DuckDB-Wasm: Fast Analytical Processing for the Web

André Kohn
andre.kohn@tum.de
Technische Universität München

Dominik Moritz
domoritz@cmu.edu
Carnegie Mellon University

Mark Raasveldt
m.raasveldt@cwi.nl
Centrum Wiskunde & Informatica

Hannes Mühleisen
hannes.muehleisen@cwi.nl
Centrum Wiskunde & Informatica

Thomas Neumann
thomas.neumann@tum.de
Technische Universität München

ABSTRACT

We introduce DuckDB-Wasm, a WebAssembly version of the database system DuckDB, to provide fast analytical processing for the Web. DuckDB-Wasm evaluates SQL queries asynchronously in web workers, supports efficient user-defined functions written in JavaScript, and features a browser-agnostic filesystem that reads local and remote data in pages. DuckDB-Wasm outperforms previous data processing libraries for the Web in the TPC-H benchmark at multiple scale factors. We demonstrate the capabilities of an analytical database in the browser using an interactive SQL shell.

PVLDB Reference Format:
André Kohn, Dominik Moritz, Mark Raasveldt, Hannes Mühleisen, and Thomas Neumann. DuckDB-Wasm: Fast Analytical Processing for the Web. PVLDB, 15(12): 3574 - 3577, 2022.
doi:10.14778/3554821.3554847

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/duckdb/duckdb-wasm.

1 INTRODUCTION

The web browser has evolved to a universal computation platform. Its rise has been accompanied by increasing requirements for the browser programming language JavaScript. JavaScript was designed to be flexible which comes at the cost of a reduced processing efficiency. This is pronounced when considering the execution times of complex data analysis tasks that often fall behind the native execution by orders of magnitude. In the past, analysis tasks have therefore been pushed to servers that tie any client-side processing to additional round-trips over the internet. These round-trips introduce network latencies that negatively affect interactive data exploration [4].

The processing capabilities of browsers were boosted significantly in 2017 with the release of WebAssembly [1]. WebAssembly is a collaborative effort to design a portable low-level binary instruction format for a safe stack-based virtual machine. It is supported by major browser engines today and serves as efficient compilation target for programming languages like C++. WebAssembly aims to execute programs at native speed and supersedes JavaScript for performance-critical applications in browsers.

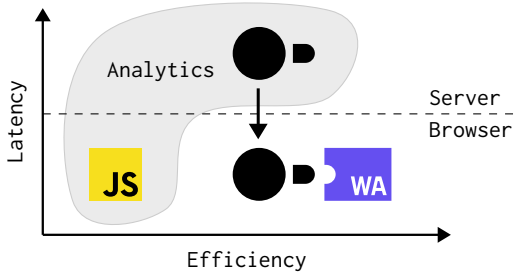


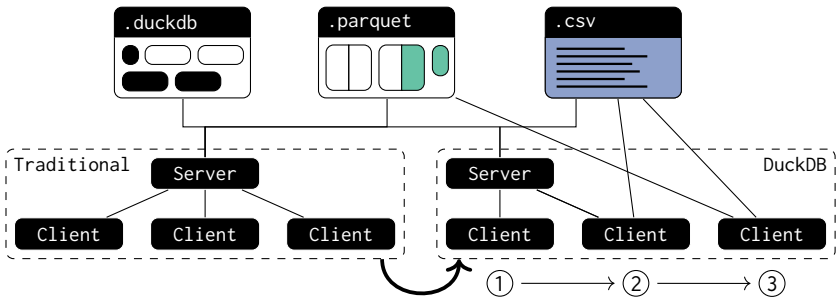
Figure 1: Browser-based analytics tools process data either locally with a low efficiency or on servers with a high latency. DuckDB-Wasm pushes the boundaries with fast analytical processing for the Web.

The rise of WebAssembly presents an opportunity for the database DuckDB to bring fast analytical data processing to the Web. DuckDB is a purpose-built *embeddable* database for interactive analytics [5, 6]. Embeddable databases are linked to programs as libraries and run in their processes. This design distinguishes DuckDB from stand-alone data management systems and allows for tight integrations into different environments. We identified one such environment to be the browser and introduce DuckDB-Wasm, a comprehensive data analysis library for the Web.

