Imperial College
London

MEng Individual Project
Department of Computing

Schema Compilation with Query-Set Aware Optimisation for Embedded Database Generation

Author	**Supervisor**	**Second Marker**
Oliver Killane	Holger Pirk	Tobe Decided

14th November 2023

Abstract

Embedded databases allow developers to easily embed a data store within an application while providing a convenient query interface. For many applications, the schema of the data store is static, and the set of parameterized queries is known at application compile time.

No existing embedded databases take advantage of this knowledge for logical optimisation. The goal of this project is to build one that does.

The main outcome of this project is *emdb*: a prototype embedded database compiler that generates the optimised code for a data store from a schema and queries.

Acknowledgements

I would like to thank my suprvisor, Dr. Holger Pirk, for his willingness to supervise my wild idea, and for his teaching of the data processing module that inspired it.

I would also like to thank my friends, family,and the staff of Imperial College for enabling me to persue my passion in computer science.

Finally I would like to thank the public at large for not implementing this project before I could.

Contents

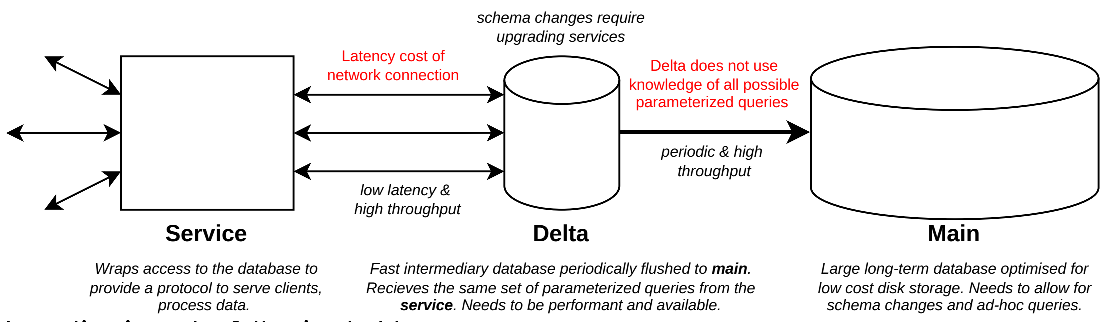
Chapter 1

Introduction

1.1 Motivation

Many applications require a fast, structured data store. Correctly implementing a correct, performant custom solution in a general purpose programming language is a significant development cost. Relational databases provide an attrative interface for describing the structure and interfactions with a store, but come with performance caveats and are not trivially integrated into applications.

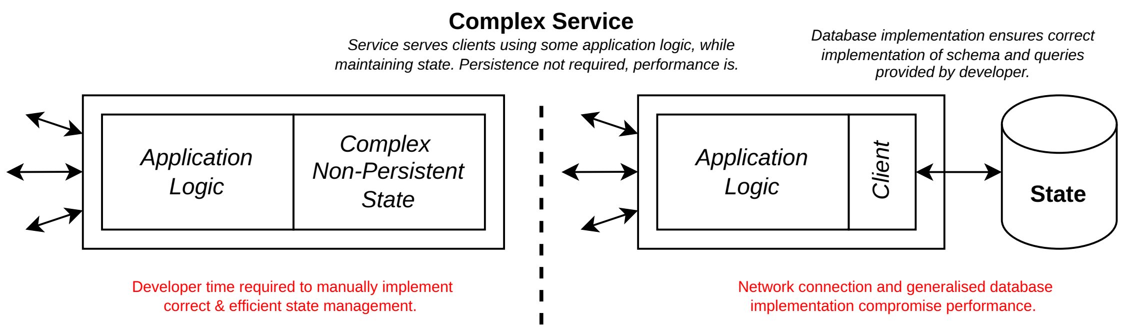
For example a common OLTP pattern is to use a database to store the current state of a service.


schema changes require upgrading services
Latency cost of network connection
Delta does not use knowledge of all possible parameterized queries
periodic & high throughput
low latency & high throughput
Service
Delta
Main
Wraps access to the database to provide a protocol to serve clients, process data.
Fast intermediary database periodically flushed to **main**. Recieves the same set of parameterized queries from the **service**. Needs to be performant and available.
Large long-term database optimised for low cost disk storage. Needs to allow for schema changes and ad-hoc queries.

For many such applications the following holds.

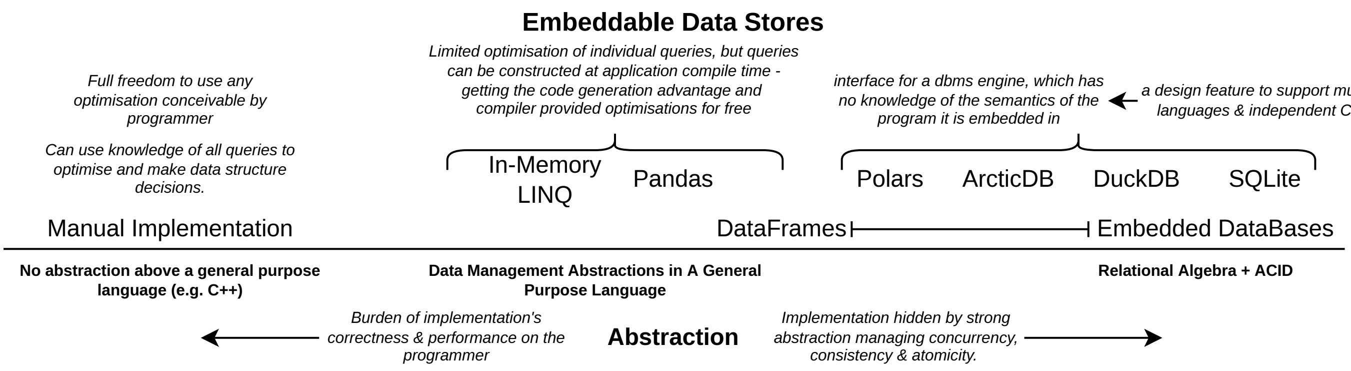
1. Persistence requirements are weak enough to allow for in-memory only storage (optionally durability by replication).
2. Each instance of the application is the sole interactor with the data store.
3. Invariants/Constraints about the data must be maintained by the store.
4. The schema and parameterized queries used are known at compile time.

Another more common pattern is to use implement complex state of an application in a general purpose programming language.

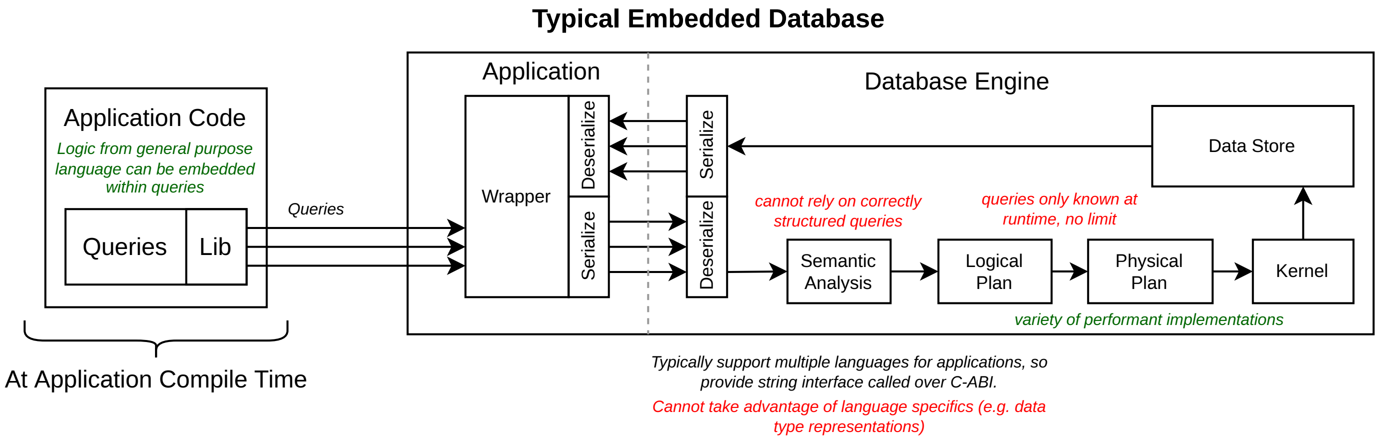

Complex Service
Service serves clients using some application logic, while maintaining state. Persistence not required, performance is.
Database implementation ensures correct implementation of schema and queries provided by developer.
Application Logic
Complex Non-Persistent State
Application Logic
Client
State
Developer time required to manually implement correct & efficient state management.
Network connection and generalised database implementation compromise performance.

Much like the first pattern, a schema and set of queries including constraints are present. The burden is either on the programmer to implement, or on the performance of the application when using the convenience of a query language & database.

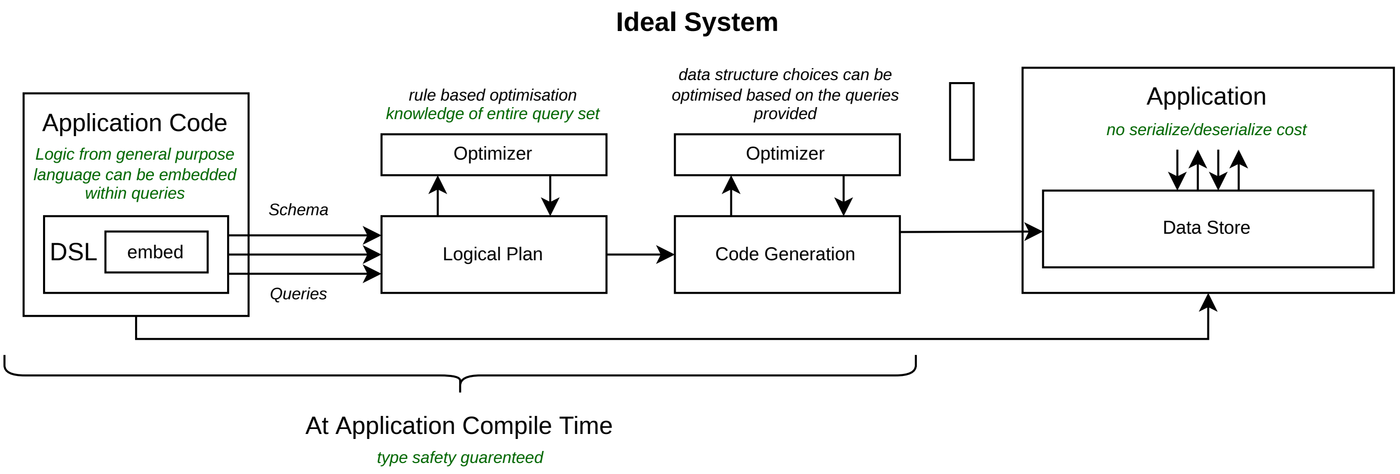
A spectrum of solutions exist to embed a store in an application, here generally placed according to the strength of abstraction provided.


