Exploring Just-in-Time Compilation in Relational Database Engines

Contents

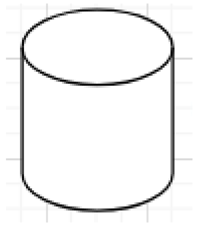
Contents

Importance of databases

1. Relational databases are used by almost every company – finance, technology, manufacturing, and healthcare are some industries.
2. It's widely accepted that in most contexts, the database is the bottleneck of the entire system. Some of these systems handle trillions of queries a day.*
3. Meaning, even small changes can save billions of dollars of electricity in data centres.
4. According to the 2024 Stack Overflow developer survey, the most popular database is PostgreSQL with 51.9% of developers using it extensively in the last year.**

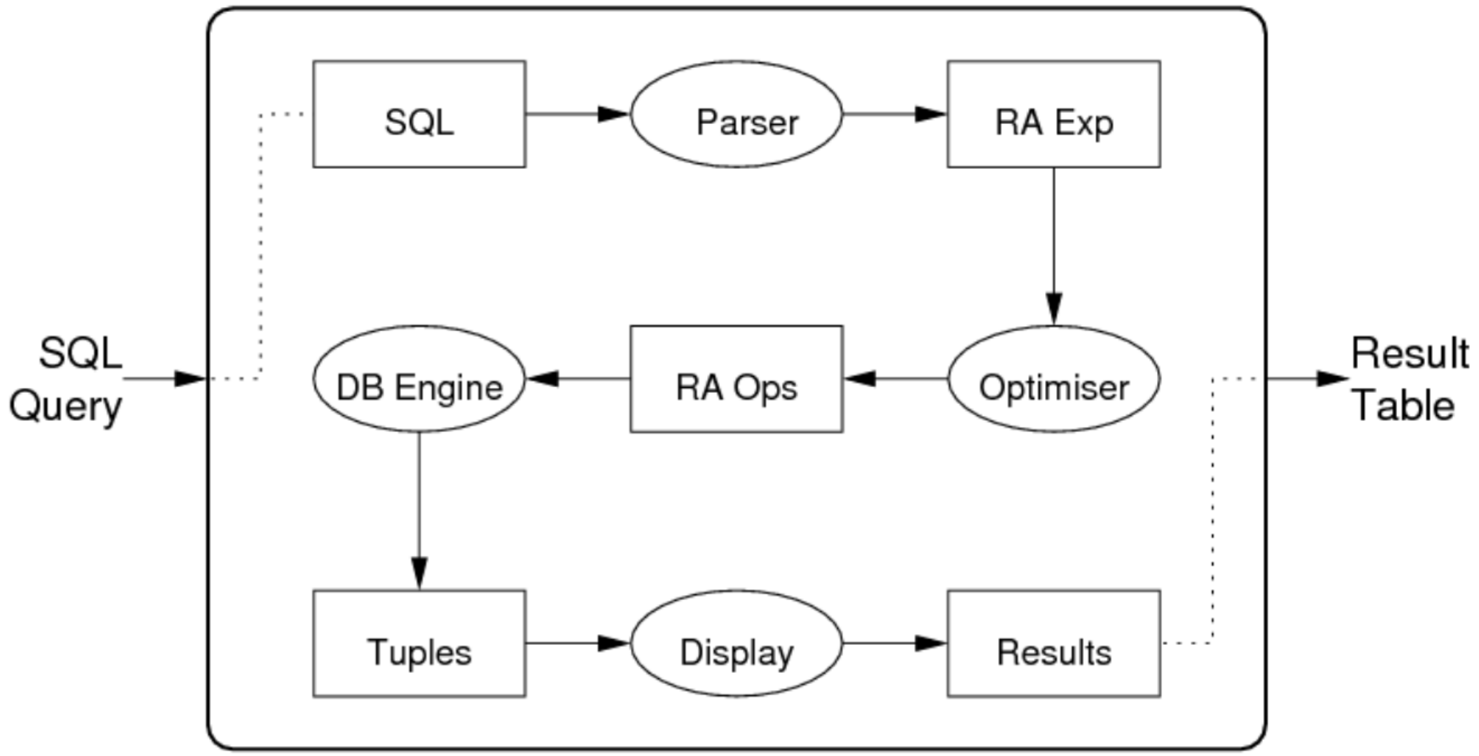
*Designing Data-Intensive Applications: Book by Martin Kleppmann

**https://survey.stackoverflow.co/2024/technology#most-popular-technologies-database-prof



Databasing Query Execution

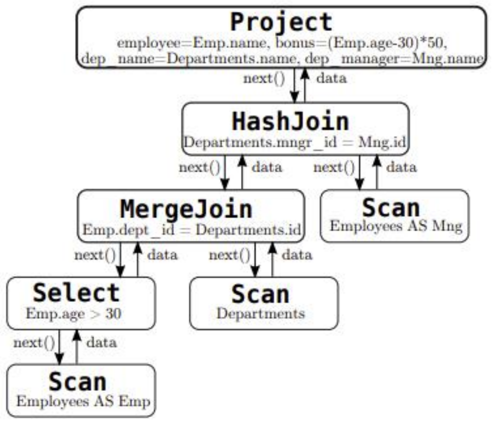
1. As a reminder, these are generally the steps that a database takes to run a query
2. Most of the execution time usually sits inside the DB Engine



Iterator model

1. Most popular database structure internally.
2. Also known as a Volcano model.
3. Works by having a main loop that calls next() on an iterator which calls the child.
4. For instance, Select would call next() on its child there and filter the result. It will only return the result if it satisfies its filter to the parent.
5. Isn't very effective at using caches because it only handles one tuple at a time.

$$\Pi_{\substack{employee=Emp.name,\\ bonus=(Emp.age-30)*50,\\ dep\_name=Departments.name,\\ dep\_manager=Mng.name}} \left(\begin{array}{c} (\sigma_{Emp.age>30}(\rho_{Emp/Employees}(Employees)) \bowtie_{\substack{Emp.dept\_id=\\ Departments.id}} \\ Departments) \bowtie_{\substack{Mng.id=\\ Departments.mngr\_id}} \rho_{Mng/Employees}(Employees) \end{array} \right)$$

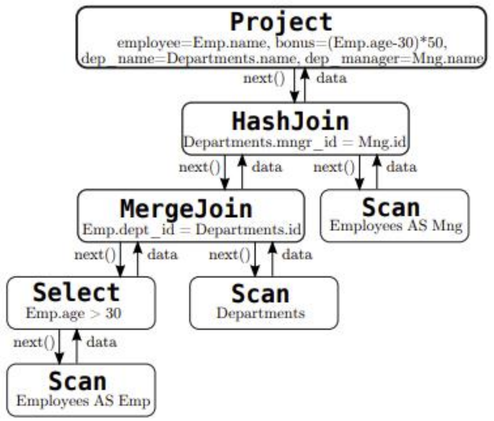


Query result			
employee	*bonus*	*dep_name*	*dep_manager*
Adam	450	Production	Adam
Tom	50	Production	Adam
Mark	1500	Sales	Mark
Chris	600	Sales	Mark

Vector Model

1. Issue is the iterator model barely uses the caches on CPUs, so people introduced the vector model
2. Instead of individual tuples on each next() function, they work on batches.
3. This reduces overhead of repeated function calls.
4. Mostly used in column-wise databases.
5. Has less flexibility and needs more RAM.
6. The big downside is that since it isn't pipelined anymore, it introduces a lot of copy operations for data.

$$\Pi_{\substack{employee=Emp.name,\\ bonus=(Emp.age-30)*50,\\ dep\_name=Departments.name,\\ dep\_manager=Mng.name}} \left(\begin{array}{c} (\sigma_{Emp.age>30}(\rho_{Emp/Employees}(Employees)) \bowtie_{\substack{Emp.dept\_id=\\ Departments.id}} \\ Departments) \bowtie_{\substack{Mng.id=\\ Departments.mngr\_id}} \rho_{Mng/Employees}(Employees) \end{array} \right)$$

