School of Computer Science and Engineering

Faculty of Engineering

The University of New South Wales

pgx-lower: Bridging Modern Database Compiler Advances With Battle-Tested Database Systems

by

Nicolaas Johannes van der Merwe

Thesis submitted as a requirement for the degree of

Bachelor of Computer Science (Honours)

Submitted: November 2024

Supervisor: Dr Zhengyi Yang Student ID: z5467476

pgx-lower: Bridging Modern Database Compiler Advances With Battle-Tested Database Systems

Nicolaas Johannes van der Merwe

Abstract

Abbreviations

ACID Atomicity, consistency, isolation, durability

AST Abstract Syntax Tree

CPU Central Processing Unit

DB Database

EXP Expression (expressions inside queries)

IR Intermediate Representation

JIT Just-in-time (compiler)

JVM Java Virtual Machine

LLC Last Level Cache

MLIR Multi-Level Intermediate Representation

QEP Query Execution Plan

RA Relational Algebra

SQL Structured Query Language

SSD Solid State Drive

TPC-H Transaction Processing Performance Council

Contents

List of Figures

Chapter 1

Introduction

Databases are a heavily used type of system that rely on correctness and speed. Nowadays, they are often the primary bottleneck in many systems - especially on webservers and other large data applications [Kle19].

With modern hardware advances, the optimal way to structure these databases has drastically changed, but most databases are using architectures defined by older hardware. Older databases assume the disk operations are the vast majority of runtime, but that has shifted to the CPU for heavy queries.

These projects typically create standalone databases, but that means that distribution becomes harder and the projects need to implement their own database as well, which might require a large number of additional steps for serious projects. To productionise the system, this might include implementing ACID, MVCC, query plan optimisation, and more. By using an established database, we can address this issue.

pgx-lower replaces PostgreSQL's execution engine with a LingoDB's compiler to bridge the gap of modern compilers with established systems. PostgreSQL's extension system is utilised to override the executor, and shows there are features that can be used within PostgreSQL that can assist with this research. One concern, however, is the additional complexities in implementation and testing.

This thesis is separated into a background in Chapter 2 which includes fundamental concepts and the definition/goal of the project, then a light literature survey will be conducted in Chapter **??**. The project's solution will be introduced in Chapter 3, and finally conclusions will be drawn in Chapter 4.

Chapter 2

Background

2.1 Fundamentals

2.1.1 Database Background

The majority of databases are structured like Figure 2.1. Structure Query Language (SQL) is parsed, turned to RA (relational-algebra), optimized, executed, then materialized into a table.

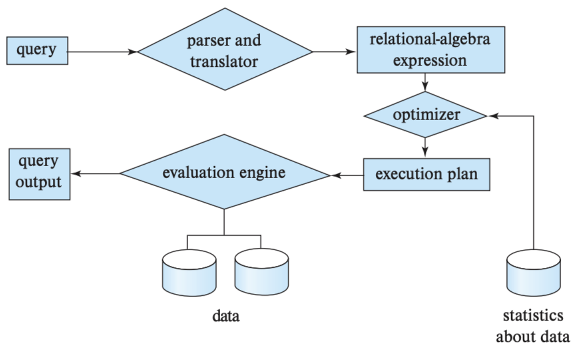

Figure 2.1: Database Structure
[SKS19]

For non-compiler databases they use a volcano operator model tree, such as Figure 2.2The root node has a `produce()` function which calls its children's `produce()`, until it calls a leave node, which calls `consume()` on itself, then that calls its parent's `consume()` function. In other words, a post-order traversal through this tree where tuples are dragged upwards.

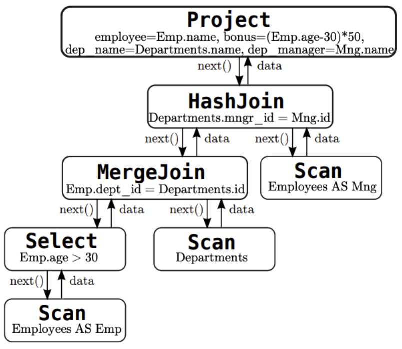

Figure 2.2: Volcano operator model tree.
[ZVBB11]

The fundamental issue with this classical model is that it is heavily under utilising the hardware. If we are only pulling up a single tuple, our CPU caches are barely used. This has lead to the vectorized execution model and the compiled model. With the vectorized model, we pull up groups of tuples instead. However, this leads to problems where it's common to have too many copy operations instead of having a pointer going upwards. For instance, if a sort or a join allocates new space that is too much for the cache, the handling can become poor. With the compiled approach, it introduces a lot of implementation complexity.