Embeddable Data Stores
Limited optimisation of individual queries, but queries can be constructed at application compile time - getting the code generation advantage and compiler provided optimisations for free
Full freedom to use any optimisation conceivable by programmer
Can use knowledge of all queries to optimise and make data structure decisions.
interface for a dbms engine, which has no knowledge of the semantics of the program it is embedded in
a design feature to support multiple languages & independent CLIs
In-Memory LINQ
Pandas
Polars
ArcticDB
DuckDB
SQLite
Manual Implementation
DataFrames
Embedded DataBases
No abstraction above a general purpose language (e.g. C++)
Data Management Abstractions in A General Purpose Language
Relational Algebra + ACID
Burden of implementation's correctness & performance on the programmer
Abstraction
Implementation hidden by strong abstraction managing concurrency, consistency & atomicity.

Unfortunately the strongest abstractions are present in embedded databases that do not take advantage of 4 as by design they both support shcema changes, and are independent from the host language. For example duckDB is embedable in Python, Java, Julia (and other) applications. The typical embedded database design is just an in-memory database, packaged to be in the same processa an application.


Typical Embedded Database
Application Code
Logic from general purpose language can be embedded within queries
Queries
Lib
Queries
At Application Compile Time
Application
Wrapper
Deserialize
Serialize
Database Engine
Serialize
Deserialize
Data Store
cannot rely on correctly structured queries
queries only known at runtime, no limit
Semantic Analysis
Logical Plan
Physical Plan
Kernel
variety of performant implementations
Typically support multiple languages for applications, so provide string interface called over C-ABI.
Cannot take advantage of language specifics (e.g. data type representations)

An ideal system would allow such a datastore to be expressed in a query language, with an embeddable implementation generated for use, and optimised using the full knowledge of the query set & schema. It would also provide a strong compile time guarentee on the correctness of all queries.


Ideal System
Application Code
Logic from general purpose language can be embedded within queries
DSL
embed
Schema
Queries
rule based optimisation
knowledge of entire query set
Optimizer
Logical Plan
data structure choices can be optimised based on the queries provided
Optimizer
Code Generation
Application
no serialize/deserialize cost
Data Store
At Application Compile Time
type safety guarenteed

No such ideal system exists.

1.2 Experimentation

We can quantify the potential upside of such an ideal system by comparing its performance against both a naive/unoptimised implementation, and against existing embedded databases. in-memory embedded databases against a *naive* and *foresight* implementations.

Naive	Implemented assuming the provided queries are only a subset of all required. uses a generational arena[6] that supports insert, delete & update .
Foresight	Implemented assuming provided queries are the only queries required (as is the case for emdb).

The schema chosen is designed as a simple OLTP workload, that is too complex to be expressed with a key-value store or dataframes without comparable programmer effort to a manual implementation, by requiring:

- Automatically generated unique ids.
- Predicate constraints.
- Queries that mutate multiple rows in atomic transactions.
- Updates collecting returning values.

The schema is implemented in the emQL language designed for this project.

```
table users {
    name: String,
    premium: bool,
    credits: i32,
} @ [ genpk(id), pred(premium || credits > 0) as prem_credits ]
```

Both sqlite and duckdb support `CHECK` constraints for `prem_credits`, in sqlite an `AUTOINCREMENT` constraint is used for id. In DuckDB a default value attached to a sequence is used to increment for every insert.

For the naive and foresight implementations we must ensure all writes are checked in advance to ensure the constraint is maintained, or the entire query has no effect. When multiple rows are changed, this includes iterating through and pre-generating results before applying any.

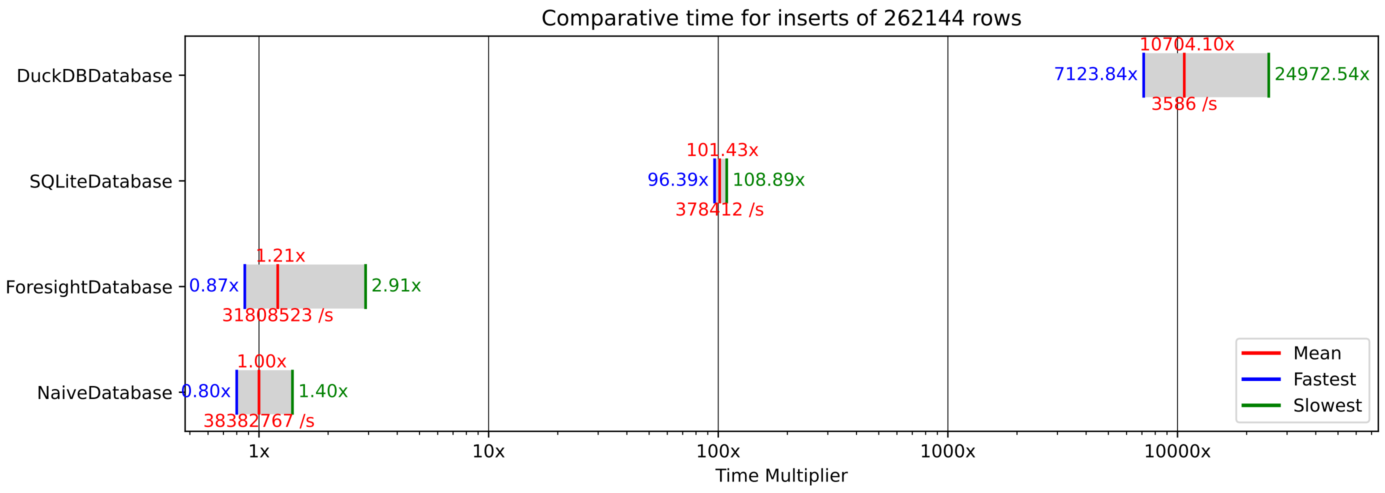
```
query new_user(username: String, prem: bool) {
    row(name: String = username, premium: bool = prem, credits: i32 = 0 ) |> insert(users) ~> return;
}
```

For the integrated implementations the cost can be reduced to a inserting to an arena, the `username` does not need to be copied. This demonstrates the advantage in integration, the database implementation can both use the rust string representation (no copy), and rely on the rust compiler to ensure the string is not mutated from elsewhere.

For the *Foresight* implementation there is an additional early check on `premium` to place those users in a separate arena, to improve the performance of the `reward_premium` query that iterate over only premium users.

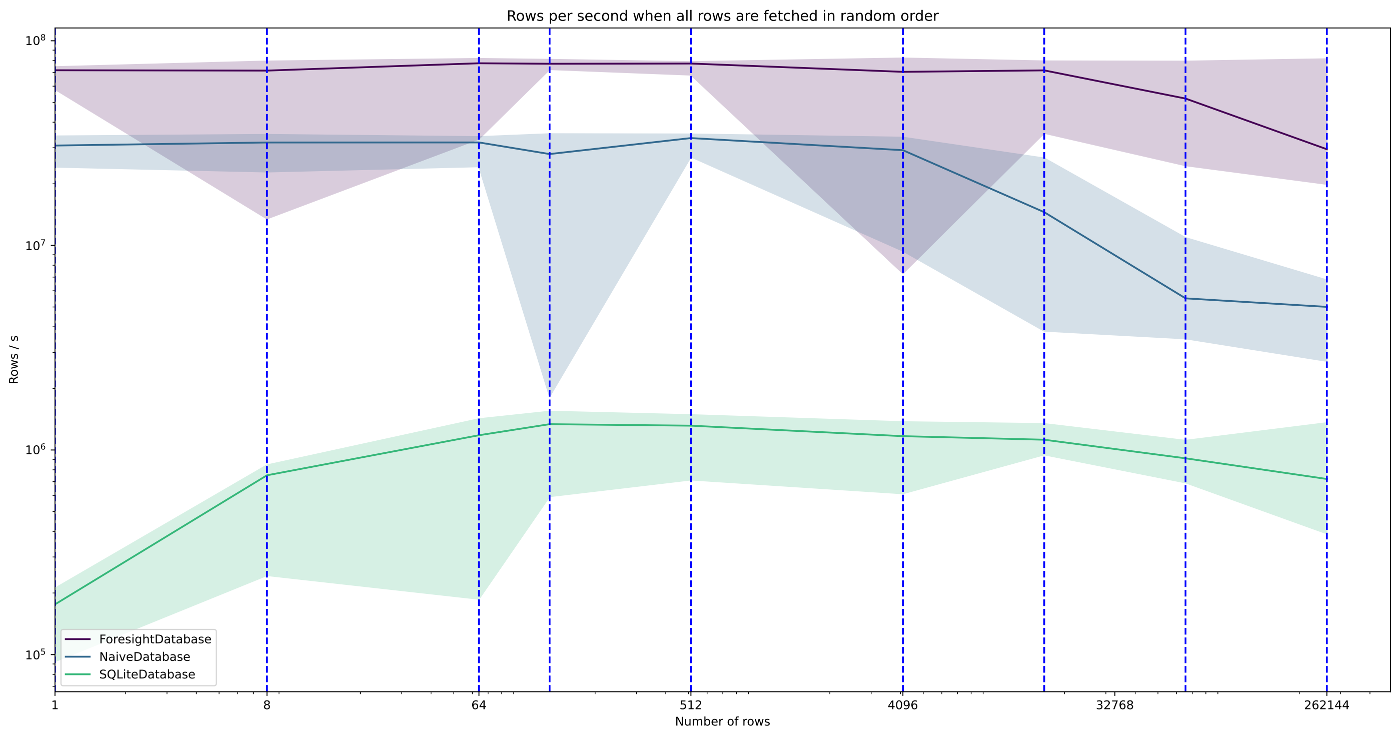
DuckDB and SQLite both suffer in performance due to the need to construct and then parse the query, retrieve the cached prepared query, and execute. As a columnar storage DuckDB is at a significant disadvantage. Performance can be significantly improved by batching the inserts, however this is not possible for many OLTP applications.

This can be seen in the comparative performance.



```
query get_info(user_id: usize) {
    use users |> unique(use user_id as id) ~> return;
}
```

For the *foresight* implementation, unlike the others, no copy of the name is required to return the user's information, as no queries mutate the username a reference (qualified to the lifetime of the database) can be used.

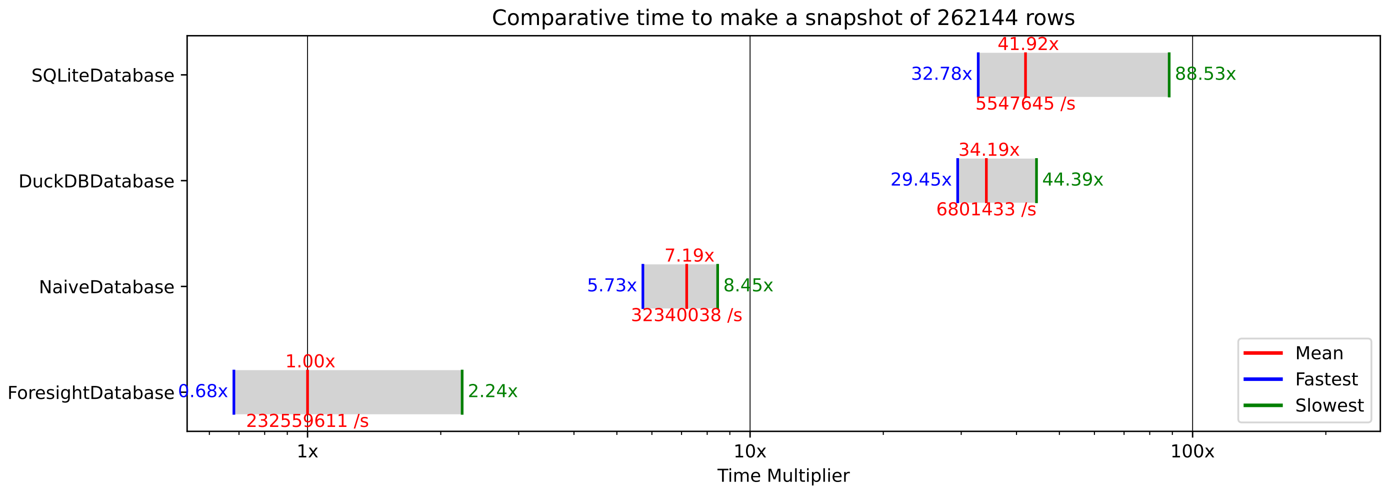


Note that we do not compare duckDB, it is designed for OLAP and as such it is not does not support fast primary key lookups that determine the time required for these queries.

```
query get_snapshot() {
    use users |> collect() ~> return;
}
```

Much like the `get_info` query, we can use information that no queries ever mutate the username in order to remove the cost of copying strings and instead return a reference qualified to the lifetime of the database. For this benchmark all names were a simple `"User{id}"` string.