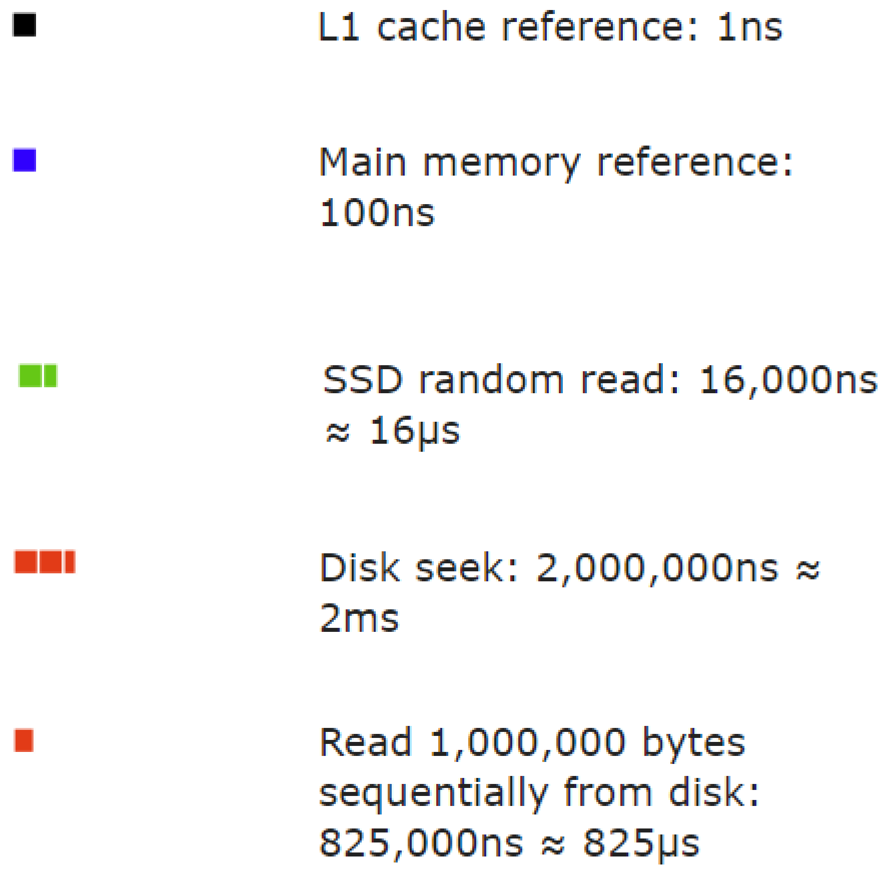


Query result			
employee	*bonus*	*dep_name*	*dep_manager*
Adam	450	Production	Adam
Tom	50	Production	Adam
Mark	1500	Sales	Mark
Chris	600	Sales	Mark

Issue with both models

1. Most databases consider all CPU operations to be O(1) time complexity – instant.
2. This is because accessing secondary memory is significantly slower than main memory.
3. Instead, we valued simplifying implementation complexity by opting for these models.
4. However, with in-memory databases some of them opted for compiling their queries with JIT.
5. The other thing to consider is databases are so heavily used that improving Postgres by fractions of percentages would save tremendous amounts of money.

Latencies on the right: https://colin-scott.github.io/personal_website/research/interactive_latency.html



L1 cache reference: 1ns

Main memory reference: 100ns

SSD random read: 16,000ns ≈ 16μs

Disk seek: 2,000,000ns ≈ 2ms

Read 1,000,000 bytes sequentially from disk: 825,000ns ≈ 825μs

What is Just-In-Time compilation (JIT)?

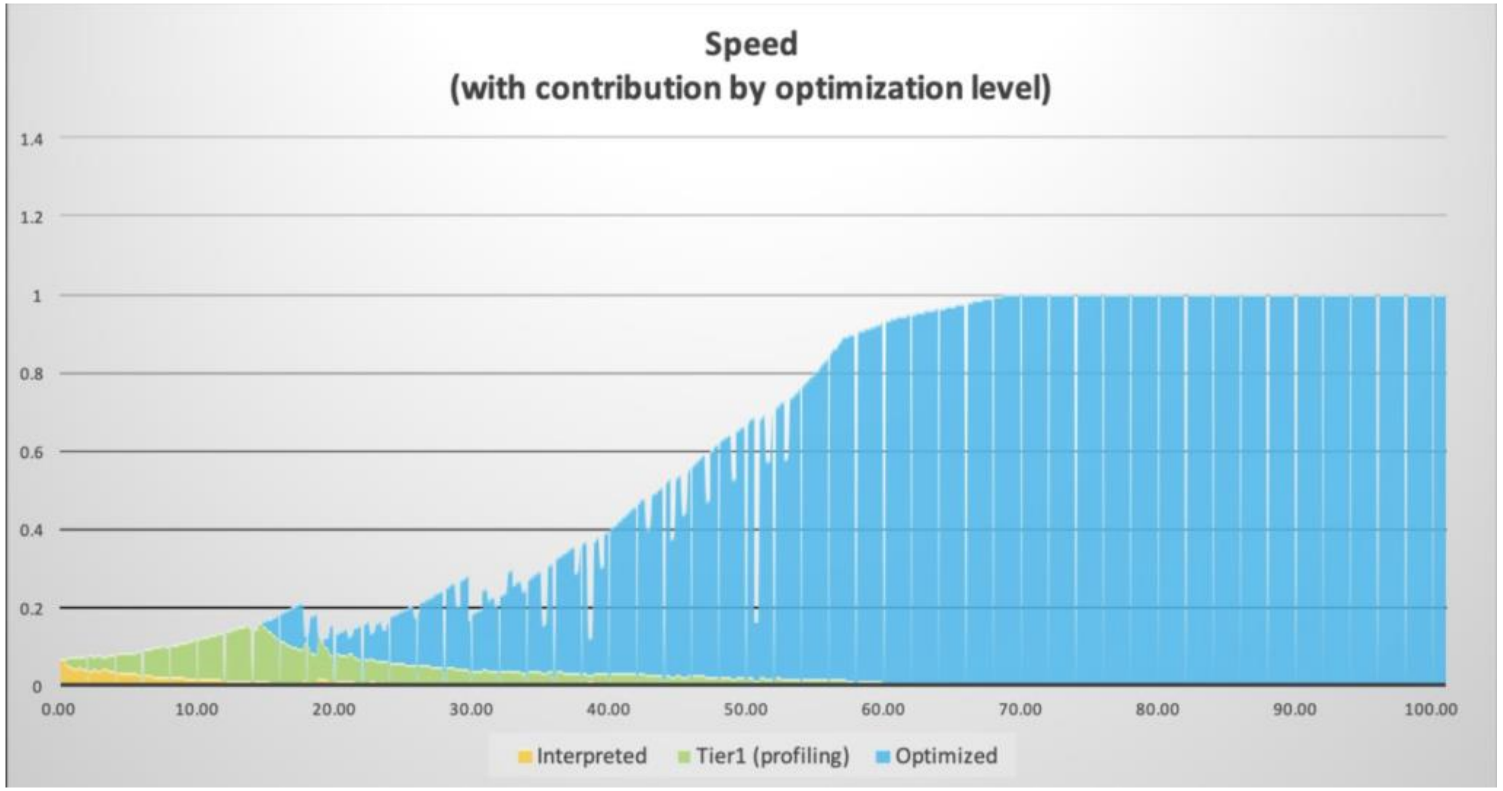
1. Interpreted languages (Python, JVM languages, C#) can be very slow.
2. Many get around this by using JIT, which triggers after a section of code has run a certain number of times.
3. When triggered, it compiles a chunk of code into machine code during execution of the program.
4. This includes all the generic compiler optimisations – branch prediction, cache locality, loop unrolling, method inlining.
5. There are situations where JVM code that has been optimised can be faster than C++ code because JIT has metrics about how the code runs. It’s very effective and impactful.

#5 https://stackoverflow.com/questions/4516778/when-is-java-faster-than-c-or-when-is-jit-faster-then-precompiled

How impactful is JIT?

1. The x axis is time, y axis is performance
2. Yellow is the first round of interpreted execution, green is a profiling round and blue is after optimisations
3. Azul (high performance java platform) quotes that JIT can lead up to a 50% more performance than ahead of time compilation.

From https://www.azul.com/blog/jit-performance-ahead-of-time-versus-just-in-time/



What other options are there than JIT?

1. There are some languages that simply don’t do JIT, like Python.
2. Others do give you the ability to do ahead of time compilation (AOT), like C#.
   1. This is a popular choice in UI development because there’s a reasonable chance the code paths will never be triggered enough for JIT, but still important enough for requiring optimization.
3. Hybrid compilation.
   1. In some languages, (Java), there is support to do ahead of time compilation on specific segments of code while leaving others as interpreted to be optimised with JIT later.

