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

With the advancement of hardware research, many modern battle-tested databases underutilise their caches and disk read speeds due to them being designed when I/O was the bottleneck. This thesis proposes pgx-lower, which demonstrates how just-in-time compilation from a research system can be retrofitted into an established database using its extension system. By integrating LingoDB's MLIR-based JIT compiler into PostgreSQL, a columnar in-memory compile was adapted into a disk-oriented architecture while maintaining ACID properties and MVCC support. The evaluations on TPC-H showed improved branch prediction and cache efficiency compared to the original executor. More importantly, pgx-lower demonstrates that a production database can adopt modern compiler techniques as third-party extensions, providing a practical pathway for database research productionisation.

Abbreviations

ACID Atomicity, consistency, isolation, durability

API Application Programming Interface

AQP Adaptive Query Processing

AST Abstract Syntax Tree

CPU Central Processing Unit

CTE Common Table Expression

DB Database

DSA Data Structures and Algorithms

EXP Expression (expressions inside queries)

GDB GNU Debugger

GVN Global Value Numbering

IR Intermediate Representation

IPC Instructions Per CPU Cycle

JIT Just-in-time (compiler)

JVM Java Virtual Machine

LICM Loop Invariant Code Motion

LLC Last Level Cache

LLVM Low-Level Virtual Machine

MLIR Multi-Level Intermediate Representation

MVCC Multi-Version Concurrency Control

OLAP Online Analytical Processing

OLTP Online Transaction Processing

ORC On-Request-Compilation

psql PostgreSQL

QEP Query Execution Plan

RA Relational Algebra

RAM Random Access Memory

SIMD Single Instruction, Multiple Data

SPI Server Programming Interface

SROA Scalar Replacement of Aggregates

SQL Structured Query Language

SSA Static Single Assignment

SSD Solid State Drive

TPC-H Transaction Processing Performance Council: Decision Support Benchmark H

V8 JavaScript Engine (Google Chrome)

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 [**?**].

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.

Research projects typically create standalone databases. However, such approaches make distribution harder, requiring projects to implement all supporting infrastructure themselves. For serious projects, supporting infrastructure requires implementing ACID, MVCC, query plan optimisation, and more. By using an established database, we can address this entirely.

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 features that can be used within PostgreSQL to assist with this research. One concern, however, is the additional complexities in implementation and testing.

Chapter 2 covers fundamental concepts and project definition. Chapter 3 provides a literature survey, followed by the solution in Chapter 4. Conclusions are drawn in Chapter 6.

Chapter 2

Background

2.1 Database Background

Most databases follow the structure shown in Figure 2.1. Database systems parse *Structured Query Language* (SQL) into *relational algebra* (RA), optimize it, execute it, and materialize the results into a table [**?**].

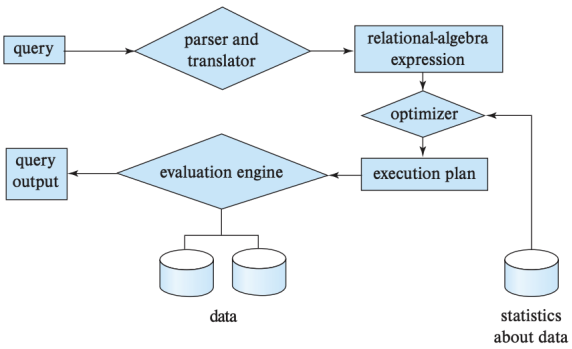


Figure 2.1: Database Structure.
[**?**]

Non-compiler databases use a *volcano operator model tree*, such as Figure 2.2 [**?**]. A `produce()` function at the root node calls its children's `produce()`, until it calls a leaf node, which calls `consume()` on itself, then that calls its parent's `consume()` func-

tion. In other words, a post-order traversal through the tree where tuples are dragged upwards.

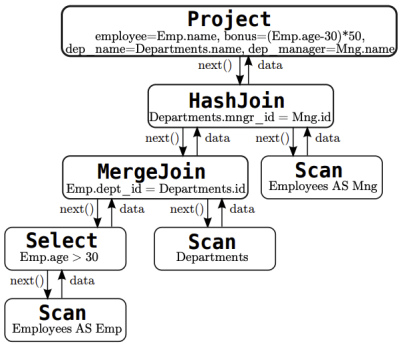


Figure 2.2: Volcano operator model tree.
[**?**]

Classical models suffer a fundamental issue: heavy under-utilisation of hardware [**?**]. If only a single tuple is pulled, CPU caches are barely used. An i5-570, a popular CPU in 2010 had an 8 MB L3 cache, but in 2024 an i5-14600K has a 24 MB L3 cache [**?**], [**?**]. 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 [**?**, **?**]. Such dramatic increases could mean the algorithms can fundamentally change.

These observations led to the *vectorized* and *compiled execution models*. The vectorized model pulls multiple tuples 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 [**?**]. However, this can cause deep copy operations to be required, or more disk spillage than necessary [**?**]. For instance, if a sort or a join allocates new space that is too much for the cache, the handling can become poor. Section 2.2 and Chapter 3 explore compilation approaches.

Relational databases prioritise ACID requirements - Atomicity, Consistency, Isolation and Durability [**?**]. A critical requirement in database systems, ACID compliance is usually one of the main reasons people choose relational databases. Atomicity means transactions are a single unit of work, while 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. [**?**]

On-disk databases often assume CPU operation costs are constant or negligible [**?**]. Such assumptions stem from the era in which these systems were developed, when disks were much slower and caches were much smaller. Previous analysis disproved such assumptions for PostgreSQL, finding that CPU time constitutes substantial overhead: between 34.87% and 76.56% with an average of 49.32% across tested queries.

Most of the recent databasing research has focused on the optimiser, which is visible in Figure 2.1. Common optimization techniques include reordering join statements where the left and right sides are flipped, and predicate push down, where a conditional/filter on a node is moved onto a lower node [**?**]. Additionally, extracting common subexpressions prevents recomputation, while constant folding evaluates constant operations inside the optimiser rather than at runtime. Despite these advances, query optimisers frequently produce suboptimal plans [**?**, **?**]. 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 [**?**]. A thesis could be written just to summarise the list of optimisations.

Within traditional and volcano databases, the cache is managed through buffer techniques. Fixed-size pages (typically eight kilobytes) are read and loaded into a *buffer pool* object which holds them in RAM. Memory placement in L1/L2/L3 and RAM caches depends on access patterns, managed by the operating system or environment [**?**]. Buffer pools employ different caching strategies (last-recently-used, most-recently-used, etc.) based on the situation, decisions often made by the optimiser [**?**]. Cache effectiveness is measured by *last level cache hit rate* (LLC), which represents how many instructions were resolved inside the CPU cache [**?**].

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 [**?**]. OLAP systems can be highly distributed, such as Apache Hive, which allows compute to be distributed across the system [**?**]. PostgreSQL implements a hybrid architecture supporting both OLTP and OLAP operations [**?**]. Debate continues about whether such hybrid designs remain useful, as load pressure on user-serving databases commonly causes reliability issues [**?**].

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 [**?**]. Advanced compilers can run the primary program while background threads improve code optimisation and swap in the optimized version when ready [**?**]. Such multi-stage approaches provide faster initial compilation and faster development cycles.

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 [**?**]. 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 $2.6\times$ faster [**?**]. Azul's JIT compiler measurements show that speculative optimisations can lead up to 50% performance gains [**?**].

Evaluating good branch prediction is difficult. A reasonable baseline is that a 1% misprediction rate is too high and should be optimised in low latency environments [**?**]. Such baselines lack formality but rest on empirical knowledge [**?**]. Depending on the CPU, a branch mispredict can cost between 10 and 35 CPU cycles, with a safe range being 14-25 cycles. With 1 branch every 10 instructions and a 5% misprediction rate,

a 20 cycle penalty per misprediction translates to approximately 10% of runtime spent resolving mispredictions [**?**].

In the context of databases, compilers fall into two categories: those that compile only *expressions* (typically called EXP), and those that compile the entire *Query Execution Plan* (QEP) [**?**]. Within PostgreSQL itself, they have EXP support using `llvm-jit`. Chapter 3 examines a variety of research databases.

2.3 LLVM and MLIR

The LLVM Project is a compiler infrastructure that eliminates the need to re-implement common compiler optimisations, making it easier to build compilers [**?**]. *Multi-Level Intermediate Representation* (MLIR) is another, newer toolkit that is tightly coupled with the LLVM project [**?**]. It provides a framework to define custom dialects and progressively lower them to machine code. Developers can define high-level dialects that others can target and build upon.

LLVM defines a language-independent *intermediate representation* (IR) based on *Static Single Assignment* (SSA) form, while MLIR extends this concept [**?**]. 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 [**?**]. 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 [**?**]. 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 [**?**]. 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 [**?**]. The JVM has also been used for compiled query execution engines [**?**]. 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) [**?**].

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) [**?**].

PostgreSQL provides the `EXPLAIN` command to inspect query execution plans, which is essential for understanding and optimizing query performance [**?**]. 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 difficult because of the variety of workloads. Many systems create their own benchmarking libraries, such as `pg_bench` by PostgreSQL [**?**] or LinkBench [**?**] by Facebook, but in academics the more common benchmarks are from the Transaction Processing Council, which is a group that defines benchmarks [**?**]. 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. TPC-H is the most common in research, where the H informally means "hybrid". It has a mix of analytical and transactional elements inside it [**?**].

