IMPERIAL

MEng Individual Project
Department of Computing

Imperial College of Science, Technology and Medicine

emDB

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

Author
Oliver Killane

Supervisor
Holger Pirk

Second Marker
Tony Field

17th June 2024

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: an embedded database compiler wrapped in a procedural macro that generates the code for a data store from a schema and queries, and achieves significant performance improvements over SQLite and DuckDB on a selection of queries.

Acknowledgements

I would like to thank my supervisor, 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 pursue my passion in computer science.

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

Produced Work

The code the project, documentation and this report can be found at OliverKillane/emDB

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 attractive interface for describing the structure and interactions 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. For many such ap-

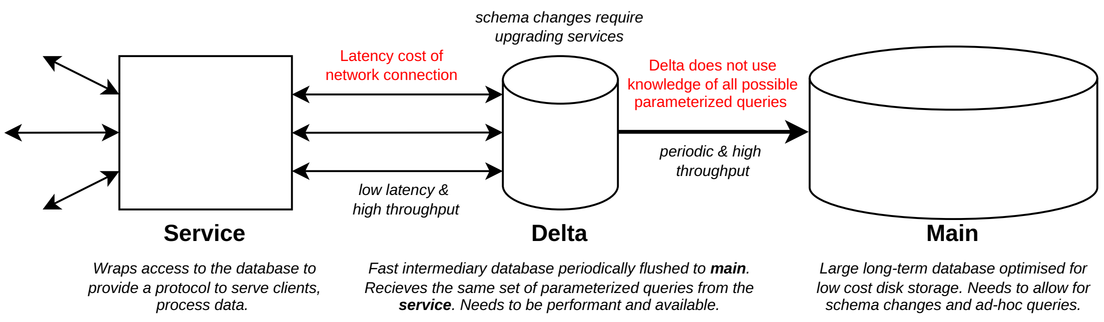

schema changes require upgrading services
Latency cost of network connection
Delta does not use knowledge of all possible parameterized queries
periodic & high throughput
low latency & high throughput
Service
Delta
Main
Wraps access to the database to provide a protocol to serve clients, process data.
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.

Figure 1.1: A slow delta-database.

plications 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.
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. Unfortunately the strongest abstractions are present in embedded databases that do not take advantage of 4 as by design they both support schema changes, and are independent from the host language. For example duckDB is embeddable in Python, Java, Julia (and other) applications.

The typical embedded database design is just an in-memory database, packaged to be in the same process an application. 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 guarantee on the correctness of all queries.

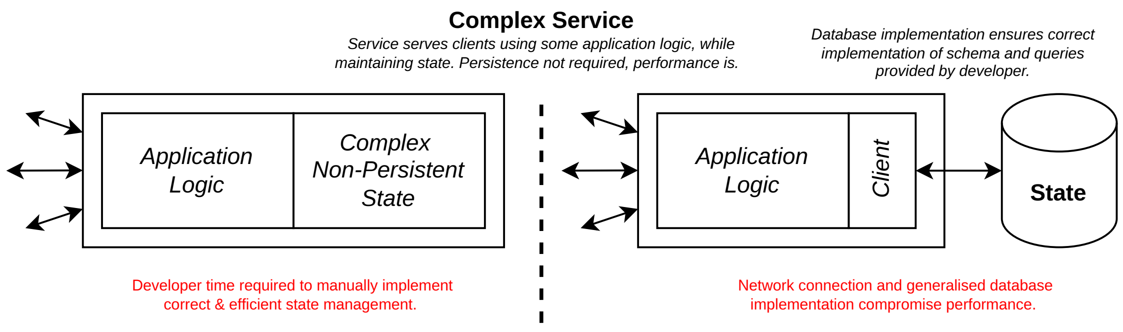

Complex Service
Service serves clients using some application logic, while maintaining state. Persistence not required, performance is.
Application Logic
Complex Non-Persistent State
Developer time required to manually implement correct & efficient state management.
Database implementation ensures correct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

Figure 1.2: A typical complex service.

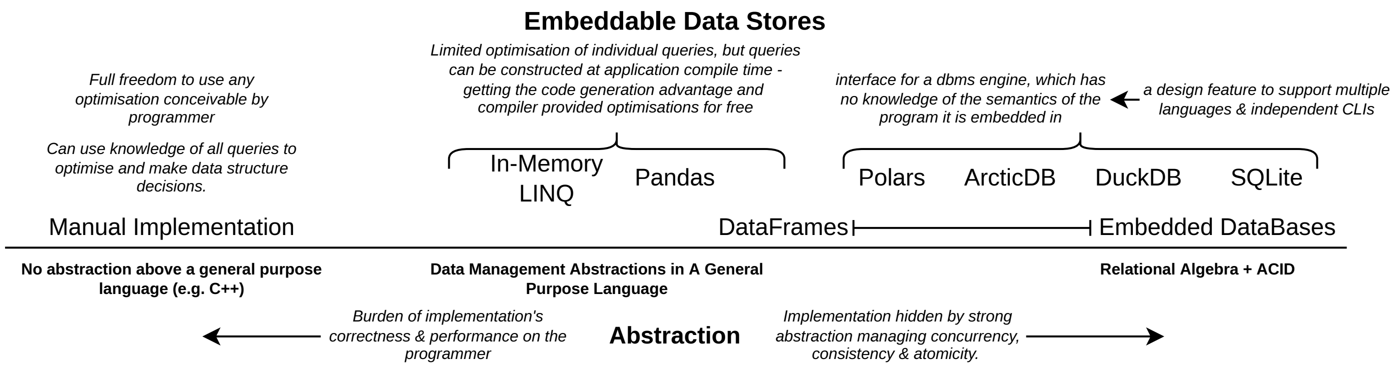

Embeddable Data Stores
Full freedom to use any optimisation conceivable by programmer
Can use knowledge of all queries to optimise and make data structure decisions.
Manual Implementation
Limited optimisation of individual queries, but queries can be constructed at application compile time - getting the code generation advantage and compiler provided optimisations for free
In-Memory LINQ
Pandas
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
Polars
ArcticDB
DuckDB
SQLite
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.

Figure 1.3: A spectrum of data processing abstractions.

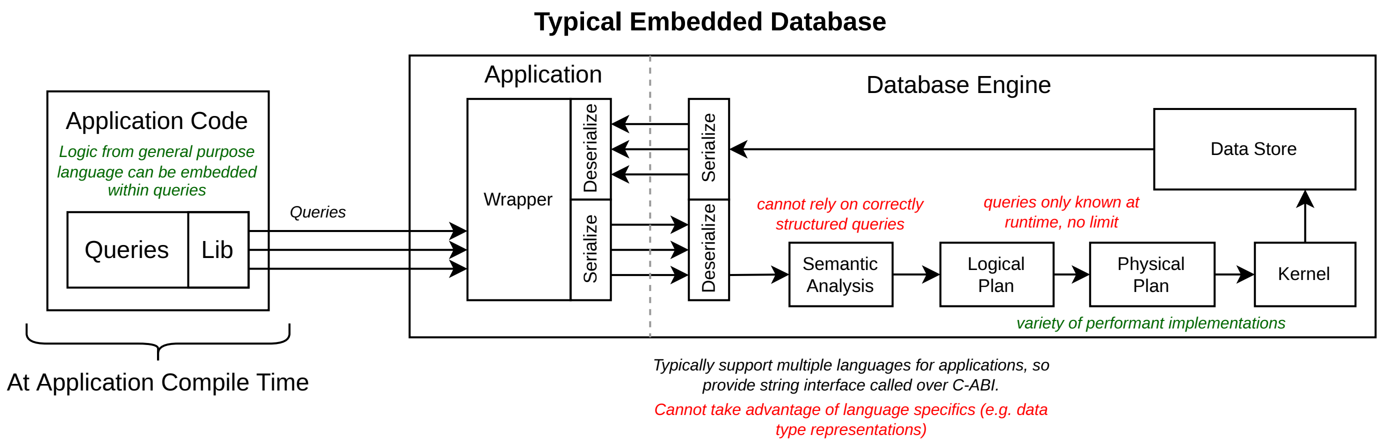

Typical Embedded Database
Application Code
Logic from general purpose language can be embedded within queries
Queries
Lib
At Application Compile Time
Queries
Application
Wrapper
Deserialize
Serialize
Database Engine
Serialize
Deserialize
Data Store
cannot rely on correctly structured queries
queries 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)

Figure 1.4: Typical Embedded Database System

No such ideal system exists, so we shall build it.

1.2 Contributions

emDB A schema-compiler demonstrating significant performance improvements over SQLite and DuckDB on a selection queries.

Combi A performant combinator library used for parsing rust tokenstreams, supporting multiple syntax errors without needing error node in the produced AST.

Chapter 2

Concepts

2.1 Compile Time Generated Database

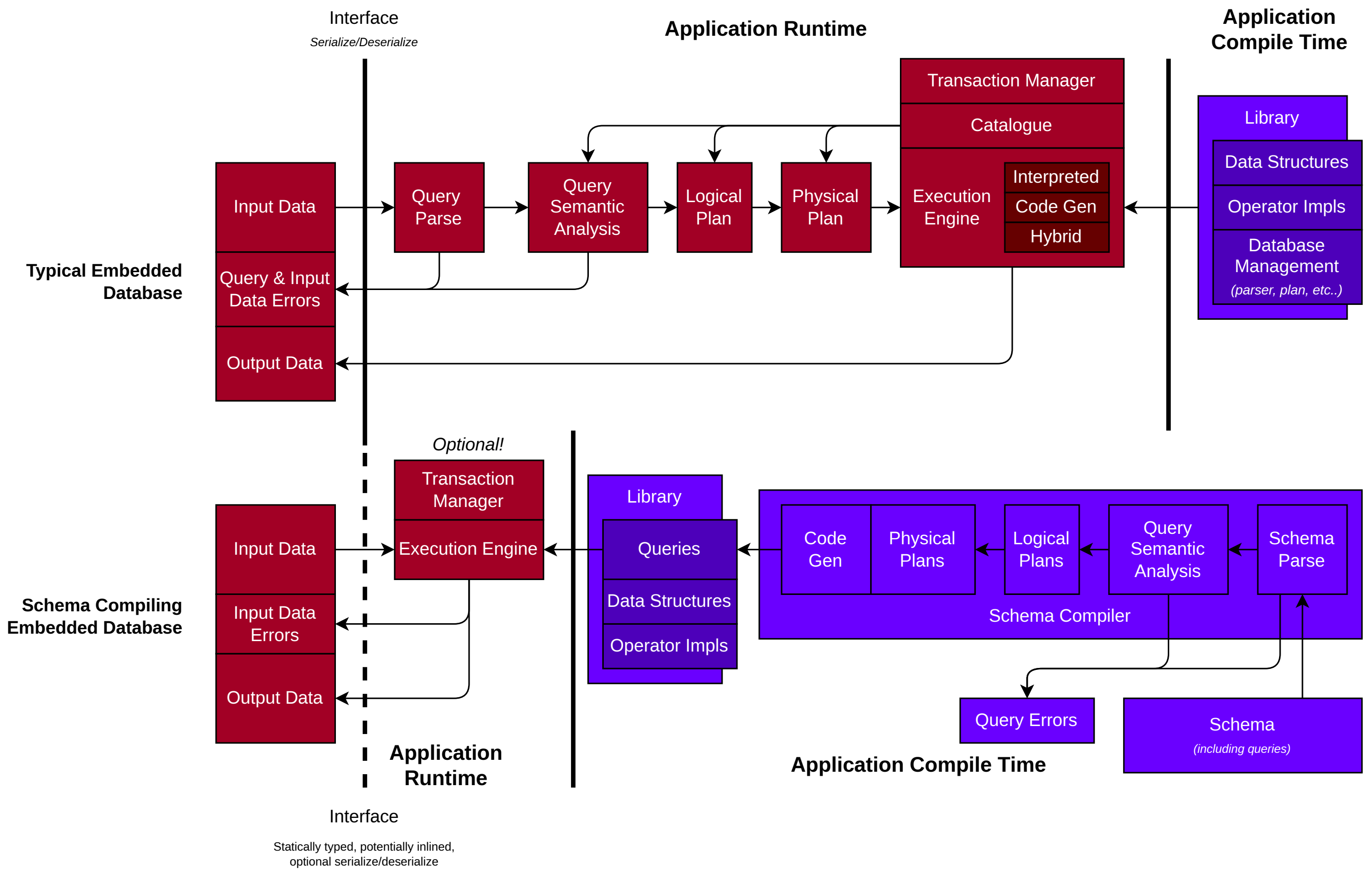


Figure 2.1: Comparison between schema compilation and typical embedded databases

By requiring a static schema, and known parameterized-queries at application compile time, rather than at runtime, we gain the following benefits:

- Logical optimisations on both queries and tables (i.e. data structure selection, index selection) can occur, with the entire context of all possible queries.
- Errors in queries (that are not dependent on the input data, e.g type errors, invalid table access) can be reported at application compile time.
- Time to query parse, analyse and plan is entirely eliminated, this allows more expensive query optimisation to be performed with no cost to the query's execution time. This is beneficial for code generation, which can provide performance benefits, at a large upfront cost (for example degrading performance of small queries for HIQUE[16], and the motivation for generating code at the LLVM IR level (rather than C++) for hyper[22].)

"The parsing of SQL statements is a significant consumer of CPU cycles in any SQL database engine. On-going efforts to optimize SQLite have caused the developers to spend a lot of time tweaking Lemon to generate faster parsers."

– SQLite Website[28]

As the schema is compiled alongside the application, a greater level of integration with the application is possible.

- One of the major benefits touted for embedded databases such as SQlite, DuckDB and monetdb/e is that they provide language agnostic interfaces, allowing them to be portable to embed in applications of many different languages.
- Portability comes at the cost of providing a uniform interface across different languages (SQL strings). Which thus cannot take advantage of semantics specific to the host language (for example move semantics in C++ & Rust, or lifetime bound references in Rust).

By forgoing portability and generating an interface in the host language, we can take advantage of such semantics, and allow for further optimisation (e.g. inlining queries, constant propagation into queries).

It also makes it possible to embed complex logic into the database.

```rust
fn my_complex_logic(borrowed_string: &str) -> bool { /* ... */ }

const SOME_LIMIT: usize = /* ... */;

emql!{
    table some_data { string_field: String, const_string_field: &'static str, /* ... */ }

    // For example a query defined in emQL, using user types and functions, with relational algebra
    query some_query(foo: crate::UserDefinedType, bar: &'qy (String, usize)) {
        use some_data  // scan table
            |> filter(crate::my_complex_logic(string_field) || bar.1 < crate::SOME_LIMIT)
            |> // ... do some aggregations: groupby, join, map, etc.
            ~> return;
    }
    // ...
}
```

2.1.1 Logical Optimisation

No Update - Immutable Values

When values are not updated, can be shared when accessed from the database, without needing to copy, with the caveat that the data returned needs to be wrapped in some safe reference type. There are several options for this:

- Borrows (pointer qualified by the lifetime of the database) are the simplest, but require pointer stability (the data cannot be freed or moved while the reference is valid).
- Reference counting is more flexible, but requires a more complex implementation (including reference counts).
- With an extra level of indirection, reference counting can be applied to mutable values (always allocate a new tuple on update, requires copying over unchanged values from the old allocation).

The advantage of avoiding copies of immutable data is directly tied to the cost of the copy. For small values (i.e. 8 byte and less) this is irrelevant as it is equal or less cost than moving a pointer (8 bytes on 64 bit systems).

No Delete - Append Only Tables

Keys to data structures that allow deletions require checks that append only data structures do not.

1. If the database supports queries returning stable row references / `rowid`, if there are no deletions then no additional data is required other than an index (which can be used to lookup the table), versus requiring a generation count and generation check for tables supporting deletion. DuckDB supports returning unsatble `rowid` (row ids for deleted rows may be later reused)[19], SQLite supports stable `rowid` if the table has an `INTEGER PRIMARY KEY` for rowid tables[27].
2. If the database is using immutable value optimisation (2.1.1), the lack of deletion operations can inform the choice of data structure & hence the type of reference.
3. Negligable cost of checking for free slots in a table can be removed, only need to append.

No Insert - Static Data

If some data is required, but never updated, inserted or deleted. It can be used for optimisations:

- Given a known set of values, *perfect hashing* can be used to accelerate aggregations and joins.

```
table countries {
    name: String, // country names immutable, and no new countries added
    population: usize, // other fields can be updated
    // ...
} // with some static data at the start
```

Limited Table Sizes

For some schemas a bound on the maximum size of a table is known at compile time. This can be used to allocate a fixed size table.

- This can be used to eliminate the need to allocate (e.g. extend vectors, or allocate new blocks). This can be a requirement for memory constrained systems, such as in embedded programming.
- Fixed size buffers have both fast lookup, and pointer stability.

Incremental View Maintenance (IVM)

Views

Recompute on Access

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

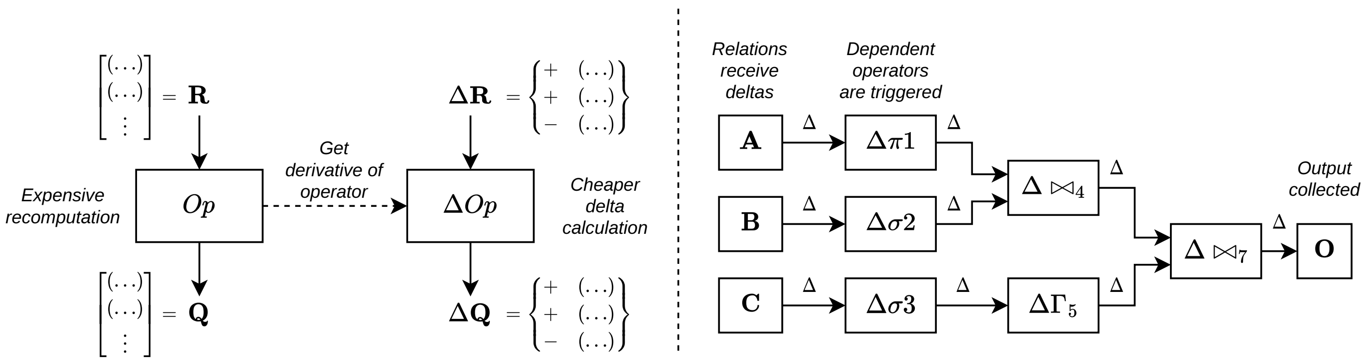
An alias for a query, typically a `SELECT` statement, that can be queried like a table. The view's query is used each time it is accessed.[12]

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. The cost reduction from recomputing operators



Expensive recomputation
$\begin{bmatrix}(\ldots)\\(\ldots)\\\vdots\end{bmatrix} = \mathbf{R}$
Op
$\begin{bmatrix}(\ldots)\\(\ldots)\\\vdots\end{bmatrix} = \mathbf{Q}$
Get derivative of operator
$\Delta\mathbf{R} = \begin{Bmatrix}+ & (\ldots)\\+ & (\ldots)\\- & (\ldots)\end{Bmatrix}$
ΔOp
$\Delta\mathbf{Q} = \begin{Bmatrix}+ & (\ldots)\\+ & (\ldots)\\- & (\ldots)\end{Bmatrix}$
Cheaper delta calculation
Relations receive deltas
Dependent operators are triggered
A
B
C
Δ
$\Delta\pi 1$
$\Delta\sigma 2$
$\Delta\sigma 3$
$\Delta \bowtie_4$
$\Delta\Gamma_5$
$\Delta \bowtie_7$
Output collected
O

Figure 2.2: Incremental View Maintenance as a Calculus

to recomputing the deltas of operators can also be applied recursively. DBToaster (5.3.1) generates IVM C++ code for a schema of select queries and input streams. As a result it cannot support complex mutation:

```
UPDATE .. WHERE ( COMPLEX EXPRESSION )
```

There is no fundamental barrier to an IVM system supporting this.

