Anders Hallem Iversen
Sveinung Øverland

Compiling expressions in MySQL

Master's thesis in Computer Science
Supervisor: Norvald H. Ryeng
June 2022

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Anders Hallem Iversen
Sveinung Øverland

Compiling expressions in MySQL

Master's thesis in Computer Science
Supervisor: Norvald H. Ryeng
June 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Abstract

The bottleneck of many modern database systems has shifted to inefficient utilization of CPU and memory resources, as most of the data now fits within the main memory capacity. As a result, database research has focused on code generation and runtime compilation to produce a more CPU-friendly execution environment. Despite this growing interest in runtime compilation, MySQL has not taken any action on this matter and is still affected by its interpreted engine's poor CPU utilization.

This thesis explores the nature of runtime compilation in MySQL and investigates the possible performance impacts of compiling filter expressions. This work aims to determine whether the MySQL ecosystem can improve its performance by replacing the logic of expressions with native machine code and removing the substantial amount of inflated instructions introduced by the current system for expressions. We experiment by implementing a just-in-time compilation system in MySQL using the LLVM compiler framework and provide performance details of two conducted experiments. We achieve a 100x speedup on evaluating expressions for extremely large queries and reduce the expression evaluation time by half on business-oriented queries. The results from our work confirm that MySQL is indeed affected by poor CPU utilization and gains substantial performance improvements by compiling expressions.

In this thesis, we propose an approach to improve CPU and memory usage in MySQL by compiling expressions at runtime and providing fresh insights into the instruction overhead of MySQL expressions. We show that MySQL can gain significant performance improvements from compiling expressions and further suggest that MySQL should adopt the essence of runtime compilation to enhance performance. This research lays the first steps for MySQL on compilation and code generation, where we present an approach on how to adopt JIT compilation on expressions in MySQL.

Sammendrag

Flaskehalsen til mange moderne databasesystemer har gått over til å være ineffektiv utnyttelse av CPU- og minneressurser, ettersom de fleste dataene nå passer innenfor kapasiteten til hovedminnet. Som et resultat har databaseforskning rettet fokuset mot generering av kode og kompilering i kjøretid for å produsere et mer CPU-vennlig eksekveringmiljø. Til tross for denne økende interessen for sanntidskompilering, har ikke MySQL tatt noen grep i denne saken og er fortsatt i dag påvirket av den dårlige CPU-utnyttelsen av spørringsmotoren.

Denne masteroppgaven utforsker mulighetene for sanntidskompilering i MySQL og undersøker mulige ytelseseffekter av å kompilere filteringsuttrykk under eksekvering. Dette arbeidet tar sikte på å finne ut om MySQL-økosystemet kan forbedre ytelsen ved å erstatte logikken for uttrykk med maskinkode, og fjerne en urimelig mengde med oppblåste instruksjoner introdusert av eksisterende uttrykkslogikk. Vi eksperimenterer ved å implementere et "just-in-time" kompileringssystem i MySQL ved å bruke LLVM-kompilatorrammeverket og gir ytelsesdetaljer for to eksperimenter. Resultatene fra vårt arbeid bekrefter at MySQL faktisk er påvirket av dårlig utnyttelse av CPU-en og oppnår betydelige ytelsesforbedringer ved å kompilere uttrykk. Vi oppnår en ytelsesforbedring på 100x i evaluering av uttrykk for ekstremt store spørringer og videre halverer uttrykksevalueringstiden på forretningsorienterte spørringer.

I denne oppgaven legger vi fram en metode for å forbedre CPU- og minnebruken i MySQL ved å kompilere uttrykk ved kjøring og gir ny innsikt i instruksjonsoppblåsningen til uttrykk i MySQL. Vi viser at MySQL kan oppnå betydelige ytelsesforbedringer ved å kompilere uttrykk og foreslår videre at MySQL bør ta i bruk essensen av sanntidskompilering for å forbedre ytelsen. Denne forskningen legger de første stegene i kompilering og kode generering under kjøretid for MySQL, der vi presenterer en metode for hvordan man kan ta i bruk sanntidskompilering av uttrykk i databasesystemet.

Acknowledgements

We thank Associate Prof. Norvald H. Ryeng for excellent guidance throughout this thesis and valuable insights into the MySQL engine as Software Developer Director for the MySQL optimizer in Oracle. His guidance has ensured that we have been working on the right track. It has provided valuable information and knowledge about MySQL, which has been essential for making this thesis happen.

Additionally, we would like to thank Tor Didriksen for supporting us in obtaining a painless experience while working on MySQL's build system and ensuring we achieved an efficient development process throughout this thesis.

Finally, we would like to express our appreciation to Oracle for providing us with the opportunity to work on this thesis. A huge thanks to all of them!

Table of contents

List of Figures

List of Tables

Chapter 1

Introduction

Many database systems were created at a time when I/O was a substantial bottleneck of execution. The bottleneck is a consequence of the architectural mindset of early-day database systems. The assumption was that main memory was limited in size, thus relying on heavy disk accesses. With the recent leap in hardware technology, the capacity of computer memory has increased substantially, allowing the fitting of a large portion of the database into memory. Subsequently, the need for frequent disk accesses was markedly reduced, making the CPU and main memory latency the bottleneck of modern in-memory database systems [3].

However, many database systems have struggled to keep up with this transformation, where their core architecture is still built around minimizing I/O. For instance, many systems employ the traditional *iterator model* for executing queries, consisting of an abstract operator interface that supports a chain of repeated calls to virtual functions for processing the data. This paradigm exacerbates the underutilization of CPU resources, making insufficient use of temporal and memory locality, causing frequent memory accesses and inefficient CPU registers. Furthermore, it significantly increases the number of instructions used to execute a query and is frequently outperformed by handwritten code. This indicates that optimizing for CPU-friendly execution has a great potential to increase the performance of common-scaled memory systems by strengthening the principle of locality and minimization of instructions.

Researchers began applying principles and techniques from compiler theory to database query-planning methodologies to confront this growing CPU bottleneck. The essence of the research consisted of compiling queries into native machine code utilizing the extra information obtained at runtime. This entails generating query-specific code, which eliminates the instruction overhead provided by traditional execution engines and boosts execution performance. For instance, some early researchers on this topic were the authors of HIQUE, a query engine that generates custom code for a specific query [24]. In their paper, they were able to minimize the number of function calls, reduce the number of instructions, improve cache locality, and outperform interpretation-based commercial database systems by a substantial factor [24, pg. 12].

Another research paper on this topic claims that the most significant way to improve throughput by 10-100x is to dramatically reduce the number of instructions executed by a query [13, pg. 2]. The paper introduces an in-memory database engine, Hekaton, which transforms SQL queries into machine code, and achieves 15x fewer instructions for a set of specific queries [13, pg. 8]. The authors stated that a 90% reduction is needed to achieve a 10x improvement, which was impossible by optimizing their traditional interpreted execution engine.

For this reason, there has been an increasing adoption of query compilation in many commercial systems, such as Hekaton[13], Cloudera Impala [49], SingleStore [6] and Spark [1], as well as a large portion of academic research and open-source database systems, such as Hyper [23], MonetDB [46] and Postgres [16], which have shown substantial performance improvements by applying this technique [13, 44, 22].

For MySQL, adopting query compilation is considered an enormous task and would introduce significant changes to the existing query execution engine. On the other hand, MySQL is a well-known relational database system that originates from when I/O was the primary bottleneck and naturally adopted the iterator architecture. In compliance with other such systems, MySQL is also troubled by the poor CPU performance of the iterator model, affected by poor locality and an inflated number of instructions. Postgres has an architectural nature similar to MySQL, utilizing the Volcano-style iterator model. To combat their most significant bottlenecks, Postgres enhanced just-in-time compilation using the LLVM compiler framework to accelerate the evaluation of *expressions* and *tuple deforming*, which has shown many promising performance improvements [5, 30]. A more sensible approach would be to take small and similar steps as in Postgres, exploring the possibilities for MySQL to compile similar bottlenecks and join the growth of database systems compiling parts of the query plan.

Therefore, this thesis will investigate the possibilities of just-in-time (JIT) compiling filter expressions in MySQL, connecting the database with the research on query code generation. The implementation of expressions in MySQL has similarities with the behavior of the iterator model and includes many of the same performance disadvantages. In addition, the evaluation of complicated expressions can be a key bottleneck according to a research paper on *choke points* in the TPC-H [45] benchmark [4]. This causes poor performance behaviors, as many parts of query execution, such as filters, table scans, aggregations, and joins, rely on the efficient evaluation of expressions.

Our JIT implementation in MySQL will excite notable runtime savings. We speed up the expression evaluation process by 100x for huge queries and decrease the evaluation time of business-oriented expressions by half. Furthermore, we show that MySQL produces a significant amount of instruction overhead for evaluating expressions, which we eliminate to a considerable extent, showing how query execution in MySQL can benefit from compiling SQL expressions. This work will demonstrate how MySQL can improve query performance by adopting JIT compilation, where we build a prototype in MySQL using the LLVM compiler framework, and show how the compilation of expression evaluation can coexist with the existing interpreted engine.

The goal of this thesis is to look at the potential of JIT compiling filter expressions in MySQL, reveal its impacts on performance, and present how it can be conducted within the existing MySQL ecosystem. The work of this thesis will therefore aim to answer the following research question:

RQ1. **Is it possible to improve query performance in MySQL through JIT compiling expressions?**

This study aims to determine how these aspects are affected in a complete JIT implementation of expressions in MySQL using the LLVM compiler framework. To fully understand the essence of the research question, we further realize that a small additional set of subquestions needs to be addressed:

RQ2. *What is the expected performance impact of JIT compiling expressions in MySQL?*

RQ3. *Under what circumstances is it considered beneficial to JIT compile an expression?*

RQ4. *Is the choice of using LLVM for JIT considered a good approach for MySQL?*

This research focuses on enhancing expression evaluation in MySQL by adopting JIT compilation and utilizing the LLVM compiler framework. We will determine the answer to these questions by conducting experiments on our LLVM-designed JIT prototype and further elaborate on our observations and empirical findings to understand the topic. We want to stress that we exclusively look at filter expressions, and future mentions of expressions will also refer to filter expressions exclusively.

The remainder of the thesis will be structured as follows: Chapter 2 will strive to provide the background knowledge that we consider essential to understanding the remainder of this thesis, in addition to providing an overview of related work on this topic. This will consist of a brief introduction to compiler theory, databases, the internal workings of expressions in MySQL, a shallow presentation of the LLVM compiler framework, and then a brief presentation of existing query compiler engines. Afterward, Chapter 3 will discuss the possible approaches for adopting JIT compilation in MySQL and then explain our JIT implementation for evaluating expressions. Moving on, Chapter 4 will describe the experiments we conducted, where we will evaluate and discuss our observations and findings in the same chapter. Finally, in Chapter 5, we will draw the conclusion, provide an answer to the research question, and further suggest future work on this topic for MySQL.

Chapter 2

Background

To grasp how JIT compilation can be conducted in MySQL, it becomes necessary to understand the different theories and aspects of compilation and to know enough about the internal workings of databases and MySQL. This chapter will provide an overview of the relevant theory and research considered essential to understanding the remainder of this thesis. Moreover, to follow our decisions and considerations throughout this thesis, this chapter will also give an overview of related work in this area, where we will briefly explore some other database systems that have adopted query compilation on different levels of scale.

To start, in Section 2.1, we first briefly introduce compiler theory, explaining the internal components of compilers and their similarities to databases. This introduction will provide the necessary knowledge to help understand what it means to compile and the internal steps for achieving native executable machine code. Afterward, in Section 2.2, we will focus on how this applies to databases and explain the inner workings of a standard iterator model used in many database engines. The idea is to explain how databases execute a query and show why their internal workings are not necessarily considered CPU and memory friendly. Then, in Section 2.3, we will give a brief presentation on the internals of MySQL and how MySQL implements expressions by code. This presentation will highlight how expressions work within MySQL and underline its performance drawbacks. The following section, 2.4, will present the LLVM compiler framework, which we use to implement JIT in MySQL, described later in this thesis. The focal point of this section is to provide a brief understanding of what LLVM is, how it works, and why we can use it in our implementation.

The very last section of this chapter, Section 2.5, will describe some existing database systems on this topic, where we focus on their approaches to adopting JIT and their performance achievements.

2.1 Compilation, interpretation & just-in-time

To properly understand what our main research question tries to answer, it is essential to understand what compilation is, especially in light of just-in-time (JIT) compilation, and how implementing JIT compilation differs from traditional approaches.

2.1.1 Compilation & Compilers

In computing, compilation is a general term that describes a process that takes some language input (usually a programming language) and produces some output of another language (usually machine code). Compilers are programs that use compilation to produce some executable output. An example of a compiler is GCC (GNU Compiler Collection) or Clang (C language frontend for LLVM). Compilers can usually be divided into three conceptual parts: the *frontend* part, the *optimizer* part and the *backend* part. [43, 42, 8]

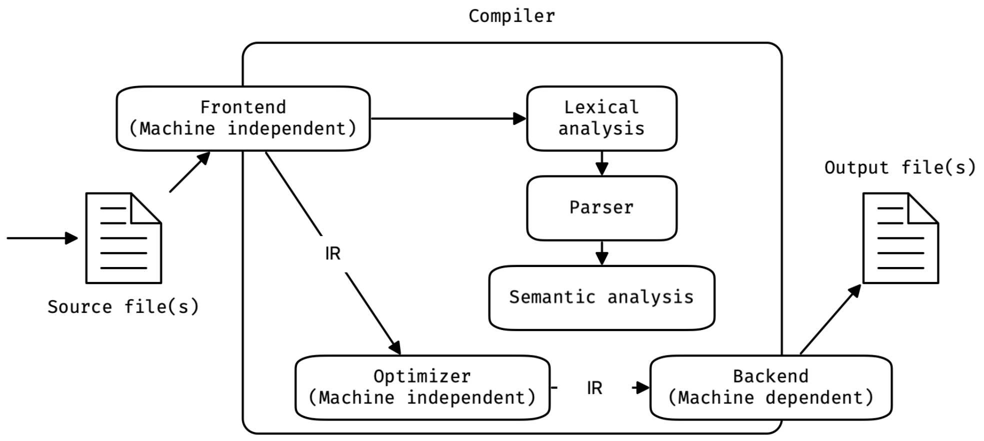
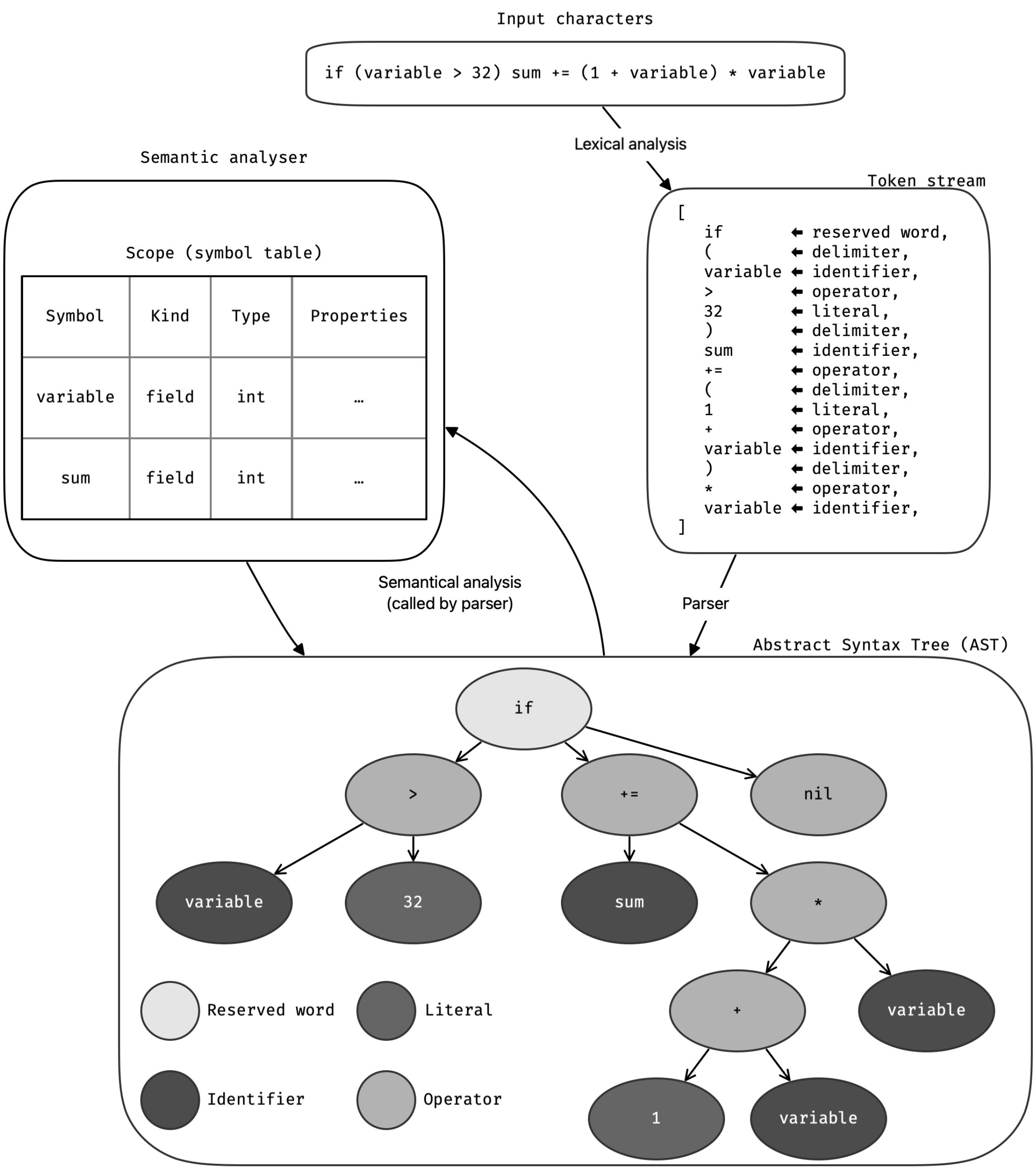

Compiler
Frontend (Machine independent)
Lexical analysis
Parser
Semantic analysis
Source file(s)
IR
Output file(s)
Optimizer (Machine independent)
IR
Backend (Machine dependent)

Figure 2.1: Compiler conceptual overview

Figure 2.1 shows an overview of a typical compiler, showing the three mentioned parts and the *frontend* in more detail. The *frontend* has the responsibility of interfacing with the source language and converting it to an *intermediate representation* (IR) that can be used later internally in the compiler. The *frontend* achieves this by first running a *lexical analysis*, also known as a *scanner*, which in turn outputs a *token* stream. A *token* is quite simply a set of characters and a corresponding type for those characters, where an example of this is "+=" with the corresponding type "operator". The *token* stream is then passed on to the *parser.* The *parser* turns the *token* stream into a *Abstract Syntax Tree* (AST). The parser will iterate through the *token* stream and create the AST as it goes along; meanwhile, reasoning about the *token* stream and making sure it makes sense, for example, by checking that all operators have the right number of operands. The *semantic analyzer* is responsible for ensuring that the input makes sense for the language. It does this by having a defined language "grammar" which determines what the language allows and keeping track of variables in a *symbol table.* There will usually be several *symbol tables* that each corresponds to a scope, be it global scope, function scope, or any other scope. [43]



Symbol	Kind	Type	Properties
variable	field	int	…
sum	field	int	…

```
[
  if       ← reserved word,
  (        ← delimiter,
  variable ← identifier,
  >        ← operator,
  32       ← literal,
  )        ← delimiter,
  sum      ← identifier,
  +=       ← operator,
  (        ← delimiter,
  1        ← literal,
  +        ← operator,
  variable ← identifier,
  )        ← delimiter,
  *        ← operator,
  variable ← identifier,
]
```

Figure 2.2: Lex + Parser example

Figure 2.2 shows a simple model of the *lexical analysis*, *parser* and *semantical analysis* steps that the *frontend* part goes through and what the internal data structures could look like. An AST is a tree structure that contains a bunch of information about each token and places it logically for creating IR. For example, an operator in the AST would contain information about its operands and their type so that the *semantical analysis* can reason about the types and make sure that a result can be created. The AST also contains information about where each token is found in the source. This information helps a compiler tell the programmer where potential errors are located and mean that compiler errors are often underlined so they can be found easily.

