MotherDuck: DuckDB in the cloud and in the client

RJ Atwal, Peter Boncz, Ryan Boyd, Antony Courtney, Till Döhmen, Florian Gerlinghoff, Jeff Huang, Joseph Hwang, Raphael Hyde, Elena Felder, Jacob Lacouture, Yves LeMaout, Boaz Leskes, Yao Liu, Alex Monahan, Dan Perkins, Tino Tereshko, Jordan Tigani, Nick Ursa, Stephanie Wang, Yannick Welsch
firstname@motherduck.com

ABSTRACT

We describe and demo MotherDuck: a new service that connects DuckDB to the cloud. MotherDuck provides the concept of *hybrid query processing*: the ability to execute queries partly on the client and partly in the cloud. We cover the motivation for MotherDuck and some of its use cases; and outline its system architecture, which heavily uses the extension mechanisms of DuckDB.

MotherDuck allows existing DuckDB users who use a laptop, like data scientists, to start using cloud computing without changing their queries: this can provide better performance as well as scalability to larger datasets. It also provides them the ability to share DuckDB databases with others through the cloud for collaboration.

Hybrid query processing opens the door to new data-intensive applications, such as low-latency analytical web apps, with DuckDB-wasm as the client running inside a browser. It also leads on to research questions, some of which we describe in the paper.

1 INTRODUCTION

MotherDuck is a new service that offers DuckDB data storage and serverless query processing in the cloud. DuckDB is an embedded analytical database system, originally developed at CWI [32]: where "embedded" means that it runs *inside* the driver/API library used by the client process that issues SQL queries. This implies that a complete DuckDB system runs inside its JDBC driver. Therefore, all clients using DuckDB have a local database engine, but with MotherDuck, queries can now also use a cloud DuckDB engine. Starting to use MotherDuck is as simple as typing `.open md;` in the DuckDB SQL shell. DuckDB's popularity stems from its high performance, as an analytical system; and its ease of use, being tightly integrated with Python and R data science libraries. MotherDuck inherits these properties, since its user interface is DuckDB.

Hybrid Query Processing. MotherDuck allows customers to store DuckDB databases in the cloud and query cloud-hosted Parquet, CSV, or JSON files, like other cloud databases. However, MotherDuck users can simultaneously have local DuckDB databases and local files and query all of these data sources within one same query. The MotherDuck optimizer will plan query operators to be executed close to where the data is. If all data sources are local, DuckDB will process the query completely locally, and if all are remote, the query will be executed in the cloud. If some data is local and some remote, then part of the query will be executed locally, and part remotely, using "bridge" operators that upload and download tuple streams between local and cloud – this we call *hybrid query processing*.

This paper is published under the Creative Commons Attribution 4.0 International (CC-BY 4.0) license. Authors reserve their rights to disseminate the work on their personal and corporate Web sites with the appropriate attribution, provided that you attribute the original work to the authors and CIDR 2024. 14th Annual Conference on Innovative Data Systems Research (CIDR '24). TODO, Chaminade, USA

Reducing Cloud Footprint. An observation from analyzing traces of cloud data warehouses [2, 10] is that >95% of databases are <1TB in size and >95% of queries involve <10GB of data. However, in order to address very large scale problems, current analytical cloud data platforms [5, 34, 13] rely on *scale-out* architectures, where a cloud-provisioned cluster of multiple nodes processes each single query. This adds to system complexity, which arguably 95% of the users and use cases do not need. Further, scale-out is not free performance-wise, since multi-node task scheduling unavoidably increases end-to-end query latency and introduces multi-node communication overhead: queries must shuffle data over the network between the nodes, adding network and [de]serialization CPU cost (see: "Scalability, but at what COST?" [27]).

Under the slogan "Big Data is Dead", MotherDuck therefore advocates a serverless architecture that avoids scale-out, opting for the simplicity of a single node architecture.[1] Here, elasticity for a single user is provided by varying the amount of cores and RAM given to a container (scale-up and -down). Workloads with many users can be scaled by spreading users over more/fewer nodes.

There is a carbon emission argument to be made for the efficiency of a single-node engine approach, that eschews mentioned multi-node overheads. Additionally, hybrid query processing allows to use local devices more intensively. Both these features of MotherDuck arguably lead to a reduced need for cloud hardware. While energy provisioning in cloud data centers is more efficient than in a laptop, the most recent calculations indicate that the carbon emissions of compute are dominated by hardware manufacturing cost regardless [15]. This certainly holds for laptops, which have a short lifespan and consequently their usage-related carbon footprint is dwarfed by their manufacturing carbon footprint. As such, one can argue that better use of local hardware that otherwise sits idle can reduce carbon emissions due to less cloud hardware being needed.

Enabling New Applications. Beyond any carbon argument, processing in both client and server opens up new applications, since the client can process certain queries locally. DuckDB is the only analytical DBMS that can be compiled to Web Assembly (wasm [8]), thanks to its policy to avoid taking any dependencies on other software packages (which are prone to break wasm transpilation). One can construct web-apps that embed DuckDB-wasm [19] in HTML, and connect these to MotherDuck in the cloud. When operating on local data or cached query results (temporary tables), this can power very low-latency applications, which are impossible in a cloud architecture due to wide-area network latency. An early demonstrator is the Mosaic framework for scalable, interactive data visualizations.[2] The web-interface of DuckDB also uses this new application architecture. MotherDuck can also make certain applications economically viable, as its architecture can reduce cloud usage

[1]MotherDuck blog entry: bit.ly/big-data-is-dead
[2]See uwdata.github.io/mosaic

and thus cloud cost. It could also make applications more secure: e.g., privacy-sensitive data could only be decrypted on the client and postprocessed there [33]. For data minimization (as demanded by e.g., GDPR), one can envision applications that query a *personal data store* [12] with privacy-sensitive data under the control of the user, and join this with cloud-resident data, under the control of the company that provides the application (a MotherDuck customer).