IVM performance can benefit from restrictions of the operations that can occur on tables.

2.1.2 Physical Optimisation

Profile Guided Optimisation

Given the queries are known, they can be instrumented to collect statistics at runtime, for use in subsequent compilations.

- These features are already available for many languages, by compiling the schema to a language that supports profile-guided optimisation this feature is atained for free.

Ownership Transfer

Given the host language supports move-semantics, queries can *move* arguments. Moving owning objects only requires a copy of the data structure, and not a deep copy of data (i.e. heap allocated) owned by it.

- Ensuring the data is not accidentally mutated after ownership is given to the database, or mitigating ease of doing this is a memory safety requirement for the database.

User Defined Types

Rather than enforcing users to serialize data for use in the database, or rely on general implementations for copying, hashing &

- Removal of serialization can benefit performance, particularly when combined with ownership transfer2.1.2.

Chapter 3

Implementation

3.1 Tool Choice

3.1.1 C++ with a Separate Tool

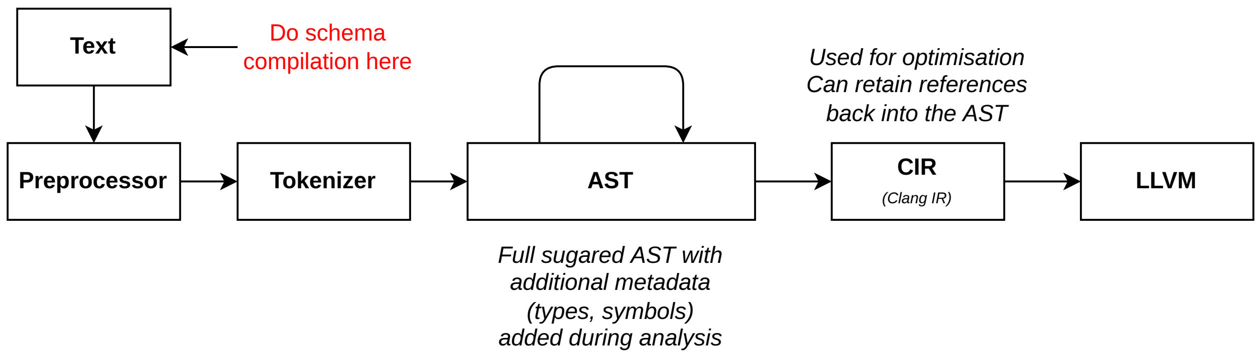

Text
Do schema compilation here
Used for optimisation
Can retain references back into the AST
Preprocessor
Tokenizer
AST
CIR
(Clang IR)
LLVM
Full sugared AST with additional metadata (types, symbols) added during analysis

Figure 3.1: Clang Compiler Pipeline

Due to the lack of unification on build systems, and the lack of support for abstractions powerful enough to do schema compilation at compile time in C++, an external tool invocable on schema files is required.

- C++ has powerful metaprogramming capabilities for building the data structures to be used by the database implementation.
- Substantial tooling (profilers, debuggers) available.
- Some analysis of embedded C++ code can be done using clang libtools (much like clang static-analyzer & clang tidy), but propagation of error messages is difficult.
- Without additional tools being developed, language server/IDE support is impossible.

3.1.2 Zig Comptime

Zig supports compile time reflection and code execution. It is a far more powerful and complete implementation than those currently present in C++23'S `constexpr` &`consteval`, or Rust's `const` generics.

comptime	Types are values, can be assigned, passed, and have data structures, functions generated, can pass errors to the compiler. Restrictions on IO.
runtime	Types erased, access only to non-comptime values.

Rather than parsing, we allow users to construct comptime structs that describe the operations of queries, and tables.

At comptime these can be passed to a function, which analyses the structure and can then generate a structure that implements the database.

- Limitations on flexibility, produced plans need to be constructed as valid zig datastructures, and implement some method for *execute given query*.
- Limited interaction with non-zig code, for example generating plan debug diagrams, or representations that can be easily debugged.

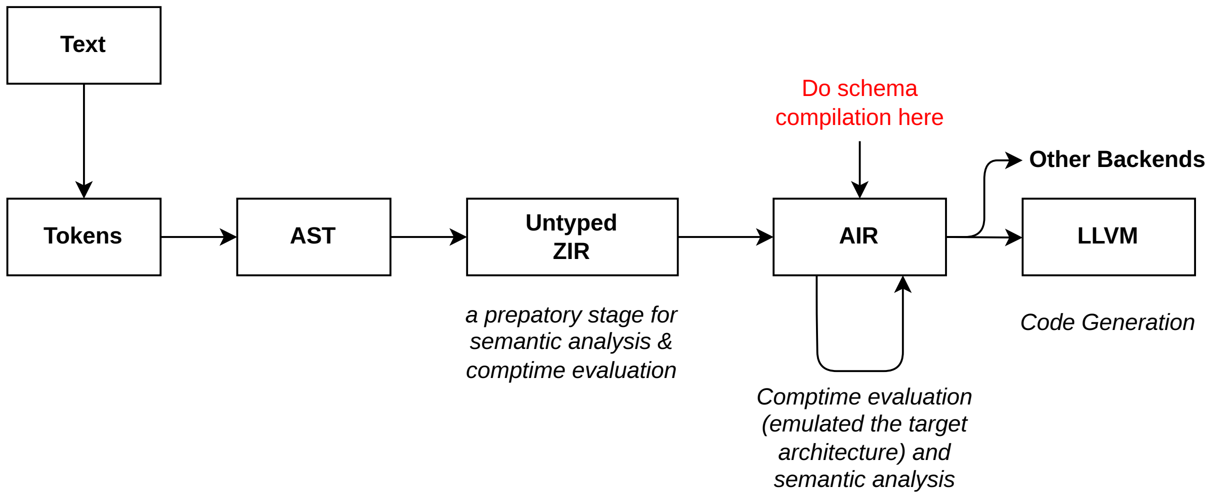

Text
Tokens
AST
Untyped ZIR
a prepatory stage for semantic analysis & comptime evaluation
Do schema compilation here
AIR
Comptime evaluation (emulated the target architecture) and semantic analysis
Other Backends
LLVM
Code Generation

Figure 3.2: Zig Compiler Pipeline

- Error messages can be made by the library, but are managed by zig, so while expression errors are trivial and managed entirely by zig, assigning spans to certain structures using the query language semantics is difficult.

3.1.3 Rust Procedural Macros

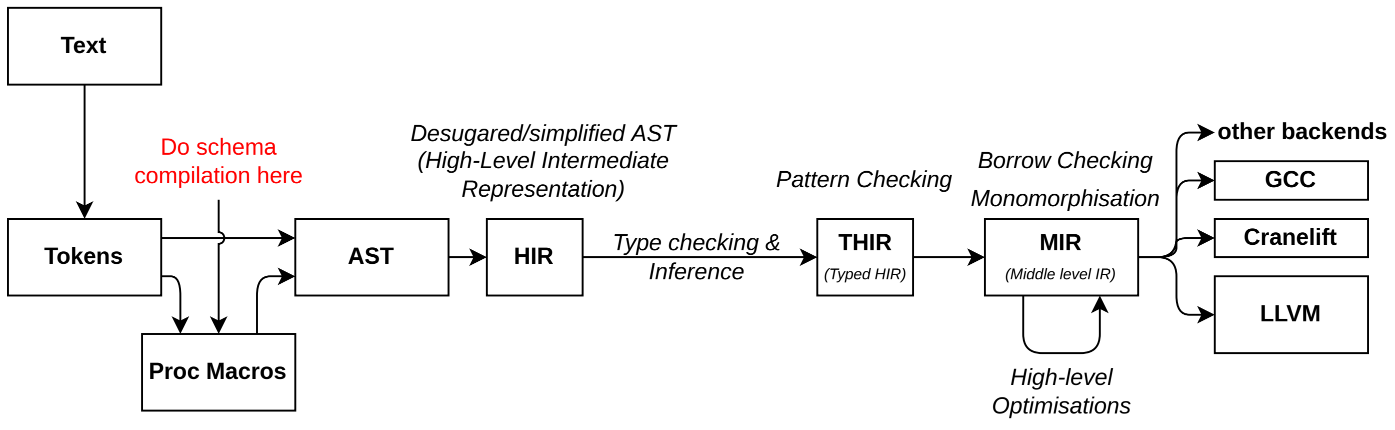


Figure 3.3: Rust Compiler Pipeline

Procedural macros are functions which take rust compiler provided tokenstreams as input, and return rust tokenstreams as output and run at compile time. They can interact with the compiler, and the environment (file system, network).

- They are a key language feature of rust, and are ubiquitous across the rust standard library and ecosystem.
- They can contain arbitrary code, which includes a schema compiler.
- The tokens provided to proc macros are basic tokens (identifiers/words, punctuation, literals, brackets), and do not have to conform to rust syntax.

Rust proc macros can also generate error messages and output code, using spans from the original tokens.

- Rust reports error diagnostics from rust, alongside errors generated by procedural macros (meaning they are also passed to the language server and displayed in IDE).
- User code passed through to generated code by macros retain their original spans, so errors can attributed to user code in their original position.

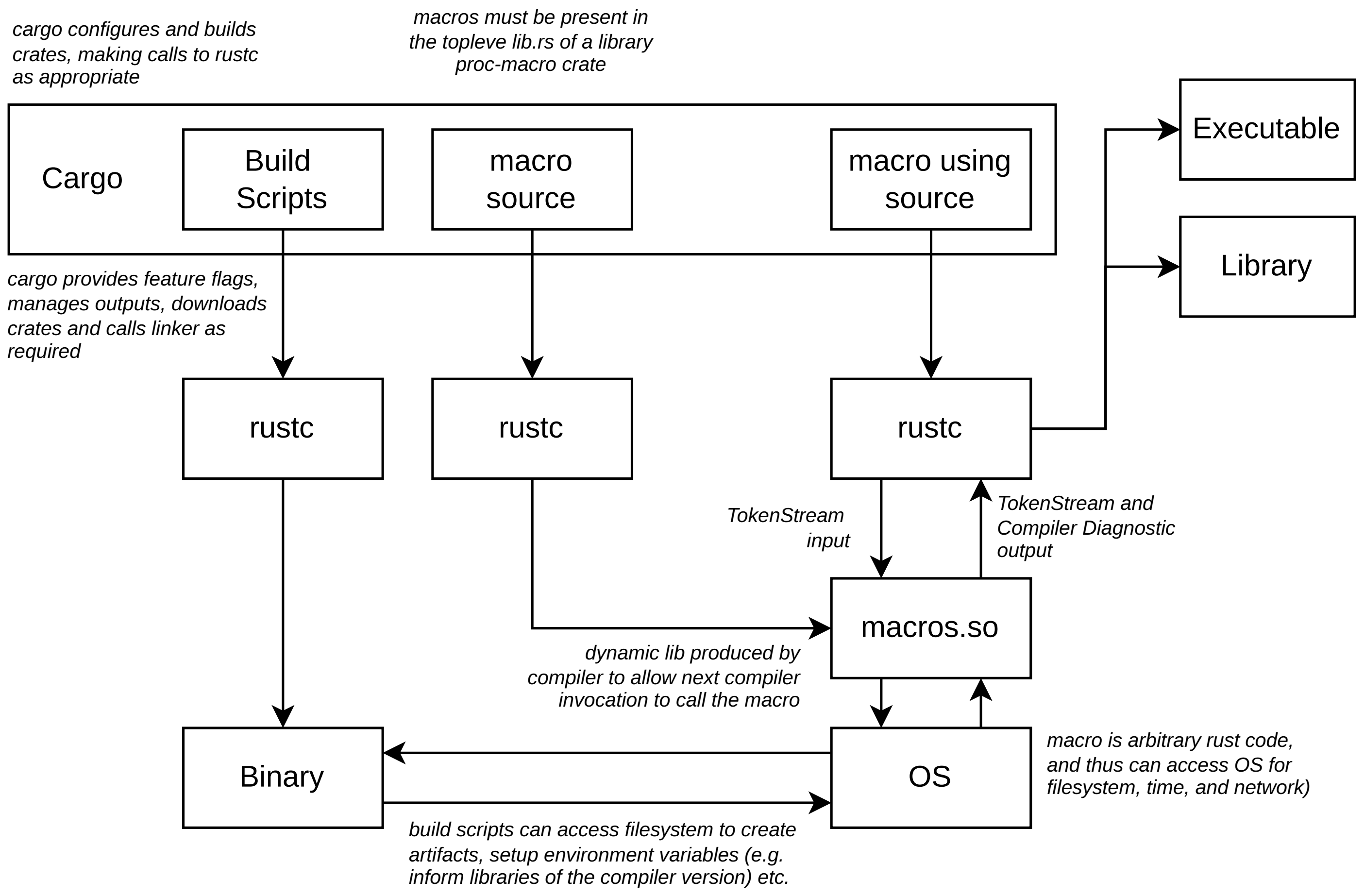

cargo configures and builds crates, making calls to rustc as appropriate
macros must be present in the topleve lib.rs of a library proc-macro crate
Cargo
Build Scripts
macro source
macro using source
Executable
Library
cargo provides feature flags, manages outputs, downloads crates and calls linker as 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.

Figure 3.4: Procedural Macros in the Rust Build System

3.1.4 Decision

		Tool			
	Feature	zig	rust	tool → cpp	scala
	Custom Syntax	No	Yes	Yes	Yes
	Error Propagation	Yes	Yes	No	Yes
Requirement	Performance	Good	Good	Good	Poor

- Rust's support for lifetime-bound references is particularly advantageous in the context of returning references to immutable data.
- Rust's safety features make generating code far safer than zig, or C++.
- Existing support for parsing rust ASTs allow for easier development of the schema compiler.
- Zig does not allow IO at comptime, so generating diagrams, or other debug information separate from code is difficult.
- Other languages supporting code generation (in particular scala 3 macros) were not considered due to lacking performance (garbage collection).

Hence the chosen option was a rust procedural macro.

3.2 System Overview

Given the limited time, and the requirement to complete the entire end-to-end compilation in order to test optimisations:

- Implement only the append only (2.1.1) & immutable value (2.1.1) optimisations.
- Build the system to be easily extendable (need to support multiple backends, and have a backend-agnostic logical plan).
- Rather than implementing and optimising a small subset of functionality, to implement all functionality required for a database and use the remaining time for optimisation.

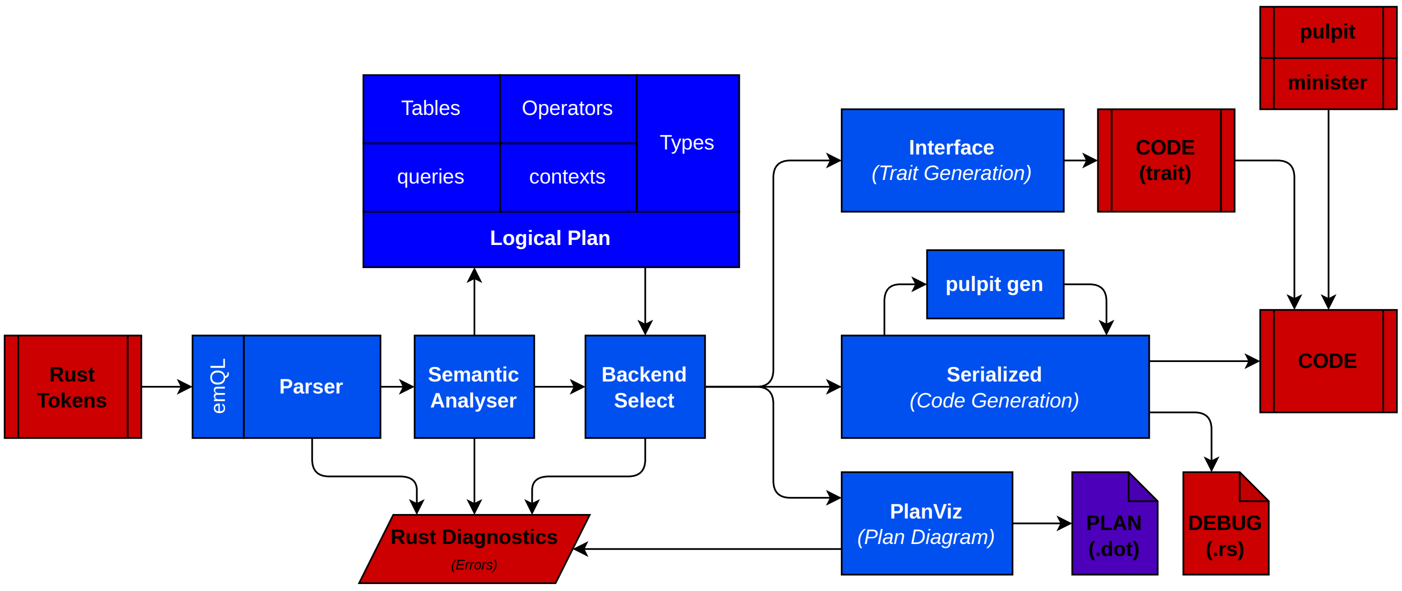


Figure 3.5: System Design

3.3 Language Design

In deciding the query language for any database, SQL is the default choice, however it has several disadvantages in the context of an embedded (in host language) database.