The advantage of the foresight implementation can be arbitrarily increased by making the strings longer & more expensive to copy.

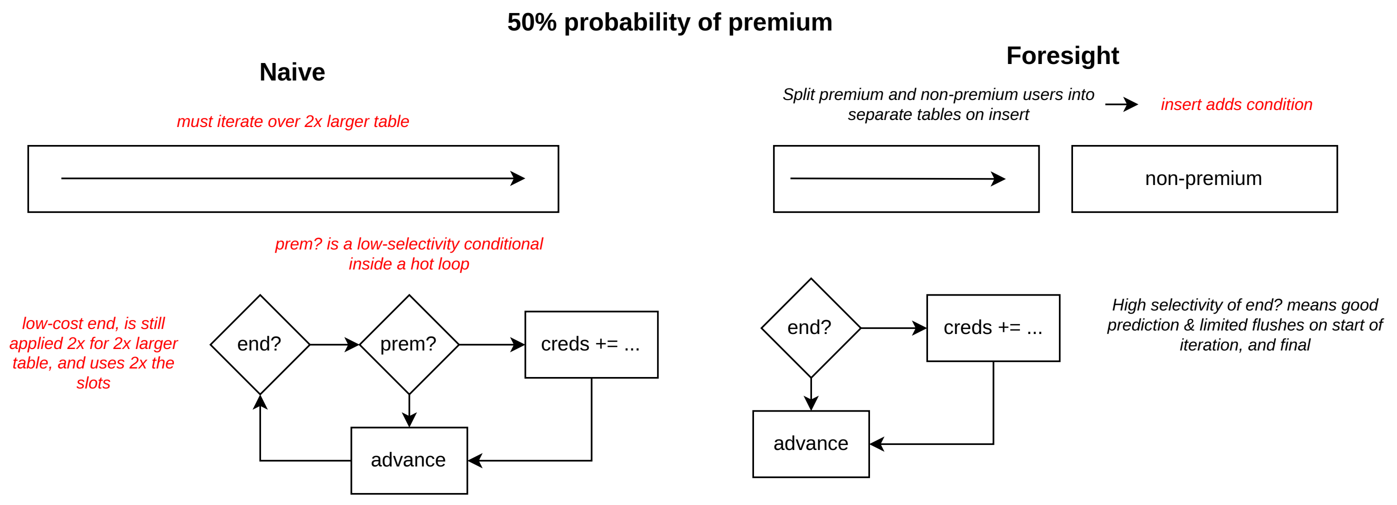


```
query add_credits(user: usize, creds: i32) {
    ref users |> unique(use user as id) ~> update(it use credits = credits + creds);
}
```

This query is useful only in ensuring it is not possible to optimise the `credits` column away as dead code.

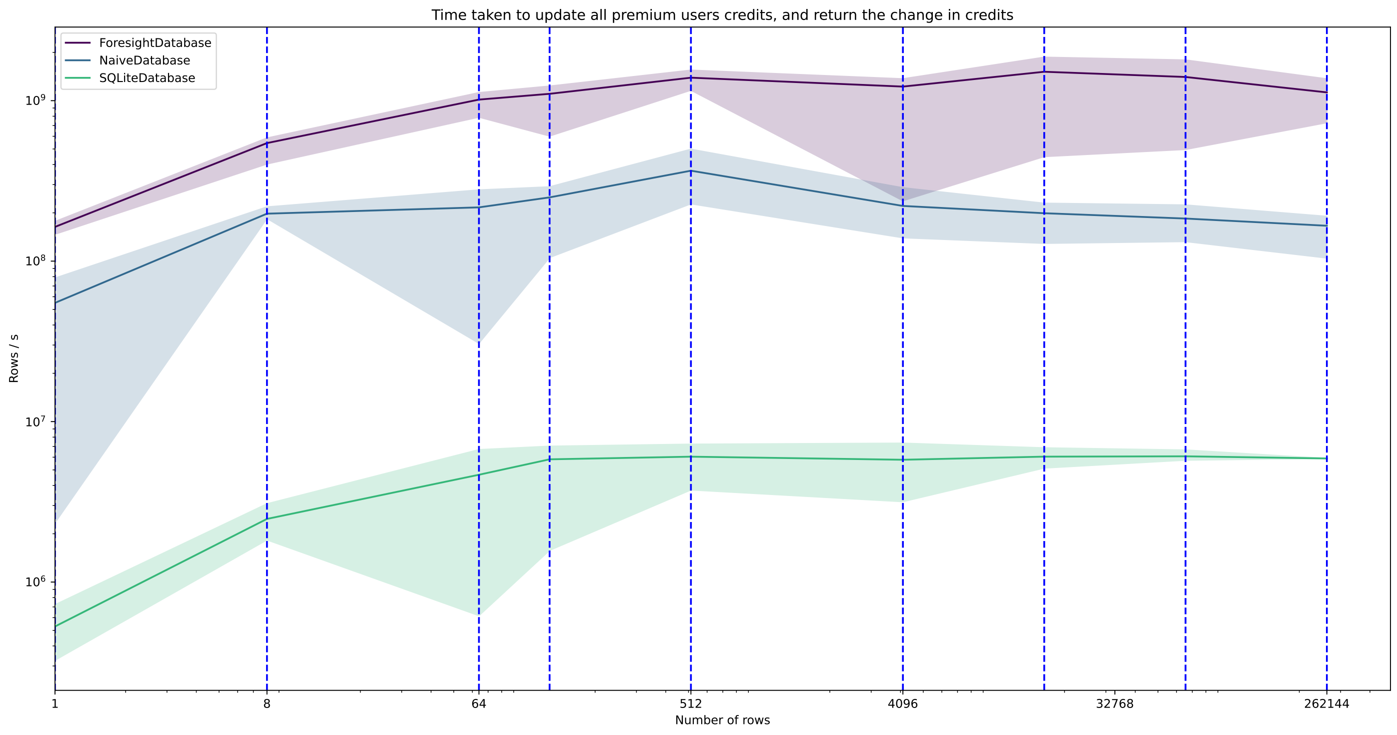
```
query reward_premium(cred_bonus: f32) {
    ref users
        |> filter(it.premium)
        |> map(user: users::Ref = it, new_creds: i32 = ((it.credits as f32) * cred_bonus) as i32)
        |> update(it use credits = new_creds)
        |> map(creds: i32 = new_creds)
        |> fold((sum: i64 = 0) => (sum = sum + creds))
        ~> return;
}
```

The `filter(..)` can be pushed into the `users` relation itself for the *foresight* implementation.



50% probability of premium
Naive
must iterate over 2x larger table
prem? is a low-selectivity conditional inside a hot loop
low-cost end, is still applied 2x for 2x larger table, and uses 2x the slots
end?
prem?
creds += ...
advance
Foresight
Split premium and non-premium users into separate tables on insert
insert adds condition
non-premium
end?
creds += ...
advance
High selectivity of end? means good prediction & limited flushes on start of iteration, and final

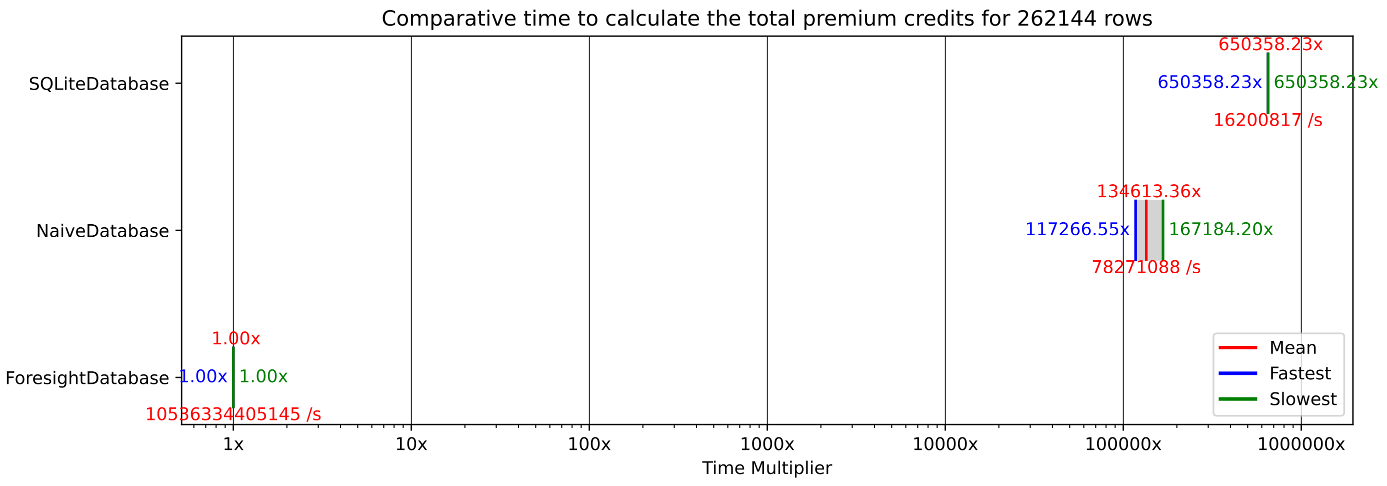
This is reflected in the benchmarks for both `reward_premium(..)`, and the lower *foresight* performance for `new_user(..)`.



```
query total_premium_credits() {
    use users
        |> filter(premium)
        |> map(credits: i64 = credits)
        |> fold((sum: i64 = 0) => (sum = sum + credits))
        ~> return;
}
```

