School of Computer Science and Engineering

Faculty of Engineering

The University of New South Wales

pgx-lower: Productionising Database Compiler Research

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

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 - Decision Support Benchmark H

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 4, and finally conclusions will be drawn in Chapter 6.

Chapter 2

Background

2.1 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.

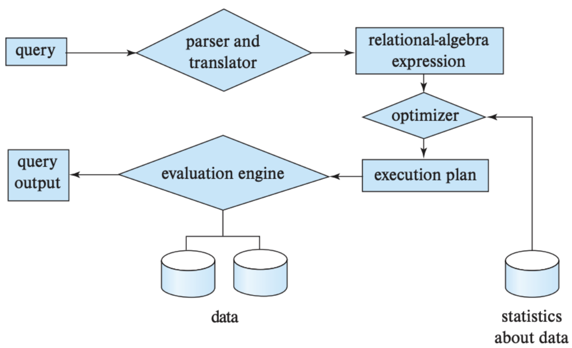

Figure 2.1: Database Structure
[SKS19]

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.

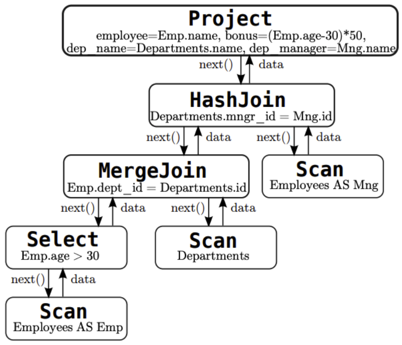

Figure 2.2: Volcano operator model tree.
[ZVBB11]

The fundamental issue with this classical model is that it is heavily under-utilising the hardware. If only a single tuple is pulled, our CPU caches are barely used. An i5-570, a popular CPU in 2010 had a 8MB L3 cache, but in 2024 an i5-14600K has a 24MB L3 cache [Pas25], [Tec25]. For disks, in 2010 the Segate Barracuda 7200.12 was popular, which had a sustained read of 138MB/s, but in 2022 the Samsung V-NAND SSD 990 PRO released with a sustained read of 7450MB/s. Increases this large mean the algorithms can fundamentally change.

This has lead to the vectorized execution model and the compiled model. With the vectorized model, multiple tuples are pulled up in a group rather than one at a time. A core advantage is that instructions per CPU cycle (IPC) can increase through single instruction, multiple data (SIMD) operations [KLK+18]. However, this can cause deep copy operations to be required, or more disk spillage than necessary. For instance, if a sort or a join allocates new space that is too much for the cache, the handling can become poor. The alternative approach, compilation, will be explored in section 2.2 and chapter 3.

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 constant or cost-free [SKS19]. This is partially due to when these systems were made, the disks were much slower and the caches were much smaller. 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.

Arguably most of the recent databasing research has been inside the optimiser which is visible in figure 2.1. This includes reordering join statements where the left and right sides are flipped, predicate pushdown where a conditional/filter on a node is moved onto a lower node, extracting common subexpressions to prevent recomputation, constant folding where constant operations are evaluated inside the optimiser rather than in runtime, and more [Ine21]. A list of optimisations would be long enough for a thesis to be written to summarise them.

Within traditional and volcano databases, the cache is managed through a set of buffer techniques while reading tuples. This works by the database reading a page, which usually has a fixed size and can be configured, such as 8KB. This is loaded into a buffer pool object which holds it inside of RAM. The operating system or environment manages where this memory goes inside the context of the L1/L2/L3 and ram caches depending on the access patterns [SKS19]. This buffer pool can change caching strategies depending on the context, such as last-recently-used, most-recently-used, and more, which can be decided in the optimiser [EH84].

Databases are commonly split into Online Transaction Processing (OLTP) and Online Analytical Processing (OLAP). OLTP has a focus on supporting atomicity, running multiple queries at once, and typically supports the work profile of an online service that does key-value lookups frequently. On the other hand, OLAP databases focus on analytical work profiles where an aggregation is requested, or some operation that spans a large chunk of the database [Kle19]. OLAP systems can be highly distributed, such as Apache Hive such that a large amount of compute can be used across the system [CRCG+19]. In the context of PostgreSQL, it is a hybrid architecture that has support for both these operations [HRTVVA21]. There is currently debate about whether this hybrid architecture is useful anymore because putting pressure on your database serving users commonly causes reliability issues [Moo24].

2.2 JIT Background

Just-in-time (JIT) compilers work with multiple layers of compilation such as raw interpretation of bytecode, unoptimized machine code, and optimized machine code. They 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]. This means the intial compilation can be faster, and the development cycle can go faster, as well as other benefits.

Due to branch-prediction optimization, JIT compilers can be 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]. 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].

It is difficult to evaluate what a good branch prediction value is, but a reasonable baseline is 1% is too high and should be optimised in a low latency environment [Far25]. This is not a formal definition, and is more based on tribal knowledge. Depending on the CPU, a branch mispredict can cost between 10 and 35 CPU cycles, with a safe interval being 14-25. Meaning, if there is 1 branch every 10 instruction, with a 5%

misprediction rate and a 20 cycle penalty per misprediction, 10% of the runtime will be spent fixing mispredictions [ESE06].

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 $llvm - jit$. A variety of research databases will be examined in chapter 3.

2.3 LLVM and MLIR

The LLVM Project is a compiler infrastructure that supports creating compilers so that common, but complex, compiler optimisations do not have to be re- implemented [LA04]. Multi-Level Intermediate Representation (MLIR) is another, newer toolkit that is tightly coupled with the LLVM project [LAB+20]. It adds a framework to define dialects, and lower through these dialects into machine code. 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.

LLVM defines a language-independent intermediate representation (IR) based on Static Single Assignment (SSA) form and MLIR is an extension of this [LA04]. The architecture follows a three-phase design: a front-end parses source code and generates LLVM IR, an optimizer applies a series of transformation passes to improve code quality, and a back-end generates machine code for the target architecture. MLIR extends this concept by introducing a flexible dialect system that enables progressive lowering through multiple levels of abstraction [LAB+20]. This addresses software fragmentation in the compiler ecosystem, where projects were creating incompatible high -level IRs in front of LLVM, and improves compilation for heterogeneous hardware by allowing target-specific optimisations at appropriate abstraction levels.

LLVM's On-Request-Compilation (ORC) JIT is a system for building just-in-time compilers with support for lazy compilation, concurrent compilation, and runtime optimization [LLV25]. ORC can compile code on-demand as it is needed, reducing startup time by deferring compilation of functions until they are first called. The JIT supports concurrent compilation across multiple threads and provides built-in dependency tracking to ensure code safety during parallel execution. This makes ORC particularly suitable for dynamic language implementations, REPLs (Read-Eval-Print Loops), and high-performance JIT compilers.

2.4 WebAssembly and others

2.5 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).

2.6 Database Benchmarking

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

Chapter 3

Related Work

This chapter 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.

This begins with PostgreSQL and their extension system in section 3.1, in section 3.2 system R will be explored as the classical example, followed by HyPer and Umbra in section 3.3 and section 3.4 which re-introduced the concept. Mutable in section 3.5 and LingoDB in section 3.6 are research databases. Lastly, PostgreSQL will be examined in section 3.1 as it uses expression-based compilation and there has been an attempt to create a compiled engine before.

3.1 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.

```
On 3/9/18 15:42, Peter Eisentraut wrote:
> The default of jit_above_cost = 500000 seems pretty good.  I constructed
> a query that cost about 450000 where the run time with and without JIT
> were about even.  This is obviously very limited testing, but it's a
> good start.

Actually, the default in your latest code is 100000, which per my
analysis would be too low.  Did you arrive at that setting based on testing?
```

Figure 3.1: Peter Eisentruat asking whether the defaults are too low. [Fre17]

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.

3.2 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.

3.3 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.

	Q1	Q2	Q3	Q4	Q5
HyPer + C++ [ms]	142	374	141	203	1416
compile time [ms]	1556	2367	1976	2214	2592
HyPer + LLVM	35	125	80	117	1105
compile time [ms]	16	41	30	16	34
VectorWise [ms]	98	-	257	436	1107
MonetDB [ms]	72	218	112	8168	12028
DB X [ms]	4221	6555	16410	3830	15212

Figure 3.2: HyPer OLAP performance compared to other engines.
[Neu11]

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.

	Q1		Q2		Q3		Q4		Q5	
	LLVM	MonetDB	LLVM	MonetDB	LLVM	MonetDB	LLVM	MonetDB	LLVM	MonetDB
branches	19,765,048	144,557,672	37,409,113	114,584,910	14,362,660	127,944,656	32,243,391	408,891,838	11,427,746	333,536,532
mispredicts	188,260	456,078	6,581,223	3,891,827	696,839	1,884,185	1,182,202	6,577,871	639	6,726,700
I1 misses	2,793	187,471	1,778	146,305	791	386,561	508	290,894	490	2,061,837
D1 misses	1,764,937	7,545,432	10,068,857	6,610,366	2,341,531	7,557,629	3,480,437	20,981,731	776,417	8,573,962
L2d misses	1,689,163	7,341,140	7,539,400	4,012,969	1,420,628	5,947,845	3,424,857	17,072,319	776,229	7,552,794
I refs	132 mil	1,184 mil	313 mil	760 mil	208 mil	944 mil	282 mil	3,140 mil	159 mil	2,089 mil

Figure 3.3: HyPer branching and cache locality benchmarks.
[Neu11]

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.

3.4 Umbra

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.

