DuckDB: an Embeddable Analytical Database

Mark Raasveldt
m.raasveldt@cwi.nl
CWI, Amsterdam

Hannes Mühleisen
hannes@cwi.nl
CWI, Amsterdam

ABSTRACT

The immense popularity of SQLite shows that there is a need for unobtrusive in-process data management solutions. However, there is no such system yet geared towards analytical workloads. We demonstrate DuckDB, a novel data management system designed to execute analytical SQL queries while embedded in another process. In our demonstration, we pit DuckDB against other data management solutions to showcase its performance in the embedded analytics scenario. DuckDB is available as Open Source software under a permissive license.

ACM Reference Format:
Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *2019 International Conference on Management of Data (SIGMOD '19), June 30-July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3299869.3320212

1 INTRODUCTION

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 [12]. 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 [4]. SQLite strongly focuses on transactional (OLTP) workloads, and contains a row-major execution engine operating on a B-Tree storage format [3]. As a consequence, SQLite's performance on analytical (OLAP) workloads is very poor.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
SIGMOD '19, June 30-July 5, 2019, Amsterdam, Netherlands
© 2019 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-5643-5/19/06.
https://doi.org/10.1145/3299869.3320212

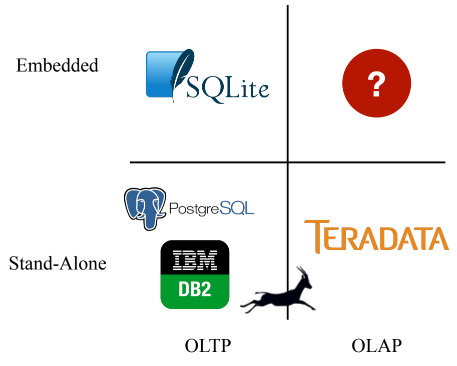

Figure 1: Systems Landscape

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 [14], Pandas [6], 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. For example, in both use cases portability and resource requirements are critical, and systems that are careful with both will do well in both usage scenarios.

In our previous research, we have developed MonetDBLite, an embedded analytical system that was derived from the MonetDB system [13]. MonetDBLite proved successfully that there is a real interest in embedded analytics, it enjoys thousands of downloads per month and is used all around the world from the Dutch central bank to the New Zealand police. However, its success also uncovered several issues that proved very complex to address in a non-purpose-built system. The following requirements for embedded analytical databases were identified:

- High efficiency for OLAP workloads, but without completely sacrificing OLTP performance. For example, concurrent data modification is a common use case in dashboard-scenarios where multiple threads update the data using OLTP queries and other threads run the OLAP queries that drive visualizations simultaneously.
- Efficient transfer of tables to and from the database is essential. Since both database and application run in the same process and thus address space, there is a unique opportunity for efficient data sharing which needs to be exploited.
- High degree of stability, if the embedded database crashes, for example due to an out-of-memory situation, it takes the host down with it. This can never happen. Queries need to be able to be aborted cleanly if they run out of resources, and the system needs to gracefully adapt to resource contention.
- Practical "embeddability" and portability, the database needs to run in whatever environment the host does. Dependencies on external libraries (e.g. `openssh`) for either compile- or runtime have been found to be problematic. Signal handling, calls to `exit()` and modification of singular process state (locale, working directory etc.) are forbidden.

In this demonstration, we present the capabilities of our new system, *DuckDB*. DuckDB is a new purpose-built embeddable relational database management system. 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. Our first-ever demonstration of DuckDB will pit it against other systems on a small device. We will allow viewers to increase the size of the dataset processed, and observe various metrics such as CPU load and memory pressure as the dataset size changes. This will demonstrate the performance of DuckDB for analytical embedded data analysis.

2 DESIGN AND IMPLEMENTATION

DuckDB's design decisions are informed by its intended use case: embedded analytics. Overall, we follow the "textbook" separation of components: Parser, logical planner, optimizer, physical planner, execution engine. Orthogonal components are the transaction and storage managers. While DuckDB is first in a new class of data management systems, none of DuckDB's components is revolutionary in its own regard. Instead, we combined methods and algorithms from the state of the art that were best suited for our use cases.

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

API	C/C++/SQLite	
SQL Parser	`libpg_query`	[2]
Optimizer	Cost-Based	[7, 9]
Execution Engine	Vectorized	[1]
Concurrency Control	Serializable MVCC	[10]
Storage	DataBlocks	[5]

Table 1: DuckDB: Component Overview

Being an embedded database, DuckDB does not have a client protocol interface or a server process, but instead is accessed using a C/C++ API. In addition, DuckDB provides a SQLite compatibility layer, allowing applications that previously used SQLite to use DuckDB through re-linking or library overloading. As with MonetDBLite, we have also implemented the database APIs for R (DBI) and Python (PEP 249).

The *SQL parser* is derived from Postgres' SQL parser that has been stripped down as much as possible [2]. This has the advantage of providing DuckDB with a full-featured and stable parser to handle one of the most volatile form of its input, SQL queries. The parser takes a SQL query string as input and returns a parse tree of C structures. This parse tree is then immediately transformed into our own parse tree of C++ classes to limit the reach of Postgres' data structures. This parse tree consists of statements (e.g. `SELECT`, `INSERT` etc.) and expressions (e.g. `SUM(a)+1`).