When evaluating benchmark results, the *coefficient of variation* (CV) is used. This is a standardized measure of dispersion that expresses the standard deviation as a percentage of the mean [**?**]. 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 datasets with different units or scales, providing a unit-free measure for relative comparison of measurement precision.

Chapter 3

Related Work

We summarise relevant works in the compiled query space and their architectures. Compiled query engines originally dominated the database industry [**?**], but the volcano model subsequently took precedence due to its simpler implementation and minimal performance cost at the time. However, modern analytical engines are revisiting compilation approaches [**?**].

PostgreSQL's extension system is covered in Section 3.2, followed by System R in Section 3.3 as a classical example. HyPer and Umbra (Sections 3.4 and 3.5) re-introduced compilation in modern databases, while Mutable (Section 3.6) and LingoDB (Section 3.7) provide research examples. PostgreSQL's JIT compilation and prior attempts conclude the survey.

3.1 OLAP Systems

Compiled query engines primarily benefit OLAP workloads, since OLTP workloads typically involve simpler retrieval queries [**?**]. At scale, Apache Hive is commonly used, but it is a data warehouse system rather than a database, storing data in Hadoop's distributed file system, which is closer to flat storage [**?**]. Common OLAP databases include MonetDB, SnowFlake, ClickHouse, RedShift, and Vectorwise [**?**]. 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) [**?**]. The Carnegie Mellon Database group created NoisePage, a columnar, in-memory system with full query expression compilation (QEP), with a single node. They targeted ML-driven self optimisation in their research, but the project archived in February 21, 2023 [**?**]. DuckDB is marketed as the SQLite for analytical loads with in-memory disk spillage, or like a more sophisticated Pandas DataFrame [**?**]. 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 the most popular database, with 51.9% of developers in a Stack Overflow survey reporting extensive use in 2024 [**?**]. In the context of compiled queries, this means PostgreSQL cannot be treated as a research prototype. Direct modifications require extensive testing, and such changes face casual code review via pull requests rather than formal peer review.

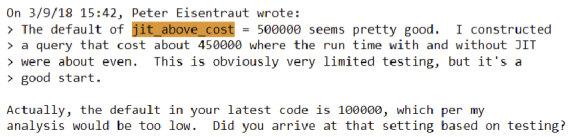
Building extensions for PostgreSQL and making entire companies around these extensions is well-established. Three such examples are Citus [**?**], TimescaleDB [**?**], and Apache AGE [**?**]. Citus aims to add more horizontal scaling through sharding, TimescaleDB, now rebranded as TigerData, transforms the engine into a time series database, and Apache AGE turns it into a graph database. All have thousands of GitHub stars, with TimescaleDB especially boasting over 500 paying customers in 2022 and claiming 20× revenue growth by 2024 [**?**] [**?**]. The extension model has proven robust and well-travelled.

There have also been several extensions that attempt to make PostgreSQL more suited

to OLAP workloads, with the most relevant one being pg_duckdb [**?**]. pg_duckdb replaces PostgreSQL's engine with DuckDB, enabling vectorised execution with reasonable popularity at roughly 2700 GitHub stars. Hydra is also worth mentioning, but this is closer to TimescaleDB [**?**] 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 [**?**, **?**]. Developers expressed doubts about adding full query compilation support, with concerns that rearchitecting such a core component introduces significant risk.

However, in September 2017 Andres Freund started implementing JIT support for expressions [**?**]. Most CPU time occurs in expression components (such as y ¿ 8 in SELECT * from table WHERE x ¿ 8;). Furthermore, tuple deformation provides significant benefits as it interacts with the cache and has poor branch prediction. PostgreSQL's JIT implementation is documented in the official PostgreSQL documentation [**?**].


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. [**?**]

In a pull request, Peter Eisentraut questioned whether the default JIT settings were too low. Despite this concern, PostgreSQL version 11 released with JIT disabled by default. Limited adoption prompted them to enable JIT by default in later releases to increase exposure and testing opportunities [**?**]. 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 2,229× slower [**?**]. Such failures reinforced the view that JIT features should remain disabled by default, leading to negative perceptions of JIT and compiled queries.

Two implementations of QEP query compilation with PostgreSQL exist. Vitesse DB, the first implementation, made public posts seeking testing assistance. They became generally available in 2015, but their website is offline now and there is not much mention of them. PgCon presented a second implementation, achieving 5.5× speedup on TPC-H query 1 with more extensive documentation [**?**]. However, they did not publicize their implementation or show that it is easy for people to use.

Full JIT implementation was preceded by profiling work showing different TPC-H benchmarks pressure different nodes (Figure 3.2), enabling informed decisions about optimization priorities. Their core method is generating a function that represents a node in the plan tree, then inlining the function into the final LLVM IR [**?**] 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 [**?**], similar to approaches taken by HyPer [**?**].

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 [**?**]. [**?**]

Other database systems also support extensions, and many systems rely on PostgreSQL's extension system. MySQL, ClickHouse, DuckDB, Oracle Extensible Optimizer all support similar operations. More than just PostgreSQL can be extended in this manner, avoiding the need to create databases from scratch.

3.3 System R

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

Due to the implementation overhead of parsing, validity checking, and access path selection, a compiler was appealing. These were not supported within the running transaction by default, and they leveraged pre-compiled fragments of Cobol for the reused functions to improve their compile times. Such custom implementation was necessary at the time due to the lack of compiler writing tools. 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 Tableau purchased in 2016 for production use [**?**], proving that in-memory JIT databases can scale to production workloads. Development began in 2010, with their flagship paper releasing in 2011 for the compiler component [**?**], and in 2018 they released another flagship paper about adaptive compilation [**?**]. However, commercialisation poses research challenges since the source code is not accessible, though a binary is available on their website 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. LLVM's JIT executor then executes the IR. 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.
[?]

Figure 3.3 demonstrates that HyPer reduced compile time by many multiples. Figure 3.4 shows they achieved many times fewer branches and branch mispredictions compared to their MonetDB baseline [?]. Such improvements resulted from HyPer's output containing 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.
[?]

In 2018, HyPer separated the compiler into multiple rounds. An interpreter executes byte code generated from LLVM IR, allowing unoptimised machine code execution in initial stages and optimised machine code in later stages. Figure 3.5 visualises compilation times for each stage. However, they had to create the byte code interpreter themselves to enable this.

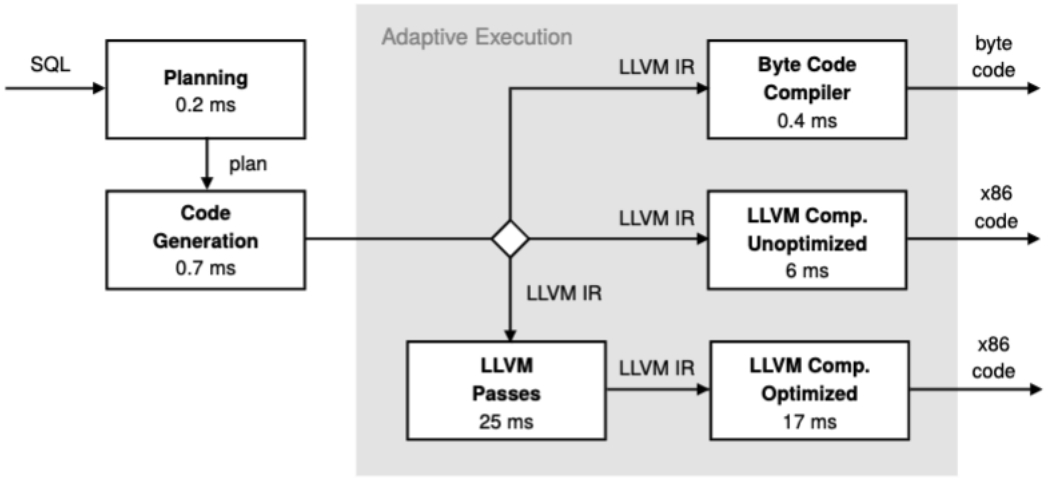


Figure 3.5: HyPer execution modes and compile times.
[?]

The 2018 paper also improved their query optimisation by adding a dynamic measurement for how long live queries are taking. 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.

They evaluated this approach using TPC-H Query 11 with 4 threads. The adaptive execution strategy outperformed bytecode-only execution by 40%, unoptimised compilation by 10%, and optimised compilation by 80%. This improvement occurs because the single-threaded compilation can run in parallel with query execution on other threads.

Utilising additional LLVM compiler stages, improved cost models, and multi-threaded compilation/execution created a viable JIT compiled-query application. A primary criticism: they effectively wrote the JIT compiler from scratch, requiring substantial engineering effort. Most additions are not unique to database JIT compilers; they mostly address compiler latency.

3.5 Umbra

Umbra, created in 2020 by Thomas Neumann (HyPer's creator), demonstrates that HyPer's in-memory concepts apply to on-disk systems [**?**]. Recent SSD improvements and buffer management advances made this possible. Umbra integrates LeanStore concepts for buffer management and latching, combined with HyPer's multi-version concurrency, compilation, and execution strategies. A hybrid approach produced an on-disk database that is scalable, flexible, and faster than HyPer itself.