Following the *frontend* part of the compiler is the *optimizer* part of the compiler. The *optimizer* takes IR and uses it to do advanced transformations and optimizations. Typical optimizations include: *loop optimization*, *dead code elimination* and *global value numbering.* IR typically looks closer to machine code than the source language but retains the same machine independence as the source language. Because of this, the *optimizer* does not need to take hardware considerations when optimizing the IR and can leave all the machine-dependent functions to the *backend* part of the compiler. At the end of the transformation and optimization, the *optimizer* itself will output an improved version of the IR, representing the same source code as the IR it was given.

Lastly, the *backend* part of the compiler takes the improved version of the IR from the *optimizer* and turns it into machine-dependent machine code that can be executed. One of the processes that the *backend* uses to achieve this is called *register allocation*, which assigns the various values and variables to a limited number of registers in the CPU. *Register allocation* results in better runtime performance since the machine code can get rid of several expensive store and load operations towards memory. What makes the *backend* machine-dependent is that it has to consider the various architectures that the compiler supports. Different architectures could have different instruction sets and different address byte lengths, different register names, and register numbers.

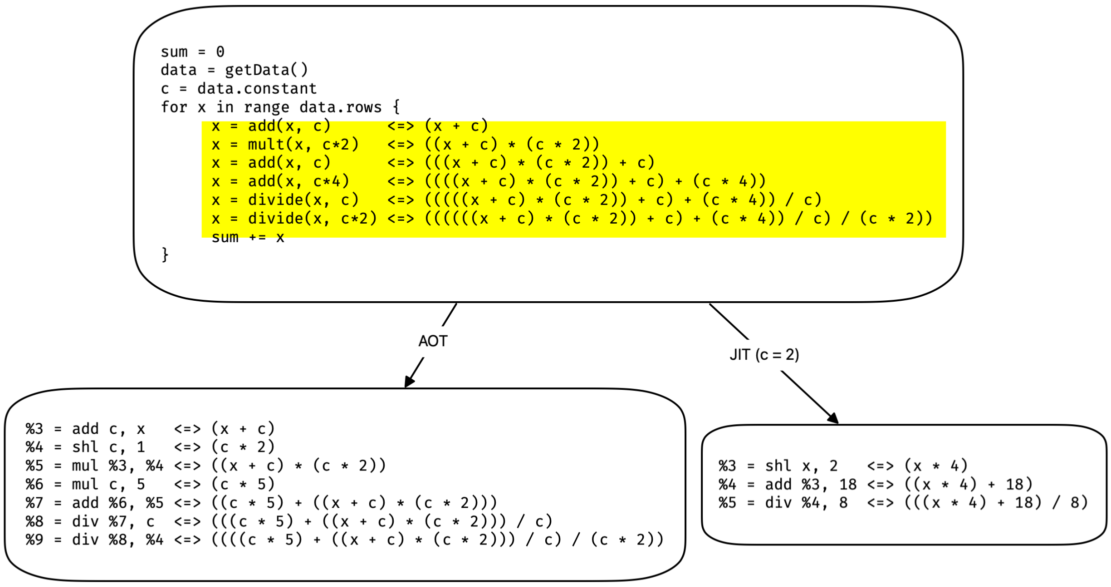
2.1.2 Interpretation

Suppose the conceptual compiler presented above replaced *backend* with a more straightforward machine-independent step that created a much closer-to-machine code representation. And then, we distributed this as our program together with a program that can read this close-to-machine code representation and execute it. We have created a compiler *interpreter* combo for a source language supported by the *frontend.*

This combination is found in the JDK and JVM for the Java programming language. Where close-to-machine code representations called *byte code* are distributed as cross-platform programs that can be run on any architecture that has a *interpreter* for the *byte code.* Another kind of *interpretation* is interpreting the source language directly. Programming languages like Python and Javascript do this. An *interpreter* works by executing the source *byte code* or language statement by statement. Essentially an *interpreter* is much like a *compiler.* It takes one or many source files, parses these, maybe runs some optimizations or JIT if it determines there can be a speed up, and passes the statements to an executor instead of outputting to the file. *Interpreters* usually prioritize fast and frequent execution of the statements. Usually, *interpreters* will be outperformed by pre-compiled binaries, but they offer other benefits like memory efficiency and easier debugging. The memory efficiency comes from not having to load huge chunks of code into memory simultaneously, as pre-compiled binaries need to. The easier debugging comes from the feedback the *interpreter* can give at runtime when it encounters errors. [43, 9]

2.1.3 Just in time compilation

Just in time compilation (JIT) differs from *interpretation* in that with JIT, IR gets compiled to actual machine-dependent machine code instead of byte code, more in line with what a traditional compiler *backend* would do. In practical terms, JIT means to do compiling at runtime. JIT is important as a concept since the best compilation choices cannot always be taken *ahead of time* (AOT). For AOT compiled languages like C, C++, or Rust, JIT becomes a tool to create more optimized code at runtime that can surpass the performance of the AOT compiled code. For *interpreted* languages like Java or Javascript, JIT becomes a tool for compiling hot code paths, thus increasing performance and possibly increasing the efficiency of the code. The downside of JIT is that compiling at runtime takes time, so if applied wrong, using JIT can slow down the overall code execution time. Another downside of JIT is that depending on how it is implemented; it can make the program's binary size much larger by having to include, at minimum, a simple compiler and probably a simple linker.



```
sum = 0
data = getData()
c = data.constant
for x in range data.rows {
    x = add(x, c)      <=> (x + c)
    x = mult(x, c*2)   <=> ((x + c) * (c * 2))
    x = add(x, c)      <=> (((x + c) * (c * 2)) + c)
    x = add(x, c*4)    <=> ((((x + c) * (c * 2)) + c) + (c * 4))
    x = divide(x, c)   <=> (((((x + c) * (c * 2)) + c) + (c * 4)) / c)
    x = divide(x, c*2) <=> ((((((x + c) * (c * 2)) + c) + (c * 4)) / c) / (c * 2))
    sum += x
}
```

```
%3 = add c, x   <=> (x + c)
%4 = shl c, 1   <=> (c * 2)
%5 = mul %3, %4 <=> ((x + c) * (c * 2))
%6 = mul c, 5   <=> (c * 5)
%7 = add %6, %5 <=> ((c * 5) + ((x + c) * (c * 2)))
%8 = div %7, c  <=> (((c * 5) + ((x + c) * (c * 2))) / c)
%9 = div %8, %4 <=> ((((c * 5) + ((x + c) * (c * 2))) / c) / (c * 2))
```

```
%3 = shl x, 2   <=> (x * 4)
%4 = add %3, 18 <=> ((x * 4) + 18)
%5 = div %4, 8  <=> (((x * 4) + 18) / 8)
```

Figure 2.3: IR without & with JIT example

Figure 2.3 shows a simple example of how knowing that `c = 2` at runtime can allow for much better optimization of a mathematical expression, thus simplifying and speeding up the code. The figure shows that the code running in the loop is decreased from seven lines of IR to three lines, which with a naive assumption that the IR operations used to take the same amount of time to run would mean that JIT could provide ~2.3x speedup. The simplified code of the JIT block could therefore provide a massive benefit if the loop runs for a long time.

2.1.4 Parallels with databases

A *database management system* (DBMS) is much like a compiler, as it takes an input language, such as SQL, which it will then parse and create a logic of computation for, in the form of a query plan. The parsing is much like a compiler *frontend* since it performs lexical and semantical analysis on the input query and ultimately outputs an intermediate representation (IR) in the form of a parse tree. A parse tree differs from an AST in that it represents the entire input and makes no considerations of the language grammar. The difference means that for the input `(x + 2)`, the parse tree would include all the tokens, including the parenthesis. On the other hand, the AST would omit the parenthesis as it would produce a node of '+' with 'x' and '2' as children. The DBMS will then typically apply transformations and optimization passes on the parse tree IR, ultimately converting it to a *logical query plan*. The *logical query plan* describes a possible logical computational flow for the query. Lastly, the DBMS converts the *logical query plan* into a *physical query plan* which the DBMS can execute. The executor of the DBMS can be seen as a *interpreter* that takes the *physical query plan* and executes it record by record. A record, or tuple, is a row of data from the database. Sometimes the *logical query plan* can be native machine code that the executor runs on each record. Another difference from typical compilers lies in the input languages. Query languages like SQL tend to be *declarative*, meaning the language describes the wanted result and not how to get there. The declarative nature of SQL leaves the DBMS open to decide for itself what the control flow for the query should be. [7]

2.2 Databases & Volcano

Before undertaking JIT compiling expressions in MySQL, it is essential to understand what databases are and how they work. We decided to split this general database theory from MySQL theory because understanding databases generally will help give context to databases used in related work. Databases have long been the foundation of everything from small-scale applications that use local databases like SQLite to enterprise-scale applications. Databases allow developers to abstract data storage and usually feature advanced data querying. This section will introduce some of the more technical sides of databases, focusing on similarities in SQL databases, such as MySQL and PostgreSQL. In addition, this section will describe a generic approach to the various steps that SQL databases take when executing a query. It should be noted that implementations will differ from this explanation and that in actual implementations, steps might overlap more than this explanation would make it appear. This section will also use MySQL terminology when presenting the different steps since this terminology is the most relevant to the research question. Thus, some wording might seem slightly off compared to other databases.

Lastly, this section will introduce a traditional query evaluation system, Volano, and present the benefits and drawbacks related to this thesis.

Structures Query Language (SQL) is a language for managing data in relational databases. A query in SQL is declarative, which means it describes what the query result should look like instead of the steps to get the result. Further, this means that an SQL database needs to figure out the path to get the result on its own.

2.2.1 Steps from query to result

Generally speaking, any relational database will go through the same general main steps when going from a query to a result. These steps are *parse*, *prepare*, *optimize*, and *execute*. However, when comparing a relational database to a compiler, the *parse* and *prepare* steps become the same step; because for a database, this step represents the step where the database resolves fields, which is equivalent to a compiler's semantic analysis step. Figure 2.4 gives an overview of these steps and how they interact.

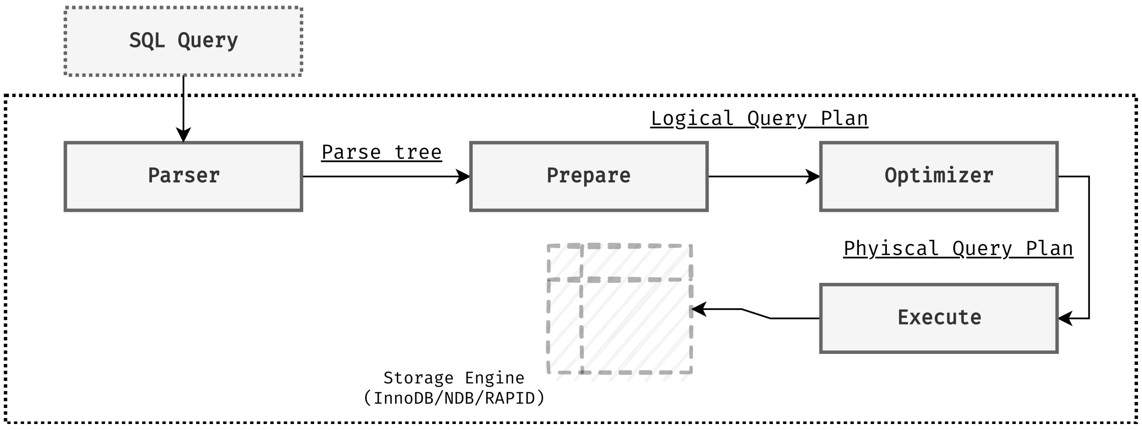


Figure 2.4: Generic SQL database step flow, showing how a query moves through a database

When a query is sent to the database, the first step for the database to run is the *parse* step. *Parsing* is the first step for the database to turn the plaintext SQL query and turn it into something workable. The result from this step is a tree structure representing the various parts of the query. *Parsing* in SQL databases is much like *parsing* a programming language in that a tree structure is created to represent the input language, and *lexical analysis* is run in addition to *semantical analysis*. This similarity with a compiler's *parser* step makes it comparable to a compiler *frontend* part.

The next step for an SQL database is the *prepare* step. In this step, the database will resolve fields and ensure the query can run. Resolving fields involves making type inferences from table information on the various fields. Resolving also results in binding the field nodes in the tree structure to actual row references with corresponding types. Another result of this resolving is that the database can give feedback to the user if specific fields referenced in the query are not actually in a corresponding table and cannot be resolved.

After resolving all the necessary fields for the query, the database applies some optimizations. These optimizations make executing the query more efficient and faster. Common optimizations include *constant folding* and *dead code elimination*. These optimizations are also common compiler optimizations. *Constant folding* is the process of computing constant expressions ahead of time. For example, given the expression `age = 20 + 25` *constant folding* would turn the expression into `age = 45`; this can save a bunch of time during runtime especially if the expression is used in a long-running loop. *Dead code elimination* is figuring out if a code path can be removed if it can never actually run. For example, given the statement `if (20 > 30)then ...` it is possible to deduce that the whole branch following the `if` statement can be removed since 20 is never larger than 30. If drawing a parallel to compilers, this step is very similar to a compiler optimizer; the only real difference is how a database will only apply optimizations that benefit their workload and not general optimizations that a compiler would apply. Overall the *prepare* step is very much like a compilers *optimize* step.

The next step a database takes is the *optimize* step. This step aims to create a physical query plan, which means creating some data structure, usually a tree structure, that can be used by the executor to get some result. The *optimizer* step has to take the tree structure from the earlier step and figure out the best way to get the client's desired result. The resulting physical query plan will describe how the executor should join which tables to be as fast and efficient as possible for the execution. The process of figuring out which combinations of joins can be exhaustive, but an exhaustive search quickly becomes unfeasible when the number of joins increases. At this point, another approach, including genetic algorithms, must be used to find a close-to-best physical query plan. The *optimizer* step used in databases should not be confused with the *optimizer* step of compilers. The database *optimizer* step is more like the compiler's *backend* step; in that, it creates some output that is ready to be executed.

Lastly, the database will execute the physical query plan through its *execute* step. This step will then read and usually query the storage engine used by the database to get rows that the physical query plan can use.

2.2.2 Volcano query evaluation system

Next, we will introduce a standard iterator model, Volcano. Since it is employed in MySQL and has similar drawbacks to the implementation of expressions, we consider it necessary to provide a brief overview of this system and its disadvantages.

Introduced in 1994 by Graefe, the Volcano query evaluation system has become the standard for query evaluation systems in relational databases [18]. Volcano is itself an infrastructure model for query evaluation. The model expresses a query as a tree of relational algebra operators; these operators include *join*, *selection*, *projection*, and other operators that are domain-specific. Another important concept in Volcano is the *iterator* interface. The *iterator* interface forces each operator to implement the functions: *open()*, *next()*, and *close()*; this interface intentionally makes each operator look like a typical file interface.

