emDB: Embedded databases as schema compilers

Oliver Killane
Imperial College London
London, United Kingdom
ok220@ic.ac.uk

Holger Pirk
Imperial College London
London, United Kingdom
hlgr@ic.ac.uk

ChatGPT 4o
OpenAI
San Francisco, California
chat.openai.com

***Abstract*—In this paper, we introduce emDB - the easy macro database system embeddable in rust programs. EmDB challenges traditional assumptions about embedded databases, such as the need for a SQL interface, schema changes, and ad-hoc queries. By using a static schema and known parameterized queries, emDB moves query parsing, semantic analysis, and code generation to compile time, and eliminates the need for a complex execution engine.**

Implemented as a Rust procedural macro, emDB reads schemas and queries, selects appropriate data structures, reports type and query errors and generates optimized query code at compile time. This approach, based on a known set of queries, allows for the use of optimal data structures and integrates seamlessly with standard debugging, benchmarking, verification, and testing tools. EmDB also makes use of language-specific features like move semantics and borrow checking, providing an efficient and safe interface for developers, including support for user-defined types and code to be embedded into the database.

***Index Terms*—databases, compilers, rust, macros**

I. Introduction

Embedded data processing tools can be placed on a spectrum of abstraction.

At the lower end of the spectrum, manual implementation requires developers to implement operators, design data structures, and conduct thorough testing. While this approach demands considerable effort, it can yield optimal performance results due to the high level of control it offers.

At a higher abstraction, dataframes provide a set of predefined operators, simplifying the development process. However, users still need to compose operators manually. Notably, there is no compilation step in dataframe libraries that allows for an optimiser to determine data structure and operator implementation choices based on the entire query and schema context.

LINQ (Language Integrated Query) offers a higher level of query abstraction, including a SQL-like interface, yet users are still responsible for making data structure choices and selecting operator implementations.

These methods generally impose several restrictions on the user:

- The schema is static.
- All parameterized queries are known at application compile time.

Transitioning to a full SQL-like abstraction, which includes transactions, operator and data structure selection, and query optimization, there is a discontinuity. This level of abstraction removes the aforementioned restrictions at significant cost.

Moving query parsing, planning, optimization, as well as schema altering queries to application runtime incurs significant implementation complexity (catalogue management, runtime types information) and runtime overhead (for example

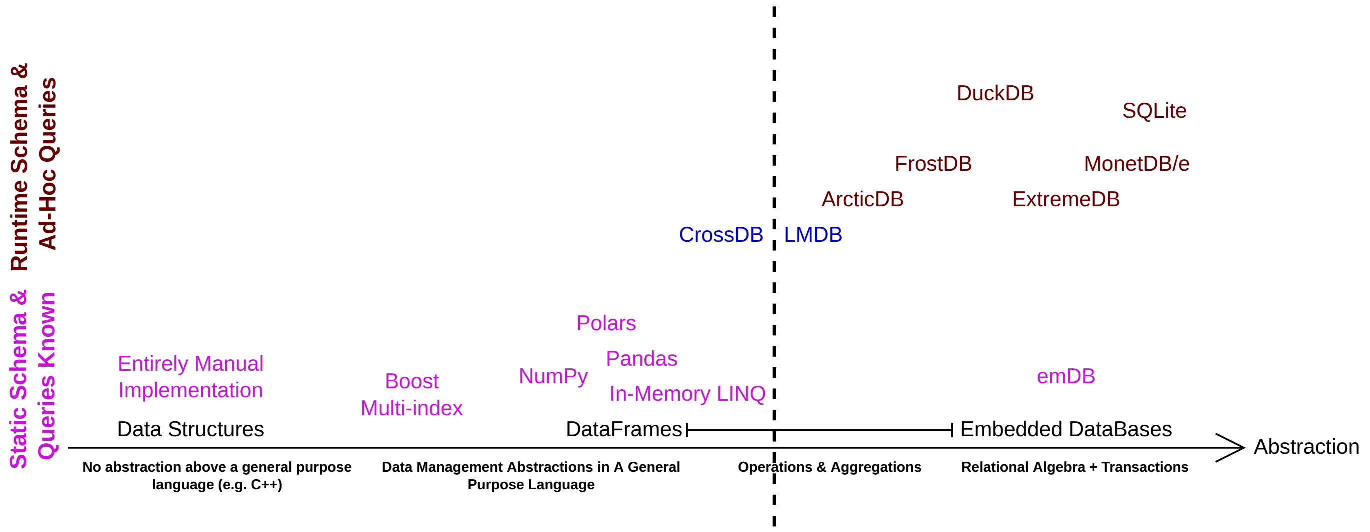
Runtime Schema & Ad-Hoc Queries
Static Schema & Queries Known
DuckDB
SQLite
FrostDB
MonetDB/e
ArcticDB
ExtremeDB
CrossDB
LMDB
Polars
Pandas
NumPy
In-Memory LINQ
Entirely Manual Implementation
Boost Multi-index
emDB
Data Structures
DataFrames
Embedded DataBases
Abstraction
No abstraction above a general purpose language (e.g. C++)
Data Management Abstractions in A General Purpose Language
Operations & Aggregations
Relational Algebra + Transactions