		Language			
	Feature	SQL	SQL rustified	flux	custom
	SQL Compliant (Tooling)	Yes	No	No	No
required	Compatible with rust types	No	Yes	No	Yes
required	Compatible with rust expressions	No	Yes	No	Yes
	Rust Syntax Shadowed Highlighting	No	No	Yes	Yes

Embedding rust expressions and types into the database avoids a costly (query language) $\rightarrow$ rust translation requirement, and allows for simple embedding of application code. Given the difficulties with a SQL-based language design, and few upsides when compared to the flexibility in designing a custom language, a custom language was chosen (emQL).

3.3.1 emQL Overview

The emQL language developed for this project is inspired by both the flux language syntax, and by iterators (specifically the `|>` elixir pipe).

- Queries are directed acyclic graphs (DAGs). Unlike SQL which represents queries (with some exceptions[1]) as trees, emQL allows for streams to me merged, forked, and have stream elements be individually lifted into a subcontext (another DAG inside a query DAG).
- Operators are ordered by dataflow (`op1 |> op2 |> op3`), rather than by operation (SQL requires a rigid ordering of `SELECT .. FROM .. WHERE ..`). This means that queries can be written in a more concise, and more intuitive imperative style.
- Row references are a core concept in emQL. They are fast lookup keys into tables that are used in `update` and `delete` operations and are provided by `insert`, `ref` and `unique` operations. As row references are just data, they can be mapped, processed and acted on by user logic, and are stable so can be sent & stored outside the database for use in fast lookups later. Row references are a significant performance advantage for OLTP workloads that allow the database to select a unique identifier.
- Mutable and ure operations mixed. Rather than requiring separate `SELECT` and `UPDATE` operations, emQL allows for a single query to both read and write data in a single pipeline of operators on a stream. This allows for more concise queries.
- All queries are just transactions. Though they are called `query`, emQL queries are equivalent to SQL transactions with potentially many operations.

[1]Recursive Common Table Expressions

- Reference types are explicit at accessed using `'qy` and `'db`. The backend chooses the types provided by `deref` and `use` during table selection, and can optimise these. The chosen types are exposed to the user so they can decide to for example (use `Cow` on an immutable reference rather than cloning when mapping with mutation).

For example the following schema is valid emQL from .

```
impl my_db as Serialized {
    // options can be specified for implementations
    aggressive_inlining = on,
};

table people {
    name: String,
    birth_year: u16,
    friend: Option<String>,
} @ [
```

Constraints can be applied for unique columns, table length limit and row predicates. The provided names are used in the error return types on constraint breach

```
    unique(name) as unique_names,
    limit(1000) as max_people,
    pred(name.len() < 100) as sensible_name_length
]
```

Each query is expressed as a method on the generated database object

- The self borrow is determined by the mutation operations in the query
- The return type is inferred from the query, and is free for the backend to make optimisation decisions on.

```
// DESCRIPTION: Get the number of friendships between people born in the same decade
query same_year_friendships() {
    use people
        |> map(
            name: &'db String = name,
            birth_decade: u16 = (birth_year / 10) * 10,
            friend: &'db Option<String> = friend
        )
        |> groupby(birth_decade for let same_year_people in {
            // Queries are not nested like SQL, but are a DAG, streams can be duplicated.
            use same_year_people |> fork(let friend_side, friended_side);

            join(use friend_side [
                // predicate (inner), cross and equi (inner) joins are supported
                inner pred {
                    if let Some(friend_name) = &left.friend {
                        friend_name == right.name
                    } else {
                        false
                    }
                }
            ] use friended_side)
                |> count(num_friendships)
                ~> map(decade: u16 = birth_decade, num_friendships: usize = num_friendships)
                ~> return;
            }
        )
        |> collect(decades)
        ~> return; // The return type is inferred and includes the errors that can occur.
}

// DESCRIPTION: update an incorrect birth year using a name, and return a row reference
query fix_birth_year(name: &'qy String, new_year: u16) {
    row(
        // using a reference type for input ('qy is the lifetime of the borrow of the database)
        name: &'qy String = name,
    )
        ~> unique(name for people.name as ref person) // access to columns with unique indexes
        ~> update(person use birth_year = new_year)
        ~> map(person: ref people = person) // use of a row reference type
        ~> return;
}

// DESCRIPTION: Add a new person, and return the number of people who declare them a friend.
query new_friendship(user_name: &str, friend: Option<String>) {
    row(
        name: String = String::from(user_name),
        friend: Option<String> = friend,
        birth_year: u16 = crate::CURRENT_YEAR,
    )
        ~> insert(people as ref new_person_id)
        ~> let person_id; // streams and singles can be assigned to (single read) variables.
```

The lift operator opens a new context, with the stream/single's record fields available to use in expressions, here it means we can refer to `new_person_id` in an expression, while also using `user_name` from the query's top context

```
    use person_id
        ~> lift(
            ref people as other_person_id
                    // access with a row reference to the column `friend`
                |> deref(other_person_id as other_person use friend)
                |> filter(
                    if let Some(other_name) = &other_person.friend {
                        other_name == user_name && *other_person_id != new_person_id
                    } else {
                        false
                    })
                |> count(current_frienders)
                ~> return;
        ) ~> return;
}
```

Future Work: Stream in and out

emQL queries are currently limited to taking in values (singles) and returning single values.

- Allowing queries to consume streams requires modifying the interface to take a struct argument with the specified fields.
- Returning streams without collecting values is more difficult, and requires allowing records to contain a variable number of generic parameters for the types of contained streams, implementing some trait (for example `impl Iterator<Item=T>`).

This feature would remove the requirement for immediate collection before return (some potential performance benefits if users intend to iterate over results), and for flattening nested streams (rather than collecting to `Vec`, and then concatenating them - expensive).

Future Work: Subqueries

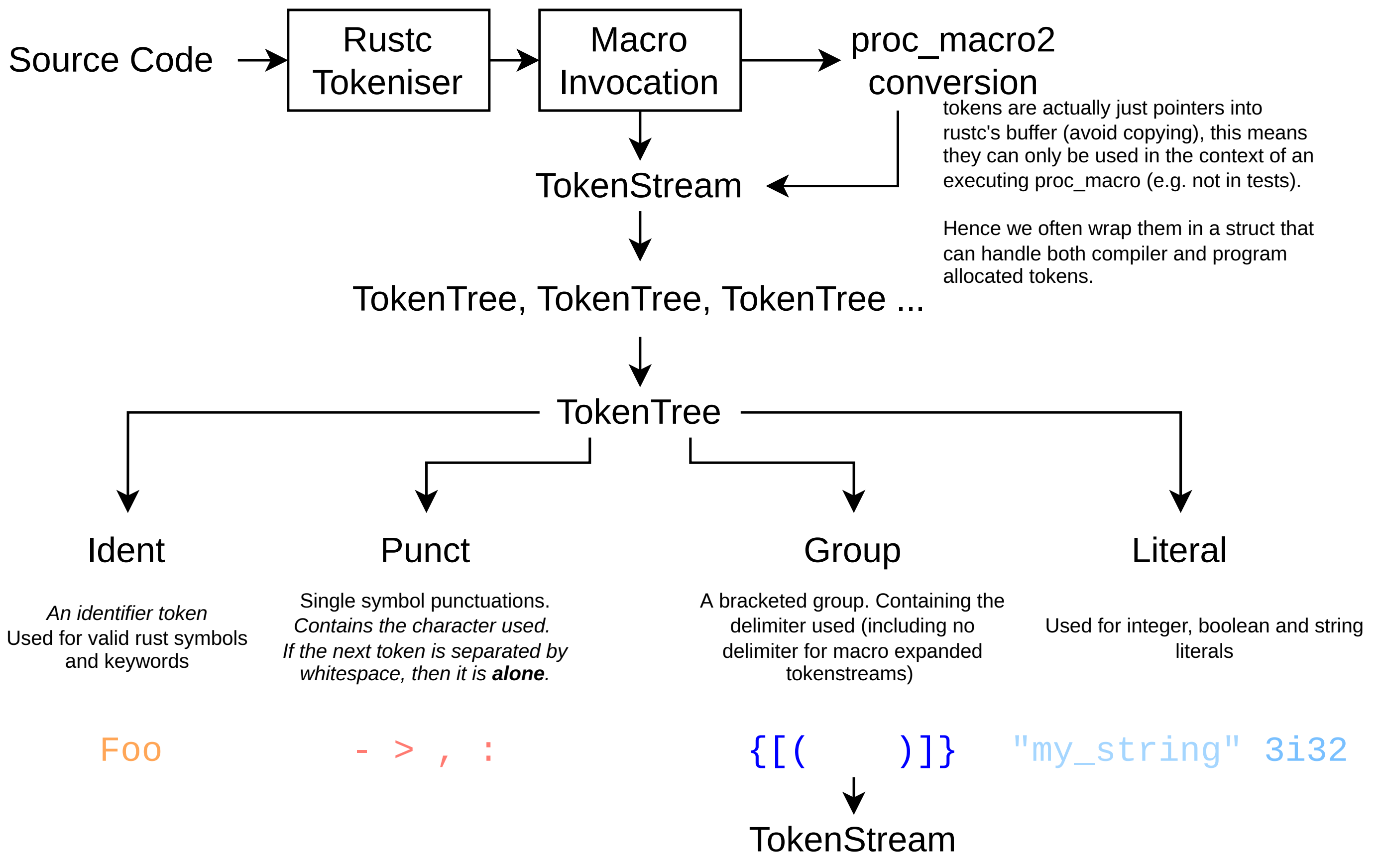
Subqueries can be enabled by connecting a `call` operator to the context used by the other query. The semantic analysis however requires the called query's return type to be known. This requires significant modification to the current semantic analyser to track incomplete analyses, and must include a restriction on recursive (or even mutually recursive) queries.

3.4 Query Parsing with Combi

3.4.1 Rust Tokenstreams

Rust procedural macros can access tokens provided by the rust compiler through the compiler provided `proc_macro` crate.

- The `proc_macro` crate provides a stable interface for tokens which are indexes into an AST held in compiler memory.
- The `proc_macro2` crate provides a wrapper for `proc_macro` types to allow them to be used outside of a procedural macro context (for example in unit tests).



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

Figure 3.6: Rust Tokenstreams

Each tokentree and tokenstream is associated with a `Span` which tracks its original file & position, and can be used in generating diagnostics (spans used to highlight in error messages, and by the language server for highlighting in IDE).

3.4.2 Tokenstream Parsers

The parser used needs to either unfold the tokenstream, or be able to parse tree data structures.

- Unfolding the tree is expensive, and can loose some information (for example brackets are associated with spans for open close)
- Parsing a tree is not supported by most parser generators, and requires carful management of spans (for example reporting parent error spans when descending into an empty tokenstream).

One additional requirement for emQL was to be capable of producing multiple syntax errors, more errors ensures a better development experience. Currently there are only a few parser combinator libraries that can complete this task:

		Parser			
	Feature	Nom	Chumsky	Chumksy-proc	Winnow
required	Multiple Syntax Error	Manual	Yes	Yes	Manual
required	Tokenstreams Parsable	Yes	Yes	Yes	Yes
	Tokenstream Combinators	No	No	Yes	No
	Actively Maintained	Yes	Yes	No	Yes
	Performance	Good	Poor	Poor	Good

The only library meeting requirements was chumsky-proc, rather than modify and build atop this library I decided to go for a custom solution, *Combi*.

3.4.3 Combi Combinators

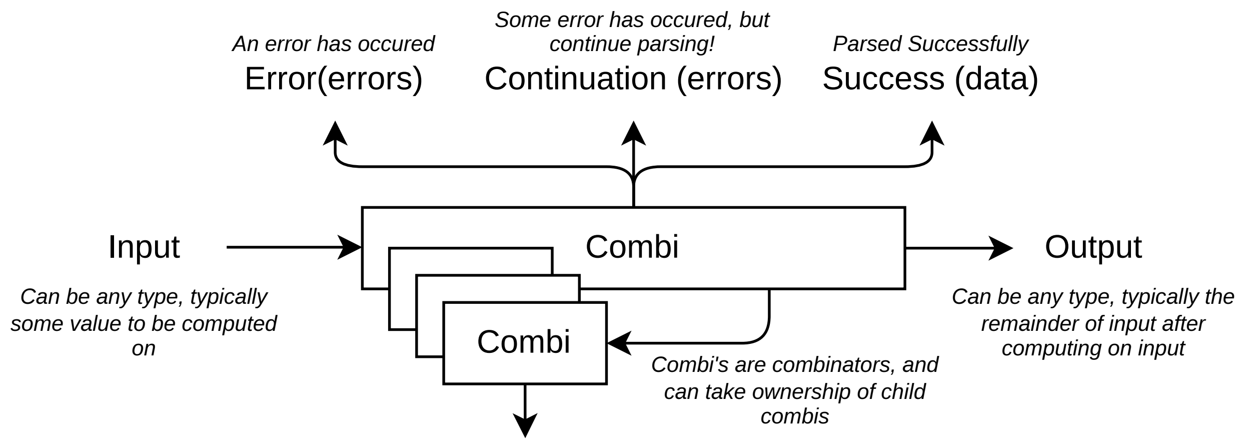

An error has occured
Error(errors)
Some error has occured, but continue parsing!
Continuation (errors)
Parsed Successfully
Success (data)
Input
Can be any type, typically some value to be computed on
Combi
Combi
Output
Can be any type, typically the remainder of input after computing on input
Combi's are combinators, and can take ownership of child combis

Figure 3.7: Combi Combinators

The core trait of the library is Combi, it is used to express parsers and recovery mechanisms.

```
pub trait Combi {
    type Suc;
    type Err;
    type Con;
```

Unlike typical parser combinator libraries, the input and output types are independent. This allows for combinators such as `terminal` which convert `TokenStream` → `()`

```
    type Inp;
    type Out;
```

Apply the computation encoded by the `Combi` on some input data.

```
    fn comp(&self, input: Self::Inp) -> (Self::Out, CombiResult<Self::Suc, Self::Con, Self::Err>);
```

A helper method for describing a combinator (useful in debug messages and for expected error output).

```
    fn repr(&self, f: &mut Formatter<'_>) -> Result<(), Error>;
}
```

Rust token parsing combinators are implemented atop a group of generic core combinators (such as `seq` (sequence two combis), `nothing`, `mapsuc`(map a combinator's success value) and `recursive`).

- Additional derived parsers include `listseptrailing`, and the `Fields` builder which generate combinators that parse a set of options (with embedded semantic checks for duplicate, optional, must and default choices), without needing to implement any additional Combi combinators.
- Error embellishment is also included for wrapping errors in expectation messages, based on the structure of Combis, and for rewriting

3.4.4 Performance

A generated tokenstream (see repository for details) was applied to each parser. All parsers should succeed for all inputs however they have different error capabilities.

Handrolled	Panic on failure. Best possible performance at the cost of any error generation.
Chumsky-Proc	Attempt to parse, includes error generation, no recovery.
Combi	Full error recovery for brackets / multiple syntax errors for brackets, includes error generation.

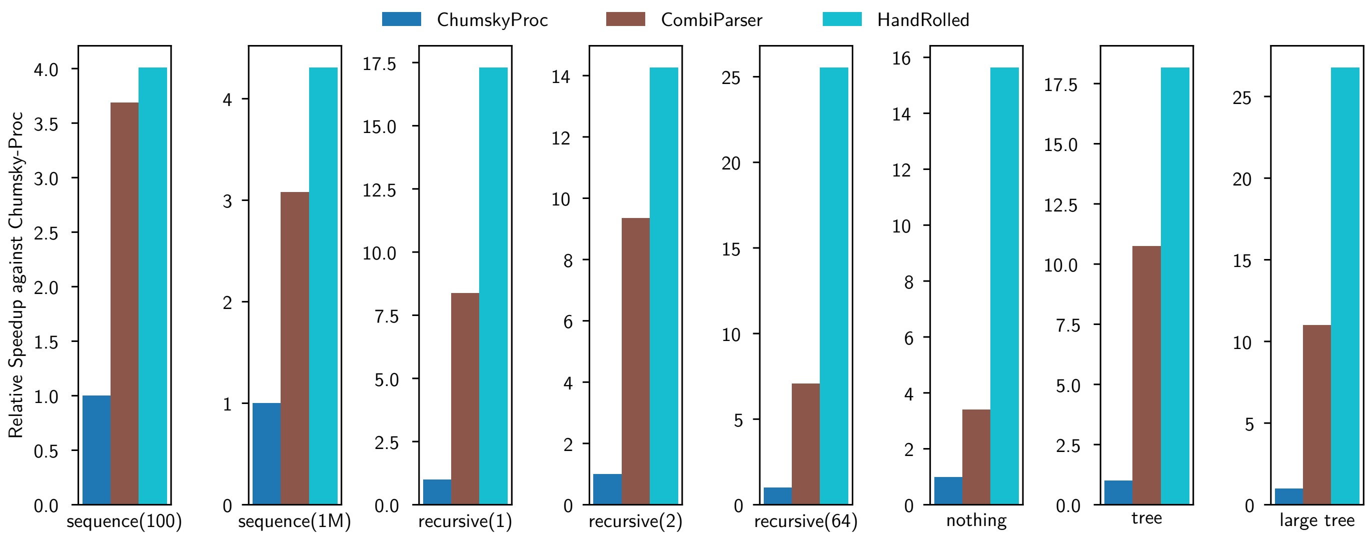


Figure 3.8: Comparative Benchmarks

Unsurprisingly the best performance is for the hand-written, panicking parser. However Combi demonstrates a significant performance advantage over Chumsky-Proc.

Combi's versatility is evidenced by its use for the emDB's emQL frontend, pulpit's table generation macro, and enumtrait's option parsing.

3.5 Table Generation with Pulpit

3.5.1 Window Pattern

In order to provide mutability optimisation, we need to be able to provide a safe interface to store.

Immutable	Borrows can last lifetime of object, and are independent from borrows of the mutable side
Mutable	Normal borrow rules apply

In order to get a lifetime with which to bind references to the immutable part, we use a *window pattern*.