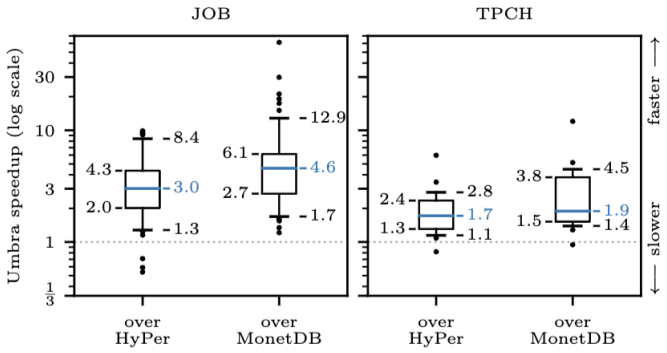


Figure 3.6: Umbra benchmarks. [**?**]

They introduced an optimisation enabling the compiler to dynamically change the query plan [**?**]. Using metrics collected during execution, they swap join order or join types. Such dynamic planning improved data-centric query runtimes by a factor of 2. Other databases achieve this by invoking the query optimiser multiple times; Umbra's approach of invoking the compiler multiple times during execution provides additional benefits.

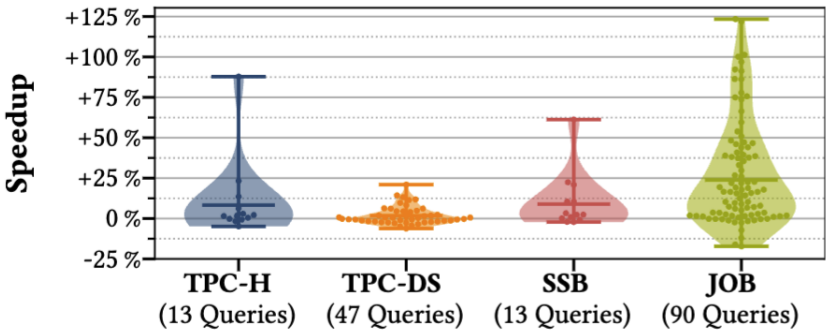


Figure 3.7: Umbra benchmarks after *adaptive query processing* (AQP). [?]

Umbra is currently ranked as the most effective database on ClickHouse's benchmarks [?]. Compiler overhead remains a criticism, but direct JIT compiler access enabling adaptive compilation for optimisation choice changes provides distinct advantages. Additionally, Umbra supports user-defined operators, enabling efficient custom algorithm integration [?].

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 [?]. Its primary purpose, however, is to serve as a framework for implementing other concepts in database research so that they do not need to rewrite the framework later [?]. However, using WebAssembly meant they can omit most of the optimisations that HyPer did while maintaining higher performance. Furthermore, they have a minimal query optimiser and instead rely on the V8 engine.

V8's Liftoff component adds early-stage execution to reduce query startup overhead [?]. Liftoff produces machine code quickly while skipping optimisations; TurboFan then provides second-stage compilation in the background during execution. HyPer's direct bytecode interpreter achieves lower execution startup times.

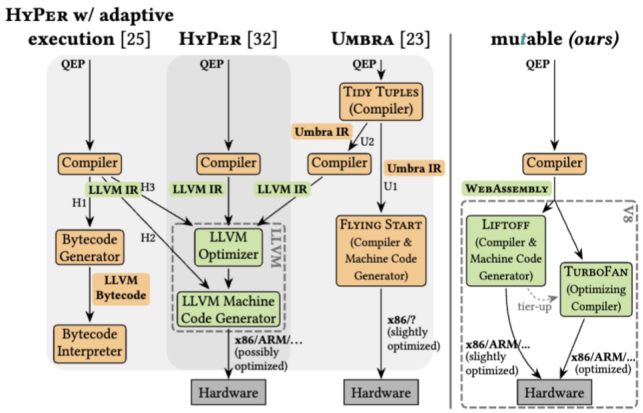


Figure 3.8: Comparison of mutable to HyPer and Umbra.
[?]

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

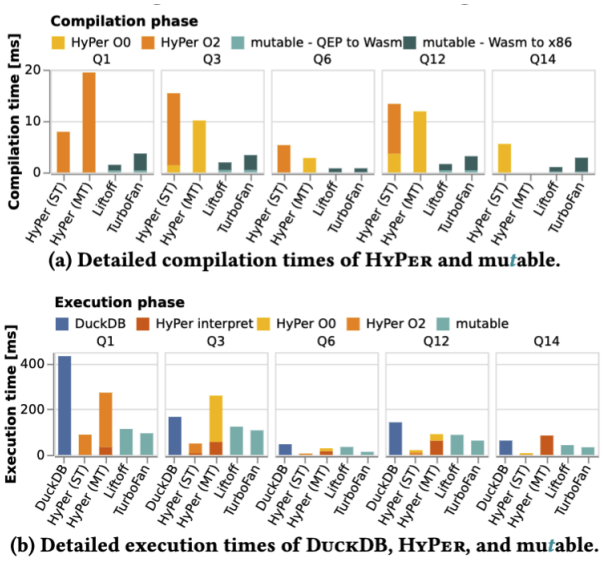


Figure 3.9: Benchmarks produced by Mutable.
[?]

3.7 LingoDB

LingoDB, introduced in 2022, proposed using the MLIR framework for optimisation layers [**?**]. Traditional databases follow a standard pipeline: parsing SQL into a query tree, converting to relational algebra, optimizing using manual implementations, creating a plan tree, and either executing or compiling to a binary. MLIR streamlines this by parsing directly to a high-level MLIR dialect, applying optimisation passes to the plan, and using LLVM compilation directly without intermediate conversions.

The LingoDB architecture can be seen in Figure 3.10, which begins by parsing SQL into a relational algebra dialect. MLIR's dialect system and code generation define these dialects. The compiler consists of multiple dialect layers: a relational algebra dialect for high-level queries, a database dialect for database-specific types and operations, a DSA dialect for data structures and algorithms, and a utility dialect for support functions. This multi-stage design splits the intermediate representation into three levels: relational algebra, a mixed dialect layer, and finally LLVM code [**?**].

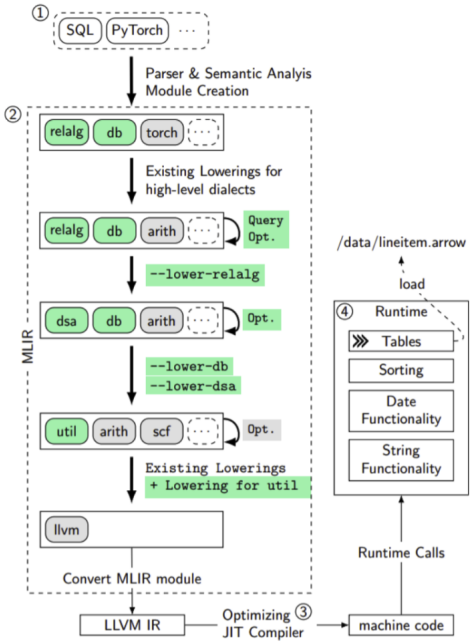


Figure 3.10: LingoDB architecture [**?**].
[**?**]

Results show lower performance than HyPer but better than DuckDB [**?**]. Performance was not the primary focus; rather, implementing standard optimisation patterns within the compiler was key. Notably, LingoDB uses approximately 10,000 lines of code for query execution model, while Mutable uses 22,944 lines despite skipping query optimization. Comparisons show LingoDB uses three times less code than DuckDB and 5× less than NoisePage.

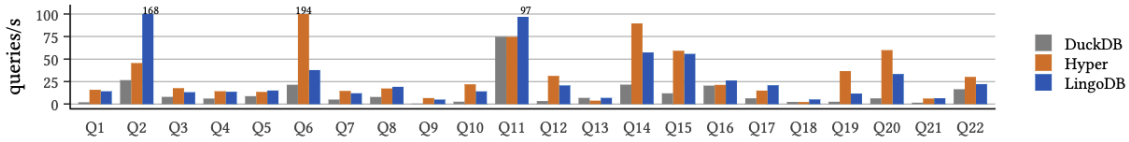


Figure 9: Query execution performance (compilation not included) for DuckDB, Hyper and LingoDB (SF=1)

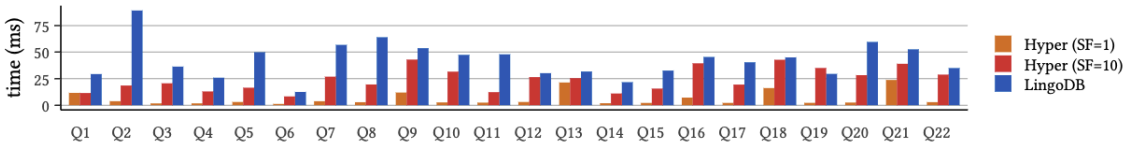


Figure 3.11: LingoDB benchmarking.
[**?**]

In later research, LingoDB also explores obscure operations such as GPU acceleration, using the Torch-MLIR project's dialect, representing queries as sub-operators for acceleration, non-relational systems, and more [**?**]. For our purposes, the appealing part of their architecture is that they use `pg_query` to parse the incoming SQL, which means their parser is the closest to PostgreSQL's. Section 4.1 explores this in the design.