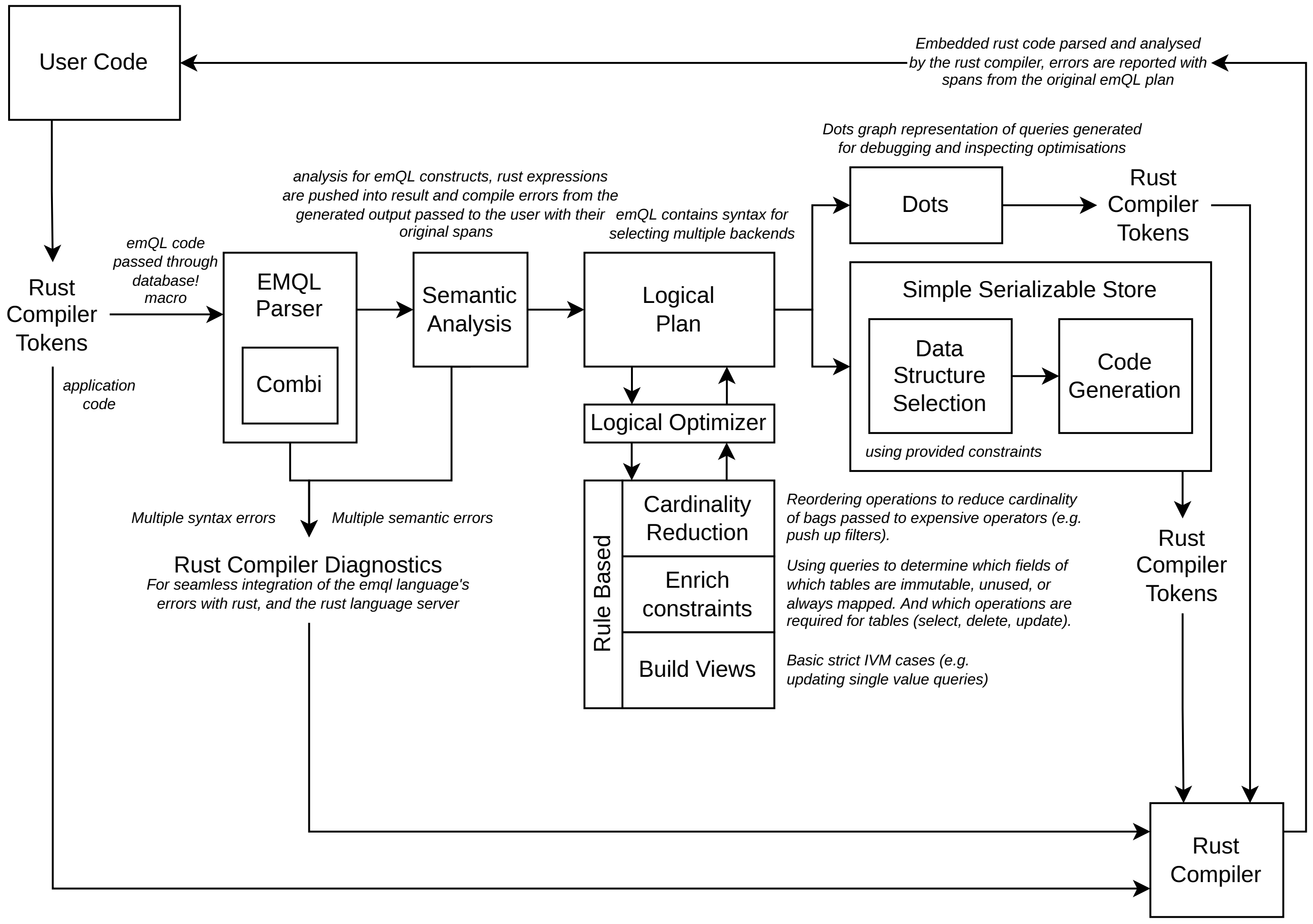
We can also keep a view of this query. By updating a counter on changes to the credits of premium users (at the cost of insert and update) we reduce the query to a single pointer lookup of a stack value. This can be inlined to remove the cost of the query function call.

The result is a significant performance improvement, at the cost of updating counters on `new_user(..)`, `add_credits(..)` and `reward_premium(..)`.



1.3 Objectives

The objective of the project is to build the following system.



User Code
Embedded rust code parsed and analysed by the rust compiler, errors are reported with spans from the original emQL plan
Dots graph representation of queries generated for debugging and inspecting optimisations
Dots
Rust Compiler Tokens
analysis for emQL constructs, rust expressions are pushed into result and compile errors from the generated output passed to the user with their original spans
emQL contains syntax for selecting multiple backends
emQL code passed through database! macro
Rust Compiler Tokens
EMQL Parser
Combi
Semantic Analysis
Logical Plan
Simple Serializable Store
Data Structure Selection
Code Generation
using provided constraints
application code
Logical Optimizer
Multiple syntax errors
Multiple semantic errors
Rust Compiler Diagnostics
For seamless integration of the emql language's errors with rust, and the rust language server
Rule Based
Cardinality Reduction
Enrich constraints
Build Views
Reordering operations to reduce cardinality of bags passed to expensive operators (e.g. push up filters).
Using queries to determine which fields of which tables are immutable, unused, or always mapped. And which operations are required for tables (select, delete, update).
Basic strict IVM cases (e.g. updating single value queries)
Rust Compiler Tokens
Rust Compiler

Objective 1. *Construct a basic emQL query compiler the supporting a subset of SQL functionality*

This will demonstrate the viability of the concept even without complex optimisation by allowing seamless embedding of schemas in rust code, and rust code inside schemas.

Objective 2. *Develop a logical optimiser that demonstrates a performance improvement by using the queries to make data structure choices*

None of the currently available systems can take advantage of knowing the entire query set, doing so provides a unique optimisation advantage.

Objective 3. *Implement a comprehensive set of tested, benchmarked example schemas*

Important to ensure the correctness of the system, as well as to demonstrate a performance improvement by the optimizer over a large variety of plans.

Additional Contributions

As part of the project we will also make contributions to the following.

Additional 1. *Developing an easy to use parser-combinator library for rust tokenstreams called Combi.*
Additional 2. *Contributions to the divan benchmarks library to support the project.*
Additional 3. *A well-documented standalone data structure code generation library (as will be used by the emdb compiler)*

Chapter 2

Background

2.1 Embedded Databases

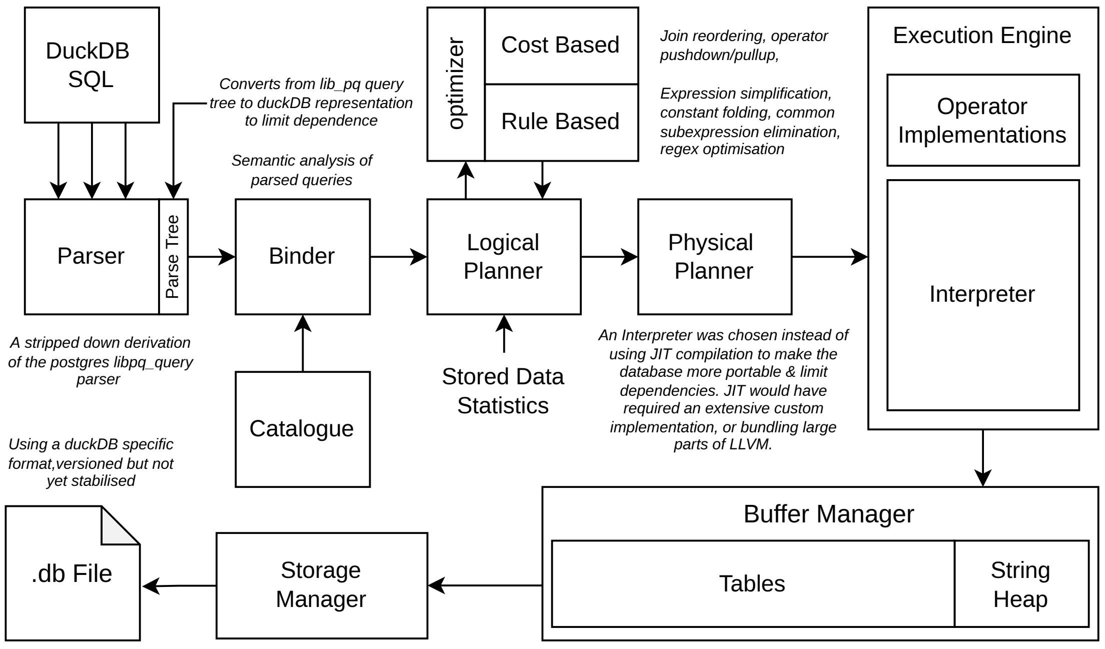
2.1.1 DuckDB

DuckDB is an in-memory, embedded, columnar, OLAP Database[14] developed as a successor to MonetDBLite[13] in the embedable database OLAP niche.

Embedable	The database is a single file with no dependencies, and is easily embeddable in and usable from applications written in Python, R, Java, Julia, Swift and Rust[11].
Cost-Aware	In addition to common rule-based optimisations, DuckDB uses a cost-model and statistics collected at runtime for its logical optimizer.
Extensible	DuckDB supports loadable WASM extensions.
Language Agnostic	DuckDB communicates through the C-ABI and uses its heap, as a result it cannot take advantage of language/compiler specifics (e.g. data representation).
Columnar	To better support OLAP access patterns. This hurts OLTP performance, which is better suited to a nary storage, furthermore the only supported indexes are zonemaps/min-max indexes and adaptive radix trees. Neither offer lookup performance comparable to hash indecies, or indexed arenas as demonstrated in 1.2.
Concurrency	DuckDB uses a combination of optimistic and multiversion concurrency control.

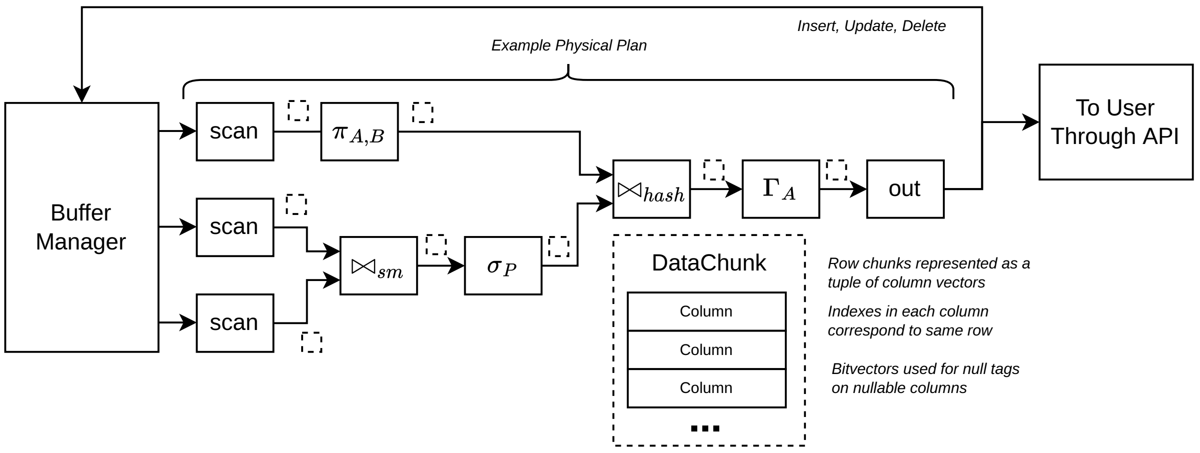
System Design

The authors describe the design as *"textbook"*[14].