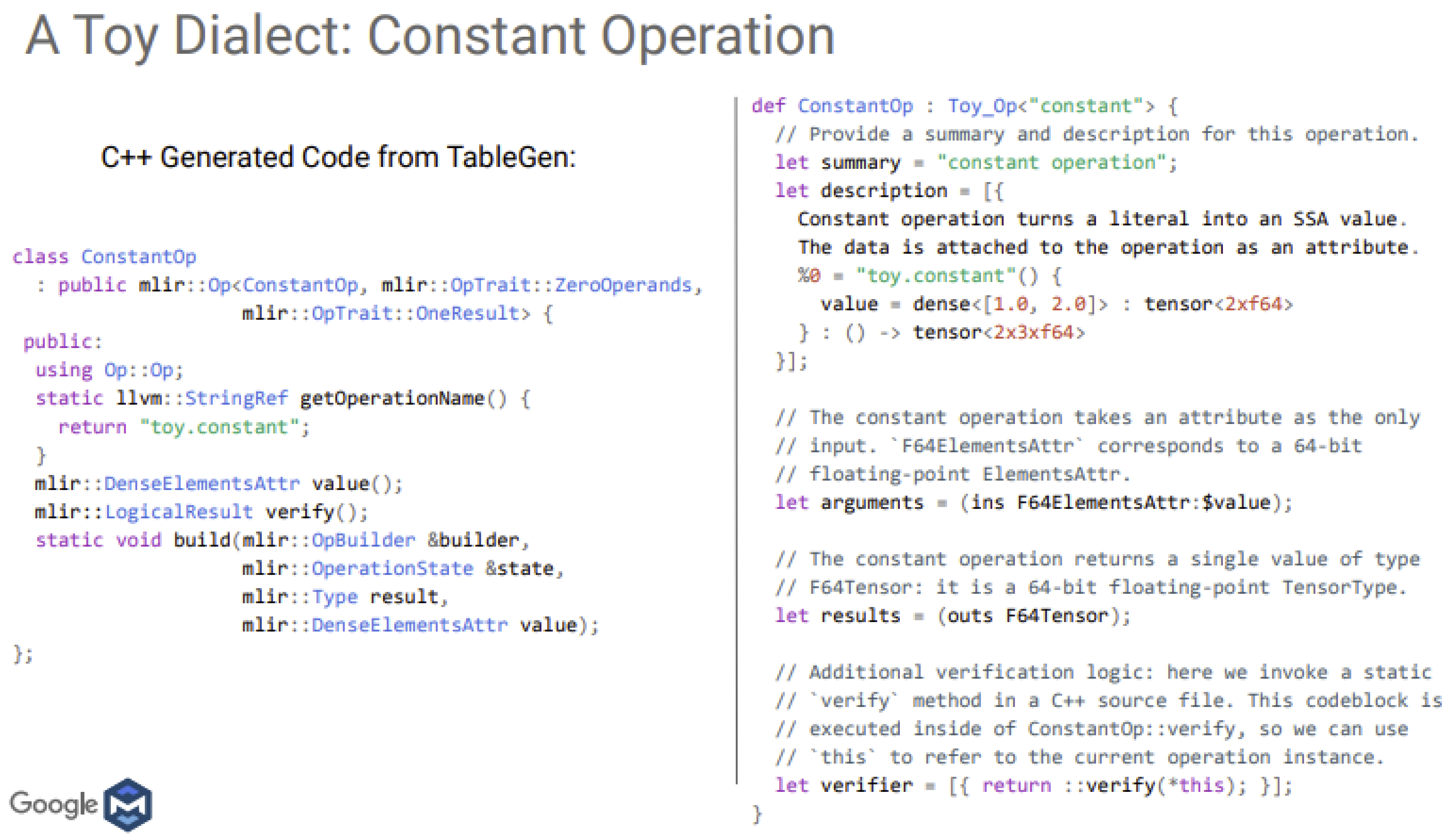
What tools support JIT?

1. Due to the difficulty of implementing JIT itself, most applications of it use some pre-existing tooling.
2. It’s generally accepted that the most developed JIT is in the JVM due to its age.
3. LLVM is another library that supports JIT, with Julia being a major user.
4. Some languages do still make their own JIT, like PyPy or WASM (WebAssembly).
5. OMR JitBuilder is made by Eclipse and JVM uses parts of it.
6. MLIR can also be used to turn into LLVM then use the LLVM JIT compiler, and other JIT compilers can use it.

What is MLIR?

1. MLIR supports building compilers by providing common infrastructure.
2. Most compilers end up rewriting common parts every time, so MLIR lowers the cost significantly.
3. The MLIR (the output of the code you write) is like a JSON representation of the operations you defined in C++.
4. Each entry represents operations, whereas typically compilers function in terms of instructions.
5. It can do rounds of “lowering” which enables just in time compilation.
6. We can dive into some short examples of MLIR.

Example of MLIR code



A Toy Dialect: Constant Operation

C++ Generated Code from TableGen:

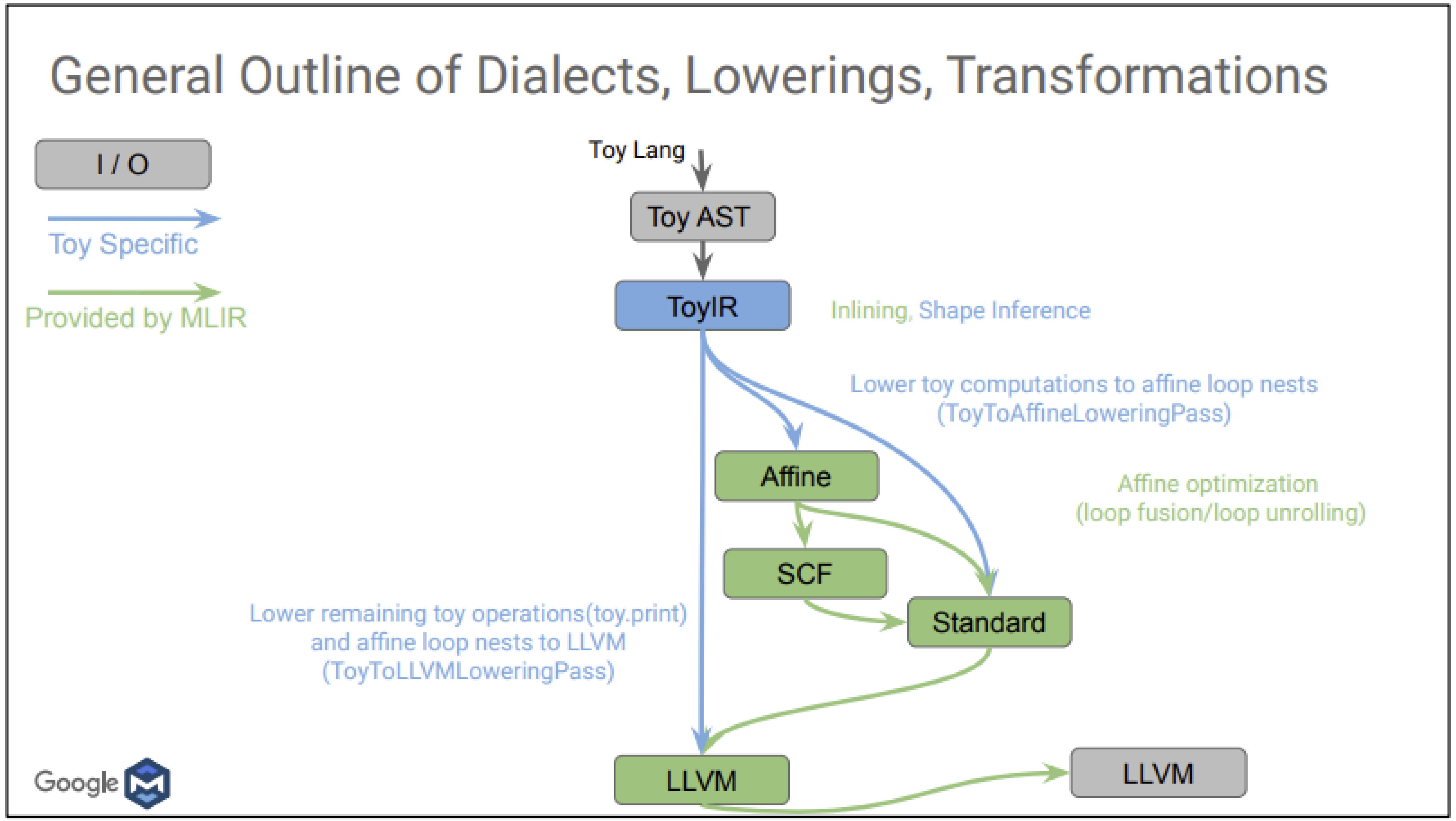
```
class ConstantOp
  : public mlir::Op<ConstantOp, mlir::OpTrait::ZeroOperands,
                    mlir::OpTrait::OneResult> {
 public:
  using Op::Op;
  static llvm::StringRef getOperationName() {
    return "toy.constant";
  }
  mlir::DenseElementsAttr value();
  mlir::LogicalResult verify();
  static void build(mlir::OpBuilder &builder,
                    mlir::OperationState &state,
                    mlir::Type result,
                    mlir::DenseElementsAttr value);
};
```

```
def ConstantOp : Toy_Op<"constant"> {
  // Provide a summary and description for this operation.
  let summary = "constant operation";
  let description = [{
    Constant operation turns a literal into an SSA value.
    The data is attached to the operation as an attribute.
    %0 = "toy.constant"() {
      value = dense<[1.0, 2.0]> : tensor<2xf64>
    } : () -> tensor<2x3xf64>
  }];

  // The constant operation takes an attribute as the only
  // input. `F64ElementsAttr` corresponds to a 64-bit
  // floating-point ElementsAttr.
  let arguments = (ins F64ElementsAttr:$value);

  // The constant operation returns a single value of type
  // F64Tensor: it is a 64-bit floating-point TensorType.
  let results = (outs F64Tensor);

  // Additional verification logic: here we invoke a static
  // `verify` method in a C++ source file. This codeblock is
  // executed inside of ConstantOp::verify, so we can use
  // `this` to refer to the current operation instance.
  let verifier = [{ return ::verify(*this); }];
}
```