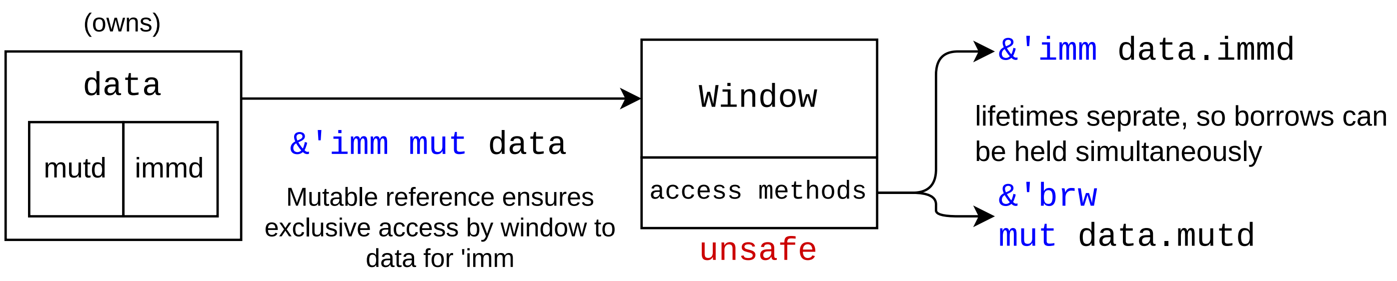

(owns)
data
mutd
immd
&'imm mut data
Mutable reference ensures exclusive access by window to data for 'imm
Window
access methods
unsafe
&'imm data.immd
lifetimes seprate, so borrows can be held simultaneously
&'brw
mut data.mutd

Figure 3.9: Window Pattern

Valid Simultaneous Borrows

```rust
let mut val = Value{imm_data, mut_data};
let mut window = ValueWindow::new_window(&mut val);
let (imm_ref, imm_mut_ref) = window.brw();
let mut_ref = window.brw_mut();

let imm_available = imm_ref; // still available
let mut_available = mut_ref; // new mutable taken over from imm_mut_ref
```

Conflicting borrows on the mutable side

This demonstrates borrow checking working properly still, but for the mutable member.

```
let mut val = Value{imm_data, mut_data};
let mut window = ValueWindow::new_window(&mut val);
let (imm_ref, imm_mut_ref) = window.brw();
let mut_ref = window.brw_mut(); // ERROR! borrow of mut_ref not possible as imm_mut_ref used later

let value_imm = imm_ref; // still available
let old_mut_unavailable = imm_mut_ref;
```

No Dangling references

```
let imm_ref_dangling;
{
    let mut val = Value{imm_data, mut_data};
    // ERROR! needs to borrow long enough for imm_ref_dangling, but val does not live that long
    let mut window = ValueWindow::new_window(&mut val);

    let (imm_ref, imm_mut_ref) = window.brw();
    let mut_ref = window.brw_mut();

    imm_ref_dangling = imm_ref;
}
let imm_ref_dangling_unavailable = imm_ref_dangling;
```

3.5.2 Table Structure

Flexible Design	Can be easily configured to support new column data structures, and new selectors.
Row vs Column Store	For emDB an n-ary selector is used, however it is also possible to configure selectors to store data separately from the keys, or to spread data across several columns (decomposed storage).
Transactions	Required for emDB queries, a simple rollback log is used. Due to the support for row references, deletion operations do not drop data or free indices (just ide the row) until commit if transactions are enabled to prevent insertions during the commit reusing indices.
Precise Errors	The return types of methods are constrained to only include possible errors (rather than a generalised error type). This allows for easy matching on errors, and for some operations (such as insert) to have their error type removed when no constraints can be violated.

Side Effects of Clone

When using a `get`, to get a mutable value from a column the value is cloned.

- Pulpit generates clones only for these columns (and as a result needs a separate `Ok(Values { .. })` return type for each get method).
- Rust does not eliminate redundant clones for unused values unless the clone is inlined into the scope where the value is used, and it is able to determine the clone has no side effects. This conservative approach is in contrast with more aggressive systems languages such as C++, which allow some optimisations that cause observable side effects (e.g. copy elision[6]).
- Given the ability to store **any** `Sized` rust types in emDB, a user may implement a side-effecting clone.

We can see this in a comparison of generated code for different selects, when streaming out values, the copy implementation does not have its `clone` of the value (a string) eliminated, decreasing performance.

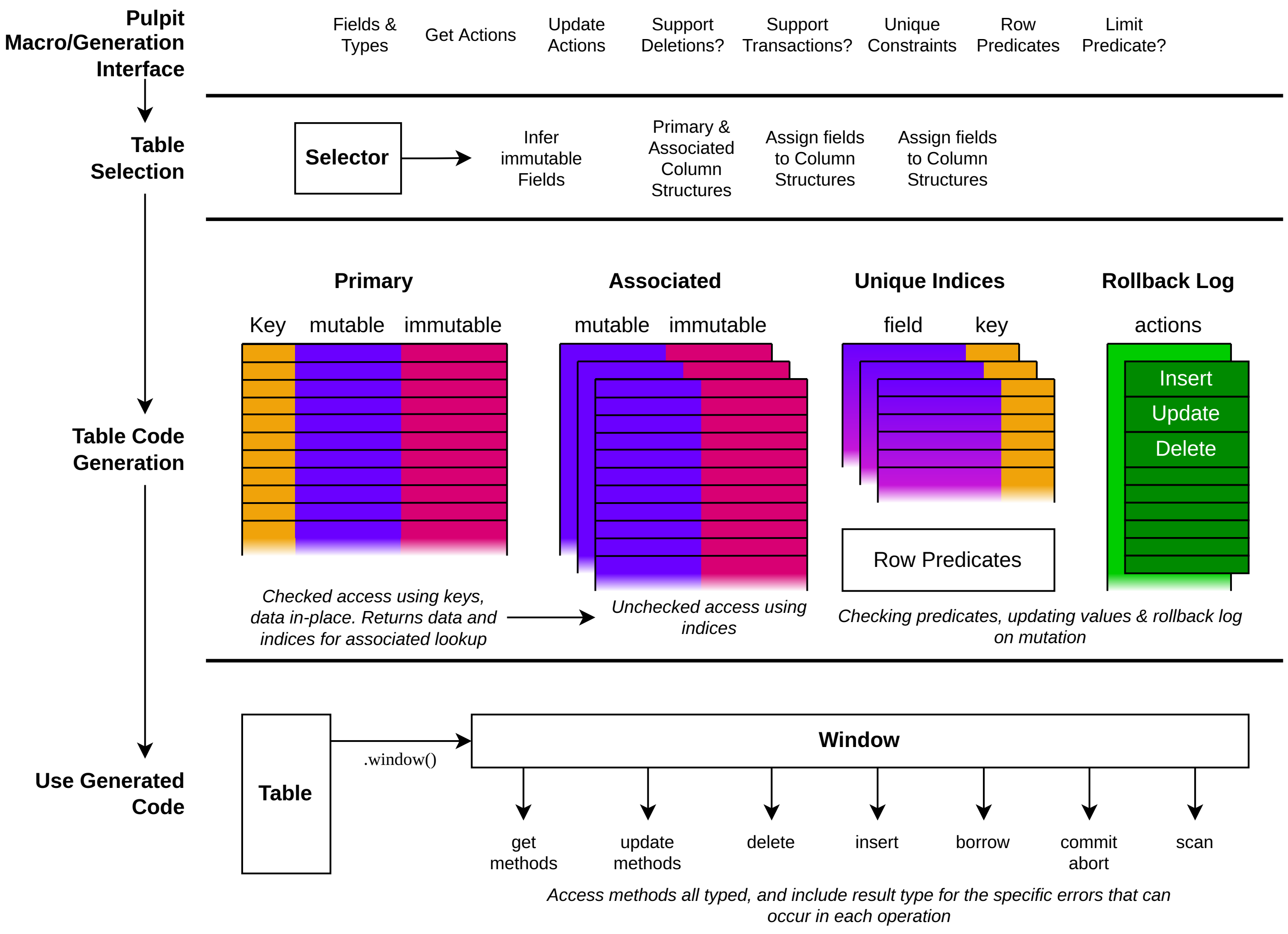

Pulpit Macro/Generation Interface
Fields & Types
Get Actions
Update Actions
Support Deletions?
Support Transactions?
Unique Constraints
Row Predicates
Limit Predicate?
Table Selection
Selector
Infer immutable Fields
Primary & Associated Column Structures
Assign fields to Column Structures
Assign fields to Column Structures
Table Code Generation
Primary
Key
mutable
immutable
Associated
mutable
immutable
Unique Indices
field
key
Rollback Log
actions
Insert
Update
Delete
Row Predicates
Checked access using keys, data in-place. Returns data and indices for associated lookup
Unchecked access using indices
Checking predicates, updating values & rollback log on mutation
Use Generated Code
Table
.window()
Window
get methods
update methods
delete
insert
borrow
commit abort
scan
Access methods all typed, and include result type for the specific errors that can occur in each operation

Figure 3.10: Pulpit Table Generation

EmDBCopyIgnore and EmDBRefIgnore

```
query count_values() {
    use values // stream out values
        |> map(unrelated_value: () = ())
        |> count(count)
        ~> return;
}
```

EmDBCopy and EmDBRef

```
query count_values() {
    use values as ()
        |> map(unrelated_value: () = ())
        |> count(count)
        ~> return;
}
```

3.5.3 Retaining Values

For some data workloads it is possible to support deletions while also returning references for immutable data, if the immutable data is retained on deletion.

Cost of accumulated, potentially unused memory   versus   Cost of alternative wrapping (copy, reference counting)

For some workloads this leakage is acceptable (e.g. low deletion rate, short lived jobs (data cleaning, storing internal data in compilers)). To take advantage of this, pulpit contains a novel `PrimaryRetain` arena that uses this.

- Immutable data is stored separately in blocks, to allow for pointer stability.
- The mutable data (and associated immutable data pointer) are stored in a large vector (fast lookup, at cost of reallocation for some extensions).
- The immutable data pointer is used as a generation counter in keys (and hence row references for enclosing tables) for the column.

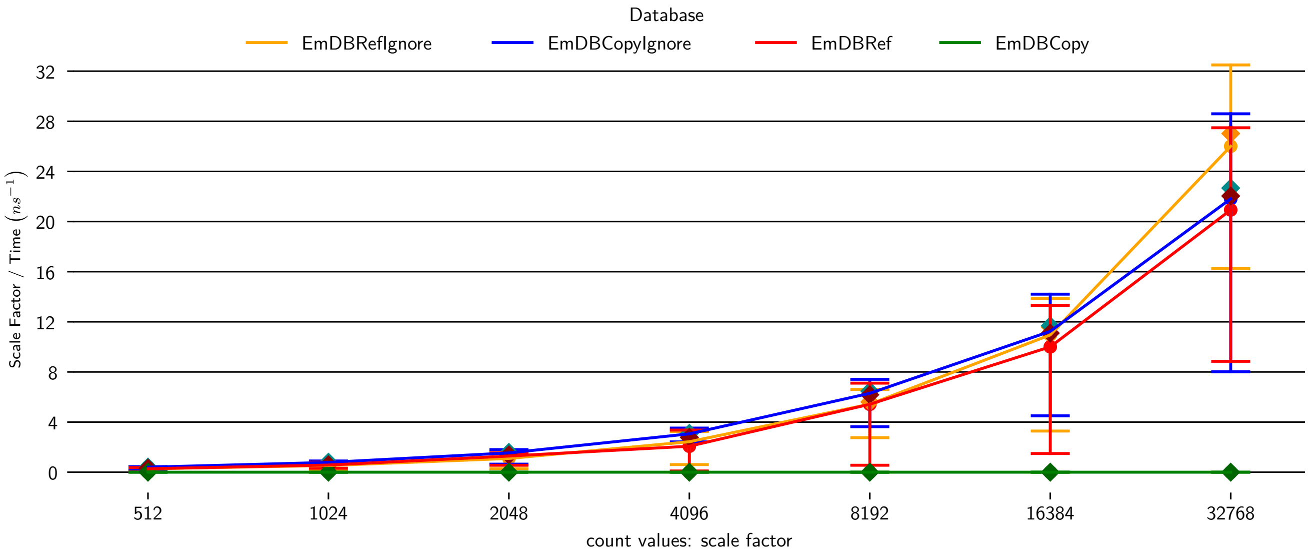


Figure 3.11: Comparison showing the effect of a redundant string clone not being eliminated

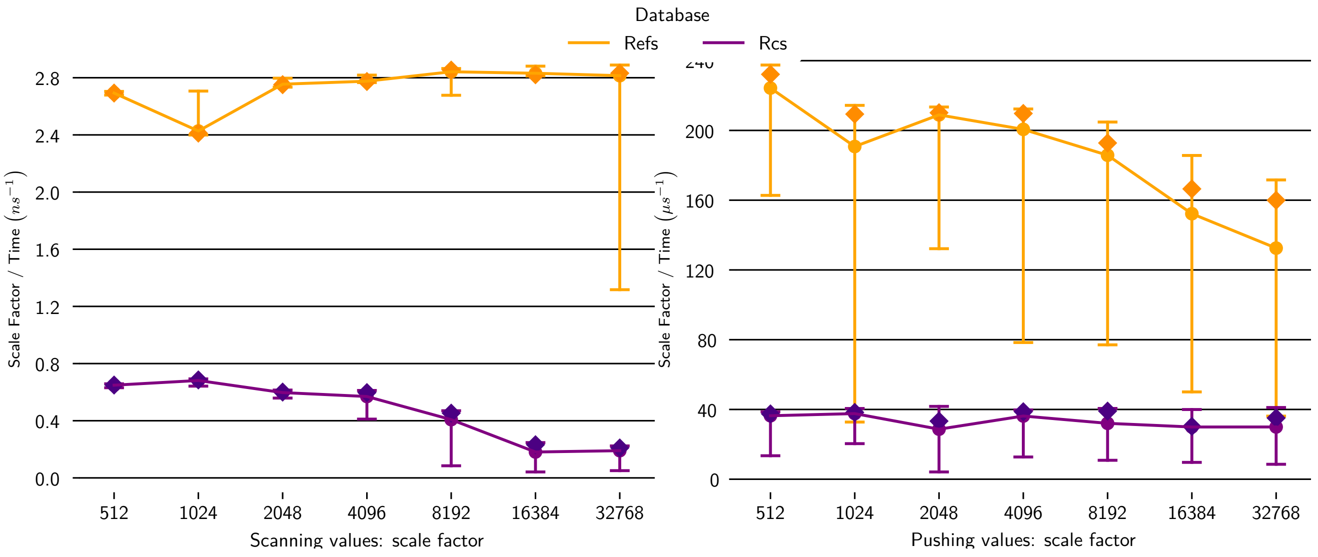


Figure 3.12: Heap allocated & reference counted versus Borrow

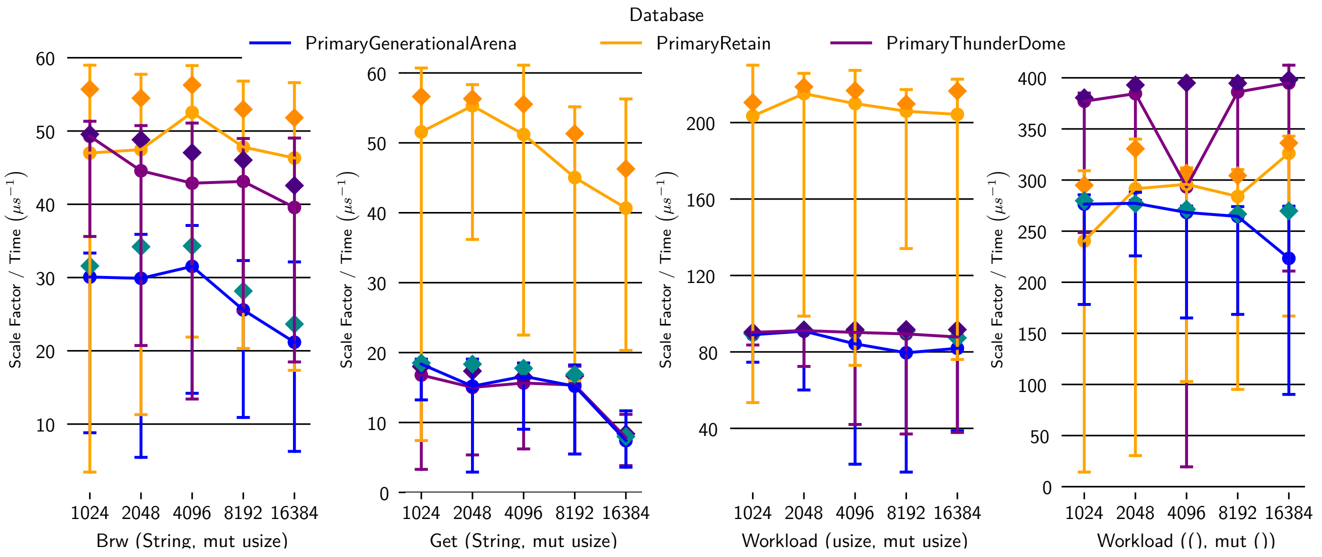


Figure 3.13: Comparison Against Thunderdome, and Generational Arena

3.6 Logical Plan

3.6.1 Rust Expressions

emQL uses rust types and expressions, so need to be able to properly pass these through to the backend for code generation.

There are significant advantages possible from analyzing these rust expressions:

- Detecting unused record fields in `map` operations (can allow us to remove fields and improve performance, for example in unused accessed fields3.5.2)
- Rewriting user expressions to avoid conflict with emDB generated fields.
- Embedding emDB types within rust types (for example `Option<ref table>`).

However the current procedural macro interface has limitations:

1. emQL only receives tokens the macro is invoked with, meaning it is not possible to analyse user types defined outside this context.
2. Even for `std` and `core` types, a user may shadow these inside the expression.

```
{
    use SomeUserType as i32;
    // i32 is no longer what we expect, so we cannot rely on scraping names in syntax to determine types.
    let x: i32 = SomeUserType::not_i32_construction();
    x
}
```

Building an external tool that analyses an entire crate could work for context, however has many of the weaknesses discussed in 3.1. It is possible to analyse rust expressions by using the `rustc_interface` library shipped with the compiler.

