>
<td>datediff(part, starttime, endtime)</td>
<td>Alias of date_diff. The number of partition boundaries between the times</td>
<td>datediff('hour', TIME '01:02:03', TIME '06:01:03')</td>
<td>5</td>
</tr>
</tbody>
</table>
### Timestamp Functions

This section describes functions and operators for examining and manipulating TIMESTAMP values.

### Timestamp Operators

The table below shows the available mathematical operators for TIMESTAMP types.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>addition of an INTERVAL</td>
<td>TIMESTAMP '1992-03-22 01:02:03' + INTERVAL 5 DAY</td>
<td>1992-03-27 01:02:03</td>
</tr>
<tr>
<td>-</td>
<td>subtraction of TIMESTAMPs</td>
<td>TIMESTAMP '1992-03-27' - TIMESTAMP '1992-03-22'</td>
<td>5 days</td>
</tr>
<tr>
<td>-</td>
<td>subtraction of an INTERVAL</td>
<td>TIMESTAMP '1992-03-27 01:02:03' - INTERVAL 5 DAY</td>
<td>1992-03-22 01:02:03</td>
</tr>
</tbody>
</table>

The only date parts that are defined for times are epoch, hours, minutes, seconds, milliseconds and microseconds.

---

**Date Part Function**

- `date_part(part, time)`
  - Get subfield (equivalent to `extract`)
  - Example: `date_part('minute', TIME '14:21:13')`
  - Result: 21

- `datepart(part, time)`
  - Alias of `date_part`. Get subfield (equivalent to `extract`)
  - Example: `datepart('minute', TIME '14:21:13')`
  - Result: 21

- `date_sub(part, starttime, endtime)`
  - The number of complete partitions between the times
  - Example: `date_sub('hour', TIME '01:02:03', TIME '06:01:03')`
  - Result: 4

- `datesub(part, starttime, endtime)`
  - 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)`
  - Get subfield from a time
  - Example: `extract('hour' FROM TIME '14:21:13')`
  - Result: 14

- `make_time(bigint, bigint, double)`
  - The time for the given parts
  - Example: `make_time(13, 34, 27.123456)`
  - Result: 13:34:27.123456
Adding to or subtracting from infinite values produces the same infinite value.

### Timestamp Functions

The table below shows the available scalar functions for TIMESTAMP values.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>age(timestamp, timestamp)</code></td>
<td>Subtract arguments, resulting in the time difference between the two timestamps</td>
<td><code>age(TIMESTAMP '2001-04-10', TIMESTAMP '1992-09-20')</code></td>
<td>8 years 6 months 20 days</td>
</tr>
<tr>
<td><code>age(timestamp)</code></td>
<td>Subtract from current date</td>
<td><code>age(TIMESTAMP '1992-09-20')</code></td>
<td>29 years 1 month 27 days</td>
</tr>
<tr>
<td><code>century(timestamp)</code></td>
<td>Extracts the century of a timestamp</td>
<td><code>century(TIMESTAMP '1992-03-22')</code></td>
<td>20</td>
</tr>
<tr>
<td><code>date_diff(part, startdate, enddate)</code></td>
<td>The number of partition boundaries between the timestamps</td>
<td><code>date_diff('hour', TIMESTAMP '1992-09-30 23:59:59', TIMESTAMP '1992-10-01 01:58:00')</code></td>
<td>2</td>
</tr>
<tr>
<td><code>datediff(part, startdate, enddate)</code></td>
<td>Alias of date_diff. The number of partition boundaries between the timestamps</td>
<td><code>datediff('hour', TIMESTAMP '1992-09-30 23:59:59', TIMESTAMP '1992-10-01 01:58:00')</code></td>
<td>2</td>
</tr>
<tr>
<td><code>date_part(part, timestamp)</code></td>
<td>Get subfield (equivalent to extract)</td>
<td><code>date_part('minute', TIMESTAMP '1992-09-20 20:38:40')</code></td>
<td>38</td>
</tr>
<tr>
<td><code>datepart(part, timestamp)</code></td>
<td>Alias of date_part. Get subfield (equivalent to extract)</td>
<td><code>datepart('minute', TIMESTAMP '1992-09-20 20:38:40')</code></td>
<td>38</td>
</tr>
<tr>
<td><code>date_part([part, ...],timestamp)</code></td>
<td>Get the listed subfields as a struct. The list must be constant.</td>
<td><code>date_part(['year', 'month', 'day'], TIMESTAMP '1992-09-20 20:38:40')</code></td>
<td><code>{year: 1992, month: 9, day: 20}</code></td>
</tr>
<tr>
<td><code>datepart([part, ...],timestamp)</code></td>
<td>Alias of date_part. Get the listed subfields as a struct. The list must be constant.</td>
<td><code>datepart(['year', 'month', 'day'], TIMESTAMP '1992-09-20 20:38:40')</code></td>
<td><code>{year: 1992, month: 9, day: 20}</code></td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>Example</td>
<td>Result</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>------------------------------------</td>
<td>-------------------------------------------------------------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>date_sub(\textit{part}, \textit{startdate}, \textit{enddate})</td>
<td>The number of complete partitions between the timestamps</td>
<td>date_sub('hour', TIMESTAMP '1992-09-30 23:59:59', TIMESTAMP '1992-10-01 01:58:00')</td>
<td>1</td>
</tr>
<tr>
<td>datesub(\textit{part}, \textit{startdate}, \textit{enddate})</td>
<td>Alias of \textit{date_sub}. The number of complete partitions between the timestamps</td>
<td>datesub('hour', TIMESTAMP '1992-09-30 23:59:59', TIMESTAMP '1992-10-01 01:58:00')</td>
<td>1</td>
</tr>
<tr>
<td>date_trunc(\textit{part}, \textit{timestamp})</td>
<td>Truncate to specified precision</td>
<td>date_trunc('hour', TIMESTAMP '1992-09-20 20:38:40')</td>
<td>1992-09-20 20:00:00</td>
</tr>
<tr>
<td>datetrunc(\textit{part}, \textit{timestamp})</td>
<td>Alias of \textit{date_trunc}. Truncate to specified precision</td>
<td>datetrunc('hour', TIMESTAMP '1992-09-20 20:38:40')</td>
<td>1992-09-20 20:00:00</td>
</tr>
<tr>
<td>dayname(\textit{timestamp})</td>
<td>The (English) name of the weekday</td>
<td>dayname(TIMESTAMP '1992-03-22')</td>
<td>Sunday</td>
</tr>
<tr>
<td>epoch(\textit{timestamp})</td>
<td>Converts a timestamp to seconds since the epoch</td>
<td>epoch('2022-11-07 08:43:04'::TIMESTAMP);</td>
<td>1667810584</td>
</tr>
<tr>
<td>epoch_ms(\textit{timestamp})</td>
<td>Converts a timestamp to milliseconds since the epoch</td>
<td>epoch_ms('2022-11-07 08:43:04.123456'::TIMESTAMP);</td>
<td>1667810584123</td>
</tr>
<tr>
<td>epoch_ms(\textit{ms})</td>
<td>Converts ms since epoch to a timestamp</td>
<td>epoch_ms(701222400000)</td>
<td>1992-03-22 00:00:00</td>
</tr>
<tr>
<td>epoch_ds(\textit{timestamp})</td>
<td>Return the total number of milliseconds since the epoch</td>
<td>epoch_ds(timestamp '2021-08-03 11:59:44.123456')</td>
<td>1627991984123</td>
</tr>
<tr>
<td>epoch_us(\textit{timestamp})</td>
<td>Return the total number of microseconds since the epoch</td>
<td>epoch_us(timestamp '2021-08-03 11:59:44.123456')</td>
<td>1627991984123456</td>
</tr>
<tr>
<td>epoch_ns(\textit{timestamp})</td>
<td>Return the total number of nanoseconds since the epoch</td>
<td>epoch_ns(timestamp '2021-08-03 11:59:44.123456')</td>
<td>1627991984123456000</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>Example</td>
<td>Result</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>----------------------------------</td>
<td>-------------------------------------------------------------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td><code>extract(field from timestamp)</code></td>
<td>Get subfield from a timestamp</td>
<td><code>extract('hour' FROM TIMESTAMP '1992-09-20 20:38:48')</code></td>
<td>20</td>
</tr>
<tr>
<td><code>isfinite(timestamp)</code></td>
<td>Returns true if the timestamp is finite, false otherwise</td>
<td><code>isfinite(TIMESTAMP '1992-03-07')</code></td>
<td>true</td>
</tr>
<tr>
<td><code>isinf(timestamp)</code></td>
<td>Returns true if the timestamp is infinite, false otherwise</td>
<td><code>isinf(TIMESTAMP '-infinity')</code></td>
<td>true</td>
</tr>
<tr>
<td><code>last_day(timestamp)</code></td>
<td>The last day of the month.</td>
<td><code>last_day(TIMESTAMP '1992-03-22 01:02:03.1234')</code></td>
<td>1992-03-31</td>
</tr>
<tr>
<td><code>least(timestamp, timestamp)</code></td>
<td>The earlier of two timestamps</td>
<td><code>least(TIMESTAMP '1992-09-20 20:38:48', TIMESTAMP '1992-03-22 01:02:03.1234')</code></td>
<td>1992-03-22 01:02:03.1234</td>
</tr>
<tr>
<td><code>make_timestamp(microseconds)</code></td>
<td>The timestamp for the given number of μs since the epoch</td>
<td><code>make_timestamp(1667810584123456)</code></td>
<td>2022-11-07 04:43:04.123456</td>
</tr>
<tr>
<td><code>monthname(timestamp)</code></td>
<td>The (English) name of the month.</td>
<td><code>monthname(TIMESTAMP '1992-09-20')</code></td>
<td>September</td>
</tr>
<tr>
<td><code>strftime(timestamp, format)</code></td>
<td>Converts timestamp to string according to the format string</td>
<td><code>strftime(timestamp '1992-01-01 20:38:40', '%a, %d %B %Y - %I:%M:%S %p')</code></td>
<td>Wed, 1 January 1992 - 08:38:40 PM</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>Example</td>
<td>Result</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>---------</td>
<td>--------</td>
</tr>
<tr>
<td><code>strptime(text, format)</code></td>
<td>Converts string to timestamp according to the format string. Throws on failure.</td>
<td><code>strptime('Wed, 1 January 1992 - 08:38:40 PM', '%a, %d %B %Y - %I:%M:%S %p')</code></td>
<td>1992-01-01 20:38:40</td>
</tr>
<tr>
<td><code>strptime(text, format-list)</code></td>
<td>Converts string to timestamp applying the format strings in the list until one succeeds. Throws on failure.</td>
<td><code>strptime('4/15/2023 10:56:00', ['%d/%m/%Y %H:%M:%S', '%m/%d/%Y %H:%M:%S'])</code></td>
<td>2023-04-15 10:56:00</td>
</tr>
<tr>
<td><code>time_bucket(bucket_width, times-tamp[,origin])</code></td>
<td>Truncate timestamp by the specified interval bucket_width. Buckets are aligned relative to origin timestamp. origin defaults to 2000-01-03 00:00:00 for buckets that don’t include a month or year interval, and to 2000-01-01 00:00:00 for month and year buckets.</td>
<td><code>time_bucket(INTERVAL '2 weeks', TIMESTAMP '1992-04-20 15:26:00', TIMESTAMP '1992-04-01 00:00:00')</code></td>
<td>1992-04-15 00:00:00</td>
</tr>
<tr>
<td><code>time_bucket(bucket_width, times-tamp[,offset])</code></td>
<td>Truncate timestamp by the specified interval bucket_width. Buckets are offset by offset interval.</td>
<td><code>time_bucket(INTERVAL '10 minutes', TIMESTAMP '1992-04-20 15:26:00-07', INTERVAL '5 minutes')</code></td>
<td>1992-04-20 15:25:00</td>
</tr>
<tr>
<td><code>to_timestamp(double)</code></td>
<td>Converts seconds since the epoch to a timestamp with time zone</td>
<td><code>to_timestamp(1284352323.5)</code></td>
<td><code>2010-09-13 04:32:03.5+00</code></td>
</tr>
<tr>
<td><code>try_strptime(text, format)</code></td>
<td>Converts string to timestamp according to the format string. Returns NULL on failure.</td>
<td><code>try_strptime('Wed, 1 January 1992 - 08:38:40 PM', '%a, %d %B %Y - %I:%M:%S %p')</code></td>
<td>1992-01-01 20:38:40</td>
</tr>
</tbody>
</table>
### Function Description Example Result

**try_strptime(  
   text,  
   format-list)  
** Converts string to timestamp applying the format strings in the list until one succeeds. Returns NULL on failure.

try_strptime('4/15/2023 10:56:00', ['%d/%m/%Y %H:%M:%S', '%m/%d/%Y %H:%M:%S'])

2023-04-15 10:56:00

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.

### Timestamp Table Functions

The table below shows the available table functions for TIMESTAMP types.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>generate_series(</td>
<td></td>
<td></td>
</tr>
<tr>
<td>timestamp, timestamp,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>interval)</td>
<td>Generate a table of timestamps in the closed range,</td>
<td>generate_series(TIMESTAMP '2001-04-10', TIMESTAMP '2001-04-11',</td>
</tr>
<tr>
<td></td>
<td>stepping by the interval</td>
<td>INTERVAL 30 MINUTE)</td>
</tr>
<tr>
<td>range(</td>
<td>Generate a table of timestamps in the half open range,</td>
<td>range(TIMESTAMP '2001-04-10', TIMESTAMP '2001-04-11', INTERVAL 30</td>
</tr>
<tr>
<td>timestamp,</td>
<td>stepping by the interval</td>
<td>MINUTE)</td>
</tr>
<tr>
<td>timestamp, interval)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Infinite values are not allowed as table function bounds.

### Timestamp with Time Zone Functions

This section describes functions and operators for examining and manipulating TIMESTAMP WITH TIME ZONE (or TIMESTAMPTZ) values.

Despite the name, these values do not store a time zone - just an instant like TIMESTAMP. Instead, they request that the instant be binned and formatted using the current time zone.

Time zone support is not built in but can be provided by an extension, such as the ICU extension that ships with DuckDB.
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.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>current_timestamp</td>
<td>Current date and time (start of current transaction)</td>
<td>current_timestamp</td>
<td>2022-10-08 12:44:46.122-07</td>
</tr>
<tr>
<td>get_current_timestamp()</td>
<td>Current date and time (start of current transaction)</td>
<td>get_current_timestamp()</td>
<td>2022-10-08 12:44:46.122-07</td>
</tr>
<tr>
<td>greatest(TIMESTAMPTZ,</td>
<td>The later of two timestamps</td>
<td>greatest(TIMESTAMPTZ</td>
<td>1992-09-20 20:38:48-07</td>
</tr>
<tr>
<td>TIMESTAMPTZ)</td>
<td></td>
<td>'1992-09-20 20:38:48',</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>TIMESTAMPTZ '1992-03-22</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>01:02:03.1234'</td>
<td></td>
</tr>
<tr>
<td>isfinite(TIMESTAMPTZ</td>
<td>Returns true if the timestamp with time zone is finite, false otherwise</td>
<td>isfinite(TIMESTAMPTZ</td>
<td>true</td>
</tr>
<tr>
<td>'1992-03-07')</td>
<td></td>
<td>'1992-03-07')</td>
<td></td>
</tr>
<tr>
<td>isinf(TIMESTAMPTZ</td>
<td>Returns true if the timestamp with time zone is infinite, false otherwise</td>
<td>isinf(TIMESTAMPTZ</td>
<td>true</td>
</tr>
<tr>
<td>'-infinity')</td>
<td></td>
<td>'-infinity')</td>
<td></td>
</tr>
<tr>
<td>least(TIMESTAMPTZ,</td>
<td>The earlier of two timestamps</td>
<td>least(TIMESTAMPTZ</td>
<td>1992-03-22 01:02:03.1234-08</td>
</tr>
<tr>
<td>TIMESTAMPTZ)</td>
<td></td>
<td>'1992-09-20 20:38:48',</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>TIMESTAMPTZ '1992-03-22</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>01:02:03.1234'</td>
<td></td>
</tr>
<tr>
<td>now()</td>
<td>Current date and time (start of current transaction)</td>
<td>now()</td>
<td>2022-10-08 12:44:46.122-07</td>
</tr>
</tbody>
</table>
### Function Description Example Result

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>transaction_timestamp()</td>
<td>Current date and time (start of current transaction)</td>
<td>transaction_timestamp()</td>
<td>2022-10-08</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>12:44:46.122-07</td>
</tr>
</tbody>
</table>

### 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:

```sql
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:

```sql
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.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>addition of an INTERVAL</td>
<td>TIMESTAMPTZ '1992-03-22 01:02:03' + INTERVAL 5 DAY</td>
<td>1992-03-27</td>
</tr>
<tr>
<td>-</td>
<td>subtraction of TIMESTAMPTZs</td>
<td>TIMESTAMPTZ '1992-03-22' - TIMESTAMPTZ '1992-03-27'</td>
<td>5 days</td>
</tr>
</tbody>
</table>

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.
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>age(TIMESTAMPTZ '2001-04-10', TIMESTAMPTZ '1992-09-20')</code></td>
<td>Subtract arguments, resulting in the time difference between the two timestamps</td>
<td><code>age(TIMESTAMPTZ '2001-04-10', TIMESTAMPTZ '1992-09-20')</code></td>
<td>8 years 6 months 20 days</td>
</tr>
<tr>
<td><code>age(TIMESTAMP '1992-09-20')</code></td>
<td>Subtract from current date</td>
<td><code>age(TIMESTAMP '1992-09-20')</code></td>
<td>29 years 1 month 27 days 12:39:00.844</td>
</tr>
<tr>
<td><code>date_part('minute', TIMESTAMPTZ '1992-09-20 20:38:40')</code></td>
<td>Get subfield (equivalent to extract)</td>
<td><code>date_part('minute', TIMESTAMPTZ '1992-09-20 20:38:40')</code></td>
<td>38</td>
</tr>
<tr>
<td><code>datepart('minute', TIMESTAMPTZ '1992-09-20 20:38:40')</code></td>
<td>Alias of date_part. Get subfield (equivalent to extract)</td>
<td><code>datepart('minute', TIMESTAMPTZ '1992-09-20 20:38:40')</code></td>
<td>38</td>
</tr>
<tr>
<td><code>date_part(['year', 'month', 'day'], TIMESTAMPTZ '1992-09-20 20:38:40-07')</code></td>
<td>Get the listed subfields as a struct. The list must be constant.</td>
<td><code>date_part(['year', 'month', 'day'], TIMESTAMPTZ '1992-09-20 20:38:40-07')</code></td>
<td><code>{year: 1992, month: 9, day: 20}</code></td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>Example</td>
<td>Result</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>---------</td>
<td>--------</td>
</tr>
<tr>
<td><code>datepart([part,...],timestamptz)</code></td>
<td>Alias of <code>date_part</code>. Get the listed <strong>subfields</strong> as a struct. The list must be constant.</td>
<td><code>datepart(['year', 'month', 'day'], Timestamptz '1992-09-20 20:38:40-07')</code></td>
<td><code>{year: 1992, month: 9, day: 20}</code></td>
</tr>
<tr>
<td><code>date_sub(part,startdate,enddate)</code></td>
<td>The number of complete <strong>partitions</strong> between the timestamps</td>
<td><code>date_sub('hour', Timestamptz '1992-09-30 23:59:59', Timestamptz '1992-10-01 01:58:00')</code></td>
<td>1</td>
</tr>
<tr>
<td><code>datesub(part,startdate,enddate)</code></td>
<td>Alias of <code>date_sub</code>. The number of complete <strong>partitions</strong> between the timestamps</td>
<td><code>datesub('hour', Timestamptz '1992-09-30 23:59:59', Timestamptz '1992-10-01 01:58:00')</code></td>
<td>1</td>
</tr>
<tr>
<td><code>date_trunc(part,timestamptz)</code></td>
<td>Truncate to specified precision</td>
<td><code>date_trunc('hour', Timestamptz '1992-09-20 20:38:40')</code></td>
<td><code>1992-09-20 20:00:00</code></td>
</tr>
<tr>
<td><code>datetrunc(part,timestamptz)</code></td>
<td>Alias of <code>date_trunc</code>. Truncate to specified precision</td>
<td><code>datetrunc('hour', Timestamptz '1992-09-20 20:38:40')</code></td>
<td><code>1992-09-20 20:00:00</code></td>
</tr>
<tr>
<td><code>extract(field from timestamptz)</code></td>
<td>Get <strong>subfield</strong> from a timestamp with time zone</td>
<td><code>extract('hour' FROM Timestamptz '1992-09-20 20:38:48')</code></td>
<td>20</td>
</tr>
<tr>
<td><code>epoch_ms(timestamptz)</code></td>
<td>Converts a timestamp to milliseconds since the epoch</td>
<td><code>epoch_ms('2022-11-07 08:43:04.123456+00':::Timestamptz);</code></td>
<td><code>1667810584123</code></td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>Example</td>
<td>Result</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>--------------------------------------------------</td>
<td>---------------------------------------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td><code>epoch_us(timestamptz)</code></td>
<td>Converts a timestamptz to microseconds since the epoch</td>
<td><code>epoch_us('2022-11-07 08:43:04.123456+00 '::TIMESTAMPTZ);</code></td>
<td>1667810584123456</td>
</tr>
<tr>
<td><code>epoch_ns(timestamptz)</code></td>
<td>Converts a timestamptz to nanoseconds since the epoch</td>
<td><code>epoch_ns('2022-11-07 08:43:04.123456+00 '::TIMESTAMPTZ);</code></td>
<td>1667810584123456000</td>
</tr>
<tr>
<td><code>last_day(timestamptz)</code></td>
<td>The last day of the month.</td>
<td><code>last_day(TIMESTAMPTZ '1992-03-22 01:02:03.1234')</code></td>
<td>1992-03-31</td>
</tr>
<tr>
<td><code>make_timestamptz(bigint, bigint, bigint, bigint, bigint, double)</code></td>
<td>The timestamp with time zone for the given parts in the current time zone</td>
<td><code>make_timestamptz(1992, 9, 20, 13, 34, 27.123456)</code></td>
<td>1992-09-20 13:34:27.123456-07</td>
</tr>
<tr>
<td><code>make_timestamptz(microseconds)</code></td>
<td>The timestamp with time zone for the given μs since the epoch</td>
<td><code>make_timestamptz(1667810584123456)</code></td>
<td>2022-11-07 08:43:04.123456-08</td>
</tr>
<tr>
<td><code>make_timestamptz(bigint, bigint, bigint, bigint, bigint, double, string)</code></td>
<td>The timestamp with time zone for the given parts and time zone</td>
<td><code>make_timestamptz(1992, 9, 20, 15, 34, 27.123456, 'CET')</code></td>
<td>1992-09-20 06:34:27.123456-07</td>
</tr>
<tr>
<td><code>strftime(timestamptz, format)</code></td>
<td>Converts timestamp with time zone to string according to the format string</td>
<td><code>strftime(timestamptz '1992-01-01 20:38:40', '%a, %-d %B %Y - %I:%M:%S %p')</code></td>
<td>Wed, 1 January 1992 - 08:38:40 PM</td>
</tr>
<tr>
<td><code>strptime(text, format)</code></td>
<td>Converts string to timestamp with time zone according to the format string if %Z is specified.</td>
<td><code>strptime('Wed, 1 January 1992 - 08:38:40 PST', '%a, %-d %B %Y - %H:%M:%S %Z')</code></td>
<td>1992-01-01 08:38:40-08</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>Example</td>
<td>Result</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>---------</td>
<td>--------</td>
</tr>
<tr>
<td><code>time_bucket(bucket_width, times-tamptz[,origin])</code></td>
<td>Truncate <code>timestamptz</code> by the specified interval <code>bucket_width</code>. Buckets are aligned relative to origin <code>timestamptz</code>. <code>origin</code> defaults to 2000-01-03 00:00:00+00 for buckets that don't include a month or year interval, and to 2000-01-01 00:00:00+00 for month and year buckets.</td>
<td><code>time_bucket(INTERVAL '2 weeks', TIMESTAMPTZ '1992-04-20 15:26:00-07', TIMESTAMPTZ '1992-04-01 00:00:00-07')</code></td>
<td>1992-04-15 00:00:00-07</td>
</tr>
<tr>
<td><code>time_bucket(bucket_width, times-tamptz[,offset])</code></td>
<td>Truncate <code>timestamptz</code> by the specified interval <code>bucket_width</code>. Buckets are offset by <code>offset</code> interval.</td>
<td><code>time_bucket(INTERVAL '10 minutes', TIMESTAMPTZ '1992-04-20 15:26:00-07', INTERVAL '5 minutes')</code></td>
<td>1992-04-20 15:25:00-07</td>
</tr>
<tr>
<td><code>time_bucket(bucket_width, times-tamptz[,timezone])</code></td>
<td>Truncate <code>timestamptz</code> by the specified interval <code>bucket_width</code>. Bucket starts and ends are calculated using <code>timezone</code>. <code>timezone</code> is a varchar and defaults to UTC.</td>
<td><code>time_bucket(INTERVAL '2 days', TIMESTAMPTZ '1992-04-20 15:26:00-07', 'Europe/Berlin')</code></td>
<td>1992-04-19 15:00:00-07</td>
</tr>
</tbody>
</table>

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.
**Function Description Example**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>generate_series(timestamptz, timestamptz, interval)</code></td>
<td>Generate a table of timestamps in the closed range (including both the starting timestamp and the ending timestamp), stepping by the interval</td>
<td><code>generate_series('2001-04-10', '2001-04-11', INTERVAL 30 MINUTE)</code></td>
</tr>
<tr>
<td><code>range(timestamptz, timestamptz, interval)</code></td>
<td>Generate a table of timestamps in the half open range (including the starting timestamp, but stopping before the ending timestamp), stepping by the interval</td>
<td><code>range('2001-04-10', '2001-04-11', INTERVAL 30 MINUTE)</code></td>
</tr>
</tbody>
</table>

Infinite values are not allowed as table function bounds.

**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.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>current_localtime()</code></td>
<td>Returns a <code>TIME</code> whose GMT bin values correspond to local time in the current time zone.</td>
<td><code>current_localtime()</code></td>
<td><code>08:47:56.497</code></td>
</tr>
<tr>
<td><code>current_localtimestamp()</code></td>
<td>Returns a <code>TIMESTAMP</code> whose GMT bin values correspond to local date and time in the current time zone.</td>
<td><code>current_localtimestamp()</code></td>
<td><code>2022-12-17 08:47:56.497</code></td>
</tr>
<tr>
<td><code>localtime</code></td>
<td>Synonym for the <code>current_localtime()</code> function call.</td>
<td><code>localtime</code></td>
<td><code>2022-12-17 08:47:56.497</code></td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>Example</td>
<td>Result</td>
</tr>
<tr>
<td>------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-----------------------------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>localtimestamp</td>
<td>Synonym for the current_localtimestamp() function call.</td>
<td>localtimestamp</td>
<td>2022-12-17 08:47:56.497</td>
</tr>
<tr>
<td>timezone(text, timestamp)</td>
<td>Use the date parts of the timestamp in GMT to construct a timestamp in the given time zone. Effectively, the argument is a &quot;local&quot; time.</td>
<td>timezone('America/Denver2001-02-16 TIMESTAMP '2001-02-16 19:38:40-08 20:38:40')</td>
<td>2001-02-16 19:38:40-08</td>
</tr>
<tr>
<td>timezone(text, timestamp)</td>
<td>Use the date parts of the timestamp in the given time zone to construct a timestamp. Effectively, the result is a &quot;local&quot; time.</td>
<td>timezone('America/Denver2001-02-16 TIMESTAMP '2001-02-16 18:38:40-05')</td>
<td>2001-02-16 18:38:40</td>
</tr>
</tbody>
</table>

**At Time Zone**  
The AT TIME ZONE syntax is syntactic sugar for the (two argument) timezone function listed above:

```sql
timestamp '2001-02-16 20:38:40' AT TIME ZONE 'America/Denver';
-- 2001-02-16 19:38:40-08
timestamp with time zone '2001-02-16 20:38:40-05' AT TIME ZONE 'America/Denver';
-- 2001-02-16 18:38:40
```

**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.
## Utility Functions

### Utility Functions

The functions below are difficult to categorize into specific function types and are broadly useful.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>alias(\textit{column})</td>
<td>Return the name of the column</td>
<td>alias(column1)</td>
<td>'column1'</td>
</tr>
<tr>
<td>checkpoint(\textit{database})</td>
<td>Synchronize WAL with file for (optional) database without interrupting transactions.</td>
<td>checkpoint(my_db)</td>
<td>success boolean</td>
</tr>
<tr>
<td>coalesce(\textit{expr}, ...)</td>
<td>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.</td>
<td>coalesce(NULL, NULL, 'default_string')</td>
<td>'default_string'</td>
</tr>
<tr>
<td>constant_or_null(\textit{arg1}, \textit{arg2})</td>
<td>If \textit{arg2} is NULL, return NULL. Otherwise, return \textit{arg1}.</td>
<td>constant_or_null(42, NULL)</td>
<td>NULL</td>
</tr>
<tr>
<td>count_if(\textit{x})</td>
<td>Returns 1 if \textit{x} is true or a non-zero number</td>
<td>count_if(42)</td>
<td>1</td>
</tr>
<tr>
<td>error(\textit{message})</td>
<td>Throws the given error \textit{message}</td>
<td>error('access_mode')</td>
<td></td>
</tr>
<tr>
<td>ifnull(\textit{expr}, \textit{other})</td>
<td>A two-argument version of coalesce</td>
<td>ifnull(NULL, 'default_string')</td>
<td>'default_string'</td>
</tr>
<tr>
<td>nullif(\textit{a}, \textit{b})</td>
<td>Return null if \textit{a} = \textit{b}, else return \textit{a}. Equivalent to CASE WHEN \textit{a} = \textit{b} THEN NULL ELSE \textit{a} END.</td>
<td>nullif(1+1, 2)</td>
<td>NULL</td>
</tr>
</tbody>
</table>
## DuckDB Documentation

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>current_schema()</code></td>
<td>Return the name of the currently active schema. Default is main.</td>
<td><code>current_schema()</code></td>
<td>'main'</td>
</tr>
<tr>
<td><code>current_schemas()</code></td>
<td>Return list of schemas. Pass a parameter of true to include implicit schemas.</td>
<td><code>current_schemas(true)</code></td>
<td>['temp', 'main', 'pg_catalog']</td>
</tr>
<tr>
<td><code>current_setting()</code></td>
<td>Return the current value of the configuration setting</td>
<td><code>current_setting('access_mode')</code></td>
<td>'automatic'</td>
</tr>
<tr>
<td><code>currval()</code></td>
<td>Return the current value of the sequence. Note that <code>nextval</code> must be called at least once prior to calling <code>currval</code>.</td>
<td><code>currval('my_sequence_name')</code></td>
<td>1</td>
</tr>
<tr>
<td><code>force_checkpoint()</code></td>
<td>Synchronize WAL with file for (optional) database interrupting transactions.</td>
<td><code>force_checkpoint(my_db)</code></td>
<td>success boolean</td>
</tr>
<tr>
<td><code>gen_random_uuid()</code></td>
<td>Alias of <code>uuid</code>. Return a random uuid similar to this: <code>eccccb8c5-9943-b2bb-bb5e-222f4e14b687</code>.</td>
<td><code>gen_random_uuid()</code></td>
<td>various</td>
</tr>
<tr>
<td><code>hash(value)</code></td>
<td>Returns an integer with the hash of the <code>value</code></td>
<td><code>hash('123')</code></td>
<td>2595805878642663834</td>
</tr>
<tr>
<td><code>icu_sort_key()</code></td>
<td>Surrogate key used to sort special characters according to the specific locale. Collator parameter is optional. Valid only when ICU extension is installed.</td>
<td><code>icu_sort_key('ö', 'DE')</code></td>
<td>460145960106</td>
</tr>
<tr>
<td><code>md5(string)</code></td>
<td>Return an md5 one-way hash of the <code>string</code>.</td>
<td><code>md5('123')</code></td>
<td>'202cb962ac59075b964b07152d2'</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>Example</td>
<td>Result</td>
</tr>
<tr>
<td>-----------------------</td>
<td>------------------------------------------------------------------</td>
<td>----------------------------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>nextval('sequence_name')</td>
<td>Return the following value of the sequence.</td>
<td>nextval('my_sequence_name')</td>
<td>2</td>
</tr>
<tr>
<td>pg_typeof(expression)</td>
<td>Returns the lower case name of the data type of the result of the expression. For PostgreSQL compatibility.</td>
<td>pg_typeof('abc')</td>
<td>'varchar'</td>
</tr>
<tr>
<td>repeat_row(varargs, num_rows)</td>
<td>Returns a table with num_rows rows, each containing the fields defined in varargs</td>
<td>repeat_row(1, 2, 'foo', num_rows = 3)</td>
<td>3 rows of 1, 2, 'foo'</td>
</tr>
<tr>
<td>stats(expression)</td>
<td>Returns a string with statistics about the expression. Expression can be a column, constant, or SQL expression.</td>
<td>stats(5)</td>
<td>'['Min: 5, Max: 5][Has Null: false]'</td>
</tr>
<tr>
<td>txid_current()</td>
<td>Returns the current transaction's ID (a BIGINT). It will assign a new one if the current transaction does not have one already.</td>
<td>txid_current()</td>
<td>various</td>
</tr>
<tr>
<td>typeof(expression)</td>
<td>Returns the name of the data type of the result of the expression.</td>
<td>typeof('abc')</td>
<td>'VARCHAR'</td>
</tr>
<tr>
<td>uuid()</td>
<td>Return a random uuid similar to this: eeccb8c5-9943-b2bb-bb5e-222f4e14b687.</td>
<td>uuid()</td>
<td>various</td>
</tr>
<tr>
<td>version()</td>
<td>Return the currently active version of DuckDB in this format</td>
<td>version()</td>
<td>various</td>
</tr>
</tbody>
</table>
Utility Table Functions

A table function is used in place of a table in a FROM clause.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>glob(search_path)</td>
<td>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.</td>
<td>glob('*')</td>
</tr>
</tbody>
</table>

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"
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.
**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.

```sql
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, therefore, this clause is parsed and applied, which is inefficient, but has an effect on the results. 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:

```sql
CREATE TABLE tbl AS SELECT s FROM range(1, 4) r(s);
SELECT string_agg(s, ', ') ORDER BY s DESC AS countdown FROM tbl;
```

<table>
<thead>
<tr>
<th>countdown</th>
<th>varchar</th>
</tr>
</thead>
<tbody>
<tr>
<td>3, 2, 1</td>
<td></td>
</tr>
</tbody>
</table>

**General Aggregate Functions**

The table below shows the available general aggregate functions.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
<th>Alias(es)</th>
</tr>
</thead>
<tbody>
<tr>
<td>any_value(arg)</td>
<td>Returns the first non-null value from arg. This function is affected by ordering.</td>
<td>any_value(A)</td>
<td>arbitrary(A), first(A)</td>
</tr>
<tr>
<td>arbitrary(arg)</td>
<td>Returns the first non-null value from arg. This function is affected by ordering.</td>
<td>arbitrary(A)</td>
<td>any_value(A), first(A)</td>
</tr>
<tr>
<td>arg_max(arg, val)</td>
<td>Finds the row with the maximum val. Calculates the arg expression at that row.</td>
<td>arg_max(A, B)</td>
<td>argMax(arg, val), max_by(arg, val)</td>
</tr>
<tr>
<td>arg_min(arg, val)</td>
<td>Finds the row with the minimum val. Calculates the arg expression at that row.</td>
<td>arg_min(A, B)</td>
<td>argMin(arg, val), min_by(arg, val)</td>
</tr>
<tr>
<td>avg(arg)</td>
<td>Calculates the average value for all tuples in arg.</td>
<td>avg(A)</td>
<td>mean</td>
</tr>
<tr>
<td>bit_and(arg)</td>
<td>Returns the bitwise AND of all bits in a given expression.</td>
<td>bit_and(A)</td>
<td>-</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>Example</td>
<td>Alias(es)</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>----------</td>
<td>-----------</td>
</tr>
<tr>
<td>bit_or(arg)</td>
<td>Returns the bitwise OR of all bits in a given expression.</td>
<td>bit_or(A)</td>
<td>-</td>
</tr>
<tr>
<td>bit_xor(arg)</td>
<td>Returns the bitwise XOR of all bits in a given expression.</td>
<td>bit_xor(A)</td>
<td>-</td>
</tr>
<tr>
<td>bitstring_agg(arg)</td>
<td>Returns a bitstring with bits set for each distinct value.</td>
<td>bitstring_agg(A)</td>
<td>-</td>
</tr>
<tr>
<td>bool_and(arg)</td>
<td>Returns true if every input value is true, otherwise false.</td>
<td>bool_and(A)</td>
<td>-</td>
</tr>
<tr>
<td>bool_or(arg)</td>
<td>Returns true if any input value is true, otherwise false.</td>
<td>bool_or(A)</td>
<td>-</td>
</tr>
<tr>
<td>count(arg)</td>
<td>Calculates the number of tuples in arg.</td>
<td>count(A)</td>
<td>-</td>
</tr>
<tr>
<td>favg(arg)</td>
<td>Calculates the average using a more accurate floating point summation (Kahan Sum).</td>
<td>favg(A)</td>
<td>-</td>
</tr>
<tr>
<td>first(arg)</td>
<td>Returns the first non-null value of a column. This function is affected by ordering.</td>
<td>first(A)</td>
<td>any_value(A), arbitrary(A)</td>
</tr>
<tr>
<td>fsum(arg)</td>
<td>Calculates the sum using a more accurate floating point summation (Kahan Sum).</td>
<td>fsum(A)</td>
<td>sumKahan, kahan_sum</td>
</tr>
<tr>
<td>geomean(arg)</td>
<td>Calculates the geometric mean for all tuples in arg.</td>
<td>geomean(A)</td>
<td>geometric_mean(A)</td>
</tr>
<tr>
<td>histogram(arg)</td>
<td>Returns a MAP of key-value pairs representing buckets and counts.</td>
<td>histogram(A)</td>
<td>-</td>
</tr>
<tr>
<td>last(arg)</td>
<td>Returns the last value of a column. This function is affected by ordering.</td>
<td>last(A)</td>
<td>-</td>
</tr>
<tr>
<td>list(arg)</td>
<td>Returns a LIST containing all the values of a column. This function is affected by ordering.</td>
<td>list(A)</td>
<td>array_agg</td>
</tr>
<tr>
<td>max(arg)</td>
<td>Returns the maximum value present in arg.</td>
<td>max(A)</td>
<td>-</td>
</tr>
</tbody>
</table>
### Function Description Example Alias(es)

- **min(arg)**
  - Returns the minimum value present in arg.
  - Example: `min(A)`
  - Alias(es): -

- **product(arg)**
  - Calculates the product of all tuples in arg.
  - Example: `product(A)`
  - Alias(es): -

- **string_agg(arg, sep)**
  - Concatenates the column string values with a separator.
  - Example: `string_agg(S, ',')`
  - This function is affected by ordering.
  - Alias(es): `group_concat(arg, sep)`, `listagg(arg, sep)`

- **sum(arg)**
  - Calculates the sum value for all tuples in arg.
  - Example: `sum(A)`
  - Alias(es): -

- **sum_no_overflow(arg)**
  - Calculates the sum value for all tuples in arg without overflow checks. Unlike sum, which works on floating-point values, sum_no_overflow only accepts INTEGER and DECIMAL values.
  - Example: `sum_no_overflow(A)`
  - Alias(es): -

### Approximate Aggregates

The table below shows the available approximate aggregate functions.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
<th>Alias(es)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>approx_count_distinct(x)</strong></td>
<td>Gives the approximate count of distinct elements using HyperLogLog.</td>
<td><code>approx_count_distinct(A)</code></td>
<td></td>
</tr>
<tr>
<td><strong>approx_quantile(x, pos)</strong></td>
<td>Gives the approximate quantile using T-Digest.</td>
<td><code>approx_quantile(A, 0.5)</code></td>
<td></td>
</tr>
<tr>
<td><strong>reservoir_quantile(x, quantile, sample_size = 8192)</strong></td>
<td>Gives the approximate quantile using reservoir sampling, the sample size is optional and uses 8192 as a default size.</td>
<td><code>reservoir_quantile(A, 0.5, 1024)</code></td>
<td></td>
</tr>
</tbody>
</table>

### Statistical Aggregates

The table below shows the available statistical aggregate functions.

---

602
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Formula</th>
<th>Alias</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>corr(y, x)</code></td>
<td>Returns the correlation coefficient for non-null pairs in a group.</td>
<td><code>covar_pop(y, x) / (stddev_pop(x) * stddev_pop(y))</code></td>
<td></td>
</tr>
<tr>
<td><code>covar_pop(y, x)</code></td>
<td>Returns the population covariance of input values.</td>
<td><code>(sum(x*y) - sum(x) * sum(y) / count(*)) / count(*)</code></td>
<td></td>
</tr>
<tr>
<td><code>covar_samp(y, x)</code></td>
<td>Returns the sample covariance for non-null pairs in a group.</td>
<td><code>(sum(x*y) - sum(x) * sum(y) / count(*)) / (count(*) - 1)</code></td>
<td></td>
</tr>
<tr>
<td><code>entropy(x)</code></td>
<td>Returns the log-2 entropy of count input-values.</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><code>kurtosis(x)</code></td>
<td>Returns the excess kurtosis (Fisher’s definition) of all input values, with a bias correction according to the sample size.</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><code>mad(x)</code></td>
<td>Returns the median absolute deviation for the values within x. NULL values are ignored. Temporal types return a positive INTERVAL.</td>
<td><code>median(abs(x - median(x)))</code></td>
<td>-</td>
</tr>
<tr>
<td><code>median(x)</code></td>
<td>Returns the middle value of the set. NULL values are ignored. For even value counts, quantitative values are averaged and ordinal values return the lower value.</td>
<td><code>quantile_cont(x, 0.5)</code></td>
<td></td>
</tr>
<tr>
<td><code>mode(x)</code></td>
<td>Returns the most frequent value for the values within x. NULL values are ignored.</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><code>quantile_cont(x, pos)</code></td>
<td>Returns the interpolated quantile number between 0 and 1. If pos is a LIST of FLOATs, then the result is a LIST of the corresponding interpolated quantiles.</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>Formula</td>
<td>Alias</td>
</tr>
<tr>
<td>-------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>quantile_disc(x, pos)</td>
<td>Returns the exact quantile number between 0 and 1. If pos is a LIST of FLOATs, then the result is a LIST of the corresponding exact quantiles.</td>
<td>-</td>
<td>quantile</td>
</tr>
<tr>
<td>regr_avgx(y, x)</td>
<td>Returns the average of the independent variable for non-null pairs in a group, where x is the independent variable and y is the dependent variable.</td>
<td>(sum(x<em>y) - sum(x) * sum(y) / count(</em>)) / count(*)</td>
<td>-</td>
</tr>
<tr>
<td>regr_avgy(y, x)</td>
<td>Returns the average of the dependent variable for non-null pairs in a group, where x is the independent variable and y is the dependent variable.</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>regr_count(y, x)</td>
<td>Returns the number of non-null number pairs in a group.</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>regr_intercept(y, x)</td>
<td>Returns the intercept of the univariate linear regression line for non-null pairs in a group.</td>
<td>avg(y) - regr_slope(y, x) * avg(x)</td>
<td>-</td>
</tr>
<tr>
<td>regr_r2(y, x)</td>
<td>Returns the coefficient of determination for non-null pairs in a group.</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>regr_slope(y, x)</td>
<td>Returns the slope of the linear regression line for non-null pairs in a group.</td>
<td>covar_pop(x, y) / var_pop(x)</td>
<td>-</td>
</tr>
<tr>
<td>regr_sxx(y, x)</td>
<td>Returns the population variance (covariance of input values) for non-null pairs in a group.</td>
<td>regr_count(y, x) * var_pop(x)</td>
<td>-</td>
</tr>
<tr>
<td>regr_sxy(y, x)</td>
<td>Returns the population covariance of input values.</td>
<td>regr_count(y, x) * covar_pop(y, x)</td>
<td>-</td>
</tr>
<tr>
<td>regr_syy(y, x)</td>
<td>Returns the population variance (covariance of input values) for non-null pairs in a group.</td>
<td>regr_count(y, x) * var_pop(y)</td>
<td>-</td>
</tr>
<tr>
<td>skewness(x)</td>
<td>Returns the skewness of all input values.</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>stddev_pop(x)</td>
<td>Returns the population standard deviation.</td>
<td>sqrt(var_pop(x))</td>
<td>-</td>
</tr>
</tbody>
</table>
**DuckDB Documentation**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Formula</th>
<th>Alias</th>
</tr>
</thead>
<tbody>
<tr>
<td>stddev_samp(x)</td>
<td>Returns the sample standard deviation.</td>
<td>$\sqrt{\text{var}_\text{samp}(x)}$</td>
<td>stddev(x)</td>
</tr>
<tr>
<td>var_pop(x)</td>
<td>Returns the population variance.</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>var_samp(x)</td>
<td>Returns the sample variance of all input values.</td>
<td>$\frac{(\sum(x^2) - \sum(x)^2 / \text{count}(x))}{\text{count}(x) - 1}$</td>
<td>variance(arg, val)</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>Function</th>
<th>Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mode()</code> <code>WITHIN GROUP</code></td>
<td><code>mode(sort_expression)</code></td>
</tr>
<tr>
<td><code>percentile_cont(fraction)</code> <code>WITHIN</code></td>
<td><code>quantile_cont(sort_expression, fraction)</code></td>
</tr>
<tr>
<td><code>percentile_cont(fractions)</code> <code>WITHIN</code></td>
<td><code>quantile_cont(sort_expression, fractions)</code></td>
</tr>
<tr>
<td><code>percentile_disc(fraction)</code> <code>WITHIN</code></td>
<td><code>quantile_disc(sort_expression, fraction)</code></td>
</tr>
<tr>
<td><code>percentile_disc(fractions)</code> <code>WITHIN</code></td>
<td><code>quantile_disc(sort_expression, fractions)</code></td>
</tr>
</tbody>
</table>

**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 also be reset to their original values using the `RESET` statement.

**Examples**

```sql
-- set the memory limit of the system to 10GB
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';

-- show a list of all available settings
SELECT * FROM duckdb_settings();

-- return the current value of a specific setting
-- this example returns 'automatic'
SELECT current_setting('access_mode');

-- reset the memory limit of the system back to the default
RESET memory_limit;

Configuration Reference

Below is a list of all available settings.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Input type</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calendar</td>
<td>The current calendar</td>
<td>VARCHAR</td>
<td>System (locale) calendar</td>
</tr>
<tr>
<td>TimeZone</td>
<td>The current time zone</td>
<td>VARCHAR</td>
<td>System (locale) timezone</td>
</tr>
<tr>
<td>access_mode</td>
<td>Access mode of the database ([AUTOMATIC, READ_ONLY or READ_WRITE])</td>
<td>VARCHAR</td>
<td>automatic</td>
</tr>
<tr>
<td>allocator_flush_threshold</td>
<td>Peak allocation threshold at which to flush the allocator after completing a task.</td>
<td>VARCHAR</td>
<td>128.0 MiB</td>
</tr>
<tr>
<td>allow_persistent_secrets</td>
<td>Allow the creation of persistent secrets, that are stored and loaded on restarts</td>
<td>BOOLEAN</td>
<td>1</td>
</tr>
<tr>
<td>allow_unsigned_extensions</td>
<td>Allow to load extensions with invalid or missing signatures</td>
<td>BOOLEAN</td>
<td>false</td>
</tr>
<tr>
<td>arrow_large_buffer_size</td>
<td>If arrow buffers for strings, blobs, uuids and bits should be exported using large buffers</td>
<td>BOOLEAN</td>
<td>false</td>
</tr>
<tr>
<td>autoinstall_extension_repository</td>
<td>Overrides the custom endpoint for extension installation on autoloading</td>
<td>VARCHAR</td>
<td></td>
</tr>
</tbody>
</table>
## DuckDB Documentation

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Input type</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>autoinstall_known_extensions</td>
<td>Whether known extensions are allowed to be automatically installed when a query depends on them</td>
<td>BOOLEAN</td>
<td>true</td>
</tr>
<tr>
<td>autoload_known_extensions</td>
<td>Whether known extensions are allowed to be automatically loaded when a query depends on them</td>
<td>BOOLEAN</td>
<td>true</td>
</tr>
<tr>
<td>binary_as_string</td>
<td>In Parquet files, interpret binary data as a string.</td>
<td>BOOLEAN</td>
<td></td>
</tr>
<tr>
<td>checkpoint_threshold,wal_autocheckpoint</td>
<td>The WAL size threshold at which to automatically trigger a checkpoint (e.g., 1GB)</td>
<td>VARCHAR</td>
<td>16.0 MiB</td>
</tr>
<tr>
<td>custom_extension_repository</td>
<td>Overrides the custom endpoint for remote extension installation</td>
<td>VARCHAR</td>
<td></td>
</tr>
<tr>
<td>custom_user_agent</td>
<td>Metadata from DuckDB callers</td>
<td>VARCHAR</td>
<td></td>
</tr>
<tr>
<td>default_collation</td>
<td>The collation setting used when none is specified</td>
<td>VARCHAR</td>
<td></td>
</tr>
<tr>
<td>default_null_order,null_order</td>
<td>Null ordering used when none is specified (null_order: NULLS_FIRST or NULLS_LAST)</td>
<td>VARCHAR</td>
<td>NULLS_LAST</td>
</tr>
<tr>
<td>default_order</td>
<td>The order type used when none is specified (order: ASC or DESC)</td>
<td>VARCHAR</td>
<td>ASC</td>
</tr>
<tr>
<td>default_secret_storage</td>
<td>Allows switching the default storage for secrets</td>
<td>VARCHAR</td>
<td>local_file</td>
</tr>
<tr>
<td>disabled_filesystems</td>
<td>Disable specific file systems preventing access (e.g., LocalFileSystem)</td>
<td>VARCHAR</td>
<td></td>
</tr>
<tr>
<td>duckdb_api</td>
<td>DuckDB API surface</td>
<td>VARCHAR</td>
<td>duckdb/v0.9.3-dev2267(linux_amd64_gcc4)</td>
</tr>
<tr>
<td>enable_external_access</td>
<td>Allow the database to access external state (through e.g., loading/installing modules, COPY TO/FROM, CSV readers, pandas replacement scans, etc)</td>
<td>BOOLEAN</td>
<td>true</td>
</tr>
<tr>
<td>enable_fsst_vectors</td>
<td>Allow scans on FSST compressed segments to emit compressed vectors to utilize late decompression</td>
<td>BOOLEAN</td>
<td>false</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Input type</td>
<td>Default value</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------------</td>
<td>---------------</td>
</tr>
<tr>
<td>enable_http_metadata_cache</td>
<td>Whether or not the global http metadata is used to cache HTTP metadata</td>
<td>BOOLEAN</td>
<td>false</td>
</tr>
<tr>
<td>enable_object_cache</td>
<td>Whether or not object cache is used to cache e.g., Parquet metadata</td>
<td>BOOLEAN</td>
<td>false</td>
</tr>
<tr>
<td>enable_profiling</td>
<td>Enables profiling, and sets the output format (JSON, QUERY_TREE, QUERY_TREE_</td>
<td>VARCHAR</td>
<td>NULL</td>
</tr>
<tr>
<td></td>
<td>OPTIMIZER)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>enable_progress_bar_print</td>
<td>Controls the printing of the progress bar, when 'enable_progress_bar' is true</td>
<td>BOOLEAN</td>
<td>true</td>
</tr>
<tr>
<td>enable_progress_bar</td>
<td>Enables the progress bar, printing progress to the terminal for long queries</td>
<td>BOOLEAN</td>
<td>false</td>
</tr>
<tr>
<td>explain_output</td>
<td>Output of EXPLAIN statements (ALL, OPTIMIZED_ONLY, PHYSICAL_ONLY)</td>
<td>VARCHAR</td>
<td>physical_only</td>
</tr>
<tr>
<td>extension_directory</td>
<td>Set the directory to store extensions in</td>
<td>VARCHAR</td>
<td></td>
</tr>
<tr>
<td>external_threads</td>
<td>The number of external threads that work on DuckDB tasks.</td>
<td>BIGINT</td>
<td>0</td>
</tr>
<tr>
<td>file_search_path</td>
<td>A comma separated list of directories to search for input files</td>
<td>VARCHAR</td>
<td></td>
</tr>
<tr>
<td>force_download</td>
<td>Forces upfront download of file</td>
<td>BOOLEAN</td>
<td>0</td>
</tr>
<tr>
<td>home_directory</td>
<td>Sets the home directory used by the system</td>
<td>VARCHAR</td>
<td></td>
</tr>
<tr>
<td>http_keep_alive</td>
<td>Keep alive connections. Setting this to false can help when running into connection failures</td>
<td>BOOLEAN</td>
<td>1</td>
</tr>
<tr>
<td>http_retries</td>
<td>HTTP retries on I/O error (default 3)</td>
<td>BIGINT</td>
<td>3</td>
</tr>
<tr>
<td>http_retry_backoff</td>
<td>Backoff factor for exponentially increasing retry wait time (default 4)</td>
<td>FLOAT</td>
<td>4</td>
</tr>
<tr>
<td>http_retry_wait_ms</td>
<td>Time between retries (default 100ms)</td>
<td>BIGINT</td>
<td>100</td>
</tr>
<tr>
<td>http_timeout</td>
<td>HTTP timeout read/write/connection/retry (default 30000ms)</td>
<td>BIGINT</td>
<td>30000</td>
</tr>
<tr>
<td>immediate_transaction_mode</td>
<td>Whether transactions should be started lazily when needed, or immediately when BEGIN TRANSACTION is called</td>
<td>BOOLEAN</td>
<td>false</td>
</tr>
<tr>
<td>integer_division</td>
<td>Whether or not the / operator defaults to integer division, or to floating point division</td>
<td>BOOLEAN</td>
<td>0</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Input type</td>
<td>Default value</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------------</td>
<td>---------------</td>
</tr>
<tr>
<td>lock_configuration</td>
<td>Whether or not the configuration can be altered</td>
<td>BOOLEAN</td>
<td>false</td>
</tr>
<tr>
<td>log_query_path</td>
<td>Specifies the path to which queries should be logged (default: empty string, queries are not logged)</td>
<td>VARCHAR</td>
<td>NULL</td>
</tr>
<tr>
<td>max_expression_depth</td>
<td>The maximum expression depth limit in the parser. WARNING: increasing this setting and using very deep expressions might lead to stack overflow errors.</td>
<td>UBIGINT</td>
<td>1000</td>
</tr>
<tr>
<td>max_memory, memory_limit</td>
<td>The maximum memory of the system (e.g., 1GB)</td>
<td>VARCHAR</td>
<td>80% of RAM</td>
</tr>
<tr>
<td>old_implicit_casting</td>
<td>Allow implicit casting to/from VARCHAR</td>
<td>BOOLEAN</td>
<td>false</td>
</tr>
<tr>
<td>ordered_aggregate_threshold</td>
<td>The number of rows to accumulate before sorting, used for tuning</td>
<td>UBIGINT</td>
<td>262144</td>
</tr>
<tr>
<td>password</td>
<td>The password to use. Ignored for legacy compatibility.</td>
<td>VARCHAR</td>
<td>NULL</td>
</tr>
<tr>
<td>perfect_ht_threshold</td>
<td>Threshold in bytes for when to use a perfect hash table (default: 12)</td>
<td>BIGINT</td>
<td>12</td>
</tr>
<tr>
<td>pivot_filter_threshold</td>
<td>The threshold to switch from using filtered aggregates to LIST with a dedicated pivot operator</td>
<td>BIGINT</td>
<td>10</td>
</tr>
<tr>
<td>pivot_limit</td>
<td>The maximum number of pivot columns in a pivot statement (default: 100000)</td>
<td>BIGINT</td>
<td>100000</td>
</tr>
<tr>
<td>prefer_range_joins</td>
<td>Force use of range joins with mixed predicates</td>
<td>BOOLEAN</td>
<td>false</td>
</tr>
<tr>
<td>preserve_identifier_case</td>
<td>Whether or not to preserve the identifier case, instead of always lowercasing all non-quoted identifiers</td>
<td>BOOLEAN</td>
<td>true</td>
</tr>
<tr>
<td>preserve_insertion_order</td>
<td>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.</td>
<td>BOOLEAN</td>
<td>true</td>
</tr>
<tr>
<td>profile_output, profiling_output</td>
<td>The file to which profile output should be saved, or empty to print to the terminal</td>
<td>VARCHAR</td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Input type</td>
<td>Default value</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------------</td>
<td>---------------</td>
</tr>
<tr>
<td>profiler_history_size</td>
<td>Sets the profiler history size</td>
<td>BIGINT</td>
<td>NULL</td>
</tr>
<tr>
<td>profiling_mode</td>
<td>The profiling mode (STANDARD or DETAILED)</td>
<td>VARCHAR</td>
<td>NULL</td>
</tr>
<tr>
<td>progress_bar_time</td>
<td>Sets the time (in milliseconds) how long a query needs to take before we start printing a progress bar</td>
<td>BIGINT</td>
<td>2000</td>
</tr>
<tr>
<td>s3_access_key_id</td>
<td>S3 Access Key ID</td>
<td>VARCHAR</td>
<td></td>
</tr>
<tr>
<td>s3_endpoint</td>
<td>S3 Endpoint (empty for default endpoint)</td>
<td>VARCHAR</td>
<td></td>
</tr>
<tr>
<td>s3_region</td>
<td>S3 Region (default us-east-1)</td>
<td>VARCHAR</td>
<td>us-east-1</td>
</tr>
<tr>
<td>s3_secret_access_key</td>
<td>S3 Access Key</td>
<td>VARCHAR</td>
<td></td>
</tr>
<tr>
<td>s3_session_token</td>
<td>S3 Session Token</td>
<td>VARCHAR</td>
<td></td>
</tr>
<tr>
<td>s3_uploader_max_filesize</td>
<td>S3 Uploader max filesize (between 50GB and 5TB, default 800GB)</td>
<td>VARCHAR</td>
<td>800GB</td>
</tr>
<tr>
<td>s3_uploader_max_parts_per_file</td>
<td>S3 Uploader max parts per file (between 1 and 10000, default 10000)</td>
<td>BIGINT</td>
<td>10000</td>
</tr>
<tr>
<td>s3_uploader_thread_limit</td>
<td>S3 Uploader global thread limit (default 50)</td>
<td>BIGINT</td>
<td>50</td>
</tr>
<tr>
<td>s3_url_compatibility_mode</td>
<td>Disable Globs and Query Parameters on S3 URLs</td>
<td>BOOLEAN</td>
<td>0</td>
</tr>
<tr>
<td>s3_url_style</td>
<td>S3 URL style ('vhost' (default) or 'path')</td>
<td>VARCHAR</td>
<td>vhost</td>
</tr>
<tr>
<td>s3_use_ssl</td>
<td>S3 use SSL (default true)</td>
<td>BOOLEAN</td>
<td>1</td>
</tr>
<tr>
<td>schema</td>
<td>Sets the default search schema. Equivalent to setting search_path to a single value.</td>
<td>VARCHAR</td>
<td>main</td>
</tr>
<tr>
<td>search_path</td>
<td>Sets the default catalog search path as a comma-separated list of values</td>
<td>VARCHAR</td>
<td></td>
</tr>
<tr>
<td>secret_directory</td>
<td>Set the directory to which persistent secrets are stored</td>
<td>VARCHAR</td>
<td>/home/runner/.duckdb/stored_secrets/c6b29d5d1d</td>
</tr>
<tr>
<td>temp_directory</td>
<td>Set the directory to which to write temp files</td>
<td>VARCHAR</td>
<td></td>
</tr>
<tr>
<td>threads,worker_threads</td>
<td>The number of total threads used by the system.</td>
<td>BIGINT</td>
<td># Cores</td>
</tr>
<tr>
<td>username, user</td>
<td>The username to use. Ignored for legacy compatibility.</td>
<td>VARCHAR</td>
<td>NULL</td>
</tr>
</tbody>
</table>
**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.

**Syntax**

**Check**

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.

```sql
CREATE TABLE students (name VARCHAR CHECK (NOT contains(name, ' ')));
INSERT INTO students VALUES ('this name contains spaces');
-- Constraint Error: CHECK constraint failed: students
```

**Null**

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.

```sql
CREATE TABLE students (name VARCHAR NOT NULL);
INSERT INTO students VALUES (NULL);
-- Constraint Error: NOT NULL constraint failed: students.name
```

**Primary Key/Unique**

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.

```sql
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
```

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.
**Note.** Indexes have certain limitations that might result in constraints being evaluated too eagerly, see the indexes section for more details

### Foreign Key

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.

```sql
CREATE TABLE students (id INTEGER PRIMARY KEY, name VARCHAR);
CREATE TABLE exams (exam_id INTEGER REFERENCES students(id), grade INTEGER);
INSERT INTO students VALUES (1, 'Student 1');
INSERT INTO exams VALUES (1, 10);
INSERT INTO exams VALUES (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.

**Note.** Indexes have certain limitations that might result in constraints being evaluated too eagerly, see the indexes section for more details

### Indexes

#### Index Types

DuckDB currently uses two index types:

- A **min-max index** (also known as zonemap and block range index) is automatically created for columns of all general-purpose data types.
- 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. Such an index is automatically created for columns with a UNIQUE or PRIMARY KEY constraint and can be defined using CREATE INDEX.

**Note.** ART indexes must currently be able to fit in-memory. Avoid creating ART indexes if the index does not fit in memory.

### Persistence

Both min-max indexes and ART indexes are persisted on disk.

**CREATE INDEX and DROP INDEX**

To create an index, use the **CREATE INDEX statement.** To drop an index, use the **DROP INDEX statement.**
**Index Limitations**

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.

**Updates Become Deletes and Inserts**  When an update statement is executed on a column that is present in an index - the statement is transformed into a delete of the original row followed by an insert. This has certain performance implications, particularly for wide tables, as entire rows are rewritten instead of only the affected columns.

**Over-Eager Unique Constraint Checking**  Due to the presence of transactions, data can only be removed from the index after (1) the transaction that performed the delete is committed, and (2) no further transactions exist that refer to the old entry still present in the index. As a result of this - transactions that perform deletions followed by insertions may trigger unexpected unique constraint violations, as the deleted tuple has not actually been removed from the index yet. For example:

```sql
CREATE TABLE students (id INTEGER PRIMARY KEY, name VARCHAR);
INSERT INTO students VALUES (1, 'Student 1');
BEGIN;
DELETE FROM students WHERE id = 1;
INSERT INTO students VALUES (1, 'Student 2');
-- Constraint Error: Duplicate key "id: 1" violates primary key constraint
```

This, combined with the fact that updates are turned into deletions and insertions within the same transaction, means that updating rows in the presence of unique or primary key constraints can often lead to unexpected unique constraint violations.

```sql
CREATE TABLE students (id INTEGER PRIMARY KEY, name VARCHAR);
INSERT INTO students VALUES (1, 'Student 1');
UPDATE students SET name = 'Student 2', id = 1 WHERE id = 1;
-- Constraint Error: Duplicate key "id: 1" violates primary key constraint
```

Currently this is an expected limitation of the system - although we aim to resolve this in the future.

**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.

**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:
<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
<th>Type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>catalog_name</td>
<td>Name of the database that the schema is contained in.</td>
<td>VARCHAR</td>
<td>NULL</td>
</tr>
<tr>
<td>schema_name</td>
<td>Name of the schema.</td>
<td>VARCHAR</td>
<td>'main'</td>
</tr>
<tr>
<td>schema_owner</td>
<td>Name of the owner of the schema.</td>
<td>VARCHAR</td>
<td>NULL</td>
</tr>
<tr>
<td>default_character_set_catalog</td>
<td>Applies to a feature not available in DuckDB.</td>
<td>VARCHAR</td>
<td>NULL</td>
</tr>
<tr>
<td>default_character_set_schema</td>
<td>Applies to a feature not available in DuckDB.</td>
<td>VARCHAR</td>
<td>NULL</td>
</tr>
<tr>
<td>default_character_set_name</td>
<td>Applies to a feature not available in DuckDB.</td>
<td>VARCHAR</td>
<td>NULL</td>
</tr>
<tr>
<td>sql_path</td>
<td>The file system location of the database. Currently unimplemented.</td>
<td>VARCHAR</td>
<td>NULL</td>
</tr>
</tbody>
</table>

**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:

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
<th>Type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>table_catalog</td>
<td>The catalog the table or view belongs to.</td>
<td>VARCHAR</td>
<td>NULL</td>
</tr>
<tr>
<td>table_schema</td>
<td>The schema the table or view belongs to.</td>
<td>VARCHAR</td>
<td>'main'</td>
</tr>
<tr>
<td>table_name</td>
<td>The name of the table or view.</td>
<td>VARCHAR</td>
<td>'widgets'</td>
</tr>
<tr>
<td>table_type</td>
<td>The type of table. One of: BASE TABLE, LOCAL TEMPORARY VIEW.</td>
<td>VARCHAR</td>
<td>'BASE TABLE'</td>
</tr>
<tr>
<td>self_referencing_column_name</td>
<td>Applies to a feature not available in DuckDB.</td>
<td>VARCHAR</td>
<td>NULL</td>
</tr>
<tr>
<td>reference_generation</td>
<td>Applies to a feature not available in DuckDB.</td>
<td>VARCHAR</td>
<td>NULL</td>
</tr>
</tbody>
</table>
DuckDB Documentation

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
<th>Type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>user_defined_type_catalog</td>
<td>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.</td>
<td>VARCHAR</td>
<td>NULL</td>
</tr>
<tr>
<td>user_defined_type_schema</td>
<td>If the table is a typed table, the name of the schema that contains the underlying data type, else null. Currently unimplemented.</td>
<td>VARCHAR</td>
<td>NULL</td>
</tr>
<tr>
<td>user_defined_type_name</td>
<td>If the table is a typed table, the name of the underlying data type, else null. Currently unimplemented.</td>
<td>VARCHAR</td>
<td>NULL</td>
</tr>
<tr>
<td>is_insertable_into</td>
<td>YES if the table is insertable into, NO if not (Base tables are always insertable into, views not necessarily.)</td>
<td>VARCHAR</td>
<td>'YES'</td>
</tr>
<tr>
<td>is_typed</td>
<td>YES if the table is a typed table, NO if not.</td>
<td>VARCHAR</td>
<td>'NO'</td>
</tr>
<tr>
<td>commit_action</td>
<td>Not yet implemented.</td>
<td>VARCHAR</td>
<td>'NO'</td>
</tr>
</tbody>
</table>

**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:

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
<th>Type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>table_catalog</td>
<td>Name of the database containing the table.</td>
<td>VARCHAR</td>
<td>NULL</td>
</tr>
<tr>
<td>table_schema</td>
<td>Name of the schema containing the table.</td>
<td>VARCHAR</td>
<td>'main'</td>
</tr>
<tr>
<td>table_name</td>
<td>Name of the table.</td>
<td>VARCHAR</td>
<td>'widgets'</td>
</tr>
<tr>
<td>column_name</td>
<td>Name of the column.</td>
<td>VARCHAR</td>
<td>'price'</td>
</tr>
<tr>
<td>ordinal_position</td>
<td>Ordinal position of the column within the table (count starts at 1).</td>
<td>INTEGER</td>
<td>5</td>
</tr>
<tr>
<td>column_default</td>
<td>Default expression of the column.</td>
<td>VARCHAR</td>
<td>1.99</td>
</tr>
<tr>
<td>Column</td>
<td>Description</td>
<td>Type</td>
<td>Example</td>
</tr>
<tr>
<td>------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>----------</td>
<td>---------------------</td>
</tr>
<tr>
<td><code>is_nullable</code></td>
<td>YES if the column is possibly nullable, NO if it is known not nullable.</td>
<td>VARCHAR</td>
<td>'YES'</td>
</tr>
<tr>
<td><code>data_type</code></td>
<td>Data type of the column.</td>
<td>VARCHAR</td>
<td>'DECIMAL(18, 2)'</td>
</tr>
<tr>
<td><code>character_maximum_length</code></td>
<td>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.</td>
<td>INTEGER</td>
<td>255</td>
</tr>
<tr>
<td><code>character_octet_length</code></td>
<td>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.</td>
<td>INTEGER</td>
<td>1073741824</td>
</tr>
<tr>
<td><code>numeric_precision</code></td>
<td>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.</td>
<td>INTEGER</td>
<td>18</td>
</tr>
<tr>
<td><code>numeric_scale</code></td>
<td>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.</td>
<td>INTEGER</td>
<td>2</td>
</tr>
</tbody>
</table>
### Column Description

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
<th>Type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>datetime_precision</td>
<td>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.</td>
<td>INTEGER</td>
<td>0</td>
</tr>
</tbody>
</table>

### Catalog Functions

Several functions are also provided to see details about the schemas that are configured in the database.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>current_schema()</td>
<td>Return the name of the currently active schema. Default is main.</td>
<td>current_schema()</td>
<td>'main'</td>
</tr>
<tr>
<td>current_schemas(boolean)</td>
<td>Return list of schemas. Pass a parameter of true to include implicit schemas.</td>
<td>current_schemas(true)</td>
<td>['temp', 'main', 'pg_catalog']</td>
</tr>
</tbody>
</table>

### 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:

```sql
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.