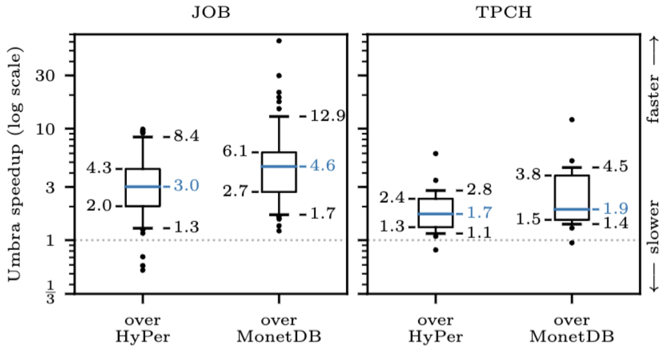

Figure 3.4: Umbra benchmarks.
[NF20]

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.

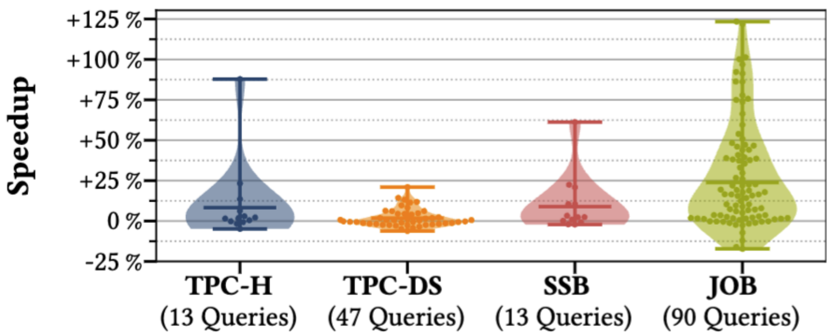

Figure 3.5: Umbra benchmarks after adaptive query processing (AQP).
[SFN22]

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.

3.5 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.

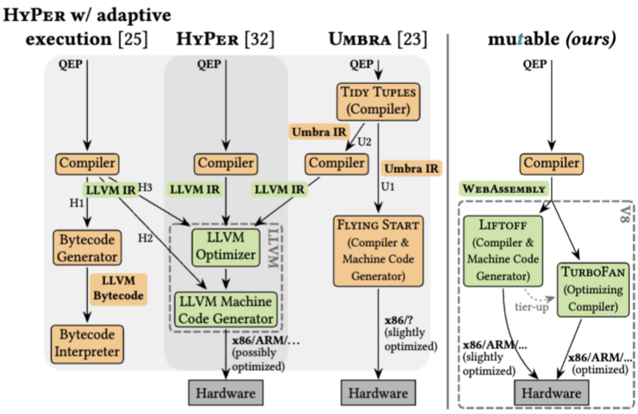

Figure 3.6: Comparison of mutable to HyPer and Umbra.
[HD23]

lation phase graph

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

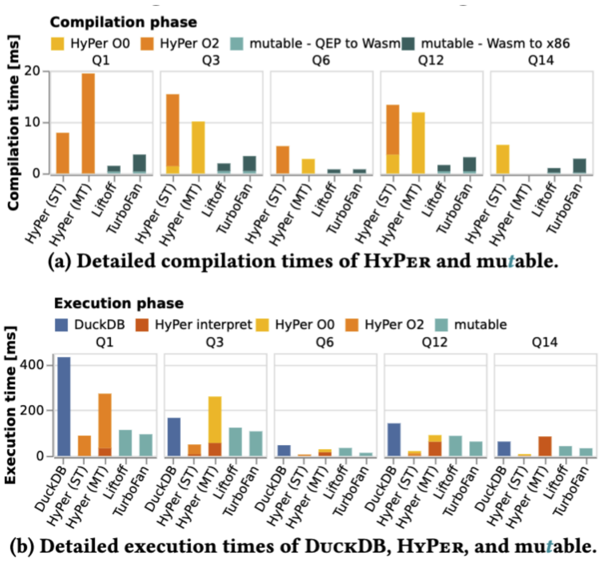

Figure 3.7: Benchmarks produced by Mutable.
[HD23]

3.6 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.

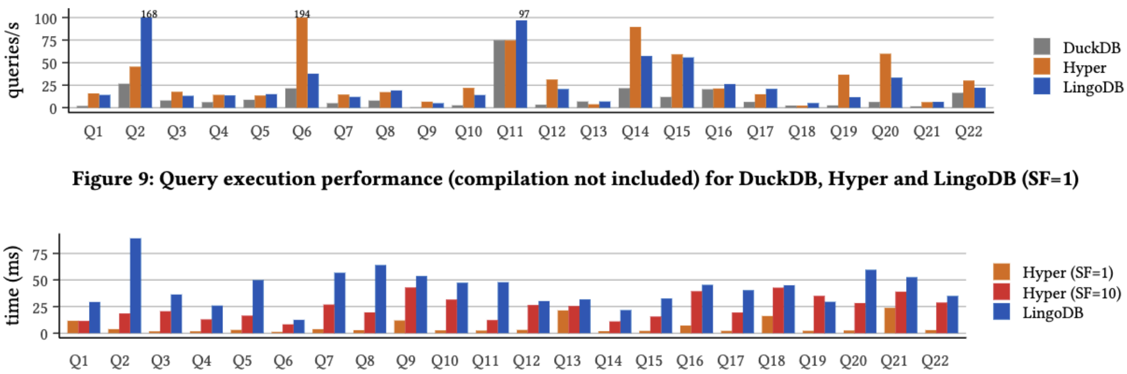

Figure 3.8: LingoDB benchmarking.
[JKG22]

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.

3.7 Benchmarking

These systems produced their own benchmarks and could selectively pick which systems to involve, so a recreation of the benchmarks was done. DuckDB, HyPer, Mutable, LingoDB and PostgreSQL were all compared to one another, and is visible in Figure 3.9. TPC-H was used as most of the involved pieces used it themselves [Tra24], and docker containers were chosen to make deploying it easier. These benchmarks were created by relying on the Mutable codebase as they had significant infrastructure to support this, and is visible at `https://github.com/zyros-dev/benchmarking-dockers`.

The benchmarks show that PostgreSQL is significantly slower than the rest, likely because it is an on-disk database and most of the others are in-memory. With PostgreSQL removed from the graph, HyPer and DuckDB are the fastest, and with a single core DuckDB is the slowest.

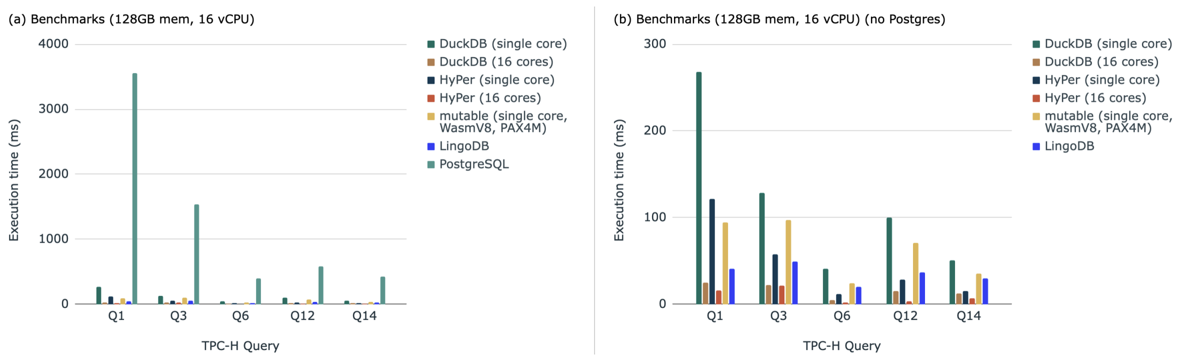

Figure 3.9: Benchmarking results.

To identify how much potential gain there is in a majory on-disk database, `perf` was used on PostgreSQL during TPC-H queries 1, 3, 6, 12 and 14 in figure 3.10. This shows that the CPU time varied from between 34.87% and 76.56%, with an average of 49.32%. These metrics were identified using the `prof` graph. With this much time in the CPU, it is clear that the queries can become several times faster if optimised.

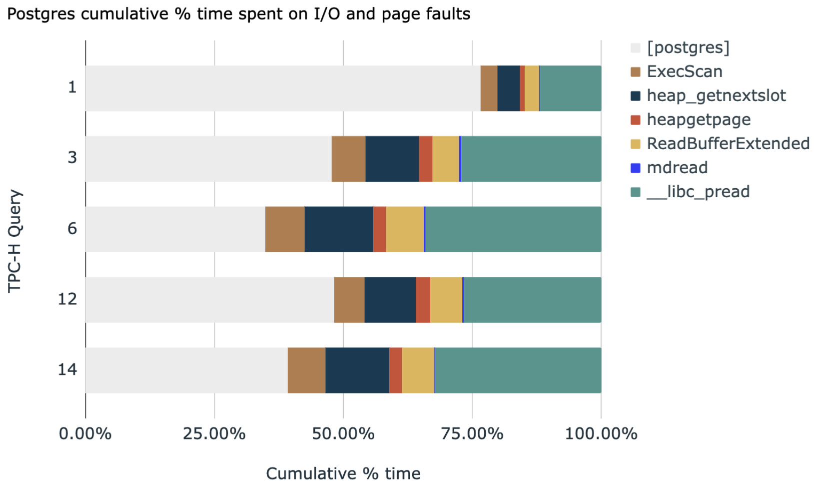

Figure 3.10: PostgreSQL time spent in the CPU, measured with prof.

As stated in subsection ??, one core acceleration of JIT is improving the branch predictions. To measure this, `perf` was used on the same set of queries as shown in Table 3.1. This shows that at worst, 43.02% of LLC loads are missed. A 0.69% branch miss rate has room for improvement, as systems with even a 99% branch prediction accuracy can have a significant bottleneck from it.