Scaling Existing Applications. The data processing scenarios that made DuckDB popular include data scientists designing and evaluating ETL or analysis pipelines on a laptop. Such scenarios can now be enhanced with the ability to schedule the designed pipelines to run in production in the cloud via MotherDuck, e.g., regularly feeding data from/into a data lake. These production pipelines also benefit from MotherDuck's ability to scale memory and CPU resources beyond what a laptop can offer. Data scientists can also use the MotherDuck `SHARE` and `ATTACH` features to *share* databases in the cloud and collaborate in a team. Scaling to the cloud requires only a connection to MotherDuck and no logic changes.

2 BACKGROUND: DUCKDB

DuckDB is a database system originally developed at CWI by Hannes Mühleisen and Mark Raasveldt. It is now being further developed by their spin-off company DuckDB Labs (which co-founded MotherDuck), while its core IP is in an open-source foundation.

State-of-the-Art Architecture. DuckDB uses these techniques[3]:

- columnar storage that is skippable (using MinMax statistics) [1]; with a variety of lightweight compression schemes [35][4].
- primary keys and foreign keys, backed by the ART index [24].
- storing columns with nested datatypes (lists of structs of list of..) efficiently in "shredded" sub-columns.
- database transactions (insert, delete, update) optimized for handling changes in bulk, rather than to individual rows.
- multi-version concurrency control optimized for fast scans [30].
- a vectorized query execution engine [4], that supports operating directly on lightweight-compressed data (RLE, Dict, FSST [3]).
- a LeanStore-inspired buffer manager [23], aiming for main-memory query processing speeds when data is SSD resident.
- a rich set of relational operators, including holistic window functions [25][5], but also `PIVOT` and lateral/range/AS-OF joins.[6]
- hyperloglog powered statistics and cardinality estimation [17].
- a rule-based optimizer using dynamic-programming join order enumeration, that also works in absence of PK/FK constraints [7].
- decorrelation of all correlated subqueries, using special joins [29].
- relational operator implementations for join, aggregation and sort [21], that run out-of-core with only gradual performance degradation as memory gets more tight.
- push-based operator execution[7], driven by a morsel-driven pipeline scheduler for self-balancing multi-core query processing [22].
- parallel execution that can efficiently maintain order from an order-aware data source into an order-aware sink (e.g., DataFrames).

[3]See also the CMU spring 2023 DuckDB invited lecture: bit.ly/duckdb-internals-cmu
[4]DuckDB blog entry: bit.ly/duckdb-blog-compression
[5]https://bit.ly/duckdb-blog-holistic-aggregates and bit.ly/duckdb-blog-window
[6]DuckDB blog entry: bit.ly/duckdb-blog-rangejoin
[7]DSDSD (Dutch Seminar on Data Systems Design) lecture: bit.ly/duckdb-dsdsd-push

DuckDB first became popular among data-engineers and -scientists: users of R and Python, for whom DuckDB became a powerful tool to manipulate DataFrames in SQL, and manage data in Parquet files. In aiming to address a wide audience, DuckDB has emphasized social media, HackerNews and the DuckDB news blog over articles in scientific venues, to describe many of its innovations.

User-Friendly Features. While DuckDB has good performance, this is not its most important focus: the project puts a lot of attention on reliability and testing[8], and broad functionality, e.g.:

- "friendly" SQL language extensions such as `GROUP BY ALL`, `SELECT * EXCEPT col`, omitting `SELECT` with the `SELECT *` semantics, and placing `FROM` first in a query: `FROM .. SELECT ..`[9]
- rich timezone support and SQL macros.[10]
- reading and writing Parquet files[11] + Iceberg support.
- automatically inferring (nested) tables from JSON on read.[12]
- geographical data processing (PostGEESE[13]).
- vectorized Python UDFs, and full-text-search.[14]
- "zero copy" integration with dplyr, numpy, pandas and Arrow[15]: one can query these with SQL, and materialize such dataframes from a DuckDB query result without data copying.

Extension Modules. DuckDB can get new functionality via *extension* modules. The system can be extended on a number of dimensions: data types, operators, optimizer rules, and even the parser. If a query causes a syntax error, DuckDB will try to parse the query using the alternative parsers supplied by extension modules. Another extensibility dimension is to allow using new *catalogs* supplied by an extension module. An example is the PostgreSQL catalog that allows to connect to such a database and read its tables; so one can pose queries over a mix of PostgreSQL and DuckDB tables.[16]

To use a DuckDB extension, one first needs to install it on the client's file system. This is done using the `INSTALL path;` command in its SQL. The `path` can also be a URL, and DuckDB Labs hosts an S3 bucket, that serves the standard extensions it provides. Functionalities like GIS and full-text search are typically not bundled in the downloadable DuckDB binaries, but can be installed in this way. The next step is then to `LOAD` the extension, which means that it gets loaded into DuckDB as a dynamically loadable library, and from then on the new functionality is available. To prevent malicious or insecure code from being executed, only *signed* extensions are loadable without explicitly overriding security settings.