Relational databases prioritise ACID requirements - Atomicity, Consistency, Isolation and Durability [SKS19]. This is a critical requirement in this type of system, and usually one of the main reasons people pick a relational database. Atomicity refers to transactions are a single unit of work, consistency means it must be in a valid state before and after the query, isolation means concurrent transactions do not interact with each other, and durability means once something is committed it will stay committed. [SKS19]

It is common for on-disk databases to consider the cost of CPU operations to be $O(1)$ [SKS19]. This is partially due to when these systems were made, the disks were much slower and the caches were much slower. In part A of this project, this was disproved for PostgreSQL as it was found that the time spent in the CPU was substantial: between 34.87% and 76.56% with an average of 49.32% across the tested queries.

2.1.2 JIT Background

Just-in-time (JIT) compilers function by having multiple layers of compilation and are mostly used by interpreted languages to eliminate the ill-effects on performance [ZVBB11]. Advanced compilers can run the primary program, then dedicate some background threads to improving the optimisation of the code, and swap it over to the optimized version when it is ready [KLN18].

Due to branch-prediction optimization, JIT compilers can become faster than ahead of time compilers. In 1999, a benchmarking paper measured four commercial databases and found $10\%-20\%$ of their execution time was spent fixing poor predictions [ADHW99]. similarly, research specifically into branch prediction has said, "although branch prediction attained 99% accuracy in predicting static branches, ... branches are still a major bottleneck in the system performance" [Jos21]. Modern measurements still find 50% of their query times are spent resolving hardware hazards, such as mispredictions, with improvements in this area making their queries 2.6x faster [Ker21]. The Azul JIT compiler measured that their JIT solution's speculative optimizations can lead up to 50% performance gains [Azu22].

In the context of databases, most compilers can be split into only compiling expressions (typically called EXP for expression), and others that compile the entire Query Execution Plan (QEP) [MYH+24]. Within PostgreSQL itself, they have EXP support using $llvm - jit$.

The LLVM Project is a compiler infrastructure that supports making compilers so that common, but complex, compiler optimisations do not have to be re- implemented. Multi-Level Intermediate Representation is another, newer toolkit that is tightly coupled with the LLVM project. It adds a framework to define dialects, and lower through these dialects. One of the primary benefits of this is if you make a compiler, you can define a high level dialect, then another person can target your custom high-level dialect.

2.1.3 PostgreSQL Background

An arena allocator is a data structure that supports allocating memory and freeing the entire data structure. This improves memory safety by consolidating allocations into a single location. Within PostgreSQL, memory contexts are used which is an advancement of this concept. There is a set of statically defined memory contexts (TopMemoryContext, TopTransactionContext, CurTransactionContext, Transaction-Context, they are defined in the mmgr README), and with these you can create child contexts. When a context is freed, all the child contexts are also freed.

PostgreSQL defines query trees, plan trees, plan nodes, and expression nodes. A query tree is the initial version of the parsed SQL, which is passed through the optimiser which is then called a plan tree. The nodes in these plan trees can broadly be identified as plan nodes or expression nodes. Plan nodes include an implementation detail (aggregation, scanning a table, nest loop joins) and expression nodes consist of individual operations (binaryop, null test, case expressions).

2.1.4 Database Benchmarking

Need to include information here about common benchmarks, and how the industry has gone towards TPC-H.

2.2 Related Works

This following section summarises relevant works in the compiled queries space, and their architectures. Originally, the industry began with compiled query engines, but this was overtaken by volcano models as they simplified the implementation details with little cost at the time. However, now analytical engines are examining compilers again.

In subsection 2.2.2 system R will be explored as the classical example, followed by HyPer and Umbra in subsection 2.2.3 and subsection 2.2.4 which re-introduced the concept. Mutable in subsection 2.2.5 and LingoDB in subsection 2.2.8 are research databases. Lastly, PostgreSQL will be examined in subsection 2.2.1 as it uses expression-based compilation and there has been an attempt to create a compiled engine before.

2.2.1 PostgreSQL and Extension Systems

PostgreSQL is a battle-tested system and is currently the most popular database in the world with 51.9 of developers in a stackoverflow survey saying they use it extensively in 2024. Within the context of compiled queries this means the database itself cannot be treated as a research system. Changes directly to it requires heavy-testing, but also, these changes will not be peer-reviewed research. Instead, it is pull-requests online with more casual interaction.