Figure 1 presents a difficult trade-off that motivates a more efficient analytical processing in the browser. Web-based analysis tools can either process data locally or on more powerful remote servers. Browsers are limited by the efficiency of the language JavaScript but increase the interactivity by saving costly round-trips over the internet. This contrast asks for a continuous assessment, if the higher efficiency of remote servers justifies higher base latencies. The decision is non-trivial and offers the popular escape-hatch to always run the entire analysis remotely. DuckDB-Wasm accelerates the data processing in browsers and sheds new light on local processing as driver for interactive analytics.

This paper demonstrates the concept of WebAssembly-driven analytics. We give an overview about the design of DuckDB-Wasm in Section 2. Section 3 compares the performance with existing libraries in the Web. Section 4 demonstrates the capabilities of DuckDB-Wasm using an interactive SQL shell in the browser. We close with a summary of the paper in Section 5.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 15, No. 12 ISSN 2150-8097.
doi:10.14778/3554821.3554847




```
CREATE VIEW recentdaily AS (
  SELECT date_trunc('day', ts) AS day,
         name, max(price) AS price
  FROM 's3://bucket/stocks.parquet'
  WHERE ts > current_date - 30
  GROUP BY day);

SELECT r.day,p.name,max(p.count*r.price)
  FROM 'https://remote/portfolio.csv' p,
       recentdaily r
  WHERE p.name = r.name GROUP BY ALL;
```

Figure 2: A SQL script that downloads stock data from AWS S3 stored in a Parquet file and joins it with a portfolio stored in a CSV file. The left side presents multiple ways to execute the script in a distributed setting. ① shows the traditional separation between client and server, ③ a fully local execution, ② a hybrid mode in between.

2 DESIGN AND IMPLEMENTATION

In this section, we introduce four key design aspects of DuckDB-Wasm. We describe the interaction with WebAssembly, a browser-agnostic web filesystem, the role of web workers and the efficient integration of user-defined functions.

2.1 Embedding WebAssembly

We translate DuckDB to WebAssembly using the compiler *Emscripten* that build on the LLVM framework [7]. DuckDB is written in C++, a language that differs significantly from JavaScript in areas such as function calls, data types and memory ownership. WebAssembly does not conceal these language differences but pronounces them further through the memory isolation towards the JavaScript heap. DuckDB-Wasm therefore uses Arrow for efficient data exchange between the two languages. Arrow is a columnar format that is organized in chunks of column vectors, called record batches, and supports zero-copy reads with a small overhead. DuckDB-Wasm serializes results as Arrow IPC streams in C++ and then reads them directly from the WebAssembly heap using JavaScript.

2.2 Web Filesystem

DuckDB-Wasm integrates a filesystem that is agnostic to the browser environment. DuckDB is built on top of a virtual filesystem that decouples higher level tasks, such as reading a Parquet file, from low-level filesystem APIs that are specific to the operating system. We use this abstraction in DuckDB-Wasm to tailor filesystem implementations to the different WebAssembly environments, such as the browser and `Node.js`. In the browser, file operations are mapped either to the web File API for local files or synchronous `XMLHttpRequests` for remote data. We use the HTTP range header to request parts of remote files and maintain exponentially growing readahead buffers to reduce the total number of requests.

Figure 2 shows an example script with two SQL statements. The script filters stock price data of the last 30 days, stored in a Parquet file on AWS S3. Afterwards, it joins the data with a stock portfolio in a CSV file that is specified as raw HTTP URL. The green color hints at relevant stock prices in the Parquet file and indicates, that DuckDB-Wasm can skip row groups based on the filter predicate. The CSV file is colored in blue and is fetched completely.

The figure also displays different execution strategies for the two statements. Traditionally, the capabilities of browsers have been limited, favoring server-based analytics. With this model, a rising number of clients increases the load on the infrastructure and demands for elastic scaling. DuckDB-Wasm, in contrast, offers a choice between the three options labeled with ① to ③. ① adopts the traditional approach where the entire computation would be done by dedicated servers. ③ presents distributed computations where every client runs the analysis locally. ② combines both approaches by aggregating and filtering stock data using a server and joining the result with the portfolio on the local device.