MotherDuck takes advantage of the DuckDB extension mechanism: connectivity from DuckDB clients to MotherDuck in the cloud using hybrid query processing is provided via a signed (trusted) extension. This extension adds new operators, e.g., bridge operators for sending tuple streams between cloud and client. It introduces optimizer rules to do - among others - hybrid query planning. It also introduces a remote catalog that gives local DuckDB users access to MotherDuck databases in the cloud.

[8]DBTest2022 keynote: bit.ly/duckdb-dbtest-keynote
[9]DuckDB blog entry bit.ly/duckdb-blog-friendly-sql
[10]DuckDB blog entry: bit.ly/duckdb-blog-time
[11]DuckDB blog entry: bit.ly/duckdb-blog-parquet
[12]DuckDB blog entry: bit.ly/duckdb-blog-json
[13]DuckDB blog entry: bit.ly/duckdb-blog-gis
[14]DuckDB blog entries: bit.ly/duckdb-blog-udf and bit.ly/duckdb-blog-fulltext
[15]DuckDB blog entries: bit.ly/duckdb-blog-pandas and bit.ly/duckdb-blog-arrow
[16]DuckDB blog entry: bit.ly/duckdb-blog-postgres

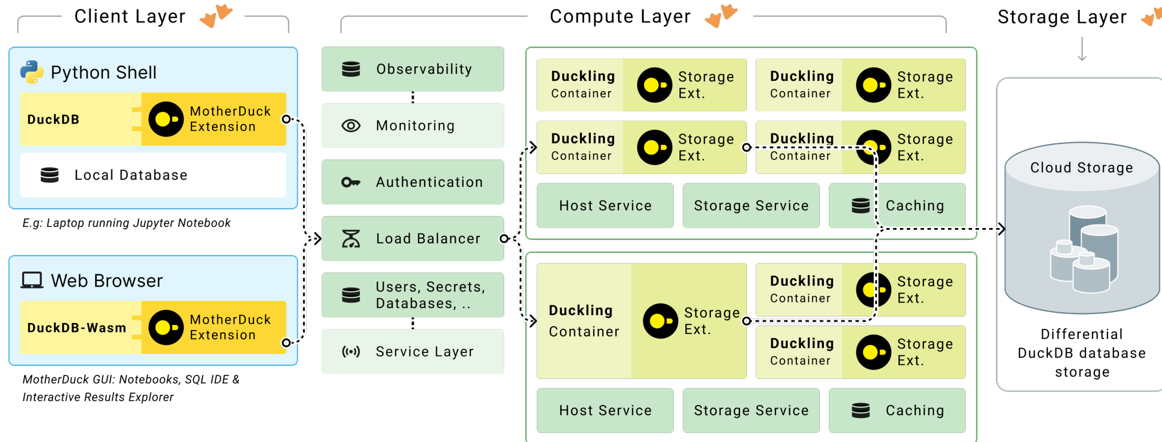

Figure 1: MotherDuck clients always have a local DuckDB, even in web-apps where DuckDB runs as Web Assembly (wasm) embedded in a HTML page. The cloud compute layer runs the remote (parts of) queries of each user on a DuckDB in a variable-sized container called "duckling". The cloud storage layer is separate from compute, and stores DuckDB data in object storage.

3 MOTHERDUCK ARCHITECTURE

3.1 Infrastructure

MotherDuck is a SaaS offering, which means it runs a control plane with components that are responsible for many administrative - yet crucial - aspects to manage data. A non-comprehensive list includes - identity management, catalog metadata (databases names, snapshots, ownership, lineage, etc.), network load balancers, certificate management, observability and alerting. In the following, though, we detail two core aspects: compute and storage.

Compute. MotherDuck's compute platform is built on top of short-lived, on-demand, allocated containers. Each container runs a single DuckDB instance (which we call a "duckling") that is used to serve requests for a single user. The containers run in a VM that may have a local SSD, used for caching by the MotherDuck storage service. Using containers gives us a simple way to put isolation boundaries between users while enjoying most benefits of shared-everything multi-tenant platforms. During their lifetime, we can change the amount of RAM and CPU resources allocated to a container. These changes must then also be reflected in DuckDB: it has been made possible to adjust the size of its buffer manager over time, and its morsel-driven scheduling allows dynamic changes to the amount of cores used for query processing.

Being the first embedded analytical database system, an ongoing direction of innovation in DuckDB is the functionality it offers to the embedding application to express resource constraints that DuckDB will abide to: embedding applications typically have limited resources available for their data analytics, and these resources can also change over time. We expect this feature area of open-source DuckDB to further develop in the future, and MotherDuck can then exploit any such new features in its container resource management.

MotherDuck shuts down containers completely, when they are not used. There are multiple challenges for this, one of them being workload management that accurately predicts the required resources in the future and performs up- or down-scaling of a container accordingly. Scaling up down of containers leads also to technical challenges in terms of conserving state (addressed e.g. with hibernation) and re-arranging containers over machines (addressed e.g. with migration).