1. On the right is a TableGen syntax which can generate a C++ class.
2. This is one essential part of the tools that makes MLIR powerful – the ability to define operators without the repetitive C++.

Sample from https://llvm.org/devmtg/2020-09/slides/MLIR_Tutorial.pdf

How MLIR works


General Outline of Dialects, Lowerings, Transformations
I / O
Toy Specific
Provided by MLIR
Toy Lang
Toy AST
ToyIR
Inlining, Shape Inference
Lower toy computations to affine loop nests (ToyToAffineLoweringPass)
Affine
Affine optimization (loop fusion/loop unrolling)
SCF
Standard
Lower remaining toy operations(toy.print) and affine loop nests to LLVM (ToyToLLVMLoweringPass)
LLVM
LLVM
Google

1. MLIR allows you to create custom dialects by doing “lowerings”.
2. The author would write the parser, then write the lowering to ToyIR, but from there MLIR provides the support for common lowerings.

Image from https://llvm.org/devmtg/2020-09/slides/MLIR_Tutorial.pdf

Contents

What are the different types of JIT choices in databases

1. Most applications of JIT in databases can be split into having JIT in Query Plan Execution (QPE) and Expression (EXP).
2. QPE-optimised databases are when you use JIT on the entire query plan.
3. At this stage, QPE mostly exists in research databases or in-memory databases.
4. EXP is when specific operators are compiled, like age > 30.
5. Only some databases have EXP jit-support, with the flagship one being PostgreSQL.
6. PostgreSQL also supports tuple deforming, which is transforming on-disk tuples into in-memory representations.

LLVM based JIT compilers in databases

1. **HyPer**
   1. The pioneer in the space. However, it’s hard to test them because they’re commercial.
2. Umbra
   1. Umbra is a close relative of HyPer that mostly followed the same implementation model for JIT.
3. **LingoDB**
   1. Uses MLIR and focuses on being a concise implementation of their idea.
4. Apache Impala
   1. One of the more established JIT databases, and brags a5x improvement due to JIT. It still lacks common features like nested schemas and indexes.
5. **PostgreSQL**
   1. Very established and strong support, but it only compiles expressions, and tuple transforms so it doesn’t fully use JIT.

JVM based JIT compilers in databases

Most of these are JIT-enabled more as a side effect of JVM rather than by choice, but their success shows that JIT is a viable approach

1. Apache Derby – a lightweight relational database purely created in Java.
2. Neo4J – The world's leading Graph Database.
3. PrestoDB – Created by Facebook, it specialises in data analytics.
4. Apache Spark – Large scale datasets database that's heavily used in data science.

I'm not going to dig too deeply into any of these here.

Other JIT databases

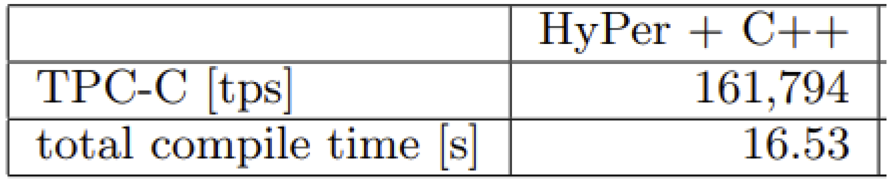
1. **Mutable** - Quite a promising and well-known JIT-supported research database. It compiles to WebAssembly so it’s also a unique JIT.
2. QuestDB – a time series database which uses “asmjit” for producing machine code.
3. RaptorDB – a key-value store for JSON documents which uses the .NET runtime framework.
4. BlazingSQL – A GPU-accelerated database that supports JIT through CUDA.

Case: HyPer

1. HyPer is an in-memory database developed by Technical University of Munich and was acquired by Tableau which is now part of Salesforce.

2. They are considered the pioneer in adding JIT to databasing. They were so early to this because they are an in-memory database, and it gave them large performance benefits.

3. In their first attempt they were converting their relational algebra in C++ then compiling it, but the compiler would take too long.



	HyPer + C++
TPC-C [tps]	161,794
total compile time [s]	16.53

Case: HyPer

1. Instead, they decided to write all the simple operations in LLVM directly, and tie pre-compiled C++ modules together through LLVM.
2. This dramatically dropped their compile times and improved their cache usage.
3. It is proprietary though, so benchmarking it is challenging.
4. In later iterations, they introduce adaptive execution which compiles the code while the interpreter is running

https://www.vldb.org/pvldb/vol4/p539-neumann.pdf

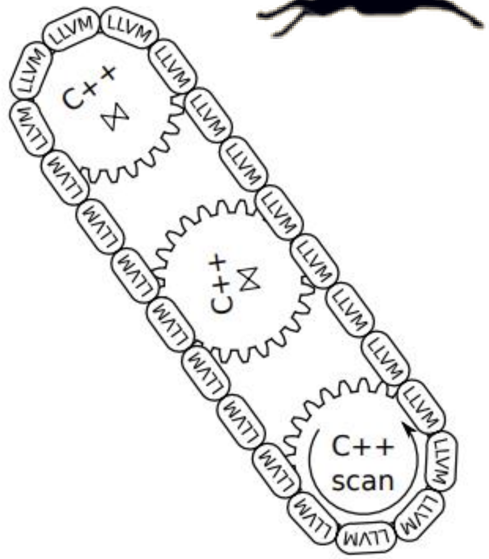
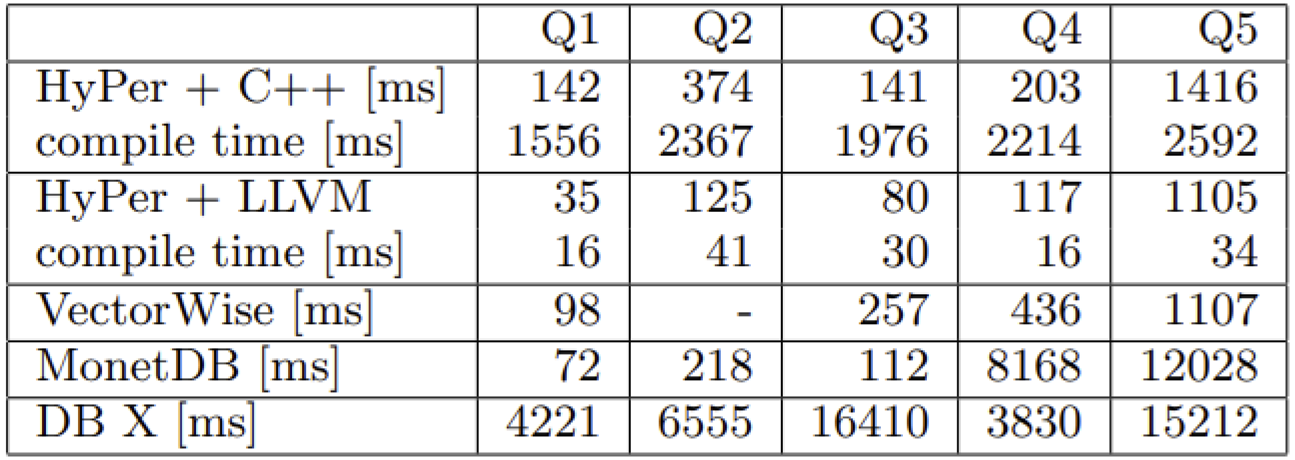