DuckDB SQL
Converts from lib_pq query tree to duckDB representation to limit dependence
Semantic analysis of parsed queries
optimizer
Cost Based
Rule Based
Join reordering, operator pushdown/pullup,
Expression simplification, constant folding, common subexpression elimination, regex optimisation
Execution Engine
Operator Implementations
Interpreter
Parser
Parse Tree
Binder
Logical Planner
Physical Planner
A stripped down derivation of the postgres libpq_query parser
Stored Data Statistics
An Interpreter was chosen instead of using JIT compilation to make the database more portable & limit dependencies. JIT would have required an extensive custom implementation, or bundling large parts of LLVM.
Catalogue
Using a duckDB specific format,versioned but not yet stabilised
.db File
Storage Manager
Buffer Manager
Tables
String Heap

Vector Volcano Processing

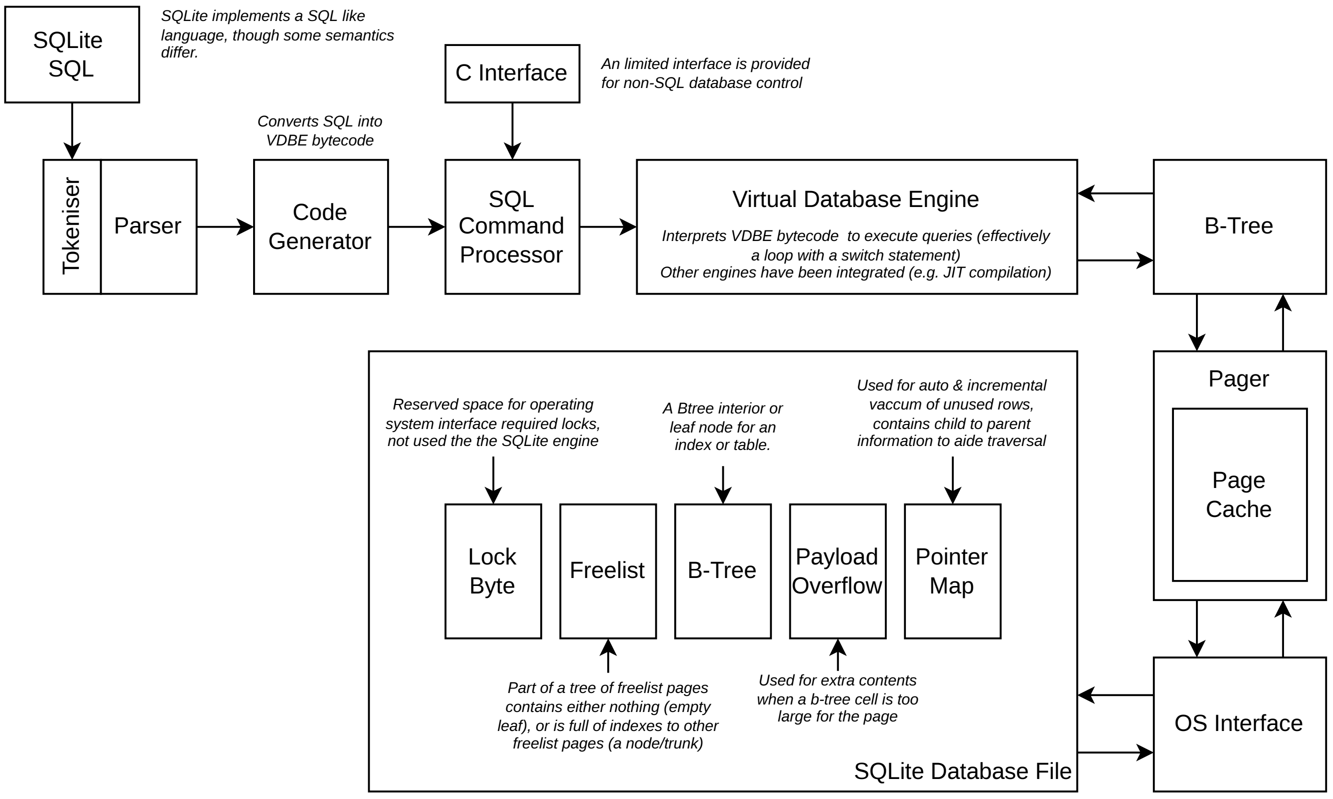
DuckDB uses a *vector volcano* processing model. The interpreter takes a dynamically constructed tree of operators, wherein each operator pulls data from input operators on demand much like with volano processing. However instead of pulling individual tuples, `DataChunks` are passed, each containing a tuple of column vectors for a row-range of the previous operator's output.



2.1.2 SQLite

SQlite is a lightweight embedded database[7], and is currently the most deployed database in the world[4]. Unlike DuckDB it is designed for OLTP workloads, and as such stores rows in an nary record format.

System Design



SQLite SQL
SQLite implements a SQL like language, though some semantics differ.
C Interface
An limited interface is provided for non-SQL database control
Converts SQL into VDBE bytecode
Tokeniser
Parser
Code Generator
SQL Command Processor
Virtual Database Engine
Interprets VDBE bytecode to execute queries (effectively a loop with a switch statement)
Other engines have been integrated (e.g. JIT compilation)
B-Tree
Pager
Page Cache
OS Interface
Reserved space for operating system interface required locks, not used the the SQLite engine
A Btree interior or leaf node for an index or table.
Used for auto & incremental vaccum of unused rows, contains child to parent information to aide traversal
Lock Byte
Freelist
B-Tree
Payload Overflow
Pointer Map
Part of a tree of freelist pages contains either nothing (empty leaf), or is full of indexes to other freelist pages (a node/trunk)
Used for extra contents when a b-tree cell is too large for the page
SQLite Database File

Virtual Database Engine Bytecode

One of the key elements of SQLite's design is that rather than using traditional physical plan (i.e. trees of operators) it instead uses a simple bytecode, interpreted on the virtual database engine.

```
CREATE TABLE users (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name VARCHAR NOT NULL,
    premium BOOLEAN NOT NULL,
    credits MEDIUMINT NOT NULL,
    CONSTRAINT premcredits CHECK (premium OR credits >= 0)
);

-- Get Total Premium credits
EXPLAIN SELECT SUM(credits) FROM users WHERE premium = TRUE;
```

When run with SQLite (compiled with `-DSQLITE_ENABLE_EXPLAIN_COMMENTS`) the following bytecde is returned:

		Registers					
addr	**opcode**	**p1**	**p2**	**p3**	**p4**	**p5**	**comment**
0	Init	0	13	0	null	0	`Start at 13`
1	Null	0	1	2	null	0	`r[1..2]=NULL`
2	OpenRead	0	5	0	4	0	`root=3 iDb=0; users`
3	Rewind	0	9	0	null	0	
4	Column	0	2	3	null	0	`r[3]= cursor 0 column 2`
5	Ne	4	8	3	BINARY-8	83	`if r[3]!=r[4] goto 8`
6	Column	0	3	3	null	0	`r[3]= cursor 0 column 3`
7	AggStep	0	3	2	sum(1)	1	`accum=r[2] step(r[3])`
8	Next	0	4	0	null	1	
9	AggFinal	2	1	0	sum(1)	0	`accum=r[2] N=1`
10	Copy	2	5	0	null	0	`r[5]=r[2]`
11	ResultRow	5	1	0	null	0	`output=r[5]`
12	Halt	0	0	0	null	0	
13	Transaction	0	0	3	0	1	`usesStmtJournal=0`
14	Integer	1	4	0	null	0	`r[4]=1`
15	Goto	0	1	0	null	0	

JIT Compilation for SQLite

In order to improve performance, without burdening developers with the additional development & maintenance cost of writing a JIT compiler, one can be generated from the interpreter. This strategy has been attempted with SQLite[9] and advertised a $1.72\times$ speedup over a seelction of TPC-H queries.

2.2 Code Generation for Databases

2.2.1 Holistic Integrated Query Engine

The Holistic Integrated Query Engine[10] is a single-threaded, JIT code generating, general purpose relational database that implements queries using a C++ code generator and attached C++ compiler. Typical just-in-time compilation powered databases use a lower-level representation, and pass this to a bundled compiler (e.g. LLVM), however there are several advantages to using C++ as the target representation.

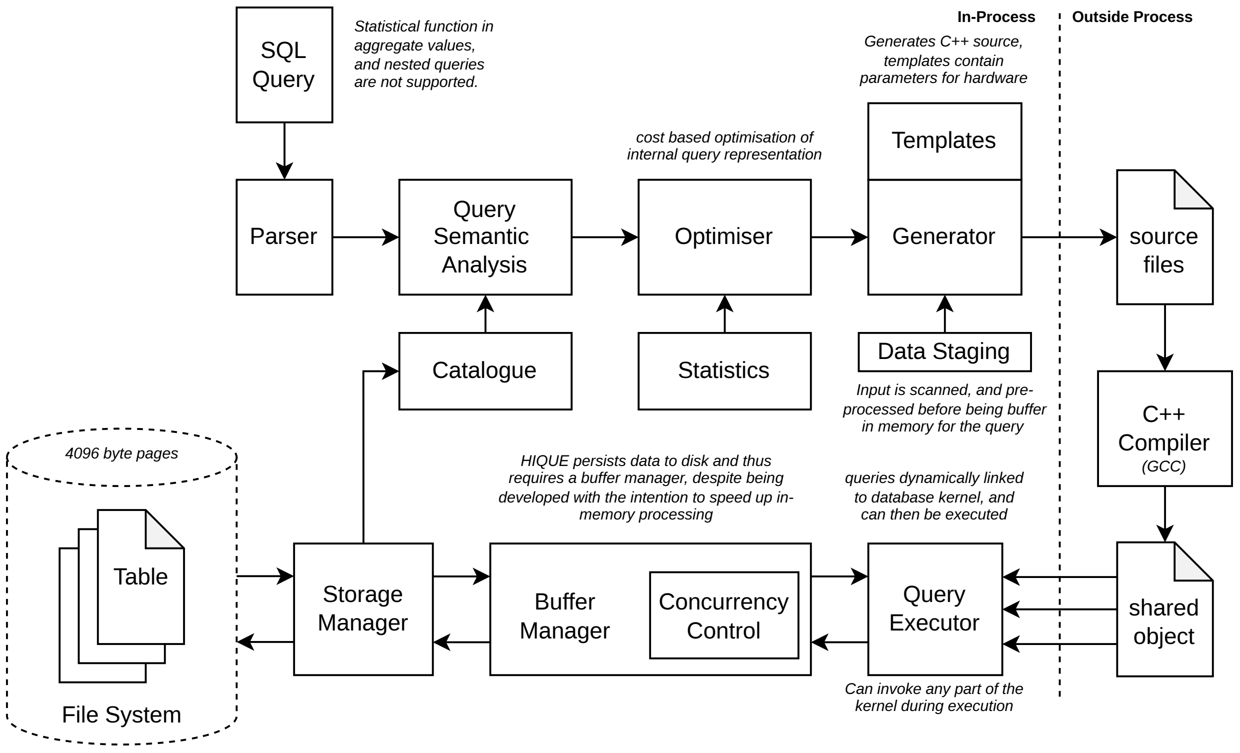
Visibility	The output is easily inspectable C++, and the compiler can optionally include debug information, or additional instrumentation when compiling queries to assist in debugging the engine, and the compiler itself verifies the type safety of code. While HIQUE was evaluated using GCC, it is possible to swap out the compiler for another (e.g. Clang) for additional features without difficulty.
Templates	The templates used by the code generator are also written in C++, making them easier to write and maintain.
Optimisation	A broad range of optimisations (including for the native hardware) can be performed, and the compiler has access to the entire context of the query code.

The significant downside to source generation is the time & system resource taken by using a full C++ compiler, which increases with query complexity, and the level op optimisation. This is particularly problematic for small queries associated with OLTP workloads.

