School of Computer Science and Engineering

Faculty of Engineering

The University of New South Wales

Exploring Just-in-Time Compilation in Relational Database Engines

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

Databases are used extensively by almost every system. With the recent advances in secondary storage (SSDs, NVMe), and compiler technology (MLIR), the opportunities to improve them continues to improve. Since they are widely used, the cost of powering all the databases is high, and lowering this slightly can have large impacts. This thesis investigates this by reviewing the current state of the literature, gaps in the literature, and proposes a project to explore this by extending PostgreSQL with MLIR.

Abbreviations

ACID	Atomicity, consistency, isolation, durability
CPU	Central Processing Unit
DB	Database
EXP	Expression (expressions inside queries)
IR	Intermediate Representation
JIT	Just-in-time (compiler)
JVM	Java Virtual Machine
LLC	Last Level Cache
MLIR	Multi-Level Intermediate Representation
QEP	Query Execution Plan
RA	Relational Algebra
SQL	Structured Query Language
SSD	Solid State Drive
TPC-H	Transaction Processing Performance Council

Contents

List of Figures

Chapter 1

Introduction

Databases have been used widely for a long time. Nowadays, they are the main bottleneck in many systems - especially on web servers and other large data applications [Kle19]. They are a critical component of many systems and a fundamental reason why these can function.

As modern hardware advances, the optimal algorithms for supporting databases changes. In the last several years, CPU cache sizes have become larger and secondary storage has become faster. Older databases assume that operations inside the CPU are constant due to the overhead of reading from disk, but this might not be true. One way to support modern hardware is to use a just in time compiler instead of the existing model.

In this thesis, the necessary background will be introduced in Chapter 2. The current literature on this topic will be explored in Chapter 3, and a project to explore this further will be defined in Chapter 4. This will be followed by Chapter 5 where the preliminary results and skills required will be explored.

Chapter 2

Background

2.1 Database Architecture

Most databases have an internal architecture similar to Figure 2.1. Structured Query Language (SQL) is parsed, then converted into a relational algebraic expression, which is used in the optimiser, and this is converted into an execution plan for the evaluation engine.

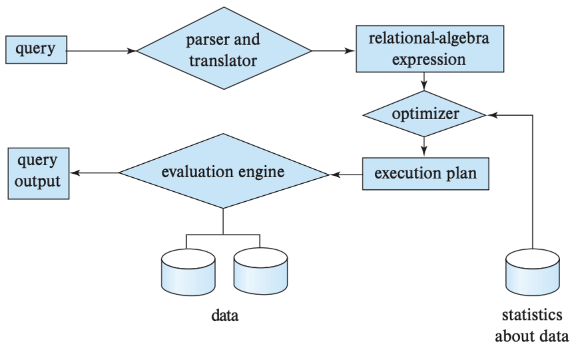


Figure 2.1: Database Structure
[SKS19]

For example, `select salary from instructor where salary < 75000;` can be translated into $\sigma salary < 75000(\Pi salary(instructor))$ inside the relational-algebra expression stage (understanding the meaning of these operators is not needed at this stage). In most cases, there can be alternative ways to write the algebraic expression, and there can also be multiple algorithms that can be used for each operator. The optimiser would pick what it thinks is the best algorithm to use, and the execution plan will generate a tree, or some representation of the query that can be executed. [SKS19]

Most query engines use a pull-based Volcano execution model. This calls a `next()` function on the root node, which requests one tuple from its children. For instance, in Figure 2.2, the `Project` operator would call hash join, which would then call its children until scan operators read from the disk, which passes the information back up [KLK+18]. `Select` might filter this tuple and request the next, or a node can change the tuple before returning it, such as `MergeJoin` [ZVBB11].

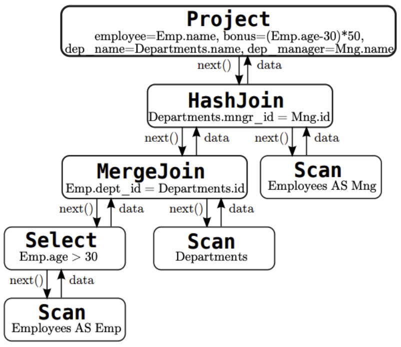


Figure 2.2: Volcano operator model tree.
[ZVBB11]

This model can be problematic in modern hardware because unless a single tuple fills the L1 cache, the CPU's caching will be underutilised. With modern hardware advancements making these caches larger, this has also become a more common and impactful issue. As a result, modern databases use either vectorised execution or compiled execution [KLK+18]. Vectorised execution requests multiple tuples, but this has downsides of more copying, and potentially overflowing the cache on operators such as merge join [KLK+18]. Compiled queries, on the other hand, will use caches almost fully but are more challenging to develop.

Relational databases typically prioritise ACID - Atomicity, Consistency, Isolation, and

Durability [SKS19]. Atomicity refers to transactions being treated as single units of work, consistency is that the data must be in a valid state before and after the query. Isolation means concurrent transactions do not interact with each other, and durability is that once something is committed it will stay committed even in system failure [SKS19].

2.2 Just-in-time (JIT) compilers

Just-in-time (JIT) compilers work by compiling code to machine code or byte code after it has been run multiple times. They are mostly used by interpreted languages as an alternative to eliminate ill-effects on performance [ZVBB11]. Advanced compilers can run the compilation a different thread while still running the program then swap in the compiled artifacts when it's done. They also have multiple layers, such as interpreting text, then byte code, then machine code, then optimised machine code [KLN18]. The main issue this introduces is that before the code is compiled, performance is poor. Depending on the language, there can be support for caching JIT compiled code, but in the context of database systems the produced machine code can require pointers to addresses, so it is not always viable.

On-disk databases consider CPU-based operations to be O(1) time complexity because of how expensive loading in pages from the disk is [SKS19]. As a result, JIT compilers are considered in the context of in-memory databases because their transactions are not bottlenecked by the speed of secondary memory. However, due to advances in secondary-storage some papers question whether CPU-time is still constant, and compiling queries would optimise cache usage which would lower how many page faults occur [NF20].

In the context of databases, most applications of JIT can be split into compiling only specific sections of queries (typically called EXP for expression), and others that compile the entire Query Execution Plan (QEP) [MYH$^+$24]. The results of this will be explored in Chapter 3.

2.3 Relevant Operating System features

In the context of a CPU with multiple cores, a CPU can transform a sequential execution into a pipelined execution. Essentially, CPU instructions are split into Fetch (IF), Instruction Decode (ID), Execute (EX) and Write-back (WB). The CPU can do multiple of these within a single cycle, however, they need to be independent [ZVBB11].

CPUs can avoid waiting for the calculation of a branch by using branch prediction, however, if a branch is wrong it can cost cycles to fix; for some processors it is at least 17 cycles [ZVBB11]. When a branch is incorrect, it could affect more than the thread that it's on, and it also has to reschedule when it is incorrect [ZVBB11]. Improving this is impactful - benchmarks show that Java outperforms C++ even on the maximum optimisation due to the advantage of collecting runtime statistics [RPML06] [OO20].

In 1999, a benchmarking paper measured four commercial databases and found 10-20% of their execution time was spent fixing bad branch predictions [ADHW99]. These databases were kept anonymous in the paper because of legal reasons, and this was also a long time ago, but it shows the importance of reducing branch prediction. Similarly, research specifically into branch prediction has said "although branch prediction attained 99% accuracy in predicting static branches, ... branches are still a major bottleneck in the system performance" [Jos21]. Modern measurements still find 50% of their query times are spent resolving hardware hazards, such as mispredictions, with improvements in this area making their queries 2.6x faster [Ker21].