Storage. Our storage layer is built on on a shared, scale-out, distributed storage fabric, as offered by cloud providers. This is the most economical way to achieve the durability expectations that customers have of cloud infrastructure, and enable rapid failover of instances. MotherDuck's storage service provides for client data isolation, and improves performance by (i) leveraging local SSD resources for caching and (ii) adapting DuckDB database storage to better suit cloud storage systems. One notable aspect is that a service like S3 cannot *modify* files, so data changes are done by writing (multiple) new files and sometimes deleting files. Whereas DuckDB's native data format stores compressed columnar data in a single file, the duckling storage extension allows for *differential* storage, where changed data is stored independently as a mutation tree. This enables zero-copy duplication, sharing, branching and time travel. The DuckDB storage extension thus seamlessly maps byte-ranges of DuckDB's single-file storage onto multiple files, which are then further mapped onto storage resources, such as cloud blob storage and ephemeral SSD instance storage.

DuckDB provides multi-database storage in the sense that a single DuckDB can `ATTACH` to multiple database files at the same time, in either read-only or read-write mode. In the latter case, the access must be exclusive, whereas in the read-only case multiple DuckDB processes could attach to the same database file. MotherDuck exploits this multi-database concept, both on the duckling as on the client. On the client, it allows to attach to remote (cloud) databases; in addition to any local databases that the user may have. On the cloud side, ducklings are attaching in exclusive mode to a database owned by that MotherDuck user in read-write mode. However, it also introduces the `SHARE` feature in which multiple ducklings may attach to the same MotherDuck database *snapshot* in read-only mode. The concept of a snapshot is powered by the previously mentioned differential storage: by not taking into account newer

differential files, a database appears to be frozen in time. As such, MotherDuck allows multiple users (and thus duckling containers) to have read-only access to a database, created and modified by another user. MotherDuck further introduces a mechanism in which the users of a database `SHARE` can be on the *latest* snapshot, such that they get notified and refreshed periodically when the owner of the database publishes updates. Database `SHARE`s are identified by a URI that users can communicate to each other, e.g. by email or chat, to ease collaboration.

3.2 Hybrid Query Processing

A DuckDB query goes through four phases: parsing, binding (catalog lookup and type checking), query optimization, and execution[3]. The MotherDuck client extension module (depicted in the blue area of Figure 1) hooks into all four phases. It enriches the SQL dialect supported, performs remote binding for (e.g., CSV) files in the cloud, augments the optimizer to plan hybrid query processing, and extends execution to run both locally and in the cloud.

Order-aware Bridge Operators. The MotherDuck client extension introduces new pipelined "bridge" operators that download and upload tuple streams between client and duckling. These operators aim to work well in situations where upload and download bandwidth is asymmetrical and handle the possibly different endianness from the client. They maintain the special ability of DuckDB's parallel pipelines to materialize tuples in the sink in the *same order* they were stored in the original pipeline source, a feature expected by dataframe library users. Doing so without expensive sorting is not supported by other relational engines. It requires keeping track of batch order and some buffering of batches before the sink.

In hybrid query plans, a certain fragment may *produce* data, that is *consumed* by another query fragment that runs on another machine (i.e., client and cloud or vice versa). The producer-consumer nature of the processing comes with a well-known buffering issue: if the producer runs faster than the consumer, the amount of intermediate data accumulates, leading to inefficiencies or even resource exhaustion and failures. To prevent that, we introduced flow-control, that pauses the producing pipeline for while, until the consumer has consumed enough data, in which case it can resume the producer. Open-source DuckDB for this purpose enhanced its query scheduling to make pipelines pausable and resumable. This feature not only benefits MotherDuck, but will also enhance future open-source DuckDB features, such as async I/O.

Remote-local Optimizer. The MotherDuck client extension also adds a new rule to the DuckDB optimizer for planning hybrid query processing. After DuckDB has optimized the plan, this rule splits the plan into fragments and designates each fragment to run either locally or remote, inserting bridge operators in between. The optimizer starts by collecting constraints – each leaf node is assigned the locality of the data it accesses. It marks data sources that *generate* data (like `range()` or `repeat()`) as able to run equally well on either side. Also, some pipeline segments must run in the same fragment because they rely on shared state (e.g., in a `RECURSIVE` SQL query, two operators repeatedly add and scan data from a shared hash-table). Honoring such constraints, the optimizer rule then looks for the plan with smallest estimated data transfer cost.

Virtual Catalog. Since query planning is done locally, we must make sure that local DuckDB has access to metadata information about cloud-resident databases. For table schemas and statistics to be available during the binding and the query optimization phases, we use DuckDB's extensible catalog to create and maintain a MotherDuck proxy catalog for all relevant cloud databases. This lightweight *virtual catalog* offers the same functionality as the DuckDB catalogs for planning purposes but does not need to support actual data operations, as those run in the cloud.

SQL Language Extensions. MotherDuck enriches DuckDB's SQL in multiple ways. We changed the binding process of the table functions to read Parquet, JSON and CSV files, which can be used in DuckDB's `FROM` clause, to support an additional parameter `MD_RUN =REMOTE` (or `=LOCAL`) to specify whether the table scan should execute on the duckling or the client.[17] The MotherDuck client extension further registers a parser that adds support for new SQL statements. MotherDuck allows users to `CREATE` and `USE` multiple databases. Typically, MotherDuck users have a local database in their client, and possibly multiple databases in the cloud. The `CREATE SHARE` statement allows to share a cloud database with other MotherDuck users. It returns a URL that another user can use in an `ATTACH <url>` statement. At introduction, only the owner could modify data in these databases, but this functionality is being expanded, including offering time-travel to older `ATTACH`-ed snapshots.