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. The benchmarks used TPC-H as most of the involved pieces used it themselves [**?**], 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.

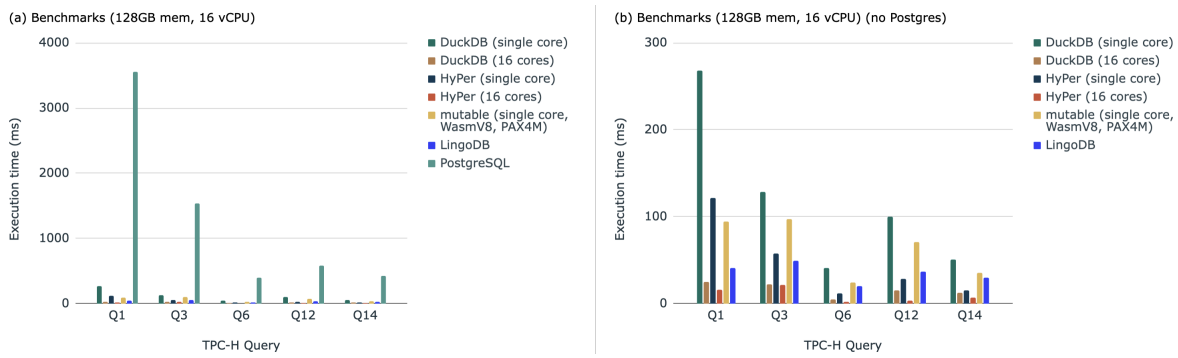


Figure 3.12: Benchmarking results.

To identify how much potential gain there is in a major on-disk database, The analysis used `perf` on PostgreSQL during TPC-H queries 1, 3, 6, 12 and 14 in Figure 3.13 [**?**]. 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.

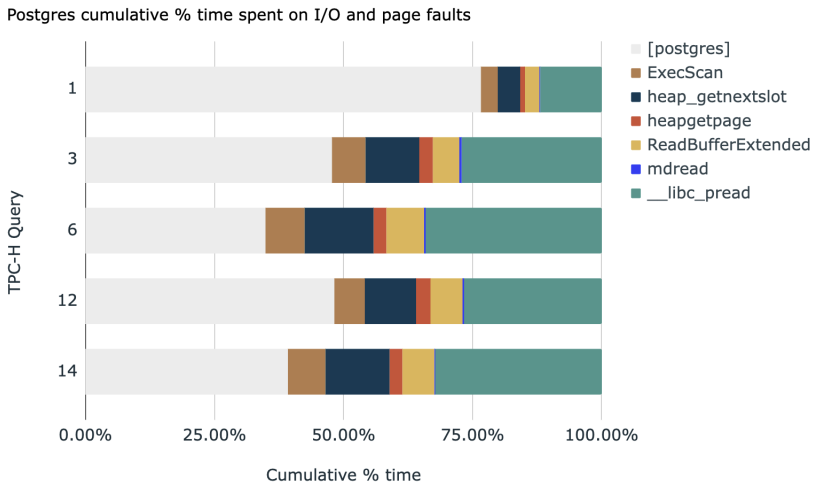


Figure 3.13: PostgreSQL's time spent in the CPU, measured with prof.

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 [**?**]. These commercial databases reaching high performance is a symptom of this. Testing the wide variety of ACID requirements is a significant undertaking.

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

Relying on an extension system such that it is an optional feature means users can install the optimisations, and testing can occur 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 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

A database and compiler selection required several criteria: strong extension support, wide-spread usage, high performance, and a volcano execution model as a base. For the compiler, ideally it would use a similar interface to the target database when parsing SQL, demonstrate strong performance results, and be open source. Such criteria eliminated HyPer, Umbra, and System R, leaving Mutable and LingoDB. LingoDB parses inputs with `pg_query`, matching PostgreSQL well.

PostgreSQL and LingoDB were selected. PostgreSQL offers strong extension support, enabling runtime hook-based execution engine overrides. TimescaleDB (now TigerData), discussed in Section 3.2, exemplifies such approaches [**?**]. A significant challenge arose: LingoDB's columnar, in-memory architecture required substantial adjustments. Additionally, LingoDB lacks index support, potentially biasing benchmarks against PostgreSQL. LingoDB's recent versions contain numerous unneeded features and opti-

misations, so the 2022 version from their initial paper was selected to simplify implementation effort.

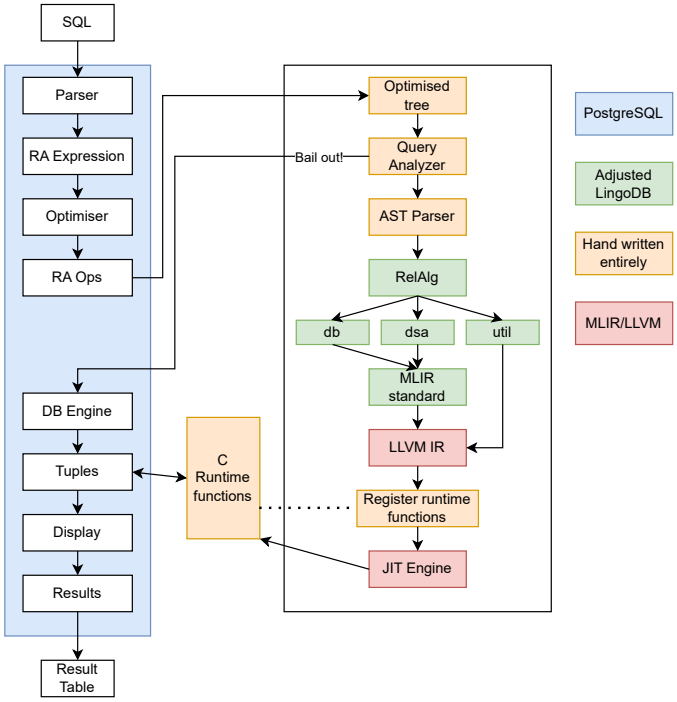


Figure 4.1: System design with labels of component sources.

LingoDB is integrated into PostgreSQL as seen in Figure 4.1. Blue components represent PostgreSQL, with the left pipeline showing the entire PostgreSQL execution flow. Queries reach runtime hooks, where a handwritten analyser determines executability before parsing. Handwritten components appear in light-peach. Processing continues through LingoDB code with custom runtime hooks and minor edits (annotated in green). Finally, compilation produces LLVM IR with embedded runtime hooks for PostgreSQL data access.

Query failures should still allow result return and graceful fallback to PostgreSQL. A try-catch pattern at the AST parser entrance routes failed queries back to PostgreSQL. However, such protection does not prevent system panics such as segmentation faults.

AST Parser implementation is expected to be most time-consuming, as it receives the

optimised plan tree from PostgreSQL. LingoDB was designed to parse query trees from the Parser stage in Figure 4.1. The implementation includes 18 plan nodes and 14 expression nodes.

TPC-H query support was the final goal. Test-driven development drove implementation using PostgreSQL's `pg_regress` module for SQL query creation and expected output definition. A progressive test set built from basic queries up to TPC-H queries. Such progression enabled incremental node implementation during development and quick validation of safe changes.

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)
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

Continued on next page

Table 4.1 – *Continued from previous page*

File Name	**Aim**
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, surrounding steps exist since the intention is not to replace the entire executor with hooks, requiring memory context activation and switching.

Next, the `QueryDesc` pointer, containing the query request, needed to be passed to C++. A design decision arose from this requirement: 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. Debugging requires navigating several layers: the PostgreSQL postmaster, the client connection, the runtime hook entrance, C++ code, and the JIT runtime. Bugs can occur at any of these levels, making debugging challenging. This poses a particular difficulty when dealing with segmentation faults and other errors that lack logging information.

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 handle 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. The scope included 94 operations total across all dialects (93 in LingoDB, 94 in pgx-lower) and 21 types (identical count), with most changes addressing LLVM 20 API compatibility.

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` adds several custom runtime implementations: reading tuples from PostgreSQL and storing them for streaming, modifying LingoDB's runtime implementations, and replacing the sort and hash table implementations to use PostgreSQL's API instead of standard library functions.

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.

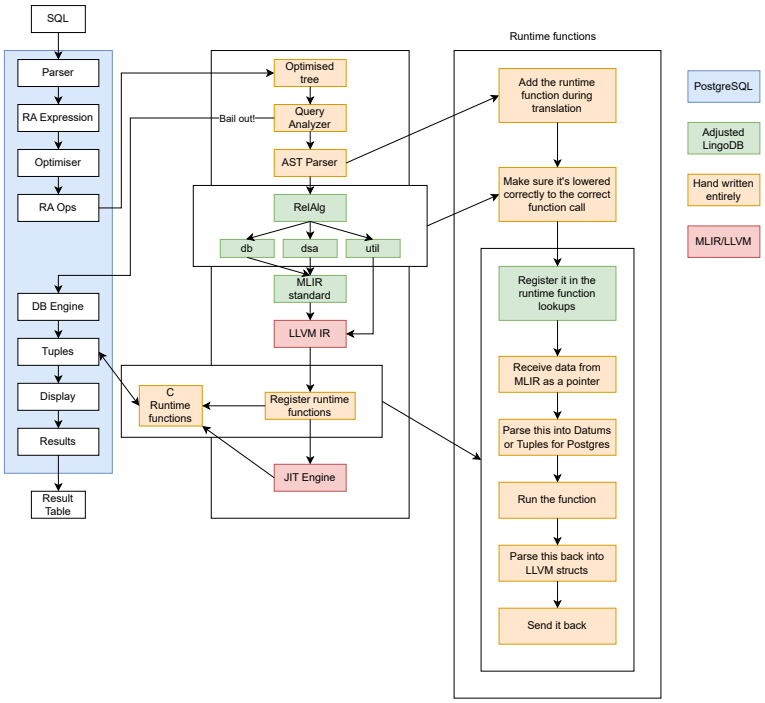


Figure 4.2: Runtime function integration diagram.