There has been significant discussion about HyPer and JIT with regards to PostgreSQL in 2017. The general response is doubt that someone will add support for full compiling full query expressions, and rearchitecting such a core component introduces large risk.

However, in September 2017 Andres Freudn started implementing JIT support for expressions. The reasoning was that most of the CPU time is in the expression components, (e.g. y ¿ 8 in SELECT * from table WHERE x ¿ 8;). Furthermore, there are significant benefits to tuple deformation as it interacts with the cache and has poor branch prediction.

In the pull request Peter Eisentruat asked whether the default JIT settings are too low, but in version 11 of PostgreSQL they went ahead with the release but with the JIT disabled by default. This didn't get much usage, and they decided that enabling it by default it would give it exposure and testing. However, when released, the United

Kingdom's critical service for a COVID-19 dashboard automatically updated and spiked to a 70% failure rate as some of their queries ran 2,229x slower. This affected the general reception that JIT features should be disabled by default, and has lead to people having negative opinions about JIT and compiled queries.

Two cases where QEP query compilation with PostgreSQL was implemented were found. The first is Vitesse DB, which made a series of public posts about getting people to assist with testing it. They became generally available in 2015, but their website is offline now and there is not much mention of them. The second was at a PgCon presentation and achieved a 5.5x speedup on TPC-H query 1, and has more documentation. However, they did not publicize their methods or show that it's easy for people to use.

Other database systems also support extensions, and there are many systems that rely on PostgreSQL's extension system. MySQL, ClickHouse, DuckDB, Oracle Extensible Optimizer all support similar operations. This means more than only PostgreSQL can be extended in this same manner rather than creating databases from scratch.

2.2.2 System R

System R is a flagship paper in the databasing space that introduced SQL, compiling engines, and ACID. Their vision described ACID requirements, but was explained as seven dot points as it was not a concept yet. Their goal was to run at a "level of performance comparable to existing lower-function database systems." Reviewers commented that the compiler is the most important part of the design.

Due to the implementation overhead of parsing, validity checking, and access path selection, a compiler was appealing. These were not supported within the running transaction by default, and they leveraged pre-compiled fragments of Cobol for the reused functions to improve their compile times. This was completely custom-made at the time because there were not many tools to support writing compilers. System R shows the idea of compiled queries is as old as databases, and over time the priorities of the systems changed.

2.2.3 HyPer

HyPer was a flagship system, and Umbra supersedes it. Both were made by Thomas Neumann, and the core sign is of its viability is that HyPer was purchased by ableau in 2016 to be used in production. This shows that it is is possible to use an in-memory JIT database at scale. The project began in 2010, with their flagship paper releasing in 2011 for the compiler component, and in 2018 they released another flagship paper about adaptive compilation. However, the database being commercialised poses issues for outside research because the source code is not accessible, but there is a binary on their website that can be used for benchmarking.

Their 2011 paper on the compiler identifies that translating queries into C or C++ introduced significant overhead compared to compiling into LLVM. As a result, they suggested using pre-compiled C++ objects of common functions then inlining them into the LLVM IR. This LLVM IR is executed by the LLVM's JIT executor. By utilising LLVM IR, they can take advantage of overflow flags and strong typing which prevent numerous bugs in their original C++ approach.

HyPer shows they reduced their compile time by doing this in figure XYZ by many multiples, and in figure ABC they show they achieve many times less branches, branch mispredicts, and other measurements compared to their baseline of MonetDB. The cause of this is HyPer's output had less code in the compiled queries.

Hyper continues on in 2018 where they separated the compiler into multiple rounds. They introduced an interpreter on the byte code generated from LLVM IR, then they can run unoptimised machine code, and on the final stage they can run optimised machine code. Figure XYZ visualises this with the compile times of each stage. However, they had to create the byte code interpreter themselves to enable this.

The 2018 paper also improved their query optimisation by adding a dynamic measurement for how long live queries are taking. This is because the optimiser's cost model did not lead to accurate measurements for compilation timing. Instead, they introduced an execution stage for workers, then in a "work-stealing" stage they log how long the job took. With a combination of the measurements and the query plan, they calculate estimates for jobs and optimal levels to compile them to.