Note. 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:

-- duckdb_views table function: returns all views, including those marked internal
SELECT * FROM duckdb_views();
-- 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.

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>database_name</td>
<td>The name of the database that contains the column object.</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>database_oid</td>
<td>Internal identifier of the database that contains the column object.</td>
<td>BIGINT</td>
</tr>
<tr>
<td>schema_name</td>
<td>The SQL name of the schema that contains the table object that defines this column.</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>schema_oid</td>
<td>Internal identifier of the schema object that contains the table of the column.</td>
<td>BIGINT</td>
</tr>
<tr>
<td>table_name</td>
<td>The SQL name of the table that defines the column.</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>table_oid</td>
<td>Internal identifier (name) of the table object that defines the column.</td>
<td>BIGINT</td>
</tr>
<tr>
<td>column_name</td>
<td>The SQL name of the column.</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>column_index</td>
<td>The unique position of the column within its table.</td>
<td>INTEGER</td>
</tr>
<tr>
<td>internal</td>
<td>true if this column built-in, false if it is user-defined.</td>
<td>BOOLEAN</td>
</tr>
<tr>
<td>column_default</td>
<td>The default value of the column (expressed in SQL)</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>is_nullable</td>
<td>true if the column can hold NULL values; false if the column cannot hold NULL-values.</td>
<td>BOOLEAN</td>
</tr>
</tbody>
</table>
DuckDB Documentation

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>data_type</td>
<td>The name of the column datatype.</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>data_type_id</td>
<td>The internal identifier of the column data type</td>
<td>BIGINT</td>
</tr>
<tr>
<td>character_maximum_length</td>
<td>Always NULL. DuckDB text types do not enforce a value length restriction based on a length type parameter.</td>
<td>INTEGER</td>
</tr>
<tr>
<td>numeric_precision</td>
<td>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.</td>
<td>INTEGER</td>
</tr>
<tr>
<td>numeric_precision_radix</td>
<td>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.</td>
<td>INTEGER</td>
</tr>
<tr>
<td>numeric_scale</td>
<td>Applicable to decimal type. Indicates the maximum number of fractional digits (i.e., the number of digits that may appear after the decimal separator).</td>
<td>INTEGER</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>database_name</td>
<td>The name of the database that contains the constraint.</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>database_oid</td>
<td>Internal identifier of the database that contains the constraint.</td>
<td>BIGINT</td>
</tr>
<tr>
<td>Column</td>
<td>Description</td>
<td>Type</td>
</tr>
<tr>
<td>-----------------</td>
<td>------------------------------------------------------------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>schema_name</td>
<td>The SQL name of the schema that contains the table on which the constraint is defined.</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>schema_oid</td>
<td>Internal identifier of the schema object that contains the table on which the constraint is defined.</td>
<td>BIGINT</td>
</tr>
<tr>
<td>table_name</td>
<td>The SQL name of the table on which the constraint is defined.</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>table_oid</td>
<td>Internal identifier (name) of the table object on which the constraint is defined.</td>
<td>BIGINT</td>
</tr>
<tr>
<td>constraint_index</td>
<td>Indicates the position of the constraint as it appears in its table definition.</td>
<td>BIGINT</td>
</tr>
<tr>
<td>constraint_type</td>
<td>Indicates the type of constraint. Applicable values are CHECK, FOREIGN KEY, PRIMARY KEY, NOT NULL, UNIQUE.</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>constraint_text</td>
<td>The definition of the constraint expressed as a SQL-phrase. (Not necessarily a complete or syntactically valid DDL-statement.)</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>expression</td>
<td>If constraint is a check constraint, the definition of the condition being checked, otherwise NULL.</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>constraint_column_indexes</td>
<td>An array of table column indexes referring to the columns that appear in the constraint definition</td>
<td>BIGINT[]</td>
</tr>
<tr>
<td>constraint_column_names</td>
<td>An array of table column names appearing in the constraint definition</td>
<td>VARCHAR[]</td>
</tr>
</tbody>
</table>

### 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.

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>database_name</td>
<td>The name of the database, or the alias if the database was attached using an ALIAS-clause.</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>database_oid</td>
<td>The internal identifier of the database.</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>path</td>
<td>The file path associated with the database.</td>
<td>VARCHAR</td>
</tr>
</tbody>
</table>
## Column Description Type

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>internal</code></td>
<td>true indicates a system or built-in database. False indicates a user-defined database.</td>
<td>BOOLEAN</td>
</tr>
<tr>
<td><code>type</code></td>
<td>The type indicates the type of RDBMS implemented by the attached database. For DuckDB databases, that value is duckdb.</td>
<td></td>
</tr>
</tbody>
</table>

### duckdb_dependencies

The `duckdb_dependencies()` function provides metadata about the dependencies available in the DuckDB instance.

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>classid</code></td>
<td>Always 0</td>
<td>BIGINT</td>
</tr>
<tr>
<td><code>objid</code></td>
<td>The internal id of the object.</td>
<td>BIGINT</td>
</tr>
<tr>
<td><code>objsubid</code></td>
<td>Always 0</td>
<td>INTEGER</td>
</tr>
<tr>
<td><code>refclassid</code></td>
<td>Always 0</td>
<td>BIGINT</td>
</tr>
<tr>
<td><code>refobjid</code></td>
<td>The internal id of the dependent object.</td>
<td>BIGINT</td>
</tr>
<tr>
<td><code>refobjsubid</code></td>
<td>Always 0</td>
<td>INTEGER</td>
</tr>
<tr>
<td><code>deitype</code></td>
<td>The type of dependency. Either regular (n) or automatic (a).</td>
<td>VARCHAR</td>
</tr>
</tbody>
</table>

### duckdb_extensions

The `duckdb_extensions()` function provides metadata about the extensions available in the DuckDB instance.

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>extension_name</code></td>
<td>The name of the extension.</td>
<td>VARCHAR</td>
</tr>
<tr>
<td><code>loaded</code></td>
<td>true if the extension is loaded, false if it's not loaded.</td>
<td>BOOLEAN</td>
</tr>
<tr>
<td><code>installed</code></td>
<td>true if the extension is installed, false if it's not installed.</td>
<td>BOOLEAN</td>
</tr>
<tr>
<td><code>install_path</code></td>
<td>(BUILT-IN) if the extension is built-in, otherwise, the filesystem path where binary that implements the extension resides.</td>
<td>VARCHAR</td>
</tr>
<tr>
<td><code>description</code></td>
<td>Human readable text that describes the extension's functionality.</td>
<td>VARCHAR</td>
</tr>
</tbody>
</table>
**duckdb_functions**

The `duckdb_functions()` function provides metadata about the functions (including macros) available in the DuckDB instance.

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>database_name</td>
<td>The name of the database that contains this function.</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>schema_name</td>
<td>The SQL name of the schema where the function resides.</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>function_name</td>
<td>The SQL name of the function.</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>function_type</td>
<td>The function kind. Value is one of: table, scalar, aggregate, pragma, macro</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>description</td>
<td>Description of this function (always NULL)</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>return_type</td>
<td>The logical data type name of the returned value. Applicable for scalar and aggregate functions.</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>parameters</td>
<td>If the function has parameters, the list of parameter names.</td>
<td>VARCHAR[]</td>
</tr>
<tr>
<td>parameter_types</td>
<td>If the function has parameters, a list of logical data type names corresponding to the parameter list.</td>
<td>VARCHAR[]</td>
</tr>
<tr>
<td>varargs</td>
<td>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.</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>macro_definition</td>
<td>If this is a macro, the SQL expression that defines it.</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>has_side_effects</td>
<td>false if this is a pure function. true if this function changes the database state (like sequence functions <code>nextval()</code> and <code>currval()</code>).</td>
<td>BOOLEAN</td>
</tr>
<tr>
<td>function_oid</td>
<td>The internal identifier for this function</td>
<td>BIGINT</td>
</tr>
</tbody>
</table>

**duckdb_indexes**

The `duckdb_indexes()` function provides metadata about secondary indexes available in the DuckDB instance.
## DuckDB Documentation

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>database_name</td>
<td>The name of the database that contains this index.</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>database_oid</td>
<td>Internal identifier of the database containing the index.</td>
<td>BIGINT</td>
</tr>
<tr>
<td>schema_name</td>
<td>The SQL name of the schema that contains the table with the secondary index.</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>schema_oid</td>
<td>Internal identifier of the schema object.</td>
<td>BIGINT</td>
</tr>
<tr>
<td>index_name</td>
<td>The SQL name of this secondary index</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>index_oid</td>
<td>The object identifier of this index.</td>
<td>BIGINT</td>
</tr>
<tr>
<td>table_name</td>
<td>The name of the table with the index</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>table_oid</td>
<td>Internal identifier (name) of the table object.</td>
<td>BIGINT</td>
</tr>
<tr>
<td>is_unique</td>
<td>true if the index was created with the UNIQUE modifier, false if it was not.</td>
<td>BOOLEAN</td>
</tr>
<tr>
<td>is_primary</td>
<td>Always false</td>
<td>BOOLEAN</td>
</tr>
<tr>
<td>expressions</td>
<td>Always NULL</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>sql</td>
<td>The definition of the index, expressed as a CREATE INDEX SQL statement.</td>
<td>VARCHAR</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>keyword_name</td>
<td>The keyword.</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>keyword_category</td>
<td>Indicates the category of the keyword. Values are column_name, reserved, type_function and unreserved.</td>
<td>VARCHAR</td>
</tr>
</tbody>
</table>

### 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`. 

---

623
**duckdb_schemas**

The `duckdb_schemas()` function provides metadata about the schemas available in the DuckDB instance.

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>oid</td>
<td>Internal identifier of the schema object.</td>
<td>BIGINT</td>
</tr>
<tr>
<td>database_name</td>
<td>The name of the database that contains this schema.</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>database_oid</td>
<td>Internal identifier of the database containing the schema.</td>
<td>BIGINT</td>
</tr>
<tr>
<td>schema_name</td>
<td>The SQL name of the schema.</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>internal</td>
<td>true if this is an internal (built-in) schema, false if this is a user-defined schema.</td>
<td>BOOLEAN</td>
</tr>
<tr>
<td>sql</td>
<td>Always NULL</td>
<td>VARCHAR</td>
</tr>
</tbody>
</table>

The `information_schema.schemata` system view provides a more standardized way to obtain metadata about database schemas.

**duckdb_sequences**

The `duckdb_sequences()` function provides metadata about the sequences available in the DuckDB instance.

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>database_name</td>
<td>The name of the database that contains this sequence</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>database_oid</td>
<td>Internal identifier of the database containing the sequence.</td>
<td>BIGINT</td>
</tr>
<tr>
<td>schema_name</td>
<td>The SQL name of the schema that contains the sequence object.</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>schema_oid</td>
<td>Internal identifier of the schema object that contains the sequence object.</td>
<td>BIGINT</td>
</tr>
<tr>
<td>Column</td>
<td>Description</td>
<td>Type</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>sequence_name</td>
<td>The SQL name that identifies the sequence within the schema.</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>sequence_oid</td>
<td>The internal identifier of this sequence object.</td>
<td>BIGINT</td>
</tr>
<tr>
<td>temporary</td>
<td>Whether this sequence is temporary. Temporary sequences are transient and only visible within the current connection.</td>
<td>BOOLEAN</td>
</tr>
<tr>
<td>start_value</td>
<td>The initial value of the sequence. This value will be returned when nextval() is called for the very first time on this sequence.</td>
<td>BIGINT</td>
</tr>
<tr>
<td>min_value</td>
<td>The minimum value of the sequence.</td>
<td>BIGINT</td>
</tr>
<tr>
<td>max_value</td>
<td>The maximum value of the sequence.</td>
<td>BIGINT</td>
</tr>
<tr>
<td>increment_by</td>
<td>The value that is added to the current value of the sequence to draw the next value from the sequence.</td>
<td>BIGINT</td>
</tr>
<tr>
<td>cycle</td>
<td>Whether the sequence should start over when drawing the next value would result in a value outside the range.</td>
<td>BOOLEAN</td>
</tr>
<tr>
<td>last_value</td>
<td>null if no value was ever drawn from the sequence using nextval(...). 1 if a value was drawn.</td>
<td>BIGINT</td>
</tr>
<tr>
<td>sql</td>
<td>The definition of this object, expressed as SQL DDL-statement.</td>
<td>VARCHAR</td>
</tr>
</tbody>
</table>

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.

**Note.**

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.
<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the setting.</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>value</td>
<td>Current value of the setting.</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>description</td>
<td>A description of the setting.</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>input_type</td>
<td>The logical datatype of the setting's value.</td>
<td>VARCHAR</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>database_name</td>
<td>The name of the database that contains this table</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>database_oid</td>
<td>Internal identifier of the database containing the table.</td>
<td>BIGINT</td>
</tr>
<tr>
<td>schema_name</td>
<td>The SQL name of the schema that contains the base table.</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>schema_oid</td>
<td>Internal identifier of the schema object that contains the base table.</td>
<td>BIGINT</td>
</tr>
<tr>
<td>table_name</td>
<td>The SQL name of the base table.</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>table_oid</td>
<td>Internal identifier of the base table object.</td>
<td>BIGINT</td>
</tr>
<tr>
<td>internal</td>
<td>false if this is a user-defined table.</td>
<td>BOOLEAN</td>
</tr>
<tr>
<td>temporary</td>
<td>Whether this is a temporary table. Temporary tables are not persisted and only visible within the current connection.</td>
<td>BOOLEAN</td>
</tr>
<tr>
<td>has_primary_key</td>
<td>true if this table object defines a PRIMARY KEY.</td>
<td>BOOLEAN</td>
</tr>
<tr>
<td>estimated_size</td>
<td>The estimated number of rows in the table.</td>
<td>BIGINT</td>
</tr>
<tr>
<td>column_count</td>
<td>The number of columns defined by this object</td>
<td>BIGINT</td>
</tr>
<tr>
<td>index_count</td>
<td>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.</td>
<td>BIGINT</td>
</tr>
<tr>
<td>check_constraint_count</td>
<td>The number of check constraints active on columns within the table.</td>
<td>BIGINT</td>
</tr>
</tbody>
</table>
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_types**

The `duckdb_types()` function provides metadata about the data types available in the DuckDB instance.

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>database_name</td>
<td>The name of the database that contains this schema.</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>database_oid</td>
<td>Internal identifier of the database that contains the data type.</td>
<td>BIGINT</td>
</tr>
<tr>
<td>schema_name</td>
<td>The SQL name of the schema containing the type definition. Always main.</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>schema_oid</td>
<td>Internal identifier of the schema object.</td>
<td>BIGINT</td>
</tr>
<tr>
<td>type_name</td>
<td>The name or alias of this data type.</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>type_oid</td>
<td>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).</td>
<td>BIGINT</td>
</tr>
<tr>
<td>type_size</td>
<td>The number of bytes required to represent a value of this type in memory.</td>
<td>BIGINT</td>
</tr>
<tr>
<td>logical_type</td>
<td>The 'canonical' name of this data type. The same logical_type may be referenced by several types having different type_names.</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>type_category</td>
<td>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.</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>internal</td>
<td>Whether this is an internal (built-in) or a user object.</td>
<td>BOOLEAN</td>
</tr>
</tbody>
</table>
duckdb_views

The `duckdb_views()` function provides metadata about the views available in the DuckDB instance.

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>database_name</td>
<td>The name of the database that contains this view</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>database_oid</td>
<td>Internal identifier of the database that contains this view</td>
<td>BIGINT</td>
</tr>
<tr>
<td>schema_name</td>
<td>The SQL name of the schema where the view resides.</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>schema_oid</td>
<td>Internal identifier of the schema object that contains the view.</td>
<td>BIGINT</td>
</tr>
<tr>
<td>view_name</td>
<td>The SQL name of the view object.</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>view_oid</td>
<td>The internal identifier of this view object.</td>
<td>BIGINT</td>
</tr>
<tr>
<td>internal</td>
<td>true if this is an internal (built-in) view, false if this is a user-defined view.</td>
<td>BOOLEAN</td>
</tr>
<tr>
<td>temporary</td>
<td>true if this is a temporary view. Temporary views are not persistent and are only visible within the current connection.</td>
<td>BOOLEAN</td>
</tr>
<tr>
<td>column_count</td>
<td>The number of columns defined by this view object.</td>
<td>BIGINT</td>
</tr>
<tr>
<td>sql</td>
<td>The definition of this object, expressed as SQL DDL-statement.</td>
<td>VARCHAR</td>
</tr>
</tbody>
</table>

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_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.

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>path</td>
<td>The name of the temporary file</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>size</td>
<td>The size in bytes of the temporary file</td>
<td>INT64</td>
</tr>
</tbody>
</table>
Pragmas

The PRAGMA statement is an 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).

List of Supported PRAGMA Statements

Below is a list of supported PRAGMA statements.

**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;
```

**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
nonnull 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
```

To also show table structure, but in a slightly different format (included for compatibility):

```
PRAGMA show('table_name');
```
**Memory Limit**  Set the memory limit for the buffer manager:

```sql
SET memory_limit = '1GB';
SET max_memory = '1GB';
```

Note. 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:

```sql
SET threads = 4;
```

**Database Size**  Get the file and memory size of each database:

```sql
SET 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

**Collations**  List all available collations:

```sql
PRAGMA collations;
```

Set the default collation to one of the available ones:

```sql
SET default_collation = 'nocase';
```

**Ordering**  Set the ordering for NULLs to be either NULLS FIRST or NULLS LAST:

```sql
SET default_null_order = 'NULLS FIRST';
SET default_null_order = 'NULLS LAST';
```

Set the default result set ordering direction to ASCENDING or DESCENDING:
**SET default_order** = 'ASCENDING';
**SET default_order** = 'DESCENDING';

**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 on the extension loading explainer.

**PRAGMA** platform;
**CALL** pragma_platform();

**ProgressBar**  Show progress bar when running queries:

**PRAGMA** enable_progress_bar;

Don't show a progress bar for running queries:

**PRAGMA** disable_progress_bar;

**Profiling**

**Enable Profiling**  To enable profiling:

**PRAGMA** enable_profiling;
**PRAGMA** enable_profile;

**Profiling Format**  The format of the resulting profiling information can be specified as either json, query_tree, or query_tree_optimizer. The default format is query_tree, which prints the physical operator tree together with the timings and cardinalities of each operator in the tree to the screen.

To return the logical query plan as JSON:

**SET** enable_profiling = 'json';

To return the logical query plan:

**SET** enable_profiling = 'query_tree';

To return the physical query plan:

**SET** enable_profiling = 'query_tree_optimizer';
**Disable Profiling**  To disable profiling:

```
PRAGMA disable_profiling;
PRAGMA disable_profile;
```

**Profiling Output**  By default, profiling information is printed to the console. However, if you prefer to write the profiling information to a file the PRAGMA profiling_output can be used to write to a specified file. **Note that the file contents will be overwritten for every new query that is issued, hence the file will only contain the profiling information of the last query that is run.**

```
SET profiling_output = '/path/to/file.json';
SET profile_output = '/path/to/file.json';
```

**Optimizer**  To disable the query optimizer:

```
PRAGMA disable_optimizer;
```

To enable the query optimizer:

```
PRAGMA enable_optimizer;
```

**Logging**  Set a path for query logging:

```
SET log_query_path = '/tmp/duckdb_log/';
```

Disable query logging:

```
SET log_query_path = '';
```

**Explain Plan Output**  The output of EXPLAIN output can be configured to show only the physical plan. This is the default configuration.

```
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';
```

**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 of External Operators**  Enable verification of external operators:

```sql
PRAGMA verify_external;
```

Disable verification of external operators:

```sql
PRAGMA disable_verify_external;
```

**Verification of Round-Trip Capabilities**  Enable verification of round-trip capabilities for supported logical plans:

```sql
PRAGMA verify_serializer;
```

Disable verification of round-trip capabilities:

```sql
PRAGMA disable_verify_serializer;
```

**Object Cache**  Enable caching of objects for e.g., Parquet metadata:

```sql
PRAGMA enable_object_cache;
```

Disable caching of objects:

```sql
PRAGMA disable_object_cache;
```

**Checkpoint**

**Force Checkpoint**  When `CHECKPOINT` is called when no changes are made, force a checkpoint regardless.

```sql
PRAGMA force_checkpoint;
```

**Checkpoint on Shutdown**  Run a `CHECKPOINT` on successful shutdown and delete the WAL, to leave only a single database file behind:

```sql
PRAGMA enable_checkpoint_on_shutdown;
```

Don't run a `CHECKPOINT` on shutdown:

```sql
PRAGMA disable_checkpoint_on_shutdown;
```

**Progress Bar**  Enable printing of the progress bar (if it's possible):

```sql
PRAGMA enable_print_progress_bar;
```

Disable printing of the progress bar:

```sql
PRAGMA disable_print_progress_bar;
```
**Temp Directory for Spilling Data to Disk**  By default, DuckDB uses the .tmp directory to spill to disk. To change this, use:

```
SET temp_directory = '/path/to/temp.tmp'
```

**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:

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>row_group_id</td>
<td>BIGINT</td>
<td></td>
</tr>
<tr>
<td>column_name</td>
<td>VARCHAR</td>
<td></td>
</tr>
<tr>
<td>column_id</td>
<td>BIGINT</td>
<td></td>
</tr>
<tr>
<td>column_path</td>
<td>VARCHAR</td>
<td></td>
</tr>
<tr>
<td>segment_id</td>
<td>BIGINT</td>
<td></td>
</tr>
<tr>
<td>segment_type</td>
<td>VARCHAR</td>
<td></td>
</tr>
<tr>
<td>start</td>
<td>BIGINT</td>
<td>The start row id of this chunk</td>
</tr>
<tr>
<td>count</td>
<td>BIGINT</td>
<td>The amount of entries in this storage chunk</td>
</tr>
<tr>
<td>compression</td>
<td>VARCHAR</td>
<td>Compression type used for this column - see blog post</td>
</tr>
<tr>
<td>stats</td>
<td>VARCHAR</td>
<td></td>
</tr>
<tr>
<td>has_updates</td>
<td>BOOLEAN</td>
<td></td>
</tr>
<tr>
<td>persistent</td>
<td>BOOLEAN</td>
<td>false if temporary table</td>
</tr>
<tr>
<td>block_id</td>
<td>BIGINT</td>
<td>empty unless persistent</td>
</tr>
<tr>
<td>block_offset</td>
<td>BIGINT</td>
<td>empty unless persistent</td>
</tr>
</tbody>
</table>

See [Storage](#) for more information.

**Show Databases**  The following statement is equivalent to the SHOW DATABASES statement:

```
PRAGMA show_databases;
```

**User Agent**  The following statement returns the user agent information, e.g., duckdb/v0.9.2(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).

```sql
PRAGMA metadata_info;
```

**Selectively Disabling Optimizers**  The disabled_optimizers option allows selectively disabling optimization steps. For example, to disable filter_pushdown and statistics_propagation, run:

```sql
SET disabled_optimizers = 'filter_pushdown,statistics_propagation';
```

The available optimizations can be queried using the `duckdb_optimizers()` table function.

*Note.*  The disabled_optimizers option should only be used for debugging performance issues and should be avoided in production.

**Query Verification (for Development)**  The following PRAGMAs are mostly used for development and internal testing.

Enable query verification:

```sql
PRAGMA enable_verification;
```

Disable query verification:

```sql
PRAGMA disable_verification;
```

Enable force parallel query processing:

```sql
PRAGMA verify_parallelism;
```

Disable force parallel query processing:

```sql
PRAGMA disable_verify_parallelism;
```

**Rules for Case Sensitivity**

**Keywords and Function Names**

SQL keywords and function names are case-insensitive in DuckDB.

**Examples**  The following two queries are equivalent:

```sql
select COS(Pi()) as CosineOfPi;
SELECT cos(pi()) AS CosineOfPi;
```

<table>
<thead>
<tr>
<th>CosineOfPi</th>
<th>double</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1.0</td>
<td></td>
</tr>
</tbody>
</table>
Identifiers

Following the convention of the SQL standard, identifiers in DuckDB are case-insensitive. However, each character’s case (uppercase/lowercase) is maintained as entered by the user.

To change this behavior, set the `preserve_identifier_case` configuration option to false.

Examples

Preserving Cases  The case entered by the user is preserved even if a query uses different cases when referring to the identifier:

```sql
CREATE TABLE CosPi AS SELECT cos(pi()) AS CosineOfPi;
SELECT cosineofpi FROM CosPi;
```

```
<table>
<thead>
<tr>
<th>CosineOfPi</th>
</tr>
</thead>
<tbody>
<tr>
<td>double</td>
</tr>
<tr>
<td>-1.0</td>
</tr>
</tbody>
</table>
```

Handling Conflicts  In case of a conflict, when the same identifier is spelt with different cases, one will be selected randomly. For example:

```sql
CREATE TABLE t1 (idfield INT, x INT);
CREATE TABLE t2 (IdField INT, y INT);
SELECT * FROM t1 NATURAL JOIN t2;
```

```
<table>
<thead>
<tr>
<th>idfield</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>int32</td>
<td>int32</td>
<td>int32</td>
</tr>
<tr>
<td>0 rows</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Disabling Preserving Cases  With `preserve_identifier_case` set to false, all identifiers are turned into lowercase:

```sql
SET preserve_identifier_case = false;
CREATE TABLE CosPi AS SELECT cos(pi()) AS CosineOfPi;
SELECT CosineOfPi FROM CosPi;
```

```
<table>
<thead>
<tr>
<th>cosineofpi</th>
</tr>
</thead>
<tbody>
<tr>
<td>double</td>
</tr>
</tbody>
</table>
```
**Samples**

Samples are used to randomly select a subset of a dataset.

**Examples**

```
-- select a sample of 5 rows from "tbl" using reservoir sampling
SELECT * FROM tbl USING SAMPLE 5;
-- select a sample of 10% of the table using system sampling (cluster sampling)
SELECT * FROM tbl USING SAMPLE 10%;
-- select a sample of 10% of the table using bernoulli sampling
SELECT * FROM tbl USING SAMPLE 10 PERCENT (bernoulli);
-- select a sample of 50 rows of the table using reservoir sampling with a fixed seed
SELECT * FROM tbl USING SAMPLE reservoir(50 ROWS) REPEATABLE (100);
-- select a sample of 20% of the table using system sampling with a fixed seed
SELECT * FROM tbl USING SAMPLE 10% (system, 377);
-- select a sample of 10% of "tbl" BEFORE the join with tbl2
SELECT * FROM tbl TABLESAMPLE reservoir(20%), tbl2 WHERE tbl.i = tbl2.i;
-- select a sample of 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 as a fixed number of rows (10 rows). All three sampling methods support sampling over a percentage, but only reservoir sampling supports sampling a fixed number of rows.

Samples are probabilistic, 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 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.

**Note.** Avoid using Reservoir Sample with large sample sizes if possible. Reservoir sampling requires the entire sample to be materialized in memory.

Bernoulli sampling can only be used when a sampling percentage is specified. It is rather straightforward: every tuple 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 amount of rows will generally be more or less 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 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. There is almost no extra overhead for using system sampling, whereas bernoulli sampling can add additional cost as it has to perform random number generation for every single tuple.

System sampling is not suitable for smaller data sets as the granularity of the sampling is on the order of ~1000 tuples. That means that if system sampling is used for small data sets (e.g., 100 rows) either all the data will be filtered out, or all the data will be included.

**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%);

Window Functions

Examples

-- generate a "row_number" column containing incremental identifiers for each row
SELECT row_number() OVER () FROM sales;
-- generate a "row_number" column, by order of time
SELECT row_number() OVER (ORDER BY time) FROM sales;
-- generate a "row_number" column, by order of time partitioned by region
SELECT row_number() OVER (PARTITION BY region ORDER BY time) 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;
-- 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 syntax.

General-Purpose Window Functions

The table below shows the available general window functions.

<table>
<thead>
<tr>
<th>Function</th>
<th>Return Type</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>row_number()</td>
<td>bigint</td>
<td>The number of the current row within the partition, counting from 1.</td>
<td>row_number()</td>
</tr>
<tr>
<td>rank()</td>
<td>bigint</td>
<td>The rank of the current row with gaps; same as row_number of its first peer.</td>
<td>rank()</td>
</tr>
<tr>
<td>Function</td>
<td>Return Type</td>
<td>Description</td>
<td>Example</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>dense_rank()</td>
<td>bigint</td>
<td>The rank of the current row without gaps; this function counts peer groups.</td>
<td>dense_rank()</td>
</tr>
<tr>
<td>rank_dense()</td>
<td>bigint</td>
<td>Alias for dense_rank.</td>
<td>rank_dense()</td>
</tr>
<tr>
<td>percent_rank()</td>
<td>double</td>
<td>The relative rank of the current row: (\frac{\text{rank}() - 1}{\text{total partition rows} - 1}).</td>
<td>percent_rank()</td>
</tr>
<tr>
<td>cume_dist()</td>
<td>double</td>
<td>The cumulative distribution: (\frac{\text{number of partition rows preceding or peer with current row}}{\text{total partition rows}}).</td>
<td>cume_dist()</td>
</tr>
<tr>
<td>ntile(num_buckets integer)</td>
<td>bigint</td>
<td>An integer ranging from 1 to the argument value, dividing the partition as equally as possible.</td>
<td>ntile(4)</td>
</tr>
<tr>
<td>lag(expr any [, offset integer [, default any ]])</td>
<td>same type as expr</td>
<td>Returns expr evaluated at the row that is offset rows before the current row within the partition; 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.</td>
<td>lag(column, 3, 0)</td>
</tr>
<tr>
<td>lead(expr any [, offset integer [, default any ]])</td>
<td>same type as expr</td>
<td>Returns expr evaluated at the row that is offset rows after the current row within the partition; 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.</td>
<td>lead(column, 3, 0)</td>
</tr>
</tbody>
</table>
### Function | ReturnType | Description | Example
--- | --- | --- | ---
`first_value(expr any)` | same type as `expr` | Returns `expr` evaluated at the row that is the first row of the window frame. | `first_value(column)`
`last_value(expr any)` | same type as `expr` | Returns `expr` evaluated at the row that is the last row of the window frame. | `last_value(column)`
`nth_value(expr any, nth integer)` | same type as `expr` | Returns `expr` evaluated at the `nth` row of the window frame (counting from 1); null if no such row. | `nth_value(column, 2)`

### Aggregate Window Functions

All aggregate functions can be used in a windowing context.

### Ignoring NULLs

The following functions support the `IGNORE NULLS` specification:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lag(expr any [, offset integer [, default any ]])</code></td>
<td>Skips NULL values when counting.</td>
<td><code>lag(column, 3 IGNORE NULLS)</code></td>
</tr>
<tr>
<td><code>lead(expr any [, offset integer [, default any ]])</code></td>
<td>Skips NULL values when counting.</td>
<td><code>lead(column, 3 IGNORE NULLS)</code></td>
</tr>
<tr>
<td><code>first_value(expr any)</code></td>
<td>Skips leading NULLs</td>
<td><code>first_value(column IGNORE NULLS)</code></td>
</tr>
<tr>
<td><code>last_value(expr any)</code></td>
<td>Skips trailing NULLs</td>
<td><code>last_value(column IGNORE NULLS)</code></td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td>Example</td>
</tr>
<tr>
<td>------------------------</td>
<td>------------------------------------</td>
<td>--------------------------------------</td>
</tr>
<tr>
<td>nth_value(expr any, nth integer)</td>
<td>Skips NULL values when counting.</td>
<td>nth_value(column, 2 IGNORE NULLS)</td>
</tr>
</tbody>
</table>

Note that there is no comma separating the arguments from the IGNORE NULLS specification.

The inverse of IGNORE NULLS is RESPECT NULLS, which is the default for all functions.

**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 either be specified as a number of rows or a range of values using the partition’s ordering value and a distance.

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 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:

<table>
<thead>
<tr>
<th>Plant</th>
<th>Date</th>
<th>MWh</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boston</td>
<td>2019-01-02</td>
<td>564337</td>
</tr>
<tr>
<td>Boston</td>
<td>2019-01-03</td>
<td>507405</td>
</tr>
<tr>
<td>Boston</td>
<td>2019-01-04</td>
<td>528523</td>
</tr>
<tr>
<td>Boston</td>
<td>2019-01-05</td>
<td>469538</td>
</tr>
<tr>
<td>Boston</td>
<td>2019-01-06</td>
<td>474163</td>
</tr>
<tr>
<td>Boston</td>
<td>2019-01-07</td>
<td>507213</td>
</tr>
</tbody>
</table>
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:

```sql
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:

<table>
<thead>
<tr>
<th>Plant</th>
<th>Date</th>
<th>Row</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boston</td>
<td>2019-01-02</td>
<td>1</td>
</tr>
<tr>
<td>Boston</td>
<td>2019-01-03</td>
<td>2</td>
</tr>
<tr>
<td>Boston</td>
<td>2019-01-04</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Plant</th>
<th>Date</th>
<th>MWh</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boston</td>
<td>2019-01-08</td>
<td>613040</td>
</tr>
<tr>
<td>Boston</td>
<td>2019-01-09</td>
<td>582588</td>
</tr>
<tr>
<td>Boston</td>
<td>2019-01-10</td>
<td>499506</td>
</tr>
<tr>
<td>Boston</td>
<td>2019-01-11</td>
<td>482014</td>
</tr>
<tr>
<td>Boston</td>
<td>2019-01-12</td>
<td>486134</td>
</tr>
<tr>
<td>Boston</td>
<td>2019-01-13</td>
<td>531518</td>
</tr>
<tr>
<td>Worcester</td>
<td>2019-01-02</td>
<td>118860</td>
</tr>
<tr>
<td>Worcester</td>
<td>2019-01-03</td>
<td>101977</td>
</tr>
<tr>
<td>Worcester</td>
<td>2019-01-04</td>
<td>106054</td>
</tr>
<tr>
<td>Worcester</td>
<td>2019-01-05</td>
<td>92182</td>
</tr>
<tr>
<td>Worcester</td>
<td>2019-01-06</td>
<td>94492</td>
</tr>
<tr>
<td>Worcester</td>
<td>2019-01-07</td>
<td>99932</td>
</tr>
<tr>
<td>Worcester</td>
<td>2019-01-08</td>
<td>118854</td>
</tr>
<tr>
<td>Worcester</td>
<td>2019-01-09</td>
<td>113506</td>
</tr>
<tr>
<td>Worcester</td>
<td>2019-01-10</td>
<td>96644</td>
</tr>
<tr>
<td>Worcester</td>
<td>2019-01-11</td>
<td>93806</td>
</tr>
<tr>
<td>Worcester</td>
<td>2019-01-12</td>
<td>98963</td>
</tr>
<tr>
<td>Worcester</td>
<td>2019-01-13</td>
<td>107170</td>
</tr>
</tbody>
</table>
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. This distance can either be specified as an integral number of ROWS or as a RANGE delta expression from the value of the ordering expression. For a RANGE specification, there must be only one ordering expression, and it has to support addition and subtraction (i.e., numbers or INTERVALs). The default values for frames are from UNBOUNDED PRECEDING to CURRENT ROW. It is invalid for a frame to start after it ends.