The *logical planner* consists of two parts, the binder and the plan generator. The binder resolves all expressions referring to schema objects such as tables or views with their column names and types. The logical plan generator then transforms the parse tree into a tree of basic logical query operators such as scan, filter, project, etc. After the planning phase, we have a fully type-resolved logical query plan. DuckDB keeps statistics on the stored data, and these are propagated through the different expression trees as part of the planning process. These statistics are used in the optimizer itself, and are also used for integer overflow prevention by upgrading types when required.

DuckDB's *optimizer* performs join order optimization using dynamic programming [7] with a greedy fallback for complex join graphs [11]. It performs flattening of arbitrary subqueries as described in Neumann et al. [9]. In addition, there are a set of rewrite rules that simplify the expression tree, by performing e.g. common subexpression elimination and constant folding. The result of this process is the optimized logical plan for the query. The *physical planner* transforms the logical plan into the physical plan, selecting suitable implementations where applicable. For example, a scan may decide to use an existing index instead of scanning the base tables based on selectivity estimates, or switch between a hash join or merge join depending on the join predicates.

DuckDB uses a vectorized interpreted *execution engine* [1]. This approach was chosen over Just-in-Time compilation (JIT) of SQL queries [8] for portability reasons. JIT engines depend on massive compiler libraries (e.g. LLVM) with additional transitive dependencies. DuckDB uses vectors of a fixed maximum amount of values (1024 per default). Fixed-length types such as integers are stored as native arrays. Variable-length values such as strings are represented as a native array of pointers into a separate string heap. `NULL` values are represented using a separate bit vector, which is only present if `NULL` values appear in the vector. This allows fast intersection of `NULL` vectors for binary vector operations and avoids redundant computation. To avoid excessive shifting of data within the vectors when e.g. the data is filtered, the vectors may have a selection vector, which is a list of offsets into the vector stating which indices of the vector are relevant [1]. DuckDB contains an extensive library of vector operations that support the relational operators, this library expands code for all supported data types using C++ code templates.

The execution engine executes the query in a so-called "*Vector Volcano*" model. Query execution commences by pulling the first "chunk" of data from the root node of the physical plan. A chunk is a horizontal subset of a result set, query intermediate or base table. This node will recursively pull chunks from child nodes, eventually arriving at a scan operator which produces chunks by reading from the persistent tables. This continues until the chunk arriving at the root is empty, at which point the query is completed.

DuckDB provides ACID-compliance through Multi-Version Concurrency Control (MVCC). We implement HyPer's serializable variant of MVCC that is tailored specifically for hybrid OLAP/OLTP systems [10]. This variant updates data in-place immediately, and keeps previous states stored in a separate undo buffer for concurrent transactions and aborts. MVCC was chosen over simpler schemes such as Optimistic Concurrency Control because, even though DuckDB's main use case is analytics, modifying tables in parallel was still an often-requested feature in the past.

For persistent storage, DuckDB uses the read-optimized DataBlocks storage layout [5]. Logical tables are horizontally partitioned into chunks of columns which are compressed into physical blocks using light-weight compression methods. Blocks carry min/max indexes for every column allowing to quickly determine whether they are relevant to a query. In addition, blocks carry a lightweight index for every column, which allows to restrict the amount of values scanned even further [5].

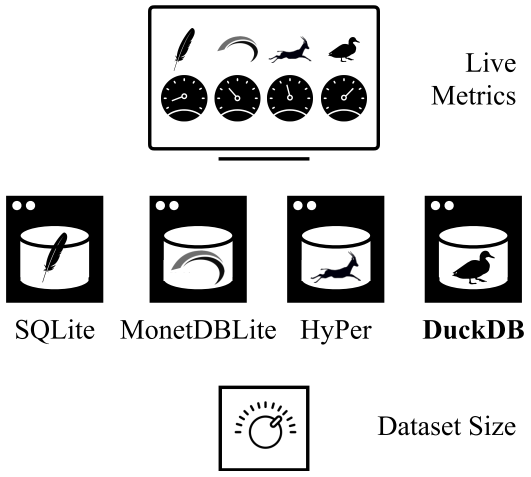

Figure 2: Demonstration setup schematic

3 DEMONSTRATION SCENARIO

In our interactive demonstration scenarios, we would like to showcase two major advantages of DuckDB: The *ability to process large data sets* on restricted hardware resources combined with the *benefit of embedded operations*.

Our demonstration is setup on a table on which a screen, a large dial and four identical benchmark computers are laid out. Each computer runs a different DBMS: SQLite, MonetDBLite, HyPer and DuckDB. Each database is pre-loaded with the TPC-H benchmark tables, chosen because the audience is likely familiar with this schema. Computers are connected using Ethernet to a fifth "management" computer which can be used to configure a query that is run in repetition on the four benchmark computers. The screen shows real-time metrics from all four benchmark computers, at least the query completion rate (QpS) and memory pressure. The dial on the table controls the amount of input data read for the currently configured query.

