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 web servers 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 many 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 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 3. 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. Structured Query Language (SQL) is parsed, turned to RA (relational-algebra), optimized, executed, then materialized into a table [SKS19].

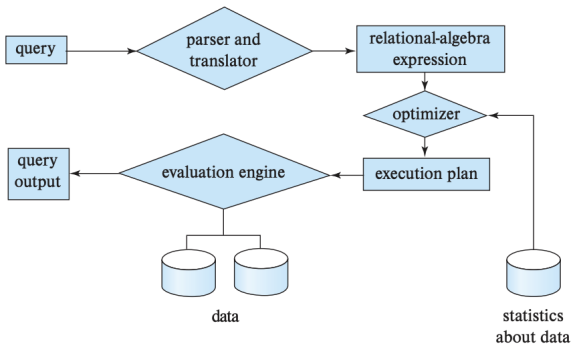


Figure 2.1: Database Structure
[SKS19]

For non-compiler databases, a volcano operator model tree is used, such as Figure 2.2 [ZVBB11]. The root node has a `produce()` function which calls its children's `produce()`, until it calls a leaf 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.

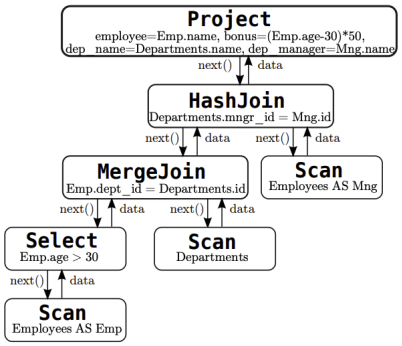


Figure 2.2: Volcano operator model tree.
[ZVBB11]

The fundamental issue with this classical model is that it is heavily under-utilising the hardware [ADHW99]. If only a single tuple is pulled, our CPU caches are barely used. An i5-570, a popular CPU in 2010 had an 8 megabyte L3 cache, but in 2024 an i5-14600K has a 24 megabyte L3 cache [Pas25], [Tec25]. For disks, in 2010 the Seagate Barracuda 7200.12 was popular, which had a sustained read of 138 MB/s, but in 2022 the Samsung V-NAND SSD 990 PRO released with a sustained read of 7450 MB/s [Sea07] [Sam24]. Such dramatic increases could mean the algorithms can fundamentally change.

This has led 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 [Zuk09]. 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 means 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 the era in which these systems were developed, when 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.

Most of the recent databasing research has focused on the optimiser, which is visible in figure 2.1. This includes reordering join statements where the left and right sides are flipped, predicate push down 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]. However, despite these advances, query optimisers frequently produce suboptimal plans [LGM+15, LRG+17]. Another essential pattern involves genetic algorithms being used in optimisers for determining things like join orders, which can cause the outputted plan to be non-deterministic [Ute97]. A thesis could be written just to summarise the list of optimisations.

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 eight kilobytes. This is loaded into a buffer pool object which holds it in 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 situation, such as last-recently-used, most-recently-used, and more, which can be decided in the optimiser [EH84]. The effectiveness of this can

be measured with last level cache hit rate measurements (LLC) which represents how many of the instructions were resolved inside the CPU cache [FRMK19].

Databases are commonly split into Online Transaction Processing (OLTP) and Online Analytical Processing (OLAP). OLTP focuses on supporting atomicity, running multiple queries at once, and typically handles the work profile of an online service that frequently performs key-value lookups. On the other hand, OLAP databases focus on analytical work profiles where aggregations are requested or operations span large chunks of the database [Kle19]. OLAP systems can be highly distributed, such as Apache Hive, which allows a large amount of compute to be used across the system [CRCG+19]. In the context of PostgreSQL, it is a hybrid architecture that supports both OLTP and OLAP operations [HRTVVA21]. There is currently debate about whether this hybrid architecture is still useful, as putting pressure on a 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 primarily used with 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 initial compilation can be faster, and the development cycle can go faster, as well as other benefits.

Due to branch-prediction optimisation, 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 optimisations 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 [Jos21]. 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 optimiser 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 JIT compilers with support for lazy compilation, concurrent compilation, and runtime optimisation [LLV25b]. 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

The V8 compiler used for WebAssembly has a unique architecture because it targets short-lived programs. Majority of JIT applications are used for long-running services, but this is used for web pages which are opened and closed frequently. To mitigate this, they have a two-phase architecture where code is first compiled with Liftoff for a quick startup, then hot functions are recompiled with TurboFan [HRS+17]. Liftoff aims to create machine code as fast as possible and skip optimisations.

Other common JIT compilers are the Java Virtual Machine (JVM), SpiderMonkey (Mozilla Firefox's JIT), JavaScriptCore/Nitro (Safari/Webkit), PyPy, various python JIT compilers, LuaJIT for Lua, HHVM for PHP, Rubinius for Ruby, RyuJIT for C#, and more [DCL25]. The JVM has also been used for compiled query execution engines [RPML06]. These all target different work profiles.

2.5 PostgreSQL Background

PostgreSQL relies on memory contexts, which are an extension of arena allocators. 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. A memory context can create child contexts, and when a context is freed it also frees all the children of this context, turning it into a tree of arena allocators. There is a set of statically defined memory contexts: TopMemoryContext, TopTransactionContext, CurTransactionContext, TransactionContext, which are managed through PostgreSQL's Server Programming Interface (SPI) [Pos25b].

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. These stages are visible in figure 2.1. 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) [Pos25d].

PostgreSQL provides the `EXPLAIN` command to inspect query execution plans, which is essential for understanding and optimizing query performance [Pos25a]. This command displays the execution plan that the planner generates, including cost estimates and optional execution statistics, making it a critical tool for database optimisation and analysis.

2.6 Database Benchmarking

Benchmarking a database is a difficult task because there is a variety of workloads. Many systems create their own benchmarking libraries, such as `pg_bench` by PostgreSQL [Pos25c] or LinkBench [APBC13] by Facebook, but in academics the more common benchmarks are from the Transaction Processing Council, which is a group

that defines benchmarks [BNE14]. Over the years they have made TPC-C for an order-entry environment, TPC-E, for a broker firm's operations, TPC-DS for a decision support benchmark. The most common one in research appears to be TPC-H, where the H informally means "hybrid". It has a mix of analytical and transactional elements inside it [BNE14].

When evaluating benchmark results, the *coefficient of variation* (CV) will be used, and it is a standardized measure of dispersion that expresses the standard deviation as a percentage of the mean [Sta25]. It is calculated as $CV = \frac{s}{\bar{X}} \cdot 100$, where s is the sample standard deviation and $\bar{X}$ is the sample mean. The coefficient of variation is particularly useful when comparing the variability of datasets with different units or scales, as it provides a unitless measure that enables relative comparison of measurement precision across different benchmarking configurations.

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 [CAB+81], 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 [KLK+18].

This begins with PostgreSQL and their extension system in section 3.2, in section 3.3 system R will be explored as the classical example, followed by HyPer and Umbra in section 3.4 and section 3.5 which re-introduced the concept. Mutable in section 3.6 and LingoDB in section 3.7 are research databases. Lastly, PostgreSQL will be examined in section 3.2 as it uses expression-based compilation and there has been an attempt to create a compiled engine before.

3.1 OLAP Systems

Most of the benefit of a compiled query engine is in OLAP contexts since the OLTP contexts are usually retrieval queries and not too complicated [Kle19]. In large-scale contexts Apache HIVE is common, but this is a data warehouse system, not a database, that contains data within Hadoop's distributed file store, which is closer to a flat-

storage [CRCG+19]. The common OLAP databases are MonetDB, SnowFlake, ClickHouse, RedShift and Vectorwise [SZC+24]. For our context, understanding ClickHouse, NoisePage, DuckDB and extensions that turn PostgreSQL into an OLAP database are important.

ClickHouse and NoisePage are standalone systems, while DuckDB is embedded and in-process, similar to SQLite. ClickHouse is a columnar, disk-oriented database with a vectorised execution engine with optional LLVM compilation for expressions (EXP) [SSY+24]. NoisePage was created by the Carnegie Mellon Database group, and is columnar and in-memory with full query expression compilation (QEP), with a single node. They targeted ML-driven self optimisation in their research, but they were archived in February 21, 2023 [MZJ+21]. DuckDB is marketed as the SQLite for analytical loads with in-memory disk spillage, or like a more sophisticated Pandas DataFrame [RM19]. Their engine supports vectorised execution because JIT would add too much overhead to their lightweight philosophy.

3.2 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 Stack Overflow survey saying they use it extensively in 2024 [Ove24]. 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.

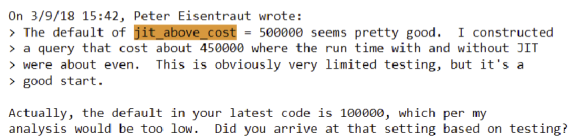
Building extensions for PostgreSQL and making entire companies around these extensions is not a new endeavour. Three such examples are Citus [CEP+21], TimescaleDB [Tim24a], and Apache AGE [Apa24]. Citus aims to add more horizontal scaling through sharding, TimescaleDB, now rebranded as TigerData, transforms the engine into a time series database, and Apache AGE turns it into a graph database. These all have thousands of GitHub stars, but TimescaleDB especially had over 500 paying customers in 2022,

and claimed their revenue climbed 20x by 2024 [Tim24b] [Bus22]. This shows that the model of relying on the extension system is a robust, travelled road.

There have also been several extensions that attempt to make PostgreSQL more suited to OLAP workloads, with the most relevant one being pg_duckdb [Duc24]. This system replaces PostgreSQL's engine with DuckDB which allows it to take advantage of the vectorised engine, and it is reasonably popular with roughly 2700 stars on GitHub. Hydra is also worth mentioning, but this is closer to TimescaleDB [Hyd24] and makes the system columnar with compression support. ParadeDB's pg_analytics initially started in the same way as pg_duckdb, but pivoted into supporting search functionality instead, similar to Elasticsearch.

There has been significant discussion about HyPer and JIT in regard to PostgreSQL in 2017 [Fre16, Fre17]. The general response expressed doubt that someone would add support for full query compilation, noting that rearchitecting such a core component introduces significant risk.

However, in September 2017 Andres Freund started implementing JIT support for expressions [Fre18]. The reasoning was that most of the CPU time is in the expression components, (such as 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. PostgreSQL's JIT implementation is documented in the official PostgreSQL documentation [Pos24].


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 Eisentraut asking whether the defaults are too low. [Fre17]

In the pull request Peter Eisentraut 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 would give it exposure and testing [She17]. 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 [Xen21]. This incident reinforced the view that JIT features should be disabled by default, and has led to 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 speed-up on TPC-H query 1, and has more documentation [ISP17]. However, they did not publicize their implementation or show that it's easy for people to use.

This implementation of full JIT in PostgreSQL was preceded by profiling work which showed that different TPC-H benchmarks place pressure on different nodes, as shown in figure 3.2. This is important because they can make informed decisions about where to focus their efforts. Their core method is generating a function that represents a node in the plan tree, then inlining the function into the final LLVM IR [ISP17] in a push-based model. Another interesting method that is used is pre-compiling the C code into LLVM, then inlining that into the LLVM IR. This avoids runtime linking back to the C code [ISP17], similar to approaches taken by HyPer [Neu11].

Function	TPC-H Q1	TPC-H Q2	TPC-H Q3	TPC-H Q6	TPC-H Q22
ExecQual	6%	14%	32%	3%	72%
ExecAgg	75%	-	1%	1%	2%
SeqNext	6%	1%	33%	-	13%
IndexNext	-	57%	-	-	13%
BitmapHeapNext	-	-	-	85%	-

Figure 3.2: PGCon Dynamic Compilation of SQL Queries Profiling [ISP17]. [Fre17]

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.

3.3 System R

System R is a flagship paper in the databasing space that introduced SQL, compiling engines, and ACID [CAB+81]. 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.