Figure 6: Interaction of LLVM and C++

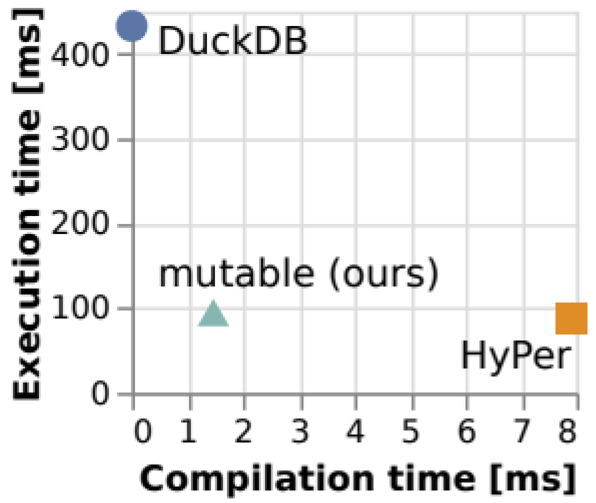


	Q1	Q2	Q3	Q4	Q5
HyPer + C++ [ms]	142	374	141	203	1416
compile time [ms]	1556	2367	1976	2214	2592
HyPer + LLVM	35	125	80	117	1105
compile time [ms]	16	41	30	16	34
VectorWise [ms]	98	-	257	436	1107
MonetDB [ms]	72	218	112	8168	12028
DB X [ms]	4221	6555	16410	3830	15212

Case: Mutable's use of JIT

mutable

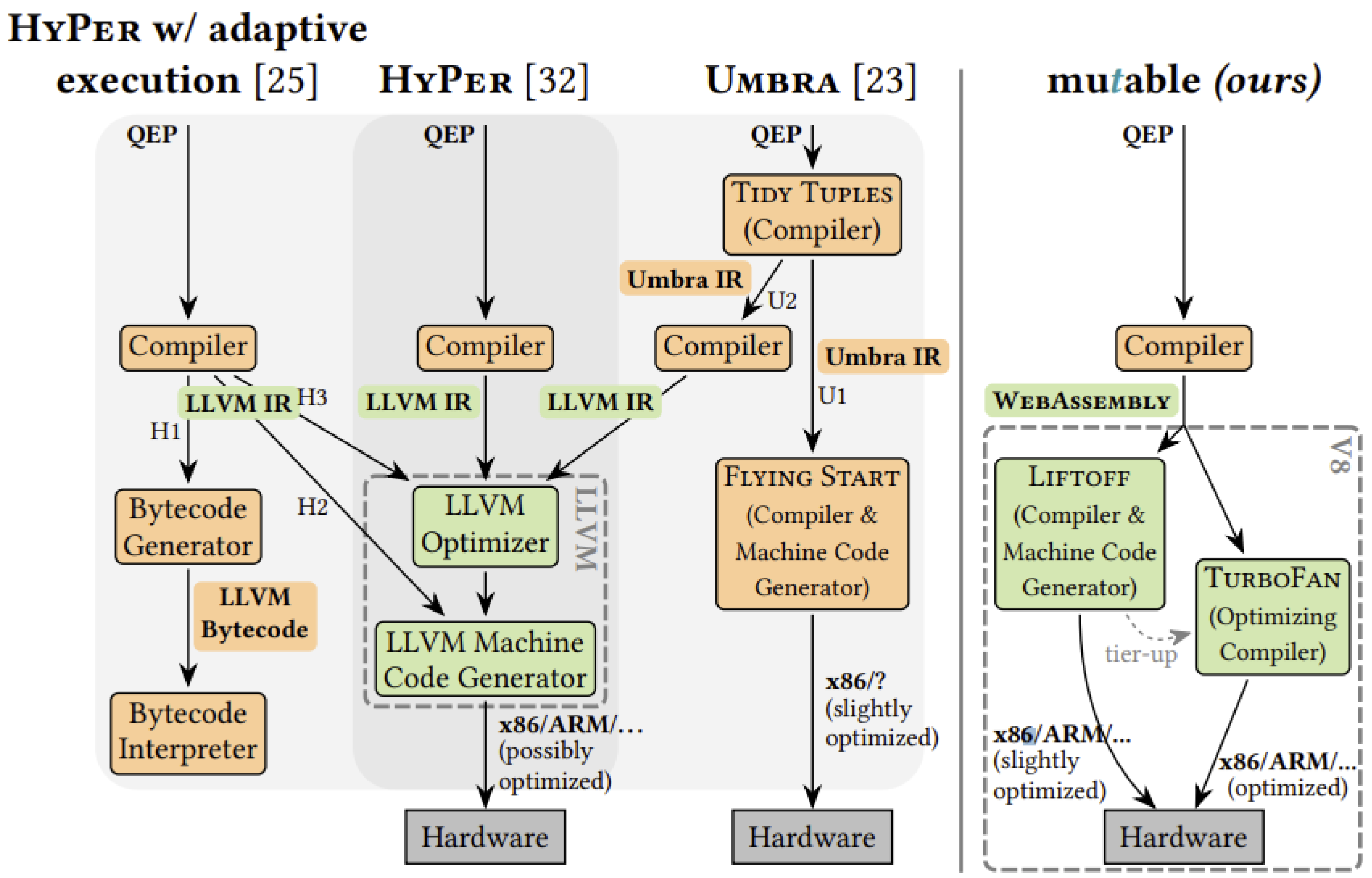
1. Mutable is a research-oriented database made for fast prototyping by Saarland University and was initially developed to support their idea for JIT.
2. Their innovation was using WebAssembly.
3. Previous solutions used LLVM, which isn't fundamentally made with JIT in mind, and then they add more engineering effort to support compiling while executing which they think adds too much complexity.
4. They argue that you can get better performance by picking a compiler that is built with JIT in mind



mutable

Case: Mutable's use of JIT

1. Hyper and Umbra both manually implemented large chunks of their compilers manually
2. Mutable does a purer form where they simply rely on web assembly's compiler



HyPer w/ adaptive execution [25]
HyPer [32]
Umbra [23]
mutable (ours)
QEP
Compiler
LLVM IR
H1
H2
H3
Bytecode Generator
LLVM Bytecode
Bytecode Interpreter
LLVM Optimizer
LLVM Machine Code Generator
LLVM
x86/ARM/... (possibly optimized)
Hardware
Tidy Tuples (Compiler)
Umbra IR
U1
U2
Flying Start (Compiler & Machine Code Generator)
x86/? (slightly optimized)
WebAssembly
V8
Liftoff (Compiler & Machine Code Generator)
TurboFan (Optimizing Compiler)
tier-up
x86/ARM/... (slightly optimized)
x86/ARM/... (optimized)

LingoDB

Case: Lingo-DB’s use of JIT

1. Lingo DB takes a different approach and uses MLIR
2. Reminder that MLIR supports custom lowerings that let you change the intermediary representation in stages
3. Instead of having their system parse the query into relational algebra, and the optimising this custom view, they do it with built in lowering infrastructure inside MLIR
4. Effectively, this turns their entire pipeline into a compiler
5. They argue the primary benefit of this is simplifying the development of the database

6 SYSTEM OVERVIEW

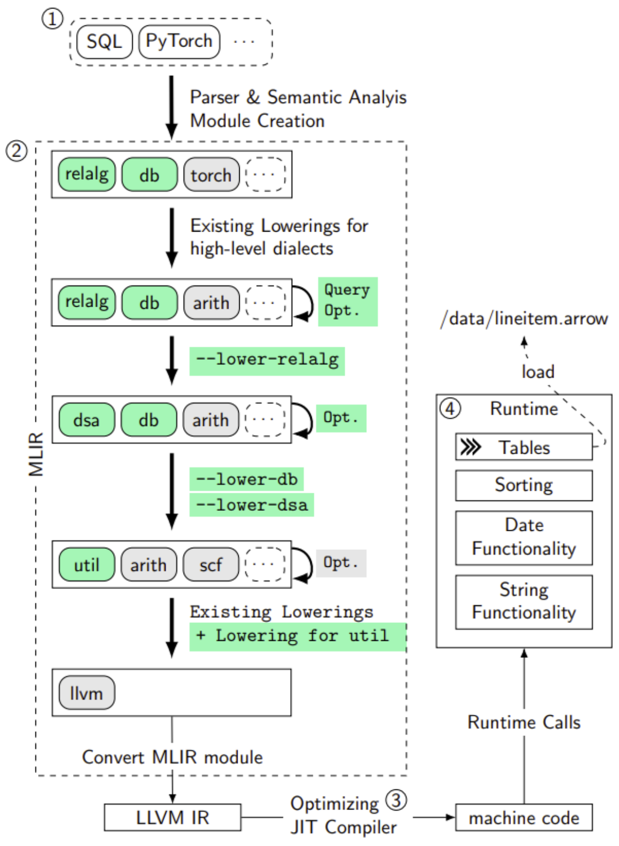


Figure 8: The developed prototype system consisting of ① a SQL frontend that performs parsing and semantic analysis, ② MLIR dialects, optimizations, and lowerings, ③ a LLVM-based optimizing JIT-compiler, and ④ a runtime system based on Apache Arrow