Fig. 1. Embedded databases on a spectrum of abstraction

query compilation in codegen, or virtual calls in interpreted queries).

Additionally some level of serialization/deserialization is needed to move data from the host application (i.e. statically typed, without a stable ABI) into the embedded database (i.e. dynamically typed, over SQL string interface and/or C-ABI).

Given for the vast majority of applications using embedded databases, the queries are known, and the schema does not change, we argue there is no clear rationale for this disjointness in the spectrum of abstraction, nor for the absence of a higher level of abstraction that still benefits from the restrictions present in lower levels.

II. Schema Compilation Advantage

A. *Eliminating the Cost of Query Compilation*

In typical systems, achieving an optimal balance between the time spent on code generation and query execution presents a significant challenge.

Query complication is used by a growing number of systems, and can perform better than interpreted queries.

A more expensive query compiler can perform more semantic checks of generated code (useful for correctness guarentees), compile higher level code (e.g. Rust or C++ rather than llvm, easier for development and debugging), and perform more optimisation. The tradeoff between the performance advantage of codegen, and the cost of compilation for a given query is lower, or even counterproductive for small (typical OLTP) queries.

There are several mitigations.

1) Ignore the problem for non-OLTP workloads, and target the system for use in OLAP workloads where this is not a weakness. ([1])
2) Allow the user to remove the problem using prepared statements.
3) Generate a lower-level representation for query code to compile (e.g. LLVM IR or asm, instead of C++), at the cost of removing checking of queries (semantic analysis) and more difficult implementation. ([2], [3], [4])
4) Reduce the level of optimisation performed, reduces the performance benefit of code generation.
5) Cache query compilations to take advantage of repeat OLTP queries, some complexity is introduced to manage the cache. ([5])
6) Hybrid execution of queries with an interpreter, before switching to the compiled version (a JIT-based system). ([6], [3])

These mitigations are often complex to implement, and require a careful balance of trade-offs between performance for different kinds of workloads.