Table 3.1: Preliminary PostgreSQL query profiling statistics collected with `perf`.

Metric	**Q1**	**Q3**	**Q6**	**Q12**	**Q14**	**Unit**
			Absolute Values			
Task-clock	992.88	566.44	349.64	514.66	368.84	msec
Page faults	4230	3236	109	497	1928	occurrences
Cycles	2.72e9	1.57e9	9.76e8	1.42e9	1.03e9	occurrences
Instructions	6.49e9	2.63e9	1.61e9	2.44e9	1.69e9	occurrences
Branches	1.03e9	4.45e8	2.79e9	4.34e8	2.89e8	occurrences
Branch misses	**1.21e6**	**2.67e6**	**1.36e6**	**3.01e6**	**1.21e6**	occurrences
LLC loads	5.17e6	7.00e6	4.48e6	6.11e6	4.88e6	occurrences
LLC load misses	**1.76e6**	**2.84e6**	**1.93e6**	**2.37e6**	**2.07e6**	occurrences
			Normalized Values			
Task-clock	0.104	0.048	0.032	0.041	0.034	CPUs
Page faults	4260	5713	311	966	5227	/sec
Cycles	2.743	2.777	2.79	2.768	2.789	GHz
Instructions	2.38	1.67	1.65	1.71	1.64	insn/cycle
Branches	1040	786	797	843	785	M/sec
Branch misses	**0.12%**	**0.60%**	**0.49%**	**0.69%**	**0.42%**	of branches
LLC loads	5.203	12.354	12.807	11.868	13.22	M/sec
LLC load misses	**34.12%**	**40.65%**	**43.02%**	**38.87%**	**42.37%**	of LL-cache

3.8 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.

3.9 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. Accomplishing this shows there is a way to rely on previous ACID-compliance and supporting code infrastructure.

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 4

Method

In section 4.1 the overarching design is described, then section 4.2 goes over the implementation.

4.1 Design

The first decision is which database and which compiler this project should use. Something with strong extension support, wide-spread usage, high performance, and a volcano execution model is needed as a base. For the compiler, it would be ideal if they already use a similar interface to the target database when they parse SQL, and have promising results in their performance. This removes HyPer, Umbra, and System R, and leaves Mutable and LingoDB. LingoDB parses its inputs with `pg_query`, so it matches with PostgreSQL.

As a result, PostgreSQL and LingoDB were chosen. PostgreSQL offers strong support for extensions, and it is possible to override its execution engine using runtime hooks. An example of this Tiger data, which was explored in section [?]. The primary challenge with this is that LingoDB is a columnar, in-memory database, so adjustments will be needed. Furthermore, LingoDB does not support indexes, which can make the benchmarks against PostgreSQL unfair. Another detail is that LingoDB's newer versions contain a large number of features and optimisations that is not relevant to us, so to simplify implementation effort the 2022 version was used from their initial paper.

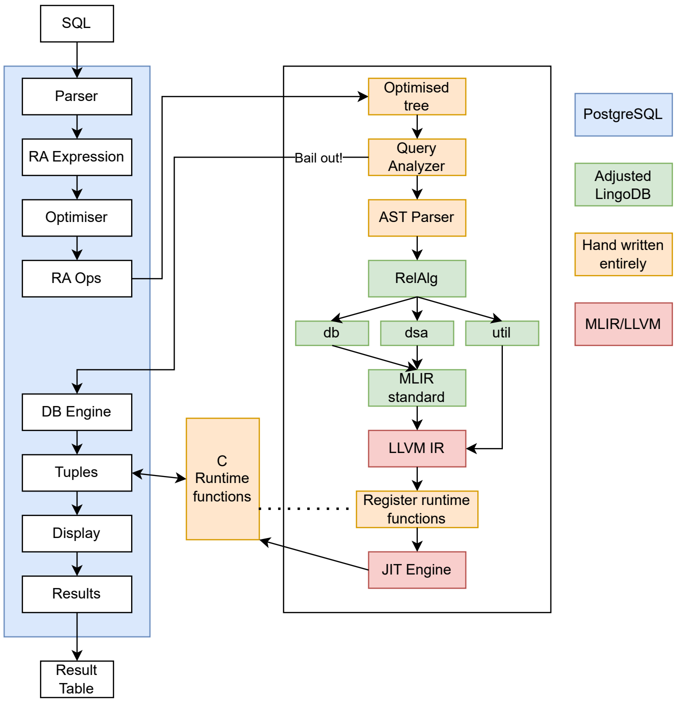

Figure 4.1: System design with labels of component sources.

LingoDB was integrated into PostgreSQL as seen in figure 4.1. The blue represents PostgreSQL components, with the pipeline on the left being the whole of PostgreSQL. A query reaches the runtime hooks, which gets analyzed by a hand-written analyzer for whether the query can be execution, and then parsed. These hand writte components are annotated in light-peach. This goes through the code LingoDB created, but with custom runtime hooks and other small edits, annotated with green. Finally this is compiled into LLVM IR, which has the runtime hooks for reading from PostgreSQL embedded inside.

In the case that a query fails, the system should still support returning the results and gracefully roll over to PostgreSQL. This was done by ensuring the AST parser entrance has a try-catch pattern that routes back to PostgreSQL even in failures. However, this does not protect from system panics such as segmentation faults.

The most time-consuming part of this is expected to be the AST Parser section, because it will be receiving the plan tree with the optimisations from PostgreSQL. LingoDB was designed to parse the query tree, which would come from the "Parser" stage in figure 4.1. 18 plan nodes and 14 expression nodes were implemented.

The final goal here is to support the TPC-H query set. To drive this implementation, a test-driven approach was used where PostgreSQL's `pg_regress` module added support for creating SQL queries and defining expected outputs. With this, a test set of basic queries was created which built up to TPC-H queries. This allowed progressive node implementation during development, and a quick way to validate changes are safe.

Node implementation ordering followed the dependency analysis. Foundational nodes such as the sequential scan and projection are in virtually every query, while other nodes build on top. By implementing in the dependency order, each new node could be tested using the previously implemented nodes, and bugs can be isolated.

4.2 Implementation

The primary system this project was developed on was a x86_64 CPU (Ryzen 3600) and on Ubuntu 25.04. The database was not tested on MacOS or Windows, and this may lead to issues when installing it independently.

4.2.1 Integrating LingoDB to PostgreSQL

The project was started from `https://github.com/mkindahl/pg_extension`, then `ExecutorRun_hook` inside of executor.h in PostgreSQL was used `https://doxygen.postgresql.org/executor_8h_source.html` as the entrance. Within PostgreSQL there are some surrounding steps since the intention is not usually to replace the entire executor with these hooks, so the memory context had to be activated and switched into.

Next, the `QueryDesc` pointer, which contains the query request, was to be passed through to `C++`. This causes a design decision. Good practice here is to use smart pointers to prevent memory leaks, but this object is large and the source of truth about the request. Furthermore, the memory is handled by the PostgreSQL memory contexts. It was decided that these objects will remain as raw pointers, causing the `C++` to break conventions.

LingoDB was installed as a git submodule and set to a read-only permission. This was maintained for reference purposes only, and the compilation phases would be extracted. LingoDB used LLVM 14, and was upgraded to LLVM 20 to modernise it and slightly better support with —C++20— (some workarounds were required with LLVM 14 that could be skipped with LLVM 20). However, since this is the C++ API for LLVM, a large amount of the LingoDB code had to be adjusted to compile.

4.2.2 Logging infrastructure

PostgreSQL has its own logging infrastructure that routes through its —elog— command, but it was decided that a two-layer logging infrastructure was required. The first layer is the level, (—DEBUG—, —IR—, —TRACE—, —WARNING_LEVEL—, —ERROR_LEVEL—, and more), and the second represents which layer of the design the log is inside of (—AST_TRANSLATE—, —RELALG_LOWER—, —DB_LOWER—, and more). This meant if the AST translation was being worked on, all the logs in only that section of the codebase could be enabled. The core benefit of this is that the logs are lengthy so it becomes easier to navigate.

An issue that was encountered was that the LLVM/MLIR logs would route through stderr, and this caused difficult to debug issues until the hook was found to redirect this into —elog— as well. Subsection˜refsubsec:debugging will explore one of the workarounds that was needed at this stage.

Lastly, for error handling mostly —std::runtime_error— was utilised. This served as a global way to log the stack trace and roll back to PostgreSQL's execution. There initially was an implementation of error handling with severity levels and messages, but the simplicity of a single command that rolled back to PostgreSQL was more generally useful.

4.2.3 Debugging Support

An important property of PostgreSQL is that each client connection creates a new process. This means there are several layers to claw through to debug issues. First, is the PostgreSQL postmaster, then the client connection, then within that is the runtime hook entrance, which leads to $C++$, and inside $C++$ we will be compiling into a JIT runtime, and the bugs can happen inside there. This poses a challenge for how to debug problems such as segmentation faults and errors without any logging.

This was solved with a combination of the regression tests, unit testing, and a script to connect —gdb— to dump the stack. The regression tests were already explored, but the unit tests consist of testing anything unconnected to PostgreSQL. The issue is that this extension creates a —pgx_lower.so— which is installed within PostgreSQL, then the PostgreSQL libraries are used from inside there. This means if we run without being inside of PostgreSQL, no psql libaries can be used. As a result, unit tests can only test MLIR functions. Most of the unit tests were highly situational, and are used when a proper interactive GDB connection was required within the IDE. Furthermore, unit tests allow the —stderr— to be visible, which assists greatly with MLIR/LLVM errors that go to —stderr— and nowhere else.