2.3 Web Workers

The format Arrow also facilitates the offloading of DuckDB-Wasm to dedicated web workers as we can pass Arrow buffers efficiently through the browser's message API. We use web workers for multiple reasons. First, they unblock the browser's main event loop and allow running complex analytical queries without pausing user interface updates. Second, DuckDB-Wasm can select between worker versions dynamically. Since the release of WebAssembly back in 2017, which is now referred to as MVP, the standard has been evolving. New features, such as WebAssembly Exceptions and SIMD, find their way into the browsers at different speeds, creating a fractured space of post-MVP functionality. These features can bring flat performance improvements and are indispensable when aiming for a maximum speed. We compile DuckDB-Wasm with multiple feature profiles and select a worker based on dynamic browser checks.

2.4 User-Defined Functions

DuckDB-Wasm further simplifies the interaction with JavaScript through user-defined functions. DuckDB follows a vectorized execution model and processes queries chunk-wise to amortize the overhead of query interpretation and benefit from superscalar capabilities of the CPU. We use this vectorization to implement efficient user-defined functions in the browser. Users can register JavaScript functions in DuckDB-Wasm and reference them within a SQL query. During execution, the runtime system reads the current chunk data directly from the WebAssembly heap and passes the tuples to the user function in a compact loop.

Table 1: Execution times in seconds for TPC-H queries at the scale factors 0.01, 0.1 and 0.5.

	SF = 0.01				SF = 0.1				SF = 0.5			
#	DuckDB	SQL.js	Arquero	Lovefield	DuckDB	SQL.js	Arquero	Lovefield	DuckDB	SQL.js	Arquero	Lovefield
1	**0.005**	0.054	0.063	0.046	**0.047**	0.584	0.823	0.805	**0.235**	3.412	9.080	4.979
2	0.002	**0.002**	0.003	–	**0.005**	0.019	0.122	–	**0.015**	0.101	3.314	–
3	**0.002**	0.014	0.047	0.020	**0.008**	0.150	0.570	0.281	**0.048**	0.791	5.923	1.626
4	**0.001**	0.003	0.028	0.014	**0.008**	0.033	0.361	0.234	**0.048**	0.181	2.060	1.573
5	**0.003**	0.013	0.020	0.008	**0.008**	0.153	0.539	0.178	**0.049**	0.875	9.498	1.415
6	**0.001**	0.010	0.009	0.007	**0.005**	0.100	0.107	0.121	**0.026**	0.532	0.622	0.793
7	**0.003**	0.017	0.049	0.016	**0.016**	0.202	0.491	0.498	**0.086**	1.174	2.574	12.391
8	**0.004**	0.020	0.016	0.008	**0.010**	0.288	0.157	0.189	**0.070**	1.831	0.894	1.399
9	**0.006**	0.027	0.716	0.211	**0.057**	0.481	–	3.243	**0.483**	3.317	–	–
10	**0.003**	0.010	0.029	0.013	**0.020**	0.106	0.420	0.270	**0.116**	0.559	7.784	1.678
11	0.001	0.004	**0.001**	–	**0.003**	0.050	0.007	–	**0.007**	0.265	0.039	–
12	**0.003**	0.009	0.012	0.017	**0.019**	0.096	0.154	0.277	**0.089**	0.493	1.992	1.704
13	**0.002**	0.020	0.012	0.044	**0.014**	0.327	0.197	0.572	**0.068**	2.246	5.200	3.251
14	**0.001**	0.009	0.039	0.013	**0.005**	0.094	0.405	0.223	**0.024**	0.498	2.261	1.475
$\varnothing_{geo}$	**0.003**	0.012	0.023	0.019	**0.013**	0.142	0.268	0.338	**0.073**	0.809	2.822	2.049

3 TPC-H BENCHMARK

In this section, we experimentally evaluate analytical query processing with DuckDB-Wasm using the TPC-H benchmark. Our experiments were performed on a Ryzen 5800X CPU with Node.js v17.6.0 that is powered by the V8 engine v9.6.