3.4 HyPer

HyPer was a flagship system, and Umbra supersedes it. These are important systems in the JIT-database space as they developed many of the core features. Both were made by Thomas Neumann, and a core sign of its viability is that HyPer was purchased by Tableau in 2016 to be used in production [Tab18]. This shows that it 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 [Neu11], and in 2018 they released another flagship paper about adaptive compilation [KLN18]. 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 3.3: HyPer OLAP performance compared to other engines.
[Neu11]

HyPer shows they reduced their compile time by doing this in figure 3.3 by many multiples, and in figure 3.4 they show they achieve many times fewer branches, branch mispredictions, and other measurements compared to their baseline of MonetDB [BZN05]. 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 3.4: 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 3.5 visualises this with the compile times of each stage. However,

they had to create the byte code interpreter themselves to enable this.

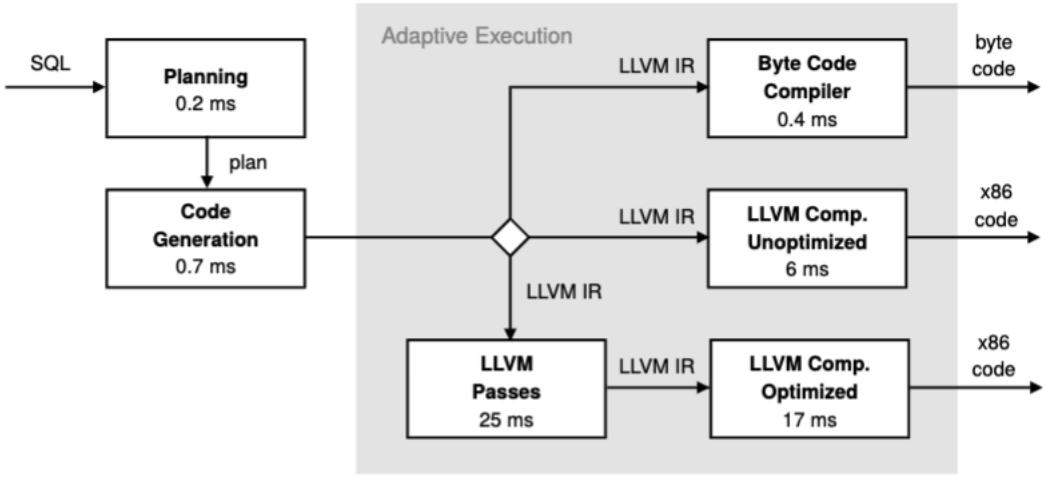


Figure 3.5: HyPer execution modes and compile times.
[KLN18]

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.

3.5 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 an on-disk database [NF20]. 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.

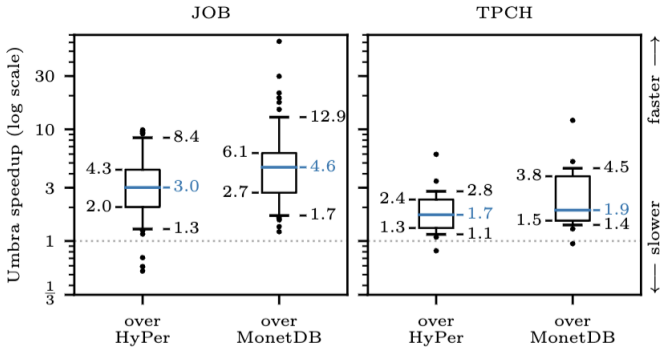


Figure 3.6: Umbra benchmarks.
[NF20]

A novel optimisation they introduced later was enabling the compiler to change the query plan [SFN22]. 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 a factor of two. 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.

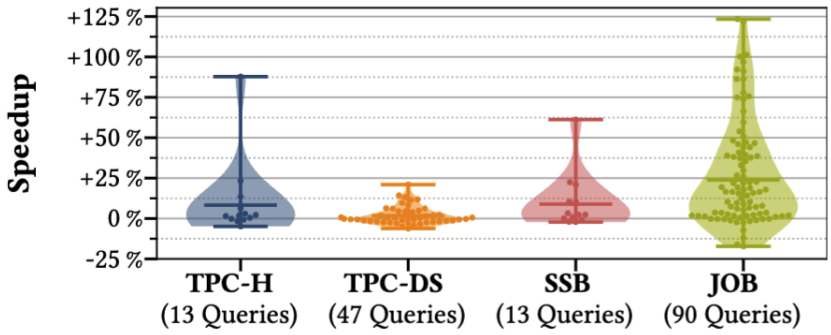


Figure 3.7: Umbra benchmarks after adaptive query processing (AQP). [SFN22]

Umbra is currently ranked as the most effective database on ClickHouse's benchmarking [Cli24]. 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. Additionally, Umbra supports user-defined operators, enabling efficient integration of custom algorithms into the database [SN22].

3.6 Mutable

In 2023, Mutable presented the concept of using a low-latency JIT compiler (WebAssembly) rather than a heavy one in their initial paper [HD23a]. 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 [HD23b]. However, using WebAssembly meant they can omit most of the optimisations that HyPer did while maintaining higher 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 [Ham18]. The LiftOff component produces machine code as fast as possible while skipping optimisations, 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.

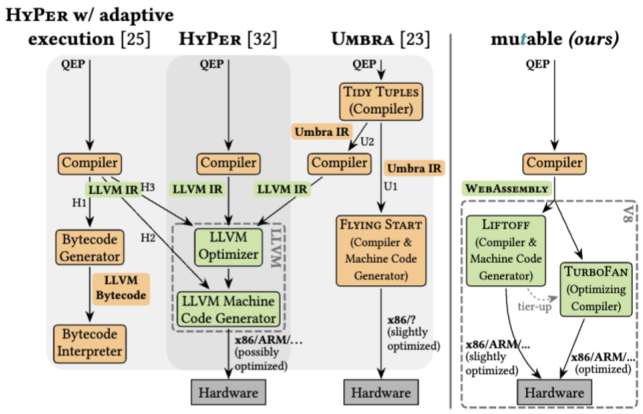


Figure 3.8: Comparison of mutable to HyPer and Umbra. [HD23b]

Mutable's benchmarks show they achieve similar compile and execution times to HyPer, and outperform them in many cases [HD23b]. 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.

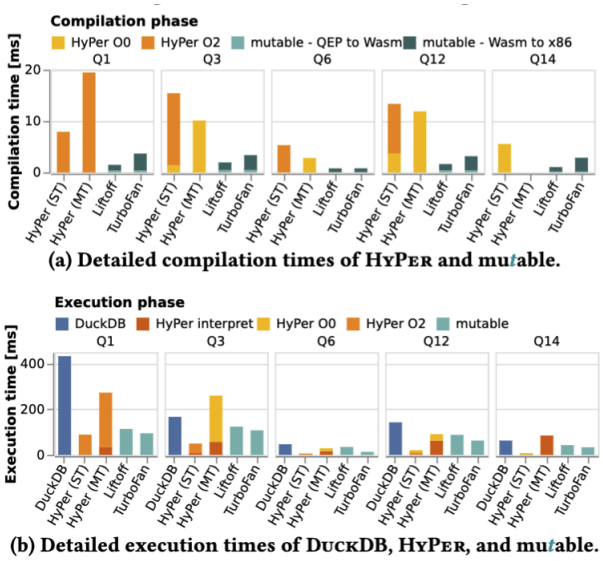


Figure 3.9: Benchmarks produced by Mutable. [HD23b]

3.7 LingoDB

LingoDB piloted in 2022 and proposed using the MLIR framework to create the optimisation layers [JKG22]. 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 3.10, 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 database-specific types and operations, a DSA dialect that represents data structures and algorithms, a utility dialect that represents utilities, 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 [JKG22].

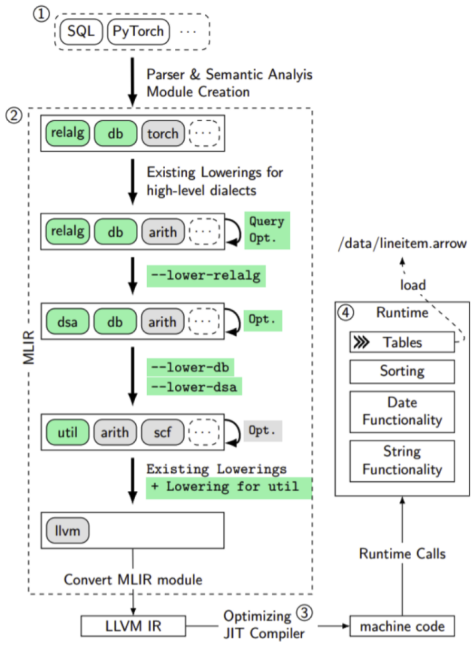


Figure 3.10: LingoDB architecture [JKG22]
[JKG22]

Their result is that they are less performant than HyPer, but do better than DuckDB [JKG22]. 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.

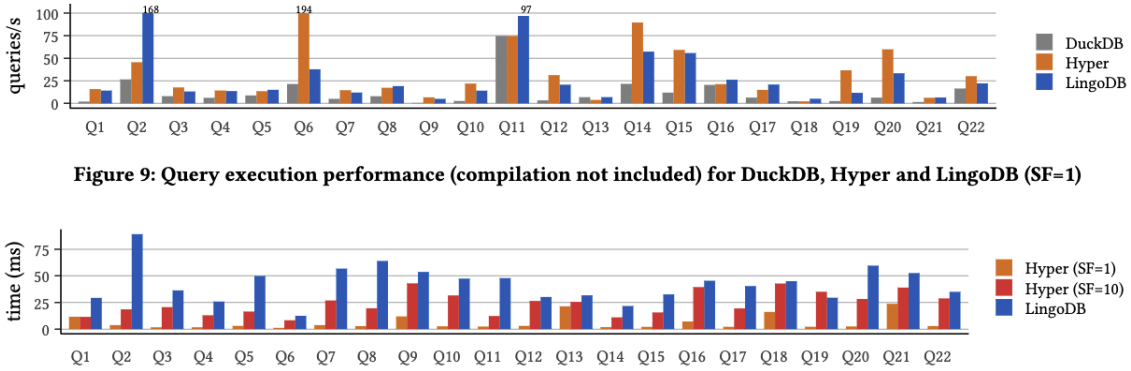


Figure 9: Query execution performance (compilation not included) for DuckDB, Hyper and LingoDB (SF=1)

Figure 3.11: 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 [JG23]. 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 Section 4.1.

3.8 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 3.12. TPC-H was used as most of the involved pieces used it themselves [BNE14], 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.

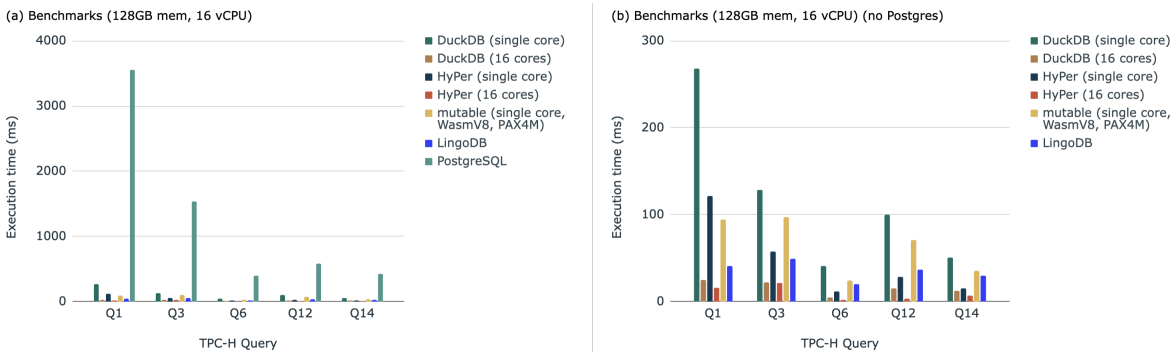


Figure 3.12: Benchmarking results.