- `rustc_interface` is the interface used by both user code, and the rust compiler application to access the library implementing the compiler.
- All stages of compilation can be accessed, for example dropping from tokens to an AST, to the HIR for running type inference on rust expressions.
- `rustc_interface` is unstable as a policy, and the compiler development team are willing to make breaking changes while improving the compiler.

The advantage gained was low compared to the maintenance and development costs to analyze, so instead emDB passes rust code through to the backend.

- The syntax for types and expressions is checked (by the syn rust ast parsing library) to prevent syntactically invalid code (which could result in spans emDB generated code being part of error messages - resulting in confusing error messages) from being passed through to the generated code.
- The `Serialized` backend carefully prevents possible name conflicts, and correctly scopes passed code. It does not use any type information from user provided types.

Future Work: Improved Error Interface

The `proc_macro::Diagnostic` type is still unstable (emDB uses the `proc_macro_error` crate to emit stable diagnostics when compiled with the stable compiler). As a result bugs are still present, including two separate internal compilers that affect error messages produced by syn for some invalid emQL expressions.

3.6.2 Logical Plan Structure

EmDB represents logical plans as a graph of operators, contexts and types.

Context	A group of operators that has access to some values for use in expressions, and can take in some streams. Analogous to a scope. Contexts include an ordering of operators to determine the order in which those operator's expressions can access available variables by borrow or move.
Types	Represented as scalar types (i.e. rust types, table references, bags (collection of records), or records), and record types (used for streams, a set of names available in expressions). Types are represented as a graph as each can also be a reference to another type, this allows for chains of equal (identity) types, meaning alterations and optimisations can be efficiently implicitly propagated through the type graph.
Operators	Each operator is a graph node connected to others through directed dataflow nodes (of streams or single values). This allows operators to have n inputs and outputs, and to name these.

All operators, types and contexts are included in arenas, and are not partitioned by query. This provides both simple unique identifiers (indexes), and allows for easier inter-query optimisation (as is intended in future).

```rust
pub struct Plan {
    pub queries: GenArena<Query>,
    pub contexts: GenArena<Context>,
    pub tables: GenArena<Table>,
    pub operators: GenArena<Operator>,
    pub dataflow: GenArena<DataFlow>,
    pub scalar_types: GenArena<ScalarType>,
    pub record_types: GenArena<RecordType>,
    _holder: (),
}
```

Figure 3.14: The emDB plan structure

3.6.3 Plan Tooling

In order to allow for easy debugging of generated plans, and of plan-mutating optimisations in future, the `Planviz` backend generates graphViz dot files at compile time.

- emDB compiles emQL fast enough to display near-live plan visualizations during editing.
- Additional display of the type graph can also be generated.

```
emql! {
    impl code_display as PlanViz{
        path = "emdb/tests/debug/code.dot",
        types = off,
        ctx = on,
        control = on,
    };

    impl my_db as Serialized;

    table customers {
        forename: String,
        surname: String,
        age: u8,
    } @ [pred(*age < 255) as sensible_ages]

    query customer_age_brackets() {
        use customers
            |> groupby(age for let people in {
                use people
                    |> collect(people as type age_group)
                    ~> map(age_bracket: u8 = *age, group: type age_group = people)
                    ~> return;
            })
            |> filter(*age_bracket > 16)
            |> collect(brackets)
            ~> return;
    }

    query new_customer(forename: String, surname: String, age: u8) {
        row(
            forename: String = forename,
            surname: String = surname,
            age: u8 = age
        )
            ~> insert(customers as ref name)
            ~> return;
    }
}
```

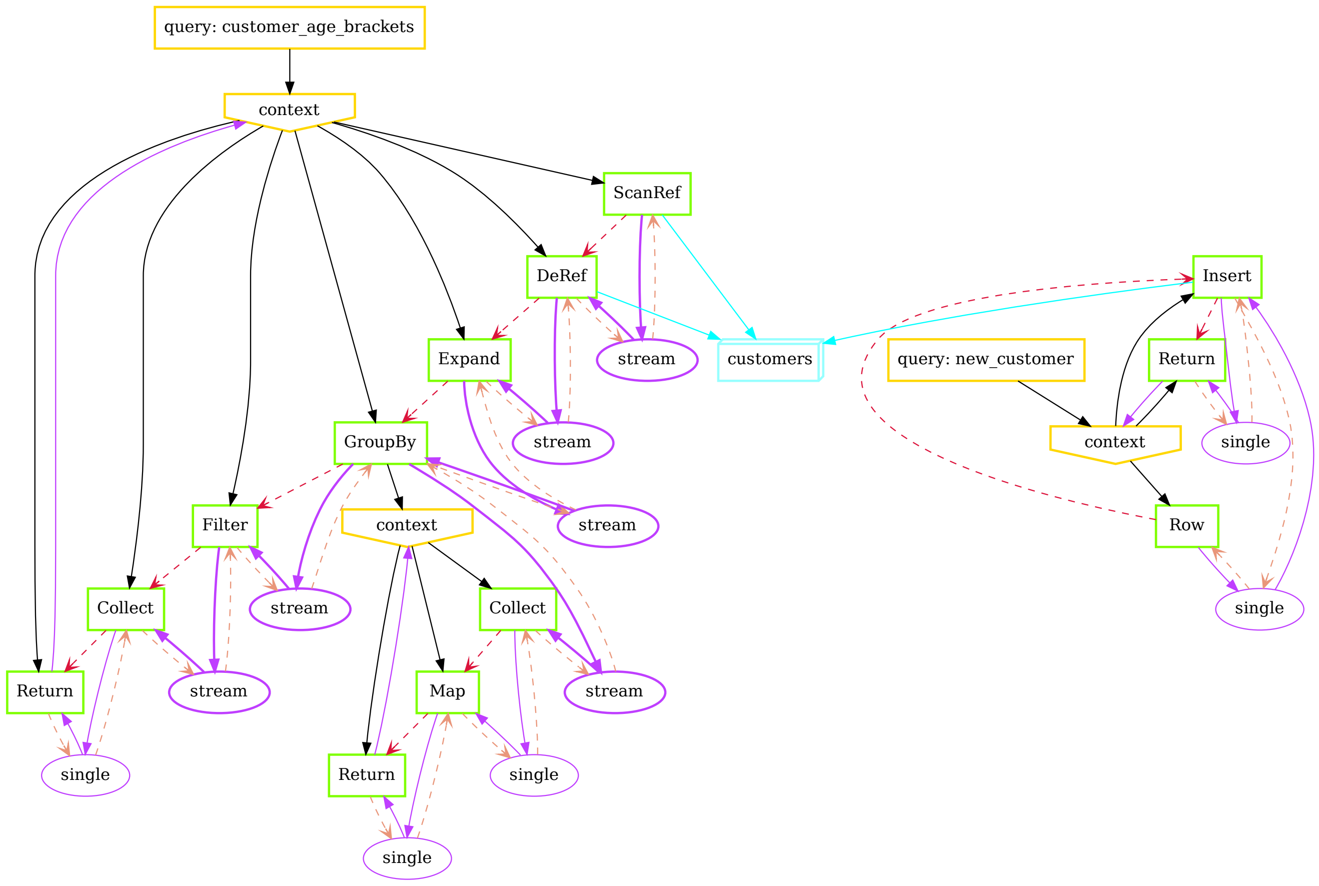


Figure 3.15: A plan visualization generated at compile time

3.7 Code Generation

3.7.1 Quasi-quoting

Rather than manually generating tokenstreams, the `quote!` macro can be used to generate tokenstreams of quasi-quotes. For example the below code is extracted from the `Interface` backend.

```
quote! {
    if #some_tokenstream {
        #other_tokenstream;
    }
}
```

Iteration patterns can be used to extract from iterators (such as vectors of objects implementing the `quote::ToToken` trait).

```
quote!{ #(#some_statements;)* }
```

Future Work: Optimising usage of quote

In part as a result of the 174 invocations of quote from the pulpit table generation library, from-fresh compilation times for`pulpit_gen` are very slow, and require significant memory (≈ 6GB on fresh compile). More work needs to be done in optimising this & emDB's from-fresh build times in general.

3.7.2 Serialized Backend

Serialized Isolation	The generated database object has no special support for concurrency. It is safe to run concurrent readonly queries (enforced by rust). Mutation can be supported by wrapping the database in a lock.
Limited Concurrency	Each operator is evaluated in the order they appear in the emQL code. The default operator implementation is not parallel, however an inefficient parallel implementation exists & compiles.
Human Readable	The generated rust code is human readable, and can be formatted and output to a separate file at compile time for debugging.
Configurable	Table selection and operator implementation are configurable from the emQL `impl` declaration.
Interface Support	The serialized backend can optionally implement a trait for the emQL schema, as is generated by the `Interface` backend, this makes generic tests and benchmarks possible, and is how this report's evaluation was built.

Future Work: Parallel Queries

There are three potential avenues for parallelism in Serialized generated queries.

1. Parallelism between queries.
2. Parallelism inside operators.
3. Parallelism in applying operators.

The latter is feasable within the current code generation, the generation of expressions & closures for user expressions (capturing variables from context) is separate from the application of operators to dataflows. So the latter can be re-ordered and parallelised.

- Some extra analysis to ensure accesses to the same tables are synchronised is required within operators, and to maintain observably serial execution between queries.

3.7.3 Operator Implementation

	Pull Based/Volcano Processing	**Push Based/Bulk Processing**
Description	Operators pull tokens from input operators.	Operators push buffers of tokens to output operators.
Evaluation	Lazy	Strict
Advantages	Keeps values in registers between operators.	Code generation for push operators is considerably simpler (Operator takes immediate inputs and assigns outputs).
Weakness	large number of calls (typically virtual for runtime physical plan generation) and with complex control flow (degrades branch predictor and code locality in icache).	Requires materialising results after every operator.

The minister crate provides the interface for operators, including various maps, filters, joins and buffering. In order to provide a flexible interface that allows `impl trait` types for streams and single values, in the absence of associated trait items for traits, minister uses a macro to define new traits for operators.

For example the trait generation specifies the filter operation as:

```
fn filter<Data>(stream: stream!(Data), predicate: impl Fn(&Data) -> bool + Send + Sync) -> stream!(Data)
where
    Data: Send + Sync;
```

For example the default operator implementation for `Serialized` (`Iter`) is defined using the trait generated by:

```
macro_rules! single { ($data:ty) => { $data }; }
macro_rules! stream { ($data:ty) => { impl Iterator<Item = $data> }; }
super::generate_minister_trait! { IterOps }
```

The `Iter` implementation makes use of the rust iterators, which are lazily evaluated, monomorphised (no virtuals, can be inlined) streams of data.

1. Operators appear push based, `let output = Iter::map(input)`. This makes code generation simple.
2. However, the `input` and `output` are actually lazy streams, so this is semantically a pull based operator.
3. At compile time, rust can inline the calls to parent iterators in `input`, converting it into an iteration over the source of the data, directly into a buffer (output). This is physically the same as an optimised push-based system (albeit between the pipeline breaking operators).

This is the same model implemented by hyper[23], though with significantly reduced implementation complexity.

"…data is always pushed from one pipeline-breaker into another pipeline-breaker. Operators in-between leave the tuples in CPU registers and are therefore very cheap to compute."
– Efficiently Compiling Efficient Query Plans for Modern Hardware[23]

Additionally the in-place-collection optimisation for vectors[3] can remove the cost of allocating the output buffer, by reusing the same allocation as the input (when running the above benchmark, the iterator implementation does not allocate, but the vector version does - tracked by the Divan AllocProfiler).

Iterator

```
fn op<I,O>(values: Vec<I>, f: impl Fn(I) -> O)
 -> Vec<O> {
    values
        .into_iter()
        .map(f)
        .collect()
}
```

Loop

```
fn op<I,O>(values: Vec<I>, f: impl Fn(I) -> O)
-> Vec<O> {
    let mut result = Vec::with_capacity(values.len());
    for item in values {
        result.push(f(item));
    }
    result
}
```

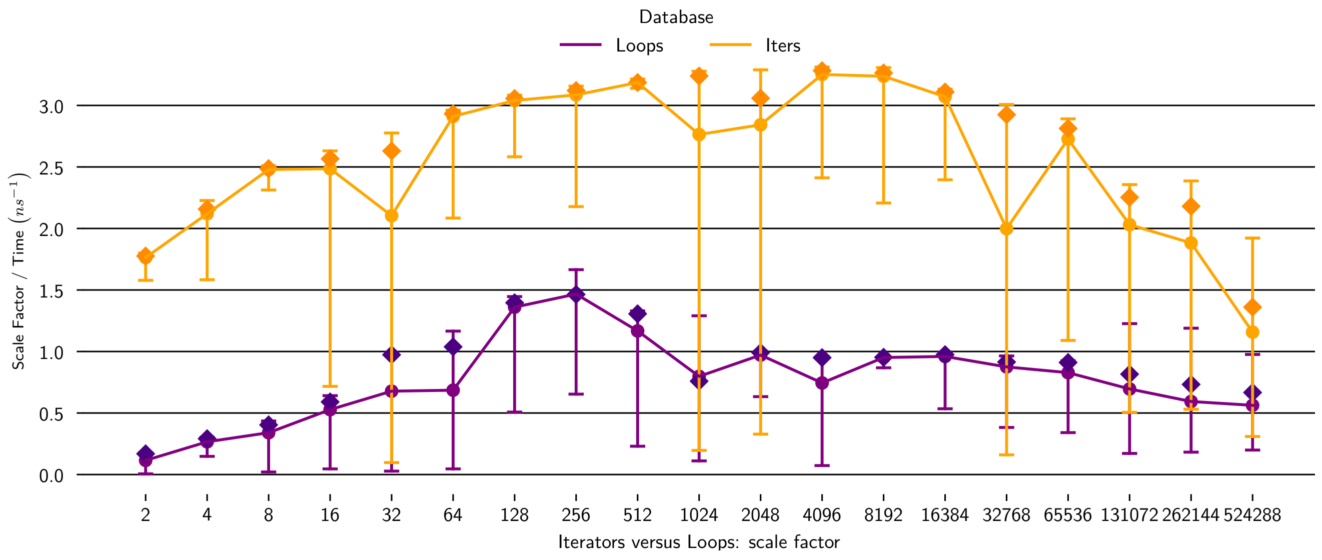


Figure 3.16: Mapping sum onto tuples (using standard allocator)

3.8 Additional Libraries

3.8.1 Enumtrait

Many of the data structures inside emDB are represented as rust `enum`s. This allows for easy pattern matching, but makes implementing traits tedious.

- Enumtrait uses tokens from `enum` and `trait` definitions to implement traits on enums of data types that implement the trait.
- Boilerplate for type $\rightarrow$ `enum` type conversion is also generated.

There are existing libraries for this, such as `enum_dispatch`, however this relies on communication between proc macro invocations at compile time through static data structures.

- Significant complexity to cope with ordering of invocations (ordering is undefined by rust).
- Needs to tokens types to strings (loosing span information) while communicating (`TokenStream` types are special and cannot be shared between threads).

Enumtrait solves the procedural macro communication issue, by generating new `macro_rules!` definitions as callbacks containing stores of tokens. AS the definition $\rightarrow$ use of macros is ordered, this allows communication including tokens and their spans intact.

```rust
struct Bar { common_field: usize }
struct Bing { common_field: usize, other_field: String }

#[enumtrait::quick_enum]
#[enumtrait::store(foo_macro_store)]
enum Foo {
    Bar,
    Bing,
}

#[enumtrait::store(foo_trait_store)]
trait FooTrait {
    const BAZ: usize;
    fn foo(&self) -> usize;
}

impl FooTrait for Bar {
    const BAZ: usize = 1;
    fn foo(&self) -> usize { self.common_field }
}
impl FooTrait for Bing {
    const BAZ: usize = 2;
    fn foo(&self) -> usize { self.common_field }
}

#[enumtrait::impl_trait(foo_trait_store for foo_macro_store)]
impl FooTrait for Foo {
    const BAZ: usize = 42;
}

fn check(f: Foo) -> usize { f.foo() }
```

Chapter 4

Evaluation

4.1 Representative Benchmarks

To determine the advantage provided by the core concept of the project requires assessing the impact of:

1. Optimising table access into returning references.
2. Optimising table structures for append only workloads.
3. Embedding application logic inside database queries.

A direct comparison and evaluation of the specific benefits of code generation is not in the scope of this evaluation. While there is clearly a performance advantage to be gained from running native, optimised code (without a runtime cost), emDB is implemented with different operators, and is currently running a very simple iterator based backend.

Ideally we would use a benchmark considered representative of embedded database workloads, and contains schemas for which the 3 features we want to investigate are applicable.

TCP-H

Covers aggregation as well as concurrent data modification, adherence to specification requires either using a separate driver - not easily embedable while adhering to the specification.

- The benchmark is designed for a persisted business database, so uses all mutations (insert update, delete) which prevents the mutability optimisations that emDB performs, some embedded database workloads are append only, and thus choosing a benchmark that also supports
- Concurrent modification is only possible with the current Serialized emDB backend by placing the entire database behind a RWLock, but TCP-H is designed in part for benchmarking *concurrent data modifications*[7]

It would be possible to heavily modify TCP-H (data generator linked with benchmarks & in-memory, on a low scale factor with benchmarks including no updates or deletes).

However this benchmark would be TCP-H in schema & queries only (not useful to compare with other TCP-H results) and would not be particularly useful in validating the 4 key optimisations implemented without modifying the sets of queries used (i.e. TCP-H, but append only).

H2O.ai Database Benchmark

This benchmark is designed for *database like-tools [in] data-science*, and benchmarks aggregations using groupby and join on an in-memory dataset[13].

- It is used by, and since 2023 has been maintained by the DuckDb project[18], and is used by that project to benchmark DuckDB's aggregations.
- As it is primarily for benchmarking aggregations over dataframes, it does not consider the impact of updates, or extracting data from the database. Meaning it cannot be used to assess append only workloads or returning references.

CrossDB Bench

Designed by the CrossDB project, the (self advertised) *"fastest embedded database"*[8]. The incuded benchmark compares against lmdb and sqlite3.

CrossDB is more comparable to a key-value store, and does not support complex operators (SELECT with computation, groupby, join etc.). As a result the benchmark benchmarks inserts, deletes, and updates.