The way Volcano executes a query is by the root node of the operator tree calling *next()* on its child node, which will, in turn, mean every operator until a leaf node will also call *next()* on its child operators. The leaf node will usually perform the actual retrieval of a row of data, and the data will then bubble up through the various operators until it reaches the root node. This call chain forms an *operator pipeline*. Thus, the pipeline has control flow from top to bottom and data flow from bottom to top. Not all operators can process data a single row at a time; these operators are called *pipeline breakers* and need to wait for child operators to finish before they can start processing data. Operators that are *pipeline breakers* include: *joins*, *sort*, and *group by*. The *pipeline breakers* mean many queries, especially complex ones, must be represented as multiple pipelines. Each pipeline forms a *materialization* which can be thought of as a temporary table including the resulting data from the pipeline, which can be used as input for other operators in other pipelines. The *materialization* is buffered in a given *pipeline breaker* operators internal state. Figure 2.5 shows an example operator tree with *pipelines* grouped by dashed lines.

```
SELECT
  *
FROM
  A,
  (
    SELECT name, COUNT(*)
    FROM B GROUP BY name
  ) AS T
WHERE A.rank < 10
AND A.name = T.name
```

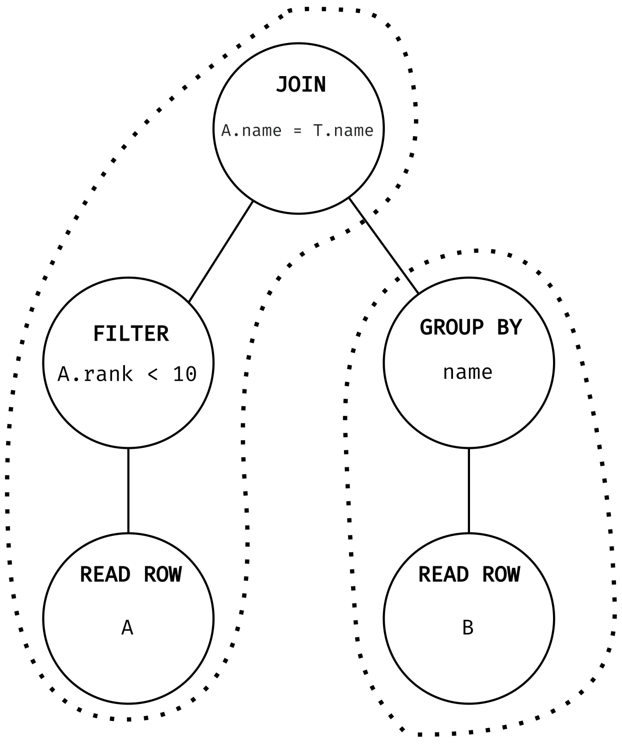


Figure 2.5: The relation between a query and its corresponding operator tree.

2.2.3 Advantages and disadvantages of Volcano

The Volcano model naturally comes with several advantages but also disadvantages. One of the advantages is how extensible the system is; its use makes adding operators very simple by using the *iterator* interface. This extensibility is achieved by the separation of concern when using an interface. When adding a new operator, it does not have to consider any of the other operators' implementation details.

The biggest downside of Volcano is the overhead created by making several repeated calls to *next()*. Each time a function is called, more stack space must be used. The program has to jump to another place in memory; this decreases temporal and spatial locality, decreases CPU branch prediction and prevents inlining and unused code deletion.

2.3 MySQL

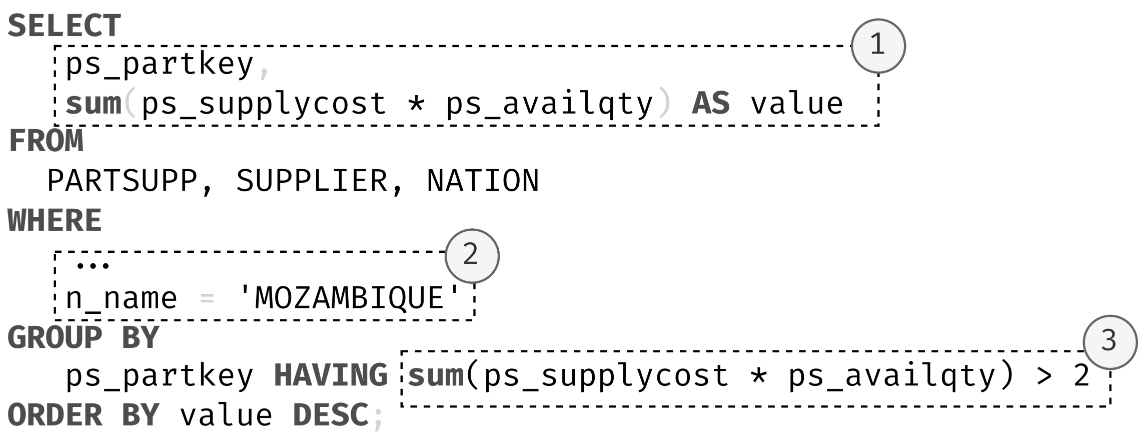
To understand how `WHERE` expressions in MySQL can be compiled, it is vital to understand how MySQL works internally and how the existing expression evaluation process is performed. This section will overview how MySQL represents expressions and how they are evaluated. Furthermore, this section will also describe how

MySQL internally executes a query from the beginning to the end. The primary focus is on the life of `WHERE` expressions and how they fit within this process.

First, MySQL is a popular open-source relational database management system that reads and mutates data in row-based table structures through SQL queries. The entire project started in 1995 and is written in C++, which Oracle Corporation mainly maintains at the time of writing. Since MySQL 8.0, the database system has adopted the Volcano iterator model, which executes queries on a per-row basis using iterators [39]. As in most traditional relational database systems, the query engine consists of four main components: *a parser*, *a prepare step*, *an optimizer* and *an executor* [39]. These four components are the core aspects of the query engine, where they lay the work of transforming an SQL query into an internal executable format, performing the execution, and making the results available. Filter expressions are involved in all these components in which they are created, transformed, or used.

2.3.1 Expressions in MySQL

In MySQL, expressions are mostly compliant with the ISO SQL standard [34] and are a collection of operators, values, and functions that can be reduced to a single value. These expressions can be found at three different places in a read query, in the projection list of the `SELECT` clause, in the selection list of the `WHERE` clause, and in the `HAVING` clause for aggregated selections. In this thesis, we will focus mainly on the filter expressions in the `WHERE` clause, which consists of expressions that evaluate a single row at a time and are marked with ② in Figure 2.6.



```
SELECT
   ps_partkey,
   sum(ps_supplycost * ps_availqty) AS value
FROM
  PARTSUPP, SUPPLIER, NATION
WHERE
   ...
   n_name = 'MOZAMBIQUE'
GROUP BY
    ps_partkey HAVING sum(ps_supplycost * ps_availqty) > 2
ORDER BY value DESC;
```

Figure 2.6: An illustration of a query, highlighting the different sections where expressions are used. ① being the project list, ② and ③ being the selection lists.

These filter expressions typically consist of a collection of arithmetic, logical, and comparison operators that can be computed using operands. For example, a typical expression found in the `WHERE` clause could be `age = 26 AND name = 'Peter'`. In MySQL, these expressions are represented in the code by a tree of instances of the C++ class called `Item`. The `Item` class is a base class for a whole family of derivations, representing the different components of an expression, such as logical operators, arithmetic operators, constant values, field references, and more. For example, an integer is represented by the `Item_int` class, which contains the value of a constant integer. The equality operator (`=`) is represented by the `Item_func_eq` class, which can calculate the equality of the value of two other `Item` references. By the nature of these classes, it is possible to construct all types of expression in the form of a tree of `Items`, where the root `Item` represents the full expression and the leaf nodes are the operands. In particular, Figure 2.7 shows the `Item` tree of the expression mentioned above. [33]

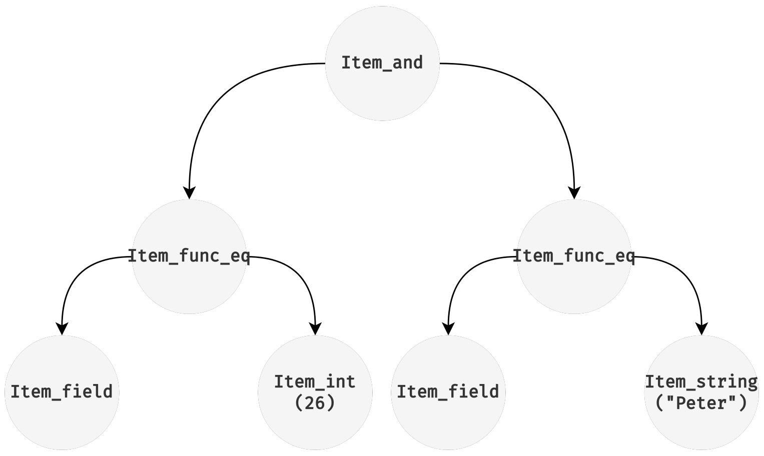


Figure 2.7: Visualization of an `Item` tree representing the expression `age = 26 AND name = 'Peter'`.

To evaluate the expression represented by the `Item` tree in Figure 2.7, every `Item` implements a set of virtual methods from the base class that are responsible for calculating the value of the `Item`. The most notable one is the `val_int()` method, which returns the value of the evaluated `Item` in the form of a 64 bit integer. In fact, there exists a family of `val_<TYPE>` methods for the most primitive types, which computes the evaluated value of the expression for the given type. For example, the `val_int()` method of `Item_func_eq` computes the equality of two other child `Items` in the tree, where the operands to be compared are extracted by calling the children's `val_int ()` method. Code listing 2.1 shows an artificial implementation of `Item_func_eq` to illustrate this concept. In such a manner, an expression is evaluated by calling the root `Item`'s `val_int()` method, which starts a traversal of the entire tree by a chain of `val_int()` function calls, where the final value is computed and returned by the root `Item`. [33]

```
class Item_func_eq : public Item {
    Item *a;
    Item *b;

    ...
    longlong val_int() {
        auto a_val = a.val_int();
        auto b_val = b.val_int();
        return a_val == b_val ? 1 : 0;
    }
    ...
}
```

Listing~2.1: Artificial implementation of the `Item_func_eq` class, performing an equality operation in the `val_int()` method.

MySQL has a `Item` derivation for every supported data type, operation, and SQL function. In this thesis only a few of them are significant to remember, and from the large collection of `Item` derivations, the following are worth noting:

- `Item_int` - Represents a constant integer value in an expression.
- `Item_string` - Represents a constant string value in an expression.
- `Item_field` - Represents a column in a table. A call to the relevant `val_<TYPE>()` method returns the column value for a given row.
- `Item_func_eq` - Represents the `=` operator.
- `Item_func_gt` - Represents the `>` operator.
- `Item_cond_and` - Represents the logical operator `AND`.
- `Item_cond_or` - Represents the logical operator `OR`.

2.3.2 Life of a query

To understand how expressions are evaluated, it is important to understand the life of a query in MySQL. As mentioned, the MySQL query engine divides the process of executing a query into four main steps: *parsing*, *preparing*, *optimizing*, and *execution* [39]. Figure 2.8 illustrates the entire life of a query with all these steps.

Parser

The life of a query begins when the server receives an SQL query from a client and sends the query to the *parser*. The parser's job is to transform the query text into a structured format that characterizes the different parts of the query. In MySQL, this format is a parse tree of C++ structures called `Query_block`. The parser uses the same parsing techniques as compilers to build the `Query_block` structure, performing lexical and semantic analysis on the query, where the parser represents *frontend* from a compiler perspective. For the expressions found in the clauses `WHERE`, `SELECT` and `HAVING`, the parser extracts the semantic meaning of these and constructs the equivalent `Item` tree, which gets stored in the `Query_block` structure. The `Items` for a given query are, thus, constructed at the very beginning of the entire execution process. The other parts of the query, such as joins and table references, go through a similar process of constructing a corresponding C++ data structure that defines their characteristics and ends up within the `Query_block` structure. When the parser finishes, the output is a full `Query_block` that describes all the different aspects of the query. [38, pg. 167]

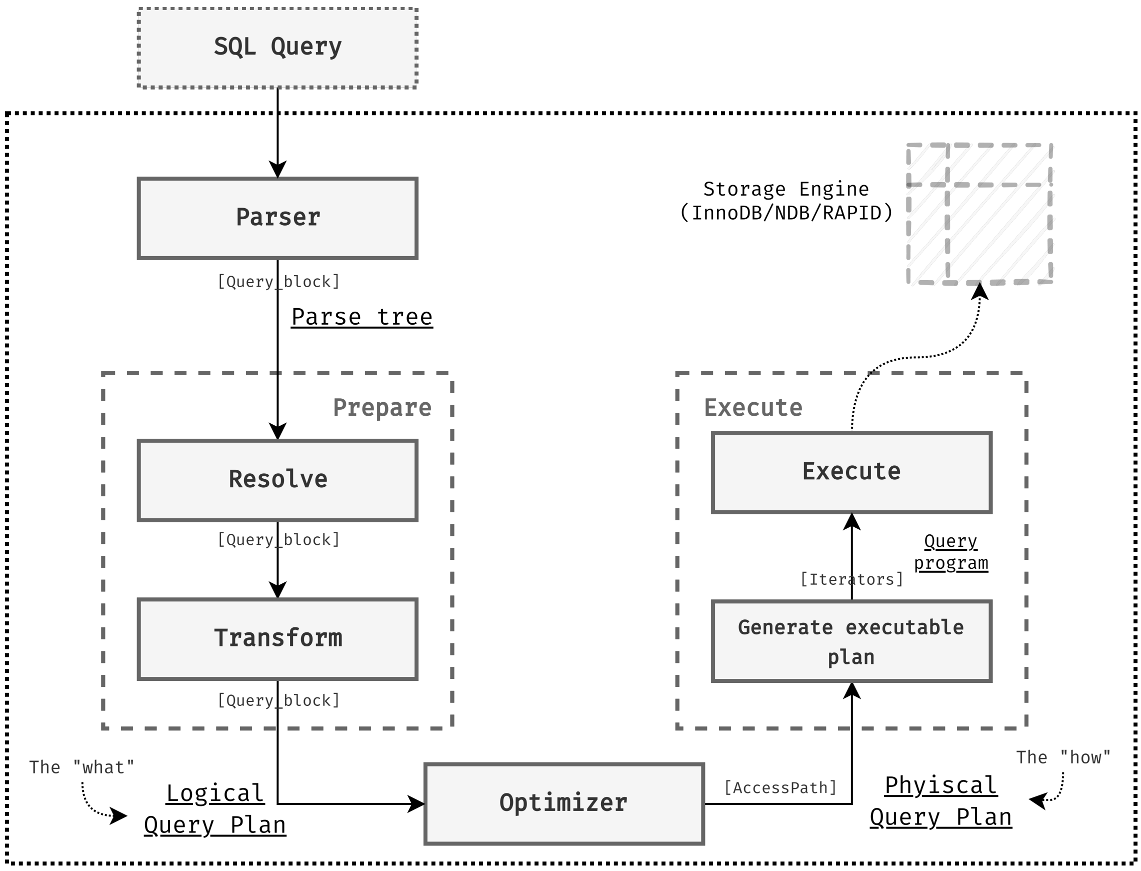


Figure 2.8: The flow of the MySQL query engine, showing the relationship between the parser, the prepare step, the optimizer and the executor.

Prepare

The next step is *preparation*, which is the MySQL terminology for resolving and transformation. At this point, the `Query_block` does not reference the storage locations of the tables and columns involved. These references will be resolved at this step, where, for example, the `Item_field` instances are initialized to the first row of the corresponding table, pointing to the storage location of the value of the first column. Additionally, this is the point where type checks are also applied.

Another important aspect of the preparation step is transformation, which simplifies the contents of the `Query_block`. The most important transformation to note in this thesis is simplifying the `Item`-trees. The `Item` expressions can be optimized to reduce the computational heights of the evaluation, for example, by applying *constant folding* and *dead code elimination*. For instance, the `Item`-tree equivalent of the expression `age = 20 + 25` will compute `20+25` for every row, which is not necessary. Instead, the MySQL query engine applies constant folding and reduces the arithmetic expression to `45` before execution, reducing the computational burden at runtime.

Another optimization technique MySQL uses is to eliminate dead code, removing `Item` logic that is considered redundant. The redundant logic could be subexpressions that always evaluate true, such as `20 > 10`. There is no need to compute this expression at runtime, and it can be removed from the `Item` tree before execution.

This optimization also applies to the opposite example for expressions always evaluated false. Such expressions make it possible to discard parts of the `Item` expression or the entire query since a conditional expression that evaluates to false always returns an empty set of rows. Moreover, these optimizations on `Items` are mainly performed on expressions of the `WHERE` and `HAVING` clauses, as the selection has the most impact on runtime performance. Figure 2.9 shows an example of the simplification of an `Item`-tree from the `WHERE` clause. After resolving and transforming the different parts of the `Query_block`, the final output of this phase is called a *logical query plan* [39].

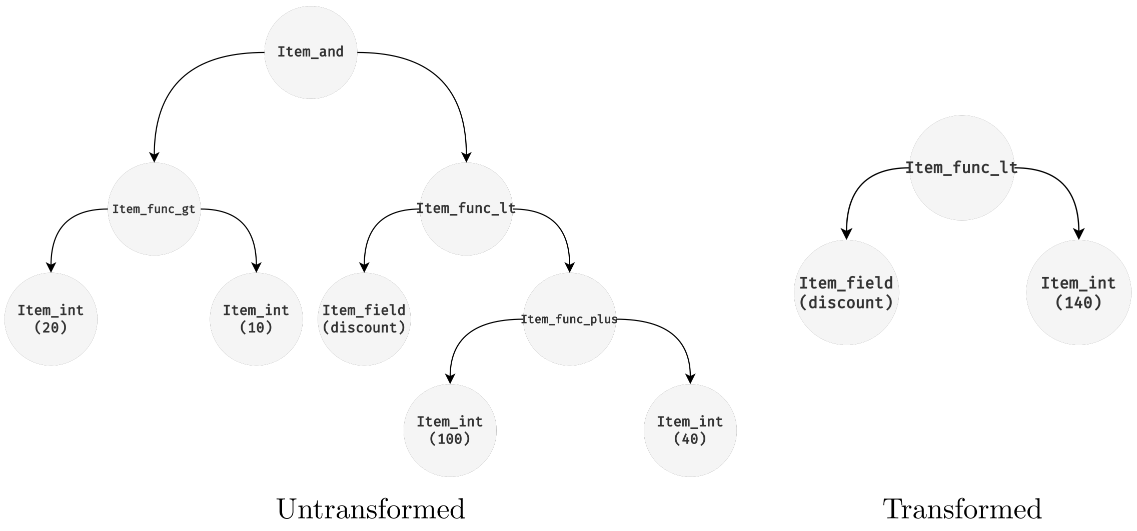


Figure 2.9: Illustration of the `Item`-tree transformation of the expression `20 > 10 AND discount < 100 + 40` during *prepare*.