LingoDB

Case: Lingo-DB's use of JIT

1. They get solid benchmarks out of this, but also, their entire codebase is much smaller than other existing ones

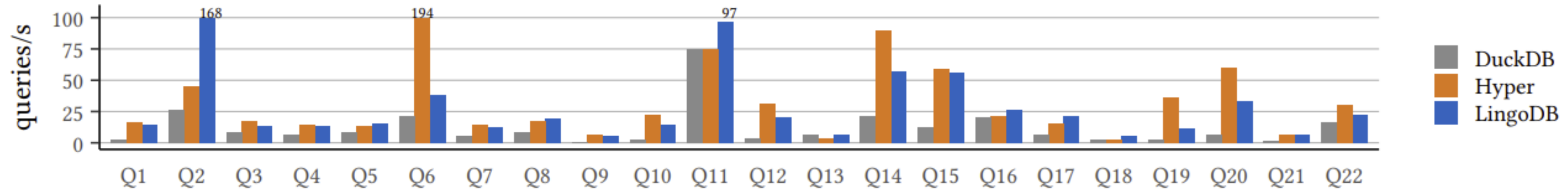


Figure 9: Query execution performance (compilation not included) for DuckDB, Hyper and LingoDB (SF=1)

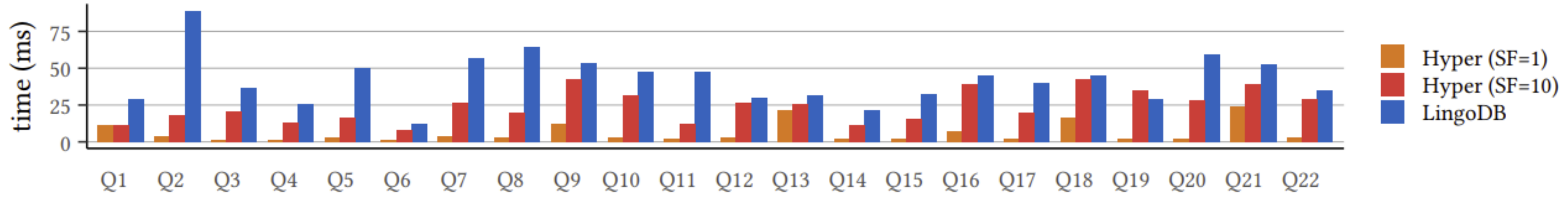
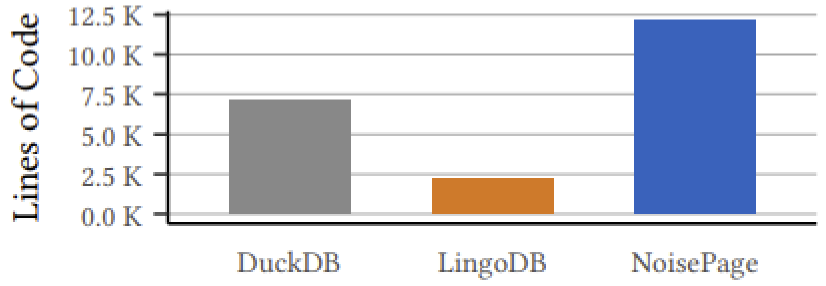


Figure 10: Total compilation times for Hyper and LingoDB.



Case: Postgres's use of JIT

1. Postgres introduced JIT in version 12 with support for query expressions and tuple deforming by using LLVM
2. Most of the user experiences that I found complained about it being unbeneficial, with it triggering very large costs
3. When it released, the UK coronavirus dashboard got a 70% failure rate on a critical service, reporting their queries becoming 2,229x slower
4. There's a hackernews user here that did ad-hoc benchmarks and found that it *does* manage to make their database faster *on average*, but it also makes a significant number of queries slower even after tuning it for an entire day
5. So why did they add it?

*2 https://blog.g-vo.org/taming-the-postgres-jit.html

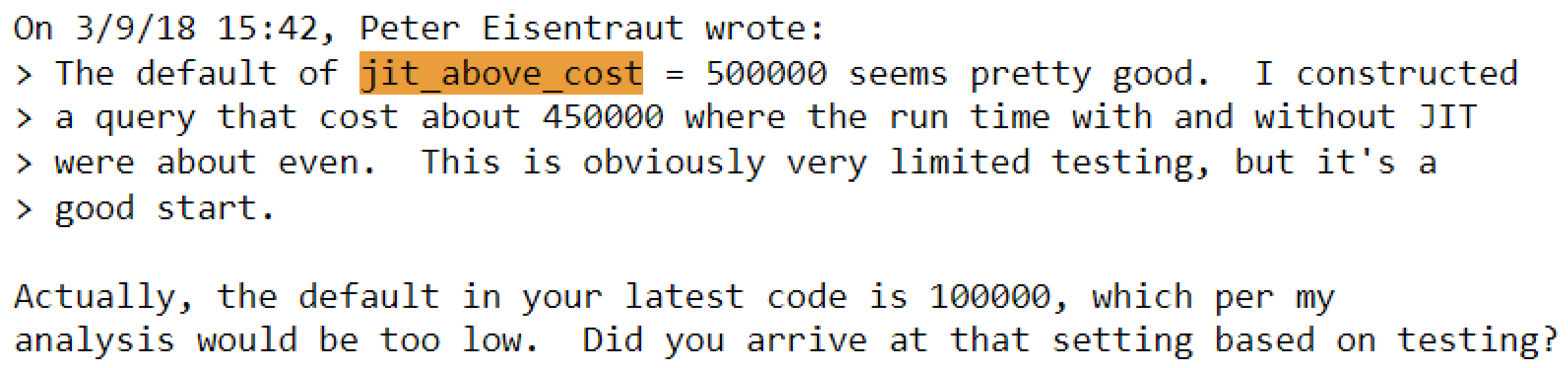
*3 https://dev.to/xenatisch/cascade-of-doom-jit-and-how-a-postgres-update-led-to-70-failure-on-a-critical-national-service-3f2a

*4 https://news.ycombinator.com/item?id=26223092

Case: Postgres's use of JIT

1. At this stage of researching JIT, you begin wandering outside of well-backed research articles.
2. Digging through why they chose to introduce this, Andres Freund suggested adding JIT back in 2016 https://www.postgresql.org/message-id/flat/20161206034955.bh33paeralxbtluv%40alap3.anarazel.de
3. He primarily suggested adding it to postgres for complex, cpu intensive queries as well as noticing tuple deforming is very slow and was met with significant discussion.
4. The discussions here are mostly about whether LLVM is the correct choice because it's a large dependency to add.
5. The big counter to most arguments were "it's an optional feature, and someone's already done most of the work, so it shouldn't hurt to add right?".

Case: Postgres’s use of JIT



```
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?
```

1. The interesting thing is Peter Eisentraut asked about how they arrived at their defaults, and this was never answered
2. The point of this is, it doesn’t seem the defaults around this feature were particularly deeply researched, and that also means there’s room for more JIT research in postgres
3. Since this was intended for very complex queries, it’s unlikely that TPC-H would represent this benefit accurately

Challenges faced when adding JIT

1. Complexity of the approach – HyPer’s approach is quite solid, however, improving it became very complicated because they were writing raw LLVM.
2. Requiring rewriting the entire database – Lingo DB’s approach is quite viable, but creating a whole database around this idea probably is daunting. PostgreSQL is around 2.5 million lines of code nowadays.
3. If integrating into a new database, not researching defaults enough – PostgreSQL caused global problems with their JIT patch.
4. If it is overhauling query expressions, maintaining ACID compliance is a new challenge.