**ROW Framing**  Here is a simple ROW frame query, using an aggregate function:

```sql
SELECT points, 
    sum(points) OVER ( 
        ROWS BETWEEN 1 PRECEDING 
        AND 1 FOLLOWING) we 
FROM results;
```

This query computes the sum of each point and the points on either side of it:

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:

```sql
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:

<table>
<thead>
<tr>
<th>Plant</th>
<th>Date</th>
<th>MWh 7-day Moving Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boston</td>
<td>2019-01-02</td>
<td>517450.75</td>
</tr>
<tr>
<td>Boston</td>
<td>2019-01-03</td>
<td>508793.20</td>
</tr>
<tr>
<td>Boston</td>
<td>2019-01-04</td>
<td>508529.83</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Boston</td>
<td>2019-01-13</td>
<td>499793.00</td>
</tr>
<tr>
<td>Worcester</td>
<td>2019-01-02</td>
<td>104768.25</td>
</tr>
<tr>
<td>Worcester</td>
<td>2019-01-03</td>
<td>102713.00</td>
</tr>
<tr>
<td>Worcester</td>
<td>2019-01-04</td>
<td>102249.50</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**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:

```sql
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:

```sql
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"
```
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.

**Box and Whisker Queries** All aggregates can be used as windowing functions, including the complex statistical functions. These function implementations have been optimised for windowing, and we can use the window syntax to write queries that generate the data for moving box-and-whisker plots:

```sql
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;
```
Extensions

Overview

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.

Note. Extensions are loadable on all clients (e.g., Python and R). Extensions distributed via the official repository are built and tested on MacOS (AMD64 and ARM64), Windows (AMD64) and Linux (AMD64 and ARM64).

We maintain a list of official extensions.

Using Extensions

Listing Extensions To get a list of extensions, run:

```sql
FROM duckdb_extensions();
```

<table>
<thead>
<tr>
<th>extension_name</th>
<th>loaded</th>
<th>installed</th>
<th>description</th>
<th>aliases</th>
</tr>
</thead>
<tbody>
<tr>
<td>arrow</td>
<td>false</td>
<td>false</td>
<td>A zero-copy data integration between Apache Arrow and DuckDB</td>
<td>[]</td>
</tr>
<tr>
<td>autocompletes</td>
<td>false</td>
<td></td>
<td>Adds support for autocomplete in the shell</td>
<td>[]</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Extension Types

DuckDB has three types of extensions.
Built-In Extensions  Built-in extensions are loaded at startup and are immediately available for use.

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

This will use the `json` extension to read the JSON file.

---

Note. To make the DuckDB distribution lightweight, it only contains a few fundamental built-in extensions (e.g., `autocomplete`, `json`, `parquet`), which are loaded automatically.

---

Autoloadable Extensions  Autoloadable extensions are loaded on first use.

```sql
SELECT * FROM 'https://raw.githubusercontent.com/duckdb/duckdb-web/main/data/weather.csv';
```

To access files via the HTTPS protocol, DuckDB will automatically load the `httpfs` extension. Similarly, other autoloadable extensions (e.g., `aws`, `fts`) will be loaded on-demand. If an extension is not already available locally, it will be installed from the official extension repository (`extensions.duckdb.org`).

---

Explicitly Loadable Extensions  Some extensions make several changes to the running DuckDB instance, hence, autoloading them may not be possible. These extensions have to be installed and loaded using the following SQL statements:

```sql
INSTALL spatial;
LOAD spatial;
```

---

Extension Handling through the Python API  If you are using the Python API client, you can install and load them with the `install_extension(name: str)` and `load_extension(name: str)` methods.

---

Note. Autoloadable extensions can also be installed explicitly.

---

Ensuring the Integrity of Extensions

Extensions are signed with a cryptographic key, which also simplifies distribution (this is why they are served over HTTP and not HTTPS). By default, DuckDB uses its built-in public keys to verify the integrity of extension before loading them. All extensions provided by the DuckDB core team are signed.

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. For the Python client, see the Loading and Installing Extensions section in the Python API documentation.

---

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 0.9.2 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 0.9.2.

See the Working with Extensions page for details on available platforms.
**Installation Location**

Extensions are by default installed under the user's home directory:

```
~/.duckdb/extensions/v{duckdb_version}/{platform_name}/
```

For example, the extensions for DuckDB version 0.9.2 on macOS ARM64 (Apple Silicon) are installed to `~/.duckdb/extensions/v0.9.2/osx_arm64/`.

**Note.** For development builds, the directory of the extensions corresponds to the Git hash of the build, e.g., `~/.duckdb/extensions/fc2e4b26a6/linux_amd64_gcc4`.

**Changing the Extension Directory** To specify a different extension directory, use the `extension_directory` configuration option:

```
SET extension_directory=/path/to/your/extension/directory
```

**Developing Extensions**

The same API that the official 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.

**Working with Extensions**

For advanced installation instructions and more details on extensions, see the [Working with Extensions page](#).

**Official Extensions**

**List of Official Extensions**

<table>
<thead>
<tr>
<th>Extension Name</th>
<th>Description</th>
<th>Aliases</th>
</tr>
</thead>
<tbody>
<tr>
<td>arrow <a href="#">GitHub</a></td>
<td>A zero-copy data integration between Apache Arrow and DuckDB</td>
<td></td>
</tr>
<tr>
<td>autocomplete</td>
<td>Adds support for autocomplete in the shell</td>
<td></td>
</tr>
<tr>
<td>aws <a href="#">GitHub</a></td>
<td>Provides features that depend on the AWS SDK</td>
<td></td>
</tr>
<tr>
<td>azure <a href="#">GitHub</a></td>
<td>Adds a filesystem abstraction for Azure blob storage to DuckDB</td>
<td></td>
</tr>
<tr>
<td>excel</td>
<td>Adds support for Excel-like format strings</td>
<td></td>
</tr>
<tr>
<td>fts</td>
<td>Adds support for Full-Text Search Indexes</td>
<td></td>
</tr>
</tbody>
</table>
### Default Extensions

Different DuckDB clients ship a different set of extensions. We summarize the main distributions in the table below.

<table>
<thead>
<tr>
<th>Extension Name</th>
<th>Description</th>
<th>CLI (duckdb.org)</th>
<th>CLI (Homebrew)</th>
<th>Python</th>
<th>R</th>
<th>Java</th>
<th>Julia</th>
<th>Node.js</th>
</tr>
</thead>
<tbody>
<tr>
<td>autocomplete</td>
<td></td>
<td>yes</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>excel</td>
<td></td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fts</td>
<td></td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>httpfs</td>
<td>Adds support for reading and writing files over a HTTP(S) connection</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>icu</td>
<td>Adds support for time zones and collations using the ICU library</td>
<td>yes</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>inet</td>
<td>Adds support for IP-related data types and functions</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>icu GitHub</td>
<td></td>
<td>yes</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>icu GitHub</td>
<td></td>
<td>yes</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>json</td>
<td>Adds support for JSON operations</td>
<td>yes</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>mysql GitHub</td>
<td>Adds support for reading from and writing to a MySQL database</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>parquet</td>
<td>Adds support for reading and writing Parquet files</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>postgres GitHub</td>
<td>Adds support for reading from and writing to a Postgres database</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>spatial GitHub</td>
<td>Geospatial extension that adds support for working with spatial data and functions</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sqlite GitHub</td>
<td>Adds support for reading from and writing to SQLite database files</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>subrastait GitHub</td>
<td>Adds support for the Subrastait integration</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tpcds</td>
<td>Adds TPC-DS data generation and query support</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tpch</td>
<td>Adds TPC-H data generation and query support</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The `jemalloc` extension's availability is based on the operating system. It is a built-in extension on Linux and macOS versions, while on Windows, it is not available.

### Working with Extensions

#### Downloading Extensions Directly from S3

Downloading an extension directly could be helpful when building a lambda 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 directly download an extension file, use the following format:

```
http://extensions.duckdb.org/v{duckdb_version}/{platform_name}/{extension_name}.duckdb_extension.gz
```

For example:

```
http://extensions.duckdb.org/v{{ site.currentduckdbversion }}/windows_amd64/json.duckdb_extension.gz
```

#### Platforms

Extension binaries must be built for each platform. We distribute pre-built binaries 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:

- `linux_amd64`
- `linux_amd64_gcc4`
- `linux_arm64`
- `osx_amd64`
- `osx_arm64`
- `windows_amd64`

Only core extensions are distributed for the following platforms:

- `windows_amd64_rtools`
• wasm_eh and wasm_mvp (see DuckDB-Wasm's extensions)

We currently do not distribute binaries for extensions on the linux_arm64_gcc4 platform.

Using a Custom Extension Repository

To load extensions from a custom extension repository, set the following configuration option;

Local Files

SET custom_extension_repository = 'path/to/folder';

This assumes the pointed folder has a structure similar to:

folder
  ├── 0fd6fb9198
  │   └── osx_arm64
  │       ├── autocomplete.duckdb_extension
  │       ├── httpfs.duckdb_extension
  │       ├── icu.duckdb_extension.gz
  │       ├── inet.duckdb_extension
  │       ├── json.duckdb_extension
  │       └── parquet.duckdb_extension
  └── tpch.duckdb_extension.gz

With at the first level the DuckDB version, at the second the DuckDB platform, and then extensions either as name.duckdb_extension or gzip-compressed files name.duckdb_extension.gz.

INSTALL icu;

for example will look for either icu.duckdb_extension.gz (first) or icu.duckdb_extension (second) in the repository structure, and install it to the extension_directory (that defaults to ~/.duckdb/extensions), if file is compressed, decompression will be handled at this step.

Remote File over http

SET custom_extension_repository = 'http://nightly-extensions.duckdb.org';

They work the same as local ones, and expect a similar folder structure.

Remote Files over https or s3 Protocol

SET custom_extension_repository = 's3://bucket/your-repository-name/';

Remote extension repositories act similarly to local ones, as in the file structure should be the same and either gzipped or non-gzipped file are supported.

Only special case here is that httpfs extension should be available locally. You can get it for example doing:
RESET custom_extension_repository;
INSTALL httpfs;

That will install the official httpfs extension locally.

This is since httpfs extension will be needed to actually access remote encrypted files.

INSTALL x FROM y  You can also use the INSTALL command’s FROM clause to specify the path of the custom extension repository. For example:

FORCE INSTALL azure FROM 'http://nightly-extensions.duckdb.org';

This will force install the azure extension from the specified URL.

Loading and Installing an Extension from Explicit Paths

Installing Extensions from an Explicit Path  INSTALL can be used with the path to either a .duckdb_extension file or a .duckdb_extension.gz file. For example, if the file was available into the same directory as where DuckDB is being executed, you can install it as follows:

-- uncompressed file
INSTALL 'path/to/httpfs.duckdb_extension';
-- gzip-compressed file
INSTALL 'path/to/httpfs.duckdb_extension.gz';

These will have the same results.

It is also possible to specify remote paths.

Force Installing Extensions

When DuckDB installs an extension, it is copied to a local directory to be cached, avoiding any network traffic. Any subsequent calls to INSTALL extension_name will use the local version instead of downloading the extension again. To force re-downloading the extension, run:

by default in ~/.duckdb/extensions but configurable via SET extension_directory = path/to/existing/directory;

FORCE INSTALL extension_name;

Loading Extension from a 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:

-- uncompressed file
LOAD 'path/to/httpfs.duckdb_extension';
This will skip any currently installed file in the specified path.
Using remote paths for compressed files is currently not possible.

**Building Extensions**

Build the extension following the extension’s README.

**Statically Linking Extensions**

To statically link extensions, follow the developer documentation’s "Using extension config files" section.

**Arrow Extension**

The arrow extension implements provides features for using Apache Arrow, a cross-language development platform for in-memory analytics.

**Installing and Loading**

The arrow 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 arrow;
LOAD arrow;
```

**Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>to_arrow_ipc</code></td>
<td>Table in-out-function</td>
<td>Serializes a table into a stream of blobs containing Arrow IPC buffers</td>
</tr>
<tr>
<td><code>scan_arrow_ipc</code></td>
<td>Table function</td>
<td>Scan a list of pointers pointing to Arrow IPC buffers</td>
</tr>
</tbody>
</table>

**GitHub Repository**

GitHub
AutoComplete Extension

The autocomplete extension adds support for autocomplete in the CLI client. The extension is shipped by default with the CLI client.

Auto-Completion Rules

The DuckDB 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

Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sql_auto_complete(query_string)</td>
<td>Attempts autocompletion on the given query_string.</td>
</tr>
</tbody>
</table>

Example

```sql
SELECT * FROM sql_auto_complete('SEL');
```

Returns:

<table>
<thead>
<tr>
<th>suggestion</th>
<th>suggestion_start</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT</td>
<td>0</td>
</tr>
<tr>
<td>DELETE</td>
<td>0</td>
</tr>
<tr>
<td>INSERT</td>
<td>0</td>
</tr>
<tr>
<td>CALL</td>
<td>0</td>
</tr>
<tr>
<td>LOAD</td>
<td>0</td>
</tr>
</tbody>
</table>
GitHub

The autocomplete extension is part of the main DuckDB repository.

AWS Extension

The aws extension provides features that depend on the AWS SDK.

**Note.** This extension is currently in an experimental state. Feel free to try it out, but be aware some things may not work as expected.

Installing and Loading

aws depends on httpfs extension capabilities, and both will be autoloaded on the first call to `load_aws_credentials`. If autoinstall or autoload are disabled, you can always explicitly install and load httpfs and aws like:

```
INSTALL aws;
LOAD aws;
```
INSTALL httpfs;
LOAD httpfs;

See also the S3 API capabilities of the httpfs extension.

**Features**

<table>
<thead>
<tr>
<th>function</th>
<th>type</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>load_aws_credentials</td>
<td>PRAGMA function</td>
<td>Automatically loads the AWS credentials through the AWS Default Credentials Provider Chain</td>
</tr>
</tbody>
</table>

**Usage**

**Load AWS Credentials**  To load the AWS credentials, run:

```sql
CALL load_aws_credentials();
```

<table>
<thead>
<tr>
<th>loaded_access_key_id</th>
<th>loaded_secret_access_key</th>
<th>loaded_session_token</th>
<th>loaded_region</th>
</tr>
</thead>
<tbody>
<tr>
<td>varchar</td>
<td>varchar</td>
<td>varchar</td>
<td>varchar</td>
</tr>
<tr>
<td>AKIAIOSFODNN7EXAMPLE</td>
<td>&lt;redacted&gt;</td>
<td></td>
<td>eu-west-1</td>
</tr>
</tbody>
</table>

The function takes a string parameter to specify a specific profile:

```sql
CALL load_aws_credentials('minio-testing-2');
```

<table>
<thead>
<tr>
<th>loaded_access_key_id</th>
<th>loaded_secret_access_key</th>
<th>loaded_session_token</th>
<th>loaded_region</th>
</tr>
</thead>
<tbody>
<tr>
<td>varchar</td>
<td>varchar</td>
<td>varchar</td>
<td>varchar</td>
</tr>
<tr>
<td>minio_duckdb_user_2</td>
<td>&lt;redacted&gt;</td>
<td></td>
<td>eu-west-2</td>
</tr>
</tbody>
</table>

There are several parameters to tweak the behavior of the call:

```sql
CALL load_aws_credentials('minio-testing-2', set_region = false, redact_secret = false);
```
The azure extension is a loadable extension that adds a filesystem abstraction for the Azure Blob storage to DuckDB.

**Note.** This extension is currently in an experimental state. Feel free to try it out, but be aware some things may not work as expected.

### Installing and Loading

To install and load the azure extension, run:

```
INSTALL azure;
LOAD azure;
```

### Usage

Authentication is done by setting the connection string:

```
SET azure_storage_connection_string = '<your_connection_string>';
```

After setting the connection string, the Azure Blob Storage can be queried:

```
SELECT count(*) FROM 'azure://<my_container>/<my_file>.<parquet_or_csv>'
```

Blobs are also supported:

```
SELECT * FROM 'azure://<my_container>/*.csv'
```
Excel Extension

This extension, contrary to its name, does not provide support for reading Excel files. It instead provides a function that wraps the number formatting functionality of the `i18npool` library, which formats numbers per Excel's formatting rules.

Excel files can be handled through the spatial extension: 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;
```

Usage

```
SELECT excel_text(1234567.897, 'h:mm AM/PM') AS timestamp;
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>timestamp</td>
<td>varchar</td>
</tr>
<tr>
<td>9:31 PM</td>
<td></td>
</tr>
</tbody>
</table>

Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>text(number, format_string)</td>
<td>Format the given number per the rules given in the format_string</td>
<td>text(1234567.897, 'h AM/PM')</td>
<td>9 PM</td>
</tr>
<tr>
<td>excel_text(number, format_string)</td>
<td>Alias for text.</td>
<td>excel_text(1234567.897, 'h:mm AM/PM')</td>
<td>9:31 PM</td>
</tr>
</tbody>
</table>

GitHub

The excel extension is part of the main DuckDB repository.
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.

```sql
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)
```

This PRAGMA that creates a FTS index for the specified table.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>input_table</td>
<td>VARCHAR</td>
<td>Qualified name of specified table, e.g., 'table_name' or 'main.table_name'</td>
</tr>
<tr>
<td>input_id</td>
<td>VARCHAR</td>
<td>Column name of document identifier, e.g., 'document_identifier'</td>
</tr>
<tr>
<td>input_values</td>
<td>VARCHAR</td>
<td>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</td>
</tr>
<tr>
<td>stemmer</td>
<td>VARCHAR</td>
<td>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'</td>
</tr>
</tbody>
</table>
### stopwords

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stopwords</td>
<td>VARCHAR</td>
<td>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</td>
</tr>
</tbody>
</table>

### ignore

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ignore</td>
<td>VARCHAR</td>
<td>Regular expression of patterns to be ignored. Defaults to '\.</td>
</tr>
</tbody>
</table>

### strip_accents

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>strip_accents</td>
<td>BOOLEAN</td>
<td>Whether to remove accents (e.g., convert á to a). Defaults to 1</td>
</tr>
</tbody>
</table>

### lower

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lower</td>
<td>BOOLEAN</td>
<td>Whether to convert all text to lowercase. Defaults to 1</td>
</tr>
</tbody>
</table>

### overwrite

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>overwrite</td>
<td>BOOLEAN</td>
<td>Whether to overwrite an existing index on a table. Defaults to 0</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>input_table</td>
<td>VARCHAR</td>
<td>Qualified name of input table, e.g., 'table_name' or 'main.table_name'</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>input_id</td>
<td>VARCHAR</td>
<td>Column name of document identifier, e.g., 'document_identifier'</td>
</tr>
<tr>
<td>query_string</td>
<td>VARCHAR</td>
<td>The string to search the index for</td>
</tr>
</tbody>
</table>
### Name Type Description

**fields** | VARCHAR | Comma-separated 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 $k_1$ 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

**stemmer** | VARCHAR | The type of stemmer to be used. One of 'arabic', 'basque', 'catalan', '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":

```sql
SELECT document_identifier, text_content, score
FROM
  (SELECT *
   FROM 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;
```

Search for documents about "small cats". This retrieves "doc2":

```sql
SELECT document_identifier, text_content, score
FROM
  (SELECT *
   FROM fts_main_documents.match_bm25(document_identifier, 'small cats')
   AS score
   FROM documents
  ) sq
WHERE score IS NOT NULL
ORDER BY score DESC;
```

Note. The FTS index will not update automatically when input table changes. A workaround of this limitation can be recreating the index to refresh.

GitHub

The `fts` extension is part of the main DuckDB repository.

**httpfs Extension**

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.

The `httpfs` extension will be, by default, autoloaded on first use of any functionality exposed by this extension. If you prefer to explicitly install this extension and load it at the start of every session, use the following commands:

```sql
INSTALL httpfs;
LOAD httpfs;
```
Running Queries over HTTP(S)

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.

```sql
SELECT * FROM 'https://domain.tld/file.extension';
```

For CSV files, files will be downloaded entirely in most cases, due to the row-based nature of the format. For Parquet files, DuckDB 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:

```sql
SELECT column_a FROM 'https://domain.tld/file.parquet';
```

In some cases even, no actual data needs to be read at all as they only require reading the metadata:

```sql
SELECT count(*) FROM 'https://domain.tld/file.parquet';
```

Scanning multiple files over HTTP(S) is also supported:

```sql
SELECT * FROM read_parquet(['https://domain.tld/file1.parquet', 'https://domain.tld/file2.parquet']);
```

```sql
-- parquet_scan is an alias of read_parquet, so they are equivalent
SELECT * FROM parquet_scan(['https://domain.tld/file1.parquet', 'https://domain.tld/file2.parquet']);
```

Running Queries over S3

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.

**Requirements** The `httpfs` filesystem is tested with AWS S3, Minio, Google Cloud, and lakeFS. Other services that implement the S3 API should also work, but not all features may be supported. Below is a list of which parts of the S3 API are required for each `httpfs` feature.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Required S3 API features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Public file reads</td>
<td>HTTP Range requests</td>
</tr>
<tr>
<td>Private file reads</td>
<td>Secret key or session token authentication</td>
</tr>
<tr>
<td>File glob</td>
<td>ListObjectV2</td>
</tr>
<tr>
<td>File writes</td>
<td>Multipart upload</td>
</tr>
</tbody>
</table>
**Configuration**  
To be able to read or write from S3, the correct region should be set:

```sql
SET s3_region = 'us-east-1';
```

Optionally, the endpoint can be configured in case a non-AWS object storage server is used:

```sql
SET s3_endpoint = '<domain>.<tld>:<port>';
```

If the endpoint is not SSL-enabled then run:

```sql
SET s3_use_ssl = false;
```

Switching between **path-style** and **vhost-style** URLs is possible using:

```sql
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`.

After configuring the correct endpoint and region, public files can be read. To also read private files, authentication credentials can be added:

```sql
SET s3_access_key_id = '<AWS access key id>';  
SET s3_secret_access_key = '<AWS secret access key>';  
```

Alternatively, session tokens are also supported and can be used instead:

```sql
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:

```sql
SELECT * FROM 's3://bucket/file.parquet?s3_access_key_id=accessKey&s3_secret_access_key=secretKey';
```

Multiple configurations per query are also allowed:

```sql
SELECT * FROM 's3://bucket/file.parquet?s3_region=region&s3_session_token=session_token' T1 INNER JOIN 's3://bucket/file.csv?s3_access_key_id=accessKey&s3_secret_access_key=secretKey' T2;
```

**Reading**  
Reading files from S3 is now as simple as:

```sql
SELECT * FROM 's3://bucket/file.extension';
```

Multiple files are also possible, for example:

```sql
SELECT * FROM read_parquet(['s3://bucket/file1.parquet', 's3://bucket/file2.parquet']);
```
Glob  File globbing is implemented using the ListObjectV2 API call and allows to use filesystem-like glob patterns to match multiple files, for example:

```
SELECT * FROM read_parquet('s3://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://bucket/folder*/100?/t[0-9].parquet');
```

A useful feature when using globs is the `FILENAME` option which adds a column with the file that a row originated from:

```
SELECT * FROM read_parquet('s3://bucket/*.parquet', FILENAME = 1);
```

could for example result in:

<table>
<thead>
<tr>
<th>column_a</th>
<th>column_b</th>
<th>filename</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>examplevalue1</td>
<td>s3://bucket/file1.parquet</td>
</tr>
<tr>
<td></td>
<td>examplevalue1</td>
<td>s3://bucket/file2.parquet</td>
</tr>
</tbody>
</table>

Hive Partitioning  DuckDB also offers support for the Hive partitioning scheme. In the Hive partitioning scheme, data is partitioned in separate files. The columns by which the data is partitioned, are not actually in the files, but are encoded in the file path. So for example let us consider three Parquet files Hive partitioned by year:

```
s3://bucket/year=2012/file.parquet
s3://bucket/year=2013/file.parquet
s3://bucket/year=2014/file.parquet
```

If scanning these files with the `HIVE_PARTITIONING` option enabled:

```
SELECT * FROM read_parquet('s3://bucket/*\file.parquet', HIVE_PARTITIONING = 1);
```

could result in:

<table>
<thead>
<tr>
<th>column_a</th>
<th>column_b</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>examplevalue1</td>
<td>2012</td>
</tr>
<tr>
<td></td>
<td>examplevalue2</td>
<td>2013</td>
</tr>
<tr>
<td></td>
<td>examplevalue3</td>
<td>2014</td>
</tr>
</tbody>
</table>

Note that the year column does not actually exist in the Parquet files, it is parsed from the filenames. Within DuckDB however, these columns behave just like regular columns. For example, filters can be applied on Hive partition columns:
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://bucket/file.extension';
```

Partitioned copy to S3 also works:

```
COPY table TO 's3://my-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://my-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://my-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.

<table>
<thead>
<tr>
<th>setting</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>s3_uploader_max_parts_per_file</td>
<td>used for part size calculation, see AWS docs</td>
</tr>
<tr>
<td>s3_uploader_max_filesize</td>
<td>used for part size calculation, see AWS docs</td>
</tr>
<tr>
<td>s3_uploader_thread_limit</td>
<td>maximum number of uploader threads</td>
</tr>
</tbody>
</table>

Additionally, most of the configuration options can be set via environment variables:

<table>
<thead>
<tr>
<th>DuckDB setting</th>
<th>Environment variable</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>s3_region</td>
<td>AWS_REGION</td>
<td>Takes priority over AWS_DEFAULT_REGION</td>
</tr>
<tr>
<td>s3_region</td>
<td>AWS_DEFAULT_REGION</td>
<td></td>
</tr>
<tr>
<td>s3_access_key_id</td>
<td>AWS_ACCESS_KEY_ID</td>
<td></td>
</tr>
<tr>
<td>s3_secret_access_key</td>
<td>AWS_SECRET_ACCESS_KEY</td>
<td></td>
</tr>
<tr>
<td>s3_session_token</td>
<td>AWS_SESSION_TOKEN</td>
<td></td>
</tr>
<tr>
<td>s3_endpoint</td>
<td>DUCKDB_S3_ENDPOINT</td>
<td></td>
</tr>
</tbody>
</table>
GitHub

The httpfs extension is part of the main DuckDB repository.

Iceberg Extension

The iceberg extension is a loadable extension that implements support for the Apache Iceberg format.

Installing and Loading

To install and load the iceberg extension, run:

```
INSTALL iceberg;
LOAD iceberg;
```

Usage

To test the examples, download the iceberg_data.zip file and unzip it.

Querying Individual Tables

```sql
SELECT count(*) FROM iceberg_scan('data/iceberg/lineitem_iceberg', allow_moved_paths = true);
```

51793

Note. The allow_moved_paths option ensures that some path resolution is performed, which allows scanning Iceberg tables that are moved.

Access Iceberg Metadata

```sql
SELECT * FROM iceberg_metadata('data/iceberg/lineitem_iceberg', allow_moved_paths = true);
```

```
<table>
<thead>
<tr>
<th>manifest_path</th>
<th>manifest_sequence_number</th>
<th>manifest_content</th>
<th>status</th>
<th>content</th>
<th>file_path</th>
<th>file_format</th>
<th>record_count</th>
</tr>
</thead>
</table>
```

668
DuckDB Documentation

<table>
<thead>
<tr>
<th>varchar</th>
<th>varchar</th>
<th>varchar</th>
<th>varchar</th>
<th>varchar</th>
</tr>
</thead>
<tbody>
<tr>
<td>varchar</td>
<td>varchar</td>
<td>int64</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sequence_number</th>
<th>snapshot_id</th>
<th>timestamp_ms</th>
<th>manifest_list</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>uint64</td>
<td>timestamp</td>
<td></td>
</tr>
<tr>
<td>varchar</td>
<td>varchar</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3776207205136740581</td>
<td>2023-02-15 15:07:54.504</td>
<td>lineitem_iceberg/metadata/snap-3776207205136740581-1-cf3d0be5-cf70-453d-ad8f-48fdc412e608.avro</td>
</tr>
<tr>
<td>2</td>
<td>7635660646343998149</td>
<td>2023-02-15 15:08:14.73</td>
<td>lineitem_iceberg/metadata/snap-7635660646343998149-1-10eaca8a-1e1c-421e-ad6d-b232e5ee23d3.avro</td>
</tr>
</tbody>
</table>

Visualizing Snapshots

**SELECT * FROM iceberg_snapshots('data/iceberg/lineitem_iceberg');**

<table>
<thead>
<tr>
<th>sequence_number</th>
<th>snapshot_id</th>
<th>timestamp_ms</th>
<th>manifest_list</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3776207205136740581</td>
<td>2023-02-15 15:07:54.504</td>
<td>lineitem_iceberg/metadata/snap-3776207205136740581-1-cf3d0be5-cf70-453d-ad8f-48fdc412e608.avro</td>
</tr>
<tr>
<td>2</td>
<td>7635660646343998149</td>
<td>2023-02-15 15:08:14.73</td>
<td>lineitem_iceberg/metadata/snap-7635660646343998149-1-10eaca8a-1e1c-421e-ad6d-b232e5ee23d3.avro</td>
</tr>
</tbody>
</table>

GitHub Repository

GitHub

ICU Extension

The `icu` extension contains an easy-to-use version of the collation/timezone part of the ICU library.

Installing and Loading

To install and load the `icu` extension, run:
Features

The icu extension introduces the following features:

- region-dependent collations
- time zones, used for timestamp data types and timestamp functions

GitHub

The icu extension is part of the main DuckDB repository.

inet Extension

The inet extension defines the INET data type for storing IPv4 network addresses. It supports the CIDR notation for subnet masks (e.g., 198.51.100.0/22).

Installing and Loading

To install and load the inet extension, run:

```
INSTALL inet;
LOAD inet;
```

Examples

```
SELECT '127.0.0.1'::INET AS addr;

<table>
<thead>
<tr>
<th>addr</th>
<th>inet</th>
</tr>
</thead>
<tbody>
<tr>
<td>127.0.0.1</td>
<td></td>
</tr>
</tbody>
</table>

CREATE TABLE tbl (id INTEGER, ip INET);
INSERT INTO tbl VALUES (1, '192.168.0.0/16'), (2, '127.0.0.1'), (2, '8.8.8.8');

<table>
<thead>
<tr>
<th>id</th>
<th>ip</th>
</tr>
</thead>
<tbody>
<tr>
<td>int32</td>
<td>inet</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
The `inet` extension is part of the main DuckDB repository.

**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.

**Availability**

The Linux and macOS versions of DuckDB ship with the `jemalloc` extension by default. On Windows, this extension is not available.

**GitHub**

The `jemalloc` extension is part of the main DuckDB repository.

**JSON Extension**

The `json` extension is a loadable extension that implements SQL functions that are useful for reading values from existing JSON, and creating new JSON data.