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.

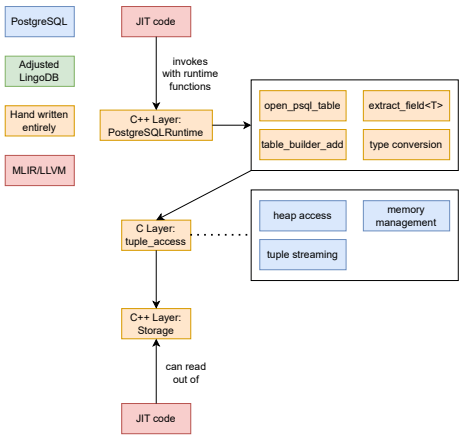


Figure 4.3: PostgreSQLRuntime.h component design.

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. Each translation function follows a consistent pattern. First, children of the plan are translated using post-order traversal. Then, the node is translated into the MLIR relational algebra dialect, and a translation result is returned.

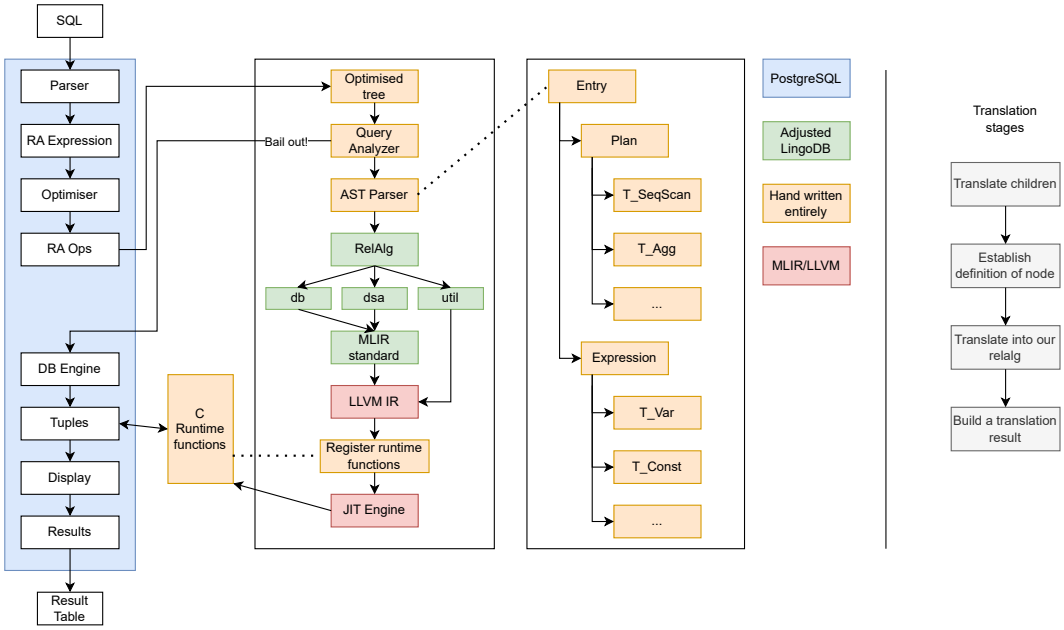


Figure 4.4: AST translation design and high-level steps in each function.

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 `QueryCtxT` object is passed down the tree, and when mutated, a new instance is created for child nodes. Meanwhile, `TranslationResult`s flow upward to represent each node's output, providing strong type-correctness in theory. However, this pattern is not strictly enforced in practice.

