

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

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 pre-defined 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

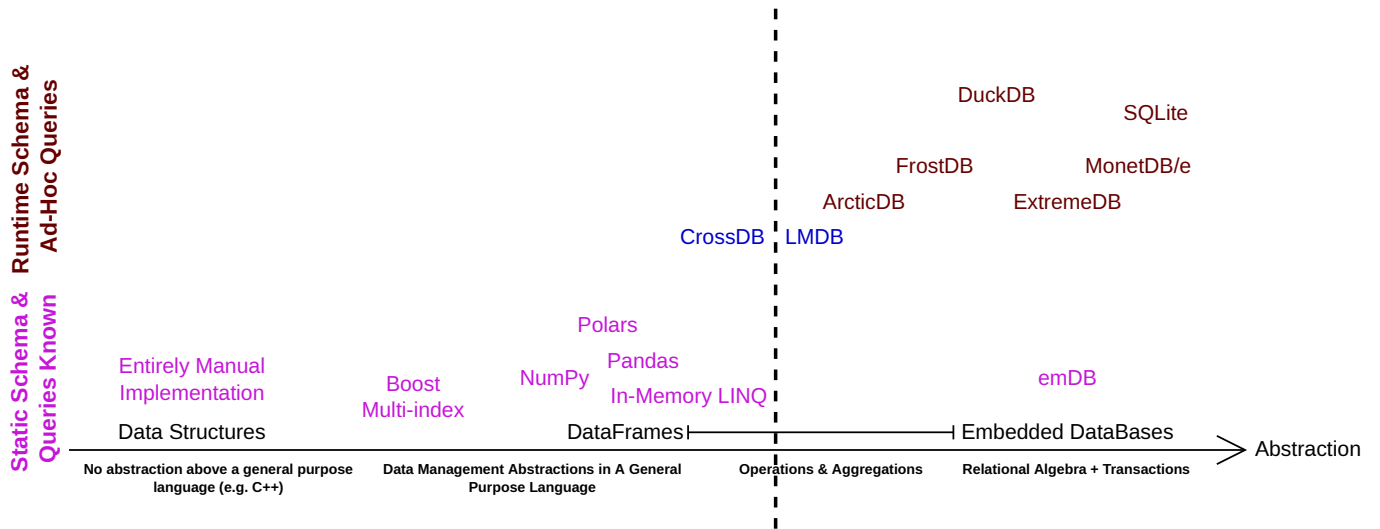


Fig. 1. Embedded databases on a spectrum of abstraction

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 COMPILE 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 compilation 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 guarantees), 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) 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])
- 3) Reduce the level of optimisation performed, reduces the performance benefit of code generation.
- 4) Cache query compilations to take advantage of repeat OLTP queries, some complexity is introduced to manage the cache. ([5])
- 5) 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.

We choose to avoid tackling this hard problem. As we have the restriction that all parameterized queries are known, we