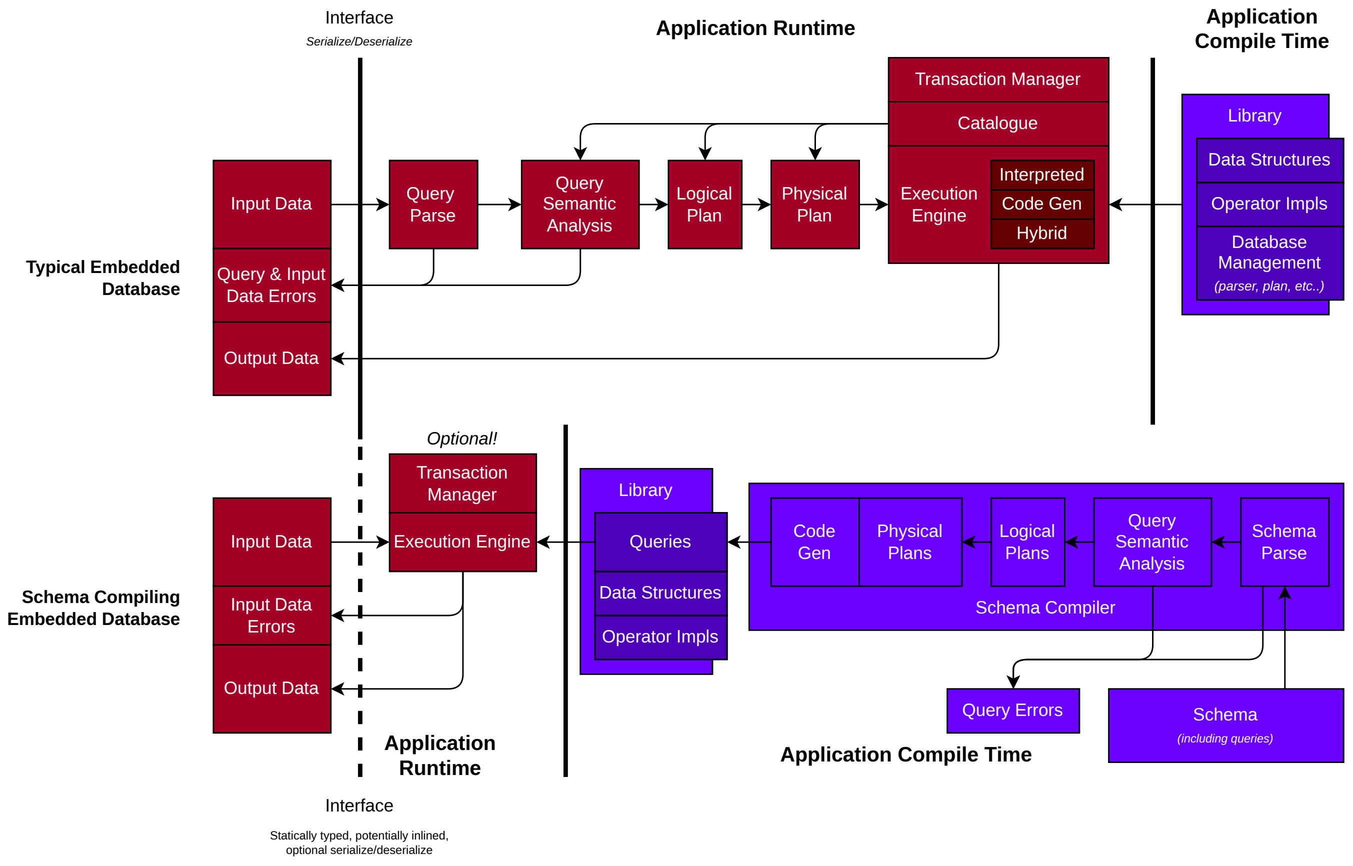

Fig. 2. A comparison of query lifespan for traditional and schema compililation based embedded databases

We choose to avoid tackling this hard problem. As we have the restriction that all parameterized queries are known, we can instead simply move query compilation from application runtime to application compile time.

This enables the maximal benefits of code generation (high level code, semantic analysis, expensive optimisations), with no runtime cost.

Furthermore as all queries and tables are generated at application compile time, the types of all data in all queries and tables are known, meaning no explicit runtime type information or catalogue is required. This also allows arbitrary types to used in the database (closures, custom user types, user data structures, smart pointers, etc.) and pure expressions (calling user defined functions) to be embedded into the database.

A significant difficulty with developing runtime codegen systems is tooling, for example requiring engine-integrated custom debuggers to be implemented. [4] Generating human readable code (compiled with the application) has auxiliary tooling benefits, it becomes easily usable with any tooling available for the language, such as debuggers, profilers, testing, benchmarking and verification tools - it's not a bespoke, complex runtime codegen system needing its own special tooling; it's *just* (mostly human readable) code.

B. Compile Time Errors

Query errors, both syntactic and semantic, can be detected and reported at compile time, and these errors can be propagated to the Integrated Development Environment (IDE). This capability significantly reduces the time required to check queries, without more compile-time complex debug database checking as in sqlx.

Furthermore, the return types for queries only need to include errors that can only be detected at runtime. These include violations of unique constraints or user-defined assertions. By restricting error handling to these specific cases, the overall complexity and overhead associated with query execution are minimized, thereby enhancing both developer productivity and system reliability.

```
table people {
    name: String,
    friend: Option<String>,
} @ [ unique(name) as unique_names ]

query catalogue_error() {
    use not_people // no such table
        |> filter( name.len() < 10 )
        |> collect(users)
        ~> return;
}
```

Fig. 3. A semantic error in IDE

This kinda of integration with the language (and IDEs), particularly reporting errors with correct spans, is difficult to implement in other languages that do not support procedural macros, or syntax extensions.