For the stack-dumping, a script was written, —debug-query.sh— which proved to be the most useful approach for complex issues. It has the ability to create a psql connection, git the process ID of the client connection, then connect GDB, run a desired query, and dump the stack trace. In this way, the majority of errors were tackled.

4.2.4 Data Types

PostgreSQL has a large set of data types (`https://www.postgresql.org/docs/current/datatype.html`), and LingoDB has significantly less. However, for TPC-H we only require a subset of these. Table 4.1 shows which of the LingoDB types are used, and table 4.2 shows the type mappings. The two primary workarounds that were implemented was for decimals and the various types of strings. For decimals, i128 is enough precision for most of the TPC-H tests, and is what LingoDB was using. However, adjustments had to be made to ensure impossible to allocate values do not appear, so the precision was capped at —¡32, 6¿— That is, 32 digits in the integer part, 6 digits in the decimal places.

For the date types, a comprimise was made that when it receives an interval type with a months column, it will turn this into days and introduce errors. However, since the TPC-H queries never use month intervals, this is acceptable.

DB Dialect Type	**LLVM Type**	**Used by pgx-lower?**
`!db.date<day>`	`i64`	Yes
`!db.date<millisecond>`	`i64`	No
`!db.timestamp<second>`	`i64`	Only if typmod specifies
`!db.timestamp<millisecond>`	`i64`	Only if typmod specifies
`!db.timestamp<microsecond>`	`i64`	Yes (default)
`!db.timestamp<nanosecond>`	`i64`	Only if typmod specifies
`!db.interval<months>`	`i64`	No
`!db.interval<daytime>`	`i64`	Yes
`!db.char<N>`	`{ptr, i32}`	No (uses !db.string)
`!db.string`	`{ptr, i32}`	Yes
`!db.decimal<p,s>`	`i128`	Yes
`!db.nullable<T>`	`{T, i1}`	Yes

Table 4.1: LingoDB type system full capabilities

PostgreSQL Type	**DB Dialect Type**	**LLVM Type**
Integers		
INT2 (SMALLINT)	i16	i16
INT4 (INTEGER)	i32	i32
INT8 (BIGINT)	i64	i64
Floating Point		
FLOAT4 (REAL)	f32	f32
FLOAT8 (DOUBLE)	f64	f64
Boolean		
BOOL	i1	i1
String Types		
TEXT / VARCHAR / BPCHAR	!db.string	{ptr, i32}
BYTEA	!db.string	{ptr, i32}
Numeric		
NUMERIC(p,s)	!db.decimal<p,s>	i128
Date/Time		
DATE	!db.date<day>	i64
TIMESTAMP	!db.timestamp<s|ms|μs|ns>	i64
INTERVAL	!db.interval<daytime>	i64
Nullable		
Any nullable column	!db.nullable<T>	{T, i1}

Table 4.2: PostgreSQL type translation through DB dialect to LLVM

This defines most of the supporting details, and the main two components of the implementation can be described: the runtime patterns and the plan tree translation.

4.2.5 Runtime patterns

Runtime functions are used in LingoDB for difficult to implement methods in LLVM, such as a sort algorithm. —pgx-lower— implemented reading tuples from PostgreSQL, storing them as a result so that they can be streamed one by one, adjusted several runtime implementations from LingoDB, and changed the sort and hashtable implementations to rely on the PostgreSQL API rather than standard collections.

Figure 4.2 shows the high-level components in a runtime function. During SQL translation to MLIR, the frontend creates `db.runtimecall` operations with a function name and arguments. These operations are registered in the runtime function registry, which maps each function name to either a `FunctionSpec` containing the mangled C++ symbol name, or a custom lowering lambda. During the DBToStd lowering pass, the `RuntimeCallLowering` pattern looks up each runtime call in the registry and replaces it with a `func.call` operation targeting the mangled C++ function. The JIT engine then links these function calls to the actual compiled C++ runtime implementations, which handle PostgreSQL-specific operations like tuple access, sorting via `tuplesort`, and hash table management using PostgreSQL's memory contexts. This pattern allows complex operations to be implemented once in C++ and reused across all queries, while maintaining type safety and null handling semantics through the MLIR type system.

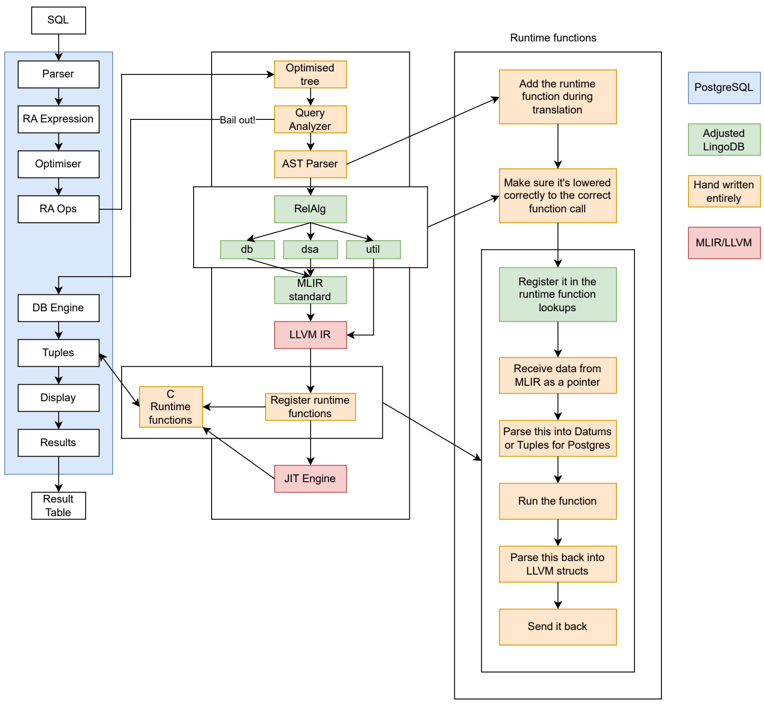

Figure 4.2: System design with labels of component sources.

The PostgreSQL runtime implements zero-copy tuple access for reading and result accumulation for output. When scanning a table, `openpostgrestable()` creates a heap scan using `heapbeginscan()`, and `readnexttuplefromtable()` stores a pointer (not a copy) to each tuple in the global `gcurrenttuplepassthrough` structure. JIT code extracts fields via `extractfield()`, which uses `heapgetattr()` and converts PostgreSQL `Datum` values to native types. For results, `tablebuilderadd()` accumulates computed values as `Datum` arrays in `ComputedResultStorage`. When a result tuple completes, `addtupletoresult()` streams it back through PostgreSQL's `TupleStreamer` by populating a `TupleTableSlot` and calling the destination receiver, enabling direct integration with PostgreSQL's tuple pipeline.

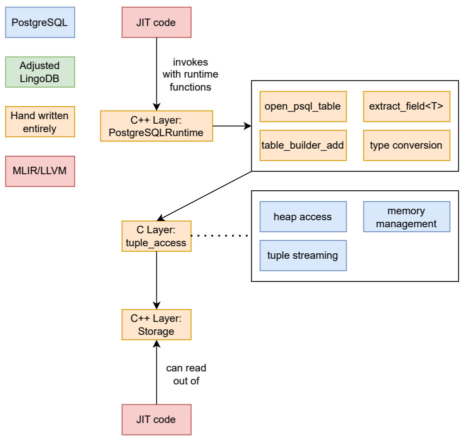

Figure 4.3: PostgreSQLRuntime.h component design

The PostgreSQL runtime allows the JIT runtime to read from the psql tables, and the design of it is visible in Figure 4.3. Generated JIT code invokes runtime functions implemented in the C++ layer, including table operations (`open_psql_table`), field extraction (`extract_field<T>`), result building (`table_builder_add`), and type conversions between PostgreSQL's `Datum` representation and native types. These runtime functions interface with PostgreSQL's C API layer, which handles heap access for reading tuples, memory management through PostgreSQL's context system, and tuple streaming for returning results to the executor. An important part is that when tuples are read from Postgres, only the pointers are stored within the C++ storage layer to maintain zero-copy semantics.

Once stored, the JIT code can read from the batch and stream tuples back through the output pipeline as well. Streaming the tuples out from JIT means that the entire table does not build up in RAM, and instead tuples are returned one by one. This was tested by doing larger table scans as avoiding this buildup is essential.

LingoDB's sort and hashtable runtimes were relying on `std::sort` and `std::unordered_map` respectively. This is problematic because as an on-disk database we need to handle disk spillage in these scenarios. Rather than reinventing these, leaning on psql's implementation of these solves these issues and creates a blueprint for further implementations.

Most of the LingoDB lowerings bake metadata (such as table names) into the compiled binary by JSON-encoding it as a string. Instead of that, for the sort and hash table runtimes a specification pointer was used. Inside the plan translation stage, a struct was built and allocated with the transaction memory context, then the pointer to this was baked into the compiled binary instead. This enabled these runtimes to trigger without doing JSON deserialisation, and creating the operations them could skip this stage. This is something that a regular compiler would be incapable of doing, because the binary needs to be a standalone program, but in this context it can be relied upon.

4.2.6 Plan Tree Translation

The plan tree translation converts PostgreSQL's execution plan nodes into RelAlg MLIR operations. Figure 4.4 shows where this fits into the broader design. Within the AST Parser component, we have an entry that reads the PostgreSQL tag on the node that reads which type of plan it is, then a recursive descent parser starts translating. Within each translation function, the pattern is typically that the children of the plan is translated (i.e. post-order traversal), then the definition of the node is established in our context, the translation into the MLIR relational algebra dialect is done, and a "translation result" is returned.