2.4 MLIR and LLVM

The LLVM Project supports building compilers, and a large portion of modern compilers use its toolkit. It assists with optimising code, linking objects, and many other requirements different compilers share to prevent rewriting code [lat].

Multi-Level Intermediate Representation (MLIR) is another, newer toolkit that is tightly integrated with the LLVM project [LAB+21]. It's a higher-level tool that allows developers to express domain-specific dialects that can operate on different abstraction levels. For instance, low-level languages like LLVM intermediate representation and data flow graphs can be represented within the same dialect [LAB+21].

Chapter 3

Literature Review

3.1 Summary

In the following section a series of relevant databases and their architecture will be explored. This consists of System R in Section 3.2 and JAMDB in Section 3.3 as early research papers for compiled queries. Following that, HyPer will be explored in Section 3.4 and Umbra in Section 3.5 because they are functioning enterprise-level systems that are widely considered to have pioneered JIT, and made by mostly the same team. After that PostgreSQL is explored in Section 3.6, as an open-source example with real world impact. Mutable in Section 3.7, and LingoDB in Section 3.8 are recent, smaller, research databases in the field. After that is a summary of the existing gaps in the literature in Section 3.9 as well as issues in the field as a whole.

Originally, we started with compiled engines which were overtaken by popular Volcano models, and HyPer reintroduced the idea of compiled engines. Later on, PostgreSQL took note of the impact of HyPer, while Mutable and LingoDB criticized HyPer's approach with alternatives.

3.2 System R

System R was the first implementation of SQL, and was made in 1974 with a query compiling engine [CAB+81]. Their vision was supporting ACID requirements, which was explained in seven dots points as ACID was not a concept yet. It strived to run at a "level of performance comparable to existing lower-function database systems". Reviewers go as far as calling their choice to use a compiler "the most important decision in the design" [CAB+81].

Their choice to use a compiler was because the overhead of parsing, validity checking, and access path selection were not inside the running transaction. The compiler had pre-compiled fragments of Cobol for reused functions to improve compile times, and the compiler was mostly custom-made because there were not many tools at the time to support writing compilers [CAB+81]. It shows that the idea of compiled queries is not new, but over time databases that chose less performant architectures such as the volcano model survived the market to now. This is because it made development easier and more correct [SKS19].

3.3 JAMDB

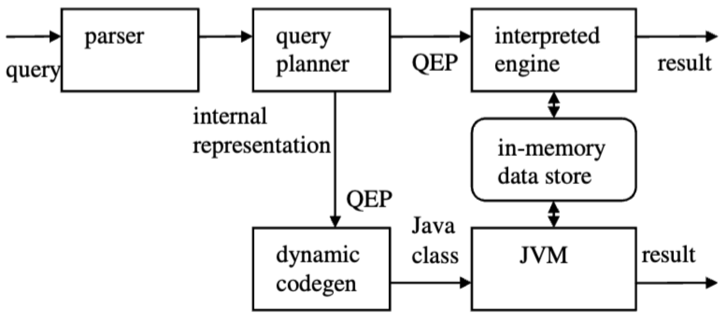
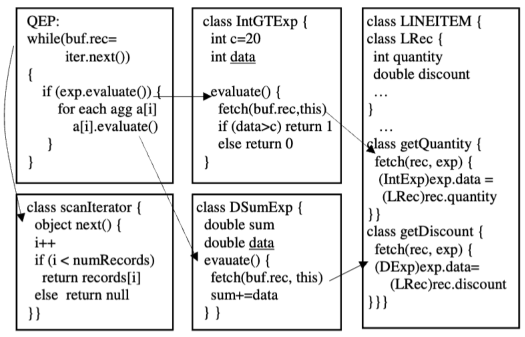


Figure 3.1: JAMDB Architecture. [RPML06]

In 2006, IBM's Research Center published an article about compiled query execution using the JVM for an in-memory database. As shown in Figure 3.1, they created two pathways for code to compare runtimes. One fed into a volcano interpreted engine, and the other used dynamic code generation to create a Java class that would run through the JVM.

In Figure 3.2 they also have an example of a produced interpreted plan (a) compared to a compiled plan (b) generated from Q1 of the TPC-H benchmark. The difference in complexity between the two shows how impactful their code generation is because the inlining, casting, and number of method invocations is improved [RPML06].



```
QEP:
while(buf.rec=
        iter.next())
{
   if (exp.evaluate()) {
      for each agg a[i]
         a[i].evaluate()
   }
}
```

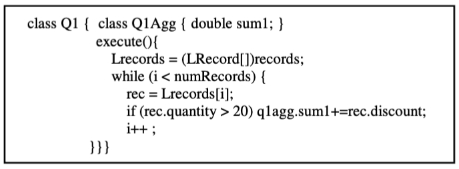
```
class IntGTExp {
  int c=20
  int data

  evaluate() {
   fetch(buf.rec,this)
   if (data>c) return 1
   else return 0
}
```

```
class LINEITEM {
class LRec {
 int quantity
 double discount
 …
}
  …
class getQuantity {
 fetch(rec, exp) {
  (IntExp)exp.data =
   (LRec)rec.quantity
}}
class getDiscount {
 fetch(rec, exp) {
 (DExp)exp.data=
    (LRec)rec.discount
}}}
```

```
class scanIterator {
  object next() {
  i++
  if (i < numRecords)
   return records[i]
  else  return null
}}
```

```
class DSumExp {
 double sum
 double data
 evauate() {
  fetch(buf.rec, this)
  sum+=data
 } }
```

(a) Interpreted Plan



```
class Q1 {  class Q1Agg { double sum1; }
            execute(){
               Lrecords = (LRecord[])records;
               while (i < numRecords) {
                  rec = Lrecords[i];
                  if (rec.quantity > 20) q1agg.sum1+=rec.discount;
                  i++ ;
            }}}
```

(b) Compiled Plan

Figure 3.2: JAMDB produced interpreted plan compared to compiled plan. [RPML06]

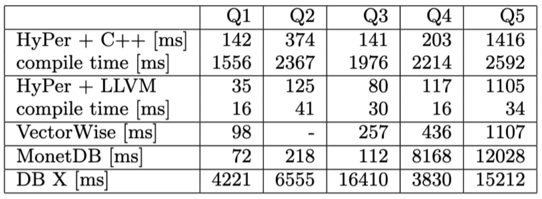
Depending on the query, their results show JIT is 2-4x faster than interpreting, and that on average their interpreted plans have twice as many memory accesses on primary, L3 and L2 memory, and over ten times as many branch prediction failures [RPML06]. The ending note is that while these results are promising, this is a research database which is missing persistence, locking, logging, recovery and space management. However, the fundamental issue is that developing a compiled executor is significantly harder to maintain, and they recommend the future work should be about simplifying making this type of database in the future [RPML06].

The issue with their comparison is that it is difficult to implement the compiler and interpreter to an equal quality. There are small things that can be improved or omitted; the Java compiler is a production-ready tool whereas the interpreted engine is handwritten by their team.

3.4 HyPer

HyPer is considered to be the pioneer in the field of JIT in databases because it is commercialised. It has enough features implemented for it to be used on a commercial scale, and showed that it is possible to use JIT on that scale. The project started in 2010, with their flagship paper releasing in 2011 for the compiler [Neu11], in 2016 they were acquired by Tableau [Tab18], and in 2018 they released another flagship paper about adaptive compilation [KLN18]. The database being commercialised does pose issues for outside research as the source code is not accessible, but they released a binary on their website that can be used for benchmarking.