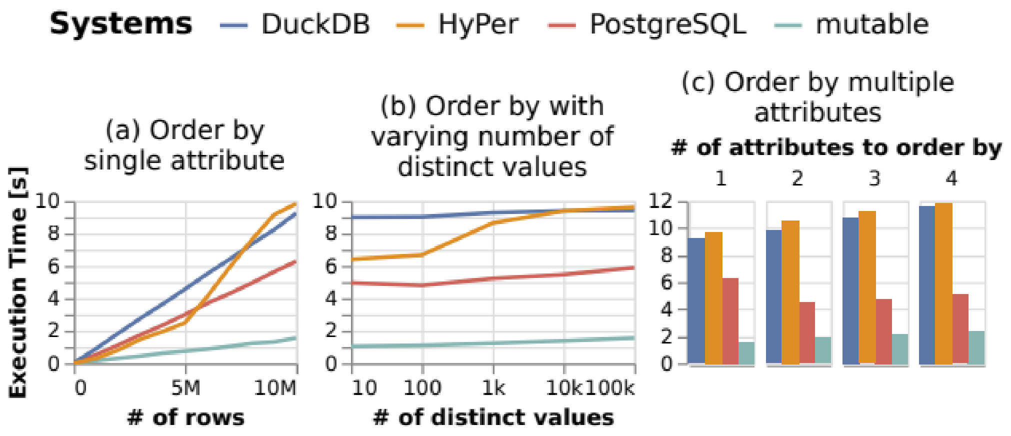
The big missing thing

1. Most of these approaches complain about JIT compile latency.
2. However, why can’t we simply cache the JIT compile objects? Most systems that use a database would be sending similar queries repeatedly, and in a realistic environment this would offer major benefits.
3. I heard this idea from https://news.ycombinator.com/item?id=39742916
4. A commenter mentions that actually, LLVM already has support for caching previously compiled objects, however, the way that PostgreSQL works simply doesn’t support this due to direct memory addresses.
5. It does seem that most TPC-H benchmarks only run the query once, and that really focuses on the initial compile time.

Contents

Current state based on papers

1. We can broadly see from the pictures we saw earlier that each paper effectively claims they are the fastest
2. The answer is it really depends between them, and some of them argue that their approach is significantly simpler
3. Let's try to benchmark these on our own



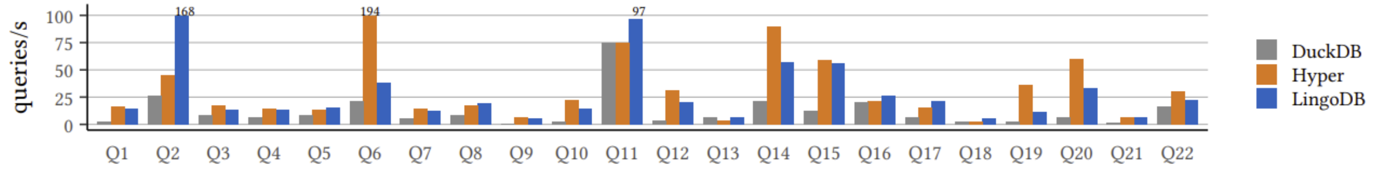


Figure 9: Query execution performance (compilation not included) for DuckDB, Hyper and LingoDB (SF=1)

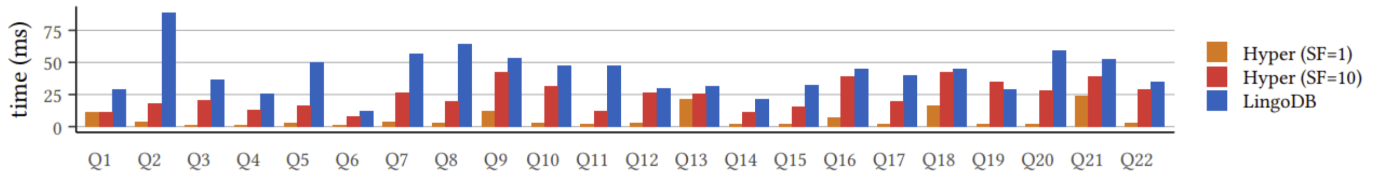


Figure 10: Total compilation times for Hyper and LingoDB.

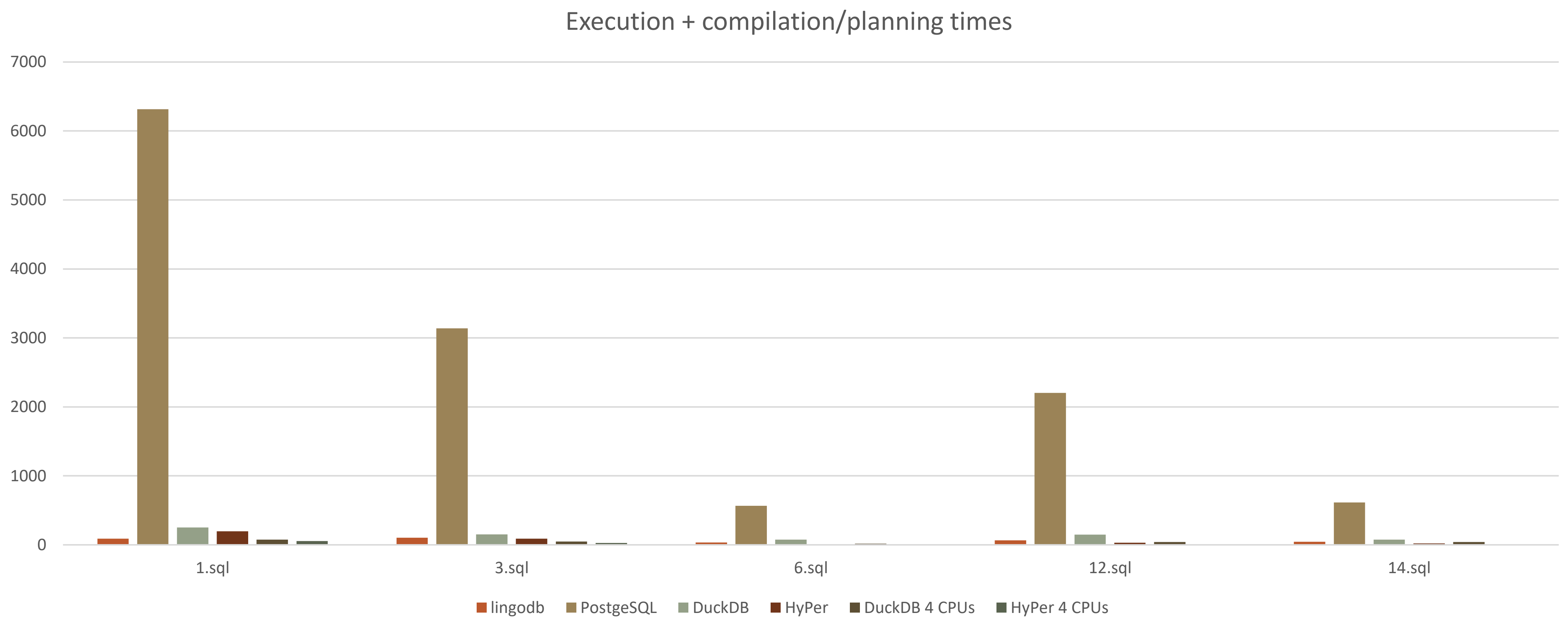
Benchmarking method

1. Keep in mind that most databasing teams allocate a significant amount of time for benchmarking, so this isn't as easy as press install and run.
2. Most databases look at TPC H as the main set of queries to benchmark, but even then, methods for how to do this varies.
3. I spent a significant amount of time trying to get other tools to work for me (DBT3, HammerDB, sysbench), and realised that I want to run this on multiple systems. My local small server as well as the research server (tods1).
4. Most of these common tools also only implement benchmarks for common databases, and they are particularly painful to extend because they focus on turning it into a continuous benchmarking system.
5. It can take in the range of hours to do all the preparation to run things though.

