Lecture #20: System Analysis (DuckDB)

15-721 Advanced Database Systems (Spring 2024)
https://15721.courses.cs.cmu.edu/spring2024/
Carnegie Mellon University
Prof. Andy Pavlo

1 Introduction to DuckDB

DuckDB [2] is an advanced, multi-threaded embedded database management system (DBMS) that is designed to execute SQL over disparate data files. It is termed as "SQLite for Analytics" due to its PostgreSQL-like dialect and embedded nature. DuckDB provides zero-copy access to query results via Apache Arrow to client code running in the same process. It is built with nearly all custom C++ code and has little to no third-party dependencies, relying on an extension ecosystem to expand its capabilities. Additionally, DuckDB supports DataFrame libraries to query databases without using SQL, such as dpylr for R-lang and Ibis for Python. It also can access external data files via extensions, including Parquet, Arrow, SQLite, JSON, and can install extensions to retrieve files from remote filesystems (HTTP, S3).

2 Design

The following section expands upon some of the design choices made by DuckDB.

2.1 High-Level Overview

DuckDB Design Decisions:

1. It uses a shared-everything architecture (Notion of compute and storage are not separated).
2. It implements push-based vectorized query processing.
3. It implements Multi-Version Concurrency Control inspired from Hyper.
4. It adopts precompiled primitives.
5. It uses Morsel-driven Parallelism.
6. It supports PAX Columnar Storage.
7. It supports both Sort-Merge Join and Hash Join operations.
8. It has a stratified Query Optimizer and supports unnesting of arbitrary subqueries.

2.2 Push-Based Processing

Push-based processing is a query processing model where each operator in the system determines whether it will execute in parallel on its own, instead of relying on a centralized executor. This model was adopted by DuckDB in 2021, after finding it challenging in the original pull-based vectorized query processing to add additional operators. The push-based processing model allows for more flexibility and efficiency in executing multiple pipelines simultaneously. This switch to push-based processing has improved the system's ability to handle complex parallelism and execute operations in a more distributed and efficient manner.

This opens the door for additional optimizations and more fine-grained control of the system. It mentions the use of a Vector Cache to buffer results between operators until it fills the vector. Additionally, Scan Sharing involves pushing results from one child operator to multiple parent operators in the Directed Acyclic Graph (DAG) plan. Storing state in a central location also enables Backpressure/Async IO, allowing pause operator execution when buffers are full or when waiting for remote I/O. This fine-grained control allows for more efficient and optimized query processing within the database system.

2.3 Vectors

DuckDB uses a unified format to process all vector types without needing to decompress them first. This approach reduces the number of specialized primitives per vector type. Additionally, DuckDB has a custom internal vector layout for intermediate results that is compatible with Velox and supports multiple vector types. These features contribute to the efficient and flexible handling of vectors in DuckDB.

The multiple vector types supported:

- Flat Vectors: physically stored as an uncompressed contiguous array.
- Constant Vectors: physically stored as a single constant value.
- Dictionary Vectors: physically stored as a dictionary vector indexed by a selection vector.
- Sequence Vectors: physically stored as a base vector and increment offset vector.