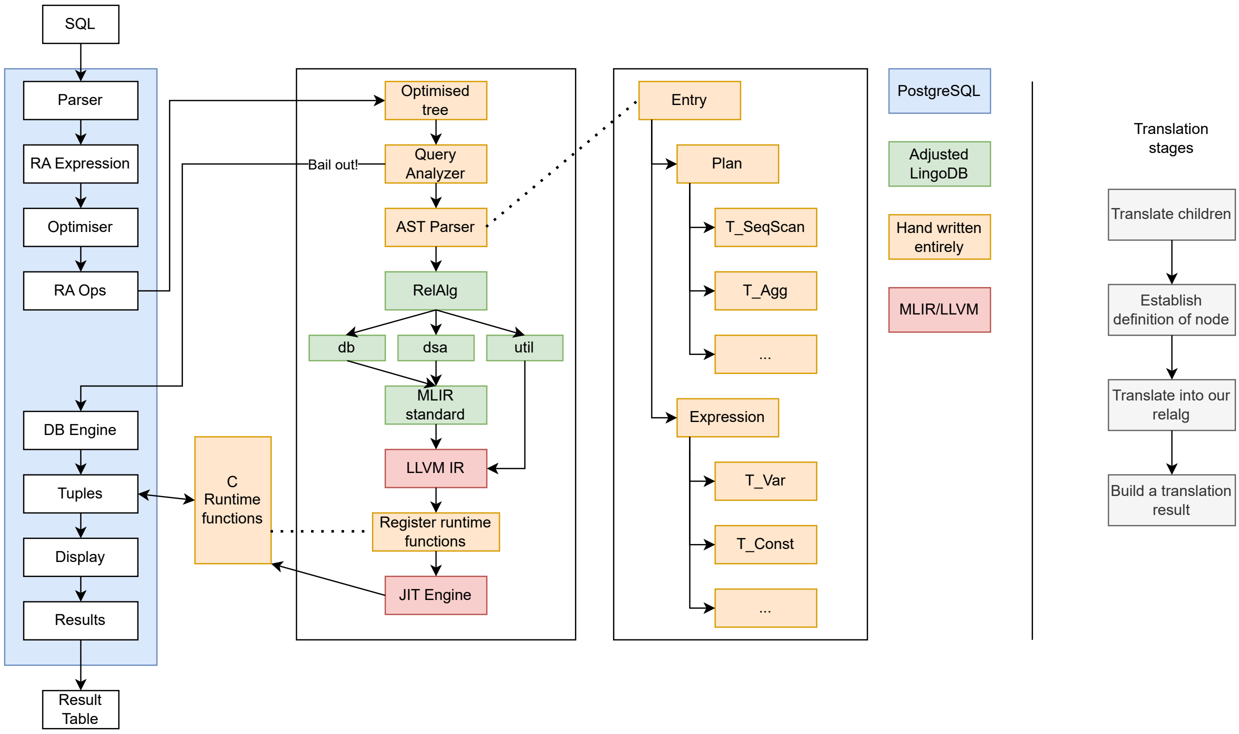

Figure 4.4: AST translation design and high-level steps

The translation functions follow a consistent pattern, as shown in Listing 4.1. Each function takes the query context and a PostgreSQL plan node pointer, performs the translation, and returns a `TranslationResult`. The concept here is that the `QueryCtxT` object is pushed down, and when it is mutated a new one is allocated and pushed to the child, while the `TranslationResults` flow upwards and represent the output of each node. This, in theory, grants strong type-correctness. However, it is not strictly followed.

Listing 4.1: Plan node translation method signatures. The expression nodes follow the same pattern.

```
auto translate_plan_node(QueryCtxT& ctx, Plan* plan) -> TranslationResult;
auto translate_seq_scan(QueryCtxT& ctx, SeqScan* seqScan) -> TranslationResult;
auto translate_index_scan(QueryCtxT& ctx, IndexScan* indexScan) -> TranslationResult;
auto translate_index_only_scan(QueryCtxT& ctx, IndexOnlyScan* indexOnlyScan) -> TranslationResult;
auto translate_bitmap_heap_scan(QueryCtxT& ctx, BitmapHeapScan* bitmapScan) -> TranslationResult;
auto translate_agg(QueryCtxT& ctx, const Agg* agg) -> TranslationResult;
auto translate_sort(QueryCtxT& ctx, const Sort* sort) -> TranslationResult;
auto translate_limit(QueryCtxT& ctx, const Limit* limit) -> TranslationResult;
auto translate_gather(QueryCtxT& ctx, const Gather* gather) -> TranslationResult;
```

```
auto translate_gather_merge(QueryCtxT& ctx, const GatherMerge* gatherMerge) -> TranslationResult;
auto translate_merge_join(QueryCtxT& ctx, MergeJoin* mergeJoin) -> TranslationResult;
auto translate_hash_join(QueryCtxT& ctx, HashJoin* hashJoin) -> TranslationResult;
auto translate_hash(QueryCtxT& ctx, const Hash* hash) -> TranslationResult;
auto translate_nest_loop(QueryCtxT& ctx, NestLoop* nestLoop) -> TranslationResult;
auto translate_material(QueryCtxT& ctx, const Material* material) -> TranslationResult;
auto translate_memoize(QueryCtxT& ctx, const Memoize* memoize) -> TranslationResult;
auto translate_subquery_scan(QueryCtxT& ctx, SubqueryScan* subqueryScan) -> TranslationResult;
auto translate_cte_scan(QueryCtxT& ctx, const CteScan* cteScan) -> TranslationResult;
```

The 14 expression node types are documented in Table 4.3, and the 18 plan node types in Table 4.4. These will be explained more specifically in the subsections.

File	Node Tag	Implementation Note
basic	`T_BoolExpr`	Boolean AND/OR/NOT - with short-circuit evaluation
basic	`T_Const`	Constant value - converts Datum to MLIR constant
basic	`T_CoalesceExpr`	COALESCE(...) - first non-null using if-else
basic	`T_CoerceViaIO`	Type coercion - calls PostgreSQL cast functions
basic	`T_NullTest`	IS NULL checks - generates nullable type tests
basic	`T_Param`	Query parameter - looks up from context
basic	`T_RelabelType`	Type relabeling - transparent wrapper
basic	`T_Var`	Column reference - resolves varattno to column
complex	`T_Aggref`	Aggregate functions - creates AggregationOp
complex	`T_CaseExpr`	CASE WHEN ... END - nested if-else operations
complex	`T_ScalarArrayOpExpr`	IN/ANY/ALL with arrays - loops over elements
complex	`T_SubPlan`	Subquery expression - materializes and uses result
functions	`T_FuncExpr`	Function calls - maps PostgreSQL functions to MLIR
operators	`T_OpExpr`	Binary/unary operators

Table 4.3: Expression node translations

File	Node Tag	Implementation Note
agg	`T_Agg`	Aggregation - AggregationOp with grouping keys
joins	`T_HashJoin`	Hash join - InnerJoinOp with hash implementation
joins	`T_MergeJoin`	Merge join - InnerJoinOp with merge semantics
joins	`T_NestLoop`	Nested loop join - CrossProductOp or InnerJoinOp
scans	`T_BitmapHeapScan`	Bitmap heap scan - SeqScan with quals
scans	`T_CteScan`	CTE scan - looks up CTE and creates BaseTableOp
scans	`T_IndexOnlyScan`	Index-only scan - treated as SeqScan
scans	`T_IndexScan`	Index scan - treated as SeqScan
scans	`T_SeqScan`	Sequential scan - BaseTableOp with optional Selection
scans	`T_SubqueryScan`	Subquery scan - recursively translates subquery
utils	`T_Gather`	Gather workers - pass-through (no parallelism)
utils	`T_GatherMerge`	Gather merge - pass-through (no parallelism)
utils	`T_Hash`	Hash node - pass-through to child
utils	`T_IncrementalSort`	Incremental sort - delegates to Sort
utils	`T_Limit`	Limit/offset - LimitOp with count and offset
utils	`T_Material`	Materialize - pass-through (no explicit op)
utils	`T_Memoize`	Memoize - pass-through to child
utils	`T_Sort`	Sort operation - SortOp with sort keys

Table 4.4: Plan node translations

Some common definitions of nodes will also serve to be useful. Nodes commonly have a `InitPlan` parameter, which is a function that should be called before the node is used and contains initiailising variables such as the parameters, catalogue lookups, or other things. `targetlist` contains the output of the node, `qual` is the qualification of the node, which means what should be filtered as outputs. Join nodes will have a left and right tree, with a more intuitive name of inner/outer children. These signify the inner and outer loops of the nested for-loop that is created.

Expression Translation - Variables, Constants, Parameters

asdf The two relevant ways PostgreSQL identifies values is with variable nodes and parameter nodes. These are stored inside a schema/column manager class as well as the `QueryCtxT` within the code. Variables are typically defined within scans, while the parameters are intermediate products. A number of difficulties were encountered with these as there are a number of interacting variables used to identify them (`varno`, `varattno`, and "special" values such as index joins). As a result, a generic function was built into the QueryCtxT object to handle this lookup logic, `resolve_var`. These will be used constantly.

Parameters are mostly defined within the InitPlan, and one key novel type is the cached scalar type.

Plan translation - Scans

PostgreSQL has a variety of scans that read from tables, and handlers for sequential scan, subquery scans, index scans, index-only scans, bitmap heap scans, and CTE (common table expression) scans were implemented. However, aside from the subquery scan and CTE scan, they all map to sequential scan. This is a tradeoff to reduce complexity, and the most impacted by this is the index scan.

Within the index scan it has specific annotations for variables with its —INDEX_VAR— which requires special resolution mapping to for us to handle. Furthermore, we need to handle the qualifiers (scan filters) `indexqual`, `recheckqual` like a `qual`. In psql, these filter at different stages, but since we are skipping the index implementation they become generic filters instead.