We compare execution times of TPC-H queries using DuckDB-Wasm and the systems SQL.js, Arquero, and Lovefield. SQL.js is the WebAssembly version of the database SQLite and supports all TPC-H queries out of the box. Lovefield only supports a custom SQL-like API but optimizes query plans internally. However, Lovefield does not support arithmetic operations and nested subqueries within the plan which makes it difficult to run more complex TPC-H queries. Arquero only provides a DataFrame-like API without any upfront optimization. We therefore constructed the TPC-H queries manually for Arquero using the optimized plans produced by the optimizer of a relational database.

We ran the benchmark at the scale factors 0.01, 0.1, and 0.5. A scale factor of 0.1 refers to approximately 100 MB of combined data, resulting in a range between 10 to 500 MB in the experiment. The WebAssembly memory is currently capped at 4 GB in browsers, leaving some room for higher scale factors. We omitted them in the benchmark because of the already significant differences between the systems at scale factor 0.5. Table 1 lists the execution times of the first 14 TPC-H queries. The table also shows the geometric means using a subset of all 22 queries, that are supported by every system. DuckDB-Wasm outperforms the competition by a factor of 10 to 100 across all scale factors. The two JavaScript libraries Arquero and Lovefield scale worse with a growing amount of data compared to the two WebAssembly systems.

The experiment confirms, that WebAssembly enables efficient data processing in the browser. It also shows that DuckDB-Wasm offers sub-second execution times for complex analytical queries on data sizes that may be considered large for the Web. We want to emphasize that DuckDB-Wasm does not substitute existing database systems when processing large amounts of data. Instead, DuckDB-Wasm aims to complement database servers to increase the interactivity for browser-manageable data subsets.

4 DEMONSTRATION SCENARIO

We demonstrate the capabilities of a WebAssembly database with an interactive SQL shell that runs entirely in the user's browser. The SQL shell is accessible at shell.duckdb.org and provides a command prompt for a local DuckDB-Wasm instance. The shell processes SQL statements and the following utility commands:

- **`.help`** prints additional information about the shell.
- **`.features`** lists browser features and the selected bundle.
- **`.timer (on|off)`** measures end-to-end execution times.
- **`.output (on|off)`** controls the printing of the query results.
- **`.examples`** lists SQL queries that scan remote Parquet files of the TPC-H benchmark at small scale factors and can be executed without further preparation.
- **`.files add`** registers local files in the virtual filesystem of DuckDB-Wasm. The registered files are not fully copied into the WebAssembly memory but are read chunk-wise through the web File API.
- **`.files (track|paging|reads) $FILE`** enables and visualizes filesystem statistics, tracking file ranges that have been read, prefetched and cached. This demonstrates the scanning of files using chunked reads or HTTP range requests.

We invite the audience to explore the remote TPCH-H data and their own local files ad-hoc in the browser using arbitrary SQL queries. We further propose to reproduce the following three observations: First, when scanning a Parquet file with a limit clause, DuckDB-Wasm only reads the metadata in the back of the file and the first bytes of required columns in the front. Second, aggregates like the global tuple count can be evaluated entirely on the Parquet metadata and finish quickly even on large remote files. Third, when

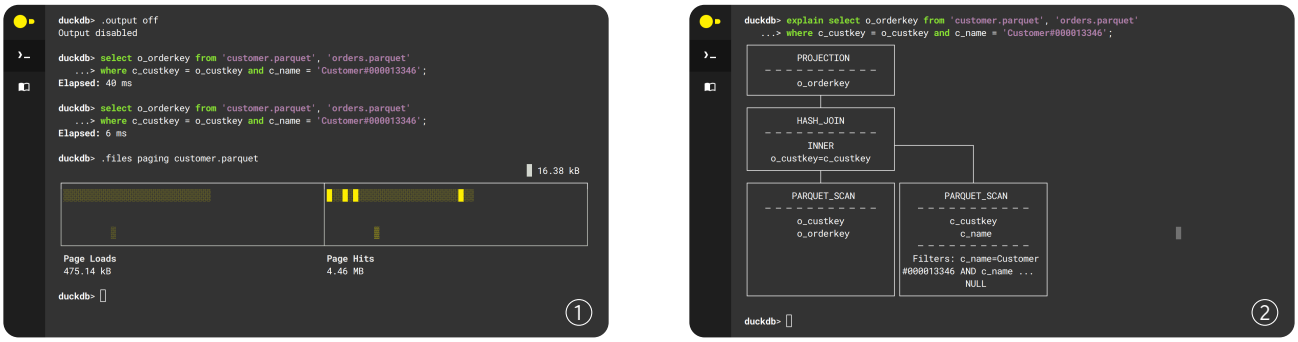


Figure 3: A shell that runs entirely in the browser and evaluates SQL queries using DuckDB-Wasm. The figure shows a query joining two parquet files with the relations *orders* and *customer* of the TPC-H benchmark at scale factor 0.1. ① lists the query results and page accesses, ② shows the query plan.