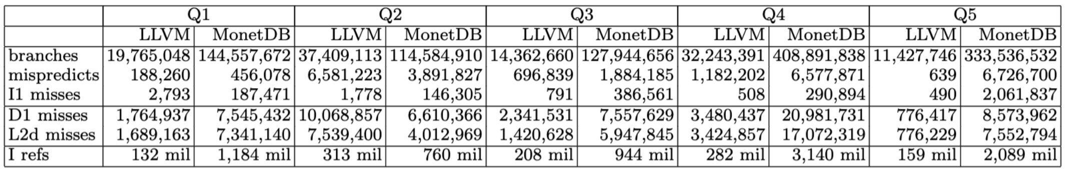
In their 2011 paper on the compiler, the HyPer team identified that translating queries into C or C++ introduced a significant overhead to their compilation latency compared to translating to LLVM. The paper proposed using pre-compiled C++ objects of common functions which are tied together using generated LLVM IR. This LLVM IR code would be executed using the LLVM's JIT executor. Using LLVM IR allows them to produce better code than C++ because they could configure specific features like the overflow flags, and it is also strongly typed which they mention prevented many bugs that was inside their original C++ [Neu11].



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

Figure 3.3: HyPer OLAP performance compared to other engines. [Neu11]

They managed to reduce their compile times by several factors as shown in Figure 3.3. Furthermore, in Figure 3.4 they show that they achieve many times less branches, branch mispredicts, and other measurements compared to MonetDB. This is due to HyPer's output having less code in its compiled queries, which results in less hazards being encountered during runtime [Neu11].



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

Figure 3.4: HyPer branching and cache locality benchmarks.
[Neu11]

In a later paper in 2018, HyPer improved their compile time by splitting their compiler into multiple stages. They used an interpreter on byte code generated from LLVM IR, on the next stage they can run unoptimised machine code, and on the final stage they can run optimised machine code. In Figure 3.5 this is visualised alongside compile times of each stage. Note also that they made the byte code interpreter themselves to be able to do this [KLN18].

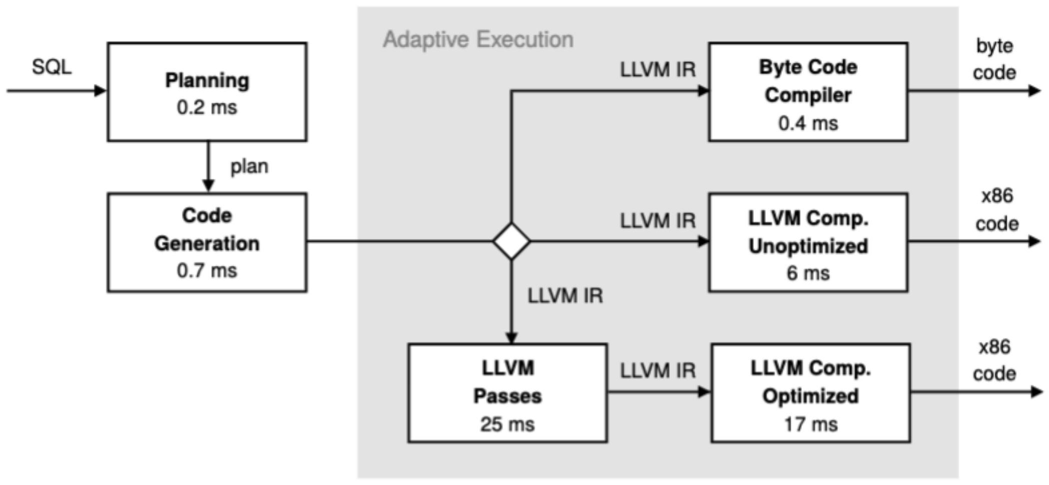


Figure 3.5: HyPer execution modes and compile times.
[KLN18]

Another adjustment in this paper is that they improved whether the engine optimised their queries. They argue that the optimiser's cost models as well as cardinality estimates are often inaccurate [LGM$^+$15] [LRG$^+$17]. Instead, they opt to split their execution into jobs for workers, and during the work-stealing stage they log how long the job took. With the query plan and these figures, they calculate accurate estimates for jobs and the optimal level to compile them to.

They benchmark TPC-H Query 11 using 4 threads, and it shows that their adaptive execution beats the alternatives of only using byte code by 40%, unoptimised compilation by 10% and optimised compilation by 80%. This is because their compilation stage is single threaded, so in the context of four threads the adaptive model can compile quickly on byte code, then while the byte code is running start compiling a better version on a single thread while the others continue to work [KLN18].

This combination of adding additional stages to their LLVM compiler, supporting multithreading between compiling and execution, and a more accurate cost analysis transforms their approach into a viable JIT application. A criticism with HyPer is that they are building a new JIT compiler from the ground-up which required significant engineering effort. In fact, most of the additions here are not unique to a database's perspective of its compiler, but are mostly ways to improve the compiler's latency. If they chose a compiler that focused on compilation latency, this might not have been an issue. However, this typically goes against the philosophy of JIT, and will be explored further in Mutable in Section 3.7.

3.5 Umbra

Umbra was created in 2020 by the same person that created HyPer, Thomas Neumann, and in their initial paper they take modern ideas from in-memory database and apply them to an on-disk database. They reason that the recent improvements in SSDs has made it possible to create an on-disk database which can achieve the same speeds as an in-memory database. It takes ideas from LeanStore for buffer management and latches, multi-version concurrency control from HyPer, and the compilation and execution strategies from HyPer as well. All of this produced a database which is more flexible, scalable, faster than HyPer [NF20] as seen in Figure 3.6.

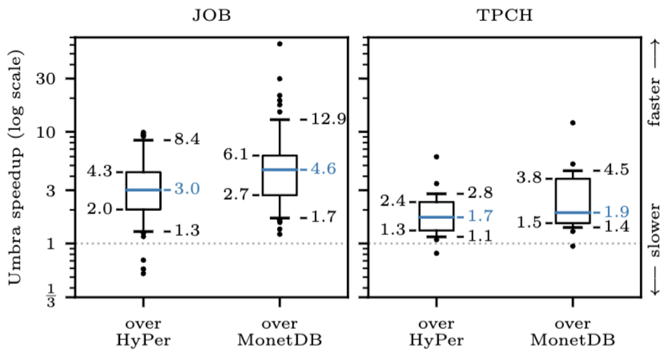


Figure 3.6: Umbra benchmarks.
[NF20]

Following this, they used Umbra as a base for several other research ideas. A key one is that they introduce a multi-level debugger [KN20]. It is not viable to attach a debugger to the database in the same way that volcano models do, because the engine compiles it into a whole new program. Attaching it to the SQL with markers added to the IR would also obfuscate how the operators work. To remedy this, they developed a time-travelling feature that replays a recording of the code generation to let them see the context [KN20].

In another paper, the approach of making their compiler have its own JIT stages has benefits. The ability to change query optimisation choices in the middle of their execution time is added - such as swapping the order of two joins. This can be particularly useful when their cost model makes an inaccurate choice, and they find that it improved the runtime of data-centric queries by more than two times [SFN22].