**Installing and Loading**

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;
```
Example Uses

-- read a JSON file from disk, auto-infer options
SELECT * FROM 'todos.json';

-- read_json with custom options
SELECT *
FROM read_json('todos.json',
format = 'array',
columns = {userId: 'UBIGINT',
id: 'UBIGINT',
title: 'VARCHAR',
completed: 'BOOLEAN'});

-- write the result of a query to a JSON file
COPY (SELECT * FROM todos) TO 'todos.json';

See more examples on the JSON data page.

JSON Type

The JSON extension makes use of 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. All JSON creation functions return values of this type.

We also allow any of our types to be casted to JSON, and JSON to be casted back to any of our types, for example:

-- Cast JSON to our STRUCT type
SELECT '{"duck": 42}::JSON::STRUCT(duck INTEGER);

-- And back:
SELECT {duck: 42}::JSON;

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

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

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 Table Functions

The following two 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 parameter format set to 'newline_delimited'

read_json_objects_auto(filename) Alias for read_json_objects with parameter format set to 'auto'

These functions have the following parameters:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Type</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>maximum_object_size</td>
<td>The maximum size of a JSON object (in bytes)</td>
<td>UINT32</td>
<td>16777216</td>
</tr>
<tr>
<td>format</td>
<td>Can be one of ['auto', 'unstructured', 'newline_delimited', 'array']</td>
<td>VARCHAR</td>
<td>'array'</td>
</tr>
<tr>
<td>ignore_errors</td>
<td>Whether to ignore parse errors (only possible when format is 'newline_delimited')</td>
<td>BOOL</td>
<td>false</td>
</tr>
<tr>
<td>compression</td>
<td>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'.</td>
<td>VARCHAR</td>
<td>'auto'</td>
</tr>
<tr>
<td>filename</td>
<td>Whether or not an extra filename column should be included in the result.</td>
<td>BOOL</td>
<td>false</td>
</tr>
<tr>
<td>hive_partitioning</td>
<td>Whether or not to interpret the path as a hive partitioned path.</td>
<td>BOOL</td>
<td>false</td>
</tr>
</tbody>
</table>

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

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

Will result in two objects being read.

With 'newline_delimited', NDJSON is read, where each JSON is separated by a newline (\n), e.g.:
{ "duck": 42 }
{ "goose": [1, 2, 3] }

Will also result in two objects being read.

With 'array', each array element is read, e.g.:

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

Again, will result in two objects being read.

Example usage:

```
SELECT * FROM read_json_objects('my_file1.json');
-- {"duck":42,"goose":[1,2,3]}
SELECT * FROM read_json_objects(["my_file1.json", 'my_file2.json']);
-- {"duck":42,"goose":[1,2,3]}
-- {"duck":43,"goose":[4,5,6],"swan":3.3}
SELECT * FROM read_ndjson_objects('*.json.gz');
-- {"duck":42,"goose":[1,2,3]}
-- {"duck":43,"goose":[4,5,6],"swan":3.3}
```

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>read_json(filename)</td>
<td>Read JSON from filename, where filename can also be a list of files, or a glob pattern</td>
</tr>
<tr>
<td>read_ndjson(filename)</td>
<td>Alias for read_json with parameter format set to 'newline_delimited'</td>
</tr>
<tr>
<td>read_json_auto(filename)</td>
<td>Alias for read_json with all auto-detection enabled</td>
</tr>
<tr>
<td>read_ndjson_auto(filename)</td>
<td>Alias for read_json_auto with parameter format set to 'newline_delimited'</td>
</tr>
</tbody>
</table>

Besides the maximum_object_size, format, ignore_errors and compression, these functions have additional parameters:
### DuckDB Documentation

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Type</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>columns</strong></td>
<td>A struct that specifies the key names and value types contained within the JSON file (e.g., <code>{key1: 'INTEGER', key2: 'VARCHAR'}</code>). If <code>auto_detect</code> is enabled these will be inferred.</td>
<td>STRUCT</td>
<td>(empty)</td>
</tr>
<tr>
<td><strong>records</strong></td>
<td>Can be one of [&quot;auto&quot;, &quot;true&quot;, &quot;false&quot;]</td>
<td>VARCHAR</td>
<td>'records'</td>
</tr>
<tr>
<td><strong>auto_detect</strong></td>
<td>Whether to auto-detect the names of the keys and data types of the values automatically.</td>
<td>BOOL</td>
<td>false</td>
</tr>
<tr>
<td><strong>sample_size</strong></td>
<td>Option to define number of sample objects for automatic JSON type detection. Set to -1 to scan the entire input file.</td>
<td>UBIGINT</td>
<td>20480</td>
</tr>
<tr>
<td><strong>maximum_depth</strong></td>
<td>Maximum nesting depth to which the automatic schema detection detects types. Set to -1 to fully detect nested JSON types.</td>
<td>BIGINT</td>
<td>-1</td>
</tr>
<tr>
<td><strong>dateformat</strong></td>
<td>Specifies the date format to use when parsing dates. See Date Format.</td>
<td>VARCHAR</td>
<td>'iso'</td>
</tr>
<tr>
<td><strong>timestampformat</strong></td>
<td>Specifies the date format to use when parsing timestamps. See Date Format.</td>
<td>VARCHAR</td>
<td>'iso'</td>
</tr>
<tr>
<td><strong>union_by_name</strong></td>
<td>Whether the schema's of multiple JSON files should be unified.</td>
<td>BOOL</td>
<td>false</td>
</tr>
</tbody>
</table>

**Example usage:**

```sql
SELECT * FROM read_json('my_file1.json', columns = {duck: 'INTEGER'});
```

```mermaid
diagram
duck
42

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

```sql
SELECT *
FROM read_json(['my_file1.json', 'my_file2.json'],
                columns = {duck: 'INTEGER', goose: 'INTEGER[]', swan: 'DOUBLE'});
```

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

---

675
DuckDB can automatically detect the types like so:

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

<table>
<thead>
<tr>
<th>goose</th>
<th>duck</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1, 2, 3]</td>
<td>42</td>
</tr>
<tr>
<td>[4, 5, 6]</td>
<td>43</td>
</tr>
</tbody>
</table>

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.:

```json
[
    {
        "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.:

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

Both can be read as the table:

<table>
<thead>
<tr>
<th>duck</th>
<th>goose</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>4.2</td>
</tr>
<tr>
<td>43</td>
<td>4.3</td>
</tr>
</tbody>
</table>

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, i.e., reading the following file with `records`:
Results in two columns:

<table>
<thead>
<tr>
<th>duck</th>
<th>goose</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>[1,2,3]</td>
</tr>
</tbody>
</table>

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

```
<table>
<thead>
<tr>
<th>json</th>
</tr>
</thead>
<tbody>
<tr>
<td>{'duck': 42, 'goose': [1,2,3]}</td>
</tr>
<tr>
<td>{'duck': 43, 'goose': [4,5,6]}</td>
</tr>
</tbody>
</table>
```

For additional examples reading more complex data, please see the [Shredding Deeply Nested JSON, One Vector at a Time blog post](https://example.com).

**JSON Import/Export**

When the JSON extension is installed, `FORMAT JSON` is supported for `COPY FROM`, `COPY TO`, `EXPORT` `DATABASE` and `IMPORT` `DATABASE`. See [Copy and Import/Export](https://example.com).

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

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

Will create the following file:

```
[
  {'range':0},
  {'range':1},
  {'range':2},
  {'range':3},
  {'range':4}
]
```

This can be read like so:

```
CREATE TABLE test (range BIGINT);
COPY test FROM 'my.json' (ARRAY true);
```

The format can be detected automatically the format like so:

```
COPY test FROM 'my.json' (AUTO_DETECT true);
```
**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](https://tools.ietf.org/html/rfc6901) and [JSONPath](https://jsonpath.org).

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>json(json)</code></td>
<td>Parse and minify <code>json</code></td>
</tr>
<tr>
<td><code>json_valid(json)</code></td>
<td>Return whether <code>json</code> is valid JSON</td>
</tr>
<tr>
<td><code>json_array_length(json[,path])</code></td>
<td>Return the number of elements in the JSON array <code>json</code>, or 0 if it is not a JSON array. If <code>path</code> is specified, return the number of elements in the JSON array at the given <code>path</code>. If <code>path</code> is a LIST, the result will be LIST of array lengths</td>
</tr>
<tr>
<td><code>json_type(json[,path])</code></td>
<td>Return the type of the supplied <code>json</code>, which is one of <code>OBJECT</code>, <code>ARRAY</code>, <code>BIGINT</code>, <code>UBIGINT</code>, <code>VARCHAR</code>, <code>BOOLEAN</code>, <code>NULL</code>. If <code>path</code> is specified, return the type of the element at the given <code>path</code>. If <code>path</code> is a LIST, the result will be LIST of types</td>
</tr>
<tr>
<td><code>json_keys(json[,path])</code></td>
<td>Returns the keys of <code>json</code> as a LIST of VARCHAR, if <code>json</code> is a JSON object. If <code>path</code> is specified, return the keys of the JSON object at the given <code>path</code>. If <code>path</code> is a LIST, the result will be LIST of LIST of VARCHAR</td>
</tr>
<tr>
<td><code>json_structure(json)</code></td>
<td>Return the structure of <code>json</code>. Defaults to JSON the structure is inconsistent (e.g., incompatible types in an array)</td>
</tr>
<tr>
<td><code>json_contains(json_haystack, json_needle)</code></td>
<td>Returns true if <code>json_needle</code> is contained in <code>json_haystack</code>. Both parameters are of JSON type, but <code>json_needle</code> can also be a numeric value or a string, however the string must be wrapped in double quotes</td>
</tr>
</tbody>
</table>

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

```sql
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:

```sql
SELECT json_extract('"duck": [1, 2, 3]', '$.duck[0]');
-- 1
```
DuckDB Documentation
Note that DuckDB's JSON data type uses 0‑based indexing.
JSONPath is more expressive, and can also access from the back of lists:
SELECT json_extract('{"duck": [1, 2, 3]}', '$.duck[#-1]');
-- 3

JSONPath also allows escaping syntax tokens, using double quotes:
SELECT json_extract('{"duck.goose": [1, 2, 3]}', '$."duck.goose"[1]');
-- 2

Other examples:
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 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 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

679


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 the previous functions.

<table>
<thead>
<tr>
<th>Function</th>
<th>Alias</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>json_extract(json, path)</code></td>
<td><code>json_extract_path</code></td>
<td><code>-&gt;</code></td>
</tr>
<tr>
<td><code>json_extract_string(json, path)</code></td>
<td><code>json_extract_path_text</code></td>
<td><code>-&gt;&gt;&gt;</code></td>
</tr>
</tbody>
</table>

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

Examples:

```sql
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->>'$.species'->>'0' FROM example;
  -- "duck"
SELECT j->>'species'->>'[0,1]' FROM example;
  -- ["duck", "goose"]
SELECT json_extract_string(j, '$.family') FROM example;
  -- anatidae
SELECT j->>'$.family' FROM example;
  -- anatidae
SELECT j->>'$.species[0]' FROM example;
  -- duck
SELECT j->>'species'->>'8' 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:

```sql
-- The following will cause the JSON to be parsed twice,
-- resulting in a slower query that uses more memory
```
```sql
SELECT json_extract(j, 'family') AS family,
       json_extract(j, 'species') AS species
FROM example;
-- The following is faster and more memory efficient
WITH extracted AS (
  SELECT json_extract(j, ['family', 'species']) extracted_list
       FROM example
)
SELECT extracted_list[1] AS family,
       extracted_list[2] AS species
FROM extracted;
```

### JSON Creation Functions

The following functions are used to create JSON.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>to_json(any)</code></td>
<td>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</td>
</tr>
<tr>
<td><code>json_quote(any)</code></td>
<td>Alias for <code>to_json</code></td>
</tr>
<tr>
<td><code>array_to_json(list)</code></td>
<td>Alias for <code>to_json</code> that only accepts LIST</td>
</tr>
<tr>
<td><code>row_to_json(list)</code></td>
<td>Alias for <code>to_json</code> that only accepts STRUCT</td>
</tr>
<tr>
<td><code>json_array([any, ...])</code></td>
<td>Create a JSON array from any number of values</td>
</tr>
<tr>
<td><code>json_object([key, value, ...])</code></td>
<td>Create a JSON object from any number of key, value pairs</td>
</tr>
<tr>
<td><code>json_merge_patch(json, json)</code></td>
<td>Merge two JSON documents together</td>
</tr>
</tbody>
</table>

Examples:

```sql
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(42, 'duck', NULL);
-- [42,"duck",null]
SELECT json_object('duck', 42);
-- {"duck":42}
```
DuckDB Documentation
SELECT json_merge_patch('{"duck": 42}', '{"goose": 123}');
-- {"goose":123,"duck":42}

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 example (k VARCHAR, v INTEGER);
INSERT INTO example VALUES ('duck', 42), ('goose', 7);
SELECT json_group_array(v) FROM example;
-- [42, 7]
SELECT json_group_object(k, v) FROM example;
-- {"duck":42,"goose":7}
DROP TABLE example;
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_group_structure(j) FROM example;
-- {"family":"VARCHAR","species":["VARCHAR"],"coolness":"DOUBLE","hair":"BOOLEAN"}

Transforming JSON
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

682


DuckDB Documentation

Function

Description

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"

De/Serializing SQL to JSON and Vice Versa
The JSON extension also provides functions to serialize and deserialize SELECT statements between SQL and
JSON, as well as executing JSON serialized statements.
Function

Type

Description

json_serialize_sql(
varchar, skip_empty
:=boolean, skip_null
:=boolean, format
:=boolean)

Scalar

Serialize a set of ; separated select statments to an
equivalent list of json serialized statements

json_deserialize_sql( json)

Scalar

Deserialize one or many json serialized statements
back to an equivalent sql string

683


**Function Type Description**

<table>
<thead>
<tr>
<th>Function</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>json_execute_serialized_sql(varchar)</td>
<td>Table</td>
<td>Execute json serialized statements and return the resulting rows. Only one statement at a time is supported for now.</td>
</tr>
<tr>
<td>PRAGMA json_execute_serialized_sql(varchar)</td>
<td>Pragma</td>
<td>Pragma version of the json_execute_serialized_sql function.</td>
</tr>
</tbody>
</table>

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:**

```sql
-- 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":null,"value":{"type":{"id":"INTEGER","type_info":null},"is_null":false,"value":2}],"from_table":{"type":"EMPTY","alias":null,"sample":null},"where_clause":null,"group_expressions":[]},"group_sets":null,"aggregate_handling":"STANDARD_HANDLING","having":null,"sample":null,"qualify":null}]}'

-- 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);
```
-- 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"
LINE 1: TOTALLY NOT VALID SQL
   ^'}

-- Example with deserialize
SELECT json_deserialize_sql(json_serialize_sql('SELECT 1 + 2'));
-- 'SELECT (1 + 2)'

-- Example with deserialize and syntax sugar
SELECT json_deserialize_sql(json_serialize_sql('FROM x SELECT 1 + 2'));
-- 'SELECT (1 + 2) FROM x'

-- Example with execute
SELECT * FROM json_execute_serialized_sql(json_serialize_sql('SELECT 1 + 2'));
-- 3

-- Example with error
SELECT * FROM json_execute_serialized_sql(json_serialize_sql('TOTALLY NOT VALID SQL'));
-- Error: Parser Error: Error parsing json: parser: syntax error at or near "TOTALLY"

Indexing

Following PostgreSQL's conventions, DuckDB uses 1-based indexing for arrays and lists but 0-based indexing for the JSON data type.

GitHub

The json extension is part of the main DuckDB repository.
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:

```
ATTACH 'host=localhost user=root port=0 database=mysql' AS mysql (TYPE mysql)
USE mysql;
```

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.

<table>
<thead>
<tr>
<th>Setting</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>host</td>
<td>localhost</td>
</tr>
<tr>
<td>user</td>
<td>current user</td>
</tr>
<tr>
<td>password</td>
<td></td>
</tr>
<tr>
<td>database</td>
<td>NULL</td>
</tr>
<tr>
<td>port</td>
<td>0</td>
</tr>
<tr>
<td>socket</td>
<td>NULL</td>
</tr>
</tbody>
</table>

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 TABLES;
```

<table>
<thead>
<tr>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>varchar</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>signed_integers</th>
</tr>
</thead>
</table>

686
### SELECT * FROM signed_integers;

<table>
<thead>
<tr>
<th>t</th>
<th>s</th>
<th>m</th>
<th>i</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>int8</td>
<td>int16</td>
<td>int32</td>
<td>int32</td>
<td>int64</td>
</tr>
<tr>
<td>-128</td>
<td>-32768</td>
<td>-8388608</td>
<td>-2147483648</td>
<td>-9223372036854775808</td>
</tr>
<tr>
<td>127</td>
<td>32767</td>
<td>8388607</td>
<td>2147483647</td>
<td>9223372036854775807</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>

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:

```sql
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.

```sql
ATTACH 'host=localhost user=root port=0 database=mysqlscanner' AS mysql_db (TYPE mysql_scanner);
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:

```sql
ATTACH 'host=localhost user=root port=0 database=mysqlscanner' AS mysql_db (TYPE mysql_scanner, READ_ONLY);
```

### Supported Operations

Below is a list of supported operations.

```sql
CREATE TABLE mysql_db.tbl(id INTEGER, name VARCHAR);
```
**INSERT INTO**

```sql
INSERT INTO mysql_db.tbl VALUES (42, 'DuckDB');
```

**SELECT**

```sql
SELECT * FROM mysql_db.tbl;
```

```
<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>DuckDB</td>
</tr>
</tbody>
</table>
```

**COPY**

```sql
COPY mysql_db.tbl TO 'data.parquet';
COPY mysql_db.tbl FROM 'data.parquet';
```

**UPDATE**

```sql
UPDATE mysql_db.tbl SET name = 'Woohoo' WHERE id = 42;
```

**DELETE**

```sql
DELETE FROM mysql_db.tbl WHERE id = 42;
```

**ALTER TABLE**

```sql
ALTER TABLE mysql_db.tbl ADD COLUMN k INTEGER;
```

**DROP TABLE**

```sql
DROP TABLE mysql_db.tbl;
```

**CREATE VIEW**

```sql
CREATE VIEW mysql_db.v1 AS SELECT 42;
```

**CREATE SCHEMA and DROP SCHEMA**

```sql
CREATE SCHEMA mysql_db.s1;
CREATE TABLE mysql_db.s1.integers(i int);
INSERT INTO mysql_db.s1.integers VALUES (42);
SELECT * FROM mysql_db.s1.integers;
```
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;

```
 i  
<table>
<thead>
<tr>
<th>int64</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
</tr>
</tbody>
</table>
```

ROLLBACK;
SELECT * FROM mysql_db.tmp;

```
 i  
<table>
<thead>
<tr>
<th>int64</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
</tr>
</tbody>
</table>
```

---

**Note.** Note that DDL statements are not transactional in MySQL.

**GitHub Repository**

GitHub

**PostgreSQL Extension**

The *postgres* extension allows DuckDB to directly read and write data from a running Postgres database instance. The data can be queried directly from the underlying Postgres database. Data can be loaded from Postgres tables into DuckDB tables, or vice versa. See the official announcement for implementation details and background.
Installing and Loading

To install the postgres extension, run:

```
INSTALL postgres;
```

The extension is loaded automatically upon first use. If you prefer to load it manually, run:

```
LOAD postgres;
```

Usage

To make a PostgreSQL database accessible to DuckDB, use the ATTACH command:

```
-- connect to the "public" schema of the postgres instance running on localhost
ATTACH 'public' AS postgres_db (TYPE postgres);

-- connect to the Postgres instance with the given parameters
ATTACH 'dbname=postgres user=postgres host=127.0.0.1' AS db (TYPE postgres);
```

The ATTACH command takes as input a `libpq connection string` - which is a set of `key=value` pairs separated by spaces. Below are some example connection strings and commonly used parameters. A full list of available parameters can be found in the Postgres documentation.

```
dbname=postgres
host=localhost
port=5432
dbname=mydb
connect_timeout=10
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>host</td>
<td>Name of host to connect to</td>
<td>localhost</td>
</tr>
<tr>
<td>hostaddr</td>
<td>Host IP address</td>
<td>localhost</td>
</tr>
<tr>
<td>port</td>
<td>Port Number</td>
<td>5432</td>
</tr>
<tr>
<td>user</td>
<td>Postgres User Name</td>
<td>[OS user name]</td>
</tr>
<tr>
<td>password</td>
<td>Postgres Password</td>
<td></td>
</tr>
<tr>
<td>dbname</td>
<td>Database Name</td>
<td>[user]</td>
</tr>
<tr>
<td>passfile</td>
<td>Name of file passwords are stored in</td>
<td>~/.pgpass</td>
</tr>
</tbody>
</table>

The tables in the PostgreSQL database can be read as if they were normal DuckDB tables, but the underlying data is read directly from Postgres at query time.

```
D SHOW TABLES;
```

<table>
<thead>
<tr>
<th>name</th>
<th>varchar</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>uuids</td>
</tr>
</tbody>
</table>

690
DuckDB Documentation

```
D  SELECT * FROM uuids;
```

```
+---+---------------------+
| u | uuid                |
+---+---------------------+
| 6d3d2541-710b-4bde-b3af-4711738636bf | NULL |
| 00000000-0000-0000-0000-000000000001 | NULL |
+---+---------------------+
```

It might be desirable to create a copy of the Postgres databases in DuckDB to prevent the system from re-reading the tables from Postgres continuously, particularly for large tables.

Data can be copied over from Postgres to DuckDB using standard SQL, for example:

```
CREATE TABLE duckdb_table AS FROM postgres_db.postgres_tbl;
```

**Writing Data to Postgres**

In addition to reading data from Postgres, the extension allows you to create tables, ingest data into Postgres and make other modifications to a Postgres database using standard SQL queries.

This allows you to use DuckDB to, for example, export data that is stored in a Postgres database to Parquet, or read data from a Parquet file into Postgres.

Below is a brief example of how to create a new table in Postgres and load data into it.

```
ATTACH 'dbname=postgresqlserver' 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 Postgres tables are supported. All these operations directly modify the Postgres database, and the result of subsequent operations can then be read using Postgres. 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=postgresqlserver' 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;

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>DuckDB</td>
</tr>
</tbody>
</table>

COPY

COPY postgres_db.tbl TO 'data.parquet';
COPY postgres_db.tbl FROM 'data.parquet';

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 int);
INSERT INTO postgres_db.s1.integers VALUES (42);
SELECT * FROM postgres_db.s1.integers;

<table>
<thead>
<tr>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
</tr>
</tbody>
</table>

DROP SCHEMA postgres_db.s1;
Transactions

```sql
CREATE TABLE postgres_db.tmp(i INTEGER);
BEGIN;
INSERT INTO postgres_db.tmp VALUES (42);
SELECT * FROM postgres_db.tmp;
```

<table>
<thead>
<tr>
<th>i</th>
<th>int64</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td></td>
</tr>
</tbody>
</table>

ROLLBACK;
SELECT * FROM postgres_db.tmp;

<table>
<thead>
<tr>
<th>i</th>
<th>int64</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>rows</td>
</tr>
</tbody>
</table>

Running SQL Queries in Postgres with `postgres_query`

The `postgres_query` function allows you to run arbitrary SQL within an attached database. `postgres_query` takes the name of the attached Postgres database to execute the query in, as well as the SQL query to execute. The result of the query is returned.

```
postgres_query(attached_database::VARCHAR, query::VARCHAR)
```

Example

```
ATTACH 'dbname=postgresscanner' AS s (TYPE POSTGRES);
SELECT * FROM postgres_query("s", 'SELECT * FROM cars LIMIT 3');
```

<table>
<thead>
<tr>
<th>brand</th>
<th>model</th>
<th>color</th>
</tr>
</thead>
<tbody>
<tr>
<td>ferari</td>
<td>testarosa</td>
<td>red</td>
</tr>
<tr>
<td>aston martin</td>
<td>db2</td>
<td>blue</td>
</tr>
<tr>
<td>bentley</td>
<td>mulsanne</td>
<td>gray</td>
</tr>
</tbody>
</table>

Settings

The extension exposes the following configuration parameters.
### Querying Individual Tables

If you prefer to not attach all tables, but just query a single table, that is possible using the `postgres_scan` function, e.g.:

```sql
SELECT * FROM postgres_scan('', 'public', 'mytable');
```

The `postgres_scan` function takes three string parameters, the `libpq` connection string (see above), a PostgreSQL schema name and a table name. The schema often used in PostgreSQL is `public`.

To use `filter_pushdown` use the `postgres_scan_pushdown` function.

**Note.** The old `postgres_attach` function is deprecated. It is recommended to switch over to the new `ATTACH` syntax.

### GitHub Repository

[GitHub](https://github.com)

### Spatial Extension

The spatial extension provides support for geospatial data processing in DuckDB. For an overview of the extension, see our [blog post](https://example.com).

### Installing and Loading

To install and load the spatial extension, run:

```
INSTALL spatial;
LOAD spatial;
```
**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.

In short, while 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), it actually stores one of several 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 GEOMETRYCOLLECTIONs 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 behaviour 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.

That said, 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 DuckDB’s native nested types, such as structs and lists. In theory it should be possible to optimize a lot of operations for these types much better than for the GEOMETRY type (which is just a binary blob), but only a couple functions are implemented so far.

All of these are implicitly castable to GEOMETRY but with a 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, Z/M coordinates, or SRID information. These features may be added in the future.

**Spatial Scalar Functions**

The spatial extension implements a large number of scalar functions and overloads. Most of these are implemented using the GEOS library, but we’d like to implement more of them natively in this extension to better utilize DuckDB’s vectorized execution and memory management. The following symbols are used to indicate which implementation is used:

- **GEOS** - functions that are implemented using the GEOS library
- **DuckDB** - functions that are implemented natively in this extension that are capable of operating directly on the DuckDB types
- **CAST (GEOMETRY)** - functions that are supported by implicitly casting to GEOMETRY and then using the GEOMETRY implementation

The currently implemented spatial functions can roughly be categorized into the following groups:

**Geometry Conversion**  Convert between geometries and other formats.
### Scalar functions

<table>
<thead>
<tr>
<th>GEOMETRY</th>
<th>POINT_2D</th>
<th>LINESTRING</th>
<th>POLYGON_2D</th>
<th>BOX_2D</th>
</tr>
</thead>
<tbody>
<tr>
<td>VARCHAR ST_</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗ (as POLYGON)</td>
</tr>
<tr>
<td>AsText(GEOMETRY)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WKB_BLOB ST_</td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AsWKB(GEOMETRY)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VARCHAR ST_</td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AsHEXWKB(GEOMETRY)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VARCHAR ST_</td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AsGeoJSON(GEOMETRY)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GEOMETRY ST_</td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GeomFromText(VARCHAR)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GEOMETRY ST_</td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GeomFromWKB(BLOB)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GEOMETRY ST_</td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GeomFromHEXWKB(VARCHAR)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GEOMETRY ST_</td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GeomFromGeoJSON(VARCHAR)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Geometry Construction

Construct new geometries from other geometries or other data.

<table>
<thead>
<tr>
<th>GEOMETRY</th>
<th>POINT_2D</th>
<th>LINESTRING</th>
<th>POLYGON_2D</th>
<th>BOX_2D</th>
</tr>
</thead>
<tbody>
<tr>
<td>GEOMETRY ST_</td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Point(DOUBLE, DOUBLE)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GEOMETRY ST_</td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ConvexHull(GEOMETRY)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GEOMETRY ST_</td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Boundary(GEOMETRY)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GEOMETRY ST_</td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Buffer(GEOMETRY)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GEOMETRY ST_</td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Centroid(GEOMETRY)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GEOMETRY ST_</td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Collect(GEOMETRY[])</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Scalar functions

<table>
<thead>
<tr>
<th>GEOMETRY ST_</th>
<th>GEOMETRY</th>
<th>POINT_2D</th>
<th>LINESTRING</th>
<th>POLYGON_2D</th>
<th>BOX_2D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normalize(GEOMETRY)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Simplify(GEOMETRY)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SimplifyPreserveTopology(GEOMETRY, DOUBLE)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Union(GEOMETRY, GEOMETRY)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intersection(GEOMETRY, GEOMETRY)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MakeLine(GEOMETRY[])</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Envelope(GEOMETRY)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FlipCoordinates(GEOMETRY)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transform(GEOMETRY, VARCHAR, VARCHAR)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Extent(GEOMETRY)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PointN(GEOMETRY, INTEGER)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>StartPoint(GEOMETRY)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EndPoint(GEOMETRY)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ExteriorRing(GEOMETRY)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reverse(GEOMETRY)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Scalar functions

<table>
<thead>
<tr>
<th>GEOMETRY ST_</th>
<th>GEOMETRY</th>
<th>POINT_2D</th>
<th>LINESTRING_</th>
<th>POLYGON_</th>
<th>BOX_2D</th>
</tr>
</thead>
<tbody>
<tr>
<td>RemoveRepeatedPoints(GEOMETRY)</td>
<td>🔴</td>
<td>🔴</td>
<td>🔴</td>
<td>🔴</td>
<td>🔴</td>
</tr>
<tr>
<td>RemoveRepeatedPoints(GEOMETRY, DOUBLE)</td>
<td>🔴</td>
<td>🔴</td>
<td>🔴</td>
<td>🔴</td>
<td>🔴</td>
</tr>
<tr>
<td>ReducePrecision(GEOMETRY, DOUBLE)</td>
<td>🔴</td>
<td>🔴</td>
<td>🔴</td>
<td>🔴</td>
<td>🔴</td>
</tr>
<tr>
<td>PointOnSurface(GEOMETRY)</td>
<td>🔴</td>
<td></td>
<td>🔴</td>
<td></td>
<td>🔴</td>
</tr>
<tr>
<td>CollectionExtract(GEOMETRY)</td>
<td>🔴</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CollectionExtract(GEOMETRY, INTEGER)</td>
<td>🔴</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Spatial Properties

Calculate and access spatial properties of geometries.

<table>
<thead>
<tr>
<th>GEOMETRY ST_</th>
<th>GEOMETRY</th>
<th>POINT_2D</th>
<th>LINESTRING_</th>
<th>POLYGON_</th>
<th>BOX_2D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Area(GEOMETRY)</td>
<td>🔴</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IsClosed(GEOMETRY)</td>
<td>🔴</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IsEmpty(GEOMETRY)</td>
<td>🔴</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IsRing(GEOMETRY)</td>
<td>🔴</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IsSimple(GEOMETRY)</td>
<td>🔴</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IsValid(GEOMETRY)</td>
<td>🔴</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X(GEOMETRY)</td>
<td>🔴</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y(GEOMETRY)</td>
<td>🔴</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Scalar functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Geometry</th>
<th>Point_2D</th>
<th>Linestring_2D</th>
<th>Polygon_2D</th>
<th>Box_2D</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOUBLE ST_ XMax(GEOMETRY)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>DOUBLE ST_ YMax(GEOMETRY)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>DOUBLE ST_ XMin(GEOMETRY)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>DOUBLE ST_ YMin(GEOMETRY)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>GeometryType ST_ GeometryType(GEOMETRY)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>DOUBLE ST_ Length(GEOMETRY)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>INTEGER ST_ NGeometries(GEOMETRY)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>INTEGER ST_ NPoints(GEOMETRY)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>INTEGER ST_ NInteriorRings(GEOMETRY)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

**Spatial Relationships**  Compute relationships and spatial predicates between geometries.

<table>
<thead>
<tr>
<th>Function</th>
<th>Geometry</th>
<th>Point_2D</th>
<th>Linestring_2D</th>
<th>Polygon_2D</th>
<th>Box_2D</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOLEAN ST_ Within(GEOMETRY, GEOMETRY)</td>
<td>✓</td>
<td>✓</td>
<td>✓ or ✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>BOOLEAN ST_ Touches(GEOMETRY, GEOMETRY)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>BOOLEAN ST_ Overlaps(GEOMETRY, GEOMETRY)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>BOOLEAN ST_ Contains(GEOMETRY, GEOMETRY)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>
### Scalar functions

<table>
<thead>
<tr>
<th>Function</th>
<th>GEOMETRY</th>
<th>POINT_2D</th>
<th>LINESTRING_2D</th>
<th>POLYGON_2D</th>
<th>BOX_2D</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOLEAN ST_ CoveredBy(GEOMETRY, GEOMETRY)</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>BOOLEAN ST_ Covers(GEOMETRY, GEOMETRY)</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>BOOLEAN ST_ Crosses(GEOMETRY, GEOMETRY)</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>BOOLEAN ST_ Difference(GEOMETRY, GEOMETRY)</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>BOOLEAN ST_ Disjoint(GEOMETRY, GEOMETRY)</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>BOOLEAN ST_ Intersects(GEOMETRY, GEOMETRY)</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>BOOLEAN ST_ Equals(GEOMETRY, GEOMETRY)</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>DOUBLE ST_ Distance(GEOMETRY, GEOMETRY)</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>BOOLEAN ST_ DWithin(GEOMETRY, GEOMETRY, DOUBLE)</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>BOOLEAN ST_ Intersects_Extent(GEOMETRY, GEOMETRY)</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
</tbody>
</table>

### Spatial Aggregate Functions
### Spatial Table Functions

**ST_Read() - Read spatial data from files**  
The spatial extension provides a `ST_Read` table function based on the GDAL translator library to read spatial data from a variety of geospatial vector file formats as if they were DuckDB tables. For example to create a new table from a GeoJSON file, you can use the following query:

```sql
CREATE TABLE <table> AS SELECT * FROM ST_Read('some/file/path/filename.json');
```

`ST_Read` can take a number of optional arguments, the full signature is:

```sql
ST_Read(CHARACTER VARYING, sequential_layer_scan : BOOLEAN, spatial_filter : WKB BLOB, open_options : CHARACTER VARYING[], layer : CHARACTER VARYING, allowed_drivers : CHARACTER VARYING[], sibling_files : CHARACTER VARYING[], spatial_filter_box : BOX_2D, keep_wkb : BOOLEAN)
```

- `sequential_layer_scan` (default: false): 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.
- `spatial_filter` (default: NULL): 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` (default: []): 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` (default: NULL): 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` (default: []): A list of GDAL driver names that are allowed to be used to open the file. If empty, all drivers are allowed.
- `sibling_files` (default: []): 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` (default: NULL): 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` (default: false): 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 doesn't 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. We're planning to implement support for the most common vector formats natively in this extension with additional
We currently support over 50 different formats. You can generate the following table of supported GDAL drivers yourself by executing `SELECT * FROM ST_Drivers()`.