To identify how much potential gain there is in a major on-disk database, `perf` was used on PostgreSQL during TPC-H queries 1, 3, 6, 12 and 14 in figure 3.13 [Linnd]. These queries were chosen because the Mutable code infrastructure directly supported them. 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.

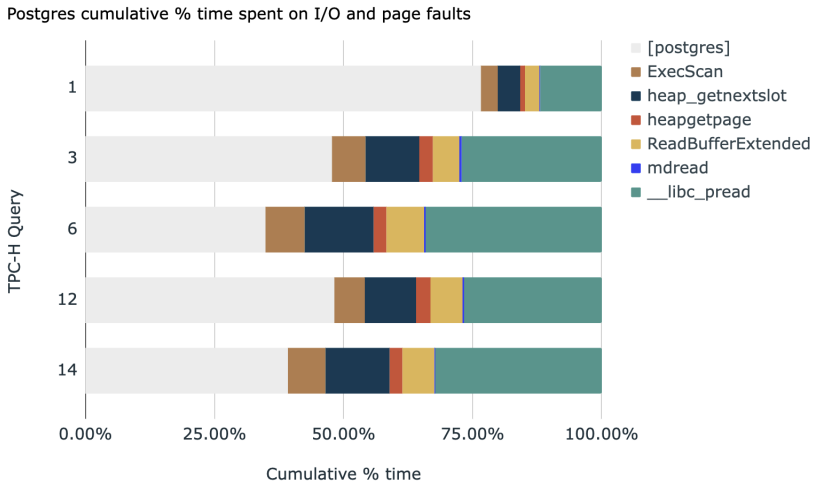


Figure 3.13: PostgreSQL's time spent in the CPU, measured with prof.

3.9 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 use case of the systems and target the 0.01% of users [Sto18]. 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 [KN20], 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.

3.10 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. Users can install the extension, have faster queries with rollbacks, and the implementation effort is lowered since core systems and algorithms can be skipped.

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, benchmarks can be executed multiple times, 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 is TimescaleDB (now TigerData) [Tim24a], which was explored in section 3.2. 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 numerous features and optimisations that are not relevant to us, so to simplify implementation effort the 2022 version was used from their initial paper.

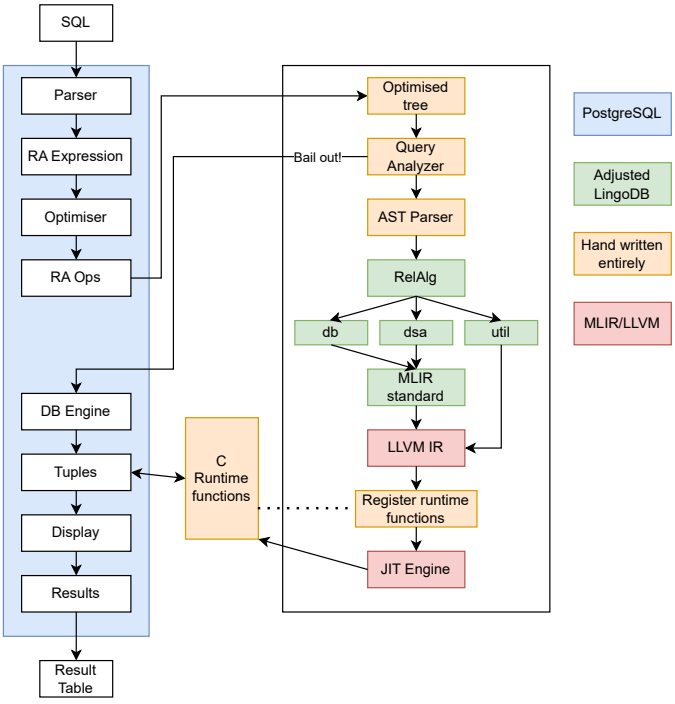


Figure 4.1: System design with labels of component sources.

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 handwritten analyser for whether the query can be executed, and then parsed. These hand written 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.

Table 4.1 shows the complete regression test suite, organized to progressively build complexity from single-row operations to full TPC-H queries.

Table 4.1: Regression test suite files and their aims

File Name	**Aim**
1_one_tuple.sql	Single row insertion and selection
2_two_tuples.sql	Multiple row retrieval
3_lots_of_tuples.sql	Large dataset handling (5000 rows)
4_two_columns_ints.sql	Multiple integer columns
5_two_columns_diff.sql	Mixed column types (INTEGER, BOOLEAN)
6_every_type.sql	All supported data types with projections
7_sub_select.sql	Column subsetting across Boolean columns
8_subset_all_types.sql	Column subsets from multi-type tables
9_basic_arithmetic_ops.sql	Arithmetic operators (+, -, *, /, %)
10_comparison_ops.sql	Comparison operators (=, ¡¿, !=, ¡, ¡=, ¿, ¿=)
11_logical_ops.sql	Logical operators (AND, OR, NOT)
12_null_handling.sql	NULL operations (IS NULL, COALESCE)
13_text_operations.sql	Text operations (LIKE, ||)
14_aggregate_functions.sql	Aggregates without GROUP BY (SUM, COUNT, AVG, MIN, MAX)

Continued on next page

Table 4.1 – *Continued from previous page*

File Name	**Aim**
15_special_operators.sql	BETWEEN, IN, CASE WHEN
16_debug_text.sql	CHAR(10) with LPAD
17_where_simple_conditions.sql	WHERE with simple comparisons
18_where_logical_combinations.sql	WHERE with AND/OR/NOT combinations
19_where_null_patterns.sql	WHERE with NULL checks and COALESCE
20_where_pattern_matching.sql	WHERE with LIKE and IN operators
21_order_by_basic.sql	ORDER BY single columns (integers, strings, decimals)
22_order_by_multiple_columns.sql	ORDER BY multiple columns with mixed directions
23_order_by_expressions.sql	ORDER BY expressions (not supported - placeholder)
24_order_by_with_where.sql	ORDER BY combined with WHERE
25_group_by_simple.sql	GROUP BY with aggregations and ORDER BY
26_before_check_types.sql	All PostgreSQL types with casts and aggregations
27_group_by_having.sql	GROUP BY with HAVING clause
28_group_by_with_where.sql	GROUP BY with WHERE filtering
29_expressions_in_aggregations.sql	Expressions in aggregates (arithmetic, ABS)
30_test_missing_expressions.sql	PostgreSQL expression types coverage
31_distinct_statement.sql	DISTINCT in SELECT and aggregates
32_decimal_maths.sql	Decimal arithmetic operations
33_basic_joins.sql	INNER JOIN
34_advanced_joins.sql	LEFT, RIGHT, SEMI, ANTI joins
35_nested_queries.sql	Nested and correlated subqueries
36_tpch_minimal.sql	TPC-H minimal schema variant 1
37_tpch_minimal_2.sql	TPC-H minimal schema variant 2
38_tpch_minimal_3.sql	TPC-H minimal schema variant 3
39_tpch_minimal.sql	TPC-H minimal schema variant 4
40_tpch_not_lowered.sql	TPC-H queries without lowering
41_sorts.sql	Sorting with joins
42_test_relalg_function.sql	Direct RelAlg MLIR execution
init_tpch.sql	TPC-H table initialization
tpch.sql	Full TPC-H benchmark in pgx-lower
tpch_no_lower.sql	TPC-H inside pure PostgreSQL for validation

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, needed 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 4.2.3 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. If an error is thrown in pgx-lower, the progress in compiling is dumped then it fully falls over to PostgreSQL, even if the result is half-complete.

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 test components unconnected to PostgreSQL. The issue is that this extension creates a `pgx_lower.so` which is loaded into PostgreSQL, and the PostgreSQL libraries are used from that context. This means if we run without being inside PostgreSQL, no psql libraries 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, get 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 4.2 shows which of the LingoDB types are used, and table 4.3 shows the type mappings. The two primary workarounds that were implemented were for decimals and the various types of strings. For decimals, `i128` provides enough precision for most TPC-H tests, which is what LingoDB was using. However, adjustments had to be made to prevent values that cannot be allocated from appearing, so the precision was capped at `<32, 6>`. That is, 32 digits in the integer part and 6 digits in the decimal places.

For the date types, a compromise 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 4.2: 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 4.3: PostgreSQL type translation through DB dialect to LLVM

4.2.5 LingoDB Dialect Changes

The MLIR dialects from LingoDB required modifications to support LLVM 20 and pgx-lower's specific needs. Table 4.4 summarizes the key changes across the DB, DSA, RelAlg, and Util dialects. The changes were primarily API compatibility updates with minimal semantic modifications: 94 operations total across all dialects (93 in LingoDB, 94 in PGX), 21 types (identical count), and mostly LLVM 20 API compatibility updates.

Category	**Count**	**Description**
MLIR API Updates	30 changes	NoSideEffect → Pure, Optional → std::optional, added OpaqueProperties
Include Path Changes	100% of files	mlir/Dialect/ → lingodb/mlir/Dialect/
Dialect Renames	1	Arithmetic → Arith (MLIR core change)
New Operations	1	SetDecimalScaleOp in DSA
Modified Operations	2	DSA_SortOp (supports collections), BaseTableOp (added column_order)
Namespace Clarifications	6 interfaces	Added explicit cppNamespace declarations
Convenience Features	4 dialects	Added useDefaultTypePrinterParser = 1
Fastmath Support	2 patterns	Added fastmath flags to arithmetic canonicalization
Code Cleanup	5 locations	Removed comments, simplified logic

Table 4.4: Summary of key differences between LingoDB and pgx-lower dialects

This defines most of the supporting details. The main two components of the implementation are the runtime patterns and the plan tree translation.

4.2.6 Query Analyser

While the query analyser is fully written, in the final state it routes all queries through pgx-lower for testing. This enables testing new features and identifying where failures occur. It functions by doing a depth-first search through the plan tree and validating that the nodes are supported by the engine. In the future, this component could be enhanced to decide whether a query is worth running based on its cost metrics.

4.2.7 Runtime patterns

Runtime functions are used in LingoDB for methods that are difficult to implement in LLVM, such as sorting algorithms. `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 hash table 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.

LingoDB had a code generation step in their CMakeLists, `gen_rt_def`, which supports this. It parses a given C++ file, then generates a header file in the build files which has the mangled name lookup, so that the developer does not need to reimplement that section repeatedly.

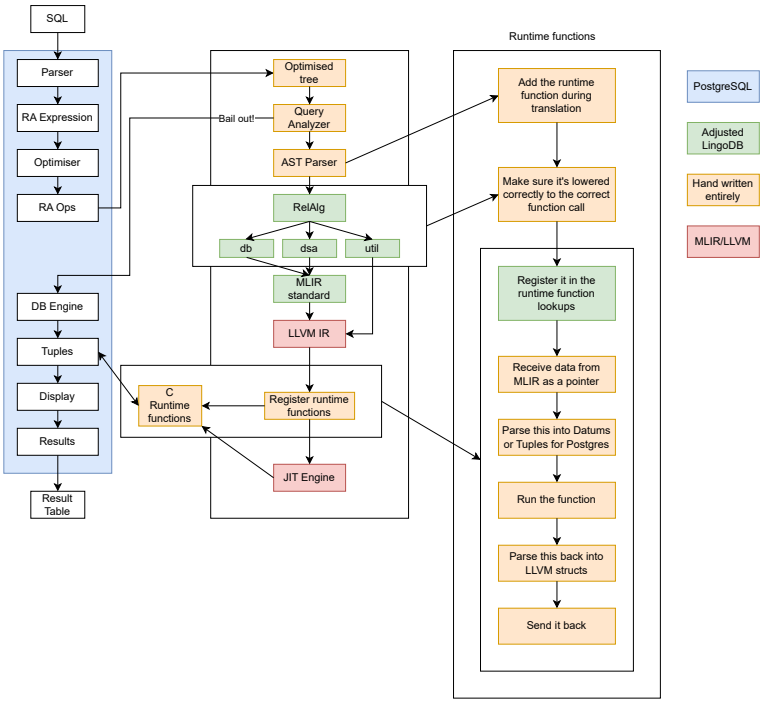


Figure 4.2: System design with labels of component sources.