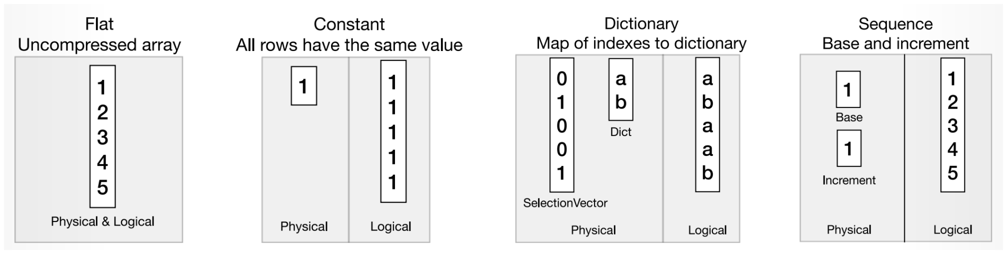

Figure 1: Vector Types Supported

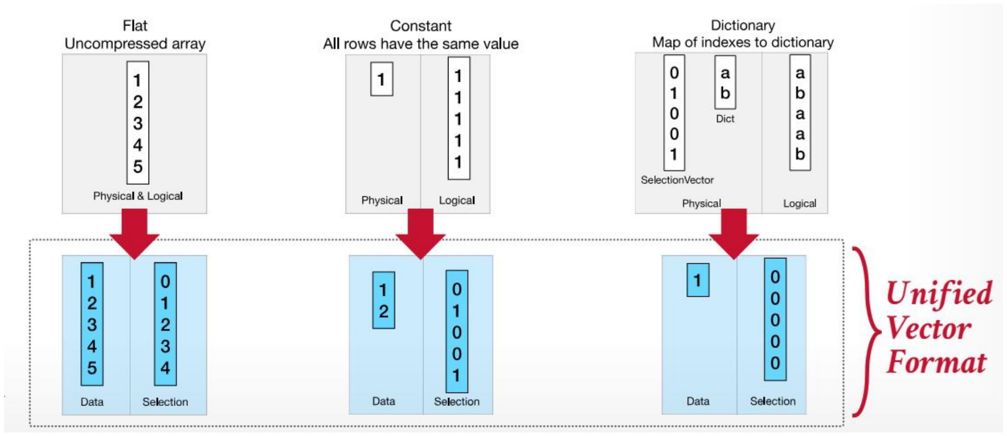

Figure 2: Unified Vector Processing Format

2.4 DataFrames

DuckDB supports DataFrame libraries to query databases without using SQL, specifically dpylr for R-lang and Ibis for Python. Integration libraries generate DuckDB logical plans that the DBMS converts into optimized physical plans, bypassing the SQL parser. This allows for seamless integration and efficient querying using DataFrame libraries with DuckDB.

2.5 Storage Format

DuckDB uses a built-in storage format that maintains a single PAX-oriented file per database. This format splits tables into row groups with 120k tuples, and the on-disk encoding is different than in-memory. Additionally, the DBMS can access external data files via extensions, including Parquet, Arrow, SQLite, and JSON. It also allows for the installation of extensions to retrieve files from remote filesystems such as HTTP and S3.

3 MotherDuck: Extending DuckDB into the Cloud

MotherDuck introduces new capabilities to DuckDB, offering DuckDB data storage and serverless query processing in the cloud [1]. It is provided as an extension to the existing DuckDb library while using the same interface. Unlike highly scalable systems such as Snowflake or Dremel, MotherDuck focuses on the efficient processing of DuckDB queries within serverless environments. It provides automatic execution of DuckDB queries on serverless compute nodes, where remote nodes are DuckDB instances running inside containers and connected to object stores.

3.1 Architecture

In the MotherDuck Architecture, the MotherDuck clients always run on a local DuckDB instance, on a python shell or in web-apps where DuckDB runs as Web Assembly (wasm) embedded in an HTML page. Remote computation occurs on a "duckling", a variable-sized container within virtual machines in the cloud, separated from the cloud storage layer that is connected to object storage [1].

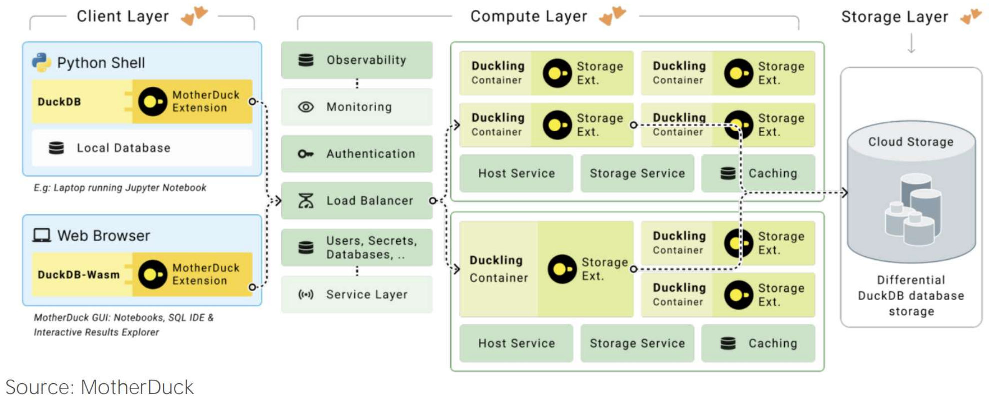

Figure 3: Overview of MotherDuck Architecture

3.2 Hybrid Query Processing

The MotherDuck optimizer exploits data locality during the query planning. If some data is local and some remote, then part of the query will be executed locally, and part remotely, with introducing a new "bridge" operators that passes tuple streams between local and remote DuckDB instances and leverages the operator pausing feature that DuckDB added from switching to push-based execution. It will do query optimization on the local instance as normal and then uses cost-based rules to decide what to run locally vs. remote. To ensure local DuckDB has access to remote metadata during binding and query optimization phases, MotherDuck maintains and exposes remote catalogs to the local instance.

4 Conclusion

In conclusion, DuckDB is a perfect illustration of how a brilliant idea paired with top-notch engineering effort can create a successful product. As they stated in the paper, while none of DuckDB's components is revolutionary in its regard, the use of a combination of methods and algorithms from the state of the art was best suited for their use cases, making it a solid choice in its field.

References

[1] R. Atwal, P. Boncz, R. Boyd, A. Courtney, T. Döhmen, F. Gerlinghoff, J. Huang, J. Hwang, R. Hyde, E. Felder, J. Lacouture, Y. LeMaout, B. Leskes, Y. Liu, D. Perkins, T. Tereshko, J. Tigani, N. Ursa, S. Wang, and Y. Welsch. MotherDuck: DuckDB in the cloud and in the client. 2024.

[2] M. Raasveldt and H. Mühleisen. DuckDB: an Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1981–1984, Amsterdam Netherlands, June 2019. ACM. ISBN 978-1-4503-5643-5. doi: 10.1145/3299869.3320212. URL `https://dl.acm.org/doi/10.1145/3299869.3320212`.