The main focus of HIQUE is to avoid the poor instruction and data cache performance associated with the volcano processing model by using hand-optimised templates to generate cache concious code for common operations. A large focus of this to improve the performance of iteration over rows. By using the known types of fixed-length tuples, and accessing through direct referencing & pointer arithmetic, no function calls are required to tuple access, and the system can use the size of the tuple to avoid random accesses to block sizes only resident in the lower levels of the memory hierarchy.

HIQUE also uses a Partitioned Attributed Across (PAX) record layout[2] that stores fixed size ranges of rows as a tuples of column vectors, this provides the row lookup advantages of nary stprage (all items of a given row are stored in the same page, requiring at most of page load for access), as well as the better cache performance of columnar storage (allowing simple linear scans over columns). This is conceptually similar to the `DataChunks` abstraction used by DuckDB.

System Design



SQL Query
Statistical function in aggregate values, and nested queries are not supported.
In-Process
Outside Process
Generates C++ source, templates contain parameters for hardware
cost based optimisation of internal query representation
Templates
Parser
Query Semantic Analysis
Optimiser
Generator
source files
Catalogue
Statistics
Data Staging
Input is scanned, and pre-processed before being buffer in memory for the query
C++ Compiler (GCC)
4096 byte pages
HIQUE persists data to disk and thus requires a buffer manager, despite being developed with the intention to speed up in-memory processing
queries dynamically linked to database kernel, and can then be executed
Table
Storage Manager
Buffer Manager
Concurrency Control
Query Executor
shared object
File System
Can invoke any part of the kernel during execution

Relevance to emdb

The cost of using a full C++ compiler at compile time is the only major downside to the many upsides of generating high-level code.

By lifting query compilation to the application's compile time, the same benefits can be achieved with emdb, without needing to do any query compilation work at runtime.

The only performance downside of this is that chip-specific parameters (e.g. cache sizes) are only available when the binary is run, if it compiled for an architecture in general, rather than for the exact chip of the machine the application will run on.

2.3 Incremental View Maintenance

Views

Recompute on Access

```
CREATE VIEW my_view AS SELECT * FROM ..;
```

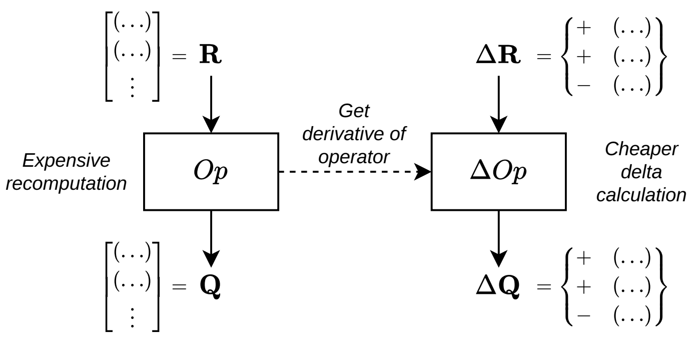
An alias for a query, typically a `SELECT` statement, that can be queried like a table. The view's query is used each time it is accessed.[8]

Materialized Views

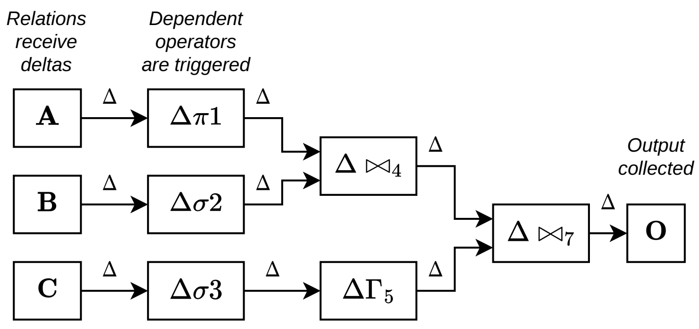
Recompute on Change

Views with results cached. The contained query is only recomputed after a change in the data the query relies on. Beneficial for expensive queries that are frequently accessed and depend on infrequently changing data.

The aim of incremental view maintenance is to support views on data that **Never Recompute** in their entirety, but instead can updated *incrementally* using changes applied to source relations.



$\begin{bmatrix}(\ldots)\\(\ldots)\\\vdots\end{bmatrix} = \mathbf{R}$
Expensive recomputation
Op
Get derivative of operator
ΔOp
Cheaper delta calculation
$\Delta\mathbf{R} = \begin{Bmatrix}+ & (\ldots)\\+ & (\ldots)\\- & (\ldots)\end{Bmatrix}$
$\begin{bmatrix}(\ldots)\\(\ldots)\\\vdots\end{bmatrix} = \mathbf{Q}$
$\Delta\mathbf{Q} = \begin{Bmatrix}+ & (\ldots)\\+ & (\ldots)\\- & (\ldots)\end{Bmatrix}$



The cost reduction from recomputing operators to recomputing the deltas of operators can also be applied recursively.

2.3.1 DBToaster

DBToaster is an incremental view maintenance code generation tool, that generates C++, Spark (including a distributed spark target) and OCaml implementations from queries.

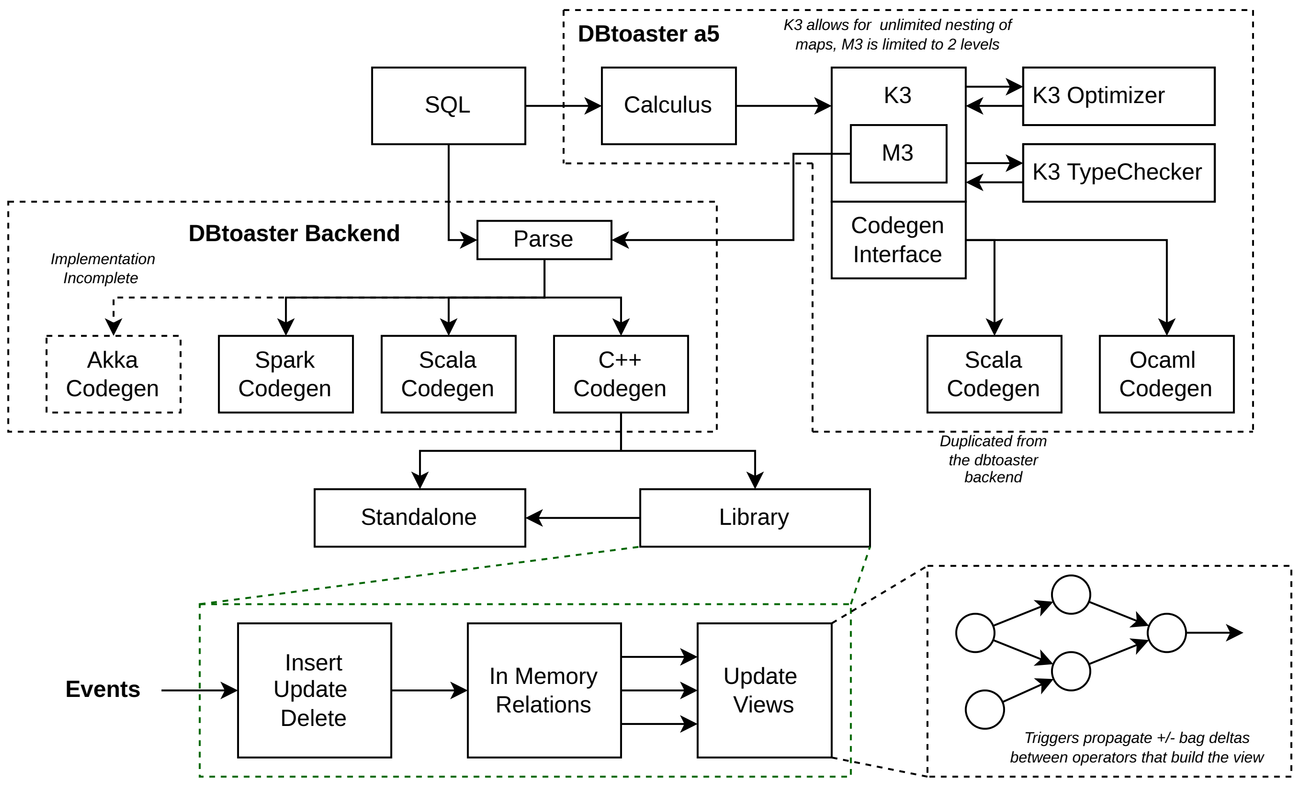
SQL Support

A SQL syntax is supported (with incomplete compliance with ASNI SQL-92) to construct select queries on streams[5] of tuple changes and are the only way to write/mutate relations in the system. Tables are supported, but are static and cannot be modified after load.

By forgoing complex write insert, update and delete queries, DBToaster avoids much of the complexity in combining transactions, constraints and delta queries (the calculus used for generating delta queries has no support for relation mutation).

A restricted set of conditionals & functions are supported, and external functions are possible (depending on backend used).

System Design

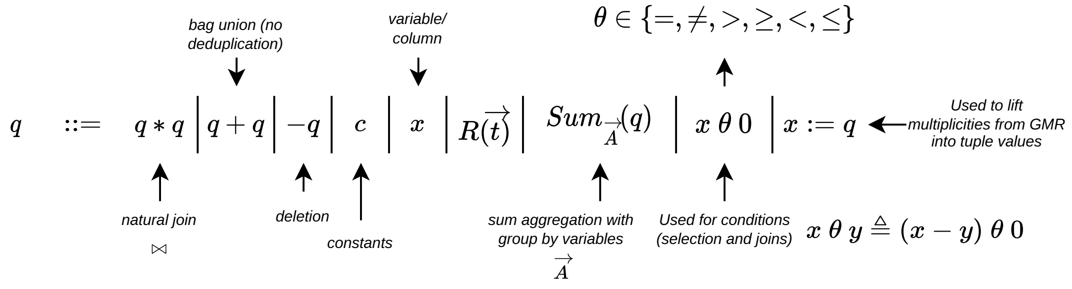


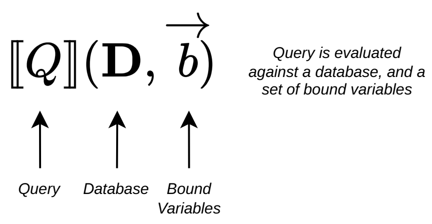
One of the key advantages for DToaster is that the code generated is easily embeddable in applications, which provides the same advantages as discussed in 1.2.

Aggregation Calculus

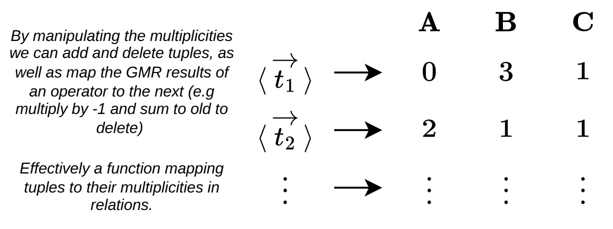
DBToaster lifts queries parsed from SQL and represented by a simplified relational algebra into its own aggregation calculus (AGCA). AGCA represents data through generalized multiset relations (GMRs) which are mappings from the set of tuples to the multiplicites of those tuples in relations.