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

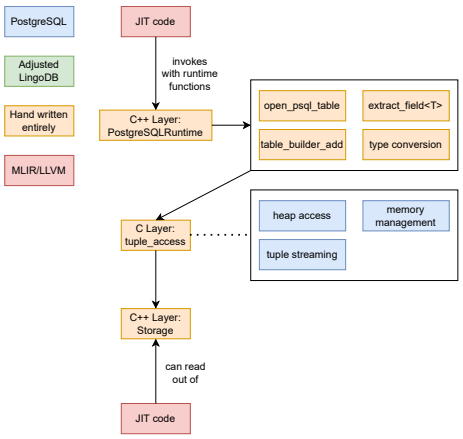


Figure 4.3: PostgreSQLRuntime.h component design

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.

4.2.8 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 examine the PostgreSQL tag on the node to determine the plan node type, then a recursive descent parser starts translating. Within each translation function, the pattern is typically: the children of the plan are translated (post-order traversal), then the node is translated into the MLIR relational algebra dialect, and a "translation result" is returned.

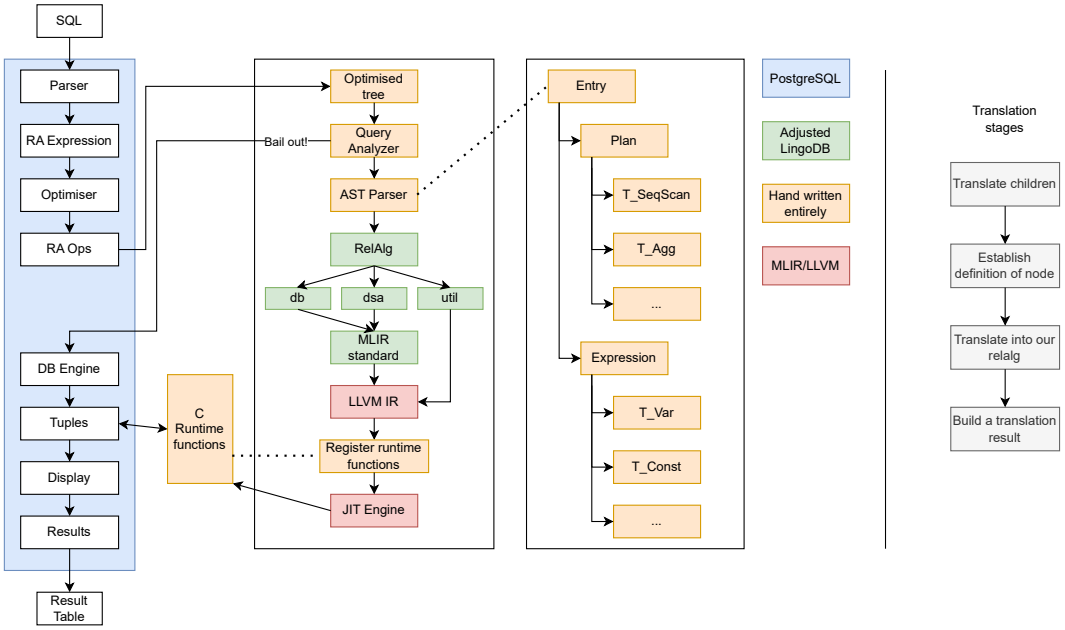


Figure 4.4: AST translation design and high-level steps in each function

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 `TranslationResult`s 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;
```

```
14 auto translate_nest_loop(QueryCtxT& ctx, NestLoop* nestLoop) -> TranslationResult;
15 auto translate_material(QueryCtxT& ctx, const Material* material) -> TranslationResult;
16 auto translate_memoize(QueryCtxT& ctx, const Memoize* memoize) -> TranslationResult;
17 auto translate_subquery_scan(QueryCtxT& ctx, SubqueryScan* subqueryScan) -> TranslationResult;
18 auto translate_cte_scan(QueryCtxT& ctx, const CteScan* cteScan) -> TranslationResult;
```

The 14 expression node types are documented in Table 4.5, and the 18 plan node types in Table 4.6. 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 4.5: 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 4.6: 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 initialising 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

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 nor 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 not utilised, 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 `EXISTS`/ `NOT EXISTS` translations 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 introduces 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 surrounding space, 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. With these casts, the SQLTypeInference object supports upcasting such that if two types have an operation, they are both first cast into the larger datatype, such as `i16` + `i32`, then the smaller one will both be cast into `i32` first.

Translation - Others

Many 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.

4.2.9 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.

These passes perform fundamental optimisations [LLV25a]. SROA (Scalar Replacement of Aggregates) promotes stack-allocated structures into SSA registers. That is, an allocation on the stack is hoisted up into global space so that the space is reused rather than reallocated every time. InstCombine simplifies instructions through algebraic transformations; PromotePass elevates memory operations to register operations; LICM (Loop Invariant Code Motion) moves loop-independent code outside of loops by hoisting to preheaders or sinking to exit blocks; the reassociation pass reorders expressions to enable further optimisations; and GVN (Global Value Numbering) eliminates redundant computations by identifying values that must be equal.

The 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.

4.2.10 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, 16 GB 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.

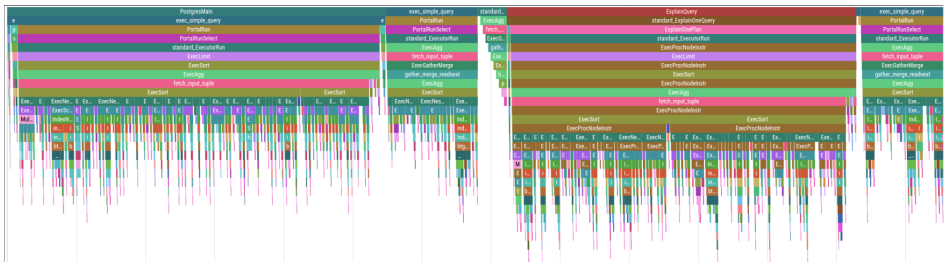


Figure 4.5: PostgreSQL's magic-trace flame chart for TPC-H query 3 at scale factor 0.05 (approximately 5 megabytes of data)

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.

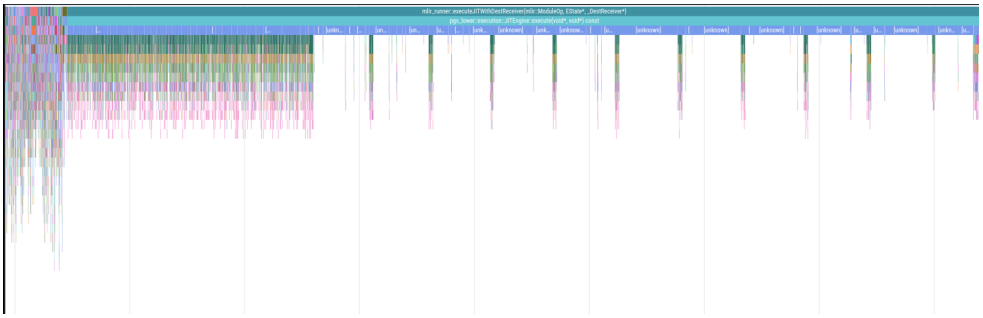


Figure 4.6: pgx-lower's magic-trace flame chart for TPC-H query 3 at scale factor 0.05 before optimisation

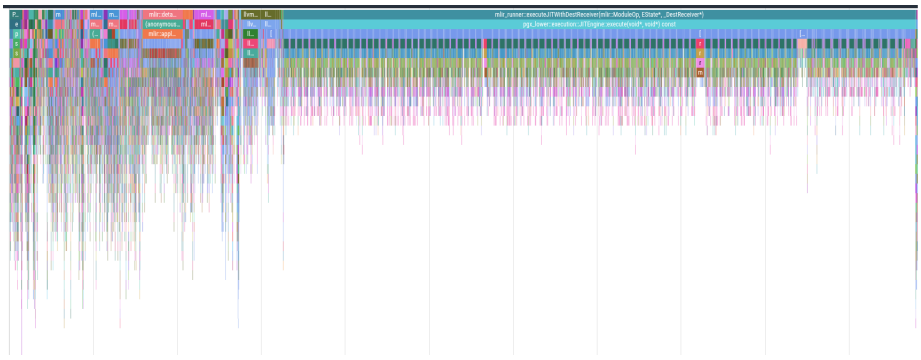


Figure 4.7: pgx-lower's magic-trace flame chart for TPC-H query 3 at scale factor 0.05 after optimisation

The specifics of running these in a stable way will be explained in subsection 4.2.12.

4.2.11 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 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, and Docker containers to support the reverse proxy with Nginx, and a private Grafana health dashboard.

4.2.12 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.

To disable indexes, `cur.execute("SET enable_indexscan = off;")` and `cur.execute("SET enable_bitmapscan = off;")` were used in conjunction. This means when the benchmarks say index scan is disabled, the bit map scan is as well.

Listing 4.2: Benchmark configurations for TPC-H testing

```
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"

```

```
27      - container: benchmark
28        scale_factor: 0.16
29        iterations: 5
30        profile: false
31        indexes: true
32        skipped_queries: ""
33        label: "SF=0.16, indexes enabled, 5 iterations"
34
35      - container: benchmark
36        scale_factor: 0.16
37        iterations: 100
38        profile: false
39        indexes: true
40        skipped_queries: "q17,q20"
41        label: "SF=0.16, indexes enabled, excluding {q17,q20}, 100 iterations"
42
43      - container: benchmark
44        scale_factor: 1
45        iterations: 100
46        profile: false
47        indexes: false
48        skipped_queries: "q02,q17,q20,q21"
49        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 3.8 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 and Discussion

5.1 Results

The following results were produced with the method detailed in subsection 4.2.12. Box plots were stacked on top of the graphs, representing the 5th, 25th, 50th, 75th, and 95th percentiles. Any outliers were marked with a hollow circle, and if they were inconvenient to show (such as in figure 5.4), an arrow annotation is utilised. Matplotlib and Seaborn were used to make these in Python.