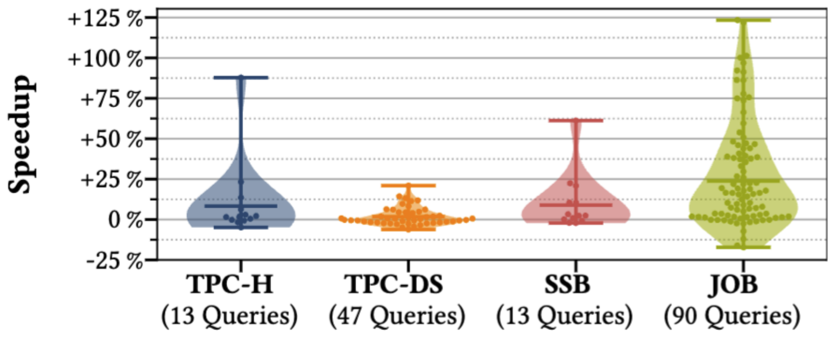


Figure 3.7: Umbra benchmarks after adaptive query processing (AQP). [SFN22]

In other databases, this feature is added by invoking the query optimiser multiple times; either during runtime, or at the start with the ability to swap between the plans. Other systems use routing where the plan itself has swapping between optimisations built into the plan itself. Since Umbra is compiled, and compilation has a high overhead cost, they opted for the second design. This also synergises well with using a JIT compiler. If the query changes an optimisation choice early in, it only needs to compile the correct choice during runtime [SFN22]. In Figure 3.7 they show a consistent improvement compared to before this change.

Overall, Umbra is rated as the most effective database in Clickhouse's benchmarks [Cli24], and is currently the most exhaustive implementation of JIT inside a database. It does still have the issue of engineering around a seemingly poor choice for a compiler, however, they also take advantage of having access to compilation choices made by the compiler.

3.6 PostreSQL

Firstly, when it comes to PostgreSQL majority of their design decisions are not made in the context of researchers writing an analysis that states something is worth doing with heavy empirical evidence. This is because it is open source software, and a number of features take the philosophy of: if someone is willing to do it, and it's either optional or clearly positive, it may as well be added. As a result, many of the sources about PostgreSQL are not going to be from peer-reviewed research papers. Most of the sources here will be from their documentation, pull-requests, and online articles.

With that being said, PostgreSQL is impossible to ignore because is the most popular database. In 2024, 51.9% of professional developers in the Stack Overflow survey said that they have done extensive work with it in the last year. A large fear in the database research community is that researchers are losing connection with their customers, so it is important to investigate [Sto18].

Before PostgreSQL added JIT, there was significant discussion about HyPer and JIT compiled queries in 2017 [She17]. People mostly responded with doubt that someone is going to add support for full query expression compilation or changing their model to a push-based one. Furthermore, rearchitecting a core component introduces significant risk.

JIT was added by Andres Freund to PostgreSQL, and he reasoned that they can benefit from JIT for CPU-heavy queries by only compiling expressions (the `x > 5` in `SELECT * FROM tbl WHERE x > 5;`), and that tuple deformation benefits significantly as well. He claims that generally when a query is CPU-heavy, it's because of expressions. Also, that tuple deformations interact with the cache, and have poor branch predictions in his benchmarks. A notable response in the pull request is in Figure 3.8 where Peter Eisentruat asks if the defaults are too low.