<table>
<thead>
<tr>
<th>short_name</th>
<th>long_name</th>
<th>can_create</th>
<th>can_copy</th>
<th>can_open</th>
<th>help_url</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESRI Shapefile</td>
<td>ESRI Shapefile</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td><a href="https://gdal.org/drivers/vector/shapefile.html">https://gdal.org/drivers/vector/shapefile.html</a></td>
</tr>
<tr>
<td>MapInfo File</td>
<td>MapInfo File</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td><a href="https://gdal.org/drivers/vector/mitab.html">https://gdal.org/drivers/vector/mitab.html</a></td>
</tr>
<tr>
<td>LVBAG</td>
<td>Kadaster LV BAG Extract 2.0</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td><a href="https://gdal.org/drivers/vector/lvbag.html">https://gdal.org/drivers/vector/lvbag.html</a></td>
</tr>
<tr>
<td>S57</td>
<td>IHO S-57 (ENC)</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td><a href="https://gdal.org/drivers/vector/s57.html">https://gdal.org/drivers/vector/s57.html</a></td>
</tr>
<tr>
<td>DGN</td>
<td>Microstation DGN</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td><a href="https://gdal.org/drivers/vector/dgn.html">https://gdal.org/drivers/vector/dgn.html</a></td>
</tr>
<tr>
<td>OGR_VRT</td>
<td>VRT - Virtual Datasource</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td><a href="https://gdal.org/drivers/vector/vrt.html">https://gdal.org/drivers/vector/vrt.html</a></td>
</tr>
<tr>
<td>Memory</td>
<td>Memory</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td><a href="https://gdal.org/drivers/vector/csv.html">https://gdal.org/drivers/vector/csv.html</a></td>
</tr>
<tr>
<td>CSV</td>
<td>Comma Separated Value (.csv)</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td><a href="https://gdal.org/drivers/vector/gml.html">https://gdal.org/drivers/vector/gml.html</a></td>
</tr>
<tr>
<td>GML</td>
<td>Geography Markup Language (GML)</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td><a href="https://gdal.org/drivers/vector/gml.html">https://gdal.org/drivers/vector/gml.html</a></td>
</tr>
<tr>
<td>GPX</td>
<td>GPX</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td><a href="https://gdal.org/drivers/vector/kml.html">https://gdal.org/drivers/vector/kml.html</a></td>
</tr>
<tr>
<td>KML</td>
<td>Keyhole Markup Language (KML)</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td><a href="https://gdal.org/drivers/vector/geojson.html">https://gdal.org/drivers/vector/geojson.html</a></td>
</tr>
<tr>
<td>GeoJSON</td>
<td>GeoJSON</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td><a href="https://gdal.org/drivers/vector/geojson.html">https://gdal.org/drivers/vector/geojson.html</a></td>
</tr>
<tr>
<td>GeoJSONSeq</td>
<td>GeoJSON Sequence</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td><a href="https://gdal.org/drivers/vector/geojsonseq.html">https://gdal.org/drivers/vector/geojsonseq.html</a></td>
</tr>
<tr>
<td>ESRIJSON</td>
<td>ESRIJSON</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td><a href="https://gdal.org/drivers/vector/esrijson.html">https://gdal.org/drivers/vector/esrijson.html</a></td>
</tr>
<tr>
<td>TopoJSON</td>
<td>TopoJSON</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td><a href="https://gdal.org/drivers/vector/topojson.html">https://gdal.org/drivers/vector/topojson.html</a></td>
</tr>
<tr>
<td>short_name</td>
<td>long_name</td>
<td>can_create</td>
<td>can_copy</td>
<td>can_open</td>
<td>help_url</td>
</tr>
<tr>
<td>--------------</td>
<td>------------------------------------</td>
<td>------------</td>
<td>----------</td>
<td>----------</td>
<td>---------------------------------------------------</td>
</tr>
<tr>
<td>OGR_GMT</td>
<td>GMT ASCII Vectors (.gmt)</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td><a href="https://gdal.org/drivers/vector/gmt.html">https://gdal.org/drivers/vector/gmt.html</a></td>
</tr>
<tr>
<td>GPKG</td>
<td>GeoPackage</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td><a href="https://gdal.org/drivers/vector/gpkg.html">https://gdal.org/drivers/vector/gpkg.html</a></td>
</tr>
<tr>
<td>SQLite</td>
<td>SQLite / Spatialite</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td><a href="https://gdal.org/drivers/vector/sqlite.html">https://gdal.org/drivers/vector/sqlite.html</a></td>
</tr>
<tr>
<td>WASP</td>
<td>WAsP .map format</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td><a href="https://gdal.org/drivers/vector/wasp.html">https://gdal.org/drivers/vector/wasp.html</a></td>
</tr>
<tr>
<td>OpenFileGDB</td>
<td>ESRI FileGDB</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td><a href="https://gdal.org/drivers/vector/openfilegdb.html">https://gdal.org/drivers/vector/openfilegdb.html</a></td>
</tr>
<tr>
<td>DXF</td>
<td>AutoCAD DXF</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td><a href="https://gdal.org/drivers/vector/dxf.html">https://gdal.org/drivers/vector/dxf.html</a></td>
</tr>
<tr>
<td>CAD</td>
<td>AutoCAD Driver</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td><a href="https://gdal.org/drivers/vector/cad.html">https://gdal.org/drivers/vector/cad.html</a></td>
</tr>
<tr>
<td>FlatGeobuf</td>
<td>FlatGeobuf</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td><a href="https://gdal.org/drivers/vector/flatgeobuf.html">https://gdal.org/drivers/vector/flatgeobuf.html</a></td>
</tr>
<tr>
<td>Geoconcept</td>
<td>Geoconcept</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td><a href="https://gdal.org/drivers/vector/georss.html">https://gdal.org/drivers/vector/georss.html</a></td>
</tr>
<tr>
<td>GeoRSS</td>
<td>GeoRSS</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td><a href="https://gdal.org/drivers/vector/georss.html">https://gdal.org/drivers/vector/georss.html</a></td>
</tr>
<tr>
<td>PGDUMP</td>
<td>PostgreSQL SQL dump</td>
<td>true</td>
<td>false</td>
<td>false</td>
<td><a href="https://gdal.org/drivers/vector/pgdump.html">https://gdal.org/drivers/vector/pgdump.html</a></td>
</tr>
<tr>
<td>OSM</td>
<td>OpenStreetMap XML and PBF</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td><a href="https://gdal.org/drivers/vector/osm.html">https://gdal.org/drivers/vector/osm.html</a></td>
</tr>
<tr>
<td>GPSBabel</td>
<td>GPSBabel</td>
<td>true</td>
<td>false</td>
<td>false</td>
<td><a href="https://gdal.org/drivers/vector/gpsbabel.html">https://gdal.org/drivers/vector/gpsbabel.html</a></td>
</tr>
<tr>
<td>WFS</td>
<td>OGC WFS (Web Feature Service)</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td><a href="https://gdal.org/drivers/vector/wfs.html">https://gdal.org/drivers/vector/wfs.html</a></td>
</tr>
<tr>
<td>OAPIF</td>
<td>OGC API - Features</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td><a href="https://gdal.org/drivers/vector/oapif.html">https://gdal.org/drivers/vector/oapif.html</a></td>
</tr>
<tr>
<td>EDIGEO</td>
<td>French EDIGEO exchange format</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td><a href="https://gdal.org/drivers/vector/edigeo.html">https://gdal.org/drivers/vector/edigeo.html</a></td>
</tr>
<tr>
<td>SVG</td>
<td>Scalable Vector Graphics</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td><a href="https://gdal.org/drivers/vector/svg.html">https://gdal.org/drivers/vector/svg.html</a></td>
</tr>
<tr>
<td>short_name</td>
<td>long_name</td>
<td>can_create</td>
<td>can_copy</td>
<td>can_open</td>
<td>help_url</td>
</tr>
<tr>
<td>------------</td>
<td>-----------</td>
<td>------------</td>
<td>----------</td>
<td>----------</td>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>XLSX</td>
<td>MS Office Open XML spreadsheet</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td><a href="https://gdal.org/drivers/vector/xlsx.html">https://gdal.org/drivers/vector/xlsx.html</a></td>
</tr>
<tr>
<td>Elasticsearch</td>
<td>Elastic Search</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td><a href="https://gdal.org/drivers/vector/elasticsearch.html">https://gdal.org/drivers/vector/elasticsearch.html</a></td>
</tr>
<tr>
<td>Carto</td>
<td>Carto</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td><a href="https://gdal.org/drivers/vector/carto.html">https://gdal.org/drivers/vector/carto.html</a></td>
</tr>
<tr>
<td>AmigoCloud</td>
<td>AmigoCloud</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td><a href="https://gdal.org/drivers/vector/amigocloud.html">https://gdal.org/drivers/vector/amigocloud.html</a></td>
</tr>
<tr>
<td>SXF</td>
<td>Storage and eXchange Format</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td><a href="https://gdal.org/drivers/vector/sxf.html">https://gdal.org/drivers/vector/sxf.html</a></td>
</tr>
<tr>
<td>Selafin</td>
<td>Selafin</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td><a href="https://gdal.org/drivers/vector/selafin.html">https://gdal.org/drivers/vector/selafin.html</a></td>
</tr>
<tr>
<td>JML</td>
<td>OpenJUMP JML</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td><a href="https://gdal.org/drivers/vector/jml.html">https://gdal.org/drivers/vector/jml.html</a></td>
</tr>
<tr>
<td>PLSCENES</td>
<td>Planet Labs Scenes API</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td><a href="https://gdal.org/drivers/vector/plscenes.html">https://gdal.org/drivers/vector/plscenes.html</a></td>
</tr>
<tr>
<td>CSW</td>
<td>OGC CSW (Catalog Service for the Web)</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td><a href="https://gdal.org/drivers/vector/csw.html">https://gdal.org/drivers/vector/csw.html</a></td>
</tr>
<tr>
<td>MVT</td>
<td>Mapbox Vector Tiles</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td><a href="https://gdal.org/drivers/vector/mvt.html">https://gdal.org/drivers/vector/mvt.html</a></td>
</tr>
<tr>
<td>NGW</td>
<td>NextGIS Web</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td><a href="https://gdal.org/drivers/vector/ngw.html">https://gdal.org/drivers/vector/ngw.html</a></td>
</tr>
<tr>
<td>MapML</td>
<td>MapML</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td><a href="https://gdal.org/drivers/vector/mapml.html">https://gdal.org/drivers/vector/mapml.html</a></td>
</tr>
<tr>
<td>TIGER</td>
<td>U.S. Census TIGER/Line</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td><a href="https://gdal.org/drivers/vector/tiger.html">https://gdal.org/drivers/vector/tiger.html</a></td>
</tr>
<tr>
<td>AVCBin</td>
<td>Arc/Info Binary Coverage</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td><a href="https://gdal.org/drivers/vector/avcbin.html">https://gdal.org/drivers/vector/avcbin.html</a></td>
</tr>
<tr>
<td>AVCE00</td>
<td>Arc/Info E00 (ASCII) Coverage</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td><a href="https://gdal.org/drivers/vector/avce00.html">https://gdal.org/drivers/vector/avce00.html</a></td>
</tr>
</tbody>
</table>
Note that far from all of these drivers have been tested properly, and some may require additional options to be passed to work as expected. If you run into any issues please first consult the GDAL docs.

**ST_ReadOsm() - Read compressed OSM data** The spatial extension also provides an experimental `ST_ReadOsm()` table function to read compressed OSM data directly from a `.osm.pbf` file.

This will use multithreading and zero-copy protobuf parsing which makes it a lot faster than using the `st_read()` OSM driver, but it only outputs the raw OSM data (Nodes, Ways, Relations), without constructing any geometries. For node entities you can trivially construct `POINT` geometries, but it is also possible to construct `LINESTRING` and `POLYGON` by manually joining refs and nodes together in SQL.

Example usage:

```sql
SELECT * FROM st_readosm('tmp/data/germany.osm.pbf')
WHERE tags['highway'] != []
LIMIT 5;
```

```
<table>
<thead>
<tr>
<th>kind</th>
<th>id</th>
<th>tags</th>
<th>refs</th>
<th>lat</th>
<th>lon</th>
<th>ref_roles</th>
<th>ref_types</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
<td>122351</td>
<td>{bicycle=yes, butt...</td>
<td></td>
<td>53.5492951</td>
<td>9.977553</td>
<td></td>
<td></td>
</tr>
<tr>
<td>node</td>
<td>122397</td>
<td>{crossing=no, high...</td>
<td></td>
<td>53.520990100000006</td>
<td>10.0156924</td>
<td></td>
<td></td>
</tr>
<tr>
<td>node</td>
<td>122493</td>
<td>{TMC:cid_58:tabcd...</td>
<td></td>
<td>53.129614600000004</td>
<td>8.1970173</td>
<td></td>
<td></td>
</tr>
<tr>
<td>node</td>
<td>123566</td>
<td>{highway=traffic_s...</td>
<td></td>
<td>54.617268200000005</td>
<td>8.9718171</td>
<td></td>
<td></td>
</tr>
<tr>
<td>node</td>
<td>125801</td>
<td>{TMC:cid_58:tabcd...</td>
<td></td>
<td>53.070685000000005</td>
<td>8.7819939</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Spatial replacement scans**

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.

```sql
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:

- ESRI ShapeFile, .shp
• GeoPackage, .gpkg
• FlatGeoBuf, .fgb

Similarly there is a .osm.pbf replacement scan for ST_ReadOsm.

**Spatial Copy Functions**

Much like the ST_Read table function the spatial extension provides a GDAL based COPY function to export duckdb tables to different geospatial vector formats. For example to export a table to a GeoJSON file, with generated bounding boxes, you can use the following query:

```sql
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. See the table above for a list of 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/OGR. This will not perform any reprojection of the input geometry though, it just sets the metadata if the target driver supports it.

**GitHub Repository**

[GitHub](https://github.com/DuckDB)

**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**

To install the sqlite extension, run:

`INSTALL sqlite;`

The extension is loaded automatically upon first use. If you prefer to load it manually, run:

`LOAD sqlite;`
Usage

To make a SQLite file accessible to DuckDB, use the ATTACH statement, which supports read & write.

For example with the sakila.db file:

```
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;
```

```
<table>
<thead>
<tr>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>actor</td>
</tr>
<tr>
<td>address</td>
</tr>
<tr>
<td>category</td>
</tr>
<tr>
<td>city</td>
</tr>
<tr>
<td>country</td>
</tr>
<tr>
<td>customer</td>
</tr>
<tr>
<td>customer_list</td>
</tr>
<tr>
<td>film</td>
</tr>
<tr>
<td>film_actor</td>
</tr>
<tr>
<td>film_category</td>
</tr>
<tr>
<td>film_list</td>
</tr>
<tr>
<td>film_text</td>
</tr>
<tr>
<td>inventory</td>
</tr>
<tr>
<td>language</td>
</tr>
<tr>
<td>payment</td>
</tr>
<tr>
<td>rental</td>
</tr>
<tr>
<td>sales_by_film_category</td>
</tr>
<tr>
<td>sales_by_store</td>
</tr>
<tr>
<td>staff</td>
</tr>
<tr>
<td>staff_list</td>
</tr>
<tr>
<td>store</td>
</tr>
</tbody>
</table>
```

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 film_cat
    ON cat.category_id = film_cat.category_id
LEFT JOIN film fil
    ON film_cat.film_id = fil.film_id
```
$\text{LEFT JOIN} \text{ inventory inv} \quad \text{ON} \quad \text{fil.film_id} = \text{inv.film_id}$

$\text{LEFT JOIN} \text{ rental ren} \quad \text{ON} \quad \text{inv.inventory_id} = \text{ren.inventory_id}$

$\text{LEFT JOIN} \text{ payment pay} \quad \text{ON} \quad \text{ren.rental_id} = \text{pay.rental_id}$

$\text{GROUP BY cat.name}$

$\text{ORDER BY} \text{ revenue DESC}$

$\text{LIMIT 5;}$

### 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:

```sql
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:

```
Error: 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:

```sql
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:

```
$ > duckdb data/db/sakila.db
D SHOW tables;
```

```
+-----------------+
| name             |
| varchar          |
+-----------------+
| actor            |
| address          |
| category         |
| staff_list       |
| store            |
+-----------------+
21 rows
(5 shown)
```

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.

```
$ r > 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.

Below is a list of supported operations.

**CREATE TABLE**

```sql
CREATE TABLE sqlite_db.tbl(id INTEGER, name VARCHAR);
```

**INSERT INTO**

```sql
INSERT INTO sqlite_db.tbl VALUES (42, 'DuckDB');
```

**SELECT**

```sql
SELECT * FROM sqlite_db.tbl;
```

```
<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>DuckDB</td>
</tr>
</tbody>
</table>
```

**COPY**

```sql
COPY sqlite_db.tbl TO 'data.parquet';
COPY sqlite_db.tbl FROM 'data.parquet';
```

**UPDATE**

```sql
UPDATE sqlite_db.tbl SET name = 'Woohoo' WHERE id = 42;
```

**DELETE**

```sql
DELETE FROM sqlite_db.tbl WHERE id = 42;
```

**ALTER TABLE**

```sql
ALTER TABLE sqlite_db.tbl ADD COLUMN k INTEGER;
```

**DROP TABLE**

```sql
DROP TABLE sqlite_db.tbl;
```

**CREATE VIEW**

```sql
CREATE VIEW sqlite_db.v1 AS SELECT 42;
```
Transactions

```sql
CREATE TABLE sqlite_db.tmp(i INTEGER);
BEGIN;
INSERT INTO sqlite_db.tmp VALUES (42);
SELECT * FROM sqlite_db.tmp;
```

```
<table>
<thead>
<tr>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>int64</td>
</tr>
<tr>
<td>42</td>
</tr>
</tbody>
</table>
```

```sql
ROLLBACK;
SELECT * FROM sqlite_db.tmp;
```

```
<table>
<thead>
<tr>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>int64</td>
</tr>
<tr>
<td>0 rows</td>
</tr>
</tbody>
</table>
```

**Note.** The old `sqlite_attach` function is deprecated. It is recommended to switch over to the new `ATTACH` syntax.

GitHub Repository

GitHub

Substrait Extension

The main goal of the Substrait extension is to support both production and consumption of Substrait query plans in DuckDB.

This extension is mainly exposed via 3 different APIs - the SQL API, the Python API, and the R API. Here we depict how to consume and produce Substrait query plans in each API.

**Note.** The Substrait integration is currently experimental. Support is currently only available on request. If you have not asked for permission to ask for support, contact us prior to opening an issue. If you open an issue without doing so, we will close it without further review.

Installing and Loading

The Substrait extension is an autoloadable extensions, meaning that it will be loaded at runtime whenever one of the substrait functions is called. To explicitly install and load the released version of the Substrait extension, you can also use the following SQL commands.
In the SQL API, users can generate Substrait plans (into a BLOB or a JSON) and consume Substrait plans.

**BLOB Generation** To generate a Substrait BLOB the `get_substrait(sql)` function must be called with a valid SQL select query.

```
CREATE TABLE crossfit (exercise TEXT, difficulty_level INT);
INSERT INTO crossfit VALUES ('Push Ups', 3), ('Pull Ups', 5), ('Push Jerk', 7), ('Bar Muscle Up', 10);

.mode line
CALL get_substrait('SELECT count(exercise) AS exercise FROM crossfit WHERE difficulty_level <= 5');
```

Plan BLOB = 
```
null
```

```
level
```

```
x12|x09|x1A|x07|x10|x01|x1A|x03|lte|x12|x11|x1A|x0F|x10|x02|x1A|x0Bis_not_
null|x12|x09|x1A|x07|x10|x03|x1A|x03and|x12|x0B|x1A|x09|x10|x04|x1A|x05count|x1A|x01|x12|x15 level|x12|x11|x0A|x07|x2B|x01|x04|x08|x0D|x18|x01|x0A|x04*|x02|x10|x01|x18|x02|x1A|x1AH|x08|x03|x1A|x1E|x08|x01|x1A|x04+|x02|x10|x01|x22|x0C|x1A|x0A|x12|x08|x0A|x04|x12|x02|x0B|x01|x22|x01|x22
```

**JSON Generation** To generate a JSON representing the Substrait plan the `get_substrait_json(sql)` function must be called with a valid SQL select query.

```
CALL get_substrait_json('SELECT count(exercise) AS exercise FROM crossfit WHERE difficulty_level <= 5');
```

```
Json =
```
```
null
```
```
level
```
```
null
```
```
extensionFunction:{"functionAnchor":1,"name":"lte"},extensionFunction:{"functionAnchor":null},extensionFunction:{"functionAnchor":3,"name":"and"},extensionFunction:{"functionAnchor":difficulty_level},struct:{"types":{"varchar":{"length":13,"nullability":"NULLABILITY_REQUIRED"},"i32":{"nullability":"NULLABILITY_REQUIRED"},"filter":{"scalarFunction":{"functionReference":3,"outputType":{"bool":{"nullability":NULLABLE}}},"arguments":[]},"arguments":[]},"selection":{"directReference":{"structField":{"field":1}},"rootReference":null},"selection":{"directReference":{"structField":null},"rootReference":null}}},
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
```
```
null
Python

Substrait extension is autoloadable, but if you prefer to do so explicitly, you can use the relevant Python syntax within a connection:

```python
import duckdb

con = duckdb.connect()
con.install_extension("substrait")
con.load_extension("substrait")
```

**BLOB Generation** To generate a Substrait BLOB the `get_substrait(sql)` function must be called, from a connection, with a valid SQL select query.

```python
proto_bytes = con.get_substrait("SELECT count(exercise) AS exercise FROM crossfit WHERE difficulty_level <= 5")
```

**Json Generation** To generate a JSON representing the Substrait plan the `get_substrait_json(sql)` function, from a connection, must be called with a valid SQL select query.

```python
json = con.get_substrait_json("SELECT count(exercise) AS exercise FROM crossfit WHERE difficulty_level <= 5")
```

**BLOB Consumption** To consume a Substrait BLOB the `from_substrait(blob)` function must be called, from the connection, with a valid Substrait BLOB plan.

```python
query_result = con.from_substrait(proto=proto_bytes)
```

R

By default the extension will be autoloaded on first use. To explicitly install and load this extension in R, use the following commands:
library("duckdb")
con <- dbConnect(duckdb::duckdb())
dbExecute(con, "INSTALL substrait")
dbExecute(con, "LOAD substrait")

**BLOB Generation**  To generate a Substrait BLOB the duckdb_get_substrait(con, sql) function must be called, with a connection and a valid SQL select query.

dbExecute(con, "CREATE TABLE crossfit (exercise TEXT, difficulty_level INT)"

dbExecute(con, "INSERT INTO crossfit VALUES ('Push Ups', 3), ('Pull Ups', 5), ('Push Jerk', 7), ('Bar Muscle Up', 10)"

proto_bytes <- duckdb::duckdb_get_substrait(con, "SELECT * FROM crossfit LIMIT 5")

**JSON Generation**  To generate a JSON representing the Substrait plan duckdb_get_substrait_json(con, sql) function, with a connection and a valid SQL select query.

json <- duckdb::duckdb_get_substrait_json(con, "SELECT count(exercise) AS exercise FROM crossfit WHERE difficulty_level <= 5")

**BLOB Consumption**  To consume a Substrait BLOB the duckdb_prepare_substrait(con, blob) function must be called, with a connection and a valid Substrait BLOB plan.

result <- duckdb::duckdb_prepare_substrait(con, proto_bytes)
df <- dbFetch(result)

**GitHub Repository**

[GitHub](https://github.com)

**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);
```

<table>
<thead>
<tr>
<th>s_store_name</th>
<th>sum(ss_net_profit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>able</td>
<td>-10354620.18</td>
</tr>
<tr>
<td>ation</td>
<td>-10576395.52</td>
</tr>
<tr>
<td>bar</td>
<td>-10625236.01</td>
</tr>
<tr>
<td>ese</td>
<td>-10076698.16</td>
</tr>
<tr>
<td>ought</td>
<td>-10994052.78</td>
</tr>
</tbody>
</table>

Limitations

The `tpchds({query_id})` 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.

GitHub

The `tpcds` extension is part of the main DuckDB repository.

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);
```

<table>
<thead>
<tr>
<th>o_orderpriority</th>
<th>order_count</th>
</tr>
</thead>
<tbody>
<tr>
<td>varchar</td>
<td>int64</td>
</tr>
<tr>
<td>1-URGENT</td>
<td>21188</td>
</tr>
<tr>
<td>2-HIGH</td>
<td>20952</td>
</tr>
<tr>
<td>3-MEDIUM</td>
<td>20820</td>
</tr>
<tr>
<td>4-NOT SPECIFIED</td>
<td>21112</td>
</tr>
<tr>
<td>5-LOW</td>
<td>20974</td>
</tr>
</tbody>
</table>

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.
Data Generator Parameters

The data generator function `dbgen` has the following parameters:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>catalog</td>
<td>VARCHAR</td>
<td>Target catalog</td>
</tr>
<tr>
<td>children</td>
<td>UINTeger</td>
<td>Number of partitions</td>
</tr>
<tr>
<td>overwrite</td>
<td>BOOLEAN</td>
<td>(Not used)</td>
</tr>
<tr>
<td>sf</td>
<td>DOUBLE</td>
<td>Scale factor</td>
</tr>
<tr>
<td>step</td>
<td>UINTeger</td>
<td>Defines the partition to be generated, indexed from 0 to <code>children</code> - 1. Must be defined when the <code>children</code> arguments is defined</td>
</tr>
<tr>
<td>suffix</td>
<td>VARCHAR</td>
<td>Append the suffix to table names</td>
</tr>
</tbody>
</table>

Generating Larger Than Memory Data Sets

To generate data sets for large scale factors, which yield larger than memory data sets, run the `dbgen` function in steps. For example, you may generate SF300 in 10 steps:

```sql
CALL dbgen(sf = 300, children = 10, step = 0);
CALL dbgen(sf = 300, children = 10, step = 1);
...
CALL dbgen(sf = 300, children = 10, step = 9);
```

Limitations

- The data generator function `dbgen` is single-threaded and does not support concurrency. Running multiple steps to parallelize over different partitions is also not supported at the moment.
- The `tpch({query_id})` 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.

GitHub

The `tpch` extension is part of the main DuckDB repository.
Guides
Data Import & Export

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');
```

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_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.

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');

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_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);

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".

HTTP Parquet Import

To load a Parquet file over HTTP(S), the httpfs extension is required. This can be installed use 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):

```sql
SELECT * FROM read_parquet('https://<domain>/path/to/file.parquet');
```

For example:

```sql
SELECT * FROM read_parquet('https://duckdb.org/data/prices.parquet');
```

The function `read_parquet` can be omitted if the URL ends with `.parquet`:

```sql
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`:

```sql
SELECT * FROM 'https://duckdb.org/data/holdings.parquet';
```

## S3 Parquet Import

To load a Parquet file from S3, the **httpfs extension** is required. This can be installed use the `INSTALL SQL` command. This only needs to be run once.

```sql
INSTALL httpfs;
```

To load the httpfs extension for usage, use the `LOAD SQL` command:

```sql
LOAD httpfs;
```

After loading the httpfs extension, set up the credentials and S3 region to read data. Firstly, the region where the data resides needs to be configured:

```sql
SET s3_region = 'us-east-1';
```

With the only the region set, public S3 data can be queried. To query private S3 data, you need to either use an access key and secret:

```sql
SET s3_access_key_id = '<AWS access key id>';
SET s3_secret_access_key = '<AWS secret access key>';
```

or a session token:

```sql
SET s3_session_token = '<AWS session token>';
```

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:

```sql
SELECT * FROM read_parquet('s3://<bucket>/<file>');
```
Google Cloud Storage (GCS)

For Google Cloud Storage (GCS), the Interoperability API enables you to have access to it like an S3 connection. You need to create HMAC keys and declare them:

```
SET s3_endpoint = 'storage.googleapis.com';
SET s3_access_key_id = 'key_id';
SET s3_secret_access_key = 'access_key';
```

Please note you will need to use the s3:// URL to read your data.

```
SELECT * FROM read_parquet('s3://<gcs_bucket>/<file>');
```

Cloudflare R2

For Cloudflare R2, the S3 Compatibility API allows you to use DuckDB’s S3 support to read and write from R2 buckets. You will need to generate an S3 auth token and update the s3_endpoint used:

```
SET s3_region = "auto"
SET s3_endpoint = '<your-account-id>.r2.cloudflarestorage.com';
SET s3_access_key_id = 'key_id';
SET s3_secret_access_key = 'access_key';
```

Note that you will need to use the s3:// URL to read your data from R2:

```
SELECT * FROM read_parquet('s3://<r2_bucket_name>/<file>');
```

S3 Parquet Export

To write a Parquet file to S3, the httpfs extension is required. This can be installed use 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 and S3 region to write data. You may either use an access key and secret, or a token.

```
SET s3_region = 'us-east-1';
SET s3_access_key_id = '<AWS access key id>';
SET s3_secret_access_key = '<AWS secret access key>';
```

The alternative is to use a token:

```
SET s3_region = 'us-east-1';
SET s3_session_token = '<AWS session token>';"
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://bucket/file.parquet';
```

Similarly, Google Cloud Storage (GCS) is supported through the Interoperability API. You need to create HMAC keys and declare them:

```sql
SET s3_endpoint = 'storage.googleapis.com';
SET s3_access_key_id = 'key_id';
SET s3_secret_access_key = 'access_key';
```

Please note you will need to use the s3:// URL to write your files.

```
COPY <table_name> TO 's3://gcs_bucket/file.parquet';
```

**S3 Iceberg Import**

**Prerequisites**

To load an Iceberg file from S3, both the httpfs and iceberg extensions are required. They can be installed use 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.

```
SET s3_region = 'us-east-1';
SET s3_access_key_id = '<AWS access key id>';
SET s3_secret_access_key = '<AWS secret access key>';
```

The alternative is to use a token:

```
SET s3_region = 'us-east-1';
SET s3_session_token = '<AWS session token>';
```

Alternatively, use the aws extension to retrieve the credentials from the aws config file. To load the credentials:

```
CALL load_aws_credentials();
```
**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:

```sql
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:

```
Error: IO Error: Cannot open file "s3://<bucket>/<iceberg-table-folder>/metadata/version-hint.text": No such file or directory
```

**JSON Import**

To read data from a JSON file, use the `read_json_auto` function in the FROM clause of a query.

```sql
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.

```sql
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.

```sql
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.

```sql
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.

```sql
COPY tbl TO 'output.json';
```

The result of queries can also be directly exported to a JSON file.

```sql
COPY (SELECT * FROM tbl) TO 'output.json';
```

For additional options, see the [COPY statement documentation](#).
Excel Import

Installing the Extension

To read data from an Excel file, install and load the spatial extension. This is only needed once per DuckDB connection.

    INSTALL spatial;
    LOAD spatial;

Importing Excel Sheets

Use the st_read function in the FROM clause of a query:

    SELECT * FROM st_read('test_excel.xlsx');

The layer parameter allows specifying the name of the Excel worksheet.

    SELECT * FROM st_read('test_excel.xlsx', layer = 'Sheet1');

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 st_read('test_excel.xlsx', layer = '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 st_read('test_excel.xlsx', layer = 'Sheet1');

Options  Several configuration options are also available for the underlying GDAL library that is doing the XLSX parsing. You can pass them via the open_options parameter of the st_read function as a list of 'KEY=VALUE' strings.

Importing a Sheet with/without a Header  The option HEADERS has three possible values:

    • FORCE: treat the first row as a header
    • DISABLE treat the first row as a row of data
    • AUTO attempt auto-detection (default)

For example, to treat the first row as a header, run:

    SELECT * FROM st_read('test_excel.xlsx', layer = 'Sheet1', open_options=['HEADERS=FORCE']);
Detecting Types  The option FIELD_TYPE defines how field types should be treated:

- STRING: all fields should be loaded as strings (VARCHAR type)
- AUTO: field types should be auto-detected (default)

For example, to treat the first row as a header and use auto-detection for types, run:

```
SELECT * FROM st_read('test_excel.xlsx', layer = 'Sheet1', open_options=['HEADERS=FORCE', 'FIELD_TYPES=AUTO']);
```

To treat the fields as strings:

```
SELECT * FROM st_read('test_excel.xlsx', layer = 'Sheet1', open_options=['FIELD_TYPES=STRING']);
```

See Also

For additional details, see the spatial extension page, the GDAL XLSX driver page, and the GDAL configuration options page.

Excel Export

To export the data from a table to an Excel file, install and load the spatial extension, then use the COPY statement. The file will contain one worksheet with the same name as the file, but without the .xlsx extension.

```
INSTALL spatial; -- Only needed once per DuckDB connection
LOAD spatial; -- Only needed once per DuckDB connection

COPY tbl TO 'output.xlsx' WITH (FORMAT GDAL, DRIVER 'xlsx');
```

The result of queries can also be directly exported to an Excel file.

```
INSTALL spatial; -- Only needed once per DuckDB connection
LOAD spatial; -- Only needed once per DuckDB connection

COPY (SELECT * FROM tbl) TO 'output.xlsx' WITH (FORMAT GDAL, DRIVER 'xlsx');
```

Note: Dates and timestamps are not supported by the xlsx writer driver. Cast columns of those types to VARCHAR prior to creating the xlsx file.

Note: The output file must not already exist.

For additional details, see the spatial extension page and the GDAL XLSX driver page.

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.

```sql
INSTALL mysql;
```

To load the `mysql` extension for usage, use the LOAD SQL command:

```sql
LOAD mysql;
```

Usage

After the `mysql` extension is installed, you can attach to a MySQL database using the following command:

```sql
ATTACH 'host=localhost user=root port=0 database=mysqlscanner' AS mysql_db (TYPE mysql_scanner, READ_ONLY);
USE mysqlscanner;
```

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.