Aggegate Calculus Syntax


$\theta \in \{=, \neq, >, \geq, <, \leq\}$
bag union (no deduplication)
variable/ column
$q \quad ::= \quad q * q \mid q + q \mid -q \mid c \mid x \mid R(\overrightarrow{t}) \mid Sum_{\overrightarrow{A}}(q) \mid x\ \theta\ 0 \mid x := q$
Used to lift multiplicities from GMR into tuple values
natural join
$\bowtie$
deletion
constants
sum aggregation with group by variables
$\overrightarrow{A}$
Used for conditions (selection and joins)
$x\ \theta\ y \triangleq (x - y)\ \theta\ 0$


$[\![Q]\!](\mathbf{D}, \overrightarrow{b})$
Query is evaluated against a database, and a set of bound variables
Query
Database
Bound Variables

AGCA Evaluation Function


By manipulating the multiplicities we can add and delete tuples, as well as map the GMR results of an operator to the next (e.g multiply by -1 and sum to old to delete)
Effectively a function mapping tuples to their multiplicities in relations.
A
B
C
$\langle \overrightarrow{t_1} \rangle \longrightarrow$ 0 3 1
$\langle \overrightarrow{t_2} \rangle \longrightarrow$ 2 1 1
$\vdots \longrightarrow \vdots\ \vdots\ \vdots$

Generalised Multiset Relations

Evaluation rules for the language are provided in the paper *DBToaster: Higher-order Delta Processing for Dynamic, Frequently Fresh Views*[1] and are ommitted for brevity.

SQL translation is done by converting to relational algebra (with bag semantics), upon which operations can be reduced to union ($+$) and join ($\bowtie$) by judicial allowance for infinite relations. For example $\sigma_{A<B}(R)$ is rewritten as $R \bowtie (A < B)$ despite $A < B$ being an infinitely large set of possible tuples.

```
SELECT * FROM R WHERE B < (SELECT SUM (D) FROM S WHERE A > C);
```

$$Sum_{[A,B]}(R(A,B) * (z := Sum_{[]}(S(C < D) * (A > C) * D)) * (B < z))$$

From AGCA the delta queries can be generated by simple recursive descent of the AGCA expression, applying the following rules:

$$\begin{aligned}
\Delta(Q_1 + Q_2) &\equiv (\Delta Q_1) + (\Delta Q_2) \\
\Delta(Q_1 * Q_2) &\equiv ((\Delta Q_2) * Q_2) + (Q_1 * (\Delta Q_2)) + ((\Delta Q_1) * (\Delta Q_2)) \\
\Delta(x := Q) &\equiv (x := (Q + \Delta Q)) - (x := Q) \\
\Delta(-Q) &\equiv -(\Delta Q) \\
\Delta(Sum_{\vec{A}} Q) &\equiv Sum_{\vec{A}}(\Delta Q)
\end{aligned}$$

$$\Delta(x\ \theta\ 0) \equiv \Delta x \equiv \Delta c \equiv 0$$

As AGCA is closed under taking the delta query, this process can be reapplied for higher order delta queries to incrementally compute delta queries.

2.4 Embedded Domain Specific Languages

Embedded Domain Specific Languages (eDSLs) allow for the expression of domain specific logic and concepts within a host (typically general purpose) programming language. Such eDSLs are typically distributed as a library and implemented using features of the host language, not requiring additional compilation steps by external tools.

eDSLs can be roughly categorised as the following:

String Based The most popular implementation, wherein a library is provided to process strings of the eDSL at runtime. By using strings, any syntax is possible at the cost of language integration.

This lack of integration prevents the eDSL from being compiled at application compile time, introducing a runtime performance cost as well as correctness as generic bugs such as type errors (in the eDSL) can exist even after the application is successfully compiled. The lack of integration also prevents automatic support by the host language's tools (e.g. language server).

Typically stringified DSLs also dont allow for the host language to be embedded/used inside the DSL, as no compiler is available at runtime to compile embeddings, and the wider context of the program is erased at application compile time.

This is the approach taken by most SQL client libraries.

Compiler Provided

Full compiler integration can be supported by implementing the eDSL within the host language's compiler.

This allows for custom syntax, eDSL compilation & analysis to occur at the same time as the host language, and full integration with the host language's tools. As the eDSL code exists in the host's compiler, the eDSL compiler can also use context and analysis from outside the eDSL to aide in analysis (e.g. type inference for SQL queries based on the types of variables their output are assigned to).

This comes at significant development & decision cost (as do all changes to major compilers).

Some common examples include the embedding of JSX expressions in javascript.

```
// ... some program logic
const element = <h1>Hello, {world_formatter(name)}</h1>;
```

Or the SQL syntax for Langiage Integrated Queries (LINQ) in C#.

```
int[] data = [ ... ];
var evenData = from num in data where (num % 2) == 0 select num;
foreach (var num in evenData) { Console.WriteLine(num); }
```

Host Native

Rather than creating new syntax, the language is instead implemented using only constructs from the host language. Often this takes the form of libraries providing rich typed APIs, and exist on a vague spectrum from eDSL, to simply a library with a complex API.

The eDSL design is restricted by the host language, and this strategy is more often employed in languages that allow custom operators and evaluations contexts (monads & effects).

For example Gigaparsec[15] is a Haskell parser combinator library implemented with a DSL.

```
-- Parse a hell[l]*o world and return the world
parser :: Parser String
parser = (atomic (string "he")) >> (manyN 2 (atomic (string "l")))
      >> (atomic (string "o ")) >> (atomic (string "world"))

parse @String parser "hellllllllllllo world"
```

Macro Based The eDSL is implemented as a macro taking inputs provided by the host compiler in a syntax agnostic form (e.g. text, or tokens). This allows for custom syntax, while also allowing the eDSL to be compiled at application compile time without requiring changes to the host compiler.

For example the Racket language is designed specifically for developing eDSLs through macros.

One of the most popular languages to support this feature is Rust, which supports procedural macros, where a user defined macro can run arbitrary rust code at compile time, taking compiler tokens and producing new tokens and compiler diagnostics.

For example the yew[16] web frontend framework for rust contains a JSX-like `html! { .. }` macro that takes in html, with embedded rust (in place of javascript).

```
let user = ... ;
let page = html! {
    <div class="container">
        <h1>{ "Hello World!" }</h1>
        <p>{
            if let Some(name) == user {
                format!("And hello {}!", name)
            } else {
                "And hello stranger!".to_owned()
            }
            }</p>
    </div>
};
```

Another example is sqlx[12], which is a SQL toolkit for rust that can validate & type check query strings against a development database, at application compile time.

```
// this query string is not a string, but rather a string tokn to be parsed &
//used by sqlx at compile time
let countries = sqlx::query!(
        "
-- Get the number of premium users in each credit bracket, given a minimum
-- number of credits
SELECT name, COUNT(*) as count
FROM users
GROUP BY (credits / 100) * 100
WHERE premium AND credits > ?
        ",
        min_credits
    )
    .fetch_all(&connection_pool)
    .await?;
```

One of the key strengths of the non-string based eDSL implementations are the ability to embed the host language in the eDSL. This limits the requirement for the user to consider an entirely different language (syntax, semantics & libraries), or to manage complex conversions where the eDSL and host language interface.

> *The problem is simple: two languages are more than twice as difficult to use as one language. The host and query languages often use different notations for the same thing, and convenient abstractions such as higher-order functions and nesting may not be available in the query language. Interfacing between the two adds to the mental burden on the programmer and leads to complex code, bugs, and security holes...*
> **- A Practical Theory of Language-Integrated Query**[3]

2.4.1 Rust Procedural Macros

Rust supports several types of macros as part of its procedural macro system. Unlike C/C++ macros are invoked and run after tokenisation.

Declarative Using `macro_rules!` to declare a set of patterns and substitutions.

```
macro_rules! log_assign {
    ($variable:ident = $value:expr) {
        let $variable = $value;
        println!("trace: {} = {}", stringify!($variable), $value)
    };
}
log_assign!(x = 5);
```

Attribute Applied to modules, structs, enums, or functions. They can take their own arguments, and the tokenstream representing the structure they are tagged to.

For example the divan benchmarking library uses an attribute macro to declare parameters for benchmarks.

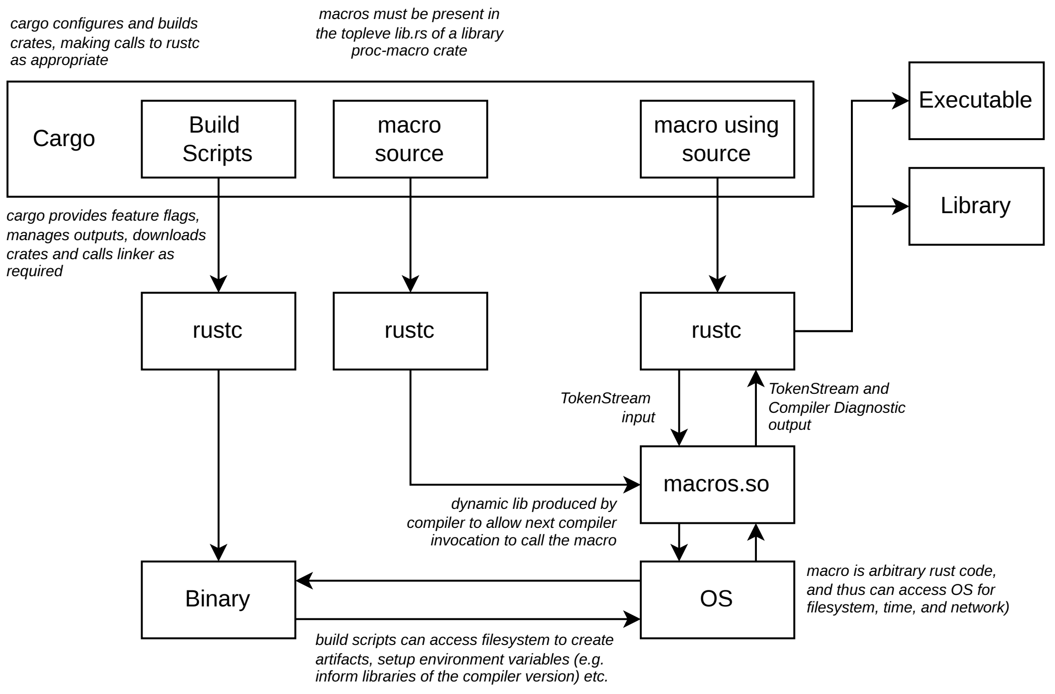
```
#[divan::bench(
    name = "random_inserts",
    types = [Foresight, Naive, DuckDB, SQLite],
    consts = TABLE_SIZES
)]
fn inserts<'a, T, const N: usize>(bencher: Bencher) { ... }
```

Derive Macros used to derive a named trait for a struct or enum. Standardised to allow for their invocation by the `#[derive(..)]` attribute macro.

```
#[derive(Clone, Debug, PartialEq, Eq, Hash)]
struct Foo {
    name: String,
    premium: bool,
    credits: i32,
}
```

