CHAPTER 7

DuckDB: an Embeddable Analytical Database

1 Introduction

In Chapter 6, we described our effort in developing MonetDBLite, an embedded analytical system that is derived from the MonetDB system. 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. We identified the following requirements for embedded analytical database systems:

- 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.

- Controlled resource consumption and ability to operate efficiently on lower-end hardware is essential. While traditional database systems expect to be the sole occupant on a big machine, embedded database systems need to "play nice" with the host application with regards to resource usage.

- 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.

- 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.

MonetDBLite was successful in achieving high efficiency for OLAP workloads and efficient transfer of tables to and from the system. However, the fact that it was designed to be a stand-alone system resulted in many complications that prevented it from being able to fully succeed as an embedded OLAP RDBMS.

The operator–at–a–time processing model used by MonetDB materializes large intermediates entirely in memory. This memory-intensive processing model combined with the fact that MonetDB does not provide hard limits on the amount of memory that it uses can lead to MonetDB quickly consuming all the available memory of the system, leaving no memory left for the host application. This processing model also suffers from performance problems when these intermediates do not fit in memory, as the intermediates will be constantly swapped to disk.

Another issue caused by this processing model is the incapability of interrupting queries in between operators. As every single operator must run completely until the operator is finished, a user cannot quickly interrupt the execution of an expensive operator (e.g. a large cross product). This is especially problematic when the database is used in interactive scenarios as the user cannot abort a query after realizing that it takes too long to complete.

Additional problems arose from MonetDB's usage of memory mapped files to load the database data from disk. While `mmap` seems like an attractive option to allow the operating system to handle loading of data from disk into memory, the uncontrolled nature of when the data is actually fetched can cause large problems. Whenever any part of a memory mapped region is read, the OS can potentially trigger a load from disk and will send a `SIGBUS` signal to the application if that load fails. As MonetDB passes around memory mapped data all around the processing pipeline, almost any piece of code can trigger a `SIGBUS` signal. While handling these signals is possible in a stand-alone application, it is not possible in a library as signal handlers are process global and can thus (accidentally) be overwritten by the user.

MonetDB also suffers from many practical embeddability problems, including ample usage of global variables, lack of namespacing for function names resulting in potential symbol conflicts, calls to `exit` in case of fatal errors, and reliance on `setlocale` and working directory modification. While these problems can be solved, they require very large rewrites that touch almost the entire codebase.

To tackle these issues, we built *DuckDB*, a new purpose-built embeddable RDBMS. In this chapter, we present the capabilities of DuckDB. DuckDB is available as Open-Source software under the permissive MIT license[1]. 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.

1.1 Contributions

We describe the internal design of DuckDB and how it interfaces with standard analytical tools and describe how it tackles the unique challenges that are faced by an embedded analytical database system.

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

API	C/C++/SQLite	
SQL Parser	`libpg_query`	[27]
Optimizer	Cost-Based	[57, 59]
Execution Engine	Vectorized	[9]
Concurrency Control	Serializable MVCC	[60]
Storage	Custom Single-File	

Table 7.1: DuckDB: Component Overview

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.

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.

```
#include "duckdb.hpp"
DuckDB db("/tmp/db.duck");
Connection con(db);
auto result = con.Query("SELECT * FROM tbl");
cout << result->GetValue(0, 0);
```

Listing 7.1: Using DuckDB from C++

The *SQL parser* is derived from Postgres' SQL parser that has been stripped down as much as possible [27]. 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 [57] with a greedy fallback for complex join graphs [61]. It performs flattening of arbitrary subqueries as described in Nuemann et al. [59]. In addition, there are a set of rewrite rules that simplify the expression tree, by performing e.g. common subexpression elimination and constant folding. Cardinality estimation is done using a combination of samples and HyperLogLog. 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* [9]. This approach was chosen over Just-in-Time compilation (JIT) of SQL queries [58] 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 by 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. 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 [9]. 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 [60]. 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 a custom single-file storage layout inspired by the DataBlocks layout [50]. The single-file is partitioned into separate fixed-size blocks that hold the data of individual columns. When the columns are too small to fill up a single block, multiple columns can be packed together into a block to avoid wasting space. The table metadata lives in separate blocks and carries lightweight indexes for the individual blocks that the table points to, including min/max indices for every block that allow for the skipping of reading certain blocks into memory.

3 Summary

In this chapter, we have presented the embedded analytical database system DuckDB. DuckDB is a purpose-built embedded analytical database system that offers efficient execution of analytical workloads and a very fast interface between the database system and analytical tools. It is built to be embedded and solves many of the problems faced by MonetDBLite with regards to resource usage and robustness. It is easy to setup and install with zero external dependencies and can be installed through standard package and library managers of popular analytical tools.