Testing. The DuckDB project has developed a testing suite that includes a fast-to-run but extensive set of SQL unit tests, supplemented with longer running tests that use memory sanitizers, multiple runs of the same queries with different internal settings (like optimizers on/off), background testing of code coverage, performance and various SQL fuzzers[8].

MotherDuck is a distributed system, albeit one that involves only two nodes: client and duckling. Distributed systems testing adds challenges in reliability (e.g., network hiccups), determinism and runtime. For developers to run a suite consistently, it is important that it completes quickly. DuckDB already had a rich set of tests which we wanted to enable. The key idea that allowed us to do so was to link client and duckling in a single process and provide an artificial network that turns RPCs into local function calls.

3.3 User Interface

MotherDuck comes with a built-in web-based user interface to perform analysis with zero user setup. Online SQL UIs have grown beyond simply displaying tabular query results, and now offer features to facilitate the analytical process, including powerful result-set post-processing and AI-powered assistance.

Interactive Result Set Exploration. An analytical task is like an iterative, user driven optimization process to find a suitable SQL query to answer an analytic question. The user will enter a SQL query, examine the query results, and then revise the query to get it closer to answering the original analytic question. The aim of a productive analytical environment is to make the next iterative step in this search process fast and cheap to discover.

The MotherDuck Web UI supports this process by providing the user with a notebook-style user interface [14]. Each cell in our

[17] By default, the remote-local optimizer runs scans of web URLs in the duckling.

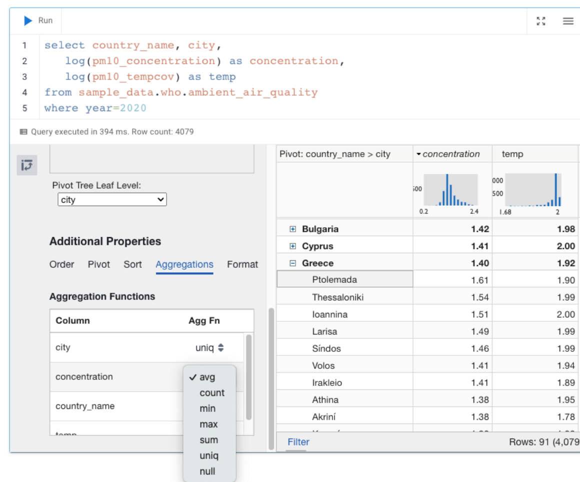

Figure 2: MotherDuck Web GUI showing exploration of WHO air quality data. DuckDB-wasm embedded in web page is leveraged for very low-latency pivoting and re-aggregation of the query result, and for showing result-set distributions, using local in-browser queries that run in a few msec.

SQL notebook provides a split view consisting of a text editor for entering a SQL query and a results pane, as shown in Figure 2. The results pane is a full-featured interactive pivot table for exploring query results, based on the open source Tad viewer.[18] This presents the user with column histograms and allows the user to perform pivot, filter, aggregation, sort, column-selection, column-ordering and formatting operations without writing further SQL. It allows the user to interactively explore a query result, to either validate it or identify opportunities to revise their SQL query.

In order to provide a good user experience for interactive data exploration, it is essential that interactive operations on the results table are extremely fast. In user interfaces, 100 ms is a well understood upper bound on response time for an operation to feel instantaneous.[19] This presents a challenge when developing a UI for a cloud-hosted database, since typical round trip times from a user's laptop to the cloud are 70-100 ms, eating the entire latency budget. This has led some cloud SQL UIs to implement interactive analytic operations for exploring tabular result data in JavaScript.

In the MotherDuck Web UI, in contrast, DuckDB-wasm is running in the browser (Figure 1 left-bottom); it manages the result sets of the SQL notebook in an in-memory DuckDB database.

This provides a powerful, SQL-complete, post-processing engine capable of multi-threading that stores data much more compactly than JavaScript-based alternatives; and thus can cache and process relatively large amounts of data efficiently[20].

Declarative Caching. The implementation of result caching to support the interactive results interface in the MotherDuck GUI depends purely on SQL abstractions and the hybrid query processing model. Conventional database optimization focuses on the time to evaluate individual queries in isolation, or overall throughput across a large workload of many independent queries. Our perspective is that we need to consider database performance in the context of an analytics session and minimize user perceived latency in the UI, rather than only maximize overall system throughput.

When executing a query, the MotherDuck GUI accumulates the results in a local results cache:

```
CREATE TABLE localMemDb.main.cacheTable1 AS
SELECT * FROM ({userQuery})
LIMIT {CLIENT_CACHE_ROW_LIMIT}
```

This in-memory table is managed by DuckDB-wasm running in the user's browser. All subsequent interactive operations on the pivot table are performed by generating SQL queries against it. This implementation of caching to support the interactive results table depends only on MotherDuck's hybrid query processing model. It guarantees that queries that depend only on local (temporary) tables will be executed locally. This caching technique is thus available to any interactive data application built on MotherDuck.