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

```
auto translate_plan_node(QueryCtxT& ctx, Plan* plan) -> TranslationResult;
auto translate_seq_scan(QueryCtxT& ctx, SeqScan* seqScan) -> TranslationResult;
auto translate_index_scan(QueryCtxT& ctx, IndexScan* indexScan) -> TranslationResult;
auto translate_index_only_scan(QueryCtxT& ctx, IndexOnlyScan* indexOnlyScan) -> TranslationResult;
auto translate_bitmap_heap_scan(QueryCtxT& ctx, BitmapHeapScan* bitmapScan) -> TranslationResult;
auto translate_agg(QueryCtxT& ctx, const Agg* agg) -> TranslationResult;
auto translate_sort(QueryCtxT& ctx, const Sort* sort) -> TranslationResult;
auto translate_limit(QueryCtxT& ctx, const Limit* limit) -> TranslationResult;
auto translate_gather(QueryCtxT& ctx, const Gather* gather) -> TranslationResult;
auto translate_gather_merge(QueryCtxT& ctx, const GatherMerge* gatherMerge) -> TranslationResult;
auto translate_merge_join(QueryCtxT& ctx, MergeJoin* mergeJoin) -> TranslationResult;
auto translate_hash_join(QueryCtxT& ctx, HashJoin* hashJoin) -> TranslationResult;
auto translate_hash(QueryCtxT& ctx, const Hash* hash) -> TranslationResult;
auto translate_nest_loop(QueryCtxT& ctx, NestLoop* nestLoop) -> TranslationResult;
auto translate_material(QueryCtxT& ctx, const Material* material) -> TranslationResult;
```

```
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. The subsections explain these more specifically.

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.

Several common node definitions are helpful to understand. Nodes commonly have an `InitPlan` parameter, which is a function called before the node executes and initializes variables such as parameters and catalogue lookups. `targetlist` contains the output of the node, and `qual` specifies which tuples should pass through. Join nodes have left and right child trees, typically referred to as *inner* and *outer* children. These signify the inner and outer loops of the nested for-loop that is created.

Expression Translation - Variables, Constants, Parameters

PostgreSQL identifies values using variable nodes and parameter nodes. These are tracked in a schema/column manager class and the `QueryCtxT` object. Variables are typically defined within scans, while parameters are intermediate products. Identifying them presented challenges due to multiple interacting identifiers (`varno`, `varattno`, and special values for index joins). To handle this complexity, a generic function was added to the QueryCtxT object: `resolve_var`. This function is used extensively throughout the translation logic.

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

Plan translation - Scans

PostgreSQL supports multiple scan types: sequential scans, subquery scans, index scans, index-only scans, bitmap heap scans, and CTE scans. However, all scan types except subquery and CTE scans map to sequential scans in this implementation. This trade-off reduces implementation complexity at the cost of query optimisation, particularly for index scans.

Index scans use special annotations for variables via `INDEX_VAR`, which requires custom variable resolution. Additionally, we handle the qualifiers (scan filters) `indexqual` and `recheckqual` as generic filters. In PostgreSQL, these qualify at different stages, but

since we skip index implementation, both become generic filters here.

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 with many properties. It includes an aggregation strategy (ignored in favour of a simpler algorithm), splitting specification (not utilised), and group columns. The node tracks the number of groups it produces and manages its own operators such as COUNT, SUM, and more. Additionally, it uses special varnos for variable lookups (represented as -2), requiring 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, two layers exist for translation: 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 in the joins implementation is preventing double computations. LingoDB handles this by computing the inner join separately, building a vector of results, and then iterating over the outer operation while reusing the pre-computed inner section. This approach prevents duplicated computation at the cost of increased memory usage. While theoretically acceptable, the vector would need to implement disk spillage. In practice, memory usage did not become problematic enough to require this feature.

Expression Translation - Nullability

PostgreSQL tracks nullability information in the plan tree passed to pgx-lower. However, LingoDB's lowering operations can create situations where previously non-null objects become nullable through outer joins, aggregations, unions, and predicate evaluation. Since nullability propagates like not-a-number (affecting everything it touches), this introduces significant implementation complexity.

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. To implement im-

plicit upcasting, a class was extracted from LingoDB's DB dialect: `SQLTypeInference`. Rather than relying on PostgreSQL's OID system for finding operations, operators are converted to strings ("¿", "¡", etc.) for lookup. This prevents issues with OID precision specifications that lead to unidentified operations. The same approach is used in function nodes, aggregation functions, sort operations, and scalar maths. When performing operations on different types, `SQLTypeInference` automatically upcasts both operands to the larger datatype. For example, with `i16 + i32`, both values are cast to `i32`.

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, several 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 [**?**]. 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, while 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. 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. A dedicated machine was configured for this purpose: an Intel i5-6500T with 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.

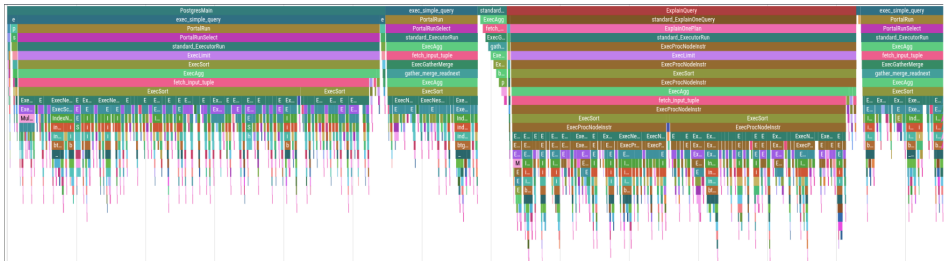


Figure 4.5: PostgreSQL's magic-trace flame chart for TPC-H query 3 at scale factor 0.05 (approximately 5 MB of data).

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.

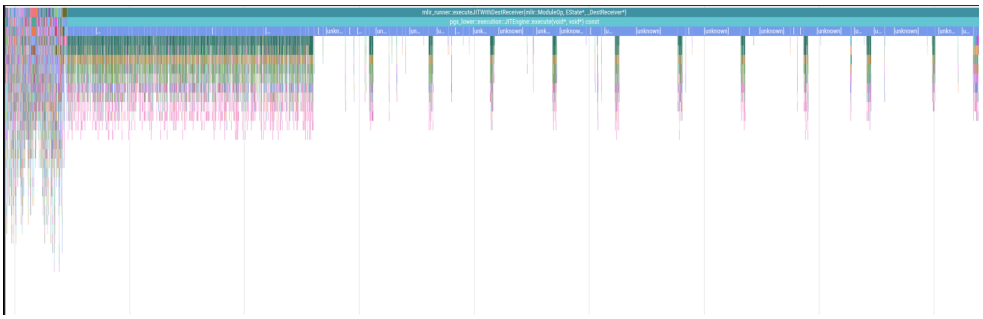


Figure 4.6: pgx-lower's magic-trace flame chart for TPC-H query 3 at scale factor 0.05 before optimisation.

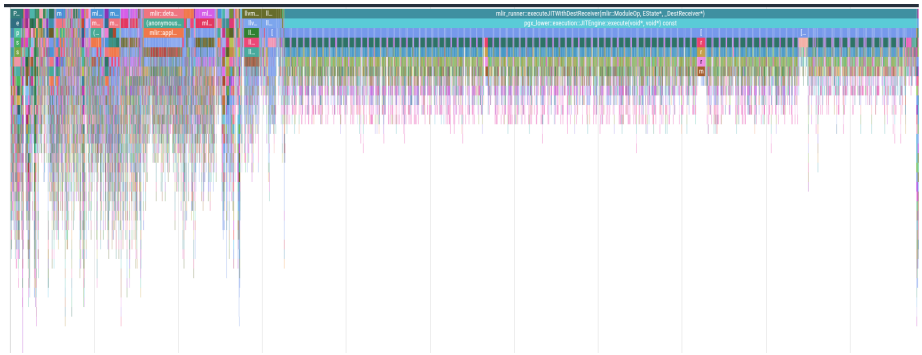


Figure 4.7: pgx-lower's magic-trace flame chart for TPC-H query 3 at scale factor 0.05 after optimisation.

Subsection 4.2.12 explains the specifics of running these in a stable way.

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 MB 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`. The implementation uses several technologies: Python for the backend server, SQLite for query caching, and React for the frontend. Docker containers support the reverse proxy with Nginx, while a private

Grafana dashboard provides health monitoring.

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 MB 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 for multiple iterations 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"

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

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

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

One thing to note here is that it was decided that only PostgreSQL and pgx-lower would be compared, rather than all the databases mentioned in Chapter 3. As Section 3.8 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

Results using the method from Subsection 4.2.12 were produced as follows. Box plots overlaid on graphs represent the 5th, 25th, 50th, 75th, and 95th percentiles. Hollow circles mark outliers; when inconvenient to display (as in Figure 5.4), arrow annotations are used instead. Matplotlib and Seaborn generated all visualizations in Python.

Another result worth mentioning is the integrated LingoDB code (including ./src/lingodb and ./include/lingodb) contains 13,875 lines of C++, while the pgx-lower section (./src/pgx-lower and ./include/pgx-lower contains 12,324 lines of C++. This was measured with tokei commands, a command line utility for counting lines of code. In comparison, the official PostgreSQL executor is roughly 82,875 lines of code, and LingoDB is roughly 30,000 lines of code [**?**, **?**].