CTE scan plans are defined within the InitPlan of nodes, but still route through the primary plan switch statement logic. Neither CTE plans or subqueries currently offer de-duplication to simplify implementation. That is, if a query uses the same CTE reference or writes the same subquery twice, they will currently be lowered into two different LLVM chunks of code rather than congregated and referenced.

Plan translation - Aggregations

Aggregation is a complicated node type. It consists of an aggregation strategy, which is ignored as we have a simple algorithm instead, splitting specification, which is also unutilised, group columns, number of groups, it can produce parameters, and it has its own operators such as COUNT, SUM, and so on. Furthermore, it uses special varnos to do lookups for variables (-2), so it requires a new context object, and supports DISTINCT statements.

Most of the pain was with specific edge cases that arise in the simplification. For instance, `COUNT(*)` behaves differently in combining mode where parallel workers provide partial counts rather than raw rows, requiring translation to SUM instead of CountRowsOp. Similarly, `HAVING` clauses can reference aggregates not present in the SELECT list, necessitating a discovery pass with `find_all_aggrefs()` to ensure all required aggregates are computed before filtering. The use of magic number varno=-2 to identify aggregate references, while necessary to distinguish them from regular column references, breaks the normal variable resolution flow and requires special handling throughout the expression translator.

Plan translation - Joins

For joins, there are two layers to translating them: the type of join, and the algorithm used by the join. The type of join refers to inner, semi, anti, right-anti, left/right joins, and full joins. The semi and anti join types are not specifically translated, and instead rely on an `EXISTS`/ `NOT EXISTS` translations instead because they are semantically the same operation.

The algorithm used by the join refers to merge, nestloop, or hash joins. Following LingoDB's pattern, the merge joins are turned into hash joins so that there does not have to be additional lowering code. A challenge was that nest loops can carry parameters, so a new query context has to be created, the parameter has to be registered and inserted into the lookups.

One issue that is still inside the joins implementations is how preventing double computations works. For this, LingoDB takes the inner join and does it on its own and builds a vector of results, then afterwards it iterates over the outer operation and uses the inner section in a pre-computed way. This prevents duplicated computation at the cost of using more memory. In theory this is fine, but the vector still needs to implement disk spillage. In practice, this did not cause enough memory issues to warrant implementation.

Expression Translation - Nullability

Postgres has nullability defined in the plan tree that is passed to pgx-lower, but LingoDB appeared to have operations inside the lowerings where previously non-null objects could become nullable (outer joins, aggregations, unions, predicate evaluation can all cause this). Since nullability affects every object and functions in the same way as not-a-number (anything it touches becomes NaN/nullable), this introduce implementation pains.

One note to be aware of is that LingoDB and PostgreSQL have inverted null flags. That is, in LingoDB 1 means valid, and in PostgreSQL 1 means null. This causes confusion with the runtime functions needing to invert flags back and forth.

Expression Translation - Operators and OID strings

Within operators, the primary challenge is the type conversions and quirks. Comparing two BPCHARs requires adding padding for the space around them, and to implicitly upcast operations a class was extracted from LingoDB's DB dialect (`SQLTypeInference`). Furthermore, rather than relying on PostgreSQL's OID system for finding which operation to use, they were converted to strings ("¿", "¡", and so on) then these were used. This prevents issues with precision specifications in the OID leading to unidentified operations, and a similar approach was used in function nodes, aggregation functions, sort operations, and scalar maths.

Translation - Others

A large number of these nodes are pass through nodes or delegated to another, sibling node, such as `T_Hash`, `T_Material`, `T_Memoize`, and `T_RelabelType`. Furthermore, nodes also come with executor hints and cost metrics which were skipped over rather than dragged through LingoDB, as the optimisations were already done by Postgres. IN/ANY operations are also converted into EXISTS operations, a number of operations such as scalar subqueries are always marked as nullable, and CastOps are also made frequently to defer casting to later layers.

4.2.7 Configuring JIT compilation settings

Not much tinkering was done with the JIT optimisation flags, the minimum optimisation passes were used so that it can compile end-to-end, and `llvm::CodeGenOptLevel::Default` was used as the optimisation level. These optimisation passes consist of SROA, InstCombinePass, PromotePass, LICM pass, reassociation pass, GVN pass, and simplify GVN pass. The general consensus appeared to be that `-O2` should be used on it and moved on. This means it is possible to do more tuning work on this.

4.2.8 Profiling Support

Code infrastructure was written to support magic-trace for profiling and isolating issues, and a physical computer with an Intel CPU was set up with an i5-6500T, 16GB of RAM and a Samsung MZVLB256HAHQ-000L7 NVMe disk. This was particularly useful for isolating obvious bottlenecks within the system and understanding the latency when compared to PostgreSQL. Figure 4.5 represents the flame chart for query 3, and has a runtime of approximately 260 milliseconds. The functions that it calls are clear, and you can see how the query runs over time.

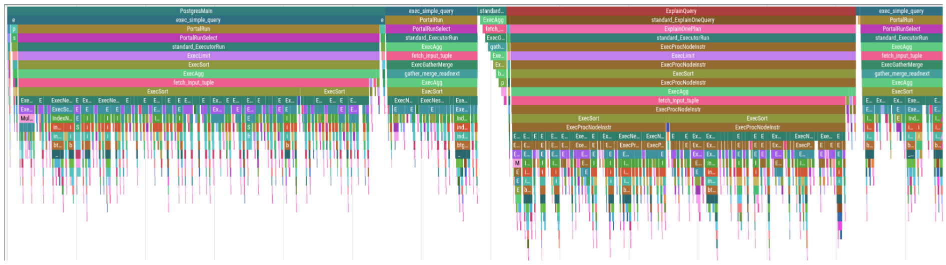

Figure 4.5: PostgreSQL's magic-trace flame chart for TPC-H query 3 at scale factor 0.05 (approximately 5 megabytes of data)

The flame chart before any optimisations were applied is visible in figure 4.6. In that chart it is visible that too much time is spent inside the LLVM execution (those spikes in the last 2/3rds are table reads). After adjusting how tuples are read, ensuring joins go to the correct algorithm, introducing Postgres's tuple-slot reading API, and disabling logs, the chart looks like figure 4.7. These adjustments improved the latency from 4.5 seconds to approximately 400 milliseconds.

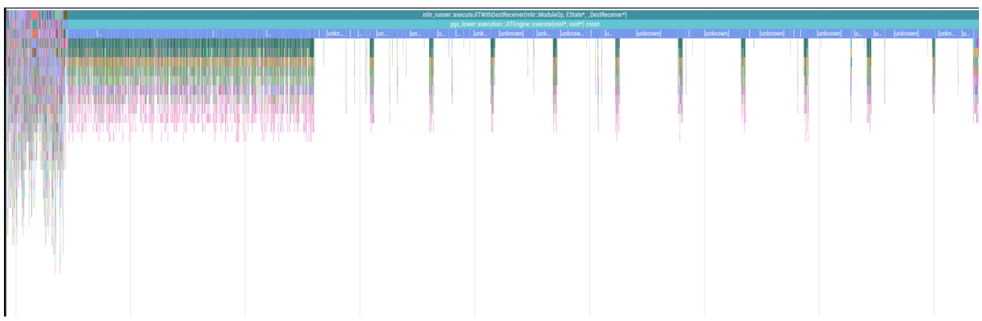

Figure 4.6: pgx-lower's magic-trace flame chart for TPC-H query 3 at scale factor 0.05 before optimisation

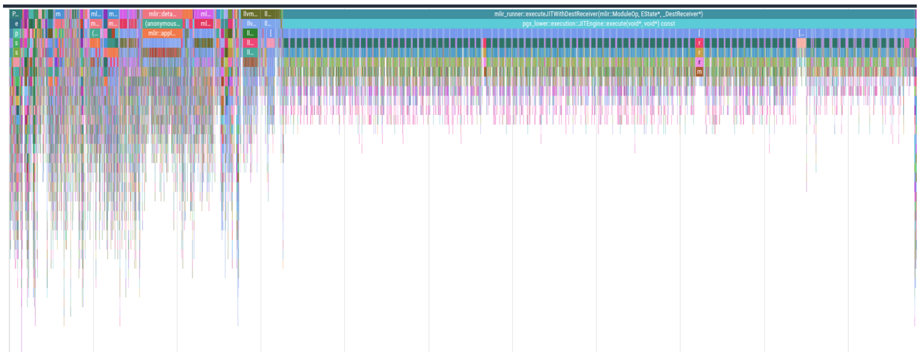

Figure 4.7: pgx-lower's magic-trace flame chart for TPC-H query 3 at scale factor 0.05 after optimisation

The specifics of running these in a stable way will be explained in subsection 4.2.10.

4.2.9 Website

A small website was prepared so that users can interact with the lowerings and the compiler without installing the system themselves at `https://pgx.zyros.dev/query`. Keep in mind that it relies on caching the results, it has a scale factor of 0.01 (10 megabytes of data), and the pgx-lower system there is (as of writing), running a debug build which has significantly longer runtimes. The implementation for this can be is at `https://github.com/zyros-dev/pgx-lower-addons`. It is implemented with python for the backend server. SQLite for the query caching, react for the frontend, has docker containers to support the reverse proxy with Nginx, and a private Grafana health dashboard.

4.2.10 Benchmarking and Validation

A challenge is that PostgreSQL contains a non-deterministic optimiser, and many small factors can affect runs. For this reason, a python script was created that reads from a yaml file, and does a benchmark run. This means we can specify runs beforehand, and run them robustly over a long period. Also, this benchmarking run computes a hash of the outputs between PostgreSQL and pgx-lower to validate the outputs are correct between all the runs, and the hashes were compared. This avoids storing large amounts of data over time, while the issue can still be rediscovered in a large batch of runs.