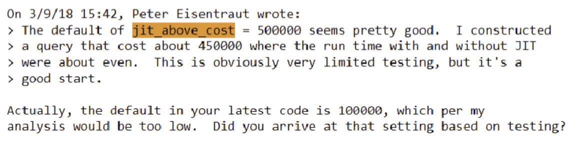

```
On 3/9/18 15:42, Peter Eisentraut wrote:
> The default of jit_above_cost = 500000 seems pretty good.  I constructed
> a query that cost about 450000 where the run time with and without JIT
> were about even.  This is obviously very limited testing, but it's a
> good start.

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

This was released in PostgreSQL version 11, and then enabled by default in version 12, and has been enabled ever since (latest as of writing is 17) [Pos24]. There was discussion about the release inside the pgsql-hackers email thread where they justified that enabling it by default will give the feature far more exposure and testing [Fre18]. However, when it was released the public reception was to disable it by default, with the United Kingdom's critical service for a COVID-19 dashboard collapsing with a 70% failure rate [Xen21]. This was due to their team having automatic updates on their database, and the default settings on JIT caused their queries to run 2,229x slower in the dashboard [Xen21].

This shows that on open source projects the out of box experience for these features must keep the end user in mind, and not only be based on benchmarks improving a theoretical scenario. Due to this, it is likely that a substantial portion of the PostgreSQL community has started disliking the idea of JIT, and changing public opinions can be difficult.

3.7 Mutable

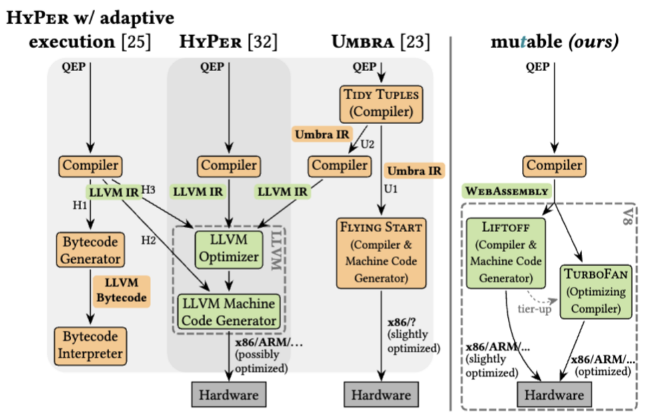
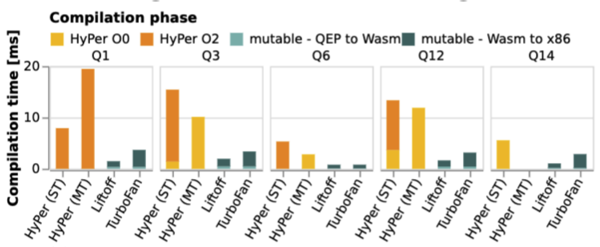


Figure 3.9: Comparison of mutable to HyPer and Umbra.
[HD23b]

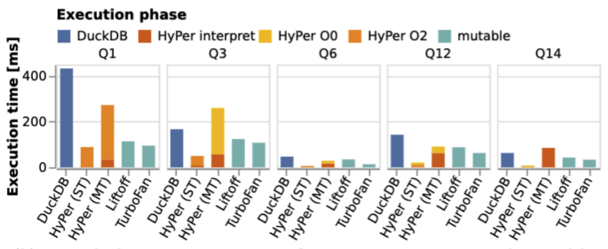
Mutable is a recent database which had its first publication in 2023, and its primary focus was being a minimal framework for research to build off of [HD23a]. It was created by the research team specifically with a JIT paper in mind. They reflected on HyPer and Umbra and came to the conclusion that most of their work was in the compiler, and what they need is to pick a compiler that prioritises latency instead [HD23a].

The architecture of mutable is compared to Umbra and HyPer in Figure 3.9. A side effect to note is mutable does not have a deeply developed query optimiser, and they rely on the V8 engine instead. Their pipeline consists of turning the SQL query into WebAssembly and feeding that into the V8 engine [HD23b].

In the V8 engine's official blog they explain that the "Liftoff" component is added to improve the compiler latency [Ham18]. Most JIT systems are created with the idea that only pieces of code that are executed with a high number of repetitions are worth compiling, but in the context of browsers that is the case [Ham18]. Inside the Liftoff stage the compiler focuses on producing machine code as fast as possible without optimisations and then runs "TurboFan", the second stage compiler which runs in the background while execution is going. It focuses on optimisations, scheduling, instruction selection, and register allocation [Ham18]. This is slightly different to HyPer, as HyPer has byte code interpretation as an option as well [KLN18].



(a) Detailed compilation times of HyPer and mu*t*able.



(b) Detailed execution times of DuckDB, HyPer, and mu*t*able.

Figure 3.10: Benchmarks produced by Mutable.
[HD23b]

The mutable team produced high quality benchmarks visible in Figure 3.10, and in their results they found that they get similar execution and compile speeds to HyPer, and in many cases they beat them. Primarily the times HyPer has a better overall time is when it has multithreading enabled.

The downside of this approach is if they want to optimise mutable further in the same ways HyPer or Umbra did with query re-planning [SFN22], they would likely need rearchitecting. However, their performance being comparable to HyPer with this much less work is significant.

3.8 LingoDB

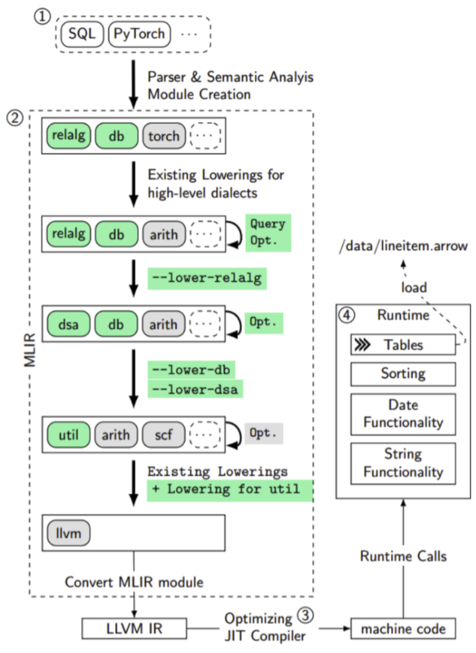


Figure 3.11: LingoDB architecture.
[JKG22]

LingoDB is a recent database with a novel approach of shifting the database query optimiser into the compiler itself by leveraging MLIR [JKG22]. This minimises how much translation they need to do between layers. Traditionally, developers need to parse the SQL into relational algebra operators, optimise on those, and parse the operators back into compiled code or an execution plan by hand. Since these shifts can be supported directly by the compiler itself, it means ensuring these translations are performant is less of a concern and that there is less manual implementation [JKG22].

An overview of the LingoDB architecture can be seen in Figure 3.11. They parse SQL into a relational algebra dialect inside MLIR, and they perform five optimisation techniques on that. This means it is not a mature optimisation engine, but shows that they can apply standard optimisation techniques within the compiler. Following this, they lower the declarative relational algebra that was produced into a combination of imperative dialects, which they can then lower to database-specific types and operations.

The outcome is that they are less performant than HyPer, but do manage to be better than DuckDB (an on-disk dataframe-based database which is widely used [JKG22]) as seen in Figure 3.12. However, their performance is not the impactful part of this research, the meaningful part is that LingoDB has approximately 10,000 lines of code in its query execution module [JKG22]. In comparison, Mutable is now at 22,944 lines of code (on doing a `git clone` of their public GitHub repository [HD23a]). In the LingoDB paper, they compare this to being three times less than DuckDB and five times less than NoisePage [JKG22].

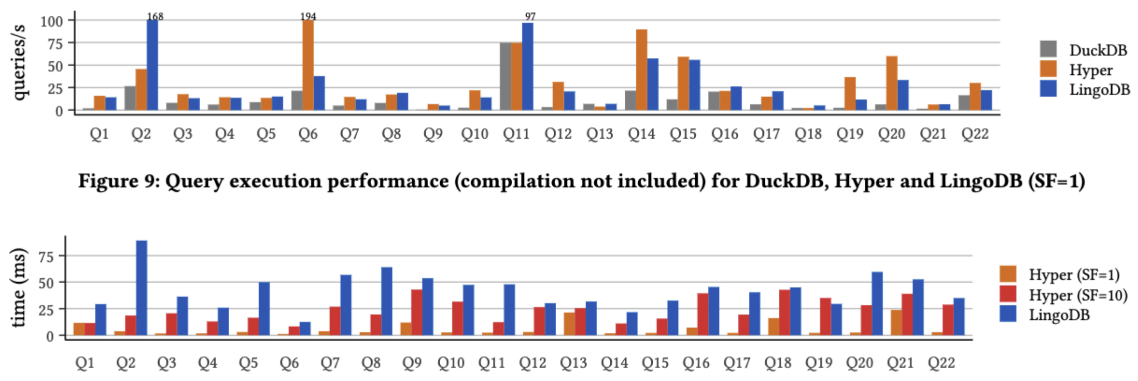


Figure 3.12: LingoDB benchmarking.
[JKG22]

Their main suggestion for future work is for supporting parallelising workloads and adding a dialect for distributed systems. In a later paper, they add lowerings for sub-operators for non-relational systems. Sub-operators consist of logical functions such as hashing, filtering, or user-added functions [JG23]. Umbra also added a similar feature to that before them [SN22]. The idea of LingoDB is novel and appealing, but as it requires rearchitecting the entire database pipeline to be inside the compiler, the idea is mostly not applicable to existing databases.

3.9 Gaps in Literature

The successful databases that were explored were HyPer and Umbra which managed to commercialise, and the other ones are research databases that in many cases did not support ACID, multithreading, or core requirements such as generating indexes on tables. The open source database, PostgreSQL, that did integrate JIT was mostly disastrous and caused an outage for a critical service in the UK. However, the potential for JIT is large as it can improve PostgreSQL's performance by multiple times.

Michael Stonebraker, a Turing Award recipient and founder of Postgres writes that a fundamental issue is that research has been forgotten by customers except for representatives of the 0.01% largest of database users [Sto18]. Commercial databases achieving performance many times faster than open source applications is a sign of this. In many real-world applications, the solution to your database being slow is not to tune your database, but to apply an application level cache either on a single server or distributed across a cluster with significant manual effort [Kle19]. This is largely because of the number of obscure ways that ACID requirements can be violated, and picking a slower, battle-hardened database is preferable [Kle19]. However, due to the popularity of PostgreSQL and other open source databases, improving them would still have a large impact.