This was benchmarked with TPC-H Query 11 using 4 threads, and they found the adaptive execution was faster than only using bytecode by 40%, unoptimised compilation by 10% and optimised compilation by 80%. The cause of this is that the compilation stage is single threaded, while with multiple threads they can compile in the background while execution is running.

Utilising additional stages of the LLVM compiler, improving the cost model, and supporting multi threading the compilation and execution combined into a viable JIT compiled-query application. The primary criticism is that they effectively wrote the JIT compiler from the ground-up, which requires large amounts of engineering time. Majority of the additions here are not unique to a database's JIT compiler, and are mostly ways to target the compiler's latency.

2.2.4 HyPer

Umbra was created in 2020 by Thomas Neumann, the creator of HyPer, and the main change is that they show it is possible to use the in-memory database concepts from HyPer inside of an on-disk database. The core reason for this is the recent improvements of SSDs and buffer management advances. They take concepts from LeanStore for the buffer management and latches, then multi-version concurrency, compilation, and execution strategies from HyPer. This combination led to an on-disk database that is scalable, flexible and faster than HyPer.

A novel optimisation they introduced later was enabling the compiler to change the query plan. That is, they can use the metrics collected during execution to swap the order of joins, or the type of join being used. This improved the runtime of the data-centric queries by two-times. Some other databases introduce this concept by invoking the query optimiser multiple times, but since their compiler is invoked multiple times during execution this adds additional benefit.

Umbra is currently ranked as the most effective database on Clickhouse's benchmarking. The main complaint of the compiler being too heavy is still there, but it shows the advantage of having direct access to the JIT compiler with its adaptive compilation to change optimisation choices.

2.2.5 Mutable

In 2023, Mutable presented the concept of using a low latency JIT compiler (WebAssembly) rather than a heavy one in their initial paper. Its primary purpose, however, is to serve as a framework for implementing other concepts in database research so that they do not need to rewrite the framework later. However, using WebAssembly meant they can omit most of the optimisations that HyPer did while maintaining a higher of performance. Furthermore, they have a minimal query optimiser and instead rely on the V8 engine.

The V8 engine contains a "Liftoff" component that adds an early-stage execution step to lower the initial overhead of running the query. The liftoff component produces machine code as fast as possible while skipping opimisations, then "turbofan" is a second-stage compiler that runs in the background while execution is running. However, HyPer has a direct bytecode interpreter which can result in a lower time to execution.

Mutable's benchmarks show they achieve similar compile and execution times to HyPer, and outperform them in many cases. While pushing this system to the same performance as HyPer or Umbra would require re-architecting, the achieved performance compared to the implementation effort is a significant outcome.

2.2.6 LingoDB

LingoDB piloted in 2022 and proposed using the MLIR framework to create the optimisation layers. In most databases, the system parses the SQL into a query tree, then relational algebra, then this is optimised using manual implementations, and parse this into a plan tree for execution, or compile this into a binary. With MLIR, this pipeline changes into parsing the plan tree into a high-level dialect in MLIR, then doing optimisation passes on the plan itself, and the LLVM compiler can be used directly to turn this into LLVM, streamlining the process.

The LingoDB architecture can be seen in figure XYZ, which begins by parsing the SQL into a relational algebra dialect. These dialects are defined using MLIR's dialect system, and supported through code generation. Their compiler is defined by a relational algebra dialect, a database dialect that represents SOMETHING, a DSA dialect that represents, a utility dialect that represents SOMETHING, and the final LLVM output. This splits the state of the intermediate representation into three stages: relational algebra, a mixed dialect, and finally the LLVM.

Their result is that they are less performant than HyPer, but do better than DuckDB. This performance is not their key output, rather, it is that they can implement the standard optimisation patterns within the compiler. Another feat is that they are approximately 10,000 lines of code in the query execution model, and Mutable is at 22,944 lines for their code despite skipping query optimisation. Within LingoDB's paper they also compare this to being three times less than DuckDB and five times less than NoisePage.

In later research, LingoDB also explores obscure operations such as GPU acceleration, using the Torch-MLIR project's dialect, representing queries as sub-operators for acceleration, non-relational systems, and more. For our purposes, the appealing part of their architecture is that they use pg_query to parse the incoming SQL, which means their parser is the closest to PostgreSQL's. This will be explored in the design in REFERENCE.

2.2.7 Benchmarking

DuckDB, HyPer, Mutable LingoDB, and PostgreSQL were compared in a benchmark. This is visible in figure XYZ. It showed that PostgreSQL was signficantly slower than the others due to it being an on-disk database. Furthermore, analysis was done on PostgreSQL using `perf` to measure how much time was spent inside of the CPU.