- CrossDB could even be used as a backend for emDB, as emDB's operators are separate from the data storage.
- Despite being integrated into C (schemas are defined with C structs, cursors into tables are directly accessible as part of the API, and reference C types)

Yahoo Cloud Serving Benchmark

A popular and highly configurable set of benchmarks for key-value stores. Much like the CrossDB benchmarks, the lack of complex queries means it is not useful in investigating the 3 features.

Custom Benchmarks

Rather than adapting an existing benchmark, designing a new set of test schemas and queries allowed the 3 key features to be targetted. Given the popularity of SQLite and DuckDB in the Rust ecosystem, these were the other embedded databases chosen for the comparison.

Embedded Database	SQLite	DuckDB	ExtremeDB	MonetDB/e
crates.io All-TIme downloads	$17,780,740$	$174,602$	$1,757$	(not available)

Other more popular *embedded databases* were ommitted from the selection as they are more akin to transactional key-value stored. The popular *"pure-rust transactional embedded database"* sled[21], LmDB[29] and CrossDB[8] were ommitted for this reason.

In order to simplify the creation of new benchmarks, emDB includes an `Interface` backend that generates traits that can be consumed and implemented by emDB's `Serialized` backend, or implemented manually (to wrap other databases).

Future Work: Develop a more comprehensive benchmark suite

Given there are no ideal existing suites that mix mutability, and test embedded logic, one will need to be properly developed for emDB (also serving as a higher coverage test suite).

4.2 [Quantitative] Performance

4.2.1 Benchmark Setup

Compilation

All benchmarks were built with the following cargo build profile on `rustc 1.80.0-nightly (032af18af 2024-06-02)`

```
[profile.release]
lto = "fat"         # Maximum link-time optimisation - important for linking for DuckDB and SQLite
codegen-units = 1   # Single codegen unit gives compiler full context of benchmarks for optimisation
```

Profile guided optimisation was not used in this case as while it is supported by DuckDB and SQLite it cannot be applied as they are built by separate build systems and compilers that do not interact with the `cargo pgo`[4] tool.

Database	Version	Crate
DuckDB	0.10.1	`duckdb =   { version = "0.10.2", features = ["bundled"] }`
SQLite	3.46.0	`rusqlite = { version = "0.31.0", features = ["bundled"] }`

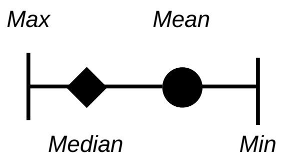
Both are build using gcc 11.4.0 in release mode. Full build configuration used can be found in their associated crates.

Benchmarking

For each benchmark emDB generates a trait that is manually implemented for DuckDB and SQLite. A single benchmark function is used which takes a generic type implementing the trait.

```
#[divan::bench(
    name = "benchmark name",
    types = [EmDB, SQLite, DuckDB],
    consts = TABLE_SIZES,
)]
fn some_benchmark<DS: Datastore, const SCALE_FACTOR: usize>(bencher: Bencher) {
    // ... benchmark code
}
```

- All implementations have freedom of return type (i.e. on failure emDB returns errors, DuckDB and SQLite panic the benchmark).
- Each benchmark runs from a single threaded interface (query must end before another begins) but implementations can use multiple threads.
- `prepare_cached("..query")` is used for the SQLite and DuckDB queries.



For each benchmark we re-scale the results by the scale factor, and take the inverse (higher is better).

Hardware

All benchmarks were run on a single machine running Ubuntu 22.04.3 LTS on WSL version: 2.1.5.0 (Windows 11) with 12th Gen Intel i7-12800H and 8GB of available memory.

4.2.2 Data Logs

Schema

Designed to demonstrate the impact of removing large copies (in this case of the `comment` string), for a query on static data (i.e. a typical ETL pattern, loading data into memory and then computing).

```
table logs { timestamp: usize, comment: Option<String>, level: LogLevel }
pub enum LogLevel { Error, Warning, Info }
```

Prior to the benchmarks being run, the table is populated with a number of rows equal to the scale factor.

- `timestamp` is added sequentially up to the scale factor.
- `level` is added randomly, with 20% `LogLevel::Error`, 40% `LogLevel::Warning` and 40% `LogLevel::Info`.
- `comment` is added with 50% `None`, and 50% containing a random string of random (uniformly distributed) lengths from 0 to 1024 characters.

Comment Summaries	For each comment, get the length and the first 30 characters.
Errors per minute	Group each error by its minute, and return the number of error logs.
Data Cleaning	Demote all `LogLevel::Error` logs to `LogLevel::Warn`.

EmDB Implementations

The `NoCopySelector` table implementation selector is enabled for the no-copy emDB implementation. It chooses the same column data structures as the default `MutabilitySelector` but places all values in the mutable side of the rows.

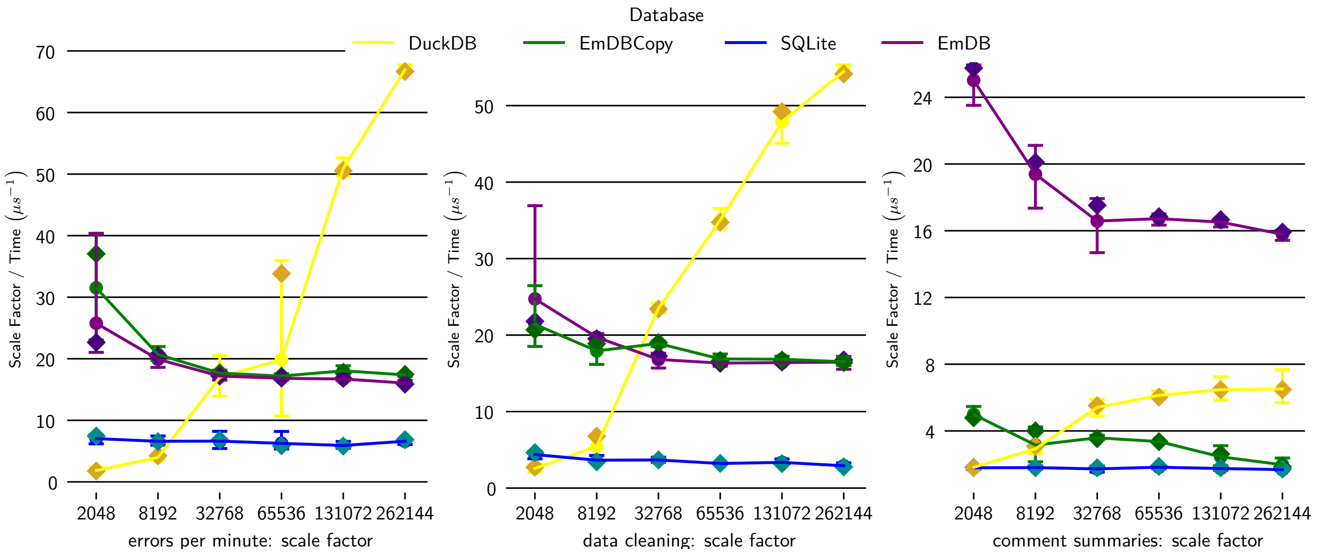


Figure 4.1: Data Logs Schema Benchmark Results

Results

- DuckDB scales extremely well over $\approx 33,000$ rows and sees a significant improvement in performance per row. Its is not due to multithreading, or the columnar storage (tested with single thread (same result) and against emDB with the `Columnar` table selector).
- I have not determined the root technique to explain this performance improvement, but suspect large scale factors are amortizing fixed overhead.
- The advantage from returning references for immutable data is significant (over $2\times$ performance improvement for comment summaries).
- There is significant inefficiency in emDB for scans and table accesses (requires collecting a buffer of row references, and re-checking bounds for each access).

4.2.3 Sales Analytics

Schema

The aim of this benchmark is to demonstrate the performance advantage of embedding logic.

- The rust compiler can use the context of functions, values passed for queries (exchange rates) in optimisation.
- For the *customer value*, *product customers* and *category sales* queries, random data is added to the tables before the query is run.
- For the *mixed workload* a loop (of scale factor iterations) either runs one of the other 3 workloads (each with probability 12.5%), or inserts a single new customer and 10 more sales (62.5%)

```
table products {
    serial: usize,
    name: String, // String of form format!("Product {serial}")
    category: crate::sales_analytics::ProductCategory,
} @ [unique(serial) as unique_serial_number]

table purchases {
    customer_reference: usize,
    product_serial: usize,
    quantity: u8,
    price: u64,
    currency: crate::sales_analytics::Currency,
} @ [pred(crate::sales_analytics::validate_price(price, currency)) as sensible_prices]

// We delete old customers, but keep their references
table current_customers {
    reference: usize,
    name: String,     // format!("Test Subject {i}")
    address: String,  // format!("Address for person {i}")
} @ [
    unique(reference) as unique_customer_reference,
    unique(address) as unique_customer_address,
    pred(name.len() > 2) as sensible_name,
    pred(!address.is_empty()) as non_empty_address,
]
```

This schema includes several types defined outside the schema by the user, and some functions.

```
pub enum Currency { GBP, USD, BTC }
struct Aggregate {
    clothes: usize,
    electronics: usize,
    food: usize,
    money_spent: u64,
}

/// Validate a proce by the rules: 1. No more than $10k in dollars, 2. Fewer than 20 in BTC
fn validate_price(price: &u64, currency: &Currency) -> bool {
    const DECIMAL: u64 = 100;
    match currency {
        Currency::GBP => true,
        Currency::USD => *price <= 10_000 * DECIMAL,
        Currency::BTC => *price < 20,
    }
}

fn exchange(btc_rate: f64, usd_rate: f64, price: u64, currency: Currency) -> u64 {
    match currency {
        Currency::GBP => price,
        Currency::USD => (price as f64 * usd_rate) as u64,
        Currency::BTC => (price as f64 * btc_rate) as u64,
    }
}
```

Customer Value	Get the total value of a customer's purchases, using the current exchange rate. Additionally get the sum of all products they have purchased in each product category.
Product Customers	For a given product get for each purchasing customer the customer reference and total spent by the customer on the product.
Category Sales	Get the total sales per category, in the different currencies.

The schema also includes a **Customer Leaving** query, this prevents the emDB database from optimising the `current_customers` table for append only workloads.

DuckDB Enums

For the DuckDB implementation `UTINYINT` instead if `enum`. This is due to a bug in duckDB 0.10.1 that causes the allocator to fail on some `CHECK` constraints using enumerations, which includes the purchase currency constraint.

```
duckdb::data_t* duckdb::Allocator::AllocateData(duckdb::idx_t): Assertion `size > 0` failed
```

Results

- EmDB performs well on the mixed workload (includes fast inserts), but has worse performance for the larger scale factors.
- DuckDB improves for large scale factors, as in the previous **data logs** benchmark.
- SQLite performs well on the **customer value** groupby and the **product customers** join, emDB has not optimised these operators, and suffers for it.

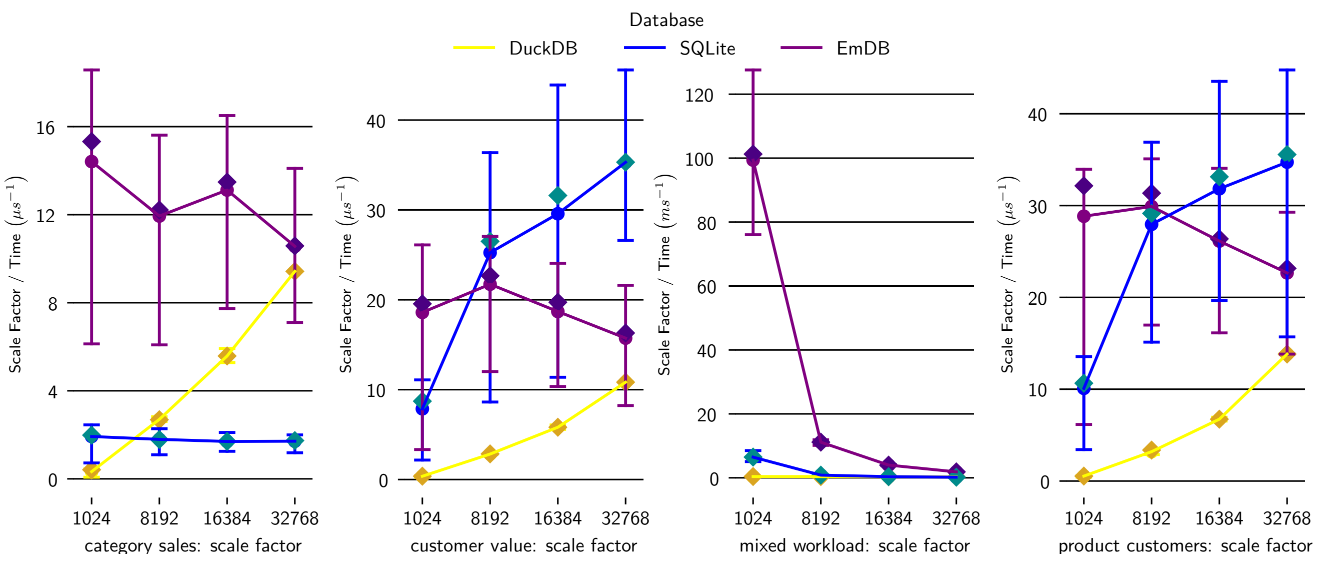


Figure 4.2: Sales Analytics Schema Benchmark Results

4.2.4 User Details

Schema

The aim of this benchmark is to demonstrate performance in simpler key-value stores.

- Include predicates and aggregation (meaning traditional, more optimised key values stores are not applicable).
- Allows access directly to data through row references, and in the case of DuckDB and SQLite a generated unique ID.
- Demonstrates a performance advantage for returning references (for snapshotting the data).

```
table users {
    name: String, // never updated
    premium: bool,
    credits: i32,
} @ [
    pred(*premium || *credits > 0) as prem_credits
]
```

Results

- This schema benefits from fast lookup of row references, DuckDB's performance is too slow for this to avoid timeout on a sufficient number of iterations.
- EmDB's fast row references are a huge performance advantage for the OLTP queries (getting random ids, random inserts), SQlite's row identifiers have significantly more expensive lookup. DuckDB is only competitive on the larger aggregations (OLAP workloads it was designed for).

Future Work: Parallel Operators

At large scale factors performance could be improved by applying operators in parallel.

- This can be trivially applied to `map`, `filter`, `asset`, `deref`
- The operator interface supports parallel operators through trait bounds on `Send + Sync` data and closures (e.g. closure provided for map).
- A basic rayon[20] based backend is present, however it has poor performance due to being maximally parallel (large number of tasks generated with considerable overhead).

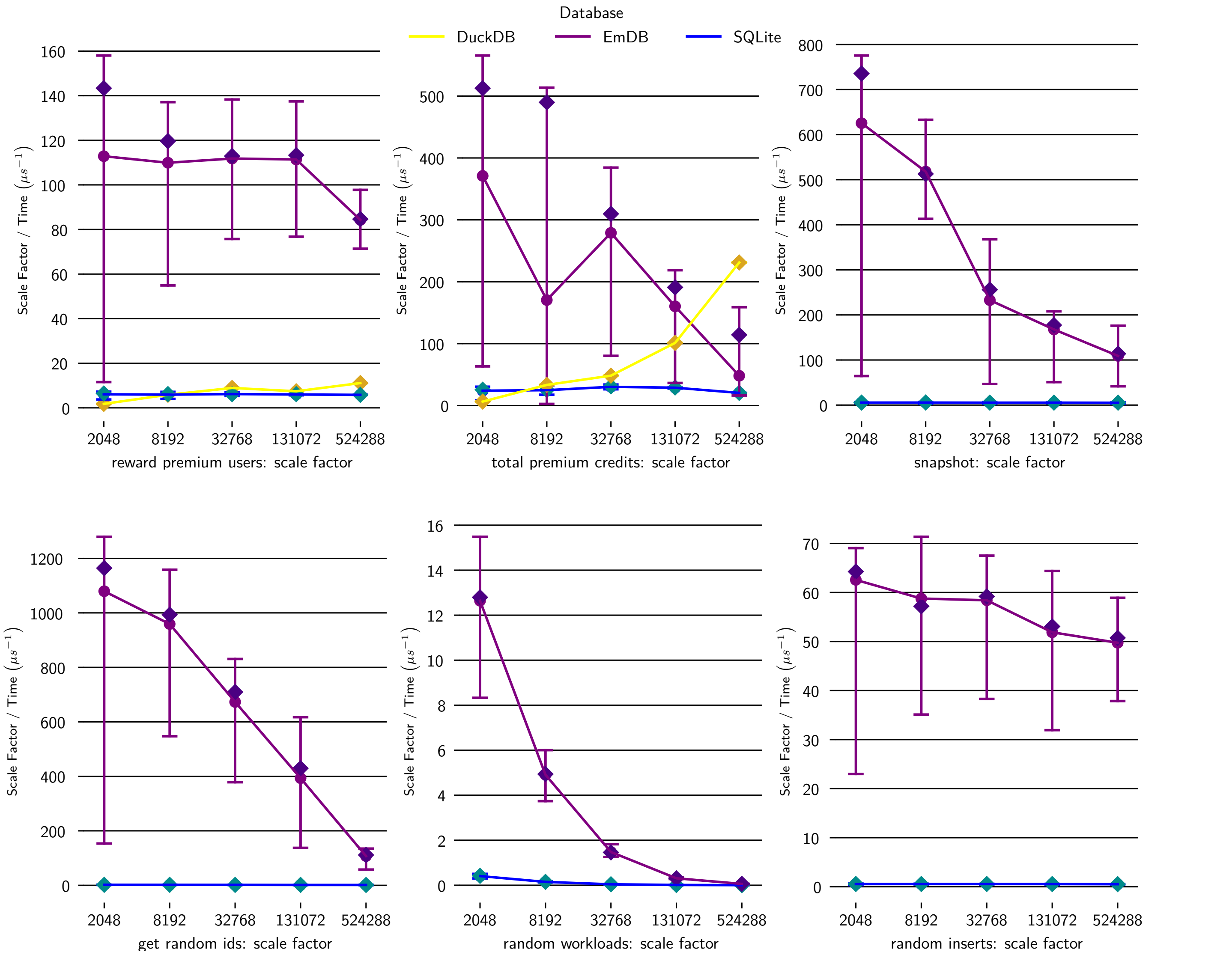


Figure 4.3: User Details Schema Benchmark Results

4.3 [Qualitative] Correctness

4.3.1 Correctness of Supporting Libraries

Very few instances of unsafe code, ensuring normal rust safety guarantees apply.