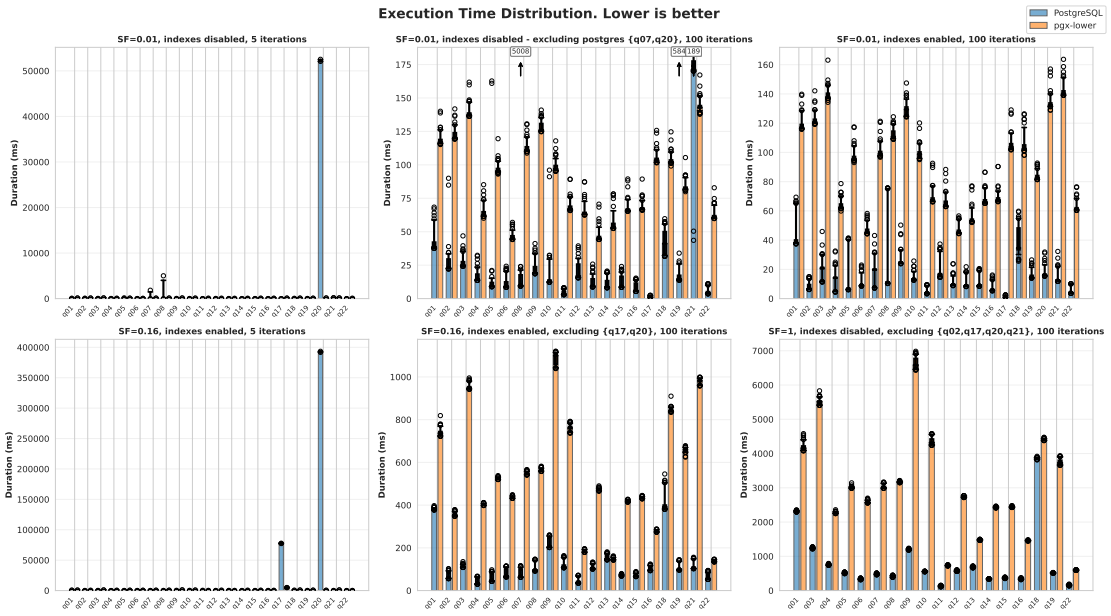


Figure 5.1: Overall benchmarking represented with box plots.

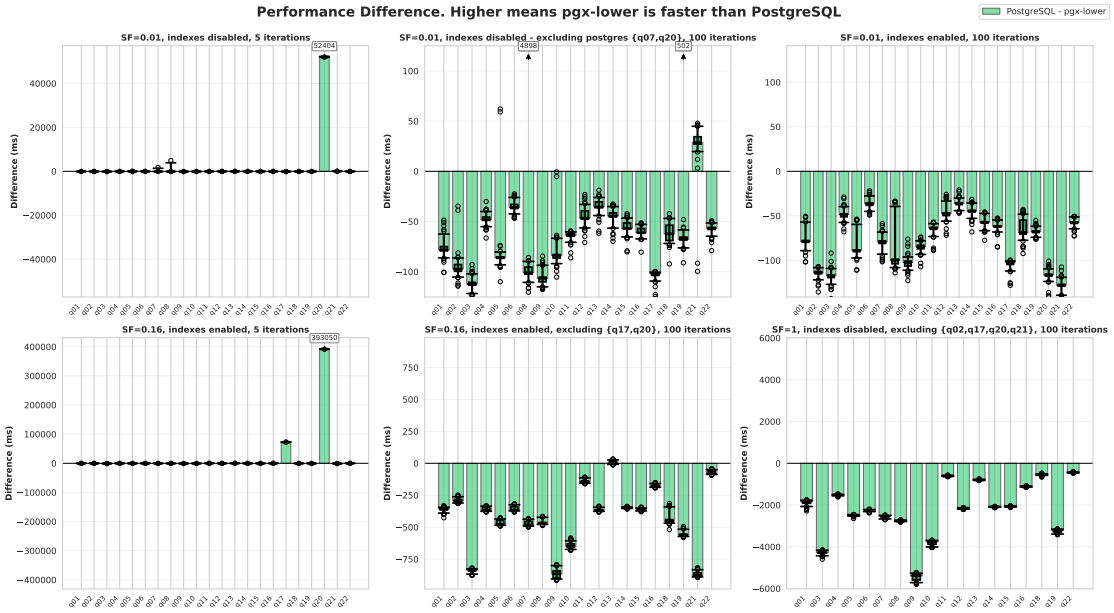


Figure 5.2: Difference in latency benchmarks between PostgreSQL and pgx-lower.

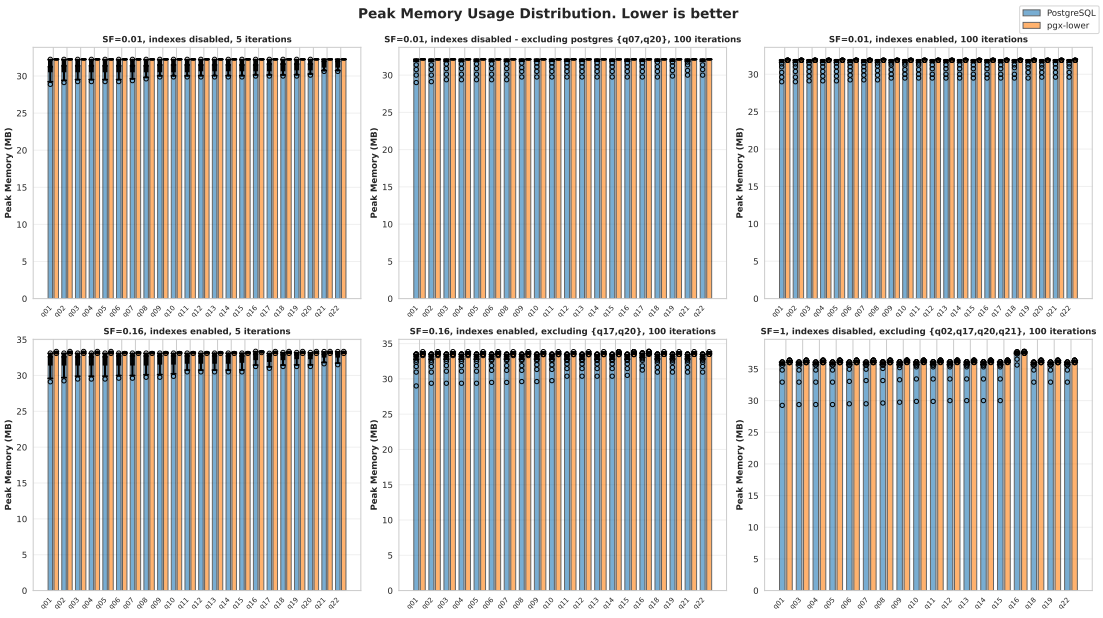


Figure 5.3: Peak memory usage of queries.

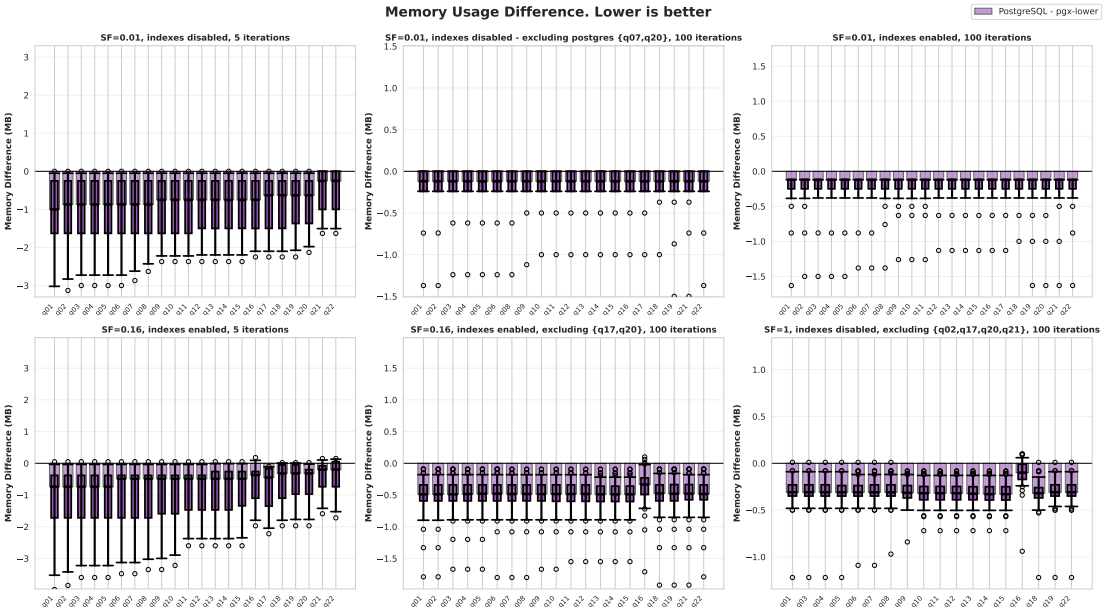


Figure 5.4: Difference in peak memory usage of queries.

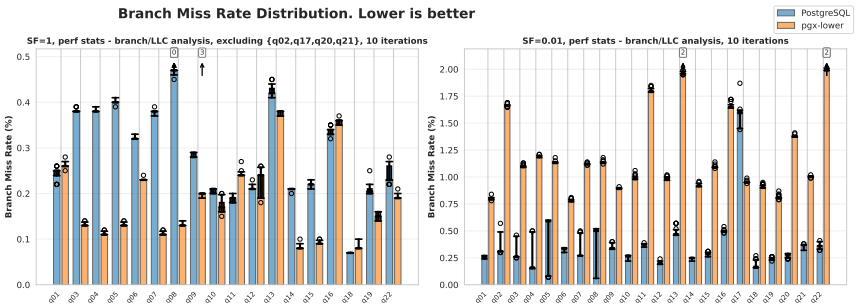


Figure 5.5: Branch miss rate.

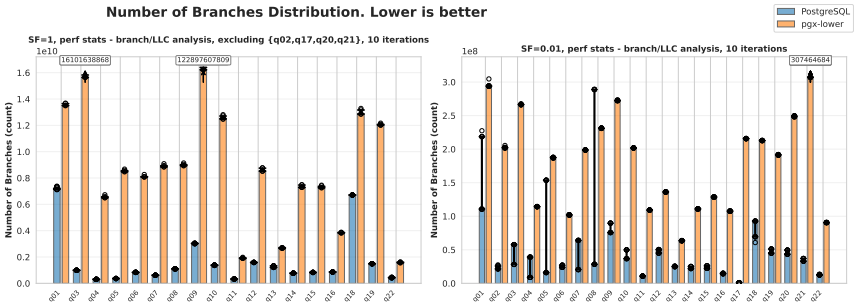


Figure 5.6: Number of branches.

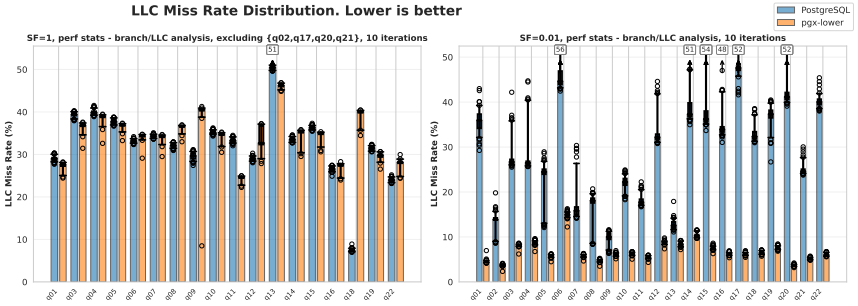


Figure 5.7: Last-level-cache miss plots.

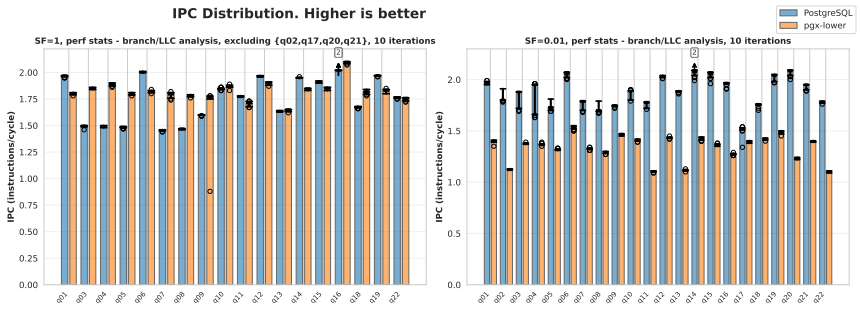


Figure 5.8: Instructions per (CPU) cycle plot.

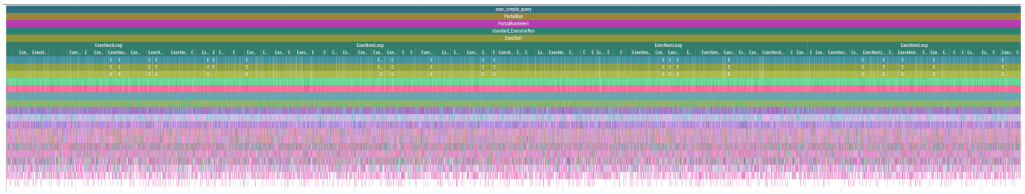


Figure 5.9: PostgreSQL TPC-H query 20 indexes enabled at SF = 0.16. Runtime: 15 minutes.

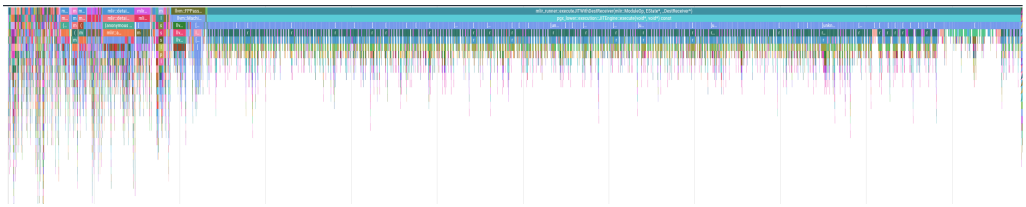


Figure 5.10: pgx-lower TPC-H query 20 indexes enabled at SF = 0.16. Runtime: 1.18 seconds.

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, results are mixed. Figures 5.1 and 5.2 show that without indexes at scale factor 0.01, queries 7 and 20 are orders of magnitude slower than other queries. Such patterns repeat at scale factor 0.16 even with indexes enabled. In contrast, pgx-

lower shows consistent performance across these queries. Visible in Figures 5.9 and 5.10, the difference appears: PostgreSQL uses nested loop joins while pgx-lower uses hash joins. Appendices A.1, A.2, and A.3 show the query and both systems' plan trees. In pgx-lower's output, joins are annotated with "hash", indicating hash join usage. This comparison is not entirely fair because pgx-lower uses a hash join while PostgreSQL uses a nested loop join (NestLoop_T), which would benefit from indexing.

Figure 5.10 reveals minimal time spent in LLVM code generation. Of the 1.18 second total query execution time, only 14.93 milliseconds occurs in `llvm::SelectionDAGISel::runOnMachineFunction` the LLVM IR to machine code conversion step. Total compilation time reaches 236.32 milliseconds, with most spent in MLIR passes. These results suggest MLIR optimization is more expensive than LLVM code generation. Possibly the LLVM ORC JIT does minimal work for each query, spreading compilation across background execution.

RAM usage appears similar between pgx-lower and PostgreSQL (Figures 5.3 and 5.4). Such similarity demonstrates successful adaptation of LingoDB's in-memory operations to PostgreSQL's disk-oriented architecture. Maximum difference is just over 3 MB, with a mean difference of 0.34 MB across all queries.

Branch prediction potential appears in Figure 5.5. At scale factor 1, pgx-lower achieved a median branch miss rate of 0.16%, compared to PostgreSQL's 0.28%. However, smaller scale factors with smaller queries inverted this: PostgreSQL had 0.38% and pgx-lower had 1.09%. Such results suggest the compiled JIT runtime has better branch prediction, but the compilation stage or pre-warm-up code has worse prediction. Number of branches explains part of the difference (Figure 5.6): PostgreSQL had 28,430,465 median branches, while pgx-lower had 195,299,021.50 (6.87× more). Higher branch counts could indicate more optimization opportunities.

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%. A

likely explanation: the LLVM compiler stage uses its cache much more effectively, but the actual JIT runtime performs comparably. Expected with the current approach, improvements would emerge if the JIT stage dynamically changed the plan, similar to HyPer's approach.

5.2.1 Test Validity

Increasing the number of iterations to make the outputs more reliable succeeds, 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 0.4% 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

Replacing PostgreSQL's execution engine with a JIT-focused approach offers multiple research directions. Improvements can be made by fully implementing the plan nodes and optimising further. However, ensuring the final product remains useful requires considering other base databases, compilers, languages, and compatible execution engines.

Full extension implementation requires complete plan tree node implementation and query analyser improvements. Only the minimum for TPC-H was implemented, requir-

ing pg_bench and isolationtester validation before user deployment. Needed additions include indexes, WINDOW functions, the other 22 missing plan nodes, and missing execution nodes.

Core optimization work involves leveraging existing research and clearer PostgreSQL API usage. Key improvements include: pre-compiling PostgreSQL functions into LLVM, then inlining instead of crossing the LLVM-C++ boundary; adaptive compilation/query planning; and Umbra's LeanStore-style buffering. Most optimisations apply specifically to LLVM/MLIR systems; WebAssembly alternatives could skip many of these. Broader improvements include parallelism enhancement, JIT tuning, cross-platform support, subquery deduplication, and further optimisations.

Research impact remains vital in database systems, reflecting Michael Stonebraker's concern about field progress [**?**]. For successful projects, database costs are typically minor relative to overall profitability, making higher throughput less critical. Practically, latency gains more often come from application-level caching (such as Redis) than database optimization. Complex OLAP queries often run on large-scale systems that migrate away from PostgreSQL to more scalable databases such as ClickHouse or Apache Hive. Alternative systems may provide better development platforms for this architecture.

PostgreSQL was selected for its large popularity; LingoDB was chosen because it matched PostgreSQL's interfaces while remaining open source. While PostgreSQL works reasonably for this approach and addresses real problems, better alternatives may exist. Most dedicated OLAP systems already employ JIT or vectorized approaches.

Development with LingoDB provided helpful constraints and faster iteration due to its established nature. However, LingoDB's columnar, in-memory architecture required extensive modifications. Additionally, its redundant query optimisation engine was unnecessary since PostgreSQL already provides thorough optimization. A better approach would involve building the engine from scratch or selecting a more suitable base system. Umbra would be ideal based on its description, but closed-source status prevents use. Alternative approaches could integrate an established OLAP system's

engine (such as ClickHouse), routing queries based on analyser rules instead of using PostgreSQL's executor.

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.

Multiple promising research directions emerge from this work. Most appealing is integrating NoisePage or ClickHouse into PostgreSQL as a drop-in engine replacement. Such an approach provides a more complete ecosystem; primary work will focus on adapters to adjust queries. Furthermore, pg_duckdb has already done this, but it is a vectorised engine.

Chapter 6

Conclusion

pgx-lower demonstrates that JIT-compiled query execution can be integrated into production databases via extensions. The implementation retrofits LingoDB's compiler into PostgreSQL while maintaining ACID properties, achieving improved branch prediction and cache efficiency on TPC-H. The key contribution is methodological. Extension mechanisms provide a practical pathway for productionising database research without rewrites or official integration.

While the implementation covers only part of the query space and performance is mixed, pgx-lower validates that production databases can adopt modern compiler techniques through extensions rather than from-scratch implementations. This bridges the gap between academic prototypes and production systems, suggesting a more realistic research methodology.

Future work includes implementing all the plan and expression nodes in the executor, improving the runtime, or pivoting and targetting a different database system, compiler, or language, as discussed in Subsection 5.2.2.

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
```

```
        and n_name = 'CANADA'
order by
        s_name
```