Optimizer

The next phase of the process is in the hands of the *optimizer*. As of MySQL version 8.0.28, there exist two optimizers; a default traditional optimizer and a new one called *hypergraph optimizer*. Since the new one is considered the default optimizer in the future [40], we have chosen to use the new one in this thesis. As a result, we hereby refer to the *hypergraph optimizer* when talking about the *optimizer*.

The optimizer aims to produce a *physical query plan*, which is a tree of relational algebra operators that the executor can execute [38, pg. 169]. The first step of this process is to transform the `Query_block` into a tree of `AccessPaths`. An `AccessPath` is a data structure that is used by the optimizer to represent a relational algebra operator and is also used to build the tree that represents a possible query plan [33]. In addition, the `AccessPath` related to the expressions of the `WHERE` clause is called `FILTER`, which is an `AccessPath` containing a reference to an `Item`-tree.

The optimizer's biggest challenge is to build an optimal query plan for this structure in a reasonable amount of time. In MySQL terminology, a query is defined by a set of *joins* and is considered the most expensive operation during execution [38, pg. 172]. From a mathematical perspective, a join is a Cartesian product of the MySQL tables, where the table's records are extracted using different access techniques, such as key range lookups, full index scans, and full table scans. For expressions of the `WHERE` clause, they are usually performed jointly with table scans, which is one of the most time-consuming processes during execution. Therefore, the optimizer must choose an optimal combination of access paths that minimizes execution cost.

In simple terms, this is done in MySQL by representing the joins of a query as a *hypergraph*. A hypergraph is a graph representation where relations are connected through *hyperedges*, which connects a set of relations with another. For each relation, the optimizer investigates all possible access techniques to access the relation and goes through all join orderings to join the relations, using a bottom-up approach. This way, the optimizer enumerates all possible join orderings and builds up a table of candidate sub plans. The best plan is found by minimizing a cost model that calculates the cost of each join and other cost properties, such as cardinality estimates. The MySQL optimizer searches through all the possible plans and selects the one with the lowest cost for further execution.

This entire search is performed by constructing possible trees of `AccessPaths`. In this process, the `Item`-trees are involved in estimating an evaluation cost of the expression, where the root `Items` issues its execution cost to the optimizer. When the optimizer has completed its search, the output represents the final execution plan for the given query, in the form of a tree of `AccessPaths`, called a *physical query plan*.

Execute

The last step is for the executor to execute the *physical query plan*. However, the plan is not in its executable form at this point, as the `AccessPath` structures are only used during planning. As mentioned, MySQL executes queries through the Volcano iterator model [39], where every relational algebra operator is represented as an iterator. In MySQL, there is a one-to-one mapping between a tree of `AccessPaths` and a tree of iterators, and performing this mapping is the last step of the optimizer.

The iterators are built by a chain of `Read()` calls to the successive iterators, processing one row at a time. An iterator that is important to note is the `FilterIterator`, which is the iterator representing the relational algebra *selection* operation and has the job of emitting rows that satisfy the conditional expression of a given `Item`-tree. A call to `Read()` on a `FilterIterator` initiates a continuous loop on the rows of a table and evaluates the `Item` expressions on each row [33]. This process ends when there is a row that satisfies the expression, where each row is evaluated by calling `val_int()` on the `Item`-tree.

The root iterator receives rows one-by-one from its successive iterators and stores the results in a dedicated buffer pool that gets sent to the client when full. In total, the executor executes the query by a chain of `Read()` calls from the root iterator that repeats until there are no more records to process.

The life of `Items`

Since this thesis focuses on expressions in MySQL, we can summarize the life of `Items` in the following steps:

1. The incoming query is parsed by the parser, which extracts the semantics of the expressions and constructs a tree of `Items`. The `Items` are then stored in the `Query_block` structure.

2. In the *prepare* phase, the `Items` are transformed and optimized into less computationally heavy representations. The most common optimizations are *constant folding* and *dead code elimination*.

3. The `Items` is then barely used by the optimizer to find an optimal query plan, where the `Item`-trees emit the expensiveness of their execution.

4. The `Items` representing the `WHERE` clause are put inside of an `AccessPath` representing the *selection* operation. The original `Items`-tree from the parser might be split into multiple `AccessPaths`, as subexpressions can be relevant for different parts of a query plan.

5. The `AccessPaths` are transformed into `FilterIterators`, which references their own `Item`-tree.

6. The referenced `Item`-trees are used during execution, where a call to the iterator's `Read()` causes a loop of row-memory lookups where every row gets evaluated by a call to `val_int()`.

2.3.3 Drawbacks

Something worth noting is that the MySQL execution engine is far from perfect, and there are some drawbacks to the presented implementation. With a focus on expressions, one of these drawbacks is the implementation of `Items`, where some derivations maintain their state. For example, `Item_field` internally maintains a state for dealing with `NULL` values and implements reset mechanisms when switching between rows to be processed. Since `Items` are not necessarily stateless, it is infeasible for multiple threads to work on the same `Item` tree, causing an obstacle in evaluating expressions in parallel. MySQL does not support multithreading for a single query, as the implementation of `Item` is considered one of the barriers.

Another drawback is the usage of the iterator model, which is considered CPU-unfriendly due to the overhead produced by the chain of `Read()` calls. The same argument can be used for the implementation of `Items`, since the evaluation process also consists of a chain of `val_int()` calls. This chain of function calls introduces a large set of overhead, as every function call introduces a set of expenses, such as storing a return address to the stack, loading function parameters, and switching program counter. A function call also amplifies cache misses, as the program code of the function is not necessarily in the cache due to code locality and might initiate extra memory accesses.

Moreover, this entire evaluation process is not considered CPU-friendly, as the code is executed in a non-linear fashion. By using these `Item` trees, the evaluation of an expression in MySQL introduces many additional instructions that slow down the entire evaluation process, intensifying for every row that needs to be evaluated. For this reason, this is where we see an opportunity to introduce JIT compilation. By compiling the expressions, we can remove the need for the abstractions of `Items` by generating linear code for the expressions without performing any function calls. This way, we can represent the expression without the additional bloat and execute more CPU, cache, and memory-friendly code.

2.4 LLVM

A significant focus of this thesis is how we use the LLVM compiler framework to JIT compile expressions in MySQL. Therefore, having some knowledge of LLVM is considered necessary for understanding our work and the remainder of this thesis. For this reason, this section will briefly introduce essential concepts relevant to this thesis and how we can use them to achieve JIT compilation in MySQL.

LLVM began as a research project at the University of Illinois but has since evolved into a large open-source collection of compiler-related tools. Notable modules of this collection will be presented later in this section. The modularity and scope of the LLVM project mean it has become one of the go-to tooling collections for several projects, all from open-source, commercial, and academic projects. One of the notable users is Apple, which uses LLVM in everything from its operating systems to its development environments. Another example of notable use cases for LLVM is its use in programming languages; languages like Julia and Rust use LLVM extensively. [28].

2.4.1 LLVM IR

At the core of LLVM is the *intermediate representation* (IR), called LLVM IR. LLVM IR is an assembly near language optimized for applying transformations, pointer modifications, and analysis [26]. LLVM IR is built up with its own instruction set that is platform-independent. It comes in three forms; it can be stored as in-memory compiler IR, human-readable assembly language, or on-disk bitcode.

```
extern "C" int int_clause(bool clause, int a, int b) {
    return clause ? a : b;
}
```

Listing~2.2: Example C++ code

```
; Function Attrs: alwaysinline mustprogress nofree norecurse nosync
    nounwind readnone ssp uwtable willreturn
define i32 @int_clause(i1 zeroext %0, i32 %1, i32 %2)
   local_unnamed_addr #0 {
  %4 = select i1 %0, i32 %1, i32 %2
  ret i32 %4
}
```

Listing~2.3: LLVM IR created by Clang from 2.2

2.4.2 Notable Parts of LLVM

LLVM Core - As the foundation of the LLVM project, LLVM Core includes libraries that most other modules of the LLVM project use. These libraries include the LLVM IR optimizer and code generation for several CPU architectures. The various libraries are meant to be used in conjunction with LLVM IR. LLVM Core also includes helper methods and data structures to help in LLVM IR generation and loading. LLVM Core could make up the backbone of a complete compiler; from the frontend, optimizer, and backend. The libraries can also be used when creating a JIT implementation.

Clang - Clang is the LLVM project's answer to GCC. It is a complete C/C++ compiler and competes on compilation speed with GCC. Clang has become the default C/C++ compiler on MacOS. The goal of Clang is to provide a fast compiler that can give better and more insightful warning and error messages than GCC can provide.

LLDB - LLDB is the LLVM project's answer to the GNU Debugger (GDB) and aims to be a faster and better memory efficiency code debugger than GDB.

2.4.3 Use cases

The use case of LLVM that is the most prominent is probably to use LLVM to create frontends for programming language compilers; Rust, Swift, and Haskell have compiler frontends written with LLVM. LLVM is a popular choice for creating compiler frontends because it allows language creators to focus on the language itself and outsource creating an efficient compiler optimizer/backend to LLVM. Outsourcing work to LLVM allows more and more languages to create very performant binaries with advanced optimizations that before would only be possible for large teams with very specialized programmers.

2.4.4 LLVM for JIT

A use case of LLVM that is relevant for this thesis is JIT. When tooling like LLVM exists that can do all the work of a compiler optimizer/backend, creating a JIT system becomes doable for many projects. Essentially, creating a JIT system means creating a compiler frontend that runs at runtime and utilizes knowledge only known at runtime to create more efficient binaries than would otherwise be possible. The difference between a compiler frontend and a JIT system is that the JIT system creates IR and only stores it in memory instead of into a file. The system then needs to link any function calls outside the created IR dynamically. After linking and compiling, the system can execute compiled functions by getting their place in memory. Fundamentally, the JIT system has only skipped the writing to file step of compilation and could use runtime knowledge to optimize the code further.

How to JIT

In this thesis, we use the collection of LLVM libraries called *On Request Compilation* (ORC) to implement our JIT. In LLVM, a JIT engine is a pipeline built based on *layers*, where typical layers are a *compiler layer* and a *linking layer*. Such layers are connected to create an entire pipeline that inputs LLVM IR and outputs a specific target code. This flow is illustrated in Figure 2.10, and in the context of this thesis, the target code is native machine code that is stored in memory for later execution.

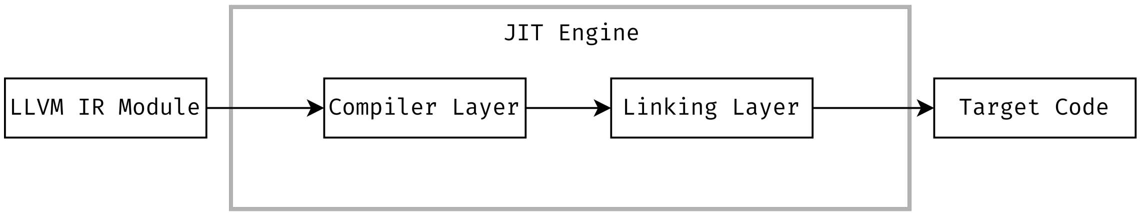


Figure 2.10: The layers of a typical LLVM JIT engine.

Creating such a JIT engine starts with an entity called `ExecutionSession`, which represents a JIT-compiled program. This entity is responsible for the memory associated with the compiled code and contains the *symbol table* for the program's symbols. The symbol table is represented by the entity `JITDylib`.

Moreover, `ExecutionSession` is known to all layers in the JIT engine and is used to produce the final target code, which will be available from the `ExecutionSession`. It is important to note that the `ExecutionSession` triggers the entire pipeline. A symbol lookup in this entity will eagerly initiate the compiler layer, which compiles the code, then initiates the linking layer, and finally outputs some representation of the looked-up symbol. In this thesis, this symbol will always be a function, where the output is a function pointer to a function created in LLVM IR.

To be able to use the JIT engine, our system needs to be able to generate LLVM IR. LLVM provides a target-independent code generator in the form of a C++ library called *IR Builder*. The *IR Builder* makes it possible to create and mutate an LLVM IR *module* in code, which is an in-memory representation of the IR. This library makes it easy to generate LLVM instructions in an existing program and conveniently integrates with an LLVM JIT engine.

The JIT system presented in this thesis uses the LLVM components described here. We use the *IRBuilder* to generate LLVM IR, and our JIT engine is built around the `ExectionSession` entity, a compiler layer, and a linking layer. With these components, the LLVM compiler framework makes implementing a JIT inside MySQL a feasible task, where most of the complexity is handled by LLVM.

2.5 Existing solutions / state of the art

Database research is an actively researched field, and query compilation has seen increasing adoption and advances. In this section, existing solutions and the state-of-the-art of query compilation will be presented. The section will explore how various databases have implemented JIT and the experience gained from the implementation. Various strategies have been utilized with varying strengths and weaknesses, from Microsoft and their Hekaton database to MySQL's probably largest competitor PostgreSQL to Umbra and Impala. This section will present a novel approach to JIT execution engines created by [23]. Research on how to compile query plans to get faster execution has been ongoing since about 2011 [35]. This research is increasingly relevant because as more and more data can fit in memory; databases are shifting the performance bottleneck from IO-operations to CPU performance [13, pg. 1]. Ultimately, this section will help understand how compilation is being used to increase query performance and provide ideas on how compilation could help get better query performance in MySQL. There are, however, several valuable contributions to this field we will not look at in more detail but have been influential in understanding how JIT can be implemented in databases. These contributions include commercial databases such as Hekaton, Cloudera Impala, SingleStore, and Spark. They include several academic research and open-source database systems like Vectorwise, Hyper, Hique, Resql, Dbtoaster, Legobase, and MonetDB. [13, 49, 6, 1, 41, 12, 24, 17, 2, 22, 47, 31, 35, 44, 23, 46]

2.5.1 PostgreSQL

Several papers have focused on implementing JIT in PostgreSQL [5, 30], but this section will focus on the implementation that is part of the main PostgreSQL repository. As one of the largest competitors to MySQL and a comparable database, many choices are probably relevant for a MySQL implementation. PostgreSQL began implementing JIT in December 2016 when Andres Freund was looking into getting faster expression processing and tuple deforming. PostgreSQL has chosen to base its JIT implementation on the LLVM toolchain, but they have taken the precaution of creating an LLVM-independent JIT wrapper to change it out later on if necessary. In doing so, they also support pre-compiling C functions as LLVM IR templates using Clang. [15].

Why did they decide to JIT

Freund [15] describes that while working on "batched execution", he concluded that the method did not create a large enough benefit towards the performance of large queries, and he describes that expression evaluation and tuple deforming are massive bottlenecks that need to be addressed. Freund explains how tuple deforming uses unpredictable branching, mostly because of having to deal with a bunch of different field types. He also explains that expression evaluation before JIT in PostgreSQL puts much pressure on the stack because of its recursive nature, which means it has to make many indirect function calls. Any LLVM-dependent code is included as a shared library loaded on-demand by an LLVM-independent wrapper. This way, it is relatively easy to swap out LLVM with different JIT tooling should that be wanted. [16]

Where is JIT applied & how does it work

PostgreSQL has implemented two operations that JIT compilation can accelerate at the time of writing. One is expression evaluation, and the other is tuple deforming. Expression evaluation is the function of taking expressions like `WHERE field > 2` and evaluating them for every row to check if the row should be part of the result or not. For PostgreSQL, JIT compilation on expression evaluation will compile WHERE clauses, target lists, aggregates, and projections. Tuple deformation is taking an on-disk tuple (or row) and transforming it into the in-memory representation of that tuple. The tuple deformation will use information about the table being queried and the columns it contains, which is only known at runtime. The JIT implementation in PostgreSQL also supports inlining smaller function calls made in a query as a bonus JIT optimization if the estimated cost of the query is higher than a set variable.

How PostgreSQL decides if it should JIT

PostgreSQL has defined three variables that determine if it should apply JIT to a query. These variables are: `jit_above_cost`, `jit_optimize_above_cost` and `jit_inline_above_cost` which are compared against the estimated cost of the query. PostgreSQL uses these variables as thresholds to choose if it should use JIT or not. These thresholds work by comparing the variables with cost estimates for the query. Setting thresholds like this is not easy, as some will perform better with certain queries than others. By default, some thresholds are chosen that should, in most cases, allow JIT to perform better, but this should be tested with the given workloads being used. The default thresholds work quite well for TPC-H, with no obvious downside.

Why LLVM

PostgreSQL states that their reasoning for choosing LLVM is that LLVM is being developed and used by several large corporations, making it unlikely to become discontinued. The LLVM license is also compatible with PostgreSQL's, and lastly, that LLVM IR can be generated from C code using the Clang compiler. IR from C generation, in turn, allows for inlining functions not specifically made for the JIT implementation. [16]

Results

Over the TPC-H benchmark PostgreSQL consistently shows that having JIT enabled gives a speedup of about 20% on scale factor 10. In the case of query 14, JIT gave an incredible speedup of about 90%. [20]

Summary & how this is relevant to the research question

PostgreSQL has implemented JIT with LLVM tooling and uses a couple of variables to determine when to apply JIT. This form of implementation is perhaps the most straightforward implementation of the existing solutions we are looking at. PostgreSQL and MySQL are pretty similar. They both use a volcano evaluation system, so we consider the speedup shown through JIT benchmarks made on PostgreSQL to be achievable in MySQL. Because of this, the benchmarks made on PostgreSQL become a goal for our JIT implementation and a way to compare if our implementation achieves the performance we hypothesize should be possible.