Another issue is that writing a compiled query engine is difficult. This is one of the primary reasons developers opt for vectorised executions. There are not many debugging tools compared to stepping through a volcano or vectorised model. Umbra has suggested and created one specifically for their database, but it is questionable how transferable this is to other systems.

Comparing different compilers has been reasonably explored, but there are not many systems that explore MLIR. LingoDB recommends using MLIR to rearchitect the entire database to function around the toolkit, however, in the majority of applications recreating a database (such as Postgres) is not a viable approach without substantial investment.

Chapter 4

Project Plan

Solutions to the gaps in the literature will be explored by integrating MLIR in PostgreSQL on a query execution plan level. This would replace the volcano model entirely with an executable version. It supports the core customers by being a useable database, provides a toolkit that can make developing a compiler easier, and extends an existing system rather than creating a new database. The compiler will parse the output of the relational operators module inside of PostgreSQL and compile that into code, so the optimiser is still used.

While LingoDB has explored MLIR, they rearchitected the entire database design rather than fitting it into an existing one. Otherwise, MLIR is new enough that it has not been widely explored by the database community. There are other ideas that can sprout from this, such as using the V8 Engine from the LLVM IR that is produced, similar to Mutable, or additional lowerings for sub-operators similar to LingoDB.

4.1 Methodology and Approach

Integrating MLIR into PostgreSQL on a query expression level is a significant undertaking. Non-functional requirements are 1) The database should still fully work (including obscure functions like `WINDOW`) and 2) It should (ideally) strictly improve query performance without tuning when it is downloaded. Causing some queries to be worse while making the median or average query better is not a worthwhile trade off in this context due to wanting a positive out-of-the-box experience. However, ACID and muliprocessing will not be included as they are a significant amount of effort. The idea is to show that core idea is possible without breaking or rearchitecting the entire database.

Supporting the TPC-H-like benchmarks would make it possible to compare it to the existing literature. Mutable only runs benchmarks on a subset of TPC-H-like queries: 1, 3, 6, 12, and 14, while LingoDB supports all of them. They provided the minimal operators they needed to run these, and these projects can be used to find what is necessary for a running example.

The non-functional requirements can be supported by having a validation module. It will inspect the query and evaluate whether the system supports it and route it to the compiled method or the regular execution engine.

Including features such as multithreading and ensuring ACID compliance could make the project too long.

4.2 Thesis Timeline

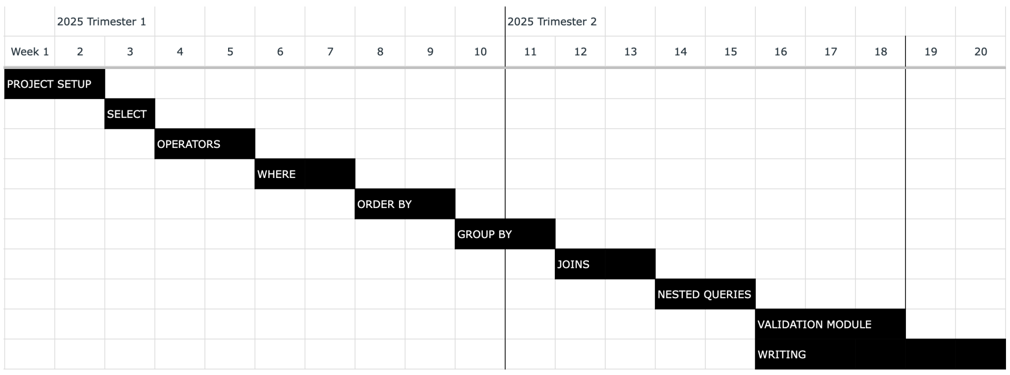


Figure 4.1: Thesis timeline

4.3 Justification of Time Allocation for Each Component

A specific goal is to have deliverables by the end of the first trimester. Being able to execute certain types of queries by then would show progress. The minimum for this is Query 6 in TPC-H, which requires `SELECT`, `sum`, `WHERE`, logical operators, some maths, and dates. A risk to consider is that PostgreSQL is vast, and it will take a while to find where things are. For instance, the nodes in the plan which comes from the query plan might have multiple types of nodes (sorts) which might mean the same thing. The project plan is based around SQL, not the specific relational algebra syntax PostgreSQL uses, so it might not completely align with the sections below. All the steps below have added ”fat” in their predictions so that there is space for blockers.

Setting up the project includes forking PostgreSQL, building from source, installing MLIR, investigating how the query tree would get parsed by MLIR, and the compiler boilerplate that will be required to support the queries. This also includes things like types - floats, integers, strings, and timestamps - and benchmarking. Two weeks might be too short for this, but it is fine if it is not completely done, and is done when necessary. Another stage here will be partially defining the context free grammars that are needed.

Implementing `SELECT` statements should be relatively straight forward after doing the project setup, as it is finding the entry in the database files and loading it into tuples. This also includes `as` for renaming columns. It is possible for work on the operators to start within the same week if there is time.

Operators include addition, subtraction, division, multiplication, `SUM`, `AVG`, and `COUNT`. There are a number of them, and particularly the projection of taking existing columns and making them into new columns will take time.

Operators will be fairly tightly coupled to `WHERE` statements which will include logical operators and comparisons. Once this is done, query 6 in the TPC-H benchmarks should be fully supported and be useable as a deliverable.

`ORDER BY` requires implementing a sorting algorithm, and might be difficult due to the importance of managing buffers. Similarly, `GROUP BY` needs to manage buffers and projecting the selection can take significant time. Once these two are implemented, several TPC-H queries will be runnable. It would be ideal if the first trimester can reach here, but that would require a low number of blockers.

`JOINS` will be a large amount of work, particularly in that some TPC-H queries have specific join statements like `LEFT OUTER JOIN`. Once implicit `JOIN`s are implemented a significant number of TPC-H queries will be runnable, and with nested queries nearly all of them will be. Nested queries might be simple in that the language specification could make it easy to implement these, however, they will lead to the database needing to store intermediate tables.

The validation module is also conceptually simple because it once it inspects whether the query is runnable and can get a speed improvement, it will feed it to the execution engine. Another option which could be explored here is putting in nodes that measure the volcano executor, and part way into execution interrupts it, measures whether it's worth compiling, and decides there. This can be a complicated task. It is also not critical to the thesis for this to be included, and in the worst case scenario could be cut out of the plan.

At the point of starting the validation module, time should start being spent writing the thesis. This is an early start, so that there can be space for thinking about structuring the writing and receiving feedback. In the last two weeks, time will be spent only on writing the thesis.

4.4 Required Training

The core tool being used is MLIR, which will need training to use. However, LingoDB has managed to be compatible with all 22 TPC-H queries with approximately 12,000 lines of code to parse and execute code. This excludes their lowering stage, and also provides examples of how to do this with MLIR.