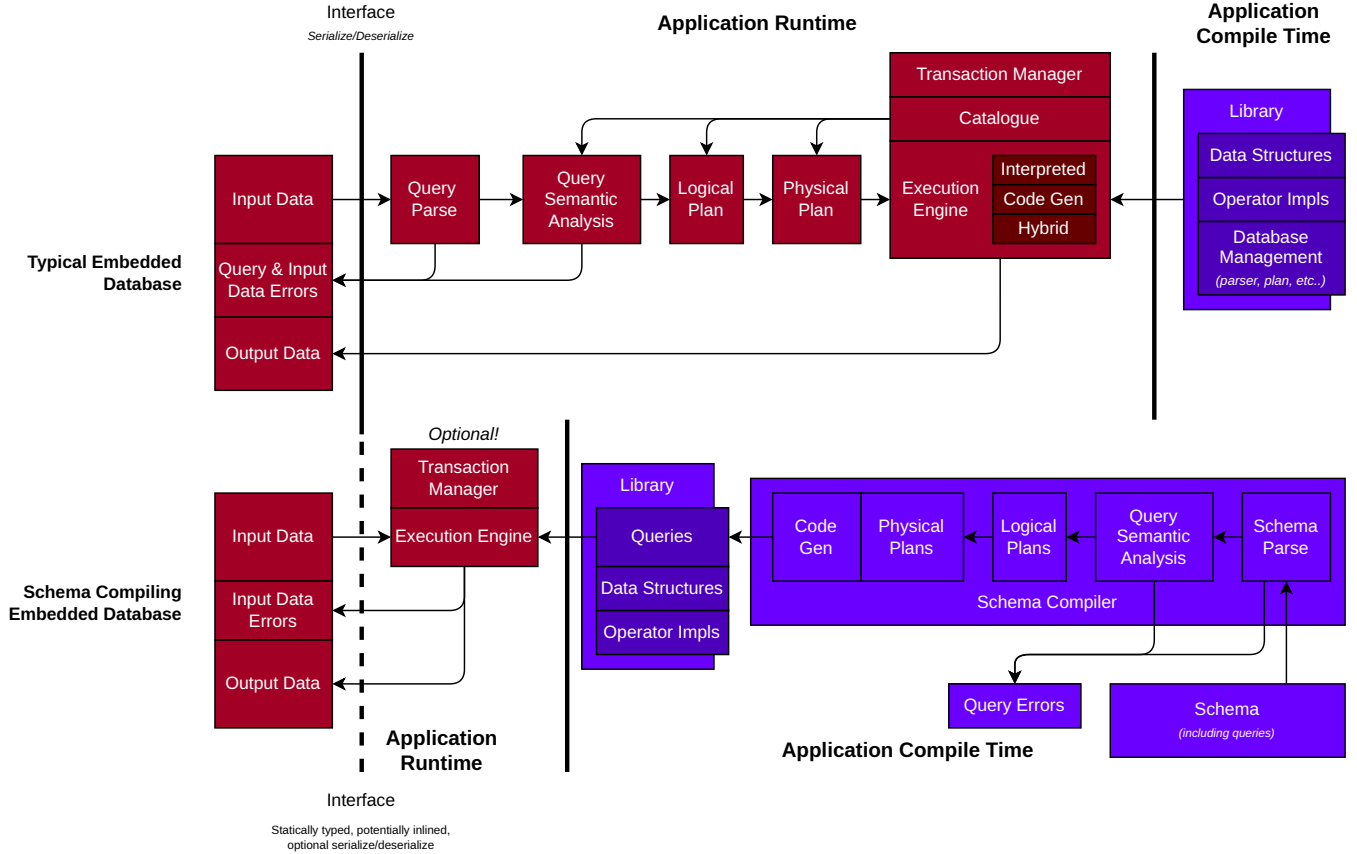


Fig. 2. A comparison of query lifespan for traditional and schema compilation based embedded databases

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 be 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.

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

B. emQL

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. Jimshelishvili, 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>
- [6] B. Wagner, A. Kohn, P. Boncz, and V. Leis, "Incremental fusion: Unifying compiled and vectorized query execution." *ICDE*, 2024.

```

1 table people {
2     name: String,
3     friend: Option<String>,
4 } @ [ unique(name) as unique_names ]
5
6 query catalogue_error() {
7     use not_people // no such table
8     |> filter( name.len() < 10 )
9     |> collect(users)
10     ~> return;
11 }

```

Fig. 3. A semantic error in IDE

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.

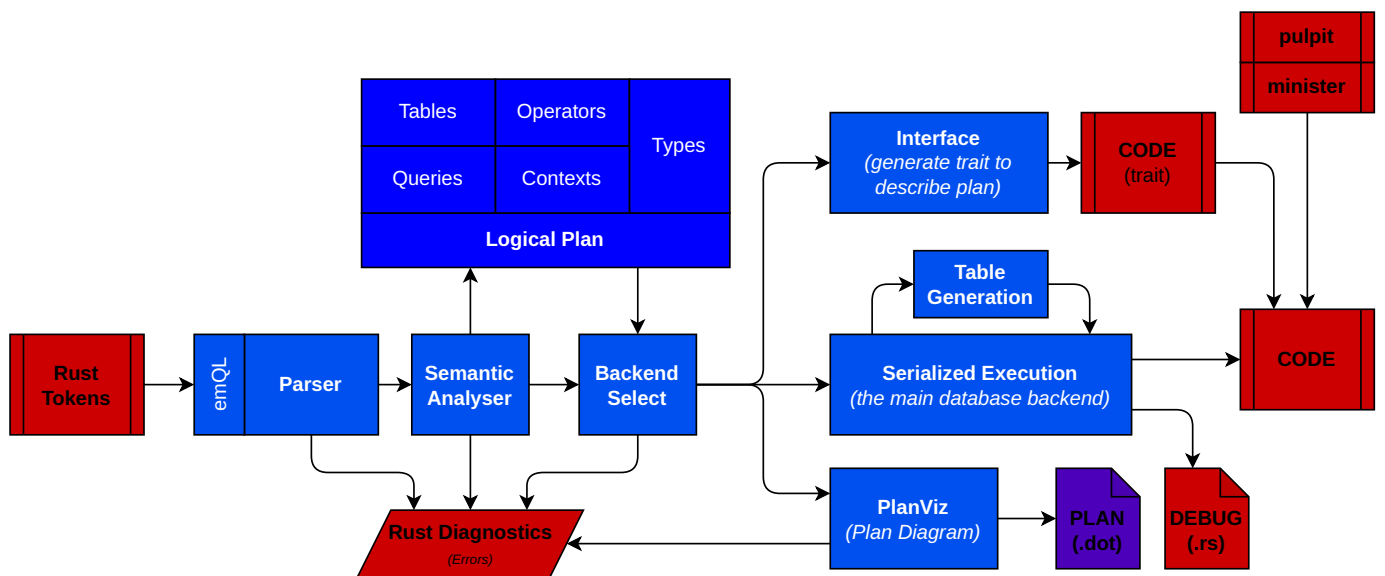


Fig. 4. The full emDB system