Benchmarking method

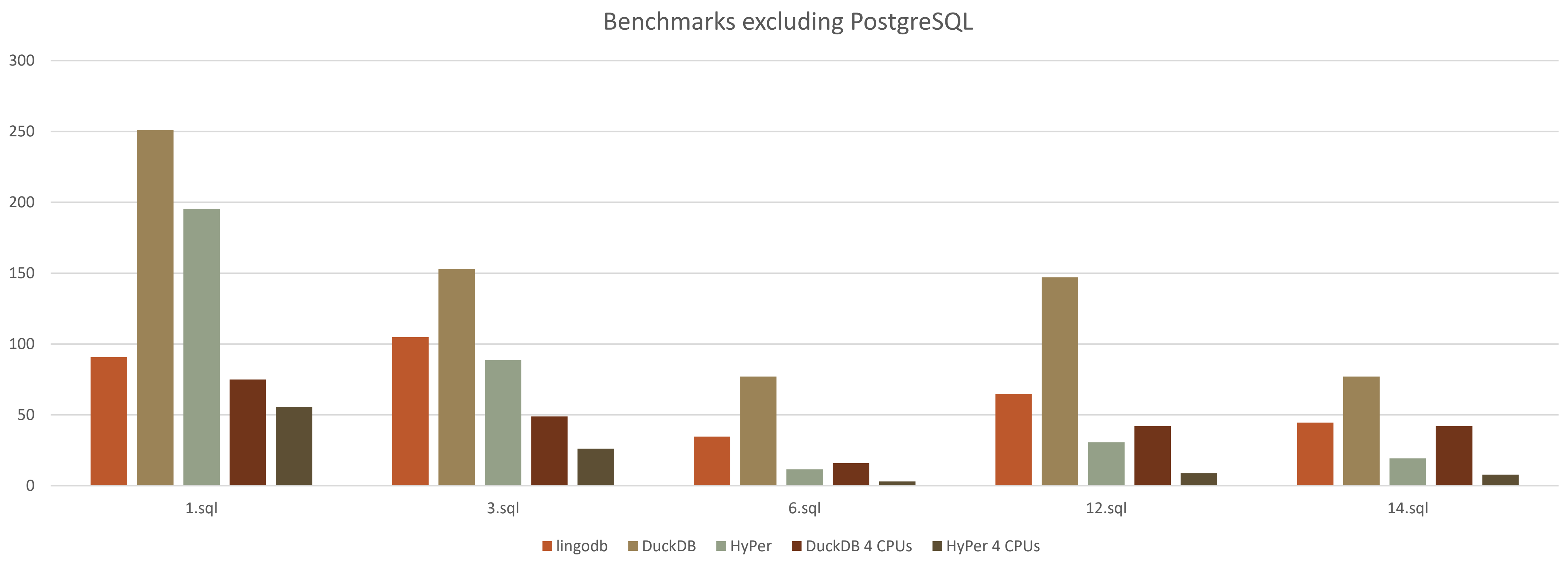
1. Due to wanting to run this on multiple environments, wanting my results to be reproducible, and that I'm only comparing my results to itself, Docker was chosen.
2. I turned to Mutable, which already had the framework done for several relevant databases – Mutable, PostgreSQL, DuckDB, and HyPer.
3. This was packed into a Docker container with all the installation steps and build steps, (the Dockerfile is over 122 lines.
4. Lingo DB was put into its own container and was substantially simpler to do.
5. Peculiarly, despite us using Mutable as the main source for benchmarking, it doesn't work inside the benchmarks (yet!)

Outcomes



Execution + compilation/planning times
7000
6000
5000
4000
3000
2000
1000
0
1.sql
3.sql
6.sql
12.sql
14.sql
lingodb
PostgeSQL
DuckDB
HyPer
DuckDB 4 CPUs
HyPer 4 CPUs

Outcomes



Benchmarks excluding PostgreSQL
300
250
200
150
100
50
0
1.sql
3.sql
6.sql
12.sql
14.sql
lingodb
DuckDB
HyPer
DuckDB 4 CPUs
HyPer 4 CPUs

Contents

Remaining work for the literature review

1. … write the literature review itself.
2. … fix the Mutable benchmark
3. Understand more databases – Umbra, a JVM one, Ignite or DuckDB for how other in-memory databases tackle these problems.
4. Benchmark how much time PostgreSQL spends inside the CPU itself. This must be enough for them to justify exploring JIT in the first place. Another paper on JitBuilder got a consistent improvements on entire queries
5. Benchmark PostgreSQL with JIT enabled, and maybe other JIT compilers. There's some content online about other JIT engines being better.

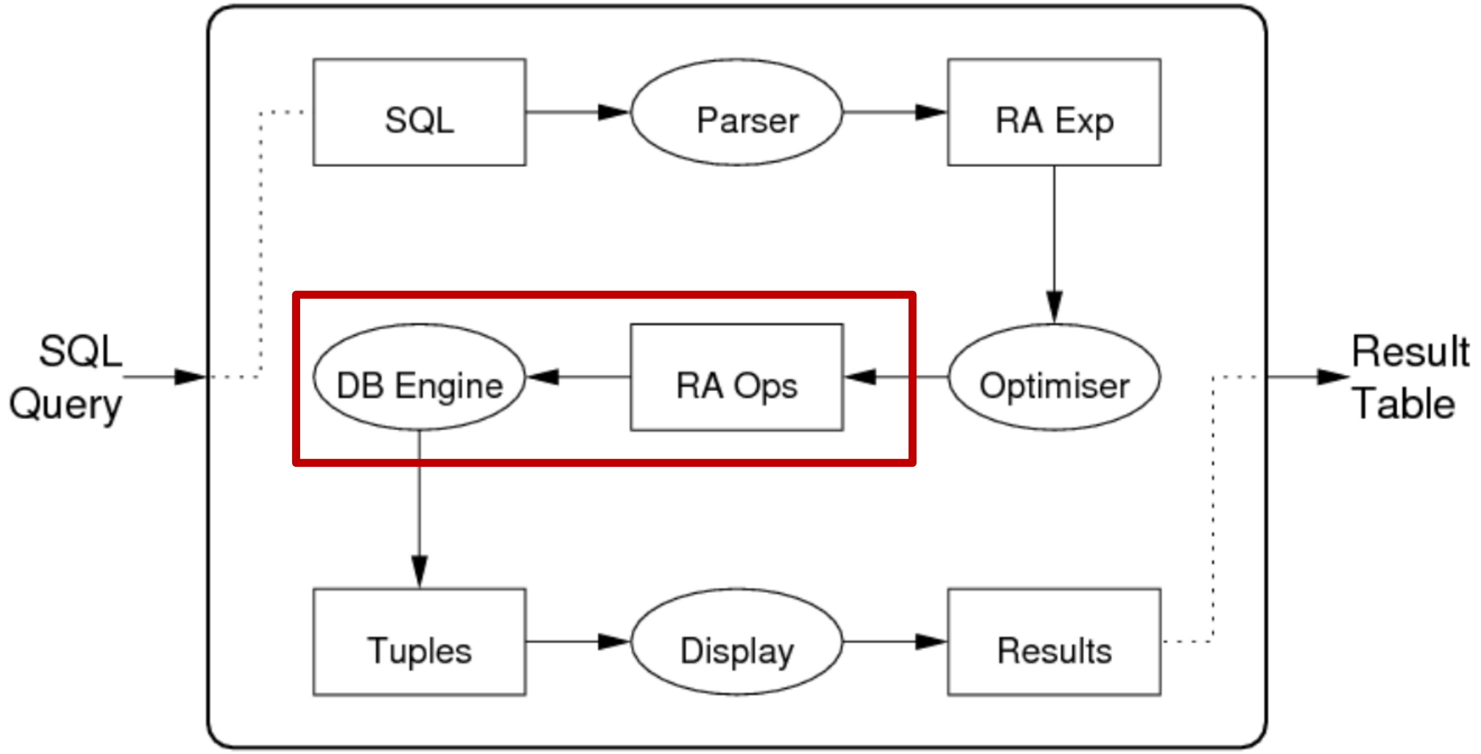
Proposed future works by others

1. Putting MLIR into an existing database.
2. Integrate a different compiler (JVM, Wasm) into an existing database.
3. Expand operators on one of these smaller papers.
4. Using novel hardware.
5. Further benchmarking to compare the best way to proceed.

Where JIT and lowerings fit

JIT would be used here inside the pipeline, which would be most of the execution time.

We want to take the relational algebra operations output, create MLIR lowerings for this, and then run the compiled output inside the DB Engine.



MLIR into an existing database

1. So, the main target of this thesis at this stage is MLIR with entire query expressions
2. The first stage here is taking RA Ops and parsing it into MLIR
3. Since it’s already quite optimised, there isn’t a need for custom lowerings like how Lingo DB does
4. Then I need to explore how to execute this instead of the way Postgres handles its DBEngine
5. The big downside here is that most likely this will break ACID compliance inside of PostgreSQL. At this stage, I’m considering this out of scope entirely
6. With MLIR IR, we should be able to explore other JIT compilers as well – wasm and python JIT tools are possible

But if there’s any obvious reasons this idea isn’t going to work, there’s a number of other interesting future work ideas that would work to explore!