Exploring PostgreSQL's codebase and understanding the operators the query plan has will likely take a significant portion of time. LingoDB parsed the SQL expression directly, whereas here we need to parse a query plan.

Another requirement is implementing benchmarking this against other reviewed databases. This will be explored directly in Chapter 5.

Other requirements are generic: programming familiarity, version control, databasing, compiler architecture, experience with long term projects, documentation and testing.

Chapter 5

Preliminary Results and Preparation

Extending PostgreSQL with MLIR is a large task. As a matter of fact, as mentioned in Section 3.6, compiling entire query expressions as been debated before in PostgreSQL discussions, and the replies were that the author will probably not do it. An issue is that showing the necessary skills to do this would take long - PostgreSQL's codebase is in the millions of lines, and MLIR tutorials are lengthy. Furthermore, there are no significant design choices to be made as this is almost a minimum viable product.

Specialised courses have been taken: COMP4403 at the University of Queensland on Compilers and Interpreters, as well as University of New South Wales' COMP9315, database implementations, where PostgreSQL itself is extended in coursework, and COMP6771, Advanced C++. Further, two years of experience working in a market maker Java and tuning its JIT compiler have been accumulated with over 80,000 lines of code written there.

This chapter is split into making an easily reproducible way to benchmark databases with Docker in Section 5.1, and then the potential for improvement inside of PostgreSQL is measured with `prod` in Section 5.2. Finally, a section of LingoDB is walked through at a very high level to understand its architecture in Section 5.3

5.1 Benchmarking reviewed databases

Recreating benchmarks from the reviewed studies makes it possible to directly compare the final results of the thesis. It provides a reference point for assessing the impact of changes made during the project, and demonstrates technical competence in producing these. Furthermore, many of these papers have not been independently validated, so reproducing their results serves as a confirmation of their findings.

An important requirement for this is that the results can be moved between systems, so Docker was chosen to containerise the applications. In many contexts, unless the user is a medium-sized corporation they will be using a form of containerisation while cloud hosting their database [Kle19]. Additionally, if everything has the overhead of containerisation then the results will stay fair between them.

TPC-H is a widely used benchmark on databases and was chosen [Tra24]. The mutable repository has a solid foundation for tools that support benchmarking [HD23a], so that was used to produce one docker container that supports HyPer, mutable, DuckDB, and PostgeSQL. LingoDB also had developed TPC-H tooling and was put into a separate docker container. Once the docker containers were created on a local machine, they were uploaded to a server on DigitalOcean with 128GB of RAM and 16 CPUs. Running `lscpu` showed it was a INTEL(R) XEON(R) GOLD 6548N. Then they were run with a scale factor of one (the database is approximately one gigabyte) with five repetitions of each query and put into Figure 5.1.

The results of these benchmarks in Figure 5.1 show that Postgres is significantly slower than the rest, and with Postgres removed from the graph HyPer and DuckDB with multiple cores are the fastest, and DuckDB is the slowest. The others contend for positions depending on the query.

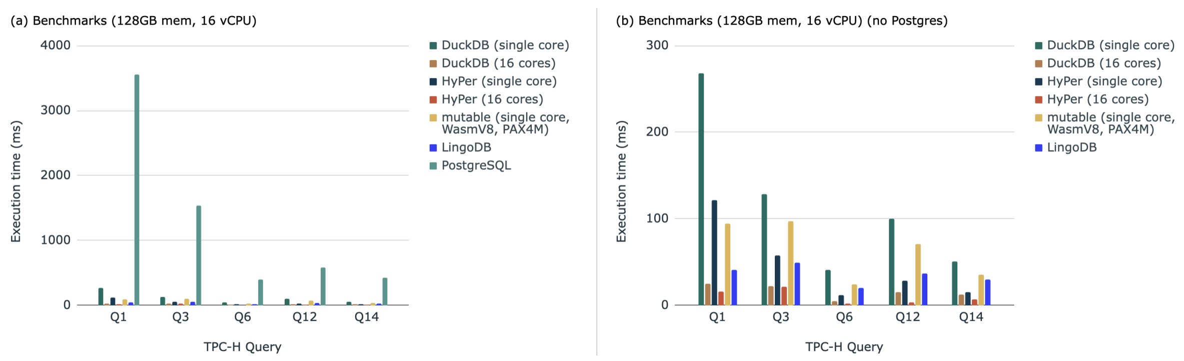


Figure 5.1: Benchmarking results.

The primary difficulties while building this was installing all the necessary libraries, as well as compiling things correctly. It was a significant time sink to prepare these containers. The Dockerfiles are available at `https://github.com/zelestis/benchmarking-dockers`.

5.2 Analysing how much of an impact JIT could have

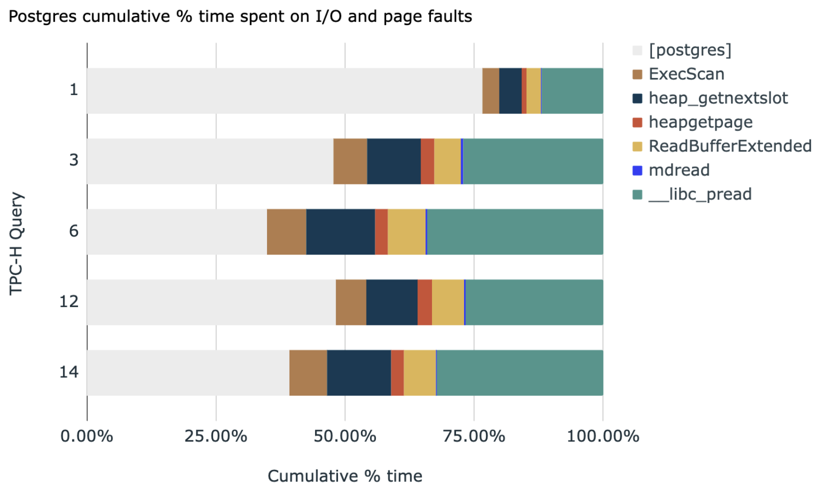
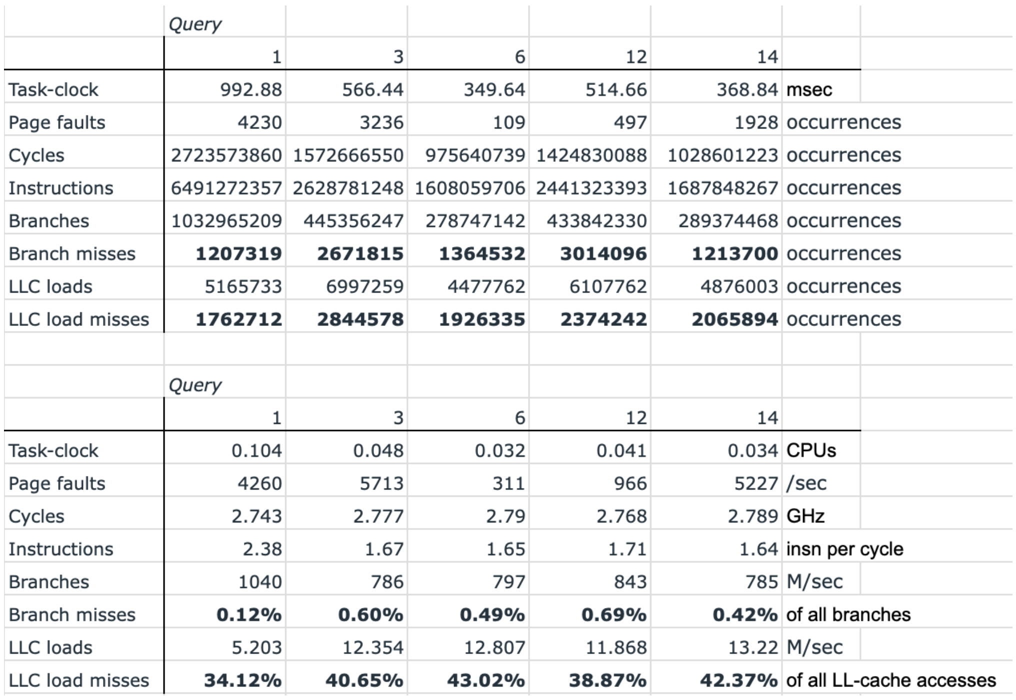


Figure 5.2: PostgreSQL time spent in the CPU, measured with prof.

A key requirement is there must be a visible amount of impact to improve PostgreSQL by. As such, the same set of queries as Section 5.1 were chosen to see how much of the time is spent on reading from secondary storage. The test was conducted on a local computer with a Samsung MZVLB256HAHQ-000L7 NVMe, 16 GB of RAM and an Intel i5-6500T. While running the tests from 5.1, `perf` was attached to the query and this data was given to `prof` [Linnd]. With the visualisations that `prof` produces, the key sections were identified and turned into a graph in Figure 5.2.

As seen in Figure 5.2, the CPU time for PostgreSQL varied from between 34.87% and 76.56% of the query with an average of 49.32%. The rest of the functions contribute to reading from I/O, and represent the profiler measuring samples inside the function (not cumulative of child methods). These methods were identified by observing the graph made by `prof` and finding all the child methods. The potential of JIT is to improve that CPU time by several factors, and we can expect a ceiling of improvement of between 2-3x in the case the CPU time reduces to 0% as a result of JIT.



	Query					
	1	3	6	12	14	
Task-clock	992.88	566.44	349.64	514.66	368.84	msec
Page faults	4230	3236	109	497	1928	occurrences
Cycles	2723573860	1572666550	975640739	1424830088	1028601223	occurrences
Instructions	6491272357	2628781248	1608059706	2441323393	1687848267	occurrences
Branches	1032965209	445356247	278747142	433842330	289374468	occurrences
Branch misses	**1207319**	**2671815**	**1364532**	**3014096**	**1213700**	occurrences
LLC loads	5165733	6997259	4477762	6107762	4876003	occurrences
LLC load misses	**1762712**	**2844578**	**1926335**	**2374242**	**2065894**	occurrences

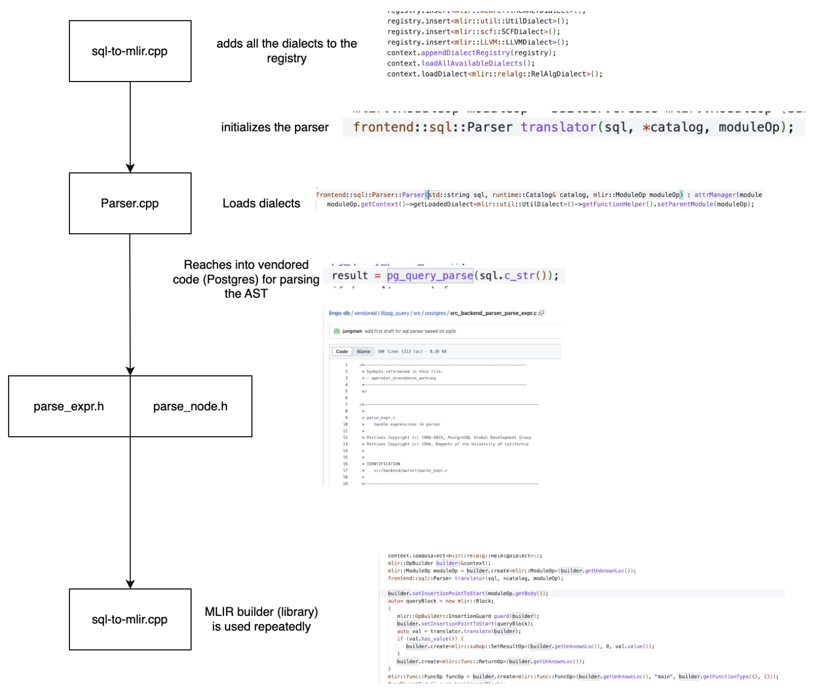
	Query					
	1	3	6	12	14	
Task-clock	0.104	0.048	0.032	0.041	0.034	CPUs
Page faults	4260	5713	311	966	5227	/sec
Cycles	2.743	2.777	2.79	2.768	2.789	GHz
Instructions	2.38	1.67	1.65	1.71	1.64	insn per cycle
Branches	1040	786	797	843	785	M/sec
Branch misses	**0.12%**	**0.60%**	**0.49%**	**0.69%**	**0.42%**	of all branches
LLC loads	5.203	12.354	12.807	11.868	13.22	M/sec
LLC load misses	**34.12%**	**40.65%**	**43.02%**	**38.87%**	**42.37%**	of all LL-cache accesses

Figure 5.3: PostgreSQL query profiling statistics collected with prof.

In Figure 5.3, we can see these queries had a worst case of 0.69% of branch predictions missed, and at worst a 43.02% of LLC loads are missed. A 0.69% could have room for improvement, as mentioned in Section 2.3, systems can have a 99% branch prediction accuracy but it's still a significant bottleneck. LLC loads are how many last-level-caches get loaded, and a miss rate of 43.02% means the cache could be significantly utilised.

5.3 LingoDB Codebase Exploration

This section is dedicated to exploring the LingoDB codebase at a high level for how it turns a query into machine code. Direct files with key points were explored, as visible in Figure 5.4. The code is accessible at `https://github.com/lingo-db/lingo-db` [JKG22].


sql-to-mlir.cpp
adds all the dialects to the registry
initializes the parser
Parser.cpp
Loads dialects
Reaches into vendored code (Postgres) for parsing the AST
parse_expr.h
parse_node.h
sql-to-mlir.cpp
MLIR builder (library) is used repeatedly