<table>
<thead>
<tr>
<th>Setting</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>host</td>
<td>localhost</td>
</tr>
<tr>
<td>user</td>
<td>current user</td>
</tr>
<tr>
<td>password</td>
<td></td>
</tr>
<tr>
<td>database</td>
<td>NULL</td>
</tr>
<tr>
<td>port</td>
<td>0</td>
</tr>
<tr>
<td>socket</td>
<td>NULL</td>
</tr>
</tbody>
</table>

You can directly read and write the MySQL database:

```sql
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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>host</td>
<td>Name of host to connect to</td>
<td>localhost</td>
</tr>
<tr>
<td>hostaddr</td>
<td>Host IP address</td>
<td>localhost</td>
</tr>
<tr>
<td>port</td>
<td>Port Number</td>
<td>5432</td>
</tr>
<tr>
<td>user</td>
<td>Postgres User Name</td>
<td>[OS user name]</td>
</tr>
<tr>
<td>password</td>
<td>Postgres Password</td>
<td></td>
</tr>
<tr>
<td>dbname</td>
<td>Database Name</td>
<td>[user]</td>
</tr>
<tr>
<td>passfile</td>
<td>Name of file passwords are stored in</td>
<td>~/.pgpass</td>
</tr>
</tbody>
</table>

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 Postgres 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](https://www.postgresql.org/docs/current/).
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 \texttt{INSTALL} SQL command. This only needs to be run once.

\texttt{INSTALL} sqlite;

To load the sqlite extension for usage, use the \texttt{LOAD} SQL command:

\texttt{LOAD} sqlite;

Usage

After the SQLite extension is installed, tables can be queried from SQLite using the \texttt{sqlite\_scan} function:

\begin{verbatim}
-- scan the table "tbl\_name" from the SQLite file "test.db"
SELECT * FROM sqlite\_scan('test.db', 'tbl\_name');
\end{verbatim}

Alternatively, the entire file can be attached using the \texttt{ATTACH} command. This allows you to query all tables stored within a SQLite database file as if they were a regular database.

\begin{verbatim}
-- 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;
\end{verbatim}

For more information see the SQLite extension documentation.
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.

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 as strings are generally slower to process.

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 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**:

```sql
SELECT avg(extract('day' FROM creationDate)) FROM Comment;
```
Second, we use the VARCHAR type and use string operations:

```sql
SELECT avg(CAST(creationDate[9:10] AS INT)) FROM Comment;
```

The results of the microbenchmark are as follows:

<table>
<thead>
<tr>
<th>Column Type</th>
<th>Storage Size</th>
<th>Query Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATETIME</td>
<td>3.3 GB</td>
<td>0.904 s</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>5.2 GB</td>
<td>3.919 s</td>
</tr>
</tbody>
</table>

The results show that using the DATETIME value yields smaller storage sizes and faster processing.

**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. We first 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 both cases, we take the data from `.csv.gz` files, and measure the time required to perform the loading.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load without primary key</td>
<td>92.168s</td>
</tr>
<tr>
<td>Load with primary key</td>
<td>286.765s</td>
</tr>
</tbody>
</table>

**Indexing**

DuckDB has two types of indexes: zonemaps and ART indexes.
DuckDB automatically creates zonemaps (also known as min-max indexes) for the columns of all general-purpose data types. These indexes are used for predicate pushdown into scan operators and computing aggregations. This means that 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., a block with a min-max range of 1000 to 2000 will be omitted when comparing for = 123 or < 400).

The Effect of Ordering on Zonemaps The more ordered the data within a column, the more useful the zonemap indexes will be. For example, in the worst case, a column could contain a random number on every row. DuckDB will be unlikely to be able to skip any row groups. The best case of ordered data commonly arises with DATETIME columns. If specific columns will be queried with selective filters, it is best to pre-order data by those columns when inserting it. Even an imperfect ordering will still be helpful.

Microbenchmark: The Effect of Ordering For an example, let’s repeat the microbenchmark for timestamps with a timestamp column that sorted using an ascending order vs. an unordered one.

<table>
<thead>
<tr>
<th>Column Type</th>
<th>Ordered</th>
<th>Storage Size</th>
<th>Query Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATETIME</td>
<td>yes</td>
<td>1.3 GB</td>
<td>0.578 s</td>
</tr>
<tr>
<td>DATETIME</td>
<td>no</td>
<td>3.3 GB</td>
<td>0.904 s</td>
</tr>
</tbody>
</table>

The results show that simply keeping the column order allows for improved compression, yielding a 2.5x smaller storage size. It also allows the computation to be 1.5x faster.

Ordered Integers Another practical way to exploit ordering is to use the INTEGER type with automatic increments rather than UUID for columns that will be queried using selective filters. UUIDs will likely be inserted in a random order, so many row groups in the table will need to be scanned to find a specific UUID value, while an ordered INTEGER column will allow all row groups to be skipped except the one that contains 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 a 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. It enables efficient constraint checking upon changes (inserts, updates, and deletes) for non-bulky changes.
2. Having an ART index makes changes to the affected column(s) slower compared to non-indexed performance. That is because of index maintenance for these operations.
Regarding query performance, an ART index has the following effects:

1. It speeds up point queries and other highly selective queries using the indexed column(s), where the filtering condition returns approx. 0.1% of all rows or fewer. When in doubt, use `EXPLAIN` to verify that your query plan uses the index scan.
2. An ART index has no effect on the performance of join, aggregation, and sorting queries.

Indexes are serialized to disk and deserialized lazily, i.e., when the database is reopened, 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 Practices:**

- 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.
- If you define an ART index, do so after bulk loading the data to the table.

**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 and Memory** As a rule of thumb, aggregation-heavy workloads require approx. 5 GB memory per CPU core and join-heavy workloads require approximately 10 GB memory per core for best performance. In AWS EC2, the former are available as general-purpose instances (e.g., M7g) and the latter as memory-optimized instances (e.g., R7g).

**Best Practice:** Aim for 5-10 GB memory per CPU core.

**Disk** DuckDB is capable of operating both as an in-memory and as a disk-based database system. In the latter case, it can spill to disk to process larger-than-memory workloads (a.k.a. out-of-core processing). In these cases, a fast disk is highly beneficial. However, if the workload fits in memory, the disk speed only has a limited effect on performance.

In general, network disks – e.g., NFS, network drives on Windows, and network-backed cloud disks (such as AWS EBS) – will result in slower DuckDB workloads than using local disks. However, different network disks can have vastly varying IO performance, ranging from very slow to almost as fast as local. Therefore, for optimal performance, only use network disks that can provide high IO performance.

**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 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. Among Linux distributions, 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.

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, 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 1.1-5.0x 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.

<table>
<thead>
<tr>
<th>Row Group Size</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>960</td>
<td>8.772s</td>
</tr>
<tr>
<td>1920</td>
<td>8.951s</td>
</tr>
<tr>
<td>3840</td>
<td>4.331s</td>
</tr>
<tr>
<td>7680</td>
<td>2.35s</td>
</tr>
<tr>
<td>15360</td>
<td>1.584s</td>
</tr>
<tr>
<td>30720</td>
<td>1.17s</td>
</tr>
<tr>
<td>61440</td>
<td>0.948s</td>
</tr>
<tr>
<td>122880</td>
<td>0.875s</td>
</tr>
<tr>
<td>245760</td>
<td>0.931s</td>
</tr>
<tr>
<td>491520</td>
<td>0.955s</td>
</tr>
<tr>
<td>983040</td>
<td>0.973s</td>
</tr>
<tr>
<td>1966080</td>
<td>0.886s</td>
</tr>
</tbody>
</table>

The results show that row group sizes <5,000 have a strongly detrimental effect, making runtimes more than 5-10x larger than ideally-sized row groups, while row group sizes between 5,000 and 20,000 are still 1.5-2.5x 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 100MB and 10GB 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.

<table>
<thead>
<tr>
<th>Schema</th>
<th>Load Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load from GZIP-compressed CSV files (.csv.gz)</td>
<td>107.1s</td>
</tr>
<tr>
<td>Decompressing (using parallel gunzip) and loading from decompressed CSV files</td>
<td>121.3s</td>
</tr>
</tbody>
</table>

Tuning Workloads

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.
Prerequisites  Spilling to disk is automatically supported when connected to a persistent database file. When running in in-memory mode, DuckDB cannot use disk to offload data if it does not fit into main memory. To enable offloading in the absence of a persistent database file, use the SET temp_directory statement:

```
SET temp_directory = '/path/to/temp.tmp'
```

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:

- sorting: ORDER BY,
- grouping: GROUP BY,
- windowing: OVER ... (PARTITION BY ... ORDER BY ...),
- joining: JOIN.

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:

- 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.
- Currently, some aggregate functions, such as list() and string_agg(), do not support offloading to disk.
- 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.

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.

The `preserve_insertion_order` Option

When importing or exporting data sets that are much larger than the available memory, out of memory errors may occur. In these cases, it’s worth setting the `preserve_insertion_order` configuration option to false:

```sql
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.
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 **5-10 GB memory per CPU core**.
2. Are you using a fast disk? Network-attached disks can cause the workload to slow down, especially for **larger than memory workloads**.
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 Issue button above 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 **nightly 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 bench-
mark frameworks, and omit other parts of the workloads such as updates.
Meta Queries

Describe

Describing a Table

In order to view the schema of a table, use DESCRIBE followed by the table name.

```
CREATE TABLE tbl (i INTEGER PRIMARY KEY, j VARCHAR);
DESCRIBE tbl;
```

<table>
<thead>
<tr>
<th>column_name</th>
<th>column_type</th>
<th>null</th>
<th>key</th>
<th>default</th>
<th>extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>INTEGER</td>
<td>NO</td>
<td>PRI</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>j</td>
<td>VARCHAR</td>
<td>YES</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>

Describing a Query

In order to view the schema of the result of a query, prepend DESCRIBE to a query.

```
DESCRIBE SELECT * FROM tbl;
```

<table>
<thead>
<tr>
<th>column_name</th>
<th>column_type</th>
<th>null</th>
<th>key</th>
<th>default</th>
<th>extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>INTEGER</td>
<td>YES</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>j</td>
<td>VARCHAR</td>
<td>YES</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>

Note that there are subtle differences: compared to the result when describing a table, nullability (null) and key information (key) are lost.

Inspecting Query Plans Using EXPLAIN

In order to view the query plan of a query, prepend EXPLAIN to a query.

```
EXPLAIN SELECT * FROM tbl;
```
By default only the final physical plan is shown. In order to see the unoptimized and optimized logical plans, change the `explain_output` setting:

```
SET explain_output = 'all';
```

Below is an example of running `EXPLAIN` on Q1 of the TPC-H benchmark.
List Tables

The SHOW TABLES command can be used to obtain a list of all tables within the selected schema.

```sql
CREATE TABLE tbl (i INTEGER);
SHOW TABLES;
```

<table>
<thead>
<tr>
<th>name</th>
<th>tbl</th>
</tr>
</thead>
</table>

SHOW or SHOW ALL TABLES can be used to obtain a list of all tables within all attached databases and schemas.

```sql
CREATE TABLE tbl (i INTEGER);
CREATE SCHEMA s1;
CREATE TABLE s1.tbl (v VARCHAR);
SHOW ALL TABLES;
```

<table>
<thead>
<tr>
<th>database</th>
<th>schema</th>
<th>table_name</th>
<th>column_names</th>
<th>column_types</th>
<th>temporary</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory</td>
<td>main</td>
<td>tbl</td>
<td>[i]</td>
<td>[INTEGER]</td>
<td>false</td>
</tr>
<tr>
<td>memory</td>
<td>s1</td>
<td>tbl</td>
<td>[v]</td>
<td>[VARCHAR]</td>
<td>false</td>
</tr>
</tbody>
</table>
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.

### Profile Queries Using EXPLAIN ANALYZE

In order to profile a query, prepend `EXPLAIN ANALYZE` to a query.

```
EXPLAIN ANALYZE SELECT * FROM tbl;
```

The query plan will be pretty-printed to the screen using timings for every operator.

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 Q1 of the TPC-H benchmark.
HASH_GROUP_BY

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>#0</td>
<td>#1</td>
<td>sum(#2)</td>
<td>sum(#3)</td>
<td>sum(#4)</td>
<td>sum(#5)</td>
<td>avg(#6)</td>
<td>avg(#7)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(0.28s)

PROJECTION

<table>
<thead>
<tr>
<th>l_returnflag</th>
<th>l_linestatus</th>
<th>l_quantity</th>
<th>l_extendedprice</th>
<th>#4</th>
<th>(#4 * (1.00 + l_tax))</th>
<th>l_quantity</th>
<th>l_extendedprice</th>
<th>l_discount</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5916591</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(0.02s)

PROJECTION

<table>
<thead>
<tr>
<th>l_returnflag</th>
<th>l_linestatus</th>
<th>l_quantity</th>
<th>l_extendedprice</th>
<th>(l_extendedprice * (1.00 - l_discount))</th>
<th>l_tax</th>
<th>l_discount</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5916591

(0.02s)
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, including \texttt{min}, \texttt{max}, \texttt{avg}, \texttt{std} and \texttt{approx unique}.

Usage

In order to summarize the contents of a table, use SUMMARIZE followed by the table name.

\texttt{SUMMARIZE tbl;}

In order to summarize a query, prepend SUMMARIZE to a query.

\texttt{SUMMARIZE SELECT * FROM tbl;}

Example

Below is an example of SUMMARIZE on the \texttt{lineitem} table of TPC-H SF1 table, generated using the \texttt{tpch} extension.

\texttt{INSTALL tpch;}
\texttt{LOAD tpch;}
\texttt{CALL dbgen(sf = 1);}
### SUMMARIZE lineitem;

<table>
<thead>
<tr>
<th>column_name</th>
<th>column_type</th>
<th>min</th>
<th>max</th>
<th>approx_unique</th>
<th>avg</th>
<th>std</th>
<th>q25</th>
<th>q50</th>
<th>q75</th>
<th>count</th>
<th>null_percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>l_orderkey</td>
<td>INTEGER</td>
<td>600000</td>
<td>1508227</td>
<td>300279.6042</td>
<td>898621</td>
<td>485223</td>
<td>298986</td>
<td>448523</td>
<td>6001215</td>
<td>0.0%</td>
<td></td>
</tr>
<tr>
<td>l_partkey</td>
<td>INTEGER</td>
<td>200000</td>
<td>202598</td>
<td>10017.989</td>
<td>340023</td>
<td>99992</td>
<td>49913</td>
<td>99992</td>
<td>150039601215</td>
<td>0.0%</td>
<td></td>
</tr>
<tr>
<td>l_suppkey</td>
<td>INTEGER</td>
<td>10000</td>
<td>10061</td>
<td>5000.6026</td>
<td>961998</td>
<td>250611499</td>
<td>7500</td>
<td>6001215</td>
<td>0.0%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>l_linenumber</td>
<td>INTEGER</td>
<td>7</td>
<td>7</td>
<td>3.0005757</td>
<td>1.73243</td>
<td>3</td>
<td>4</td>
<td>38</td>
<td>6001215</td>
<td>0.0%</td>
<td></td>
</tr>
<tr>
<td>l_quantity</td>
<td>DECIMAL</td>
<td>50.00</td>
<td>50</td>
<td>25.507967</td>
<td>1368626</td>
<td>25370169126</td>
<td>38</td>
<td>6001215</td>
<td>0.0%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>l_extendedprice</td>
<td>DECIMAL</td>
<td>104949.50</td>
<td>923139</td>
<td>38255.1384</td>
<td>2666684187</td>
<td>10883136724</td>
<td>55159</td>
<td>6001215</td>
<td>0.0%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>l_discount</td>
<td>DECIMAL</td>
<td>0.10</td>
<td>11</td>
<td>0.04999943</td>
<td>0.0056</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6001215</td>
<td>0.0%</td>
<td></td>
</tr>
<tr>
<td>l_tax</td>
<td>DECIMAL</td>
<td>0.08</td>
<td>9</td>
<td>0.04001350</td>
<td>0.0056</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6001215</td>
<td>0.0%</td>
<td></td>
</tr>
<tr>
<td>l_returnflag</td>
<td>VARCHAR</td>
<td>R</td>
<td>3</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>6001215</td>
<td>0.0%</td>
<td></td>
</tr>
<tr>
<td>l_linenumber</td>
<td>VARCHAR</td>
<td>0</td>
<td>2</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>6001215</td>
<td>0.0%</td>
<td></td>
</tr>
<tr>
<td>l_linenumber</td>
<td>DATE</td>
<td>1992-01-02</td>
<td>1998-12-01</td>
<td>2516</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>6001215</td>
<td>0.0%</td>
<td></td>
</tr>
<tr>
<td>l_linenumber</td>
<td>DATE</td>
<td>1992-01-31</td>
<td>1998-10-31</td>
<td>2460</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>6001215</td>
<td>0.0%</td>
<td></td>
</tr>
<tr>
<td>l_linenumber</td>
<td>DATE</td>
<td>1992-01-04</td>
<td>1998-12-31</td>
<td>2549</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>6001215</td>
<td>0.0%</td>
<td></td>
</tr>
<tr>
<td>l_shipinstruct</td>
<td>VARCHAR</td>
<td>COD</td>
<td>4</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>6001215</td>
<td>0.0%</td>
<td></td>
</tr>
<tr>
<td>l_shipmode</td>
<td>VARCHAR</td>
<td>TRUCK</td>
<td>7</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>6001215</td>
<td>0.0%</td>
<td></td>
</tr>
<tr>
<td>l_comment</td>
<td>VARCHAR</td>
<td>Tiresias zzle? furiously iro</td>
<td>3558599</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>6001215</td>
<td>0.0%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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.

```sql
SELECT version();
```

<table>
<thead>
<tr>
<th><code>version()</code></th>
<th>varchar</th>
</tr>
</thead>
<tbody>
<tr>
<td>v0.9.2</td>
<td></td>
</tr>
</tbody>
</table>

Using a PRAGMA:

```sql
PRAGMA version;
```

<table>
<thead>
<tr>
<th>library_version</th>
<th>source_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>v0.9.2</td>
<td>3c695d7ba9</td>
</tr>
</tbody>
</table>

Platform

The platform information consists of the operating system, compiler, and, optionally, the compiler. The platform is used when installing extensions. To retrieve the platform, use the following PRAGMA:

```sql
PRAGMA platform;
```

On macOS, running on Apple Silicon architecture, the result is:

<table>
<thead>
<tr>
<th><code>platform</code></th>
<th>varchar</th>
</tr>
</thead>
<tbody>
<tr>
<td>osx_arm64</td>
<td></td>
</tr>
</tbody>
</table>

On Windows, running on an AMD64 architecture, the platform is `windows_amd64`. On CentOS 7, running on the AMD64 architecture, the platform is `linux_amd64_gcc4`. On Ubuntu 22.04, running on the ARM64 architecture, the platform is `linux_arm64`. 

752
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_types()`: data types
- `duckdb_views()`: views
- `duckdb_temporary_files()`: the temporary files DuckDB has written to disk, to offload data from memory
ODBC

ODBC 101: A Duck Themed Guide to ODBC

• What is ODBC?
• General Concepts
• Setting up an Application
• Sample Application

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.

Note. 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.
## Handle Types

<table>
<thead>
<tr>
<th>Handle</th>
<th>Type</th>
<th>Description</th>
<th>Use Case</th>
<th>Additional Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Environment</td>
<td>SQL_HANDLE_ENV</td>
<td>Manages the environment settings for ODBC operations, and provides a global context in which to access data.</td>
<td>Initializing ODBC, managing driver behavior, resource allocation</td>
<td>Must be <strong>allocated</strong> once per application upon starting, and freed at the end.</td>
</tr>
<tr>
<td>Connection</td>
<td>SQL_HANDLE_DBC</td>
<td>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.</td>
<td>Establishing a connection to a database, managing the connection state</td>
<td>Multiple connection handles can be <strong>created</strong> as needed, allowing simultaneous connections to multiple data sources. <strong>Note:</strong> Allocating a connection handle does not establish a connection, but must be allocated first, and then used once the connection has been established.</td>
</tr>
<tr>
<td>Statement</td>
<td>SQL_HANDLE_STMT</td>
<td>Handles the execution of SQL statements, as well as the returned result sets.</td>
<td>Executing SQL queries, fetching result sets, managing statement options.</td>
<td>To facilitate the execution of concurrent queries, multiple handles can be <strong>allocated</strong> per connection.</td>
</tr>
<tr>
<td>Descriptor</td>
<td>SQL_HANDLE_DESC</td>
<td>Describes the attributes of a data structure or parameter, and allows the application to specify the structure of data to be bound/retrieved.</td>
<td>Describing table structures, result sets, binding columns to application buffers</td>
<td>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.</td>
</tr>
</tbody>
</table>
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, \texttt{SQLDriverConnect} and \texttt{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: \texttt{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 \texttt{SQLGetDiagRec} function to retrieve the error information. The following table defines the return codes:

<table>
<thead>
<tr>
<th>Return Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_SUCCESS</td>
<td>The function completed successfully.</td>
</tr>
<tr>
<td>SQL_SUCCESS_WITH_INFO</td>
<td>The function completed successfully, but additional information is available, including a warning</td>
</tr>
<tr>
<td>SQL_ERROR</td>
<td>The function failed.</td>
</tr>
<tr>
<td>SQL_INVALID_HANDLE</td>
<td>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</td>
</tr>
<tr>
<td>SQL_NO_DATA</td>
<td>The function completed successfully, but no more data is available</td>
</tr>
<tr>
<td>SQL_NEED_DATA</td>
<td>More data is needed, such as when a parameter data is sent at execution time, or additional connection information is required.</td>
</tr>
<tr>
<td>SQL_STILL_EXECUTING</td>
<td>A function that was asynchronously executed is still executing.</td>
</tr>
</tbody>
</table>

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 \texttt{SQLBindCol} and \texttt{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++.

**Note.** To install the driver as well as anything else you will need follow these instructions.

1. **Include the SQL Header Files**
2. **Define the ODBC Handles and Connect to the Database**
3. **Adding a Query**
4. **Fetching Results**
5. **Go Wild**
6. **Free the Handles and Disconnecting**

### 1. Include the SQL Header Files

The first step is to include the SQL header files:

```c
#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:

```bash
brew install unixodbc
# or
sudo apt-get install unixodbc-dev
# or
sudo yum install unixODBC-devel
```

Remember to include the header file location in your `CFLAGS`.

For `MAKEFILE`:

```bash
CFLAGS=-I/usr/local/include
# or
CFLAGS=-I/opt/homebrew/Cellar/unixodbc/2.3.11/include
```

For `CMAKE`:

```bash
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`:

```bash
target_link_libraries(ODBC_application /path/to/duckdb_odbc/libduckdb_odbc.dylib)
```

For `MAKEFILE`:

```bash
LDLIBS=-L/path/to/duckdb_odbc/libduckdb_odbc.dylib
```
2. Define the ODBC Handles and Connect to the Database

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:

```sql
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;
```

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:

```sql
SQLHANDLE stmt;
SQLAllocHandle(SQL_HANDLE_STMT, dbc, &stmt);
```

Then we can execute a query:

```sql
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:

```sql
SQLLEN int_val;
SQLLEN null_val;
SQLBindCol(stmt, 1, SQL_C_SLONG, &int_val, 0, &null_val);
```

Then we can fetch the results:

```sql
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:

```sql
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:

```sql
SQLFreeHandle(SQL_HANDLE_STMT, stmt);
```

Then we need to disconnect from the database:

```sql
SQLDisconnect(dbc);
```

And finally, we need to free the connection handle and the environment handle:

```sql
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

```cpp
#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)");
```
DuckDB Documentation

```cpp
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");

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
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)
Python

Installing the Python Client

Installing via Pip

The latest release of the Python client can be installed using `pip`.

```bash
pip install duckdb
```

The pre-release Python client can be installed using `--pre`.

```bash
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
```

Executing SQL in Python

SQL queries can be executed using the `duckdb.sql` function.

```python
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.

```python
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.

```python
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 JupyterSQL 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 Collab link above for an in-notebook example
pip install duckdb
DuckDB Documentation

# Install Jupyter Notebook (Note: you can also install JupyterLab: pip install jupyterlab)
pip install notebook

# Install supporting libraries
pip install jupysql
pip install pandas
pip install matplotlib
pip install duckdb-engine

Library Import and Configuration

Open a Jupyter Notebook and import the relevant libraries.

**Connecting to DuckDB Natively**

To connect to DuckDB, run:

```python
import duckdb
import pandas as pd

%load_ext sql
conn = duckdb.connect()
%sql conn --alias duckdb
```

**Connecting to DuckDB via SQLAlchemy Using duckdb_engine**

Alternatively, you can connect to DuckDB via SQLAlchemy using duckdb_engine. See the performance and feature differences.

```python
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

Set configurations on jupysql to directly output data to Pandas and to simplify the output that is printed to the notebook.

```
cfg SqlMagic.autopandas = True
%config SqlMagic.feedback = False
%config SqlMagic.displaycon = False
```

Connect jupysql to DuckDB using a SQLAlchemy-style connection string. Either connect to a new in-memory DuckDB, the default connection or a file backed db.

```sql
duckdb:///default:
# %sql duckdb:///memory:
# %sql duckdb:///path/to/file.db
```
Note. 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 DF.

```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 DF.

```sql
%%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 magic.

```sql
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.

```python
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
sql output_df << SELECT sum(i) AS total_i FROM input_df
```

### 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 Jupyter 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.
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;
```

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 URL of the remotely stored Parquet file.

```sqlplot boxplot --table https://d37ci6vzurychx.cloudfront.net/trip-data/yellow_tripdata_2021-01.parquet --column trip_distance
```

```
trip_distance' from 'https://d37ci6vzurychx.cloudfront.net/trip-data/yellow_tripdata_2021-01.parquet'
```

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!

SQL on Pandas

Pandas DataFrames stored in local variables can be queried as if they are regular tables within DuckDB.

```python
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 referring to the Pandas DataFrame in the query.

```python
# 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")
```

**Export to Pandas**

The result of a query can be converted to a Pandas DataFrame using the df() function.

```python
# read the result of an arbitrary SQL query to a Pandas DataFrame
results = duckdb.sql("SELECT 42").df()
```

**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.

```python
import duckdb
import pyarrow as pa

# connect to an in-memory database
con = duckdb.connect()
```
### 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.

```python
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.

```python
import duckdb
import pyarrow as pa
import tempfile

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()
```
DuckDB Documentation

```python
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.

```
# 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 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.

```python
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

```python
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

```python
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.

```python
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.

```python
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 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.

```python
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.

```python
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
            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 15+ 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:

```bash
pip install 'ibis-framework[duckdb]' 
```

or use conda:

```bash
conda install ibis-framework 
```

or use mamba:

```bash
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.

```python
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:

```python
# 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.

```python
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.

```python
penguins.head().to_pandas()
```

```
   species island bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
0      Adelie  Torgersen         39.1          18.7             181.0      3750.0
1      Adelie  Torgersen         39.5          17.4             186.0      3800.0
2      Adelie  Torgersen         40.3          18.0             195.0      3250.0
3      Adelie  Torgersen            NaN            NaN             NaN       NaN
4      Adelie  Torgersen         36.7          19.3             193.0      3450.0
```

`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!).

```python
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]
```

778
DuckDB Documentation

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.

\[
\text{ibis.options.interactive} = \text{True}
\]

\[
penguins.head()
\]

```plaintext
<table>
<thead>
<tr>
<th>species</th>
<th>island</th>
<th>bill_length_mm</th>
<th>bill_depth_mm</th>
<th>flipper_length_mm</th>
<th>body_mass_g</th>
<th>sex</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adelie</td>
<td>Torgersen</td>
<td>39.1</td>
<td>18.7</td>
<td>181</td>
<td>3750</td>
<td>male</td>
<td>2007</td>
</tr>
<tr>
<td>Adelie</td>
<td>Torgersen</td>
<td>39.5</td>
<td>17.4</td>
<td>186</td>
<td>3800</td>
<td>female</td>
<td>2007</td>
</tr>
<tr>
<td>Adelie</td>
<td>Torgersen</td>
<td>40.3</td>
<td>18.0</td>
<td>195</td>
<td>3250</td>
<td>female</td>
<td>2007</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>nan</td>
<td>nan</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>2007</td>
</tr>
<tr>
<td>Adelie</td>
<td>Torgersen</td>
<td>36.7</td>
<td>19.3</td>
<td>193</td>
<td>3450</td>
<td>female</td>
<td>2007</td>
</tr>
</tbody>
</table>
```

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:

\[
penguins.filter(penguins.species == "Gentoo")
\]

```plaintext
<table>
<thead>
<tr>
<th>species</th>
<th>island</th>
<th>bill_length_mm</th>
<th>bill_depth_mm</th>
<th>flipper_length_mm</th>
<th>body_mass_g</th>
<th>sex</th>
<th>year</th>
</tr>
</thead>
</table>
```
### Or filter for Adelie penguins that reside on the island of Torgersen:

```sql
penguins.filter((penguins.species == "Gentoo") & (penguins.body_mass_g > 6000))
```

### Example selection

```sql
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:

```python
penguins.select("species", "island", "year").limit(3)
```

<table>
<thead>
<tr>
<th>species</th>
<th>island</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>string</td>
<td>int64</td>
</tr>
<tr>
<td>Adelie</td>
<td>Torgersen</td>
<td>2007</td>
</tr>
<tr>
<td>Adelie</td>
<td>Torgersen</td>
<td>2007</td>
</tr>
<tr>
<td>Adelie</td>
<td>Torgersen</td>
<td>2007</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>

Or you can use column objects directly (this can be convenient when paired with tab-completion):

```python
penguins.select(penguins.species, penguins.island, penguins.year).limit(3)
```

<table>
<thead>
<tr>
<th>species</th>
<th>island</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>string</td>
<td>int64</td>
</tr>
<tr>
<td>Adelie</td>
<td>Torgersen</td>
<td>2007</td>
</tr>
<tr>
<td>Adelie</td>
<td>Torgersen</td>
<td>2007</td>
</tr>
<tr>
<td>Adelie</td>
<td>Torgersen</td>
<td>2007</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>

Or you can mix-and-match:

```python
penguins.select("species", "island", penguins.year).limit(3)
```

<table>
<thead>
<tr>
<th>species</th>
<th>island</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>string</td>
<td>int64</td>
</tr>
<tr>
<td>Adelie</td>
<td>Torgersen</td>
<td>2007</td>
</tr>
<tr>
<td>Adelie</td>
<td>Torgersen</td>
<td>2007</td>
</tr>
<tr>
<td>Adelie</td>
<td>Torgersen</td>
<td>2007</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>

**mutate**  
`mutate` lets you add new columns to your table, derived from the values of existing columns.

```python
penguins.mutate(bill_length_cm=penguins.bill_length_mm / 10)
```

<table>
<thead>
<tr>
<th>species</th>
<th>island</th>
<th>bill_length_mm</th>
<th>bill_depth_mm</th>
<th>flipper_length_mm</th>
<th>body_mass_g</th>
<th>sex</th>
<th>year</th>
<th>bill_length_cm</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>string</td>
<td>float64</td>
<td>float64</td>
<td>int64</td>
<td>int64</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>---------</td>
<td>---------</td>
<td>-------</td>
<td>-------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adelie</td>
<td>Torgersen</td>
<td>39.1</td>
<td>18.7</td>
<td>181</td>
<td>3750</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>male</td>
<td>2007</td>
<td>3.91</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adelie</td>
<td>Torgersen</td>
<td>39.5</td>
<td>17.4</td>
<td>186</td>
<td>3800</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>female</td>
<td>2007</td>
<td>3.95</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adelie</td>
<td>Torgersen</td>
<td>40.3</td>
<td>18.0</td>
<td>195</td>
<td>3250</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>female</td>
<td>2007</td>
<td>4.03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adelie</td>
<td>Torgersen</td>
<td>nan</td>
<td>nan</td>
<td>NULL</td>
<td>NULL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NULL</td>
<td>2007</td>
<td>nan</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adelie</td>
<td>Torgersen</td>
<td>36.7</td>
<td>19.3</td>
<td>193</td>
<td>3450</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>female</td>
<td>2007</td>
<td>3.67</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adelie</td>
<td>Torgersen</td>
<td>39.3</td>
<td>20.6</td>
<td>190</td>
<td>3650</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>male</td>
<td>2007</td>
<td>3.93</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adelie</td>
<td>Torgersen</td>
<td>38.9</td>
<td>17.8</td>
<td>181</td>
<td>3625</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>female</td>
<td>2007</td>
<td>3.89</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adelie</td>
<td>Torgersen</td>
<td>39.2</td>
<td>19.6</td>
<td>195</td>
<td>4675</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>male</td>
<td>2007</td>
<td>3.92</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adelie</td>
<td>Torgersen</td>
<td>34.1</td>
<td>18.1</td>
<td>193</td>
<td>3475</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NULL</td>
<td>2007</td>
<td>3.41</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adelie</td>
<td>Torgersen</td>
<td>42.0</td>
<td>20.2</td>
<td>190</td>
<td>4250</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NULL</td>
<td>2007</td>
<td>4.20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

```sql
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",
)
```

<table>
<thead>
<tr>
<th>species</th>
<th>island</th>
<th>bill_depth_mm</th>
<th>flipper_length_mm</th>
<th>body_mass_g</th>
<th>sex</th>
<th>year</th>
<th>bill_length_cm</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>string</td>
<td>float64</td>
<td>int64</td>
<td>int64</td>
<td>string</td>
<td>float64</td>
<td></td>
</tr>
<tr>
<td>int64</td>
<td>float64</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

782
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.

```python
import ibis.selectors as s

penguins.mutate(bill_length_cm=penguins.bill_length_mm / 10).select(~s.matches("bill_length_mm"))
```

### 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.
You can also use a selector alongside a column name.