In the future, we would like to support *background streaming* to fill the client cache asynchronously. This would enable us to reduce latency even further by allowing us to render the first page of results before all results are available.

A limitation of our current caching strategy is that the `LIMIT {CLIENT_CACHE_ROW_LIMIT}` clause means we must resort to a fallback strategy if the query result size exceeds our internal cache limit to protect user memory limits. For scrolling through even larger result sets quickly, we plan to use a two stage cache where full tables are materialized temporarily on the server, and scrolling triggers a page fault that pulls a large block of rows, enabling initial random access over much larger result sets.

AI assistant. Since early 2023, it has become clear that Large Language Models (LLMs) have significantly improved the state-of-the-art in handling natural language questions, as well as in the ability to automatically write queries from natural language prompts. From its initial launch, the MotherDuck GUI came with the possibility to formulate notebook queries in natural language. This feature is also available in all other DuckDB APIs via the `PRAGMA prompt_query(<question>)`. In the background, MotherDuck AI will automatically generate an SQL statement, iteratively fix it if needed (based on error messages), and execute it against the current database. Users can also leverage parts of the functionality to generate SQL snippets from natural language questions or to fix a broken query. As the area keeps evolving rapidly, MotherDuck expects AI assistance functionality to significantly increase its role going forward, impacting query understanding, query completion, interactive documentation help and more.

4 DEMO

The Efficiency of Hybrid Query Processing. We will demonstrate the efficiency of hybrid query processing by running queries from the Star Schema Benchmark. Our benchmark setup consists of the fact table (lineorder), stored as Parquet files in S3, with the dimension tables stored on a laptop in a DuckDB database. We then run some of the 13 join-filter-groupby queries that make up SSB, using DuckDB with or without enabling MotherDuck (i.e. fully local vs. hybrid query processing). The results in Table 1 were obtained using an Apple M1 Macbook Air (4 performance and 4 efficiency

[18]https://github.com/antonycourtney/tad
[19]bit.ly/nielsen-norman-response-times
[20]DuckDB-wasm benchmarks: shell.duckdb.org/versus

Q1.1	Q1.2	Q1.3	Q2.1	Q2.2	Q2.3	Q3.1	Q3.2	Q3.3	Q3.4	Q4.1	Q4.2	Q4.3	GB
6.5	6.6	6.7	6.7	6.7	6.8	6.1	7.0	7.4	6.2	7.6	7.7	10.2	1
91.1	90.8	89.5	96.4	97.4	91.6	85.4	91.4	90.0	106.6	116.6	109.2	93.2	10
944.8	920.5	917.0	999.2	975.8	975.2	940.8	935.9	927.7	935.6	1116.2	1094.8	1087.9	100
fully local DuckDB 0.8.1 on Macbook laptop													
1.5	1.3	1.3	1.2	1.2	1.4	1.3	1.6	1.3	1.1	1.5	1.6	1.1	1
10.1	10.7	10.3	9.0	9.2	10.1	8.6	9.0	8.4	10.1	13.8	10.2	9.2	10
117.6	107.9	105.5	101.2	95.3	173.8	111.8	129.7	194.9	89.3	110.7	116.0	106.2	100
hybrid query processing in MotherDuck (laptop+cloud)													

Table 1: Query runtimes in seconds on the Star Schema Benchmark (SSB) using a MacBook, comparing local DuckDB vs. hybrid query processing in MotherDuck (with a graviton2 duckling in AWS). The lineorder fact table is stored in Parquet on S3; the dimension tables are local in DuckDB. Scanning the Parquet files in the cloud and running the join there is faster (and cheaper) than transferring the data to the laptop.

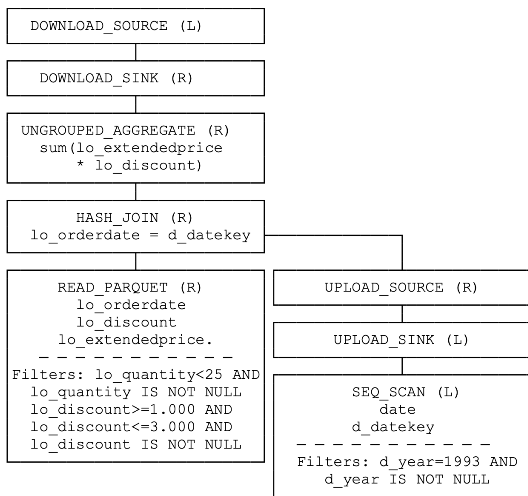

Figure 3: MotherDuck EXPLAIN output on SSB Q1.1: a bridge operator uploads the result of a dimension table-scan (date) with selection pushdown from the local (L) DuckDB to the remote duckling (R) - which runs the other operators.

cores) with 16GB RAM, whereas the duckling (running in AWS us-east) is a 16-core m6gd.4xlarge VM with 16 graviton2 cores and 64GB of RAM; though the duckling container was configured to use 8 cores and 12GB only. The Hybrid Query Plans (Q1.1 shown in Figure 3) perform the table-scan (and filter) on the dimension tables locally, then use a bridge operator to upload the data to duckling, where it is joined with the fact table and aggregated. The large differences (~5x faster on scale-factor 1; ~10x faster on scales 10 & 100) are partly thanks to the duckling having 2x more cores; but mostly due to the higher bandwidth to S3. The AWS price of all-local processing is also higher because of egress cost: on scale-factor 100, 90GB is transferred in total, which costs $8).