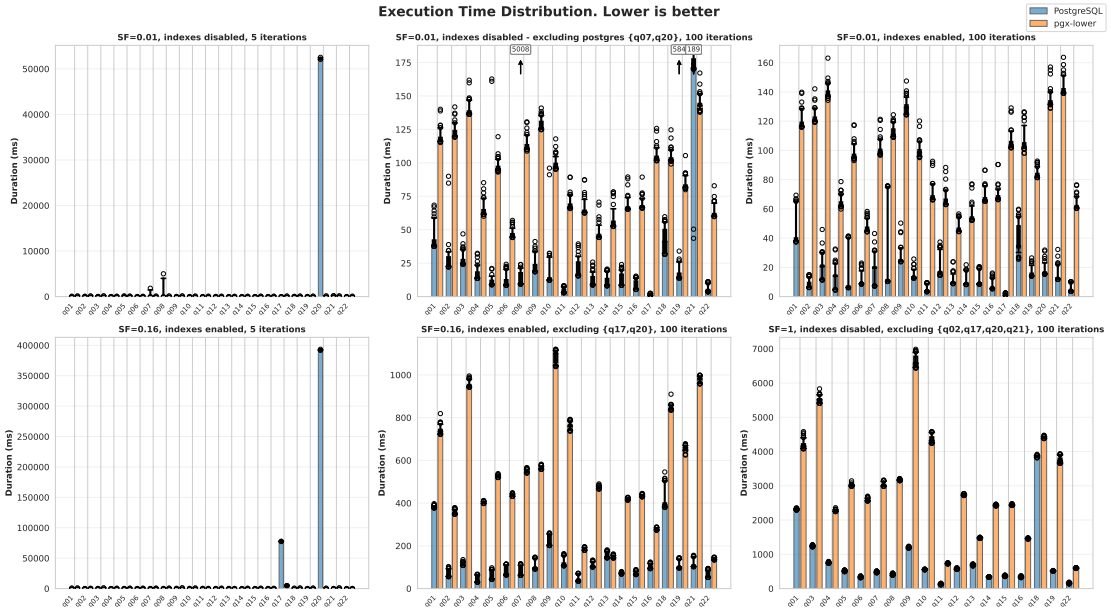


Figure 5.1: Overall benchmarking represented with box plots

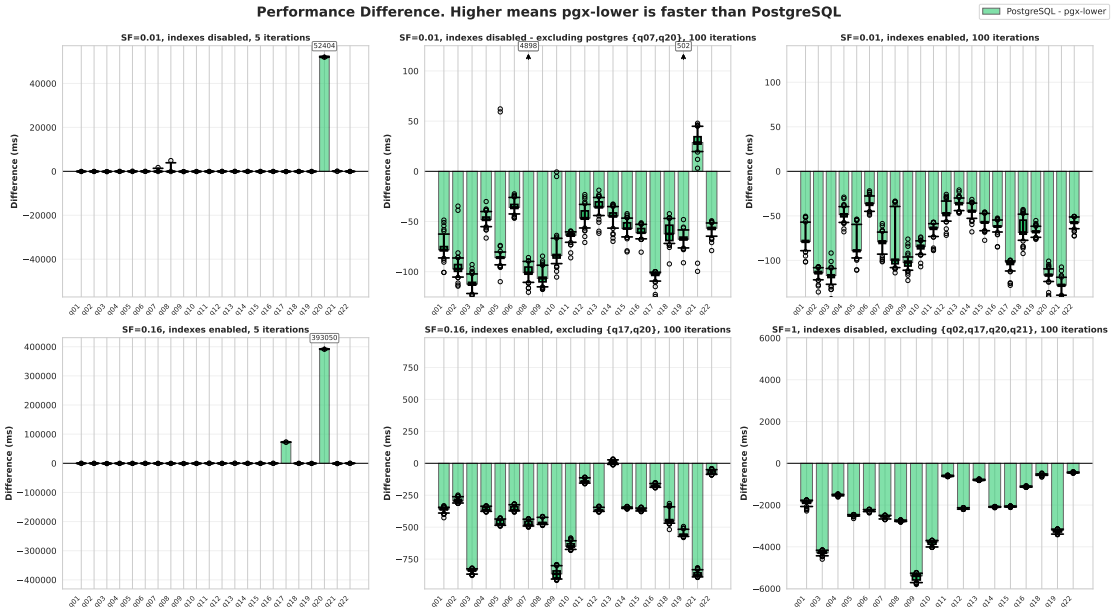


Figure 5.2: Difference in latency benchmarks between PostgreSQL and pgx-lower

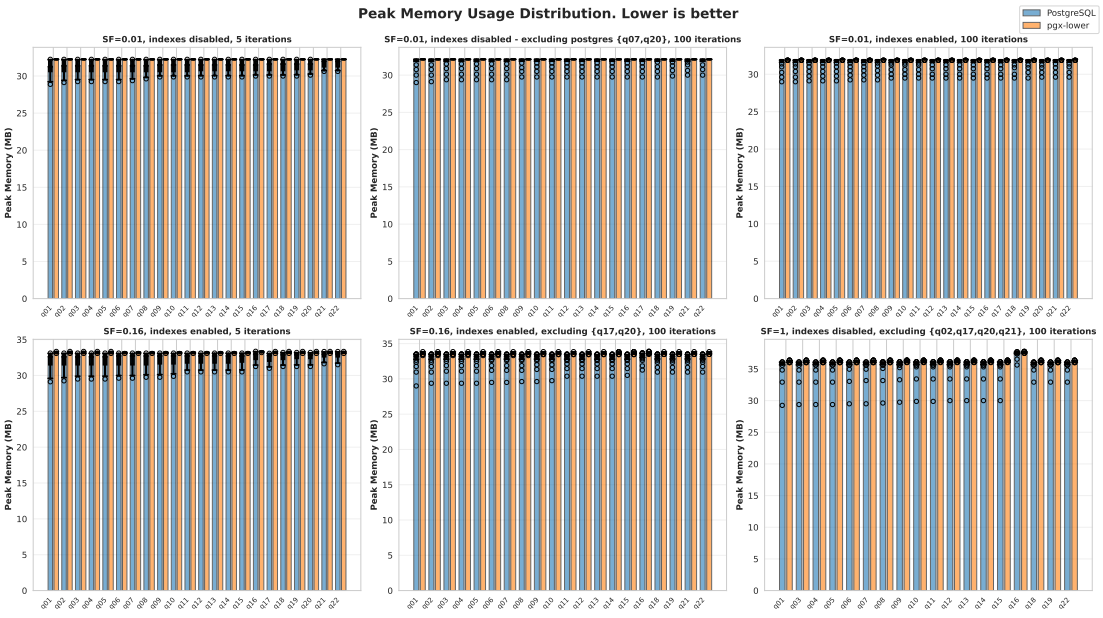


Figure 5.3: Peak memory usage of queries

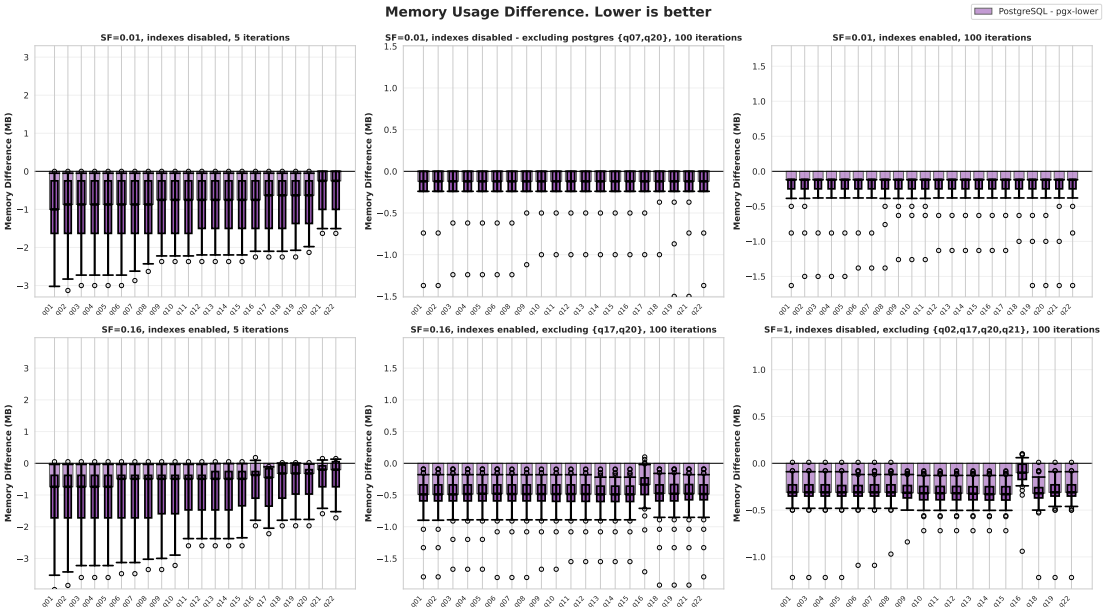


Figure 5.4: Difference in peak memory usage of queries

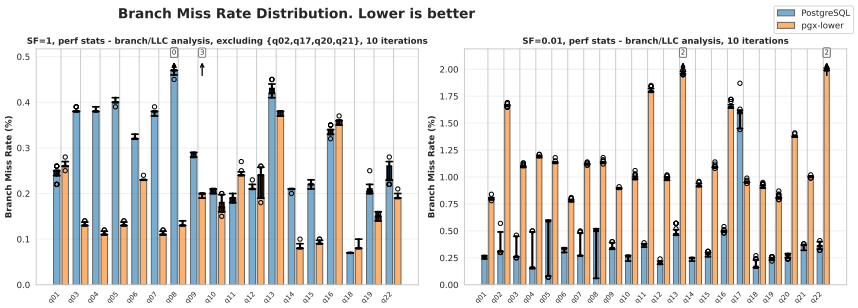


Figure 5.5: Branch miss rate

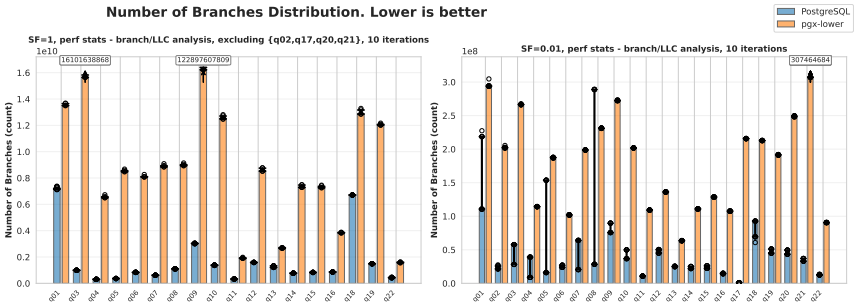


Figure 5.6: Number of branches

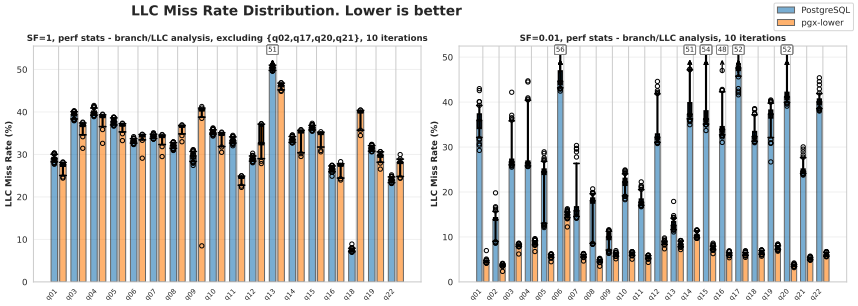


Figure 5.7: Last-level-cache miss plots

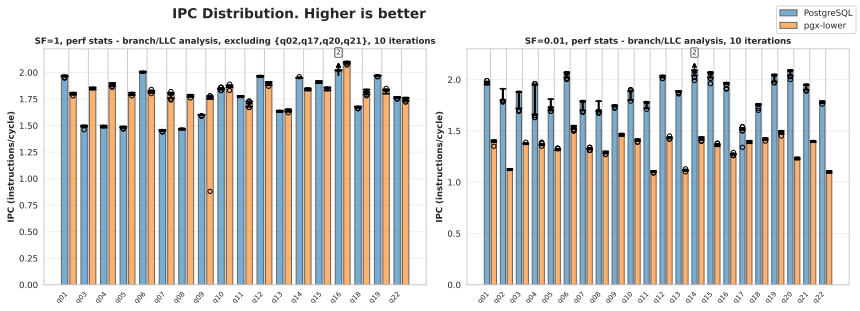


Figure 5.8: Instructions per (CPU) cycle plot

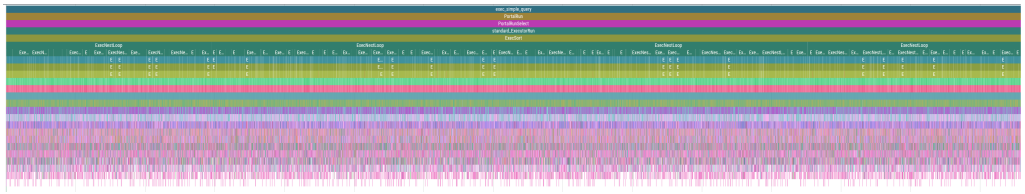


Figure 5.9: PostgreSQL TPC-H query 20 indexes enabled at SF = 0.16. Runtime: 15 minutes

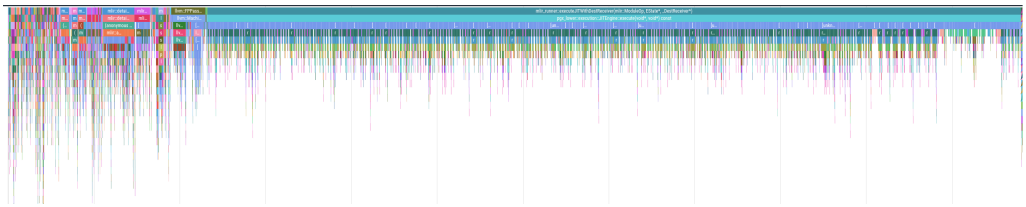


Figure 5.10: pgx-lower TPC-H query 20 indexes enabled at SF = 0.16. Runtime: 1.18 seconds

5.2 Discussion

To reiterate, the goal is to show that using the extension system is a viable approach to introduce compiled queries into battle-tested databases while maintaining their ACID properties. Previous studies have already found that this model has speed benefits.