The benchmark configurations used are displayed in Listing 4.2. These configurations allow testing across different scale factors, with and without indexes, and with varying iteration counts to understand performance characteristics. With multiple iterations, graphs that contain distributions can be created. These were decided by bucketing queries into small scale factor (0.01, or 1 megabyte of data) to show the overhead cost of the JIT compiler, medium scale factor (0.16) to show how Postgres scales while still keeping all the queries enabled with indexes, and lastly scale factor 1 with the very time-consuming queries completely disabled. These disabled queries would take on the order of hours in PostgreSQL, so benchmarking them was too time consuming.

Listing 4.2: Benchmark configurations for TPC-H testing

```yaml
full:
  runs:
    - container: benchmark
      scale_factor: 0.01
      iterations: 5
      profile: false
      indexes: false
      skipped_queries: ""
      label: "SF=0.01, indexes disabled, 5 iterations"

    - container: benchmark
      scale_factor: 0.01
      iterations: 100
      profile: false
      indexes: false
      skipped_queries: "q07,q20"
      label: "SF=0.01, indexes disabled - excluding postgres {q07,q20}, 100 iterations"

    - container: benchmark
      scale_factor: 0.01
      iterations: 100
      profile: false
      indexes: true
      skipped_queries: ""
      label: "SF=0.01, indexes enabled, 100 iterations"

    - container: benchmark
      scale_factor: 0.16
      iterations: 5
      profile: false
      indexes: true
      skipped_queries: ""
      label: "SF=0.16, indexes enabled, 5 iterations"

    - container: benchmark
```

```yaml
    scale_factor: 0.16
    iterations: 100
    profile: false
    indexes: true
    skipped_queries: "q17,q20"
    label: "SF=0.16, indexes enabled, excluding {q17,q20}, 100 iterations"

  - container: benchmark
    scale_factor: 1
    iterations: 100
    profile: false
    indexes: false
    skipped_queries: "q02,q17,q20,q21"
    label: "SF=1, indexes disabled, excluding {q02,q17,q20,q21}, 100 iterations"
```

One thing to note here is that it was decided that only PostgreSQL and pgx-lower would be compared, rather than all the databases mentioned in chapter 3. As section 3.7 showed that the impact of PostgreSQL's architecture being on disk makes it significantly slower than any of the other databases.

The magic trace profiling also functions through this script, which is what the `profile` tag there is for.

Chapter 5

Results

5.1 Results

These are produced with the method detailed in subsection 4.2.10.