MotherDuck GUI. We will open the MotherDuck GUI and demonstrate the very-low-latency postprocessing on query results it offers, encompassing pivoting column distributions, sorting and filtering.

In addition we will demonstrate the AI-enhanced features, e.g., posing certain queries in English rather than in SQL.

Sharing Databases in the Cloud for Collaboration. We will work through a scenario where one data scientist completes an analysis on her MotherDuck database and then decides to share this data with a co-worker. When receiving a link to the database, they can use this link gain instant access to this data and receive updates what the data changes.

5 RESEARCH PERSPECTIVES

MotherDuck's concept of hybrid query processing differs from distributed query processing and mobile query processing. *Mobile* databases [9] typically talk to cloud middleware using API calls; whereas in MotherDuck client and server are united in one declarative SQL system. In a *distributed* database [5] usually many similar nodes collaborate, typically organized in cloud-based cluster; whereas in MotherDuck there are just two nodes, with typically different hardware and operating systems, connected with often *asymmetrical* wide area network performance. The heterogeneous and asymmetrical aspects cause new distributed query optimization challenges. While the current architecture of hybrid query processing is two-node; one could imagine running ducklings in multiple cloud regions; if user data is spread over cloud regions; and then optimize for hybrid query processing close to the data. Taking this further, multi-node hybrid query processing could also span edge-devices or combine on-premise resources with cloud resources; but query planning in such settings is hardly explored [31].

These optimization challenges are not only in the speed efficiency space, but also in the cost space as cloud storage, cloud compute, and egress cost give a price-tag to data placement and query execution plans, that can be optimized for [26].

It may not be possible with good query optimization alone to load-balance client and duckling; but this could be achieved by elastically changing the duckling resources, as queries are planned. Dynamic resource allocation for ducklings raises research questions in scheduling containers of many users and their possible migration.

Hybrid query processing in MotherDuck is related to previous research in client-server processing. Very early work [18] studied how to partition the various software layers of a DBMS (storage, I/O, buffer management, query optimization and processing) between client and server. Rather than partitioning this functionality, MotherDuck *replicates* it on both client and server, providing maximum flexibility where to store, access and process data. The idea to do this was considered in later research into client-server architectures as one of the alternatives under the names "hybrid shipping" [16] and "enhanced client server" [6], respectively. While both works argued for hybrid query processing as the most flexible and best performing architecture in their model-based simulations, it was not implemented in an actual system. It should also be noted that hybrid query processing provides opportunities investigated in [20] when it is combined with table caching: local-remote caching decisions can start to interact with local-remote query optimization, because caching actions change the query optimization space.

MotherDuck does not go here yet. Since optimal automatic data placement requires advance use of a workload, which often is not available; and since automatic data placement will affect query performance in ways that could surprise users, the philosophy of MotherDuck is to give users SQL tools to allow them to influence data placement: e.g., by storing data in databases that are either local- or cloud-based. As mentioned in the declarative caching

section, we also use SQL syntax to specify placement of query results, starting simply with temp-tables on client- and server-side. Designing more advanced caching functionality between client is therefore another research challenge for MotherDuck. This differs from earlier client-server caching research [11] as (i) middleware is not needed in hybrid processing, (ii) queries can mix cached and base data from both client and duckling and (iii) we aim for *declarative* SQL-level extensions. In the longer term, MotherDuck caching could encompass finer-grained methods [28] that e.g. reuse the intermediate results during plan execution across similar queries.

The software engineering challenges include formal protocol analysis, as well as semantics-preserving transformations to serialized query plans that enable evolutionary change of database components. MotherDuck runs DuckDB as-a-service in the cloud, where fast iteration through frequent releases are common practice. However, with hybrid query processing, MotherDuck combines cloud compute resources with a locally installed DuckDB client that does not upgrade as fast. We partially address this difference by auto-upgrading the MotherDuck extension module to the latest duckling version; but many functionality changes could still break compatibility and present challenges in developing new features.

Large Language Models (LLMs) could help data engineering tasks by venturing into features like AI-assisted data augmentation, cleaning, and classification. However, LLMs are resource-intensive and need to be hosted in the cloud. There is a need for smaller use-case-specific language models (e.g., for SQL completion, SQL fixing) that are open-source and possibly even suited for client-side inference. Given limited storage on the client, one could envision sharing a small LLM between multiple applications on the client, possibly with different fine-tunings for different applications. This would make AI query assistants more responsive while contributing to carbon footprint reduction and better privacy preservation.

6 CONCLUSION

We introduced and motivated MotherDuck: it connects the popular, lightweight and user-friendly open-source analytics database DuckDB to the cloud. It does so in a unique way, namely by ensuring that the client that connects to it always has a local DuckDB running locally – even for web-apps, by running DuckDB-wasm (Web Assembly) in the browser. The resulting hybrid query processing model has many advantages and opens up new research challenges and application possibilities. MotherDuck, building on DuckDB, intends to empower users to use more client-side resources, while simplifying cloud data system architecture by using a scale-up approach. We truly believe that this can reduce the amount of cloud hardware needed for analytics applications for those 95% of users that are now burdened with hard-to-manage scale-out architectures, that are both clunky and expensive to use.