In terms of latency for queries, there are mixed results. In figure 5.1 and figure 5.2 it is visible that if PostgreSQL has indexes disabled, at scale factor 0.01 the latency of queries 7 and 20 is multiple magnitudes longer than the other queries, and this happens

again at scale factor 0.16 even with the indexing enabled. In the differences plot, we can see that this clearly does not happen with pgx-lower, and the reason for this is visible in figure 5.9 and figure 5.10. PostgreSQL opted to do a nested join instead of relying on an index, while pgx-lower opted to build a hash join. Appendix A.1, A.2, A.3 show the initial query, PostgreSQL's plan tree, and pgx-lower's output after the optimisation pass. Within the pgx-lower's output, the joins have a "hash" annotation which means they go towards the hash join. Overall, this means that it is not an entirely fair comparison on these queries because pgx-lower gets to use a hash join while PostgreSQL is utilising a NestLoop_T, which would have been indexed with the indexing enabled.

An interesting detail in figure 5.10 is that the LLVM to bytecode compiler is very minimal. Out of the 1.18 second query, only 14.93 milliseconds is spent inside `llvm::SelectionDAGISel::runOnMach` which is the step that converts the LLVM IR into executable code. However, the overall compiler is 236 milliseconds, with much of it being inside the MLIR passes themselves. That means this MLIR code appears to be much heavier than the LLVM optimisations. One explanation for this is that the LLVM ORC JIT compiler does a minimal amount of work, and it happens in the background during execution.

For the RAM usage, figure 5.3 and figure 5.4 show that pgx-lower and PostgreSQL's peak memory usage is approximately aligned. This is ideal as it means the in-memory functionality of LingoDB was sufficiently moved to on-disk in pgx-lower. Figure 5.4 in particular shows the difference is only a bit over 3 megabytes at most, and the tests showed there is a mean difference across all these queries of 0.34 megabytes.

The branch prediction in figure 5.5 shows potential. At scale factor 1, pgx-lower's median branch miss rate was 0.16%, compared to Postgres's 0.28%. However, at a smaller scale factor with smaller queries this is inverted such that PostgreSQL had a median of 0.38% and pgx-lower had 1.09%. This means that the compiled JIT runtime has a better branch prediction rate, but potentially the code within the compilation stage, or before warm-up, has a worse rate. The difference could be attributed to the number of branches, visible in figure 5.6. PostgreSQL had a median of 28,430,465

branches, while pgx-lower had 195,299,021.50; 6.87 times more than PostgreSQL. This increase in the number of branches could mean that there are more low-hanging fruit.

For the last level cache miss rate, pgx-lower appears to perform better than PostgreSQL at smaller scale factors and appears similar to PostgreSQL at larger ones, which can be seen in figure 5.7. PostgreSQL has a median 31.20% miss rate at scale factor 0.01, while pgx-lower has a median of 6.19%. When increased to a scale factor of 1, PostgreSQL has a median of 33.16%, and pgx-lower has a median of 34.81%. An explanation for this could be that the LLVM compiler stage uses its cache much more effectively, but the actual JIT is approximately the same. This is expected with the current approach, but this would improve if the JIT stage is permitted to dynamically change the plan, such as how HyPer does.

5.2.1 Test Validity

Increasing the number of iterations to make the outputs more reliable seems to be successful, and the variation is not too large. Some queries, such as in scale factor 0.01 with indexes disabled, in figure 5.1, show the outliers do become extreme. On query 8, PostgreSQL had an outlier of 5008 milliseconds while the median was 12.42 milliseconds. However, the coefficient of variation was only 0.44% overall during the latency measurements in the scale factor 0.01. It's stable overall, but vital to do repeated tests to exclude these outliers from the system.

These variations will primarily be caused by PostgreSQL's optimiser containing genetic algorithms, as mentioned in section 2.5. It can cause plans to change significantly and makes them non-deterministic. While these tests were done on an isolated docker container in a Linux machine running minimal processes, system interrupts can also affect the results.

5.2.2 Future work

Since replacing PostgreSQL's execution engine with a JIT-focused one is quite generic, there are a number of directions future work can take. The system itself can be improved by fully implementing the plan nodes, and it could be optimised further. However, it's important to ensure the final product is useful, so other base databases, compilers, languages, and execution engines that could be integrated can be considered.

To progress this system further and make it a full extension, the plan tree nodes need to be fully implemented, and the query analyser can be improved. Only the minimum amount for TPC-H was implemented here, but pg_bench and isolationtester should also be used before suggesting this system to users. This includes implementing indexes, WINDOW functions, the other 22 missing plan nodes, and the missing execution nodes.

The other core work is optimising the system, both with existing research and clearer ways to use the PostgreSQL API. Namely: pre-compiling functions in PostgreSQL into LLVM, then inlining them instead of crossing the LLVM-C++ boundary; adaptive compilation/query planning; and Umbra's LeanStore-style buffering system. Most of these optimisations are only applicable inside an LLVM/MLIR system, though if WebAssembly is used instead, many of them can be skipped. More generally, parallelism can be improved, JIT compilation tuning, cross-platform support, subquery deduplication, and further optimisations.

In databasing and research broadly, it is vital to keep in mind whether the research is useful and impactful, as per Michael Stonebraker's concern that the field is merely polishing a round ball [Sto18]. For successful projects, the dollar-cost of a live system database is typically much smaller than the profitability of the project itself, so paying for higher throughput is commonly not a concern, and better gains for latency can be made by adding a custom caching mechanism to the service such as Redis. These complex queries are primarily in OLAP systems, and large-scale OLAP systems will usually move away from PostgreSQL and into a more scalable database such as ClickHouse, or Apache HIVE. For this reason, there may be better systems to develop this

architecture on.

PostgreSQL was chosen due to its large popularity and LingoDB was used because it reasonably matched PostgreSQL's interfaces while being open source. While PostgreSQL is reasonably good for this approach, and this can solve a real problem with it, there might be a better database. Most of these dedicated OLAP systems will already be using a JIT or vectorized approach, though.

During development, using LingoDB provided helpful constraints and made development easier since it is an established system. However, LingoDB's columnar, in-memory architecture required extensive modifications to suit its needs. It also contains a query optimisation engine, which is unnecessary because PostgreSQL already has a thorough optimisation system. It would be better to build the engine, or find a better suited base system. The ideal here would be Umbra based on its description, but it is closed source. Another potential approach is to take an established OLAP system's engine (such as ClickHouse), and route to that instead of PostgreSQL's depending on the analyser's rules.

MLIR was useful to give a strong set of dialect systems, but the main reason LingoDB used it was to give database optimisations clear layers. Furthermore, the LLVM/MLIR ecosystem targets ahead of time compilation, or longer-running JIT systems. While WebAssembly is appealing here because it targets short-lived processes, we would not be able to inline functions in the future. Either way, switching to a different compiler, or away from C++ into C or Rust is appealing.

This establishes a large set of different directions this research can take. The most appealing direction is attempting to take NoisePage or ClickHouse and inserting it into PostgreSQL as a drop-in engine replacement. This is a more complete ecosystem, and the primary work will be around the adapters to adjust queries. Furthermore, pg_duckdb has already done this, but it is a vectorised engine.

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.

Appendices

This appendix contains the execution plans for TPC-H Query 20, demonstrating the differences between PostgreSQL's traditional query execution plan and pgx-lower's MLIR-based compilation approach.

A.1 Query 20 SQL

```
select
        s_name,
        s_address
from
        supplier,
        nation
where
        s_suppkey in (
                select
                        ps_suppkey
                from
                        partsupp
                where
                        ps_partkey in (
                                select
                                        p_partkey
                                from
                                        part
                                where
                                        p_name like 'forest%'
                        )
                        and ps_availqty > (
                                select
                                        0.5 * sum(l_quantity)
                                from
                                        lineitem
                                where
                                        l_partkey = ps_partkey
                                        and l_suppkey = ps_suppkey
                                        and l_shipdate >= date '1994-01-01'
                                        and l_shipdate < date '1995-01-01'
                        )
        )
        and s_nationkey = n_nationkey
```

```
35          and n_name = 'CANADA'
36  order by
37          s_name
```

A.2 PostgreSQL Execution Plan

```
1  Sort  (cost=46847.99..46848.00 rows=1 width=52) (actual time=69.713..69.728 rows=1 loops=1)
2    Sort Key: supplier.s_name
3    Sort Method: quicksort  Memory: 25kB
4    ->  Nested Loop Semi Join  (cost=0.42..46847.98 rows=1 width=52) (actual time=68.025..69.569 rows=1 loops=1)
5          ->  Nested Loop  (cost=0.14..29.27 rows=1 width=56) (actual time=0.493..0.667 rows=3 loops=1)
6                Join Filter: (nation.n_nationkey = supplier.s_nationkey)
7                Rows Removed by Join Filter: 97
8                ->  Index Scan using supplier_pkey on supplier  (cost=0.14..15.64 rows=100 width=60) (actual time
                       =0.029..0.131 rows=100 loops=1)
9                ->  Materialize  (cost=0.00..12.13 rows=1 width=4) (actual time=0.004..0.005 rows=1 loops=100)
10                     ->  Seq Scan on nation  (cost=0.00..12.12 rows=1 width=4) (actual time=0.414..0.421 rows=1 loops=1)
11                           Filter: (n_name = 'CANADA'::bpchar)
12                           Rows Removed by Filter: 24
13         ->  Nested Loop  (cost=0.28..46818.70 rows=1 width=4) (actual time=22.958..22.958 rows=0 loops=3)
14               ->  Seq Scan on part  (cost=0.00..66.00 rows=20 width=4) (actual time=0.111..1.250 rows=16 loops=3)
15                     Filter: ((p_name)::text ~~ 'forest%'::text)
16                     Rows Removed by Filter: 1973
17               ->  Index Scan using partsupp_pkey on partsupp  (cost=0.28..2337.63 rows=1 width=8) (actual time=1.355..1.355
                       rows=0 loops=48)
18                     Index Cond: ((ps_partkey = part.p_partkey) AND (ps_suppkey = supplier.s_suppkey))
19                     Filter: ((ps_availqty)::numeric > (SubPlan 1))
20                     Rows Removed by Filter: 0
21                     SubPlan 1
22                       ->  Aggregate  (cost=2330.51..2330.52 rows=1 width=32) (actual time=31.616..31.617 rows=1 loops=2)
23                             ->  Seq Scan on lineitem  (cost=0.00..2330.50 rows=1 width=5) (actual time=25.850..31.589 rows
                                    =2 loops=2)
24                                   Filter: ((l_shipdate >= '1994-01-01'::date) AND (l_shipdate < '1995-01-01'::date) AND (
                                          l_partkey = partsupp.ps_partkey) AND (l_suppkey = partsupp.ps_suppkey))
25                                   Rows Removed by Filter: 60173
26 Planning Time: 16.634 ms
27 Execution Time: 70.580 ms
```