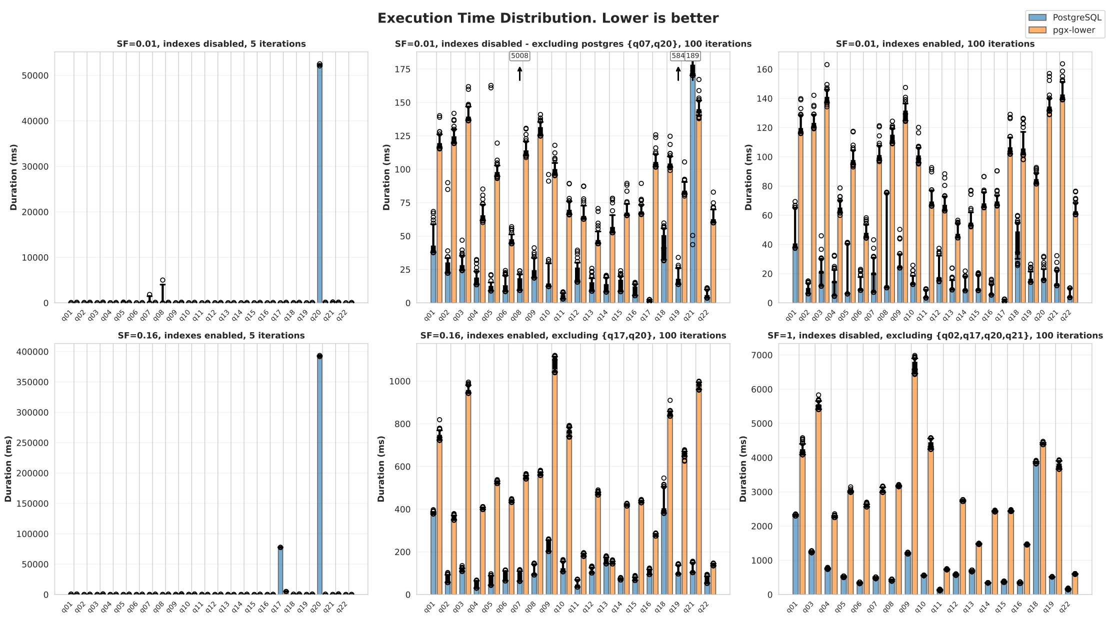

Figure 5.1: Overall benchmarking represented with box plots

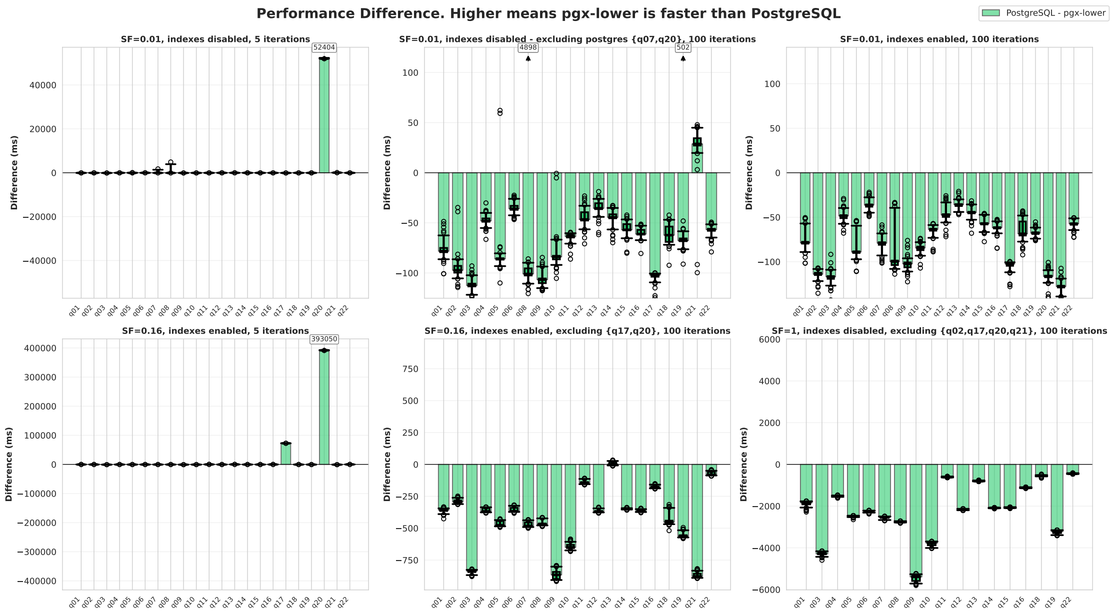

Figure 5.2: Difference in latency benchmarks between PostgreSQL and pgx-lower

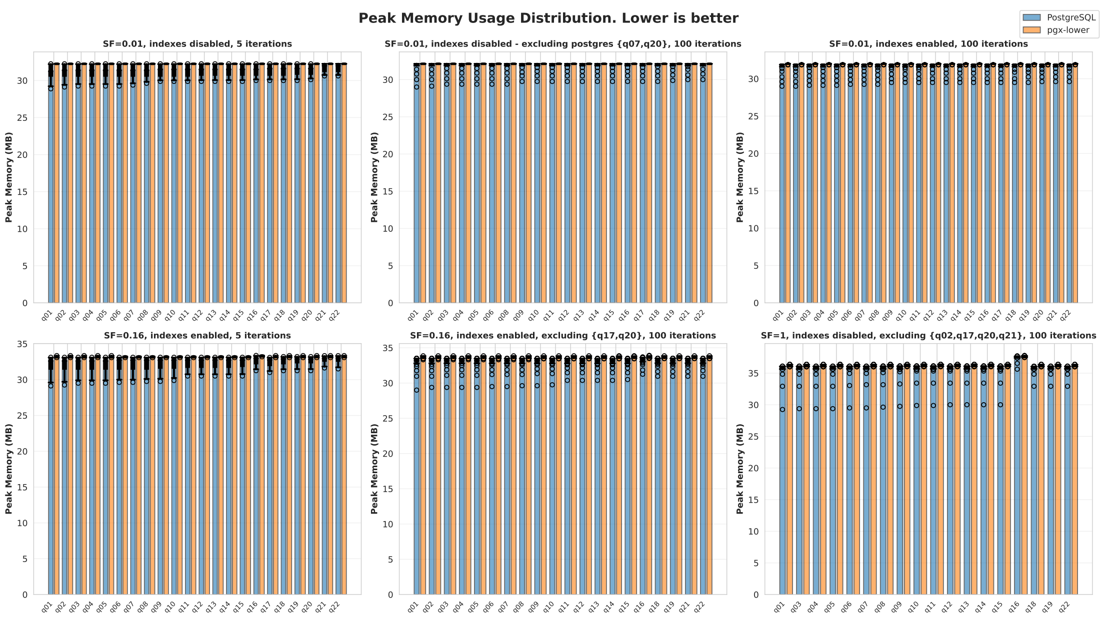

Figure 5.3: Peak memory usage of queries

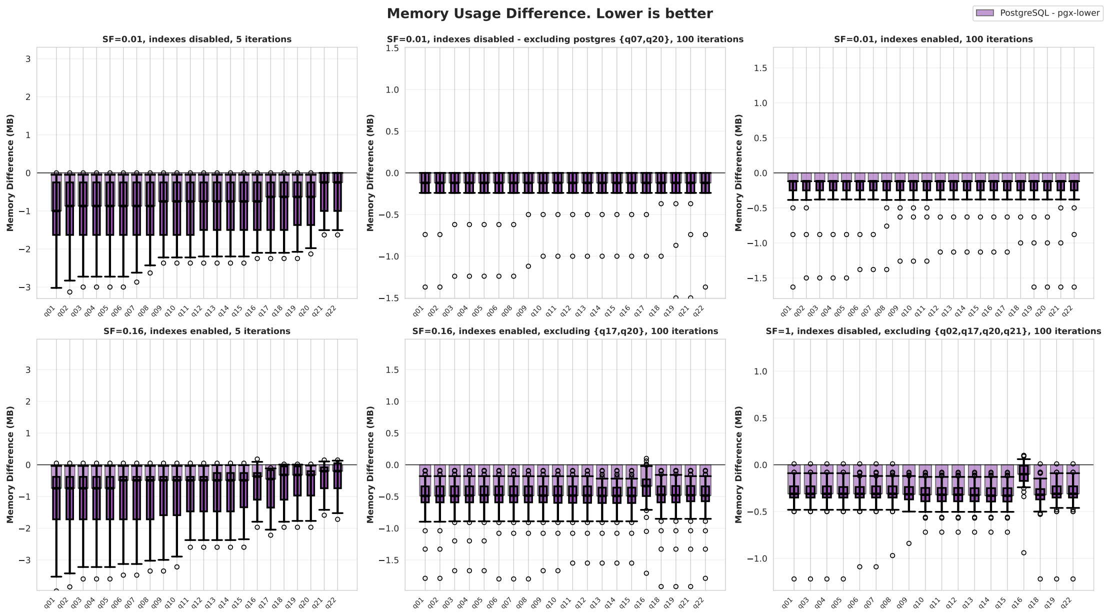

Figure 5.4: Difference in peak memory usage of queries

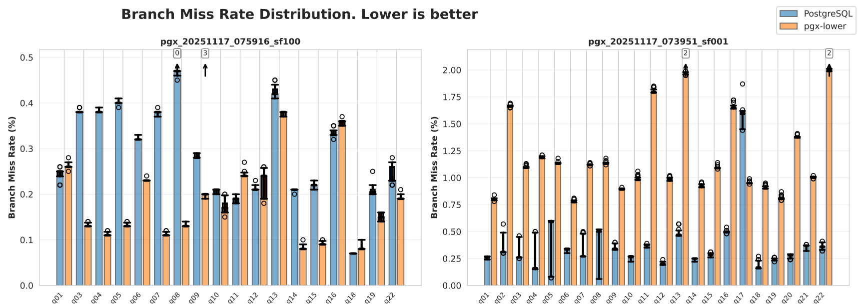

Figure 5.5: Branch miss rate

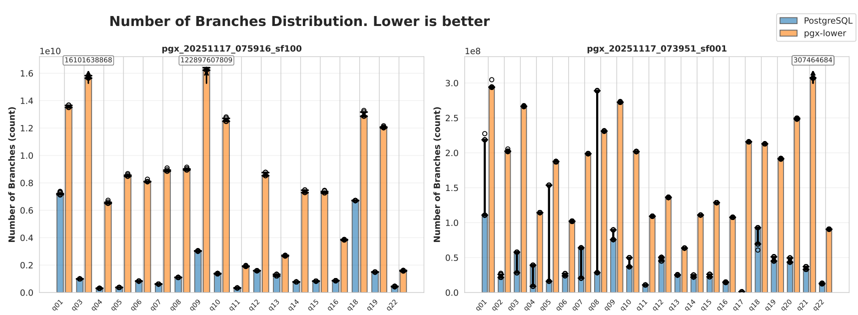

Figure 5.6: Number of branches

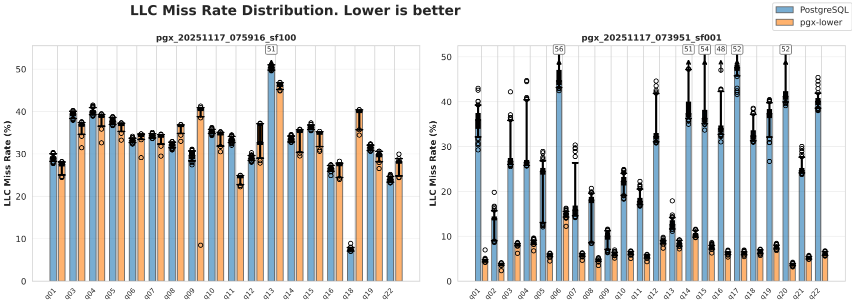

Figure 5.7: Last-level-cache miss plots

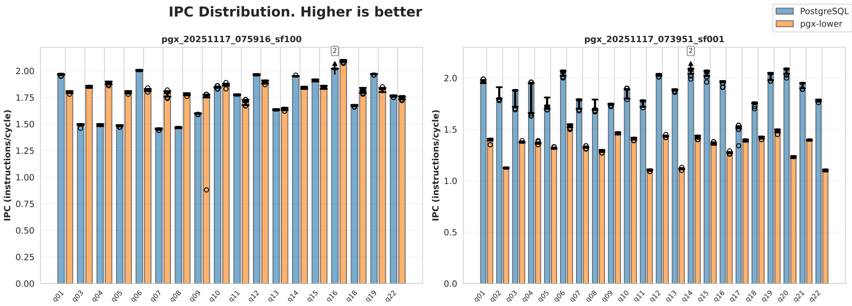

Figure 5.8: Instructions Per (cpu) Cycle plot

5.2 Discussion

5.2.1 Future work

Chapter 6

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.

[CRCG+19] Jesús Camacho-Rodríguez, Ashutosh Chauhan, Alan Gates, Eugene Koifman, Owen O'Malley, Vineet Garg, Zoltan Haindrich, Sergey Shelukhin, Prasanth Jayachandran, Siddharth Seth, Deepak Jaiswal, Slim Bouguerra, Nishant Bangarwa, Sankar Hariappan, Anishek Agarwal, Jason Dere, Daniel Dai, Thejas Nair, Nita Dembla, Gopal Vijayaraghavan, and Günther Hagleitner. Apache hive: From mapreduce to enterprise-grade big data warehousing. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, pages 1773–1786, Amsterdam, Netherlands, 2019. ACM.

[EH84] Wolfgang Effelsberg and Theo Haerder. Principles of database buffer management. *ACM Transactions on Database Systems*, 9(4):560–595, 1984.

[ESE06] Stijn Eyerman, James E. Smith, and Lieven Eeckhout. Characterizing the branch misprediction penalty. In *2006 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 48–58, Austin, Texas, USA, March 2006. IEEE.

[Far25] John Farrier. Branch prediction: The definitive guide for high-performance c++, March 2025. Accessed: 2025-11-18.

[Fre17] Andres Freund. Further insights into postgresql optimization techniques. `https://www.postgresql.org/message-id/flat/20170901064131.tazjxwus3k2w3ybh@alap3.anarazel.de`, 2017. Accessed: 2024-11-17.

[HD23] Immanuel Haffner and Jens Dittrich. A simplified architecture for fast, adaptive compilation and execution of sql queries. In *EDBT*, pages 1–13, 2023.

[HRTVVA21] José A. Herrera-Ramírez, Marlen Treviño-Villalobos, and Leonardo Víquez-Acuña. Hybrid storage engine for geospatial data using nosql and sql paradigms. *Revista Tecnología en Marcha*, 34(1):40, January–March 2021.

[Ine21] Elizabeth Inersjö. Comparing database optimisation techniques in postgresql: Indexes, query writing and the query optimiser. Bachelor's thesis, KTH Royal Institute of Technology, Stockholm, Sweden, 2021. TRITA-EECS-EX 2021:821.

[JKG22] Michael Jungmair, André Kohn, and Jana Giceva. Designing an open framework for query optimization and compilation. *Proceedings of the VLDB Endowment*, 15(11):2389–2401, 2022.

[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.

[KLK+18] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *Very Large Data Bases*, 11(13):2209–2222, 9 2018.

[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.

[LA04] Chris Lattner and Vikram S. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization*, CGO '04, pages 75–88, Palo Alto, California, 2004. IEEE Computer Society.

[LAB+20] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: A compiler infrastructure for the end of moore's law. *arXiv*, 2020.

[LLV25] LLVM Project. Orc design and implementation. `https://llvm.org/docs/ORCv2.html`, 2025. Accessed: 2025-11-18.

[Moo24] MoonCake. HTAP is dead, 2024. Accessed: 2025-11-18.

[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.

[Neu11] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment*, 4(9):539–550, 2011.

[NF20] Thomas Neumann and Michael J. Freitag. Umbra: A disk-based system with in-memory performance. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020.

[Pas25] PassMark Software. Intel core i5-760 @ 2.80ghz benchmark, 2025. Accessed: 2025-11-17.

[SFN22] Tobias Schmidt, Philipp Fent, and Thomas Neumann. Efficiently compiling dynamic code for adaptive query processing. In *ADMS@ VLDB*, pages 11–22, 2022.

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

[Tec25] TechPowerUp. Intel core i5-14600k specs, 2025. Accessed: 2025-11-18.

[Tra24] Transaction Processing Performance Council (TPC). TPC-H Benchmark, 2024. Accessed: 2024-11-17.

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