2.5.2 Hekaton

Hekaton is a database management system (DBMS) optimized for memory-resident data and OLTP (Online Transaction Processing) and is fully integrated into Microsofts SQL Server. OLTP is data processing consisting of executing numerous transactions concurrently. The team behind Hekaton noticed a trend where main memory became less expensive, and CPU usage across multiple cores increasingly became the bottleneck in DBMSs. Therefore, Hekaton was created to take full advantage of large main memory and many-core CPU systems.

Implementation goals

Hekaton sought to achieve massive speedups in the magnitude of 10-100x. Early analysis of throughput improvement showed that reducing instructions was the only optimization that could be implemented to reach these performance goals. [13]

How does Hekatons JIT work

Hekatons JIT implementation is quite different from the other databases this thesis presents. It is different in that it does not use an IR close to machine code but instead uses C code as the intermediate step, which is compiled by the Microsoft Visual C/C++ compiler. This choice can utilize many of the existing toolsets that Microsoft has created around their SQL server. Hekaton reuses the SQL server T-SQL compiler stack (metadata, parser, name resolution, type derivation, and query optimization) and the already mentioned Microsoft Visual C/C++ compiler. [13, 10]

The codebase of Hekaton is written in C/C++, which allows it to inject callback functions to the query engine in SQLServer, which they have reused. Injecting the callback functions means Hekaton supports hybrid execution, meaning running code that has not been part of the JIT process alongside code that has. Hybrid execution makes it easier to implement JIT compilation as not everything has to be rewritten for it all to work. [13, 10]

Results

The compiled code could reduce the number of instructions by up to 10x. Hekaton was not tested on TPC-H but instead on their benchmark. Over the various benchmarks and number of cores used, the compiled code increases throughput by over an order of magnitude. [13, 10]

Summary & how this is relevant to the research question

The research on Hekaton focuses on the importance of reducing the number of instructions of a query and shows how JIT compilation can help achieve this. It also shows an approach to this problem that allows hybrid execution. Since MySQL is also written in C++, this approach seems optimal as it would allow a much easier transition to a complete JIT implementation. Hybrid execution allows a rolling release of the implementation where the more manageable parts can be released before all the more difficult-to-develop parts are ready. Furthermore, using this system also allows parts of the MySQL query not to have a JIT implementation if it does not make sense in a performance or maintenance consideration.

2.5.3 Cloudera Impala

Cloudera Impala is a Massively Parallel Processing (MPP) database made for use in the Hadoop framework. Hadoop is a framework for distributed processing of large datasets across clusters of computers. Impala does runtime code generation using LLVM. Impala generates query-specific functions at runtime, meaning it can make optimizations based on knowledge only available at runtime, like knowledge about column types and expression operators. Because Impala is meant to be used on tables with trillions of rows, even a few saved instructions will have a massive impact on the overall performance of the query. Impala is also meant to be used for analytic workloads instead of OLTP (Online Transaction Processing), meaning it is made for performing complex, long-running, CPU-bound queries instead of several concurrent queries [49, 48].

Why LLVM

Impala uses LLVM because it allows Impala to perform JIT compilation within a running process, with all the benefits that come from a modern compiler and optimizer. LLVM also supports several different architectures [49].

Where does Impala apply JIT

Impala applies JIT on "inner loop" functions, meaning functions run several times as part of a loop. JIT compilation is applied explicitly to these "inner loop" functions because they make up most of the total instructions run and time spent executing a query. An example of such a function is a function used for tuple-deforming, which parses the on-disk record into the in-memory format used by Impala [49]. Impala also applies JIT to remove conditionals, propagate constant offsets and pointers, and lastly, inlining of virtual function calls [48].

UDFs

Impala also supports User-Defined Functions (UDFs), allowing users to create custom functions for queries. The UDFs are defined as C++ functions. Impala can therefore use another feature of the LLVM ecosystem, which is to use Clang to compile these C++ functions into LLVM IR. Which, in turn, can be inlined and optimized alongside the other LLVM IR generated code [49, 48]. The IR templating allows users to write functions without worrying about slowing down the query execution with added complexity through abstraction.

Results

Impala tested their JIT implementation on the TPC-DS benchmark on a scale factor of 1 terabyte. Results are shown as a compilation of the queries in this benchmark and query execution time with JIT turned off and turned on. On such a large scale factor it is expected that JIT should always be able to outperform the interpreted alternative, and in their results, this is confirmed. Impala is in their tests consistently faster with JIT enabled. The authors even describe how the JIT results could be even faster and are being held back by not every code path being able to take advantage of code generation.

Summary & how this is relevant to the research question

In summary, Impala has shown how LLVM can be used as a tool for JIT in databases. They have shown that for long-running CPU-bound queries utilizing JIT on "inner loop" functions can make a massive difference to the overall performance of the query. Impala has also shown a strategy for implementing JIT through

C/C++ templates that can be compiled into LLVM IR, thus making it easier to debug JIT functions since debugging LLVM IR can be pretty tricky.

2.5.4 Adaptive execution

Kohn, Leis, and Neumann [23] presents a framework for adaptive execution of compiled queries. The framework builds upon HyPer, which is an in-memory DBMS. HyPer produces LLVM IR for a query and compiles the query before execution, meaning it already supports a form of JIT built on LLVM. Before the JIT implementation, every query must be compiled through LLVM. When a query is computer-generated and thus lengthy, compiling it could take several seconds. HyPer would also have to compile queries when JIT would be slower than an interpreted approach. The adaptive execution framework tries to solve this problem by introducing interpretation to HyPer. [23]

Execution modes

The framework makes the most of JIT by only spending compilation time on the query parts that benefit from it instead of compiling the entire query. The framework divides each query up into several *morsels*, which are small pieces of work needed to execute on a row. Each *morsel* can be executed in three different execution modes, and the framework will shift which morsels are run on which execution mode to optimize the runtime. The three execution modes the framework introduces are:

bytecode - This execution mode provides the quickest way for the framework to take a query and start executing it. The *morsel* is quickly compiled into bytecode which Kohn, Leis, and Neumann wrote a custom VM to achieve. The bytecode compiler was made to compile the *morsel* fast and thus only implements a few handpicked linear optimizations that do not take long to apply to the bytecode.

Unoptimized compiled code - This execution mode provides the “golden middle way” between fast-to-execution and fast-execution *morsel* execution. The *morsel* is compiled using LLVM with only a few handpicked linear optimizations applied.

Optimized compiled code - This execution mode provides the fastest execution of all the modes at the cost of compile time. This time a *morsel* is also compiled using LLVM but has several optimizations. Some of these optimizations are also super-linear, meaning they do not scale linearly with the amount of code needed to compile.

How the framework determines which execution mode to choose

The adaptive framework will always start executing *morsels* in the *bytecode* execution mode. Always executing with *bytecode* first ensures that the framework will start working on the query as quickly as possible. Then while running, the framework will look at how much time each *morsel* uses and how often they are run to

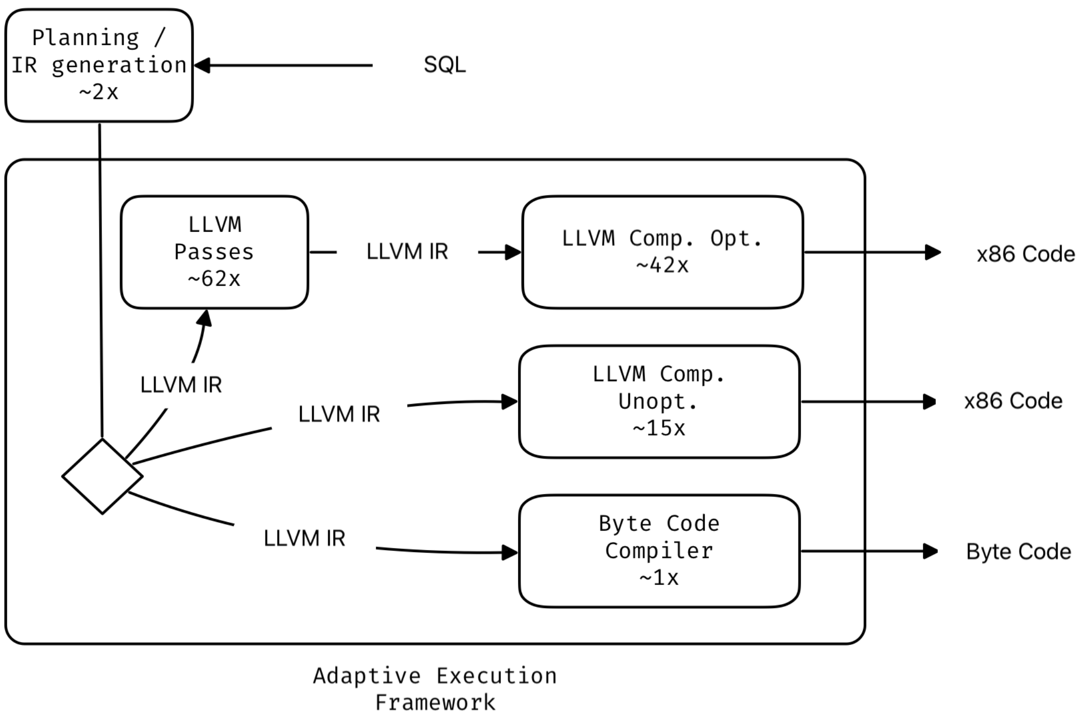


Figure 2.11: Adaptive Execution Framework Overview

determine if JIT should be applied to the *morsel*. Since HyPer is multithreaded, the framework will use a single thread for compilation while running *morsels* in whatever execution mode that is available and faster than the other available modes. The result is that the framework can dynamically change the execution mode of a *morsel* when other faster execution modes are available. The framework is tuned on TPCH, so it is unclear how their chosen limits for execution mode choice would be affected on other benchmarks. [23]

What optimizations are applied

Kohn, Leis, and Neumann [23] had to make special considerations when choosing what optimizations to apply on the LLVM IR. They argue that many optimizations do not benefit database query execution, so they can be ignored with little effect on the overall performance. Another consideration Kohn, Leis, and Neumann brings up is how in a JIT environment where compilation time makes a massive difference to the usefulness of the feature optimizations that do not scale linearly have to be applied with care. Thus, an adaptive execution framework needs to consider the amount of time compiling a *morsel* takes so that the compilation itself does not take excessive time. Excessive is when more time is spent compiling than the time gained from running the compiled code versus the interpreted code, ultimately removing any performance benefit that JIT can bring. This problem is critical to solving for a JIT implementation that has to decide before execution if it should apply JIT, an example of this being PostgreSQL. Since deciding to JIT a query that takes long to compile could mean that the user has to wait longer for compilation than the amount of time, the query would execute in the interpreted execution mode.

Results

The framework was tested on the TPCH benchmark. On scale factors ranging from 0.01 to 30. Kohn, Leis, and Neumann measured the median query time, and on that metric, the adaptive execution framework consistently beat all the other test setups running exclusively on a single execution mode. On the smallest scale factor (0.01), the framework spent equivalent time as the exclusively interpreted execution mode, which was expected. The framework should not expect any benefit from applying JIT on such a small dataset and thus not do it. Between scale factors 0.1 and 1, there was a crossing point between interpretation and unoptimized compilation exclusive execution modes. However, at that crossing point, the adaptive execution framework beat interpretation and unoptimized compilation. It beat both because it could apply unoptimized compilation to only some *morsels* instead of all. It results in less time compiling than the unoptimized compilation exclusive execution mode.

Summary & how this is relevant to the research question

The framework shows that building a system that circumvents the need for volcano-based execution by interchangeably interpreting and compiling IR is possible. The result of that system is a more performant query system since the system can better utilize the CPU, which ultimately means more of the speed capacity of the storage system can be used. For our research question, the reasoning and results gathered by Kohn, Leis, and Neumann on what optimizations to use when applying JIT are especially relevant. They shed light on the problem of how optimizing too much will ultimately reduce overall performance when applying JIT. Thus, testing how optimizations affect performance becomes essential for us to see what performance JIT can bring for MySQL correctly.

2.5.5 Umbra

The trend of falling DRAM pricing, which allowed for vast amounts of memory to be used by databases, fueled the creation of the HyPer database, an exclusively in-memory database. Since then, however, falling DRAM pricing has stopped, and pricing for fast storage, SSDs, is becoming cheaper and cheaper. These falling prices have created new possibilities for high-performance databases, which Umbra takes advantage of. Umbra is an evolution on HyPer and gets the most out of high-memory fast-storage systems by taking advantage of the performance an in-memory working set brings and the scalability of SSD storage.

Implementation goals

Umbra takes the adaptive runtime approach introduced in Kohn, Leis, and Neumann [23] for HyPer. The most significant difference between Umbra and HyPer is that Umbra will always compile code and does not use an interpreter. Umbra achieves this by not always using LLVM to compile but instead using a custom backend called Flying Start. Flying Start is optimized to compile Umbra IR fast. Using this approach, Umbra does not waste time transpiling between its custom IR and LLVM IR before execution, as HyPer would need. Umbra also differs from HyPer in dividing the work needed for a query.

How does Umbra JIT work

Consider a query like `SELECT COUNT(*)FROM supplier GROUP BY s_nationkey`, this query consists of two pipelines. The first pipeline scans the `supplier` table and does a `GROUP BY` operation on the result. The second pipeline will scan the output from the first pipeline and output the query result [36]. Umbra will separate these pipelines into smaller steps that it can compile separately. The Umbra query executor can choose which steps to run with the Flying Start backend and which steps benefit from further optimization through LLVM compilation [36].

Results

The results shown in Kersten, Leis, and Neumann [21] show that combining Umbra IR and Flying Start significantly reduces query latency compared to HyPer. Over the 22 TPC-H queries, Umbra has significantly higher queries per second, and Flying Start compilation beats databases that do not spend time compiling code even on scale factor 0.001.

Summary & how this is relevant to the research question

Umbra can be described as an evolution of HyPer and the adaptive framework it employs. It takes the ideas introduced in HyPer and refines them. The architecture of Umbra is designed to take full advantage of high memory in combination with fast storage, in line with the trend of lower and lower SSD costs. Combined with the custom compiler backend Flying Start, Umbra IR allows Umbra to achieve a quick time-to-query execution start while maintaining most of the benefits of running compiled code for query execution. Umbra shows us that JIT is very applicable when taking advantage of what modern systems can provide. Moreover, that interpretation can be beaten on small datasets if some thought is put into crafting a custom IR and compiler backend optimized for databases.

Chapter 3

MySQL JIT implemention

This chapter will describe our implementation of just-in-time compiling expressions in MySQL using the LLVM compiler framework. First, we will discuss different methods for implementing JIT based on existing work and argue what approach is best for MySQL and why we chose LLVM. Further, we will explain how our just-in-time implementation fits within the "life of a query" in MySQL and the existing query optimizer. The focal point is to show how the implementation interacts with the existing code concepts and entities. Afterward, the following section will explain the code generation process from `Item` expressions to LLVM IR. The last section will describe the compilation step and how the compiled code is run during execution.

This chapter aims to provide a detailed description of our implementation and a good understanding of why certain things were done the way they were. The purpose is to show how just-in-time compilation of expressions can be done within the complicated system of MySQL and express the possible performance outcomes of such an implementation.

3.1 Evaluating existing work

The existing work presented in 2.5 gives us a clear idea of both the challenges and possibilities of implementing JIT in a database. The Hekaton implementation [13] makes it clear that the most critical factor determining performance increase is the decrease in instructions run. Thus, instruction count can be viewed as an essential metric to track. A decrease in instruction count on account of our JIT execution will likely also indicate an increase in performance. PostgreSQL, Impala, and HyPer (used by the adaptive execution framework) show that LLVM can be used to implement JIT in databases and give a solid performance boost. Impala also shows that JIT should be focused on code that runs several times, like in "inner loop" functions.

Overall the state-of-the-art has shown just how important it can be for databases to utilize JIT, especially where large datasets are involved. The implementations show how JIT can be implemented so that it does not hurt the performance of queries where an interpreted approach would perform better. PostgreSQL shows how LLVM is a good choice for implementing JIT since it is maintained by several large companies and organizations that ensure LLVM sticks around and has the development capacity needed to maintain its features. PostgreSQL also shows that LLVM can be used in a way that does not make the database codebase inherently dependent on LLVM. PostgreSQL shows that a JIT implementation can be made so that any compiler tooling could be used if anything should happen to the LLVM project or the needs of PostgreSQL shift away from what LLVM can offer.

Interpretation alongside JIT

One approach in existing solutions is to utilize the created IR alongside a VM that can directly run the IR or a language similar to the IR. This approach generally allows for a more straightforward JIT implementation since the two execution modes utilize the same IR. HyPer, Umbra, and LegoBase use this approach. LLVM has a VM built-in that can run LLVM IR, but as HyPer showed, using that VM is very slow and not very efficient for use in a DBMS. HyPer also explained that LLVM IR is not very good at being interpreted by a VM, so for a DBMS, it is much better to use a custom-made IR that can be transpiled to or from LLVM IR. Umbra took this approach further by not transpiling from LLVM IR like HyPer but instead creating custom IR from the get-go that is then transpiled into LLVM IR for compilation. Creating custom IR from the get-go removes the time needed to transpile LLVM IR into the custom IR, resulting in less time before query execution starts. The results from running interpretation alongside JIT speak for themselves; being able to start executing the query quickly by interpreting and then using JIT to get better performance when applicable means the system gets the best of both worlds.

MySQL already has an interpretation-style execution mode through its volcano-based execution mode. However, unlike HyPer and Umbra (and possibly Legobase), MySQL does not use an assembly-like IR, meaning that compilable IR must be generated from existing data structures instead of a predefined language. If MySQL were to solve the problem of having state intertwined with these data structures, MySQL could implement a similar system to HyPer and Umbra.

For optimal performance, however, MySQL should also multithread query execution, allowing for using a thread for JIT compilation. Creating a system for multithreading for MySQL would also be significantly helped by separating state from the IR data structures.