A.3 pgx-lower MLIR Execution Plan

```
1  // MLIR Module Debug Dump: Phase 3a AFTER: RelAlg -> Optimised RelAlg
2  // Generated: 2025-11-22 23:21:32
3  // Total Operations: 120
4  // Module Valid: YES
5
6  module {
7    func.func @main() -> !dsa.table {
8      %true = arith.constant true
9      %0 = relalg.basetable  {column_order = ["l_orderkey", "l_partkey", "l_suppkey", "l_linenumber", "l_quantity", "
           l_extendedprice", "l_discount", "l_tax", "l_returnflag", "l_linestatus", "l_shipdate", "l_commitdate", "
           l_receiptdate", "l_shipinstruct", "l_shipmode", "l_comment"], rows = 0.000000e+00 : f64, table_identifier = "
           lineitem|oid:16425"} columns: {l_comment => @lineitem::@l_comment({type = !db.string}), l_commitdate =>
           @lineitem::@l_commitdate({type = !db.date<day>}), l_discount => @lineitem::@l_discount({type = !db.decimal<12,
           2>}), l_extendedprice => @lineitem::@l_extendedprice({type = !db.decimal<12, 2>}), l_linenumber => @lineitem::
           @l_linenumber({type = i32}), l_linestatus => @lineitem::@l_linestatus({type = !db.string}), l_orderkey =>
           @lineitem::@l_orderkey({type = i32}), l_partkey => @lineitem::@l_partkey({type = i32}), l_quantity => @lineitem
           ::@l_quantity({type = !db.decimal<12, 2>}), l_receiptdate => @lineitem::@l_receiptdate({type = !db.date<day>}),
           l_returnflag => @lineitem::@l_returnflag({type = !db.string}), l_shipdate => @lineitem::@l_shipdate({type = !db.
           date<day>}), l_shipinstruct => @lineitem::@l_shipinstruct({type = !db.string}), l_shipmode => @lineitem::
           @l_shipmode({type = !db.string}), l_suppkey => @lineitem::@l_suppkey({type = i32}), l_tax => @lineitem::@l_tax({
           type = !db.decimal<12, 2>})}
10     %1 = relalg.selection %0 (%arg0: !relalg.tuple){
11       %39 = relalg.getcol %arg0 @lineitem::@l_shipdate : !db.date<day>
12       %40 = db.constant(-2191 : i32) : !db.date<day>
13       %41 = db.constant(-1826 : i32) : !db.date<day>
14       %42 = db.between %39 : !db.date<day> between %40 : !db.date<day>, %41 : !db.date<day>, lowerInclusive : true,
              upperInclusive : false
15       relalg.return %42 : i1
16     } attributes {cost = 1.000000e+00 : f64, rows = 1.000000e+00 : f64}
17     %2 = relalg.basetable  {column_order = ["ps_partkey", "ps_suppkey", "ps_availqty", "ps_supplycost", "ps_comment"], rows
            = 0.000000e+00 : f64, table_identifier = "partsupp|oid:16410"} columns: {ps_availqty => @partsupp::@ps_availqty
           ({type = i32}), ps_comment => @partsupp::@ps_comment({type = !db.string}), ps_partkey => @partsupp::@ps_partkey
           ({type = i32}), ps_suppkey => @partsupp::@ps_suppkey({type = i32}), ps_supplycost => @partsupp::@ps_supplycost({
           type = !db.decimal<12, 2>})}
```

```
18    %3 = relalg.basetable  {column_order = ["n_nationkey", "n_name", "n_regionkey", "n_comment"], rows = 0.000000e+00 : f64
          , table_identifier = "nation|oid:16395"} columns: {n_comment => @nation::@n_comment({type = !db.string}), n_name
           => @nation::@n_name({type = !db.string}), n_nationkey => @nation::@n_nationkey({type = i32}), n_regionkey =>
          @nation::@n_regionkey({type = i32})}
19    %4 = relalg.selection %3 (%arg0: !relalg.tuple){
20      %39 = relalg.getcol %arg0 @nation::@n_name : !db.string
21      %40 = db.constant("CANADA                    ") : !db.string
22      %41 = db.compare eq %39 : !db.string, %40 : !db.string
23      relalg.return %41 : i1
24    } attributes {cost = 1.000000e-01 : f64, rows = 1.000000e-01 : f64}
25    %5 = relalg.basetable  {column_order = ["s_suppkey", "s_name", "s_address", "s_nationkey", "s_phone", "s_acctbal", "
          s_comment"], rows = 0.000000e+00 : f64, table_identifier = "supplier|oid:16405"} columns: {s_acctbal =>
          @supplier::@s_acctbal({type = !db.decimal<12, 2>}), s_address => @supplier::@s_address({type = !db.string}),
          s_comment => @supplier::@s_comment({type = !db.string}), s_name => @supplier::@s_name({type = !db.string}),
          s_nationkey => @supplier::@s_nationkey({type = i32}), s_phone => @supplier::@s_phone({type = !db.string}),
          s_suppkey => @supplier::@s_suppkey({type = i32})}
26    %6 = relalg.join %4, %5 (%arg0: !relalg.tuple){
27      %39 = relalg.getcol %arg0 @nation::@n_nationkey : i32
28      %40 = relalg.getcol %arg0 @supplier::@s_nationkey : i32
29      %41 = db.compare eq %39 : i32, %40 : i32
30      relalg.return %41 : i1
31    } attributes {cost = 1.110000e+00 : f64, impl = "hash", rows = 0.010000000000000002 : f64}
32    %7 = relalg.projection all [@supplier::@s_name,@supplier::@s_address,@supplier::@s_suppkey] %6
33    %8 = relalg.tmp %7 [@supplier::@s_address,@supplier::@s_name,@supplier::@s_suppkey]
34    %9 = relalg.projection distinct [@supplier::@s_suppkey] %8
35    %10 = relalg.tmp %9 [@supplier::@s_suppkey]
36    %11 = relalg.join %2, %10 (%arg0: !relalg.tuple){
37      %39 = relalg.getcol %arg0 @partsupp::@ps_suppkey : i32
38      %40 = db.as_nullable %39 : i32 -> <i32>
39      %41 = relalg.getcol %arg0 @supplier::@s_suppkey : !db.nullable<i32>
40      %42 = db.compare eq %40 : !db.nullable<i32>, %41 : !db.nullable<i32>
41      %43 = db.derive_truth %42 : !db.nullable<i1>
42      relalg.return %43 : i1
43    } attributes {cost = 2.100000e+00 : f64, impl = "hash", rows = 1.000000e-01 : f64}
44    %12 = relalg.basetable  {column_order = ["p_partkey", "p_name", "p_mfgr", "p_brand", "p_type", "p_size", "p_container",
           "p_retailprice", "p_comment"], rows = 0.000000e+00 : f64, table_identifier = "part|oid:16400"} columns: {
          p_brand => @part::@p_brand({type = !db.string}), p_comment => @part::@p_comment({type = !db.string}),
          p_container => @part::@p_container({type = !db.string}), p_mfgr => @part::@p_mfgr({type = !db.string}), p_name
          => @part::@p_name({type = !db.string}), p_partkey => @part::@p_partkey({type = i32}), p_retailprice => @part::
          @p_retailprice({type = !db.decimal<12, 2>}), p_size => @part::@p_size({type = i32}), p_type => @part::@p_type({
          type = !db.string})}
45    %13 = relalg.selection %12 (%arg0: !relalg.tuple){
46      %39 = relalg.getcol %arg0 @part::@p_name : !db.string
47      %40 = db.constant("forest%") : !db.string
48      %41 = db.runtime_call "Like"(%39, %40) : (!db.string, !db.string) -> i1
49      relalg.return %41 : i1
50    } attributes {cost = 1.000000e-01 : f64, rows = 1.000000e-01 : f64}
51    %14 = relalg.tmp %13 [@part::@p_partkey]
52    %15 = relalg.projection distinct [@part::@p_partkey] %14
53    %16 = relalg.tmp %15 [@part::@p_partkey]
54    %17 = relalg.join %11, %16 (%arg0: !relalg.tuple){
55      %39 = relalg.getcol %arg0 @partsupp::@ps_partkey : i32
56      %40 = db.as_nullable %39 : i32 -> <i32>
57      %41 = relalg.getcol %arg0 @part::@p_partkey : !db.nullable<i32>
58      %42 = db.compare eq %40 : !db.nullable<i32>, %41 : !db.nullable<i32>
59      %43 = db.derive_truth %42 : !db.nullable<i1>
60      relalg.return %43 : i1
61    } attributes {cost = 3.110000e+00 : f64, impl = "hash", rows = 0.010000000000000002 : f64}
62    %18 = relalg.tmp %17 [@partsupp::@ps_availqty,@partsupp::@ps_suppkey,@supplier::@s_suppkey,@part::@p_partkey,@partsupp
          ::@ps_partkey]
63    %19 = relalg.projection distinct [@partsupp::@ps_suppkey,@partsupp::@ps_partkey] %18
64    %20 = relalg.join %1, %19 (%arg0: !relalg.tuple){
65      %39 = relalg.getcol %arg0 @lineitem::@l_suppkey : i32
66      %40 = relalg.getcol %arg0 @partsupp::@ps_suppkey : i32
67      %41 = db.compare eq %39 : i32, %40 : i32
68      %42 = relalg.getcol %arg0 @lineitem::@l_partkey : i32
69      %43 = relalg.getcol %arg0 @partsupp::@ps_partkey : i32
70      %44 = db.compare eq %42 : i32, %43 : i32
71      %45 = db.and %44, %41 : i1, i1
72      relalg.return %45 : i1
73    } attributes {cost = 2.010000e+00 : f64, impl = "hash", rows = 0.010000000000000002 : f64}
74    %21 = relalg.crossproduct %20, %10
75    %22 = relalg.crossproduct %21, %16
76    %23 = relalg.aggregation %22 [@partsupp::@ps_suppkey,@partsupp::@ps_partkey,@part::@p_partkey,@supplier::@s_suppkey]
          computes : [@aggr0::@agg_0({type = !db.nullable<!db.decimal<32, 6>>})] (%arg0: !relalg.tuplestream,%arg1: !
          relalg.tuple){
77      %39 = relalg.aggrfn sum @lineitem::@l_quantity %arg0 : !db.nullable<!db.decimal<32, 6>>
78      relalg.return %39 : !db.nullable<!db.decimal<32, 6>>
79    }
80    %24 = relalg.map %23 computes : [@postmap::@postproc_1({type = !db.nullable<!db.decimal<32, 6>>})] (%arg0: !relalg.
          tuple){
81      %39 = db.constant("0.5") : !db.decimal<32, 6>
82      %40 = db.as_nullable %39 : !db.decimal<32, 6> -> <!db.decimal<32, 6>>
83      %41 = relalg.getcol %arg0 @aggr0::@agg_0 : !db.nullable<!db.decimal<32, 6>>
84      %42 = db.mul %40 : !db.nullable<!db.decimal<32, 6>>, %41 : !db.nullable<!db.decimal<32, 6>>
85      relalg.return %42 : !db.nullable<!db.decimal<32, 6>>
86    }
87    %25 = relalg.renaming %24 renamed : [@renaming::@renamed0({type = i32})=[@partsupp::@ps_suppkey],@renaming::@renamed1({
          type = i32})=[@partsupp::@ps_partkey]]
```

```
    %26 = relalg.renaming %25 renamed : [@renaming1::@renamed0({type = i32})=[@part::@p_partkey]]
    %27 = relalg.renaming %26 renamed : [@renaming3::@renamed0({type = i32})=[@supplier::@s_suppkey]]
    %28 = relalg.singlejoin %18, %27 (%arg0: !relalg.tuple){
      %39 = relalg.getcol %arg0 @partsupp::@ps_suppkey : i32
      %40 = relalg.getcol %arg0 @renaming::@renamed0 : i32
      %41 = db.compare eq %39 : i32, %40 : i32
      %42 = relalg.getcol %arg0 @partsupp::@ps_partkey : i32
      %43 = relalg.getcol %arg0 @renaming::@renamed1 : i32
      %44 = db.compare eq %42 : i32, %43 : i32
      %45 = relalg.getcol %arg0 @part::@p_partkey : i32
      %46 = relalg.getcol %arg0 @renaming1::@renamed0 : i32
      %47 = db.compare eq %45 : i32, %46 : i32
      %48 = relalg.getcol %arg0 @supplier::@s_suppkey : i32
      %49 = relalg.getcol %arg0 @renaming3::@renamed0 : i32
      %50 = db.compare eq %48 : i32, %49 : i32
      %51 = db.and %50, %44, %41, %47 : i1, i1, i1, i1
      relalg.return %51 : i1
    }  mapping: {@singlejoin::@sjattr({type = !db.nullable<!db.decimal<32, 6>>})=[@postmap::@postproc_1]} attributes {cost
         = 3.000000e+00 : f64, impl = "hash", rows = 1.000000e+00 : f64}
    %29 = relalg.selection %28 (%arg0: !relalg.tuple){
      %39 = relalg.getcol %arg0 @partsupp::@ps_availqty : i32
      %40 = db.cast %39 : i32 -> !db.decimal<38, 0>
      %41 = db.cast %40 : !db.decimal<38, 0> -> !db.decimal<32, 6>
      %42 = db.as_nullable %41 : !db.decimal<32, 6> -> <!db.decimal<32, 6>>
      %43 = relalg.getcol %arg0 @singlejoin::@sjattr : !db.nullable<!db.decimal<32, 6>>
      %44 = db.compare gt %42 : !db.nullable<!db.decimal<32, 6>>, %43 : !db.nullable<!db.decimal<32, 6>>
      %45 = db.derive_truth %44 : !db.nullable<i1>
      relalg.return %45 : i1
    } attributes {cost = 3.000000e+00 : f64, rows = 1.000000e+00 : f64}
    %30 = relalg.renaming %29 renamed : [@renaming2::@renamed0({type = i32})=[@part::@p_partkey]]
    %31 = relalg.join %14, %30 (%arg0: !relalg.tuple){
      %39 = relalg.getcol %arg0 @part::@p_partkey : i32
      %40 = relalg.getcol %arg0 @renaming2::@renamed0 : i32
      %41 = db.compare eq %39 : i32, %40 : i32
      %42 = db.and %true, %41 : i1, i1
      relalg.return %42 : i1
    } attributes {cost = 2.100000e+00 : f64, impl = "hash", rows = 1.000000e-01 : f64}
    %32 = relalg.projection all [@partsupp::@ps_suppkey,@supplier::@s_suppkey] %31
    %33 = relalg.map %32 computes : [@map::@tmp_attr0({type = i32})] (%arg0: !relalg.tuple){
      %39 = db.constant(1 : i32) : i32
      relalg.return %39 : i32
    }
    %34 = relalg.renaming %33 renamed : [@renaming4::@renamed0({type = i32})=[@supplier::@s_suppkey]]
    %35 = relalg.semijoin %8, %34 (%arg0: !relalg.tuple){
      %39 = relalg.getcol %arg0 @supplier::@s_suppkey : i32
      %40 = relalg.getcol %arg0 @renaming4::@renamed0 : i32
      %41 = db.compare eq %39 : i32, %40 : i32
      relalg.return %41 : i1
    } attributes {cost = 2.100000e+00 : f64, impl = "hash", rows = 1.000000e-01 : f64}
    %36 = relalg.projection all [@supplier::@s_name,@supplier::@s_address] %35
    %37 = relalg.sort %36 [(@supplier::@s_name,asc)]
    %38 = relalg.materialize %37 [@supplier::@s_name,@supplier::@s_address] => ["s_name", "s_address"] : !dsa.table
    return %38 : !dsa.table
  }
}
```