At the time of writing, MotherDuck is one year since inception and has just beta-launched; so there is still a lot to be built and to be decided (for instance, the pricing model). We already see a lot of potential and use cases, from new ultra-interactive data apps in the browser, to user-friendly collaborative data engineering, with cost-effective usage of both local and cloud compute for those activities where either excel.

REFERENCES

[1] Daniel Abadi, Peter A. Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. 2013. The design and implementation of modern column-oriented database systems. *Found. Trends Databases*, 5, 3.
[2] Viktor Leis Alex van Renen. 2023. Cloud analytics benchmark. *PVLDB*, 16, 6.
[3] Peter A. Boncz, Thomas Neumann, and Viktor Leis. 2020. FSST: fast random access string compression. *PVLDB*, 13, 11.
[4] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. Monetdb/x100: hyper-pipelining query execution. In *CIDR*.
[5] Benoıt Dageville, Thierry Cruanes, and Marcin Zukowski et al. 2016. The snowflake elastic data warehouse. In *SIGMOD*.
[6] Alex Delis and Nick Roussopoulos. 1993. Performance comparison of three modern DBMS architectures. *IEEE TKDE*, 19, 2, 120–138.
[7] Tom Ebergen. 2022. *Join Order Optimization with (Almost) No Statistics*. MA thesis. Vrije Universiteit Amsterdam. www.cwi.nl/ boncz/msc/2022-TomEbergen.pdf.
[8] Andreas Rossberg et al. 2018. Bringing the web up to speed with WebAssembly. *CACM*, 61, 12.
[9] Guy Bernard et al. 2004. Mobile databases: a selection of open issues and research directions. *SIGMOD Record*, 33, 2.
[10] Kai Ren et al. 2012. Hadoop's adolescence; A comparative workloads analysis from three research clusters. In *SuperComputing*.
[11] Mehmet Altinel et al. 2003. Cache tables: paving the way for an adaptive database cache. In *PVLDB*.
[12] Nicolas Anciaux et al. 2014. Milo-db: a personal, secure and portable database machine. *Distributed Parallel Databases*, 32, 1.
[13] Nikos Armenatzoglou et al. 2022. Amazon redshift re-invented. In *SIGMOD*.
[14] Thomas Kluyver et al. 2016. Jupyter notebooks — a publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas*. IOS Press.
[15] Udit Gupta et al. 2022. Chasing carbon: the elusive environmental footprint of computing. *IEEE Micro*, 42, 4.
[16] Michael J. Franklin, Björn Þór Jónsson, and Donald Kossmann. 1996. Performance tradeoffs for client-server query processing. In *SIGMOD*. H. V. Jagadish and Inderpal Singh Mumick, editors, 149–160.
[17] Michael J. Freitag and Thomas Neumann. 2019. Every row counts: combining sketches and sampling for accurate group-by result estimates. In *CIDR*.
[18] Robert B. Hagmann and Domenico Ferrari. 1986. Performance analysis of several back-end database architectures. *ACM TODS*, 11, 1, 1–26.
[19] André Kohn, Dominik Moritz, Mark Raasveldt, and Hannes Mühleisen et al. 2022. DuckDB-wasm: fast analytical processing for the web. *PVLDB*, 15, 12.
[20] Donald Kossmann, Michael J. Franklin, and Gerhard Drasch. 2000. Cache investment: integrating query optimization and distributed data placement. *ACM TODS*, 25, 4, 517–558.
[21] Laurens Kuiper and Hannes Mühleisen. 2023. These rows are made for sorting and that's just what we'll do. In *ICDE*.
[22] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a numa-aware query evaluation framework for the many-core age. In *SIGMOD*.
[23] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. Leanstore: in-memory data management beyond main memory. In *ICDE*.
[24] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: artful indexing for main-memory databases. In *ICDE*.
[25] Viktor Leis, Kan Kundhikanjana, Alfons Kemper, and Thomas Neumann. 2015. Efficient processing of window functions in SQL queries. *PVLDB*, 8, 10.
[26] Viktor Leis and Maximilian Kuschewski. 2021. Towards cost-optimal query processing in the cloud. *PVLDB*, 14, 9.
[27] Frank McSherry, Michael Isard, and Derek Gordon Murray. 2015. Scalability! But at what COST? In *HotOS*.
[28] Fabian Nagel, Peter A. Boncz, and Stratis Viglas. 2013. Recycling in pipelined query evaluation. In *ICDE*.
[29] Thomas Neumann, Viktor Leis, and Alfons Kemper. 2017. The complete story of joins (in hyper). In *BTW*. Vol. P-265.
[30] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast serializable mvcc for main-memory database systems. In *SIGMOD*.
[31] Vassilis Papadimos and David Maier. 2002. Mutant query plans. *Inf. Softw. Technol.*, 44, 4, 197–206.
[32] Mark Raasveldt and Hannes Mühleisen. 2020. Data management for data science - towards embedded analytics. In *CIDR*.
[33] Stephen Tu, M. Frans Kaashoek, Samuel Madden, and Nickolai Zeldovich. 2013. Processing analytical queries over encrypted data. *PVLDB*, 6, 5.
[34] Matei Zaharia. 2019. Lessons from large-scale software as a service at databricks. In *SOCC*.
[35] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter A. Boncz. 2006. Super-scalar RAM-CPU cache compression. In *ICDE*.