```python
penguins.select("island", s.numeric())
```

<table>
<thead>
<tr>
<th>island</th>
<th>bill_length_mm</th>
<th>bill_depth_mm</th>
<th>flipper_length_mm</th>
<th>body_mass_g</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>Torgersen</td>
<td>39.1</td>
<td>18.7</td>
<td>181</td>
<td>3750</td>
<td>2007</td>
</tr>
<tr>
<td>Torgersen</td>
<td>39.5</td>
<td>17.4</td>
<td>186</td>
<td>3800</td>
<td>2007</td>
</tr>
<tr>
<td>Torgersen</td>
<td>40.3</td>
<td>18.0</td>
<td>195</td>
<td>3250</td>
<td>2007</td>
</tr>
<tr>
<td>Torgersen</td>
<td>nan</td>
<td>nan</td>
<td>NULL</td>
<td>NULL</td>
<td>2007</td>
</tr>
<tr>
<td>Torgersen</td>
<td>36.7</td>
<td>19.3</td>
<td>193</td>
<td>3450</td>
<td>2007</td>
</tr>
<tr>
<td>Torgersen</td>
<td>39.3</td>
<td>20.6</td>
<td>190</td>
<td>3650</td>
<td>2007</td>
</tr>
<tr>
<td>Torgersen</td>
<td>38.9</td>
<td>17.8</td>
<td>181</td>
<td>3625</td>
<td>2007</td>
</tr>
<tr>
<td>Torgersen</td>
<td>39.2</td>
<td>19.6</td>
<td>195</td>
<td>NULL</td>
<td>2007</td>
</tr>
<tr>
<td>Torgersen</td>
<td>34.1</td>
<td>18.1</td>
<td>193</td>
<td>4675</td>
<td>2007</td>
</tr>
<tr>
<td>Torgersen</td>
<td>42.0</td>
<td>20.2</td>
<td>190</td>
<td>4250</td>
<td>2007</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

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:

```python
penguins.order_by(penguins.flipper_length_mm).select( 
    "species", "island", "flipper_length_mm"
)
```

<table>
<thead>
<tr>
<th>species</th>
<th>island</th>
<th>flipper_length_mm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adelie</td>
<td>Torgersen</td>
<td>19.3</td>
</tr>
<tr>
<td>Adelie</td>
<td>Torgersen</td>
<td>20.6</td>
</tr>
<tr>
<td>Adelie</td>
<td>Torgersen</td>
<td>17.8</td>
</tr>
<tr>
<td>Adelie</td>
<td>Torgersen</td>
<td>19.6</td>
</tr>
<tr>
<td>Adelie</td>
<td>Torgersen</td>
<td>18.1</td>
</tr>
<tr>
<td>Adelie</td>
<td>Torgersen</td>
<td>19.6</td>
</tr>
<tr>
<td>Adelie</td>
<td>Torgersen</td>
<td>18.1</td>
</tr>
<tr>
<td>Adelie</td>
<td>Torgersen</td>
<td>19.6</td>
</tr>
<tr>
<td>Adelie</td>
<td>Torgersen</td>
<td>18.1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
You can sort in descending order using the `desc` method of a column:

```python
penguins.order_by(penguins.flipper_length_mm.desc()).select(
    "species", "island", "flipper_length_mm"
)
```

Or you can use `ibis.desc`

```python
penguins.order_by(ibis.desc("flipper_length_mm")) .select(
    "species", "island", "flipper_length_mm"
)
```

<table>
<thead>
<tr>
<th>species</th>
<th>island</th>
<th>flipper_length_mm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gentoo</td>
<td>Biscoe</td>
<td>231</td>
</tr>
<tr>
<td>Gentoo</td>
<td>Biscoe</td>
<td>230</td>
</tr>
<tr>
<td>Gentoo</td>
<td>Biscoe</td>
<td>230</td>
</tr>
<tr>
<td>Gentoo</td>
<td>Biscoe</td>
<td>230</td>
</tr>
<tr>
<td>Gentoo</td>
<td>Biscoe</td>
<td>230</td>
</tr>
<tr>
<td>Gentoo</td>
<td>Biscoe</td>
<td>230</td>
</tr>
<tr>
<td>Gentoo</td>
<td>Biscoe</td>
<td>229</td>
</tr>
<tr>
<td>Gentoo</td>
<td>Biscoe</td>
<td>229</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>species</th>
<th>island</th>
<th>flipper_length_mm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adelie</td>
<td>Biscoe</td>
<td>172</td>
</tr>
<tr>
<td>Adelie</td>
<td>Torgersen</td>
<td>176</td>
</tr>
<tr>
<td>Adelie</td>
<td>Dream</td>
<td>178</td>
</tr>
<tr>
<td>Adelie</td>
<td>Dream</td>
<td>178</td>
</tr>
<tr>
<td>Adelie</td>
<td>Dream</td>
<td>178</td>
</tr>
<tr>
<td>Adelie</td>
<td>Torgersen</td>
<td>180</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>species</th>
<th>island</th>
<th>flipper_length_mm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gentoo</td>
<td>Biscoe</td>
<td>231</td>
</tr>
<tr>
<td>Gentoo</td>
<td>Biscoe</td>
<td>230</td>
</tr>
<tr>
<td>Gentoo</td>
<td>Biscoe</td>
<td>230</td>
</tr>
<tr>
<td>Gentoo</td>
<td>Biscoe</td>
<td>230</td>
</tr>
<tr>
<td>Gentoo</td>
<td>Biscoe</td>
<td>230</td>
</tr>
<tr>
<td>Gentoo</td>
<td>Biscoe</td>
<td>230</td>
</tr>
<tr>
<td>Gentoo</td>
<td>Biscoe</td>
<td>230</td>
</tr>
<tr>
<td>Gentoo</td>
<td>Biscoe</td>
<td>229</td>
</tr>
<tr>
<td>Gentoo</td>
<td>Biscoe</td>
<td>229</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
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.

```python
penguins.flipper_length_mm.mean()
```

# Output:
200.915205

You can compute multiple aggregates at once using the aggregate method:

```python
penguins.aggregate([penguins.flipper_length_mm.mean(), penguins.bill_depth_mm.max()])
```

<table>
<thead>
<tr>
<th>Mean(flipper_length_mm)</th>
<th>Max(bill_depth_mm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>float64</td>
<td>float64</td>
</tr>
<tr>
<td>200.915205</td>
<td>21.5</td>
</tr>
</tbody>
</table>

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.

```python
penguins.group_by("species").aggregate()
```
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.

```python
penguins.group_by(['species', 'island']).aggregate()
```

<table>
<thead>
<tr>
<th>species</th>
<th>island</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>string</td>
</tr>
<tr>
<td>Adelie</td>
<td>Torgersen</td>
</tr>
<tr>
<td>Adelie</td>
<td>Biscoe</td>
</tr>
<tr>
<td>Adelie</td>
<td>Dream</td>
</tr>
<tr>
<td>Gentoo</td>
<td>Biscoe</td>
</tr>
<tr>
<td>Chinstrap</td>
<td>Dream</td>
</tr>
</tbody>
</table>

Now, if we add an aggregation function to that, we start to really open things up.

```python
penguins.group_by(['species', 'island']).aggregate(penguins.bill_length_mm.mean())
```

<table>
<thead>
<tr>
<th>species</th>
<th>island</th>
<th>Mean(bill_length_mm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>string</td>
<td>float64</td>
</tr>
<tr>
<td>Adelie</td>
<td>Torgersen</td>
<td>38.950980</td>
</tr>
<tr>
<td>Adelie</td>
<td>Biscoe</td>
<td>38.975000</td>
</tr>
<tr>
<td>Adelie</td>
<td>Dream</td>
<td>38.501786</td>
</tr>
<tr>
<td>Gentoo</td>
<td>Biscoe</td>
<td>47.504878</td>
</tr>
<tr>
<td>Chinstrap</td>
<td>Dream</td>
<td>48.833824</td>
</tr>
</tbody>
</table>

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.

```python
penguins.group_by(['species', 'island']).aggregate(
    [penguins.bill_length_mm.mean(), penguins.flipper_length_mm.max()]
)
```

<table>
<thead>
<tr>
<th>species</th>
<th>island</th>
<th>Mean(bill_length_mm)</th>
<th>Max(flipper_length_mm)</th>
</tr>
</thead>
</table>
If we need more specific groups, we can add to the `group_by`.

```python
penguins.group_by(["species", "island", "sex"]).aggregate(
    [penguins.bill_length_mm.mean(), penguins.flipper_length_mm.max()]
)
```

<table>
<thead>
<tr>
<th>species</th>
<th>island</th>
<th>sex</th>
<th>Mean(bill_length_mm)</th>
<th>Max(flipper_length_mm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>string</td>
<td>string</td>
<td>float64</td>
<td>int64</td>
</tr>
<tr>
<td>---------</td>
<td>--------</td>
<td>---------</td>
<td>-----------------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>Adelie</td>
<td>Torgersen</td>
<td>male</td>
<td>40.586957</td>
<td>210</td>
</tr>
<tr>
<td>Adelie</td>
<td>Torgersen</td>
<td>female</td>
<td>37.554167</td>
<td>196</td>
</tr>
<tr>
<td>Adelie</td>
<td>Torgersen</td>
<td>NULL</td>
<td>37.925000</td>
<td>193</td>
</tr>
<tr>
<td>Adelie</td>
<td>Biscoe</td>
<td>male</td>
<td>40.590909</td>
<td>203</td>
</tr>
<tr>
<td>Adelie</td>
<td>Biscoe</td>
<td>female</td>
<td>37.359091</td>
<td>199</td>
</tr>
<tr>
<td>Adelie</td>
<td>Dream</td>
<td>female</td>
<td>36.911111</td>
<td>202</td>
</tr>
<tr>
<td>Adelie</td>
<td>Dream</td>
<td>male</td>
<td>40.071429</td>
<td>208</td>
</tr>
<tr>
<td>Adelie</td>
<td>Dream</td>
<td>NULL</td>
<td>37.500000</td>
<td>179</td>
</tr>
<tr>
<td>Gentoo</td>
<td>Biscoe</td>
<td>female</td>
<td>45.563793</td>
<td>222</td>
</tr>
<tr>
<td>Gentoo</td>
<td>Biscoe</td>
<td>male</td>
<td>49.473770</td>
<td>231</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

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?

```python
penguins.filter((penguins.sex == "female") & (penguins.year == 2008)).group_by(  
    ["island"]
).aggregate(penguins.body_mass_g.max())
```
What about the largest male penguin (by body mass) on each island for each year of data collection?

```sql
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"])
```

<table>
<thead>
<tr>
<th>island</th>
<th>year</th>
<th>max_body_mass</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dream</td>
<td>2007</td>
<td>4650</td>
</tr>
<tr>
<td>Torgersen</td>
<td>2007</td>
<td>4675</td>
</tr>
<tr>
<td>Biscoe</td>
<td>2007</td>
<td>6300</td>
</tr>
<tr>
<td>Torgersen</td>
<td>2008</td>
<td>4700</td>
</tr>
<tr>
<td>Dream</td>
<td>2008</td>
<td>4800</td>
</tr>
<tr>
<td>Biscoe</td>
<td>2008</td>
<td>6000</td>
</tr>
<tr>
<td>Torgersen</td>
<td>2009</td>
<td>4300</td>
</tr>
<tr>
<td>Dream</td>
<td>2009</td>
<td>4475</td>
</tr>
<tr>
<td>Biscoe</td>
<td>2009</td>
<td>6000</td>
</tr>
</tbody>
</table>

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**

```bash
pip install duckdb
pip install -U '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.

```python
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.

```python
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)

<table>
<thead>
<tr>
<th>id</th>
<th>fruit</th>
</tr>
</thead>
<tbody>
<tr>
<td>i32</td>
<td>str</td>
</tr>
<tr>
<td>1</td>
<td>banana</td>
</tr>
<tr>
<td>2</td>
<td>apple</td>
</tr>
<tr>
<td>3</td>
<td>mango</td>
</tr>
</tbody>
</table>
```

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 inter-compatibility 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

```python
import duckdb
from fsspec import filesystem

duckdb.register_filesystem(filesystem('gcs'))

duckdb.sql("SELECT * FROM read_csv('gcs:///bucket/file.csv')")
```

Note. 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 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:

<table>
<thead>
<tr>
<th>ticker</th>
<th>when</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>APPL</td>
<td>2001-01-01 00:00:00</td>
<td>1</td>
</tr>
<tr>
<td>APPL</td>
<td>2001-01-01 00:01:00</td>
<td>2</td>
</tr>
<tr>
<td>APPL</td>
<td>2001-01-01 00:02:00</td>
<td>3</td>
</tr>
<tr>
<td>MSFT</td>
<td>2001-01-01 00:00:00</td>
<td>1</td>
</tr>
<tr>
<td>MSFT</td>
<td>2001-01-01 00:01:00</td>
<td>2</td>
</tr>
<tr>
<td>MSFT</td>
<td>2001-01-01 00:02:00</td>
<td>3</td>
</tr>
<tr>
<td>GOOG</td>
<td>2001-01-01 00:00:00</td>
<td>1</td>
</tr>
<tr>
<td>GOOG</td>
<td>2001-01-01 00:01:00</td>
<td>2</td>
</tr>
<tr>
<td>GOOG</td>
<td>2001-01-01 00:02:00</td>
<td>3</td>
</tr>
</tbody>
</table>

We have another table containing portfolio *holdings* at various points in time:
To load these tables to DuckDB, run:

```sql
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:

```sql
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:

<table>
<thead>
<tr>
<th>ticker</th>
<th>when</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>APPL</td>
<td>2001-01-01 00:00:30</td>
<td>2.94</td>
</tr>
<tr>
<td>APPL</td>
<td>2001-01-01 00:01:30</td>
<td>48.26</td>
</tr>
<tr>
<td>GOOG</td>
<td>2001-01-01 00:00:30</td>
<td>23.45</td>
</tr>
<tr>
<td>GOOG</td>
<td>2001-01-01 00:01:30</td>
<td>21.16</td>
</tr>
</tbody>
</table>

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:

```sql
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.

<table>
<thead>
<tr>
<th>ticker</th>
<th>when</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>APPL</td>
<td>2000-12-31 23:59:30</td>
<td></td>
</tr>
<tr>
<td>APPL</td>
<td>2001-01-01 00:00:30</td>
<td>2.94</td>
</tr>
<tr>
<td>APPL</td>
<td>2001-01-01 00:01:30</td>
<td>48.26</td>
</tr>
<tr>
<td>GOOG</td>
<td>2000-12-31 23:59:30</td>
<td></td>
</tr>
<tr>
<td>GOOG</td>
<td>2001-01-01 00:00:30</td>
<td>23.45</td>
</tr>
<tr>
<td>GOOG</td>
<td>2001-01-01 00:01:30</td>
<td>21.16</td>
</tr>
<tr>
<td>DATA</td>
<td>2000-12-31 23:59:30</td>
<td></td>
</tr>
<tr>
<td>DATA</td>
<td>2001-01-01 00:00:30</td>
<td></td>
</tr>
<tr>
<td>DATA</td>
<td>2001-01-01 00:01:30</td>
<td></td>
</tr>
</tbody>
</table>

**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:

```sql
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.

**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.

```sql
CREATE TABLE corpus AS
    SELECT * FROM 'https://blobs.duckdb.org/data/shakespeare.parquet';
DESCRIBE corpus;
```

<table>
<thead>
<tr>
<th>column_name</th>
<th>column_type</th>
<th>null</th>
</tr>
</thead>
<tbody>
<tr>
<td>line_id</td>
<td>VARCHAR</td>
<td>YES</td>
</tr>
<tr>
<td>play_name</td>
<td>VARCHAR</td>
<td>YES</td>
</tr>
<tr>
<td>line_number</td>
<td>VARCHAR</td>
<td>YES</td>
</tr>
<tr>
<td>speaker</td>
<td>VARCHAR</td>
<td>YES</td>
</tr>
<tr>
<td>text_entry</td>
<td>VARCHAR</td>
<td>YES</td>
</tr>
</tbody>
</table>

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.

```sql
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?
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.

```sql
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;
```

```
<table>
<thead>
<tr>
<th>score</th>
<th>line_id</th>
<th>play_name</th>
<th>speaker</th>
<th>text_entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.68</td>
<td>H4/2.4.115</td>
<td>Henry IV</td>
<td>PRINCE HENRY</td>
<td>Didst thou never see Titan kiss a dish of ...</td>
</tr>
<tr>
<td>3.78</td>
<td>H4/1.2.21</td>
<td>Henry IV</td>
<td>FALSTAFF</td>
<td>prologue to an egg and butter.</td>
</tr>
<tr>
<td>3.78</td>
<td>H4/2.1.55</td>
<td>Henry IV</td>
<td>Chamberlain</td>
<td>They are up already, and call for eggs and ...</td>
</tr>
<tr>
<td>3.78</td>
<td>H4/4.2.21</td>
<td>Henry IV</td>
<td>FALSTAFF</td>
<td>toasts-and-butter, with hearts in their be ...</td>
</tr>
<tr>
<td>3.78</td>
<td>H4/4.2.62</td>
<td>Henry IV</td>
<td>PRINCE HENRY</td>
<td>already made thee butter. But tell me, Jac ...</td>
</tr>
<tr>
<td>3.78</td>
<td>AWW/4.1.40</td>
<td>Alls well that end ...</td>
<td>PAROLLES</td>
<td></td>
</tr>
<tr>
<td>3.78</td>
<td>AWW/5.2.9</td>
<td>Alls well that end ...</td>
<td>Clown</td>
<td>henceforth</td>
</tr>
<tr>
<td>3.78</td>
<td>AYLI/3.2.93</td>
<td>As you like it</td>
<td>TOUCHSTONE</td>
<td>right</td>
</tr>
<tr>
<td>3.78</td>
<td>KL/2.4.132</td>
<td>King Lear</td>
<td>Fool</td>
<td>kindness to his horse, buttered his hay.</td>
</tr>
<tr>
<td>3.78</td>
<td>MWW/2.2.260</td>
<td>Merry Wives of Win...</td>
<td>FALSTAFF</td>
<td>Hang him, mechanical salt-butter rogue! I ...</td>
</tr>
<tr>
<td>3.78</td>
<td>MWW/2.2.284</td>
<td>Merry Wives of Win...</td>
<td>FORD</td>
<td>rather trust a Fleming with my butter, Par ...</td>
</tr>
<tr>
<td>3.78</td>
<td>MWW/3.5.7</td>
<td>Merry Wives of Win...</td>
<td>FALSTAFF</td>
<td>Ill have my brains taen out and buttered, ...</td>
</tr>
<tr>
<td>3.78</td>
<td>MWW/3.5.102</td>
<td>Merry Wives of Win...</td>
<td>FALSTAFF</td>
<td>to heat as butter; a man of continual diss ...</td>
</tr>
<tr>
<td>6.39</td>
<td>H4/2.4.494</td>
<td>Henry IV</td>
<td>Carrier</td>
<td>As fat as butter.</td>
</tr>
<tr>
<td>14 rows</td>
<td>5 columns</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
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.
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).

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 DBaever window and clicking Database > DriverManager 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 DBaever 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.
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:

```sql
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. For now, we recommend using the latest bleeding edge JDBC driver (0.8.2) as a number of fixes have been made for time compatibility. Note that 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 snapshot jar

- 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. The current version of the Taco is not signed, so you will need to launch Tableau with signature validation disabled. (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 Tacofile to `/opt/tableau/connectors`. On Windows, copy the Tacofile 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 Desktop** Copy the Tacofile to the `/Users/[MacOS 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:

```plaintext
805
```
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`
DuckDB Documentation

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 (YOUR_SELECT_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-separated 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.

```bash
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"
```
Under the Hood
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 $\to$ **BoundStatement**
- QueryNode $\to$ **BoundQueryNode**
- TableRef $\to$ **BoundTableRef**
- ParsedExpression $\to$ **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 DPcpp 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 **BoundColumnRefExpression** 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. The execution model is a vectorized volcano model, where DataChunks are pulled from the root node of the physical operator tree. Each PhysicalOperator itself defines how it grants its result. A PhysicalTableScan node will pull the chunk from the base tables on disk, whereas a PhysicalHashJoin will perform a hash join between the output obtained from its child nodes.

Storage

The DuckDB internal storage format is currently in flux, and is expected to change with each release until we reach v1.0.0.

How to Move Between Storage Formats

When you update DuckDB and open a 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 two-liner (to be adapted with the file names and executable locations) is:

```
$ /older/version/duckdb mydata.db -c "EXPORT DATABASE 'tmp'"
$ /newer/duckdb mydata.new.db -c "IMPORT DATABASE 'tmp'"
```

After this mydata.db will be untouched with the old format, mydata.new.db will contain the same data but in a format accessible from more recent DuckDB, and folder tmp will old the same data in an universal format as different files.

Check EXPORT documentation for more details on the syntax.

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
00000010 00 00 00 00 ....
00000014 813
```
A simple example of reading the storage version using Python is below.

```python
import struct

pattern = struct.Struct('8s4sQ')

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](https://github.com/duckdb/duckdb) on GitHub. To see the commits that changed each storage version, see the [commit log](https://github.com/duckdb/duckdb/commit).

<table>
<thead>
<tr>
<th>Storage version</th>
<th>DuckDB version(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>v0.9.0, v0.9.1, v0.9.2</td>
</tr>
<tr>
<td>51</td>
<td>v0.8.0, v0.8.1</td>
</tr>
<tr>
<td>43</td>
<td>v0.7.0, v0.7.1</td>
</tr>
<tr>
<td>39</td>
<td>v0.6.0, v0.6.1</td>
</tr>
<tr>
<td>38</td>
<td>v0.5.0, v0.5.1</td>
</tr>
<tr>
<td>33</td>
<td>v0.3.3, v0.3.4, v0.4.0</td>
</tr>
<tr>
<td>31</td>
<td>v0.3.2</td>
</tr>
<tr>
<td>27</td>
<td>v0.3.1</td>
</tr>
<tr>
<td>25</td>
<td>v0.3.0</td>
</tr>
<tr>
<td>21</td>
<td>v0.2.9</td>
</tr>
<tr>
<td>18</td>
<td>v0.2.8</td>
</tr>
<tr>
<td>17</td>
<td>v0.2.7</td>
</tr>
<tr>
<td>15</td>
<td>v0.2.6</td>
</tr>
<tr>
<td>13</td>
<td>v0.2.5</td>
</tr>
<tr>
<td>11</td>
<td>v0.2.4</td>
</tr>
<tr>
<td>6</td>
<td>v0.2.3</td>
</tr>
<tr>
<td>4</td>
<td>v0.2.2</td>
</tr>
<tr>
<td>1</td>
<td>v0.2.1 and prior</td>
</tr>
</tbody>
</table>

### 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.

**Execution Format**

*Vector* is the container format used to store in-memory data during execution. *Data Chunk* 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.

```sql
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 indices 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 UnifiedVector-Format 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.

```c
struct string_t {
    union {
        struct {
            uint32_t length;
            char prefix[4];
            char *ptr;
        } pointer;
        struct {
            uint32_t length;
            char inlined[12];
        } inlined;
    }
};
```
DuckDB Documentation

```c
} 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.

```c
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 [][]`:

```json
{
    "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.
Developer Guides

Building DuckDB from Source

**Note.** DuckDB binaries are available for stable and nightly builds on the installation page. You should only build DuckDB under specific circumstances, such as when running on a specific architecture or building an unmerged pull request.

**Prerequisites**

DuckDB needs CMake and a C++11-compliant compiler (e.g., GCC, Apple-Clang, MSVC). Additionally, we recommend using the **Ninja build system**.

**Linux Packages**  Install the required packages with the package manager of your distribution.

Fedora, CentOS, and Red Hat:

```bash
sudo yum install -y git g++ cmake ninja-build
```

Ubuntu and Debian:

```bash
sudo apt-get update
sudo apt-get install -y git g++ cmake ninja-build
```

Alpine Linux:

```bash
apk add g++ git make cmake ninja
```

**macOS**  Install Xcode and **Homebrew**. Then, install the required packages with:

```bash
brew install cmake ninja
```

**Windows**  Consult the **Windows CI workflow** for a list of packages used to build DuckDB on Windows.

The DuckDB Python package requires the **Microsoft Visual C++ Redistributable package** to be built and to run.
Building DuckDB

To build DuckDB we use a Makefile which in turn calls into CMake. We also advise using Ninja as the generator for CMake.

```bash
GEN=ninja make
```

It is not advised to directly call CMake, as the Makefile sets certain variables that are crucial to properly building the package.

**Build Type**  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 DEBUG blocks. The special debug defines are not automatically set for this build however.

**relassert**  This build does not trigger the `#ifdef DEBUG` DEBUG code blocks, but 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.

**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.
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, our in-house Benchmark testing suite is built. More information about this can be found here.

**BUILD_JDBC**  When this flag is set, the Java package is built.

**BUILD_ODBC**  When this flag is set, the ODBC package is built.

Extension Flags  For every in-tree extension that is maintained by core DuckDB there exists a flag to enable building and statically linking the extension into the build.

**BUILD_AUTOCOMPLETE**  When this flag is set, the autocomplete extension is built.

**BUILD_ICU**  When this flag is set, the icu extension is built.

**BUILD_TPCH**  When this flag is set, the tpch extension is built, this enables TPCH-H data generation and query support using dbgen.

**BUILD_TPCDS**  When this flag is set, the tpcds extension is built, this enables TPC-DS data generation and query support using dsdgen.

**BUILD_TPCE**  When this flag is set, the TPCE extension is built, unlike TPC-H and TPC-DS this does not enable data generation and query support, but does enable tests for TPC-E through our test suite.

**BUILD_FTS**  When this flag is set, the fts (full text search) extension is built.

**BUILD_HTTPFS**  When this flag is set, the httpfs extension is built.
**BUILD_JSON**  When this flag is set, the **json extension** is built.

**BUILD_INET**  When this flag is set, the **inet extension** is built.

**BUILD_SQLSMITH**  When this flag is set, the **SQLSmith extension** is built.

**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.

**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 supported/can cause problems. Julia is an example of this. With this flag enabled, the sanitizers are disabled for the build.

**Troubleshooting**

**Building the R Package is Slow**  By default, R compiles packages using a single thread. To parallelize the compilation, create or edit the `~/.R/Makevars` file, and add the following content:

```
MAKEFLAGS = -j8
```

**Building the R Package on Linux aarch64**  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
```

To work around this, create or edit the `~/.R/Makevars` file:

```
ALL_CXXFLAGS = $(PKG_CXXFLAGS) -fPIC $(SHLIB_CXXFLAGS) $(CXXFLAGS)
```

**Building the httpfs Extension and Python Package on macOS**  Problem: The build fails on macOS when both the httpfs extension and the Python package are included:

```
GEN=ninja BUILD_PYTHON=1 BUILD_HTTPFS=1 make
```

```
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
```

**Solution:** In general, we recommended using the nightly builds, available under GitHub main (Bleeding Edge) on the installation page. If you would like to build DuckDB from source, avoid using the BUILD_PYTHON=1 flag unless you are actively developing the Python library. Instead, first build the httpfs extension (if required), then build and install the Python package separately using pip:

```
GEN=ninja BUILD_HTTPFS=1 make
```

If the next line complains about pybind11 being missing, or --use-pep517 not being supported, make sure you're using a modern version of pip and setuptools. python3 -m pip install pip --user first.

```
python3 -m pip install tools/pythonpkg --use-pep517 --user
```
Building the httpfs Extension on Linux

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
GEN=ninja BUILD_HTTPFS=1 make
```

Profiling

Profiling is important to help understand why certain queries exhibit specific performance characteristics. DuckDB contains several built-in features to enable query profiling that will be explained on this page.

For the examples on this page we will use the following example data set:

```
CREATE TABLE students (sid INTEGER PRIMARY KEY, name VARCHAR);
CREATE TABLE exams (cid INTEGER, sid INTEGER, grade INTEGER, PRIMARY KEY (cid, sid));
```

```
INSERT INTO students VALUES (1, 'Mark'), (2, 'Hannes'), (3, 'Pedro');
INSERT INTO exams VALUES (1, 1, 8), (1, 2, 8), (1, 3, 7), (2, 1, 9), (2, 2, 10);
```

Explain Statement

The first step to profiling a database engine is figuring out what execution plan the engine is using. The EXPLAIN statement allows you to peek into the query plan and see what is going on under the hood.

The EXPLAIN statement displays the physical plan, i.e., the query plan that will get executed.

To demonstrate, see the below example:

```
CREATE TABLE students (name VARCHAR, sid INT);
CREATE TABLE exams (eid INT, subject VARCHAR, sid INT);
INSERT INTO students VALUES ('Mark', 1), ('Joe', 2), ('Matthew', 3);
INSERT INTO exams VALUES (10, 'Physics', 1), (20, 'Chemistry', 2), (30, 'Literature', 3);
EXPLAIN SELECT name FROM students JOIN exams USING (sid) WHERE name LIKE 'Ma%';
```

![Physical Plan]
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.

**Run-Time Profiling**

The query plan helps understand the performance characteristics of the system. However, often it is also necessary to look at the performance numbers of individual operators and the cardinalities that pass through them. For this, you can create a query-profile graph.

To create the query graphs it is first necessary to gather the necessary data by running the query. In order to
do that, we must first enable the run-time profiling. This can be done by prefixing the query with EXPLAIN ANALYZE:

```sql
EXPLAIN ANALYZE SELECT name FROM students JOIN exams USING (sid) WHERE name LIKE 'Ma%';
```

```
Total Time: 0.0008s
```

```
<table>
<thead>
<tr>
<th>EXPLAIN_ANALYZE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>(0.00s)</td>
</tr>
</tbody>
</table>

```

```
<table>
<thead>
<tr>
<th>PROJECTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>(0.00s)</td>
</tr>
</tbody>
</table>

```

```
<table>
<thead>
<tr>
<th>HASH_JOIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>INNER</td>
</tr>
<tr>
<td>sid = sid</td>
</tr>
<tr>
<td>EC: 1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>(0.00s)</td>
</tr>
</tbody>
</table>

```

```
<table>
<thead>
<tr>
<th>SEQ_SCAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>exams</td>
</tr>
<tr>
<td>sid</td>
</tr>
<tr>
<td>EC: 3</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>(0.00s)</td>
</tr>
</tbody>
</table>

```

```
<table>
<thead>
<tr>
<th>FILTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>prefix(name, 'Ma')</td>
</tr>
<tr>
<td>EC: 1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>(0.00s)</td>
</tr>
</tbody>
</table>
```
The output of EXPLAIN ANALYZE contains the estimated cardinality (EC), the actual cardinality, and the execution time for each operator.

It is also possible to save the query plan to a file, e.g., in JSON format:

```
-- All queries performed will be profiled, with output in json format.
-- By default the result is still printed to stdout.
PRAGMA enable_profiling='json';
-- Instead of writing to stdout, write the profiling output to a specific file on disk.
-- This has no effect for `EXPLAIN ANALYZE` queries, which will *always* be returned as query results.
PRAGMA profile_output='/path/to/file.json';
```

**Note.** This file is overwritten with each query that is issued. If you want to store the profile output for later it should be copied to a different file.

Now let us run the query that we inspected before:

```
SELECT name FROM students JOIN exams USING (sid) WHERE name LIKE 'Ma%';
```

After the query is completed, the JSON file containing the profiling output has been written to the specified file. We can then render the query graph using the Python script, provided we have the duckdb python module installed. This script will generate a HTML file and open it in your web browser.

```
python -m duckdb.query_graph /path/to/file.json
```

**Testing**

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 master branch.
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`.

### Writing Tests

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 the section "Catch Tests". However, in most cases you should write your tests in plain SQL.

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.

```sql
FROM test_all_types();
```

<table>
<thead>
<tr>
<th>bool</th>
<th>tinyint</th>
<th>smallint</th>
<th>int</th>
<th>bigint</th>
<th>hugeint</th>
</tr>
</thead>
<tbody>
<tr>
<td>union</td>
<td>struct</td>
<td>struct_of_arrays</td>
<td>array_of_structs</td>
<td>map</td>
<td></td>
</tr>
<tr>
<td>boolean</td>
<td>int8</td>
<td>int16</td>
<td>int32</td>
<td>int64</td>
<td>int128</td>
</tr>
<tr>
<td>struct(a integer, ...</td>
<td>struct(a integer[])...</td>
<td>struct(a integer, ...</td>
<td>map(varchar, varchar...) union(&quot;name&quot; varchar...)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>false</td>
<td>-128</td>
<td>-32768</td>
<td>-2147483648</td>
<td>-9223372036854775808</td>
<td></td>
</tr>
<tr>
<td>-1781411834604692393...</td>
<td>{'a': NULL, 'b': N...</td>
<td>{'a': NULL, 'b': N...</td>
<td>[]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>{}</td>
<td>Frank</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
### test_vector_types Function

The `test_vector_types` table function takes \( n \) arguments \( \text{col1}, \ldots, \text{coln} \) and an optional BOOLEAN argument `all_flat`. The function generates a table with \( n \) columns `test_vector`, `test_vector2`, \ldots, `test_vectorn`. In each row, each field contains values conforming to the type of their respective column.

**FROM** `test_vector_types(NULL::BIGINT);`

```
<table>
<thead>
<tr>
<th>test_vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>int64</td>
</tr>
</tbody>
</table>
```

-9223372036854775808
9223372036854775807
NULL

**FROM** `test_vector_types(NULL::ROW(i INTEGER, j VARCHAR, k DOUBLE), NULL::TIMESTAMP);`

```
<table>
<thead>
<tr>
<th>test_vector</th>
<th>test_vector2</th>
<th>timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>struct(i integer, j varchar, k double)</td>
<td>{'i': -2147483648, 'j': NULL, 'k': -1.7976931348623157e+308}</td>
<td>290309-12-22 (BC) 00:00:00</td>
</tr>
<tr>
<td></td>
<td>{'i': '2147483647', 'j': goo\se, 'k': 1.7976931348623157e+308}</td>
<td>294247-01-10 04:00:54.775806</td>
</tr>
<tr>
<td></td>
<td>{'i': NULL, 'j': NULL, 'k': NULL}</td>
<td>NULL</td>
</tr>
</tbody>
</table>
```

test_vector_types has an optional argument called `all_flat` of type BOOL. This only affects the internal representation of the vector.
sqllogictest

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 the section "Catch Tests". However, in most cases you should write your tests in plain SQL.

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.
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.

**sqllogictest - 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.

```# 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.
# 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.

# 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:

# run tests the tests 200..250
build/debug/test/unittest --start-offset=200 --end-offset=250

These are also available in percentages:

# 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
# run only the tests labeled in the file
$ build/debug/test/unittest -f test.list

sqllogictest - 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.

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

```sql
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.

```sql
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.

```sql
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.

```sql
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 sqllogictest 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:
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:

```sql
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 sqllogic tests 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:

```sql
query I nosort r43
SELECT 42+1;
```

```sql
query I nosort r43
SELECT 44-1;
```
sqllogictest - 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.

```sql
# 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:

```sql
statement ok
PRAGMA disable_checkpoint_on_shutdown

statement ok
PRAGMA wal_autocheckpoint='1TB';
```

sqllogictest - 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:

```sql
# create the table integers with the 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.

<table>
<thead>
<tr>
<th>Preset</th>
<th>Expansion</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;compression&gt;</code></td>
<td>utinyint usmallint uinteger ubigint</td>
</tr>
<tr>
<td></td>
<td>none uncompressed rle bitpacking dictionary fsst chimp patas</td>
</tr>
<tr>
<td></td>
<td><code>tinyint smallint integer bigint hugeint</code></td>
</tr>
<tr>
<td></td>
<td><code>unsigned</code></td>
</tr>
<tr>
<td><code>&lt;integral&gt;</code></td>
<td>numeric bool interval varchar json</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Note.** 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.

```sql
-- create the table integers with the values [0, 1, .., 98, 99]
CREATE TABLE integers AS SELECT * FROM range(0, 100, 1) t1(i);

-- create the table strings with 100X the value "hello"
CREATE TABLE strings AS SELECT 'hello' AS s FROM range(0, 100, 1);
```

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)`.

**sqllogictest - 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:

```sql
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.

```sql
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.

```sql
concurrentloop i 1 10

statement maybe
CREATE OR REPLACE TABLE t2 AS (SELECT -5412403386577348004002656426531535114 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:

```cpp
#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.
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.