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.

[Apa24] Apache Software Foundation. Apache age: Graph extension for postgresql, 2024. Accessed: 2025-11-19.

[APBC13] Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. Linkbench: a database benchmark based on the facebook social graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1185–1196, New York, NY, USA, 2013. ACM.

[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.

[BNE14] Peter Boncz, Thomas Neumann, and Orri Erling. *TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark*, volume 8391 of *Lecture Notes in Computer Science*, pages 61–76. Springer International Publishing, 2014.

[Bus22] Business Wire. Timescale valuation rockets to over $1b with $110m round, marking the explosive rise of time-series data, February 2022. Accessed: 2025-11-19.

[BZN05] Peter Boncz, Marcin Zukowski, and Niels Nes. Monetdb/x100: Hyper-pipelining query execution. In *2nd Biennial Conference on Innovative Data Systems Research (CIDR 2005)*, Asilomar, CA, USA, January 2005.

[CAB+81] Donald D Chamberlin, Morton M Astrahan, Michael W Blasgen, James N Gray, W Frank King, Bruce G Lindsay, Raymond Lorie, James W Mehl, Thomas G Price, Franco Putzolu, et al. A history and evaluation of system r. *Communications of the ACM*, 24(10):632–646, 1981.

[CEP+21] Umur Cubukcu, Taner Erdogan, Rishab Patel, Amulya Dey, Jake Collins, et al. Citus: Distributed postgresql for data-intensive applications. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, pages 2461–2476, Virtual Event, China, 2021. ACM.

[Cli24] ClickHouse. Clickbench — a benchmark for analytical dbms, 2024. Accessed: 2024-11-17.

[CRCG+19] Jesús Camacho-Rodríguez, Ashutosh Chauhan, Alan Gates, Eugene Koifman, et al. 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.

[DCL25] Quentin Ducasse, Pascal Cotret, and Loïc Lagadec. War on jits: Software-based attacks and hybrid defenses for jit compilers – a comprehensive survey. *ACM Computing Surveys*, 57(9):1–36, May 2025.

[Duc24] DuckDB Foundation. pg_duckdb: Duckdb-powered postgres for high performance apps & analytics, 2024. Open-source PostgreSQL extension. Accessed: 2025-11-23.

[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.

[Fre16] Andres Freund. Discussion on postgresql performance optimizations. `https://www.postgresql.org/message-id/flat/20161206034955.bh33paeralxbtluv@alap3.anarazel.de`, 2016. Accessed: 2024-11-17.

[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.

[Fre18] Andres Freund. Postgresql 11 beta 2 released, 2018. Accessed: 2024-11-17.

[FRMK19] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire, and Dejan Kostić. Make the most out of last level cache in intel processors. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 1–17, Dresden, Germany, 2019. ACM.

[Ham18] Clemens Hammacher. Liftoff: a new baseline compiler for webassembly in v8. *V8 JavaScript engine*, 2018.

[HD23a] Immanuel Haffner and Jens Dittrich. mutable: A modern dbms for research and fast prototyping. In *CIDR*, 2023.

[HD23b] Immanuel Haffner and Jens Dittrich. A simplified architecture for fast, adaptive compilation and execution of sql queries. In *EDBT*, pages 1–13, 2023.

[HRS+17] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, et al. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '17, pages 185–200, Barcelona, Spain, 2017. ACM.

[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.

[Hyd24] Hydra Database Inc. Hydra columnar: Postgres-native columnar storage extension, 2024. Open-source PostgreSQL extension. Accessed: 2025-11-23.

[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.

[ISP17] ISPRAS. Dynamic compilation of sql queries in postgresql using llvm jit. In *PGCon 2017*, Ottawa, Canada, May 2017. Accessed: 2025-11-19.

[JG23] Michael Jungmair and Jana Giceva. Declarative sub-operators for universal data processing. *Proceedings of the VLDB Endowment*, 16(11):3461–3474, 2023.

[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.

[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.

[KLK+18] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, et al. 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.

[KN20] Timo Kersten and Thomas Neumann. On another level: how to debug compiling query engines. In *Proceedings of the workshop on Testing Database Systems*, pages 1–6, 2020.

[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, et al. Mlir: A compiler infrastructure for the end of moore's law. *arXiv*, 2020.

[LGM+15] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, et al. How good are query optimizers, really? *Proceedings of the VLDB Endowment (PVLDB)*, 9(3), 2015.

[Linnd] Linux man-pages project. *perf(1) - Linux Manual Page*, n.d. Accessed: 2024-11-20.

[LLV25a] LLVM Project. Llvm's analysis and transform passes, 2025. Accessed: 2025-11-23.

[LLV25b] LLVM Project. Orc design and implementation. `https://llvm.org/docs/ORCv2.html`, 2025. Accessed: 2025-11-18.

[LRG+17] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, et al. Query optimization through the looking glass, and what we found running the join order benchmark. *VLDB Journal*, 2017.

[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.

[MZJ+21] Lin Ma, William Zhang, Jie Jiao, Wuwen Wang, Matthew Butrovich, Wan Shen Lim, Prashanth Menon, and Andrew Pavlo. Mb2: Decomposed behavior modeling for self-driving database management systems. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, pages 1248–1261, Virtual Event, China, 2021. ACM.

[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.

[Ove24] Stack Overflow. Technology: Databases, 2024. Stack Overflow Developer Survey 2024.

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

[Pos24] PostgreSQL Global Development Group. *Just-in-Time Compilation (JIT) - PostgreSQL Documentation*, 2024. Accessed: 2024-11-17.

[Pos25a] PostgreSQL Global Development Group. Explain — show the execution plan of a statement, 2025.

[Pos25b] PostgreSQL Global Development Group. Memory management, 2025. Server Programming Interface.

[Pos25c] PostgreSQL Global Development Group. *pgbench*. The PostgreSQL Global Development Group, 2025. PostgreSQL Documentation.

[Pos25d] PostgreSQL Global Development Group. The query tree, 2025.

[RM19] Mark Raasveldt and Hannes Mühleisen. Duckdb: an embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, pages 1981–1984, Amsterdam, Netherlands, 2019. ACM.

[RPML06] Jun Rao, H. Pirahesh, C. Mohan, and G. Lohman. Compiled query execution engine using jvm, 2006.

[Sam24] Samsung Electronics. *Samsung NVMe SSD 990 PRO Datasheet*, 2024. Rev. 1.0, Accessed: 2025-11-18.

[Sea07] Seagate Technology LLC. *Barracuda 7200.12 Serial ATA Product Manual*, May 2007. Rev. H, Accessed: 2025-11-18.

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

[She17] Arseny Sher. [gsoc] push-based query executor discussion. PostgreSQL Mailing List, March 2017. Accessed: 2024-11-20.

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

[SN22] Moritz Sichert and Thomas Neumann. User-defined operators: Efficiently integrating custom algorithms into modern databases. *Proceedings of the VLDB Endowment*, 15(5):1119–1131, 2022.

[SSY+24] Robert Schulze, Tom Schreiber, Ilya Yatsishin, Ryadh Dahimene, and Alexey Milovidov. Clickhouse - lightning fast analytics for everyone. *Proceedings of the VLDB Endowment*, 17(12):3731–3744, 2024.

[Sta25] Statistics LibreTexts. Coefficient of variation, 2025. Accessed: 2025-11-23.

[Sto18] Michael Stonebraker. My top ten fears about the dbms field. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 24–28. IEEE, 2018.

[SZC+24] Haoze Song, Wenchao Zhou, Heming Cui, Xiang Peng, and Feifei Li. A survey on hybrid transactional and analytical processing. *The VLDB Journal*, 33(5):1485–1515, June 2024.

[Tab18] Tableau. Welcome hyper team to the tableau community, 2018. Accessed: 2024-11-17.

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

[Tim24a] Timescale Inc. Timescaledb: Time-series database for high-performance real-time analytics, 2024. Accessed: 2025-11-19.

[Tim24b] Timescale/Tiger Data. Year of the tiger: $110 million to build the future of data for developers worldwide, February 2024. Accessed: 2025-11-19.

[Ute97] Martin Utesch. Genetic query optimization in database systems. Technical report, Institute of Automatic Control, University of Mining and Technology, Freiberg, Germany, February 1997.

[Xen21] Xenatisch. Cascade of doom, jit, and how a postgres update led to 70% failure on a critical national service, 2021. Accessed: 2024-11-17.

[Zuk09] Marcin Zukowski. *Balancing Vectorized Query Execution with Bandwidth- Optimized Storage*. 2009.

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