fully scanning a file, DuckDB-Wasm reads ranges of exponentially increasing sizes to reduce the overhead of individual reads.

Figure 3 shows the WebAssembly shell in action. The figure presents the execution of a query that joins data from two local Parquet files. The files store the relations *orders* and *customers* from the TPC-H benchmark at scale factor 0.1. *orders.parquet* contains 150'000 tuples and measures 11.8 MB. *customer.parquet* contains 15'000 tuples and accounts for 2.6 MB. ② shows the query plan that consists of two Parquet scans, a hash join on the customer key and a topmost projection. DuckDB-Wasm executes the query in 40 milliseconds with cold caches and in 6 milliseconds afterwards. ① also prints paging information of the customer data after running the query twice. It shows that DuckDB-Wasm reads 475 KB in total for the metadata in the back of the file and the required attributes in the front.

5 SUMMARY

In this paper, we introduced DuckDB-Wasm, a WebAssembly version of the database system DuckDB that provides fast analytical processing for the web. We outlined implementation details and showed that DuckDB-Wasm outperforms existing systems by a large margin in the TPC-H benchmark. The demonstration scenario presents an interactive shell that allows executing analytical SQL statements in the local browser. Nevertheless, we identify two major opportunities for future improvements.

First, we believe that WebAssembly unveils hitherto dormant potential for shared query processing between clients and servers. Pushing computation closer to the client eliminates costly round-trips over the internet and thus increases interactivity and scalability of in-browser analytics. However, client-sided analytics also stresses the importance of data locality and asks for a thorough optimization of distributed query plans. Distributed query plans should take into account where data is located, how computation and bandwidth resources can be scaled, and how query latencies evolve during interactive and repeated executions. We see the tandem of DuckDB and DuckDB-Wasm as a first step towards a universal data plane spanning across traditional database servers, clients, CDN workers, and computational storage.

Second, DuckDB-Wasm barely scratches the surface of efficient browser-agnostic data processing. The browser landscape is evolving with new APIs and WebAssembly capabilities at the horizon. Extensive filesystem support, for example, will evolve DuckDB-Wasm from an in-memory analytical query engine to a persistent database system that can bypass browser memory limitations completely through out-of-core operators. WebAssembly Module Linking will further facilitate the dynamic loading of DuckDB extensions for ICU timezones and full-text search. Additionally, multithreading in browsers has been hampered by the repercussions of the Spectre and Meltdown vulnerabilities [2, 3]. DuckDB scales seamlessly to a large number of cores outside of WebAssembly which could accelerate in-browser analytics even further in the future.

ACKNOWLEDGMENTS

We would like to thank all past, current and future contributors to DuckDB and DuckDB-Wasm. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 725286).

REFERENCES

[1] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the Web up to Speed with WebAssembly. *SIGPLAN Not.* 52, 6 (2017), 185–200.

[2] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *S&P*. 1–19.

[3] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, Mike Hamburg, and Raoul Strackx. 2020. Meltdown: reading kernel memory from user space. *Commun. ACM* 63, 6 (2020), 46–56.

[4] Zhicheng Liu and Jeffrey Heer. 2014. The Effects of Interactive Latency on Exploratory Visual Analysis. *IEEE Transactions on Visualization and Computer Graphics* 20, 12 (2014), 2122–2131.

[5] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *SIGMOD*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). 1981–1984.

[6] Mark Raasveldt and Hannes Mühleisen. 2020. Data Management for Data Science - Towards Embedded Analytics. In *CIDR*.

[7] Alon Zakai. 2011. Emscripten: an LLVM-to-JavaScript compiler. In *SIGPLAN*. 301–312.