DuckDB - the SQLite for Analytics

Mark Raasveldt
Centrum Wiskunde & Informatica
Amsterdam, The Netherlands
m.raasveldt@cwi.nl

Hannes Mühleisen
Centrum Wiskunde & Informatica
Amsterdam, The Netherlands
hannes@cwi.nl

API	C/C++/SQLite
SQL Parser	`libpg_query`
Execution Engine	Vectorized
Concurrency Control	Snapshot MVCC
Storage	Column-Store, Single-File
Optimizer	Cost-Based

Table 1: DuckDB: Component Overview

Data management systems have evolved into large monolithic database servers running as stand-alone processes. This is partly a result of the need to serve requests from many clients simultaneously and partly due to data integrity requirements. While powerful, stand-alone systems require considerable effort to set up properly and data access is constricted by their client protocols. There exists a completely separate use case for data management systems, those that are *embedded* into other processes where the database system is a linked library that runs completely within a "host" process. The most well-known representative of this group is SQLite, the most widely deployed SQL database engine with more than a trillion databases in active use. SQLite strongly focuses on transactional (OLTP) workloads, and contains a row-major execution engine operating on a B-Tree storage format. As a consequence, SQLite's performance on analytical (OLAP) workloads is very poor.

There is a clear need for embeddable analytical data management. This needs stems from two main sources: Interactive data analysis and "edge" computing. Interactive data analysis is performed using tools such as R or Python. The basic data management operators available in these environments through extensions (dplyr, Pandas, etc.) closely resemble stacked relational operators, much like in SQL queries, but lack full-query optimization and transactional storage. Embedded analytical data management is also desirable for edge computing scenarios. For example, connected power meters currently forward data to a central location for analysis. This is problematic due to bandwidth limitations especially on radio interfaces, and also raises privacy concerns. An embeddable analytical database is very well-equipped to support this use case, with data analyzed on the edge node. The two use cases of interactive analysis and edge computing appear orthogonal. But surprisingly, the different use cases yield similar requirements.

In this talk, we present the capabilities of our new system, *DuckDB*. DuckDB is a new purpose-built embeddable relational database management system created at the Database Architectures group of the CWI. DuckDB is available as Open-Source software under the permissive MIT license[1]. To the best of our knowledge, there currently exists no purpose-built embeddable analytical database despite the clear need outlined above. DuckDB is no research prototype but built to be widely used, with millions of test queries run on each commit to ensure correct operation and completeness of the SQL interface.

DuckDB is built from the ground up with analytical query processing in mind. As storage, DuckDB uses a single-file format with tables partitioned into columnar segments. Data is loaded into memory using a traditional buffer manager, however, the blocks that are loaded are significantly larger than that of a traditional OLTP system to allow for efficient random seeks of blocks. Queries are processed using a vectorized query processing engine similar to the one used in Vectorwise to allow for high performance batch processing and SIMD optimizations.

DuckDB's optimizer performs join order optimization using dynamic programming with a greedy fallback for complex join graphs. It performs flattening of arbitrary subqueries using the unnesting rules described in Neumann et al. In addition, there are a set of rewrite rules that simplify the expression tree, by performing e.g. common subexpression elimination and constant folding.

DuckDB uses optimistic concurrency control. When two transactions attempt to update the same value, the second transaction will fail instead of locking the row and waiting until the first transaction completes. DuckDB does, however, provide very fine-grained concurrency control. Transactions will only conflict if the exact same *value* is updated. Different rows and different columns can be updated concurrently without any issues.

DuckDB uses MVCC to provide snapshot isolation of individual transactions. However, unlike traditional MVCC implementations the one used by DuckDB is optimized entirely for batch appends, updates and deletions, allowing for efficient updates and deletions to large subsets of the data.

[1] `https://github.com/cwida/duckdb`