This is a primary reason for such a system not being implemented previously.

C. Data Structure Selection

Choosing the optimal data structure for tables is a critical aspect of database performance optimization. This choice is driven by the specific queries that will be executed.

For instance, identifying which columns of a table are immutable and which tables are append-only allows for significant optimizations. Immutable values can be stored in a data structure with pointer stability, enabling the return of references qualified by the lifetime of the database. This approach ensures efficient access and manipulation of data.

Additionally, append-only tables benefit from not requiring a generation counter for row references, eliminating the cost of storing and checking this.

III. Implementation

A. Rust Procedural Macros

Rust procedural macros are rust code that is run by the compiler, that takes rust tokens as input, and outputs rust tokens and diagnostics as output. There are very few restrictions on what procedural macros can do, they can interact with the OS (files, network, subprocesses).

They have previously been used for non-rust embedded languages.

- Multiple rust frontend frameworks make use of html-like embedded languages (containing rust expressions) (yew, leptos), or custom interface declaring DSLs (rsx for dioxis).
- Programming languages such as C (using inline_c which invokes the system's toolchain compile and then link), or lisp (AST generated at compile time by rust_lisp).
- sqlx (checking SQL queries using a test database with migrations applied, at application compile time).

Tokenstreams contain no semantic information, this can be generated using the built-in rustc-interface library (re-invoking compiler stages on some tokenstream to extract THIR (types), MIR (borrow checking) information). For emDB we instead pass rust expressions through to the generated code, but maintain the original spans so semantic errors generated errors in user's embedded code (expressions and types) are attributed to the correct location in the user's code.

For emDB we developed our own basic parser-combinators library to efficiently parse tokenstreams, and produce multiple syntactic errors (with a simple parser, and simple (error-node type free) AST).

B. emQL Language

EmDB uses its own query language, emQL, to define queries.

Using standard compliant SQL poses limitations when embedding types and expressions from a host programming language. To address this, emDB employs its own query language, emQL, for defining queries.

While standard SQL has the advantage of extensive related tooling (such as fuzzers) and widespread familiarity as the most popular query language, it also comes with inherent disadvantages, often referred to as ”SQL smells.” These include unintuitive null semantics and the outward-in order of operators. By opting for a custom query language, emQL, emDB circumvents these issues, leveraging the syntax and semantics of Rust to enhance usability and integration. For instance, emQL uses Rust keywords to benefit from Rust's syntax highlighting.

EmQL is designed around streams, addressing a common complaint about SQL's unintuitive operator ordering. In contrast to SQL's outward-last, inner-first processing order, emQL follows a more intuitive first-in, first-out approach. Additionally, emQL incorporates row references as a fundamental part of the language, a feature supported as extensions in some SQL implementations (e.g. SQLite) but not universally (e.g., not in DuckDB). Row references are particularly advantageous for many OLTP workloads, enabling the database to generate unique identifiers.

Furthermore, emQL's contexts facilitate the convenient use of parameters from streams, analogous to constructs like groupby or loop iterations in programming languages. Queries in emQL are similar to SQL transactions, with each query potentially containing many different table access and/or mutation operations.

C. *Table Implementation*

emDB employs a rollback log for each query or transaction to ensure data integrity and support transactional operations. Each table in emDB consists of a primary subtable and multiple associated subtables, which can contain several columns marked as mutable or immutable. This structure allows for different column data types, with the primary subtable performing index checks on access, enabling various levels of decomposition to be implemented.

Tables in emDB support row predicates, unique constraints, and limit constraints, each maintained through a rollback log to facilitate transactions. Rows are accessed using row references, which serve as fast O(1) indices into the table, validated through the primary column. This efficient access mechanism is particularly beneficial for OLTP workloads that involve frequent access using generated keys, enhancing performance and reliability in transactional environments.

D. *Pull versus Push*

For code generation, a push-based model proves to be convenient. By utilizing lazily evaluated iterators between operators, we can effectively implement a pull-based model. In this approach, `Iterator::next()` calls to Rust iterators can be inlined, and values are collected at pipeline-breaking operators, such as sort. Despite this, the underlying model used remains push-based. This model is also used by Hyper [2].