In summary, while a system implemented in HyPer and Umbra would give the best performance and should be considered a long-term goal for MySQL, it is not feasible for this thesis. We also do not consider such a system integral to answering the research question.

Why LLVM

To implement JIT into a database, IR is needed, and a compiler for this IR is needed. Most existing solutions we have looked at use already-made tools for this. Impala and PostgreSQL use LLVM exclusively. Hekaton reuses the SQL server T-SQL compiler stack and the Microsoft Visual C/C++ compiler. The adaptive execution framework for HyPer and Umbra both use a combination of LLVM for their highest execution speed but longest compilation time and custom-written specialized solutions for quicker compilation but slower execution time. The choice for

HyPer and Umbra to use LLVM for their highest performance modes shows perhaps the biggest reason for not choosing to implement a custom IR and compiler from scratch. LLVM has shown itself to be good at optimizations and creating code that executes fast.

Moreover, being an open-source project with sponsors like Apple and Google [27], the project is likely to continue to grow and thrive. Thus, using LLVM can provide excellent performance, which requires significant effort to reproduce and maintain. The downside of LLVM is that it changes rapidly, and there are often breaking changes between versions, which adds the need for a bunch of maintenance and keeping track of how different versions of LLVM has to be used to achieve the same result. The downside of breaking versions is especially relevant for a JIT implementation in MySQL since MySQL provides eight years of support for a given released version. We want to show how JIT affects performance in MySQL and where it makes sense for MySQL to use it. So for this thesis, it makes more sense to use a toolchain that we know can provide excellent performance, making our results independent of our ability to create a performant IR and compiler system. Another upside to using LLVM for answering our research question is that it is already in use in production JIT systems in databases and has been for many years, meaning it is battle-tested.

Should MySQL ultimately decide to implement JIT, they would also need to determine if the license LLVM uses, the Apache 2 license, is compatible with their product. They would also have to figure out how to bundle LLVM with MySQL. PostgreSQL solves the bundling issue by not requiring LLVM. PostgreSQL has also wrapped its JIT implementation in an independent wrapper so that LLVM could be swapped out for another toolchain should the need arise.

Templating IR

From the existing implementation we looked at, we found that some use templating of IR to simplify IR generation. LLVM templating would involve using C or C++ as the higher-level language and Clang to output LLVM IR files as templates. Templating, practically, means writing functions in a higher-level language and then compiling this to IR at compile time.

In our initial testing, JIT with templates performed worse than the API approach, even though the templates included the same IR as the API approach created to our knowledge. We never discovered why the performance difference existed as we prioritized our time elsewhere since we did not consider templating an essential aspect of answering the research question. Ultimately, we did not do any templating and generated all our IR through the LLVM C++ API. However, we used the C functions and the resulting LLVM IR template files to inspire our LLVM API usage. By running the template files through all the possible optimizations in LLVM, we had a very optimized IR to base our generation on, resulting in our generator creating a very optimized IR at runtime.

Benchmarking on TPCH

There are several database benchmarks out there that test various aspects of databases. From looking at the existing solutions we presented, (all/most) of them use the TPCH benchmark. TPCH allows for creating datasets of a set scale equivalent to the gigabytes needed to store the dataset. This benchmark is ubiquitous in the industry as it provides a real-world benchmark for a typical company selling products. For our testing, TPCH includes a variety of queries that can show various aspects of how applying JIT can affect query performance.

From all these state-of-the-art implementations, we can derive a list of goals that our implementation should strive to achieve:

- Our implementation should decrease the number of instructions run in query execution
- Our implementation should only apply optimizations that make sense for the code we generate and thus not waste time on non-linear optimizations that do not make a huge difference to the performance
- Our implementation should be able to apply different optimizations based on metrics or configuration so that huge queries have an unreasonable compile time
- Our implementation should be able to compile parts of an expression and not only the entire expression. Compiling only parts of a query will help implementation since it would allow for a partial JIT implementation, and it could allow for only applying JIT to the parts of the expressions where it can make a difference.
- Our implementation should strive to be independent of LLVM or at least be built in a way where swapping out LLVM is straightforward and would not need a major refactor of the entire codebase.

3.2 The implementation flow of JIT

The main objective of our implementation is to transform the `WHERE` clause of a query into highly efficient machine code. The machine code should resolve to fewer instructions than the existing system and be available for execution inside the same process. Since the `WHERE` clause in MySQL gets transformed into one or several `Item` trees, our implementation consists of generating LLVM IR for each `Item` tree and letting LLVM handle the rest. In our implementation, we convert each `Item` tree into an LLVM IR *module*, and use `ExecutionSession` in LLVM to transform the *module* into native machine code and link it to the same process. However, the entire execution of filters in MySQL revolves around dealing with `Items`, and uses the `Item::val_int()` function call to decide if a row satisfies the `WHERE` expression in the query. Therefore, our implementation is entirely restricted to the existing logic of `Items`, where the just-in-time compiled code is run for each table row to evaluate.

MySQL is a database that supports many data types, such as integers, strings, dates, and complex geometry types, like points and polygons. Many of these types introduce a certain amount of complexity regarding their behavior in expressions. For instance, strings support different character sets and collations that must be handled correctly according to their standards. Therefore we think most of the data types and features inside MySQL are considered out of scope. Supporting all these data types, operators, functions, and different features supported by MySQL would not necessarily be of any substantial value for this research, in addition to being too time-consuming. We want to show how just-in-time compiling expressions can give any performance impact on MySQL by removing overhead from `Items`.

We have restricted our implementation to a few basic data types and operators, focusing on simplicity and those commonly used in many realistic queries. For that reason, our implementation will support the following types: `INT`, `CHAR`, and `VARCHAR`. Regarding the operators, we have chosen to support the basic logical operators of `AND`, `OR`, `NOT`, in addition to supporting the basic comparison operators of `>=?`, `<=?`, `BETWEEN`, `!=`, `=` and `LIKE`. We think this restriction indicates if a given JIT expression implementation in MySQL would benefit performance on realistic business-focused queries.

3.2.1 The life of a jitted expression

Our implementation introduces a set of extensions to the *life of a query* described in Section 2.3.2. As described, the MySQL parser transforms the `WHERE` clause of an input query into one or several `Item` trees stored inside a `Query_block`. From there, the *prepare* phase might make additional transformations on `Items` for optimization purposes, such as constant folding, which reduces the computational work for the given expressions. These `Item` trees are later referenced in `AccessPaths`, representing the different possible query plans for the query. The optimizer uses these referenced `Items` in calculating the cost of the plan and, in the end, chooses a final `AccessPath` that is suboptimal, or potentially optimal, for performance. Subsequently, the chosen `AccessPath` is converted into the form of `Iterators`, which is the data structure used during execution.

Here, the referenced `Items` from the `WHERE` clause are placed in instances of `FilterIterators`, which uses the `Item::val_int()` call on the root `Item` of the tree to determine if the current row satisfies the expression. This means that the `Item` trees referenced in `FilterIterators` preferably need to be compiled at some point before the `val_int()` call. Since modifications are made in `Items` during optimization and are later used in the search for a decent query plan. We believe that the best place to apply our logic for just-in-time compiling the `Items` is after the final `AccessPath` is chosen and during the creation of the `Iterators`. No further references to the `Items` are being used at this point in the query execution, making it the ideal location. The optimizer then completes all optimizations, and the `Item` trees will not be further changed, thus avoiding possible recompilations.

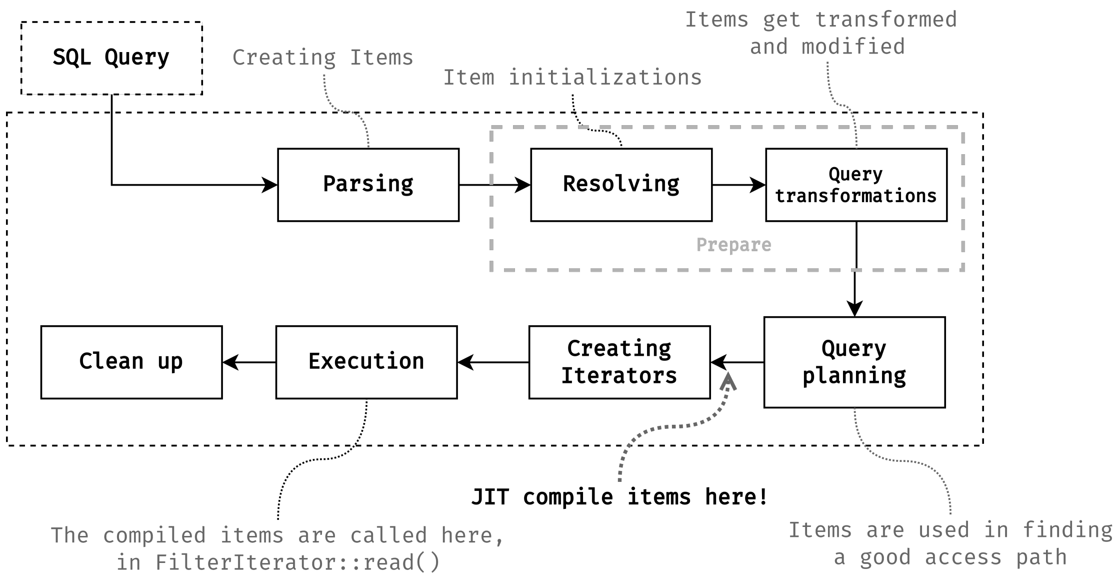


Figure 3.1: The important events for `Items` during the execution of a query

3.2.2 The building blocks

To implement this flow using the LLVM compiler framework, we have chosen to introduce some new entities into the play field: `JITExecutionEngine`, `JITBuilderContext` and `Item_compiled`.

The `JITExecutionEngine` is the entity that stands for the compiling of the `Items`, in addition, to handling the linking and storing the compiled functions in a symbol table. It is a wrapper around the set of LLVM entities and, most notably, contains a JIT session object, `ExeuctionSession`, that does eager in-memory compilation. The job of this entity is to receive LLVM IR *modules*, JIT compile the IR in memory, and make the final compiled function available for future lookups.

Next, the `JITBuilderContext` is a data object containing the required data for generating LLVM IR for a given `Item`. An instance of this entity is passed to every `Item` in an `Item` tree when compiled and is used during code generation.

Lastly, the most important entity of our implementation is the `Item_compiled` entity. `Item_compiled` is itself another derivation of `Item` and is a part of the `Item` family. It represents a JIT-compiled `Item` expression and can replace other `Item` references in the existing execution engine, such as the reference in a `FilterIterator`. This entity is a wrapper over an existing `Item` and contains a final function pointer representing the JIT-compiled code. For each `Item` expression that will be compiled, there will be created an `Item_compiled` instance for that tree, which will also replace all future references of that tree. This way, our implementation fits within the bounds of the existing logic for executing expressions in MySQL.

An instance of `Item_compiled` interacts closely with the other mentioned entities, `JITExecutionContext` and `JITBuilderContext`. `Item_compiled` creates and owns an instance of `JITBuilderContext`, and independently generates the LLVM IR for the wrapped `Item` and passes the product to an instance of the `JITExecutionContext`. With these three entities, we have a simple and intuitive API for building our JIT within the borders of MySQL.

3.2.3 Putting it all together

At this point, we have to some extent, described the where, when, and how of our implementation. Putting it all together, the main flow for just-in-time compiling expressions in MySQL goes as follows:

1. The parser will parse the incoming query and create a single `Item` tree for the expression in the `WHERE` clause.

2. The produced `Query_block` by the parser will be passed to the optimizer and start the resolving step - initializing table and column references in the `Item` tree and other internal variables.

3. The optimizer will further run a set of optimization passes on the `Items` to see if the expression can be simplified and turned into something less computational heavy. *Constant folding* and *dead code elimination* is typical optimizations here.

4. Further, the optimizer will use these `Items` in the search for a final `AccessPath` structure. Here, the `Item` tree expression can be split into multiple `Item` trees based on several filtering steps in the final query plan.

5. After choosing the final `AccessPath`, the optimizer will convert the `AccessPath` into a tree of `Iterators`. When creating a `FilterIterator`, we check if "JIT has been initialized" for the given query session. If not, we will allocate and initialize a `JITExecutionContext` for the given query session, stored globally in the thread-local storage, which means that each query session gets allocated its `JITExecutionContext` lazily for compiling all expressions in the query. We also do other initialization steps required by the LLVM compiler framework. The framework needs to know which environment and machine architecture to create code for and how it should link the JIT-compiled code correctly. Choosing the right environment and architecture is achieved by simply using a set of provided initialization functions from the core LLVM library.

6. Following the initialization step, the `Item` tree referenced in the `AccessPath` will be checked if it is *compilable* and supported by our implementation. If that is the case, we create a new instance of `Item_compiled`, wrapping the `Item`, and proceed with the process of generating code and JIT compiling the `Item` expression into a single function pointer. The compilation is done in collaboration with the newly allocated `JITExecutionContext`. The new `Item_compiled` will be stored inside the newly created `FilterIterator` instead of the original `Item`, and iterator is now ready for execution.

7. However, there is a chance that the whole `Item` tree is not compilable and supported by our implementation, but a subtree is. To support the compiling of these subtrees, we have developed a system that allows *rolling updates*, where we incrementally compile one subtree at a time. When the root `Item` indicates it is not compilable, we traverse the entire tree to check if any subtrees are compilable. If there are any such cases, we create a new instance of `Item_compiled` for each subtree and replace the child reference in the parent `Item`. This way, we can support partial JIT compiling of expressions and

do not necessarily need to support the full expression. The need for partial compilation arises, for example, when a part of the `WHERE` clause consists of an internal SQL function, such as `EXISTS(...)` and `LEN(...)`, which we do not support.

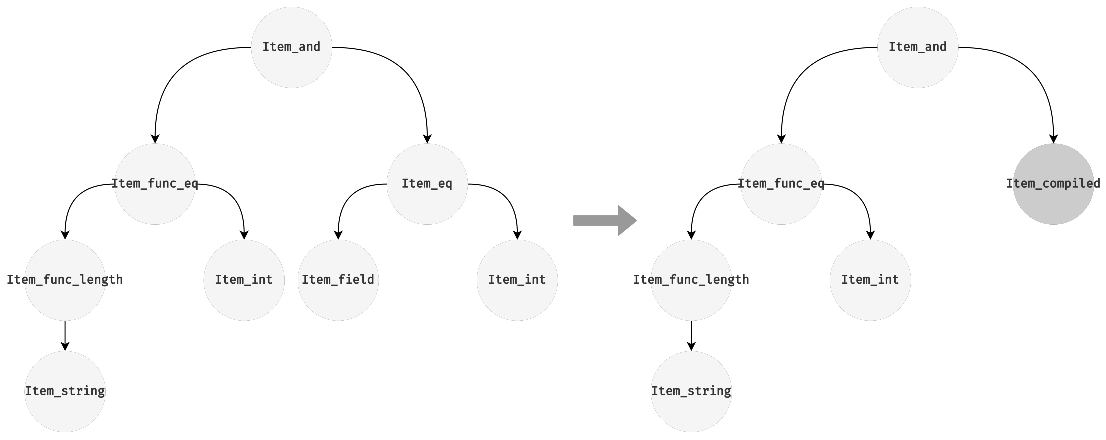


Figure 3.2: Example of compiling only a subset of the entire expression tree.

8. The next step is execution, where every call to `FilterIterator::Read()` will execute the `Item::val_int()` call on our `Item_compiled` instance, which calls upon the internal function pointer representing the JIT compiled code. During a full table scan, this `Read()` call is done for every row in the table.

9. After execution, the used memory allocated by the `JITExecutionContext` and `Item_compiled` is freed during the existing clean-up procedure for the query session. The `Item_compiled` instance is allocated using the built-in *arena allocator* for the query session and will be freed with the rest of the session.

The hypergraph optimizer

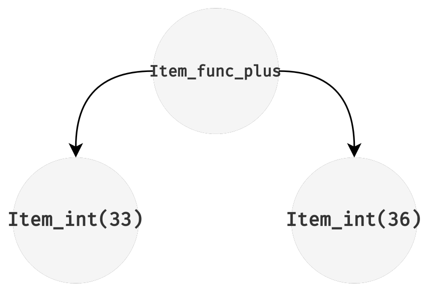
This flow describes how our implementation works in terms of entities and existing system concepts in MySQL. However, it is worth mentioning that the MySQL query engine supports multiple flows for executing a query. It supports multiple *optimizer switches* for toggling different optimizer behaviors defined by the optimizer, and one of them is the *hypergraph optimizer*. Currently, MySQL unofficially supports hypergraph-based query optimization [11], which is the concept of utilizing hypergraphs during the search for an optimal query plan, where each table in the query is represented as a node in the graph. The outcome of this feature is potentially a more efficient technique for finding a good query plan, often producing a better plan than the flow without it. Due to its better capabilities, we have decided to utilize this flow for our JIT implementation and measurement of all performance benchmarks. In addition, it is also planned to be the main MySQL optimizer in the future [40], which makes it a natural choice.

However, it is worth mentioning that at the time of writing, the hypergraph optimizer is still an experimental feature in MySQL (v8.0.28) and has not been officially launched. Nevertheless, from our experience, the hypergraph optimizer has mainly produced better query plans than the original flow and is stable for this research.

3.3 Code generation (types, restrictions)

This section will describe how we compile MySQL `Item` trees down to LLVM IR. As mentioned in previous Section 3.2, this logic for code generation happens inside the `Item_compiled` entity, as it already contains a reference to the original `Item`.

The code generation job involves transforming an `Item` tree into a function in LLVM IR, containing generated code equivalent to the logic in the tree. A call to this function should return exactly the same output as a call to `Item::val_int()` in the original `Item`. Since the output of `Item::val_int()` is `long long`, a 64 bit signed integral type, our generated LLVM IR function needs to output a 64-bit integer as well. However, in MySQL, the default integer data type, `INT`, is 32 bits wide, meaning stored integer types might have different sizes from what we want to output. Since MySQL also officially supports 64-bit integers, such as `BIGINT` and others, we have chosen to restrict our implementation only to utilize this length. It makes our implementation simpler and should not have any significant impact on performance. This choice means all stored data types less than 64-bit must be type-coerced into 64-bit. Since we have restricted ourselves only to support `INT` and `STRING` like data types, this is a straightforward implementation.