We propose two demonstration scenarios, a "teaser" and a "drilldown" scenario: For the "teaser" scenario, a suitable query is pre-configured on the benchmark computers. The audience will be invited to turn the physical dial to increase/decrease the amount of data that is read from the fact tables. This will immediately influence intermediate and result set sizes of the pre-configured query which will also immediately impact the metrics shown on the screen. Figure 2 illustrates this setup.

While for very small data sets all systems will show comparable behavior, only DuckDB will be able to continue functioning for larger ones. SQLite will begin to suffer from its row-based execution model and MonetDBLite begins to suffer from excessive intermediate result materialization due to its bulk processing model. While HyPer is extremely fast in processing queries, it will not be able to transfer result sets as quickly as DuckDB using its socket client protocol [12].

For the "drilldown" scenario, we invite the audience to propose their own query to be configured into the benchmark computers. This will allow direct appraisal of DuckDB's performance, without the demonstration authors being able to cherry-pick queries where DuckDB excels. Again, the audience member that has proposed the query will then be able to turn the dial to increase the amount of data read by the query and observe the impact on the four systems in real-time.

4 CURRENT STATE AND NEXT STEPS

As of this writing, DuckDB runs all TPC-H queries and all but two TPC-DS queries. We expect complete TPC-DS coverage by the time the demonstration is presented. DuckDB also already completes most of SQLite's SQL logic test suite that contains millions of queries.

Immediate next steps for DuckDB are completion of DataBlocks storage scheme and cardinality estimating. A buffer manager is also not yet implemented, but will be. DuckDB already supports inter-query parallelism but intra-query parallelism will be added as well. We plan to implement a work stealing scheduler to balance resources between short and long running queries. A special consideration is also to allow balancing resource usage with the host application, a special issue for embedded operations.

A more advanced future direction is self-checking. We have learned to distrust the hardware the database is running on. This is particularly relevant in the edge computing use case, where hardware failures are to be commonplace. One approach is to keep checksums on all persistent and intermediate data and piggy-back checksum verification on scan operators. This might be possible without a significant performance impact. A vectorized engine is particularly suited for this since a chunk of data typically fits in the CPU cache and additional passes are not requiring RAM access. Another approach to increasing trust in the hardware is inspired by video game developers which periodically run sanity check computation to ensure correct operation of CPU and RAM.

Acknowledgements

We would like to thank all past, current and future contributors to DuckDB at the CWI Database Architectures Group and elsewhere. We are also particularly indebted to the TUM database group for their papers on query optimization, window functions, storage and concurrency control that we used to implement DuckDB.

REFERENCES

[1] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR 2005, Second Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2005*. 225–237. http://cidrdb.org/cidr2005/papers/P19.pdf

[2] Lukas Fittl. 2019. C library for accessing the PostgreSQL parser outside of the server environment. https://github.com//fittl/libpg_query.

[3] Richard Hipp. 2019. Database File Format. https://www.sqlite.org/fileformat.html.

[4] Richard Hipp. 2019. Most Widely Deployed and Used Database Engine. https://www.sqlite.org/mostdeployed.html.

[5] Harald Lang, Tobias Mühlbauer, Florian Funke, et al. 2016. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. 311–326. https://doi.org/10.1145/2882903.2882925

[6] Wes McKinney. 2010. Data Structures for Statistical Computing in Python. In *Proceedings of the 9th Python in Science Conference*, Stéfan van der Walt and Jarrod Millman (Eds.). 51 – 56.

[7] Guido Moerkotte and Thomas Neumann. 2008. Dynamic programming strikes back. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*. 539–552. https://doi.org/10.1145/1376616.1376672

[8] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB* 4, 9 (2011), 539–550. https://doi.org/10.14778/2002938.2002940

[9] Thomas Neumann and Alfons Kemper. 2015. Unnesting Arbitrary Queries. In *Datenbanksysteme für Business, Technologie und Web (BTW), 16. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 4.-6.3.2015 in Hamburg, Germany. Proceedings*. 383–402. https://dl.gi.de/20.500.12116/2418

[10] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. 677–689. https://doi.org/10.1145/2723372.2749436

[11] Thomas Neumann and Bernhard Radke. 2018. Adaptive Optimization of Very Large Join Queries. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. ACM, New York, NY, USA, 677–692. https://doi.org/10.1145/3183713.3183733

[12] Mark Raasveldt and Hannes Mühleisen. 2017. Don't Hold My Data Hostage - A Case For Client Protocol Redesign. *PVLDB* 10, 10 (2017), 1022–1033. https://doi.org/10.14778/3115404.3115408

[13] Mark Raasveldt and Hannes Mühleisen. 2018. MonetDBLite: An Embedded Analytical Database. *CoRR* abs/1805.08520 (2018). arXiv:1805.08520 http://arxiv.org/abs/1805.08520

[14] Hadley Wickham, Romain François, Lionel Henry, and Kirill Müller. 2018. *dplyr: A Grammar of Data Manipulation*. https://CRAN.R-project.org/package=dplyr R package version 0.7.8.