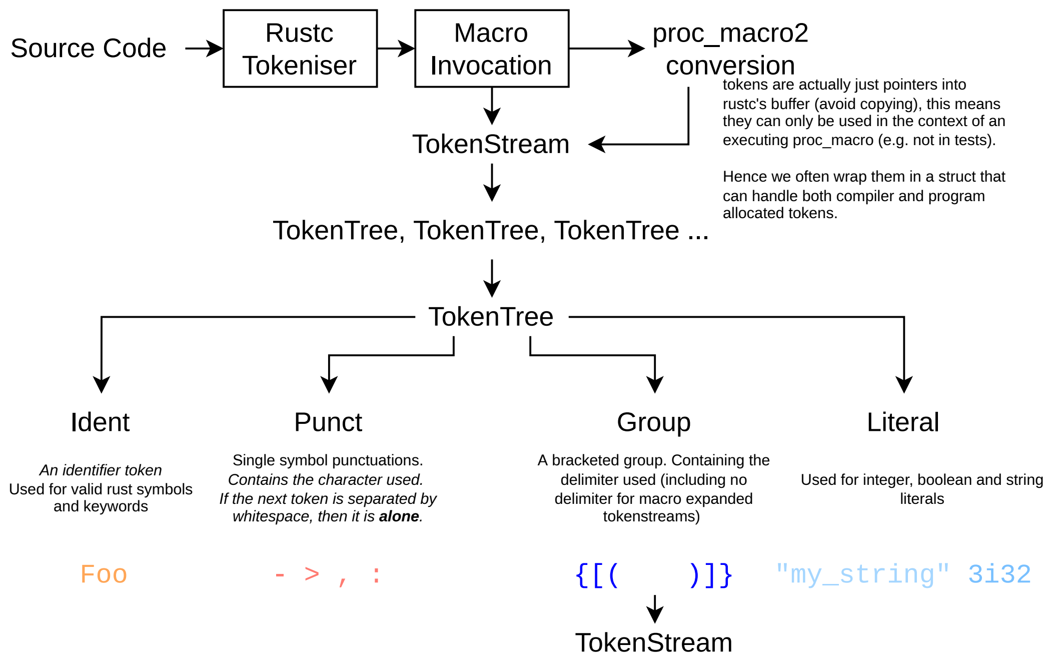
Functional Macros that can be invoked like functions, and take arbitrary tokens as input. Examples include `vec![..]` and `println!(" ... ", ..)`.

All of these macros (including `macro_rules!`) are implemented by procedural macros which are compiled and executed as follows:


cargo configures and builds crates, making calls to rustc as appropriate
macros must be present in the topleve lib.rs of a library proc-macro crate
Cargo
Build Scripts
macro source
macro using source
Executable
Library
cargo provides feature flags, manages outputs, downloads crates and calls linker as required
rustc
rustc
rustc
TokenStream input
TokenStream and Compiler Diagnostic output
macros.so
dynamic lib produced by compiler to allow next compiler invocation to call the macro
Binary
OS
macro is arbitrary rust code, and thus can access OS for filesystem, time, and network)
build scripts can access filesystem to create artifacts, setup environment variables (e.g. inform libraries of the compiler version) etc.

A simple functional macro can be declared defined as follows:

```
use proc_macro::TokenStream;
extern crate proc_macro;

#[proc_macro]
fn my_macro(tk: TokenStream) -> TokenStream {  /* Can emit compiler diagnostics from here */}
```


Source Code
Rustc Tokeniser
Macro Invocation
proc_macro2 conversion
tokens are actually just pointers into rustc's buffer (avoid copying), this means they can only be used in the context of an executing proc_macro (e.g. not in tests).
Hence we often wrap them in a struct that can handle both compiler and program allocated tokens.
TokenStream
TokenTree, TokenTree, TokenTree ...
TokenTree
Ident
An identifier token
Used for valid rust symbols and keywords
Punct
Single symbol punctuations.
Contains the character used.
If the next token is separated by whitespace, then it is alone.
Group
A bracketed group. Containing the delimiter used (including no delimiter for macro expanded tokenstreams)
Literal
Used for integer, boolean and string literals
Foo
- > , :
{[(   )]}
"my_string" 3i32
TokenStream

In order to generate output tokens we can either construct outselves, or use the quasi-quoting library `quote!` to construct tokenstreams from rust snippets.

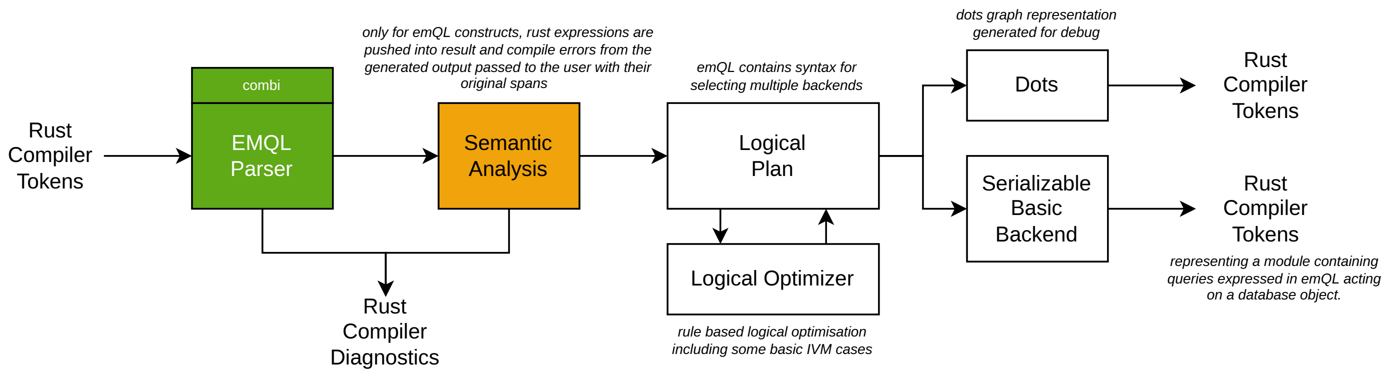
```
let row_type = ... ;
let filter_fn_code = quote!{
    fn filter(row: #row_type) -> Option<#row_type> {
        // wrap the user's code with a function to constraint its types. Any incorrect user
        // code's errors will be propagated back to the original positions in the source as
        // the spans for user_filter are from the source.
        fn do(#row_fields) -> bool { #user_filter }

        if do(#get_row_fields) { None } else { Some(row) }
    }
}
```

Chapter 3

Project Plan

3.1 Current Status

The current progress of the project is as follows (green is complete, orange in progress):



Rust Compiler Tokens
combi
EMQL Parser
only for emQL constructs, rust expressions are pushed into result and compile errors from the generated output passed to the user with their original spans
Semantic Analysis
emQL contains syntax for selecting multiple backends
Logical Plan
Logical Optimizer
rule based logical optimisation including some basic IVM cases
dots graph representation generated for debug
Dots
Serializable Basic Backend
Rust Compiler Tokens
Rust Compiler Tokens
representing a module containing queries expressed in emQL acting on a database object.
Rust Compiler Diagnostics

Github projects are being used to track the progress of tasks for the project. There is no point in copying that here. Click here for the emDB repo.

Chapter 4

Evaluation Plan

4.1 Tests for Objectives

Objective 1. *Construct a basic emQL query compiler the supporting a subset of SQL functionality*

Evaluated by implementing several schema (including the 1.2 user details schema) and testing the correctness of each.

Objective 2. *Develop a logical optimiser that demonstrates a performance improvement by using the queries to make data structure choices*

Evaluated by comparing the performance of the compiler on the aforementioned test schemas with logical optimisations on, off, and against duckDB & SQLite (with graph plan views as evidence the optimisations being applied).

Objective 3. *Implement a comprehensive set of tested, benchmarked example schemas*

Evaluated by covering all major operators, and by measuring the coverage of the compiler with coverage instrumentation.

Chapter 5

Ethical Considerations

5.1 None Present

The end goal of this project is a data store generator that can be used for any number of purposes good or evil.

Bibliography

[1] Yanif Ahmad et al. "DBToaster: higher-order delta processing for dynamic, frequently fresh views". In: *Proc. VLDB Endow.* 5.10 (June 2012), pp. 968–979. ISSN: 2150-8097. DOI: 10.14778/2336664.2336670. URL: https://doi.org/10.14778/2336664.2336670.

[2] Anastassia Ailamaki et al. "Weaving Relations for Cache Performance". In: *Proceedings of the 27th International Conference on Very Large Data Bases*. VLDB '01. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001, pp. 169–180. ISBN: 1558608044.

[3] James Cheney, Sam Lindley, and Philip Wadler. "A practical theory of language-integrated query". In: *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. ICFP '13. Boston, Massachusetts, USA: Association for Computing Machinery, 2013, pp. 403–416. ISBN: 9781450323260. DOI: 10.1145/2500365.2500586. URL: https://doi.org/10.1145/2500365.2500586.

[4] SQLite Consortium. *SQLite Website*. 2024. URL: https://www.sqlite.org/ (visited on 01/05/2024).

[5] DATA Laboratory at EPFL. *The DBToaster SQL Reference*. 2019. URL: https://dbtoaster.github.io/docs_sql.html (visited on 01/02/2024).

[6] Nick Fitzgerald. *Generational Arena: A safe arena allocator that allows deletion without suffering from the ABA problem by using generational indices*. 2023. URL: https://github.com/fitzgen/generational-arena (visited on 01/05/2024).

[7] Kevin P. Gaffney et al. "SQLite: past, present, and future". In: *Proc. VLDB Endow.* 15.12 (Aug. 2022), pp. 3535–3547. ISSN: 2150-8097. DOI: 10.14778/3554821.3554842. URL: https://doi.org/10.14778/3554821.3554842.

[8] The PostgreSQL Global Development Group. *PostgreSQL 16 Documentation*. 2023. URL: https://www.postgresql.org/docs/current/index.html (visited on 01/05/2024).

[9] Aleksei Kashuba and Hannes Mühleisen. *Automatic Generation of a Hybrid Query Execution Engine*. Aug. 2018. URL: https://arxiv.org/pdf/1808.05448.pdf.

[10] Konstantinos Krikellas, Stratis D. Viglas, and Marcelo Cintra. "Generating code for holistic query evaluation". In: *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*. 2010, pp. 613–624. DOI: 10.1109/ICDE.2010.5447892.

[11] DuckDB Labs. *DuckDB Documentation*. 2024. URL: https://duckdb.org/docs/ (visited on 01/05/2024).

[12] LaunchBadge. *sqlx Github Repository*. 2024. URL: https://github.com/launchbadge/sqlx (visited on 01/24/2024).

[13] Mark Raasveldt. "MonetDBLite: An Embedded Analytical Database". In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD '18. Houston, TX, USA: Association for Computing Machinery, 2018, pp. 1837–1838. ISBN: 9781450347037. DOI: 10.1145/3183713.3183722. URL: https://doi.org/10.1145/3183713.3183722.

[14] Mark Raasveldt and Hannes Mühleisen. "DuckDB: an Embeddable Analytical Database". In: *Proceedings of the 2019 International Conference on Management of Data*. SIGMOD '19. Amsterdam, Netherlands: Association for Computing Machinery, 2019, pp. 1981–1984. ISBN: 9781450356435. DOI: 10.1145/3299869.3320212. URL: https://doi.org/10.1145/3299869.3320212.

[15] Jamie Willis. *GigaParsec Github Repository*. 2024. URL: https://github.com/j-mie6/gigaparsec (visited on 01/24/2024).

[16] YewStack. *Yew Github Respository*. 2024. URL: https://github.com/yewstack/yew (visited on 01/24/2024).