DuckDB
An Embeddable Analytical DB

In-Process Database Management System

- Embedded into other processes where the database system is a linked library that runs completely within a “host” process.
- Typically used in scenarios where an application needs to store, retrieve, or manipulate data without the need for a separate database server or process
- Common examples - SQLite, most widely deployed engine for OLTP workloads

Need for an In-Process OLAP

- Interactive data analysis, where data is analysed and according to insights decisions are take. Tool available such as R and Python lack full-query optimization and transactional storage.

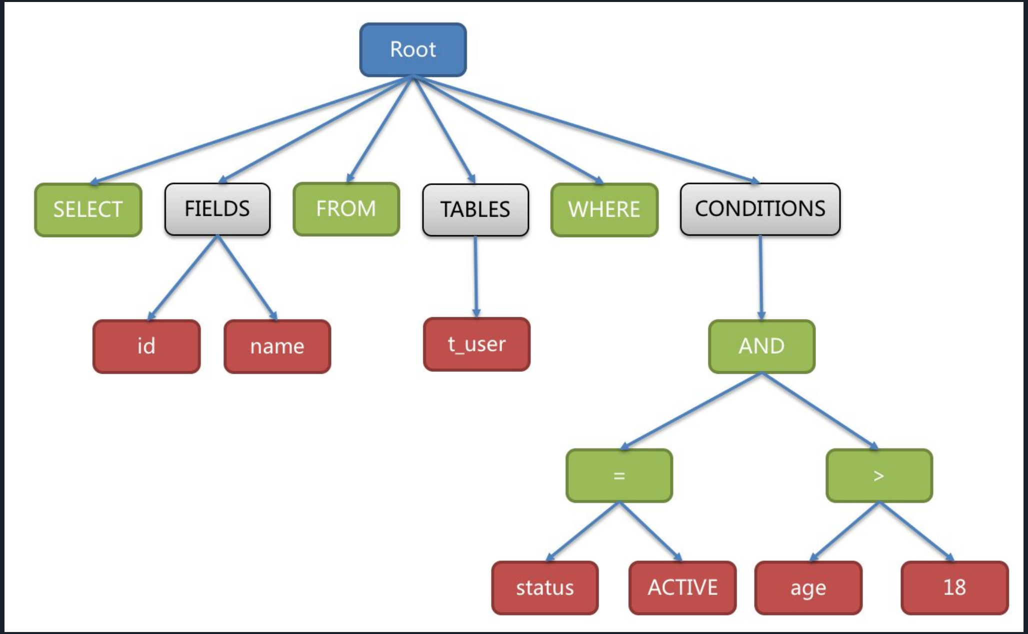
- Edge Computing in scenarios where data analysis needs to happen closer to the source. Traditional data forwarding to central locations can be inefficient due to bandwidth constraints and privacy concerns.

What is expected out of an In-Process OLAP?

- High efficiency for OLAP workloads, but without sacrificing OLTP performance.
- Efficient transfer of tables to and from the database, since database and application run in the same process.
- An embedded database should not compromise the stability of the host application.
- The database should be able to run in whatever environment the host does.

DuckDB Components - Parser

DuckDB employs a SQL parser derived from Postgres. It takes SQL query strings as input and returns a parse tree. The parse tree consists of statements (e.g., SELECT, INSERT) and expressions (e.g., SUM(a)+1).



DuckDB Components - Logical Planner

- DuckDB’s logical planner includes two parts, the binder and the plan generator.
- The binder resolves expressions referring to schema objects, such as tables or views, with their column names and types.
- The plan generator transforms the parse tree into a tree of basic logical query operators, like scan (table, view), filter (WHERE), project (columns) and join.
- The binder resolves table references and enriches the parse tree with column information, while the plan generator transforms the enriched parse tree into a tree of logical query operators that describe how the query should be executed.

DuckDB Components - Optimizer

- DuckDB's optimizer performs join order optimization using dynamic programming with a greedy fallback for complex join graphs.

- It also performs flattening of arbitrary subqueries by resolving nesting.

- It also rewrites rules to simplify the expression tree, by removing redundant calculations/expressions. It also uses constant folding which calculates constant expressions at compile-time.

DuckDB Components - Physical Planner

- The physical planner takes the optimized logical plan and transforms it into the physical plan, selecting suitable implementations where applicable.

- Decisions the physical planner takes -
  - Accessing data efficiently by deciding whether to scan the whole table or use an existing index on that table.
  - Decide on join strategies to use based on the cost incurred, in terms of I/O operations, CPU processing, memory usage.

DuckDB Components - Execution Engine

- DuckDB employs a vectorized interpreted execution engine. It uses vectors to store the data efficiently.

- Some relevant features of vectors
  - Integers are stored in arrays.
  - Strings are managed through pointers to a separate string storage.
  - To prevent unnecessary data reorganization, a selection vector is used, specifying which parts of the data are relevant for the current operation.

DuckDB Components - Vector Volcano Model

- This approach involves processing data in chunks or vectors of values, rather than row by row.

- A chunk is a horizontal subset of a result set or query intermediate or the base table. This node then recursively pulls chunks from child nodes, eventually arriving at a scan operator reading from persistent tables.

- The execution commences by pulling the first "chunk" of data from the root node of the physical plan.

- This continues until the chunk arriving at the root is empty, at which point the query is completed..

ACID Compliance

- Although DuckDB main focus is analytics, it ensures the integrity of the data using Multi-Version Concurrency Control (MVCC).

- It uses an existing variant of MVCC for OLAP/OLTP systems (from HyPer DB)

- This variant updates data in-place immediately, and keeps previous states stored in a separate undo buffer for concurrent transactions and aborts.

DataBlocks Storage

- DuckDB stores the data in-memory but for persistent storage, it employs the read-optimized DataBlocks storage.

- This approach horizontally partitions logical tables into chunks of columns, which are compressed again.

- Blocks also carry min/max indexes for every column, which enables quick determination of their relevance to a query.

Performance - Teaser Scenario

- Suitable query is pre-configured to the benchmark systems for SQLite, MonetDBLite, HyPer, and DuckDB. For small datasets all systems perform similar.

- For larger datasets, all other databases performs bad than DuckDB
    - SQLite suffers from its row-based execution model
    - MonetDBLite begins to suffer from excessive intermediate result materialisation
    - HyPer is fast in processing queries but is not able to transfer result sets as quickly as DuckDB does.

QUESTIONS?