- For the unsafe code used in implementing some column data structures, kani[14] (a CBMC[9] wrapper) is used to verify abcense of generic bugs.
- No large test suites are present, this is a weakness for logical errors.

4.3.2 Correctness of Code Generation

With one key exception (accessing secondary columns without bounds checking) no `unsafe` code is generated. Hence normal rust safety guarentees apply (no data races, no undefined behaviour, no use after free, etc.).

- Normal rust safety guarantees apply to the generated code.
- Wrappers inside emDB's code generation ensure quasi-quotes can be re-parsed as the AST nodes they should represent (only on debug mode for performance).
- The produced code is human readable, a debug write mode is included to divert generated code to separate files for inspection.

The emDB compiler currently compiles with both stable and nightly rust compilers (with improved error diagnostics).

4.3.3 Susceptibility to User Error

Given emDB allows the user to access internal state safely to immutable values, it is critical to prevent bugs in user code mainfesting in difficult to debug issues inside the database.

For example, invalid access through an emDB provided reference corrupting memory in a table, resulting in a difficult bug manifesting in failures for unrelated queries.

The qualification of user provided references with the database lifetime, and the accessibility to internal data structures only through the safe query interface limit the exposure to bugs in user code. However, there are still three ways in which this can be damaged.

1. **Use of unsafe code by the user.**
   There is no way to prevent this, and it is clear to the user that they are using unsafe code.
2. **Use of a rust soundness hole by the user.**
   This is a rust compiler issue (for example the demonstrations in cve-rs[26])
3. **Access to internal database data structures through code embedded in operators.**
   User code substituted to inside the body of a query can access internal variables that are in scope. This is somewhat limited by placing the generation of expressions and closures before the operator implementation, however symbols such as the `__internal_self_alias` used to access internal tables are still in scope.

Future Work: Barriers to Production Use

A larger test suite is required to ensure no logical errors in the 'minister' operator implementations.

4.4 [Quantitative] Compile Time Cost

Given the emql proc macro needs to run at compile time, the compile time cost is a significant factor in emDB's usability.

- Increased compile times delay delay the reporting of error messages by the language server.

By using cargo's built in compilation timing to produce this.

```
[profile.dev.package.emdb_core]
opt-level = 3 # Optional additional configuration for maximum proc macro performance
```

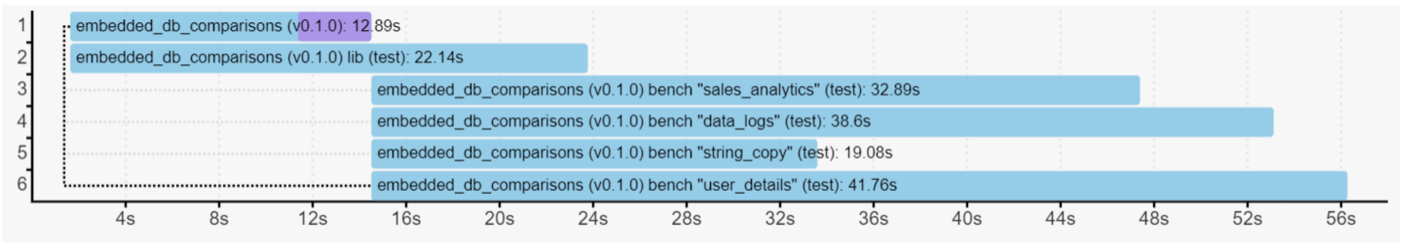


Figure 4.4: Cargo timings for a recompile on no change of the embedded database comparison benchmarks.

- The initial compile takes significant time, however incremental builds afterward are fast enough for a responsive IDE experience.
- The cost of compiling DuckDB and SQLite is comparably large.

Future Work: Feature gating crates

Improvements to the initial/from fresh compile time can be gained by feature gating unused features included in the emDB crate.

- Pulpit table generation macros are included in the emDB crate, for convenience, but are not a requirement to use the emQL macro.

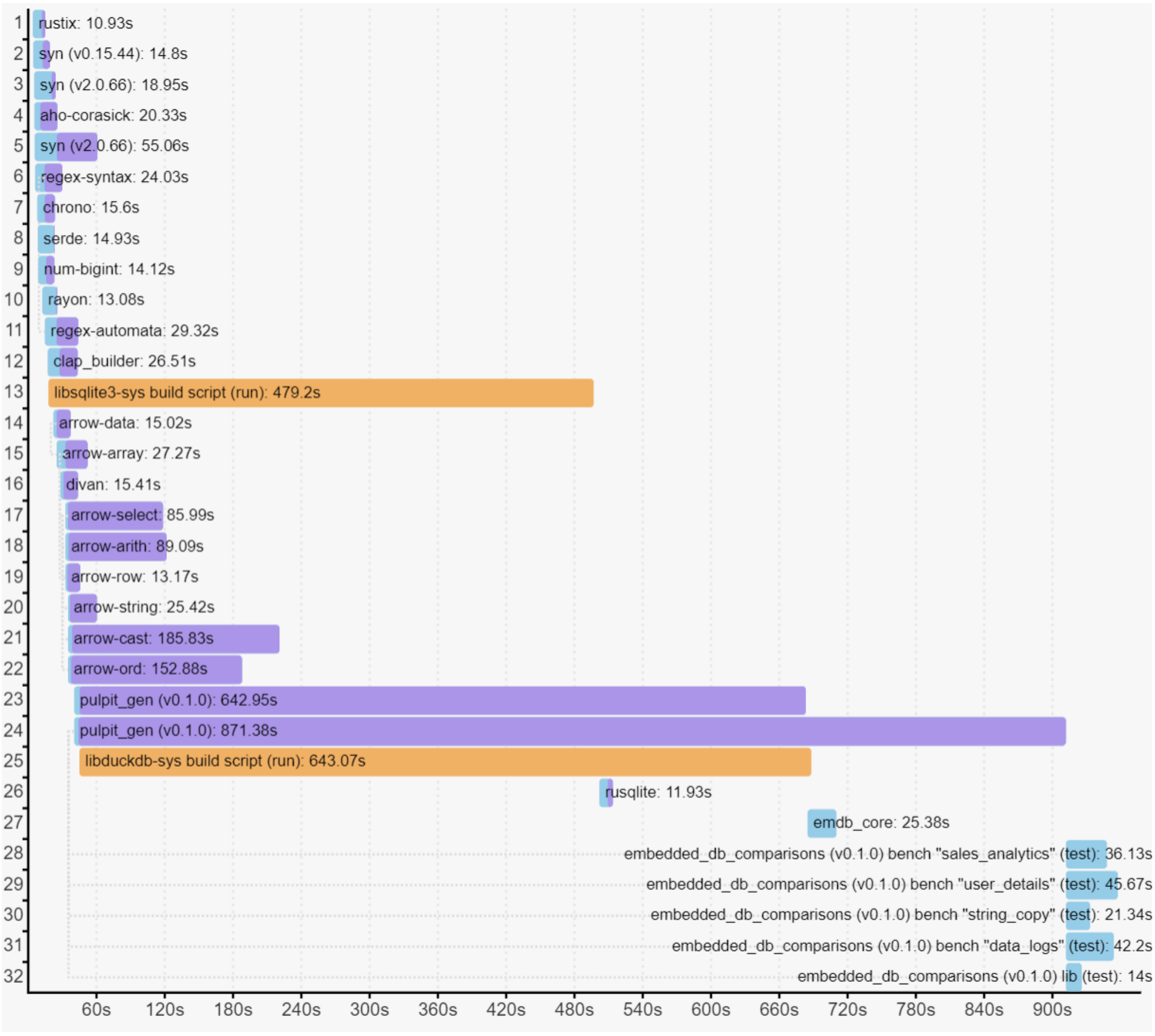


Figure 4.5: Cargo timings for a fresh compilation of the embedded database comparison benchmarks.

Future Work: Improved trait resolution

Trait resolution[31] is an expensive operation. The heavy reliance on traits to define parameters for Combi combinators is therefore a considerable cost to the initial/from fresh compile of emDB.

Combi is in fact so expensive, that some constructs cannot be compiled by the latest nightly or stable compilers (as of `rustc 1.80.0-nightly (032af18af 2024-06-02)`).
(from emDB/pulpit_gen/src/macros/new_simple).

```
        mapall( MustField::new("name", getident),
        ( DefaultField::new("transactions", on_off, ||false),
          ( DefaultField::new("deletions", on_off, ||false),
            ( /* ... singificant nesting of Fields which construct a combinator tree ... */ )
          )
        )
    ).gen(':'),
```

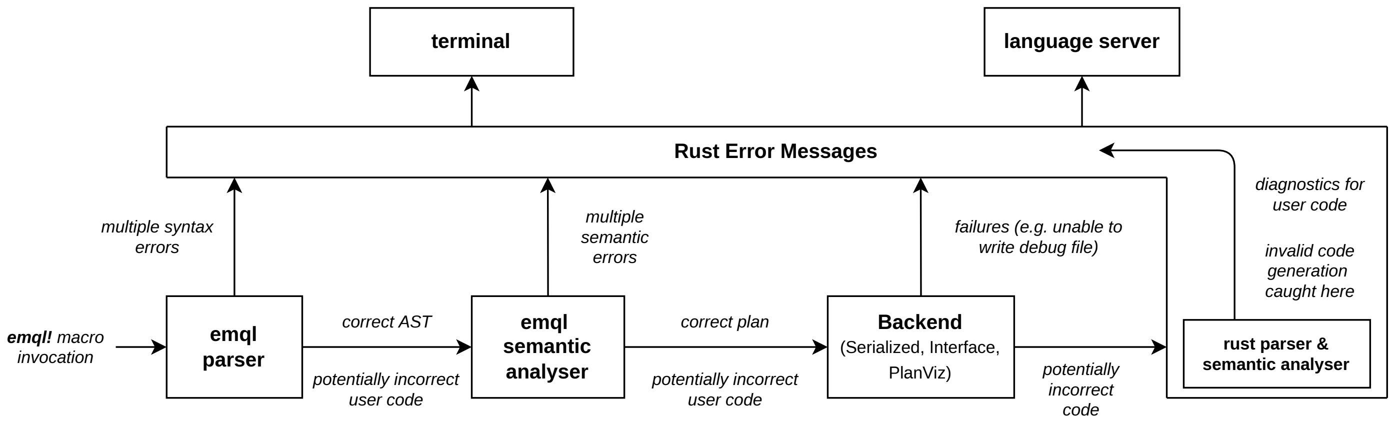
This will hopefully be improved through both performance improvements to rustc and in future the a Chalk-based trait solver.[30]

4.5 [Qualitative] Ease of Use

```toml
# in rust project `Cargo.toml`
[dependencies]
emdb = { git = "https://github.com/OliverKillane/emDB.git" }
```

emDB generates rust diagnostics, which are already integrated with rust supporting IDEs. This is not possible with either SQLite or DuckDB without using a tool like sqlx.

- The sqlx project contains several database access libraries, including the `sqlx::query!` macro, which connects to a live database to syntax & semantics check the queries.
- This requires the developer to keep a development database running for access. But also allows sqlx to work with a variety of different databases & SQL variants.
- sqlx cannot propagate errors back to individual spans inside a query string.



terminal
language server
Rust Error Messages
diagnostics for user code
invalid code generation caught here
multiple syntax errors
multiple semantic errors
failures (e.g. unable to write debug file)
emql! macro invocation
emql parser
correct AST
potentially incorrect user code
emql semantic analyser
correct plan
potentially incorrect user code
Backend (Serialized, Interface, PlanViz)
potentially incorrect code
rust parser & semantic analyser

Figure 4.6: Stages of error message generation

A significant weakness of emDB is that the correctness of user embedded code depends on the optimisations applied to data structures.

- Values versus references for items gotten from tables, only known after analyzing all queries and determining the data structure to use.
- Structure selection requires the full context, so no code is generated (and hence user code checked) if there are any emQL semantic or syntactic errors.

Multiple syntax errors is managed by the Combi library (a part of this project), which allows for multiple syntax errors without error AST nodes, keeping the semantic analysis code relatively simple.

Future Work: Allow both syntax and semantic Errors

Some language features are independent.

- The emql syntax and semantics of independent queries.
- Queries that do not use a given table are independent of syntax or semantic errors with the table.

By adding some error AST nodes, or alternatively hoisting some of the semantic analysis in to Combi reporting more errors could be facilitated.

feature	emDB	DuckDB	SQLite	SQLite + sqlx
`Cargo.toml` **only setup**	Yes	Yes	Yes	No
Compile Time Checks	Yes	No	No	Yes
Identifier Precise Errors	Yes	No	No	No

Future Work: User Survey

The syntax & semantics of emql have been chosen through an iterative *dogfooding* process, and lacks a more general justification.

- User feedback requires a stabilization of the emQL interface and a commitment to support the library long term.

4.6 Conclusion

This project has been successful in implementing a novel, usable & performant kind of embedded database that has not previously existed. There remain significant opportunities to improve performance, both through optimising the current operator and table implementation, and by improving logical optimisations (most importantly incremental view maintenance). This will be aided by the simple, modular & easy to use design of the emDB compiler.

Chapter 5

Related Work

5.1 Embedded Databases

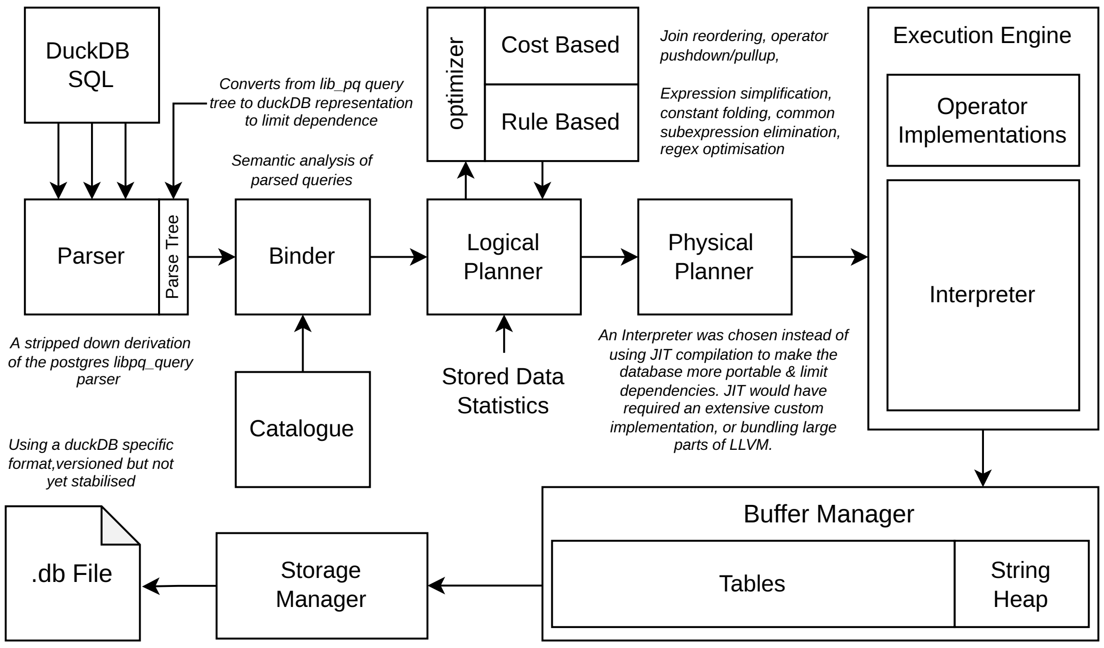
5.1.1 DuckDB

DuckDB is an in-memory, embedded, columnar, OLAP Database[25] developed as a successor to MonetDBLite[24] in the embeddable 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[17].
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 indices.
Concurrency	DuckDB uses a combination of optimistic and multiversion concurrency control.

System Design

The authors describe the design as *”textbook”*[25].

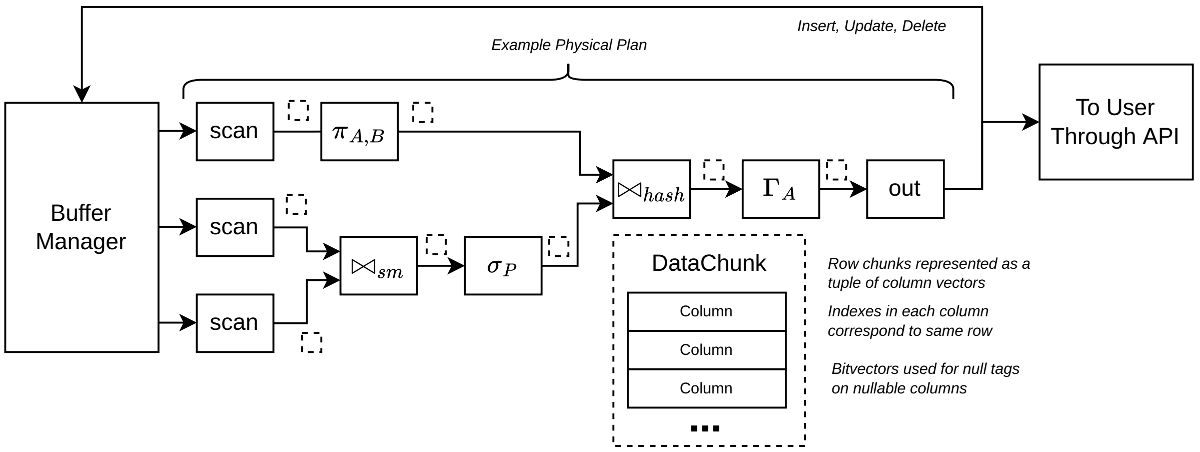


DuckDB SQL
Converts from lib_pq query tree to duckDB representation to limit dependence
optimizer
Cost Based
Rule Based
Join reordering, operator pushdown/pullup,
Expression simplification, constant folding, common subexpression elimination, regex optimisation
Execution Engine
Operator Implementations
Semantic analysis of parsed queries
Parser
Parse Tree
Binder
Logical Planner
Physical Planner
Interpreter
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
Buffer Manager
.db File
Storage Manager
Tables
String Heap

Figure 5.1: An overview of the DuckDB system design.