IV. Evaluation

References

[1] K. Krikellas, S. D. Viglas, and M. Cintra, “Generating code for holistic query evaluation,” in *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, 2010, pp. 613–624.

[2] T. Neumann, “Efficiently compiling efficient query plans for modern hardware,” *Proc. VLDB Endow.*, vol. 4, no. 9, p. 539–550, jun 2011. [Online]. Available: https://doi.org/10.14778/2002938.2002940

[3] A. Kohn, V. Leis, and T. Neumann, “Adaptive execution of compiled queries,” in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, 2018, pp. 197–208.

[4] ——, “Making compiling query engines practical,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 33, no. 2, pp. 597–612, 2021.

[5] J. Chen, S. Jindel, R. Walzer, R. Sen, N. Jimsheleishvilli, and M. Andrews, “The memsql query optimizer: a modern optimizer for real-time analytics in a distributed database,” *Proc. VLDB Endow.*, vol. 9, no. 13, p. 1401–1412, sep 2016. [Online]. Available: https://doi.org/10.14778/3007263.3007277

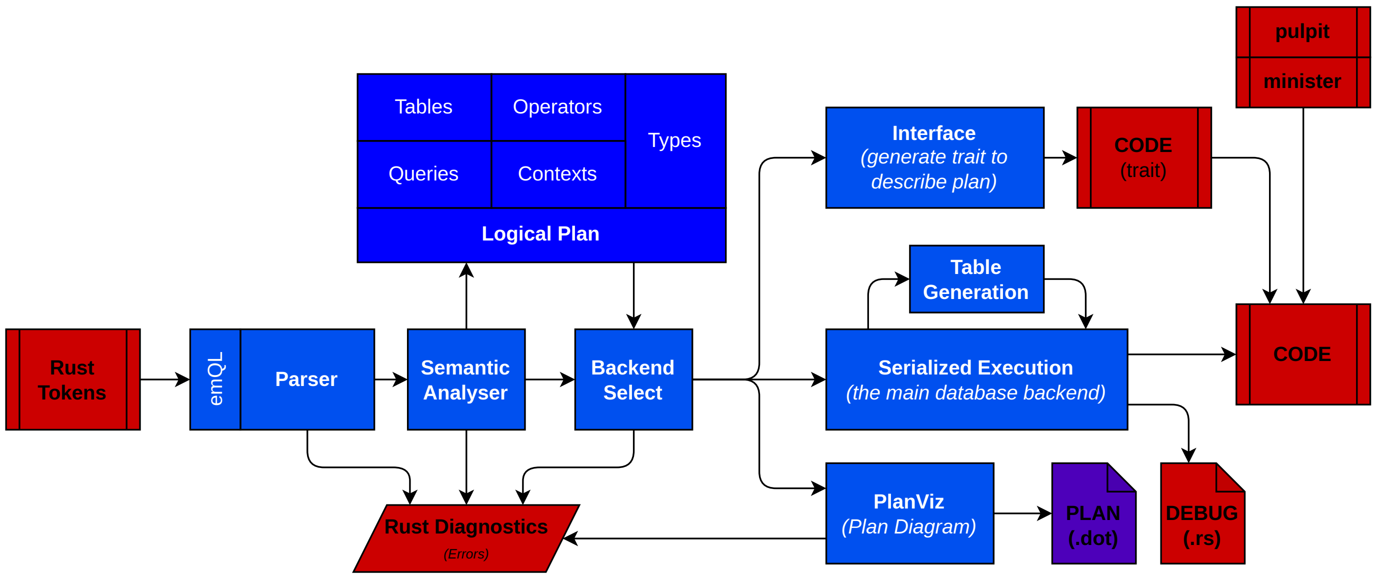

Fig. 4. The full emDB system

[6] B. Wagner, A. Kohn, P. Boncz, and V. Leis, “Incremental fusion: Unifying compiled and vectorized query execution.” ICDE, 2024.

```
query foobars(max_bar: usize) {
    use foos
        |> filter(bar < max_bar)
        |> map(
            foobar: super::UserType = {
                let logic = /* complex expressions */;
                super::user_defined_function(logic, &bar)
            },
            zing: usize = bar + 1,
          )
        |> sort(zing asc)
        |> lift(
            /* .. some subquery */
          )
        |> collect(foobars)
        ~> return;
}
```

Fig. 5. Some examples of emQL syntax