(a) `Item` for adding two numbers

```
define i64 @my_add() {
    %1 = add i64 33, 36
    ret i64 %1
}
```

(b) The generated LLVM IR

Figure 3.3: A simple illustration of an `Item` tree and its generated LLVM IR

Another critical aspect of the implementation is how we support the different data types and operators mentioned in Section 3.2. Before we decide if we should compile an `Item` tree or not, we need to determine if the `Item` can be compiled based on the restricted data types and operators we have chosen. Determining this is done by having every `Item` have their implementation of a `can_compile()` method, which determines if the `Item` and the entire tree below it can be JIT-compiled or not. For instance, `Item_and`'s implementation of this method checks if all its children can compile by calling upon the children's `can_compile()` method and conjuncts them through a logical `AND` and ending up traversing the entire tree. If it turns out that the entire tree can be compiled, we proceed by creating a new instance of `Item_compiled` for that given `Item`.

However, if the entire tree is not compilable, we benefit from our *rolling update* system. We traverse the tree and check if any subtrees are compilable, using the same call to can_compile(). If any subtrees are compilable, we replace that subtree with a new instance of Item_compiled and immediately initiate code generation and compilation. The output is an Item tree consisting only of Item_compiled or some leaf nodes being of type Item_compiled. A call to the root's Item::val_int() method will, in a successful implementation, output the same result as before, using JIT-compiled code.

3.3.1 Generating LLVM IR

Our implementation revolve around having every Item in the tree generate their own LLVM IR, based on the type of the Item. For instance, Item_int generates LLVM IR for representing a 64 bit integer value, but Item_and generates LLVM IR for doing a logical AND operation on two or more child expressions. To generate LLVM IR for the entire tree, our implementation works similar to having each Item implement their own virtual codegen() method, where Items containing children also calls their corresponding codegen() method.

However, our implementation does not utilize this kind of virtual method overloading on Items due to technical difficulties. Instead we ended up utilizing a recursive codegen_item(Item *item, ...) function that detects the type of the input Item in runtime to determine which IR to generate. Here, we utilize dynamic_cast in C++ to determine the type, which is less good for performance than virtual method overloading. However, we have not experienced any worthy performance change with such an approach and have decided that the outcome is negligible.

On the other hand, our entire implementation to generate LLVM IR consists of passing around an instance of JITBuilderContext into the recursive function, containing an LLVM *module*, an in-memory representation of LLVM IR. This *module* will be mutated by using one of LLVM's built-in libraries, the *IRBuilder*, which provides a simple API to generate different logical operations in LLVM IR. Everything needed to generate the code for a given Item is contained within the JITBuilderContext. With this, we can describe the flow for generating LLVM IR for an Item in the following way:

1. The optimizer wants to convert an AccessPath contaning an Item into a FilterIterator.

2. We first check if the given Item is compilable by calling can_compile(). If the Item is compilable, we create a new instance of Item_compiled and makes it the referenced Item in the FilterIterator. If the item is not compilable, we traverse the tree and check if any subtree is compilable and replaces the subtree with a new Item_compiled if any exists.

3. Next, for every created instance of Item_compiled, we call upon Item_compiled ::codegen_item() method. This call initiates code generation for that given Item and starts by creating a new LLVM IR function definition, returning an i64. Since we might have multiple instances of Item_compiled, these functions must have a unique name, as the function name is used in the symbol table when linked into the running process and, for that reason, has to be unique.

Therefore, every instance of `Item_compiled` gets its own randomly generated name of 12 characters, which is used in the generated function definition.

4. Further, we want to generate the body of the function. This is done by calling upon our recursive function, `codegen_item(Item *item, JITBuilderContext *context)`, which passes the reference `Item` and the `JITBuilderContext` as arguments. Every `Item` generates its own LLVM IR and also invokes `codegen_item(...)` for its own children. This generates the IR for the entire body, and is finalized by the `Item_compiled` instance. At this step, all generation of LLVM IR is completed and stored in-memory in the LLVM *module* inside of the `JITBuilderContext` instance.

Generating IR for comparison operators

The generation of LLVM IR for the different comparison operators is quite similar and only differentiates between the operation for comparison. The operators involved here are `>=?`, `<=?`, `=`, `<>`, `LIKE`, and `BETWEEN`. The code generation consists of generating the IR for two or more child `Items`, putting the resulting values into their own registers, and performing a specific operation on those registers. Most of these operators are binary operators and are, therefore, quite simple. If we restrict ourselves to integer types, our implementation can be described with the following pseudocode:

```
...
auto first_child_value = jit::codegen_item(first_child,
   context);
auto second_child_value = jit::codegen_item(second_child,
   context);

auto cmp = context->builder->my_cmp_func(first_child_value,
    second_child_value);
...
```

(a) Pseudocode for generating the LLVM IR

```
define i64 @my_func() {
  ...
  %first_child = i64 33
  %second_child = i64 36
  %3 = icmp my_cmp_func i64 %first_child, %second_child
  ...
}
```

(b) Possible generated pseudo LLVM IR

Figure 3.4: Illustrating code generating for comparison operators.

In figure 3.4 we generate LLVM IR for an arbitrary comparison `Item` containing two instances `Item_int`. In simple terms, we assign the value of each `Item_int` child into its own registers, and then uses the `icmp <cond>` syntax to define a comparison instruction on these two values. The illustrated LLVM IR above is not 100% correct semantically, but illustrates the point that the value of each child `Item` will get allocated to its own register. This works for all the supported comparison operators involving integers, excepted for `BETWEEN`. The `BETWEEN` condition in MySQL checks if an operand is within the inclusive range of two other operands. This means that multiple binary instructions are necessary for proper evaluation, and we ended up conducting the following IR for this condition, combining `>=`, `<=` and `AND`:

```
define i64 @my_func() {
  ...
  %1 = icmp sge i64 %operand1, %operand2
  %2 = icmp sle i64 %operand1, %operand3
  %3 = and i1 %1, %2
  ...
}
```

Figure 3.5: LLVM IR for the BETWEEN condition

Generating IR for logical operators

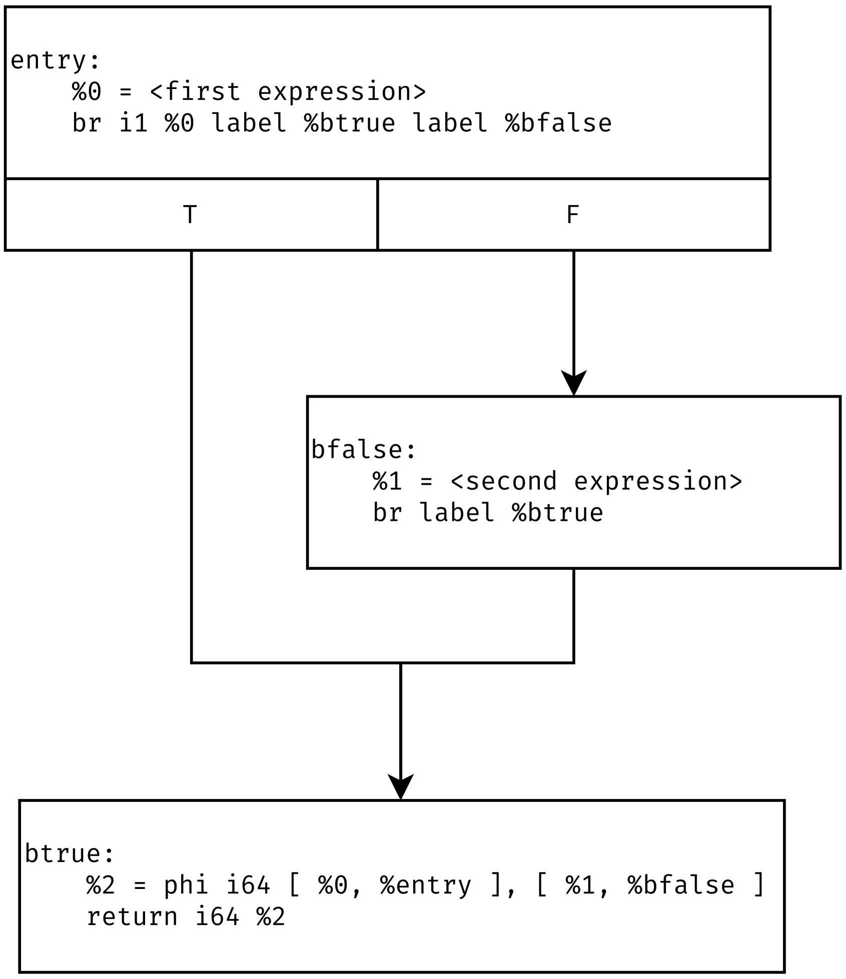
For the logical operators, such as `AND`, `OR`, and `NOT`, the approach for generating LLVM IR is not as straightforward compared to the comparison operators. A quick look at the logic of `Item_cond_and`, the `Item` representing the `AND` operation, there are at least one or more `Item` expressions that needs to be evaluated and conducted by the `AND` operation. The logic becomes increasingly complex when we start to introduce two or more expressions to evaluate. The reason for this is *short circuit evaluation*. We want to avoid unnecessary computation when the final value of the expression can be determined early. For instance, a long chain of `AND` operations can be stopped at the first argument evaluating to `false`, as computing the rest of the chain is at that point redundant.

In other words, this logic introduces branching, as the code needs to determine if it can skip the rest of the circuit after evaluating a subexpression. This concept of minimal evaluation is mostly relevant for the `AND` and `OR` operators and is quite an important step in reducing the number of instructions needed to execute. When looking at the `val_int()` method for `Item_cond_and` the following pseudocode comes to light:

```
longlong Item_cond_and::val_int() {
  for(Item *item : argument_list) {
    if(!item->val_bool()) {
       return 0;
    }
  }
  return 1;
}
```

Listing~3.1: Pseudocode of the short circuit evaluation implementation in `Item_cond_and`.

This introduces the concept of *control flow* and *branching* into our generated LLVM IR code. Control flow and branching in LLVM IR is done similarly to branching in an assembly language, utilizing jumps between *basic blocks*. In our implementation, we want to mirror the pseudocode in the listing 3.4 with LLVM IR. With only two `Item` expressions, both an `AND` or `OR` operation can be constructed with three basic blocks, one for calculating the first argument, one for evaluating the second argument, and a final block for conducting the final evaluation.



```
entry:
    %0 = <first expression>
    br i1 %0 label %btrue label %bfalse
```

```
bfalse:
    %1 = <second expression>
    br label %btrue
```

```
btrue:
    %2 = phi i64 [ %0, %entry ], [ %1, %bfalse ]
    return i64 %2
```

Figure 3.6: Control flow graph of `Item_cond_and` with two child expressions.

Figure 3.6 shows a control-flow graph (CFG) of a possible LLVM IR representation of the `AND` operator on two subexpressions, using three basic blocks. The first two blocks evaluate the value of the subexpressions, and the last block needs to figure out the final value based on the predecessor of the current block. Figuring out the final value is done using the LLVM `phi` instruction, which represents the ϕ statement in single static assignment (SSA) form, and is what LLVM IR is based on. This instruction makes it possible to choose a value based on the path to the current block. Regarding native machine instructions, the `phi` instruction ensures the predecessor blocks assign their values into the same register. The alternative would be to do an extra stack allocation and load the evaluated values onto the same location on the stack. This way, we keep most of our variables in registers and only do memory allocation onto the stack if no registers are available. This control-flow is what we use in our implementation for the `AND` and `OR` operations. However, for more than two arguments, the amount of basic blocks grows, correlated with the number of operands within the operation. For three arguments to evaluate, such as `EXPR1 AND EXPR2 AND EXPR2`, we would need four basic blocks, one for each argument and one for doing the `phi` instruction. In other words, the number of basic blocks n is equal to $n = m + 1$ for $m >= 2$ where m is the number of operands. With a growing number of branches, the higher the chance for branch misprediction is the highest cost here but is also negligible compared to evaluating all the expressions regardless of their outcome. The code below in figure 3.2 shows the LLVM IR we generate for an `Item_cond_and` with three arguments:

```
define i64 @zZyZPaGzk5EN() {
    entry:
        %1 = <first expression>
        br i64 %1, label %end_or, label %else

    else:                                   ; preds = %entry
        %2 = <second expression>
        br i64 %2, label %end_or, label %else2

    else2:                                  ; preds = %else
        %3 = <third expression>
        br label %end_or

    end_or:                                 ; preds = %else2, %else,
       %entry
        %itmp = phi i64 [ 1, %entry ], [ 1, %else ], [ %3, %else2 ]
        ret i64 %itmp


}
```

Listing~3.2: The LLVM IR for `Item_cond_and` with three arguments

Generating IR for `Item_field`

In MySQL, `WHERE` clause expressions containing references to columns in tables are in an `Item` tree represented by the `Item_field` class. The role of this class is to know the memory location of a specific column in a specific row and offer `val_TYPE` methods to extract the value of that location. Internally, this is done by maintaining a reference to the current row and a column offset. When the row changes during execution, the reference pointer automatically updates by pointing to the new row. In other words, `Item_field` maintains a pointer that always points to the data we want to extract.

Using this pointer in our implementation, we do not have to duplicate the logic of maintaining the correct reference ourselves. Therefore, when generating the LLVM IR for extracting a value from a given column, we load the number of bytes representing the size of the type we are working with from the internal pointer in `Item_field`. However, some types are stored in a particular way and do not necessarily represent a primitive data structure, such as integers, floats, and chars. For example, a `TEXT` string has a pointer to the string, in addition to its length, stored in the row. Dealing with such data types would increase the complexity of the generated code, as it would need different logic to read the different types.

However, since we are mainly restricting ourselves to a few data types in our implementation, we have only extracted data from the column pointer for integers.

Character-based data types also involve some complexity added to loading from storage and will be explained in more detail in Section 3.3.1.

With this said, for generating the LLVM IR for loading an integer value from a column, we use a pointer to a pointer that points to the current row, `p1`, and loads the value of this pointer from memory to obtain the pointer that points at the current row, `p2`. Then, by adding the column offset to `p2`, we know the exact location of the value we want to extract and then do an additional load operation to get the wanted value. This way, when the pointer of the current row, `p2`, is updated, our LLVM IR does not need to change and remains the same, as `p1` points to the updated pointer of `p2`. In other words, our generated IR consists of the following steps:

1. Do a load operation on `p1` to get the pointer to the current row `p2`. `p1` is found within `Item_field`.
2. Extract the column offset from `Item_field` and add it to `p2`. We now have a pointer pointing to the value we want to extract, which we can call `p3`.
3. Lastly, we do another load operation on `p3` in the form of a 32-bit integer, since MySQL `INT` is 32-bit, and store the value in a register. Since we have restricted our implementation to only work with 64-bit integers, we need to cast the loaded value to 64-bit.

```
define i64 @EItfcnVT0ebi() {
        ...
        %p2 = load i64*, i64 <<P1>>, align 8
        %p3 = add i64* %p2, i64 <<offset>>
        %column_value = load i32, i64* %p3, align 4
        %final_value = zext i32 %column_value to i64
        ...
}
```

Listing~3.3: LLVM IR for extract an 32 bit integer from a column in a row

Generating IR for strings

String data types are a typical type that is heavily used in most business-oriented database schemas and is considered essential in any database system. If MySQL decides to officially support the JIT compilation of expressions, an important aspect would be to figure out an excellent way to support these string data types. For this reason, we find it essential to support it in our implementation when compiling expressions.

MySQL has many vital features around these types that need to be supported, which carry much complexity on its shoulders. The most relevant features are its support for *character sets* and *collations*, which define how the different characters should be encoded, in addition to defining a set of rules for how the characters should be compared and sorted in relation to each other.

When comparing string types, for instance, with the `=` and `>=` operators, a standard byte-to-byte comparison is not the correct approach for doing this evaluation.

For instance, the default character set in MySQL is `utf8mb`, which represents the UTF-8 variable-width character encoding and stores a character within one to four bytes. MySQL already has support for a wide range of character sets and collations, which in code is a significant complexity that needs to be well maintained. Suppose we mirror this logic when generating LLVM IR. In that case, it will produce a large set of duplicated logic that needs to be maintained in parallel with the existing logic, decreasing the code quality of the codebase.

For this reason, we believe it makes sense to use the existing code in MySQL to handle the comparison situations concerning strings, as we think it would be best for MySQL in terms of maintainability and consistency. Such an approach integrates existing C++ code into the generated LLVM IR through function pointers, calling on functions in MySQL that deal with string comparisons. However, this approach comes with a set of disadvantages:

1. Calling existing code would introduce the extra overhead of a function call, which is what we want to eliminate in the first place.
2. The functions we call might include extra, unnecessary checks for types and collation rules which we already know during runtime. Generating the LLVM IR from scratch eliminates this extra work.
3. Running optimization passes on LLVM IR with an external function call will not be able to do the same level of optimization as if the code was integrated into the IR, meaning we might lose some additional optimization and performance possibilities.

In other words, this approach is not necessarily the optimal choice for performance. However, there is a trade-off between performance and maintainability in this case, where some minor performance gains come with a hefty cost for maintainability. We believe that the performance gains are not significant enough to outweigh the damages it causes to the codebase, as it would be more difficult and time-consuming to change and improve a larger and more complex MySQL. With this said, we believe that the best choice for MySQL would be to integrate less straightforward evaluation logic into the generated LLVM IR in the form of function pointers to existing C++ code and instead make a slight compromise in value for better maintainability.