Vector Volcano Processing

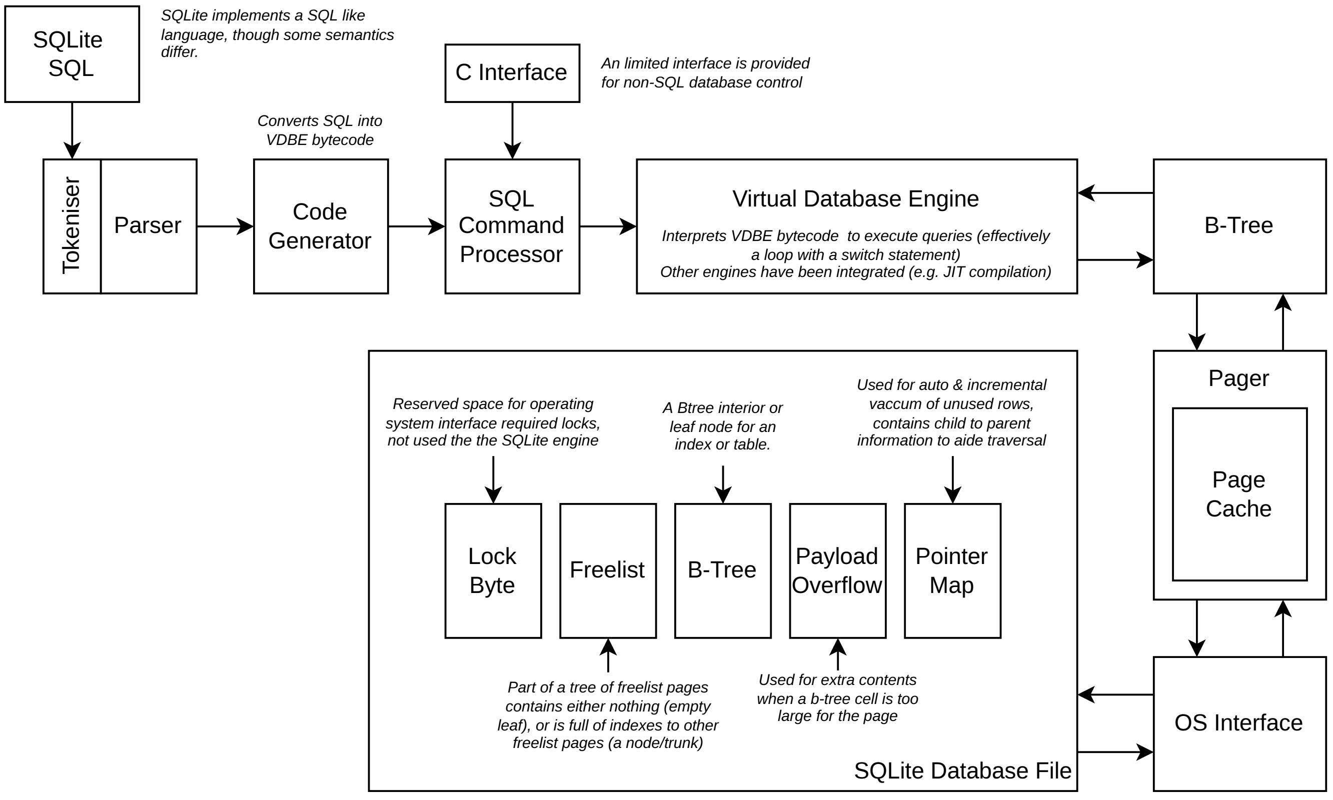
DuckDB uses a *vector volcano* processing model. The interpreter takes a dynamically constructed tree of operators, wherein each operator pulls data from input operators on demand much like with volano processing. However instead of pulling individual tuples, `DataChunks` are passed, each containing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



5.1.2 SQLite

SQlite is a lightweight embedded database[11], and is currently the most deployed database in the world[5]. 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

Figure 5.2: An overview of the SQLite system design.

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.

```sql
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[15] and advertised a $1.72\times$ speedup over a seelction of TPC-H queries.

5.2 Code Generation for Databases

5.2.1 Holistic Integrated Query Engine

The Holistic Integrated Query Engine[16] 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.

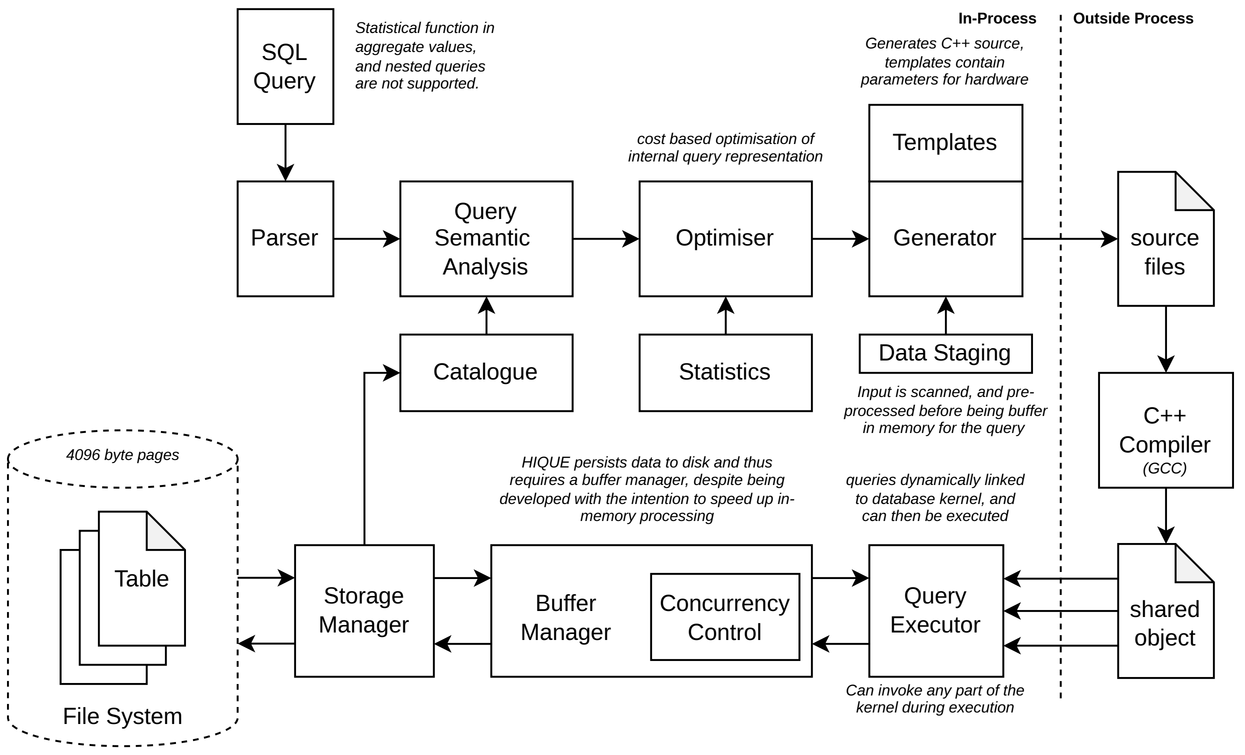
Visibility	The output is easily inspectable C++, and the compiler can optionally include debug information, or additional instrumentation when compiling queries to assist in debugging the engine, and the compiler itself verifies the type safety of code. While HIQUE was evaluated using GCC, it is possible 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
Templates
cost based optimisation of internal query representation
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
Can invoke any part of the kernel during execution
File System

Figure 5.3: An overview of the HIQUE system design.

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.

5.3 Incremental View Maintenance

5.3.1 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[10] 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

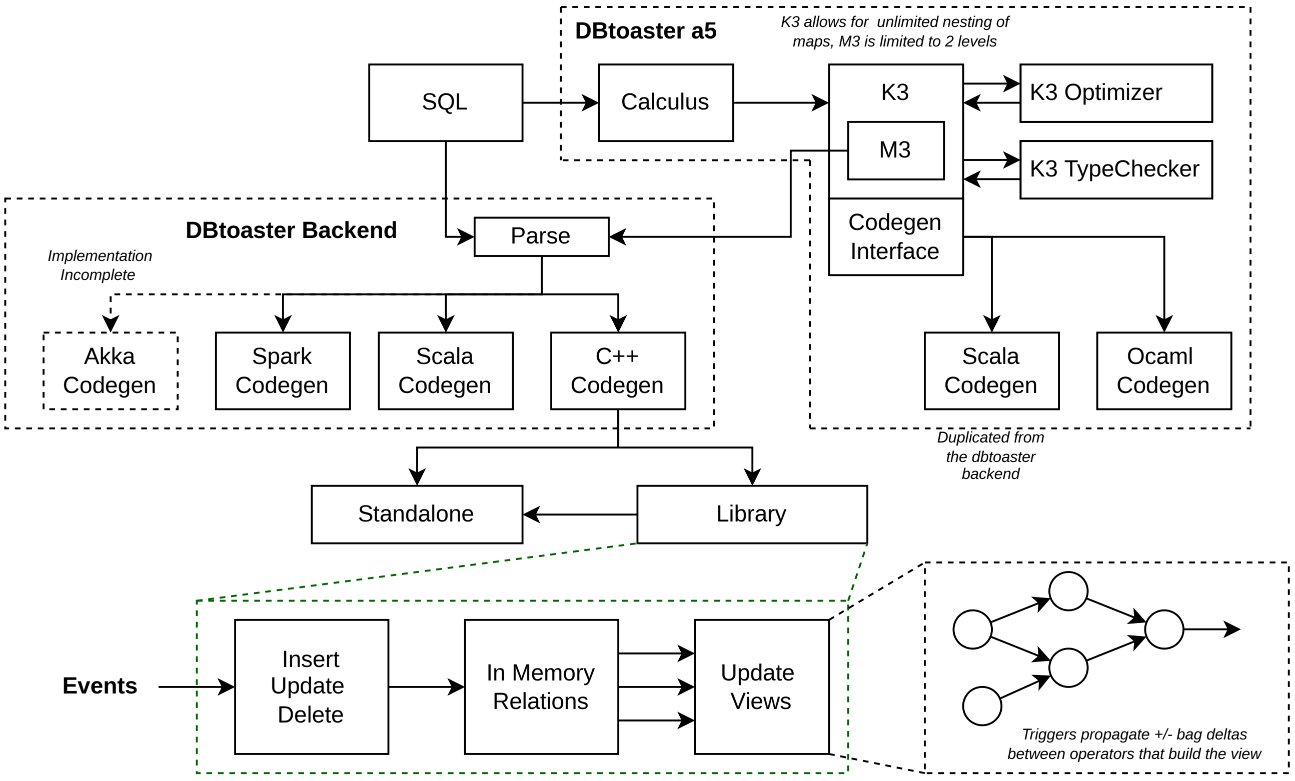


Figure 5.4: An overview of the DBToaster system design.

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

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. 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);
```

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

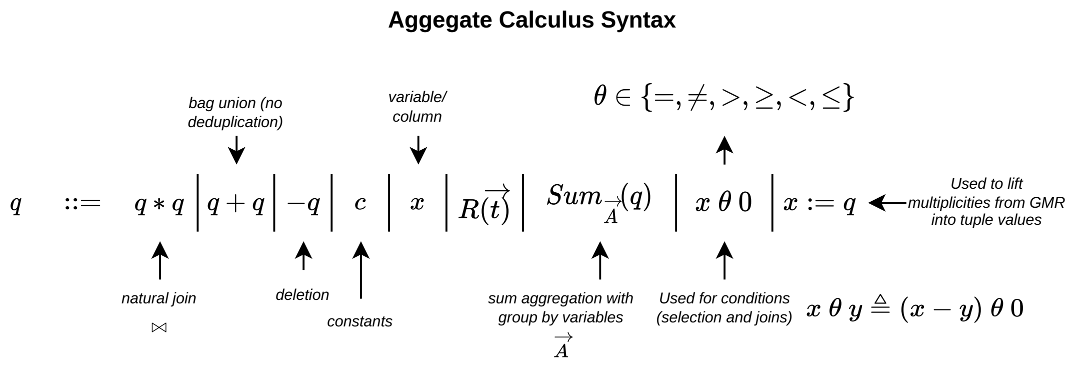

Aggegate Calculus Syntax
$\theta \in \{=, \neq, >, \geq, <, \leq\}$
bag union (no deduplication)
variable/ column
$q \ ::= \ q * q \mid q + q \mid -q \mid c \mid x \mid R(\vec{t}) \mid Sum_{\vec{A}}(q) \mid x\ \theta\ 0 \mid x := q$
Used to lift multiplicities from GMR into tuple values
natural join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

Figure 5.5: DBToaster AGCA syntax.

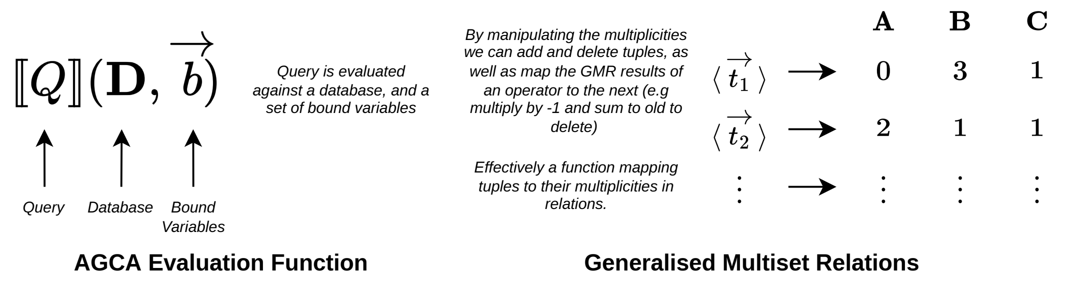

$[\![Q]\!](\mathbf{D}, \vec{b})$
Query is evaluated against a database, and a set of bound variables
Query
Database
Bound Variables
AGCA Evaluation Function
By manipulating the multiplicities we can add and delete tuples, as well as map the GMR results of an operator to the next (e.g multiply by -1 and sum to old to delete)
Effectively a function mapping tuples to their multiplicities in relations.
A B C
$\langle \vec{t_1} \rangle \longrightarrow$ 0 3 1
$\langle \vec{t_2} \rangle \longrightarrow$ 2 1 1
Generalised Multiset Relations

Figure 5.6: DBToaster AGCA Generalised Multiset Relations.

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

$$\begin{aligned}
\Delta(Q_1 + Q_2) &\equiv (\Delta Q_1) + (\Delta Q_2)\\
\Delta(Q_1 * Q_2) &\equiv ((\Delta Q_2) * Q_2) + (Q_1 * (\Delta Q_2)) + ((\Delta Q_1) * (\Delta Q_2))\\
\Delta(x := Q) &\equiv (x := (Q + \Delta Q)) - (x := Q)\\
\Delta(-Q) &\equiv -(\Delta Q)\\
\Delta(Sum_{\vec{A}} Q) &\equiv Sum_{\vec{A}}(\Delta Q)\\
\Delta(x\ \theta\ 0) \equiv \Delta x &\equiv \Delta c \equiv 0
\end{aligned}$$

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

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] Rust Repository in place collector for alloc::Vec. *Rust Library Team and Contributors*. 2024. URL: https://github.com/rust-lang/rust/blob/master/library/alloc/src/vec/in_place_collect.rs (visited on 06/12/2024).

[4] Jakub Beránek. *cargo-pgo github repository*. 2024. URL: https://github.com/Kobzol/cargo-pgo.

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

[6] *Copy Elision - Cppreference*. 2024. URL: https://en.cppreference.com/w/cpp/language/copy_elision.

[7] Transaction Processing Performance Council. *TPC-H Specification*. 2024. URL: https://www.tpc.org/TPC_Documents_Current_Versions/pdf/TPC-H_v3.0.1.pdf (visited on 06/12/2024).

[8] CrossDB. *CrossDB Website*. 2024. URL: https://crossdb.org/ (visited on 06/12/2024).

[9] DiffBlue. *CBMC: C Bounded Model Checker*. 2024. URL: https://github.com/diffblue/cbmc (visited on 06/12/2024).

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

[11] 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.

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

[13] H2O.ai. *H2O.ai db benchmark github site*. 2024. URL: https://h2oai.github.io/db-benchmark/ (visited on 06/12/2024).

[14] Kani. *Kani Github Repository*. 2024. URL: https://github.com/model-checking/kani.

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

[16] 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.

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

[18] DuckDB Labs. *DuckDB H2O.ai Benchmark*. 2024. URL: https://duckdb.org/2023/04/14/h2oai.html (visited on 06/12/2024).

[19] DuckDB Labs. *DuckDB Rowid Tables*. 2024. URL: https://duckdb.org/docs/sql/statements/select.html#row-ids (visited on 06/12/2024).

[20] Niko Matsakis. *Rayon: data parallelism in Rust*. 2024. URL: https://smallcultfollowing.com/babysteps/blog/2015/12/18/rayon-data-parallelism-in-rust/.

[21] Tyler Neely and Sled Contributors. *Sled Github Repository*. 2024. URL: https://github.com/spacejam/sled (visited on 06/12/2024).

[22] Thomas Neumann. "Efficiently compiling efficient query plans for modern hardware". In: *Proc. VLDB Endow.* 4.9 (June 2011), pp. 539–550. ISSN: 2150-8097. DOI: 10.14778/2002938.2002940. URL: https://doi.org/10.14778/2002938.2002940.

[23] Thomas Neumann. "Efficiently compiling efficient query plans for modern hardware". In: *Proc. VLDB Endow.* 4.9 (June 2011), pp. 539–550. ISSN: 2150-8097. DOI: 10.14778/2002938.2002940. URL: https://doi.org/10.14778/2002938.2002940.

[24] 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.

[25] 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.

[26] Speykious. *cve-rs github repository*. 2024. URL: https://github.com/Speykious/cve-rs/.

[27] Sqlite. *SQLite Rowid Tables*. 2024. URL: https://sqlite.org/rowidtable.html (visited on 06/12/2024).

[28] Sqlite. *The Lemon LALR(1) Parser Generator*. 2024. URL: https://sqlite.org/lemon.html (visited on 06/15/2024).

[29] Symas. *LMDB Website*. 2024. URL: https://www.symas.com/lmdb (visited on 06/12/2024).

[30] Rust Language Team. *Chalk: A Procedural Macro for Type Checking*. 2024. URL: https://rust-lang.github.io/chalk/book/.

[31] Rust Types Team. *Trait Solving in Rust*. 2024. URL: https://rustc-dev-guide.rust-lang.org/traits/resolution.html.