2.2.8 Gaps in Literature

A core gap is the extension system within existing database. HyPer and Umbra managed to commercialise their systems, but the other databases are strictly research systems and some do not support ACID, multithreading, or other core requirements such as index scans. Michael Stonebraker, a Turing Award recipient and the founder of PostgreSQL writes that a fundamental issue in research is that they have forgotten the usecase of the systems and target the 0.01% of users. These commercial databases reaching high performance is a symptom of this. Testing the wide variety of ACID requirements is a significant undertaking.

The other issue is writing these compiled query engines is a large undertaking, and the core reason why vectorised execution has gained more popularity in production systems. Debugging a compiled program within a database is challenging, and while solutions have been offered, such as Umbra's debugger, it is still challenging and questionable how transferable those tools are.

Relying on an extension system such that it's an optional feature means users can install the optimisations, and tests can be done with production systems without requesting pull requests into the system itself. Since these are large source code changes, it adds political complexity to have the solution added to the official system without production proof of it being used. The result of this would be an useable compiler accelerator, that can easily be installed into existing systems, and once used in many scenarios is easier to add to the official system.

2.3 Aims

Tying this together, this piece aims to integrate a research compiler into a battle-tested system by using an extension system. This addresses the gap of these systems being difficult to use widely, and potential to integrate it into the original system once stronger correctness and speed optimisations have been shown.

A key output is showing that the system can operate within the same order of magnitude as the target system. The purpose of this is to ensure other optimisations can be applied to fit the surrounding database later, but the expectation is not to be faster than it.

One concern is these databases are large systems while the research systems are smaller. This increases the testing difficulty because a complete system has more variables, such as genetic algorithms in the query optimiser that makes performance non-deterministic. To counter this, a large number of benchmarks can be executed, and a standard deviation can be calculated.

Chapter 3

Project

Lorem Ipsum

Chapter 4

Conclusion

Lorem Ipsum

Acknowledgements

This work has been inspired by the labours of numerous academics in the Faculty of Engineering at UNSW who have endeavoured, over the years, to encourage students to present beautiful concepts using beautiful typography.

Further inspiration has come from Donald Knuth who designed TEX, for typesetting technical (and non-technical) material with elegance and clarity; and from Leslie Lamport who contributed LATEX, which makes TEX usable by mortal engineers.

John Zaitseff, an honours student in CSE at the time, created the first version of the UNSW Thesis LATEX class and the author of the current version is indebted to his work.

Lastly my supervisor and assessor for supporting me throughout this project.

Bibliography

[ADHW99] Anastassia Ailamaki, David J DeWitt, Mark D Hill, and David A Wood. Dbmss on a modern processor: Where does time go? In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 266–277, 1999.

[Azu22] Azul. Jit performance: Ahead-of-time versus just-in-time. `https://www.azul.com/blog/jit-performance-ahead-of-time-versus-just-in-time/`, 2022. Accessed: 2025-11-08.

[Jos21] Rinu Joseph. A survey of deep learning techniques for dynamic branch prediction. *arXiv preprint arXiv:2112.14911*, 2021.

[Ker21] Timo Kersten. *Optimizing Relational Query Engine Architecture for Modern Hardware.* PhD thesis, Technische Universität München, 2021.

[Kle19] Martin Kleppmann. Designing data-intensive applications, 2019.

[KLN18] André Kohn, Viktor Leis, and Thomas Neumann. Adaptive execution of compiled queries. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 197–208, 2018.

[MYH$^+$24] Miao Ma, Zhengyi Yang, Kongzhang Hao, Liuyi Chen, Chunling Wang, and Yi Jin. An empirical analysis of just-in-time compilation in modern databases. In Zhifeng Bao, Renata Borovica-Gajic, Ruihong Qiu, Farhana Choudhury, and Zhengyi Yang, editors, *Databases Theory and Applications*, pages 227–240, Cham, 2024. Springer Nature Switzerland.

[SKS19] Abraham Silberschatz, Henry Korth, and S Sudarshan. *Database System Concepts.* McGraw-Hill, New York, NY, 7 edition, 2019.

[ZVBB11] Marcin Zukowski, BV VectorWise, Peter Boncz, and Henri Bal. Just-in-time compilation in vectorized query execution. 2011.