A.2 PostgreSQL Execution Plan

```
Sort  (cost=46847.99..46848.00 rows=1 width=52) (actual time=69.713..69.728 rows=1 loops=1)
  Sort Key: supplier.s_name
  Sort Method: quicksort  Memory: 25kB
  ->  Nested Loop Semi Join  (cost=0.42..46847.98 rows=1 width=52) (actual time=68.025..69.569 rows=1 loops=1)
        ->  Nested Loop  (cost=0.14..29.27 rows=1 width=56) (actual time=0.493..0.667 rows=3 loops=1)
              Join Filter: (nation.n_nationkey = supplier.s_nationkey)
              Rows Removed by Join Filter: 97
              ->  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)
              ->  Materialize  (cost=0.00..12.13 rows=1 width=4) (actual time=0.004..0.005 rows=1 loops=100)
                    ->  Seq Scan on nation  (cost=0.00..12.12 rows=1 width=4) (actual time=0.414..0.421 rows=1 loops=1)
                          Filter: (n_name = 'CANADA'::bpchar)
                          Rows Removed by Filter: 24
        ->  Nested Loop  (cost=0.28..46818.70 rows=1 width=4) (actual time=22.958..22.958 rows=0 loops=3)
              ->  Seq Scan on part  (cost=0.00..66.00 rows=20 width=4) (actual time=0.111..1.250 rows=16 loops=3)
                    Filter: ((p_name)::text ~~ 'forest%'::text)
                    Rows Removed by Filter: 1973
              ->  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)
                    Index Cond: ((ps_partkey = part.p_partkey) AND (ps_suppkey = supplier.s_suppkey))
                    Filter: ((ps_availqty)::numeric > (SubPlan 1))
                    Rows Removed by Filter: 0
                    SubPlan 1
                      ->  Aggregate  (cost=2330.51..2330.52 rows=1 width=32) (actual time=31.616..31.617 rows=1 loops=2)
                            ->  Seq Scan on lineitem  (cost=0.00..2330.50 rows=1 width=5) (actual time=25.850..31.589 rows
                                  =2 loops=2)
                                  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))
                                  Rows Removed by Filter: 60173
Planning Time: 16.634 ms
Execution Time: 70.580 ms
```

A.3 pgx-lower MLIR Execution Plan

```
// MLIR Module Debug Dump: Phase 3a AFTER: RelAlg -> Optimised RelAlg
// Generated: 2025-11-22 23:21:32
// Total Operations: 120
// Module Valid: YES

module {
  func.func @main() -> !dsa.table {
    %true = arith.constant true
    %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>})}
    %1 = relalg.selection %0 (%arg0: !relalg.tuple){
      %39 = relalg.getcol %arg0 @lineitem::@l_shipdate : !db.date<day>
      %40 = db.constant(-2191 : i32) : !db.date<day>
      %41 = db.constant(-1826 : i32) : !db.date<day>
      %42 = db.between %39 : !db.date<day> between %40 : !db.date<day>, %41 : !db.date<day>, lowerInclusive : true,
            upperInclusive : false
      relalg.return %42 : i1
    } attributes {cost = 1.000000e+00 : f64, rows = 1.000000e+00 : f64}
    %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
  }
}
```