```
registry.insert<mlir::util::UtilDialect>();
registry.insert<mlir::scf::SCFDialect>();
registry.insert<mlir::LLVM::LLVMDialect>();
context.appendDialectRegistry(registry);
context.loadAllAvailableDialects();
context.loadDialect<mlir::relalg::RelAlgDialect>();
```

```
frontend::sql::Parser translator(sql, *catalog, moduleOp);
```

```
frontend::sql::Parser::Parser(std::string sql, runtime::Catalog& catalog, mlir::ModuleOp moduleOp) : attrManager(module
   moduleOp.getContext()->getLoadedDialect<mlir::util::UtilDialect>()->getFunctionHelper().setParentModule(moduleOp);
```

```
result = pg_query_parse(sql.c_str());
```

lingo-db / vendored / libpg_query / src / postgres / src_backend_parser_parse_expr.c

jungmair add first draft for sql parser based on pglib

Code Blame 300 lines (213 loc) · 8.39 KB

```
/*--------------------------------------------------------------------
 * Symbols referenced in this file:
 * - operator_precedence_warning
 *--------------------------------------------------------------------
 */

/*-------------------------------------------------------------------------
 *
 * parse_expr.c
 *    handle expressions in parser
 *
 * Portions Copyright (c) 1996-2015, PostgreSQL Global Development Group
 * Portions Copyright (c) 1994, Regents of the University of California
 *
 *
 * IDENTIFICATION
 *    src/backend/parser/parse_expr.c
 *
 *-------------------------------------------------------------------------
```

```
context.loadDialect<mlir::relalg::RelAlgDialect>();
mlir::OpBuilder builder(&context);
mlir::ModuleOp moduleOp = builder.create<mlir::ModuleOp>(builder.getUnknownLoc());
frontend::sql::Parser translator(sql, *catalog, moduleOp);

builder.setInsertionPointToStart(moduleOp.getBody());
auto* queryBlock = new mlir::Block;
{
   mlir::OpBuilder::InsertionGuard guard(builder);
   builder.setInsertionPointToStart(queryBlock);
   auto val = translator.translate(builder);
   if (val.has_value()) {
      builder.create<mlir::subop::SetResultOp>(builder.getUnknownLoc(), 0, val.value());
   }
   builder.create<mlir::func::ReturnOp>(builder.getUnknownLoc());
}
mlir::func::FuncOp funcOp = builder.create<mlir::func::FuncOp>(builder.getUnknownLoc(), "main", builder.getFunctionType({}, {}));
```

Figure 5.4: LingoDB sql-to-mlir.cpp high level stages.

The process begins with the main method inside `sql-to-mlir.cpp`. This creates an MLIR DialectRegistry, and adds all the dialects they have created to it. Following this, they parse the SQL expression with their Parser.cpp, which uses PostgreSQL libraries inside `parse_expr.h` and `parse_node.h`. This brings an abstract syntax tree for them to use back up to `sql-to-mlir.cpp`, and a `mlir::OpBuilder` is used in a loop to translate the query blocks through the dialects. At the end, they can use the produced result to produce LLVM IR.

Bibliography

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[LAB+21] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14, 2021.

[lat] llvm-admin team. The llvm compiler infrastructure. Accessed: 2024-11-16.

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

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

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

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

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

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

[OO20] Justin Onyarin Ogala and Deborah V Ojie. Comparative analysis of c, c++, c# and java programming languages. *GSJ*, 8(5):1899–1913, 2020.

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

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

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

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

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

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

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

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

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

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

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