We want to support comparing strings with the operators of `=`, `>=?`, `<=?`, and `LIKE`, as they are commonly used on string data types. To generate the LLVM IR for this approach, we need to modify the logic for generating IR on the comparison operators. When we generate code for a comparison `Item`, such as `Item_func_eq`, our implementation solves this by checking if the type of each child is of type `STRING_ITEM` , meaning it represents a string data type. If this is the case, we generate LLVM IR for doing a function call on an internal compare function inside of the comparison `Item`. This function call would trigger the normal flow for comparing the two `Items` within the existing system and return the result as a 32-bit integer that we further cast to 64-bit. This function call can be described with the following pseudocode:

```
...
// Handle strings
if(first_child->type() == STRING_ITEM || second_child->type
   () == STRING_ITEM) {
    ...
    auto  value = context->builder->CreateCall(<FUNC_TYPE>,
        <FUNC_PTR>, [<ARGUMENTS>]);
    return context->builder->CreateCast(<i64>, value);
    ...
}

// Normal procedure
auto first_child_value = jit::codegen_item(first_child,
   context);
auto second_child_value = jit::codegen_item(second_child,
   context);

auto cmp = context->builder->my_cmp_func(first_child_value,
    second_child_value);
...
```

(a) Pseudocode for generating the LLVM IR for comparison operators, including support for string data types.

```
define i64 @naXPLPr3A7S0() {
    entry:
      ...
      %0 = call addrspace(64) i64 <FUNCTION_ADDRESS>(i64 <
         ARGUMENTS>)
      ...
}
```

(b) Generated IR for doing a function call to existing C++ functions.

Figure 3.7: Illustrating code generating for comparing string data types.

3.4 Compiling (where, how?)

This section will elaborate on the compiling part of our implementation. At this point, LLVM IR is generated, and everything is stored within the `Item_compiled` instance, where the IR is stored as a *module* in the `JITBuilderContext`. Compiling the IR to native machine code starts immediately after the code generation process and involves the thread-global `JITExecutionContext` instance. The `JITExecutionContext` deals with compiling, optimizing, linking, and executing our generated LLVM IR. It is built upon the LLVM ORCv2 JIT API [37], a modular API for building JIT compilers, and consists of the following LLVM entities:

1. **ExecutionSession** - The context for executing the just-in-time compiled code. It owns the memory associated with the compiler IR and is where the final function pointer to our compiled code points to.

2. **JITDylib** - An asynchronous symbol table used in synchronization with the `ExecutionSession`.

3. **RTDyldObjectLinkingLayer** - A linking layer that is responsible for linking produced program objects by a compiler into our JIT `ExecutionSession`.
4. **ConcurrentIRCompiler** - A thread-safe LLVM IR compiler that compiles IR to native machine code object, and adds the final program representations to the `JITDylib`, using the `RTDyldObjectLinkingLayer`.

Using these top-level building blocks provided by the LLVM libraries, we can conduct JIT compilation of our generated IR without any complications. Providing our IR *module* to the `ConcurrentIRCompiler` allows the entities to handle the entire process of compiling and linking. The final output is made available by `JITDylib`'s symbol table, where the code itself is located within the memory of the `ExecutionSession`.

We can execute the final code by doing a lookup in the symbol table, which provides an address representing the function pointer to our generated code and doing a normal function call on that address. Our implementation of this process is very standard, contains no custom modifications, and is very similar to the set up in the official guidelines provided by the LLVM maintainers [32]. Our `JITExecutionContext` is just an abstraction over these LLVM entities, and is used by the `Item_compiled` instance to extract the function pointer address and use it during calls to the `Item ::val_int()` method. Our abstraction makes the JIT implementation fairly simple, which is why we believe LLVM is a good choice for MySQL.

With this said, every compiled `Item` tree will, in the end, be represented in-memory simultaneously until the query execution has finished and the `JITExecutionContext` is freed. To avoid having to free up added memory by the compiled code in the `ExecutionSession` for every query, we have a `JITExecutionContext` for every query session freed once the query completes. This choice was made due to simplicity. We have not considered this choice's impact on memory and performance compared to maintaining a global `JITExeuctionContext` dealing with multiple query session simultaneously.

3.4.1 Optimizations

Another essential step in compiling our code is optimization. The generated code constructed by the code generation step is not necessarily the most optimal way to do what it is set to do. For instance, some instructions are perhaps redundant, like some other part of the code makes those instructions unnecessary. Some other instructions may only contain constant values, which can be pre-executed once and reduced into a single constant. In other words, reducing the generated LLVM IR program to fewer and faster instructions might be possible, producing a more efficient program.

For this reason, we apply optimizations on our LLVM IR right before the compiling step, utilizing the optimization libraries within LLVM. However, these optimizations are time-consuming and add extra overhead to the compilation step. For large IR programs, this optimization step might consume more time than executing the query without JIT compiling anything at all. Therefore, there will be cases where we do not want to apply optimizations to the IR, and, as a consequence, we have chosen to make applying optimizations an optional step.

The optimization step consists of mutating the IR inside an LLVM *module* using the built-in LLVM optimization libraries for IR. These optimizations are implemented through *passes* that traverse the IR, collect information, and perform transformations on the program when possible [29]. These libraries also provide flexibility in which passes should be performed, giving us full control over the optimization step. Choosing optimizations means it is possible to do different levels of optimizations based on the predicted execution time of the query. The fewer optimization passes we apply to our IR, the less the overhead of the total compile time becomes.

We want to run optimizations as long as the total query time improves, which might not always be the case when running a large number of optimizations. Therefore, it makes sense to introduce different levels of optimizations, for instance *semi-optimization*, running some optimizations, and *full-optimization*, running all relevant optimization passes. However, we figured that introducing these levels of optimization is out of the scope of this research due to time restrictions, and we have chosen to restrict ourselves to two modes of IR, *unoptimized* and *optimized*, where the latter applies all relevant optimization passes. With this said, we ended up choosing these four main optimization passes:

- **Instruction Combining Pass** - Pass for combining redundant instructions to a fewer set of instructions, in additional to doing algebraic simplifications. A typical example for such a combination could be:

  ```
  // Before:
  %P = add i32 %Q, 2
  %M = add i32 %P, 4

  // After combining:
  %M = add i32 %Q, 6
  ```

- **Reassociate Pass** - A pass for doing reassociation on commutative expressions, which facilitate doing constant propagation. A simple example could be the following algebraic expression: $(33 + x) + 36 \rightarrow x + (33 + 36)$, where $(33 + 36)$ can now be constant folded into 69.

- **Global Value Numbering** - A pass for removing entirely and partially redundant instructions.

- **CFG Simplification and Aggressive Dead Code Elimination** - Performs dead code elimination and simplifies the CFG by merging basic blocks where possible.

In our implementation, these optimization passes are run inside of the `Item_compiled` instance before we start the compilation process in the `JITExecutionContext`, and are implemented using the `FunctionPassManager` entity provided by the LLVM libraries for doing optimization passes on functions. Since the generated IR we produce only consists of a single function, we found using only the `FunctionPassManager` relevant for our use case.

To give an example of how these optimizations affect our query-based LLVM IR, consider the following `WHERE` clause: `COLUMN1 = 17000 OR COLUMN1 = 18000`. This query will generate some load instructions to load the row value of `COLUMN1` and generate some branching logic for dealing with the *short circuit evaluation* regarding the `OR` operation:

```
define i64 @qJqmXONZI9mR() {
entry:
    %column1_row_ptr = load i64*, i64 140265580035984, align 8
    %column1_column_ptr = add i64* %column1_row_ptr, i64 16
    %column1_value = load i32, i64* %column1_column_ptr, align
       4
    %0 = zext i32 %column1_value to i64
    %2 = icmp eq i64 %0, 17000
    %3 = zext i1 %2 to i64
    br i64 %3, label %end_or, label %else

else:                                             ; preds =
   %entry
    %column1_row_ptr1 = load i64*, i64 140265580035984, align 8
    %column1_column_ptr1 = add i64* %column1_row_ptr1, i64 16
    %column1_value1 = load i32, i64* %column1_column_ptr1,
       align 4
    %4 = zext i32 %column1_value1 to i64
    %5 = icmp eq i64 %4, 18000
    %6 = zext i1 %5 to i64
    br label %end_or

end_or:                                           ; preds =
   %else, %entry
    %itmp = phi i64 [ 1, %entry ], [ %6, %else ]
    ret i64 %itmp
}
```

Listing~3.4: The generated IR for `COLUMN1=17000 OR COLUMN1=18000`

The IR illustrated above (3.4) represents the code generated IR for the expression `COLUMN1=17000 AND COLUMN2=18000`. Here, each subexpression, `COLUMN1=<INT>`, gets its basic block, where they both include logic for loading the value of `COLUMN1`. Meaning we unnecessarily do the same set of instructions for loading `COLUMN1` twice, which is due to having every `Item` generate their IR independently of each other. Illustrating that the IR that gets generated might be far from optimal. However, by applying the set of optimization passes mentioned above, we get the following result:

```
define i64 @mm9Lweda00o8() {
entry:
    %column1_row_ptr = load i64*, i64 140265580035984, align 16
    %column1_column_ptr = add i64* %column1_row_ptr, i64 16
    %column1_value = load i32, i64* %column1_column_ptr, align
       4
    %cmp = icmp eq i32 %column1_value, 17000
    %0 = zext i1 %cmp to i64
    %cmp2 = icmp eq i32 %column_value, 18000
    %narrow = select i64 %0, i1 true, i1 %cmp2
    %itmp = zext i1 %narrow to i64
    ret i64 %itmp
}
```

Listing~3.5: The optimized IR for `COLUMN1=17000 OR COLUMN1=18000`

The code listing 3.5 shows an optimized version of the IR in the code listing 3.4. Here, we see a far more optimal evaluation of the example clause, where the number of instructions has been reduced from 16 to 9, in addition to having the branches removed. The optimization passes detected the unnecessity of loading the same variable twice from memory but also decided to remove the branching logic for *short circuit evaluation.* It was decided that the extra branching was more costly than simply evaluating the second expression, which is now optimized down to one single `icmp eq` instruction. Doing this kind of optimization also makes sense for these small queries, as branch mispredictions are much more costly than using a few CPU cycles to evaluate one instruction.

However, this is not the case for slightly larger expressions, such as `COLUMN1 =17000 OR COLUMN1=18000 OR COLUMN1=19000`. From what we have seen, the optimization passes do not remove any branches for these larger queries and decide that evaluating the later subexpressions are more costly than the cost of branching. Still, it detected duplicated loads of `COLUMN1` and reduced a significant number of redundant instructions by performing a single load. Since these example queries contain subexpressions that are evaluated on the same column, these optimizations are, therefore, very rewarding. However, for more extensive and complex expressions containing different columns, the same kind of optimization is not achievable at the same level. Performing optimizations might not give any noticeable difference, depending on the query expression.

Altogether, from what we have experienced, the most rewarding optimization passes have been on small expressions and on expressions that contain multiple references to the same column. These pieces of information can be further used to evaluate if optimizations are worth doing before the `Item` trees are compiled.

Chapter 4

Evaluation MySQL JIT implementation

In this chapter, we will perform a thorough evaluation of our JIT implementation in MySQL and use it to attempt to resolve several aspects of our research questions. The focal point of our research is the impact JIT compiling expressions can have on performance in the MySQL ecosystem, which we believe our implementation can give an excellent indication of. This chapter aims to give a good understanding of the performance of our JIT implementation and further give an in-depth evaluation of the JIT system per the research questions. To be able to answer the main research question, RQ1, we also need to have a good understanding of RQ2, RQ3 and RQ4, which can be described as follows:

- **RQ2: What is the expected performance impact of jitting expressions in MySQL?** This question focuses on the scale our implementation can provide in terms of performance and to what extent it can reduce the overhead within MySQL for evaluating expressions. Is such an implementation beneficial for realistic business-oriented queries, and for what data types and expression sizes is it promising performance-wise to compile expressions? Altogether, the question emphasizes the actual speedup of the implementation in different scenarios regarding different queries and workloads.

- **RQ3: When is it considered beneficial to JIT compile an expression?** As stated in several other existing JIT systems, such as Postgres[50], Impala [49], and Hyper [23], JIT is not always beneficial due to the extra time-overhead introduced by compiling. Systems like Hyper [23] makes up their mind about compiling the query during execution, utilizing the extra information gathered while executing. However, since MySQL currently allocates only a single thread for each query session, doing such evaluation in the background is not realizable. Therefore, making this decision ahead of execution is what we believe is the most feasible approach for MySQL, which is the same approach as Postgres [50] and Impala [49] in this regard. The different factors relevant to making this decision can, for instance, be the size of the workload and the size and the complexity of the expression. With this said, it is essential to figure out how MySQL can use these factors to decide when it is beneficial to compile. Furthermore, determining in what circumstances it is beneficial to perform additional optimizations before compiling is also an essential aspect of this problem, as optimizations add delay to the total execution time. The focal point is to figure out how MySQL can use the available factors in the query to make the choice that provides the best performance benefit for the given query, either by jitting or not.

- **RQ4 Is the choice of utilizing LLVM for JIT a promising approach for MySQL?** Does our approach for compiling expressions with LLVM seems like a good approach for MySQL? Is it able to provide the essence of performance and maintainability, in addition to supporting the different requirements that MySQL sets for dependencies?

To answer the above questions, we will describe a set of conducted experiments on our MySQL JIT implementation. These experiments consist of a collection of benchmark tests measuring performance metrics on MySQL with and without the JIT implementation. The results of these benchmarks will be used to answer the three questions above, which will further imply an answer to the main research question. The experiments will resolve the questions in the following way:

- **Performance**: The performance question will be answered through a set of benchmarks on different expressions, mostly different expression sizes and involved data types, which will be compared to the same MySQL benchmark without utilizing the new JIT implementation. Here we will see the performance changes introduced by JIT on these different kinds of expressions and indicate how well applying optimizations affects the result. Similar benchmarks will be run on realistic business queries provided by the TPC-H benchmark to get an idea of how the JIT implementation will affect real use cases.

- **When to JIT**: This question will be attempted to be answered through a set of benchmarks revolving around different dataset sizes, workloads, expression sizes, and expression complexity. These results will give an idea of which factors affect the performance of JIT and when it is beneficial to JIT compile expressions. Additionally, different queries will be tested to see how jitting expressions affect different kinds of queries, such as join-heavy queries and queries involving subqueries. These tests will also be conducted with/without optimizations to determine when running optimization might be applicable.

- **LLVM**: To answer the question about whether LLVM is a good choice for MySQL, we will do an empirical evaluation based on our implementation and reflect on what we believe is an approach for MySQL based on the results of the experiments. Regarding this, we will discuss what we see as valuable aspects for MySQL and whether utilizing LLVM is a scalable approach to support all the existing and possible future features regarding expressions in the MySQL ecosystem.

In its first section, this chapter will describe the experiments in more detail and thoroughly explain the setup and why certain choices were made. The description consists of the tools used, the hardware setup, the dataset, and the different test suites. Furthermore, the second section will describe the first experiment, which we will name *the expression experiment.* This section will provide the most important results of the experiment and discuss how these results apply to the performance questions above. The following section will outline the results of the third experiment, which we have named *the TPC-H experiment*. Here, we will discuss the performance impacts of running our JIT implementation on the TPC-H benchmark, which represents the effect of jitting expressions on business-oriented ad hoc queries. Lastly, we will summarize our findings and condense our observations from the experiments with the research questions in mind.

4.1 Experimental setup

The two experiments we conducted focus on the performance of the expression evaluation part of the execution. The time spent evaluating rows is considered the key metric in this research. Therefore we have chosen to carry out a set of different test suites testing the evaluation process in different ways, utilizing our JIT implementation. In this section, we will first explain the setup of software and hardware specifications. Next, we will explain *the expression experiment* and describe in detail how the experiment was carried out and measured. Furthermore, we will explain *the TPC-H experiment*, where we have done a similar TPC-H benchmark to what recent work has been done on Postgres [30], and Hyper [23] when measuring the performance of their JIT systems. The goal is to understand better the different details of the experiments and how they were conducted.

4.1.1 Hardware & software

During this thesis, we performed all of our experiments with MySQL Server version 8.0.28, the latest MySQL version when we started. This version was chosen because of the belief that it would provide the best performance, especially with the most up-to-date version of the *hypergraph optimizer*. This version of MySQL was used to implement our JIT system for expressions, where some modifications were applied to the existing code, and a new JIT library was added. Our JIT implementation uses LLVM version 13.0.1, the latest version of the project when the thesis began. Therefore, the entire MySQL project was locally compiled with most default configuration flags, except for the flag to enable the *hypergraph optimizer*.

All experiments described in this thesis are run on a Linux computer (Ubuntu, v21.04). This computer consists of a 4-core Intel Core i7-7700 CPU (3.6 GHz) processor and 32 GB of system memory. With this memory capacity, we can keep most of the test data in memory as most of the data never exceed 30 GB. However, the data MySQL keeps in memory is controlled by the size of the *buffer pool*, which is configurable and set to 128M by default. To ensure that most of our experiments are CPU bound and not highly affected by disk reads, we have chosen to set this *buffer pool* size to 10GB, as most of our datasets are 10GB or less. Moreover, one of our datasets is 30GB making the experiments using this dataset more disk-bound, as the entire dataset does not fit within the buffer pool. As a result, we tested both CPU-bound and disk-bound environments, depending on the size of the dataset, which produced results that indicate the performance impact of JIT in both scenarios.

4.1.2 Introduction to the expression experiment

The purpose of *the experiment benchmark* is to measure the performance impact our JIT implementation has on evaluating expressions. For instance, we want to be able to determine how much instruction overhead JIT compiling a single equality expression is capable of reducing and how this question applies to more complex expressions and overall performance. Therefore, this experiment runs a collection of simple table scan queries containing different kinds of expressions in size and data types. Since our JIT implementation only supports `INTEGER` and `STRING` based data types, we chose to introduce three types of expressions: *integer-only*, *string-only* and a *mix* of both integers and strings. With this, we can compare the performance outcome between generated code that includes function calls to existing C++ code and code isolated from the rest of the MySQL codebase. These expressions are generated based on a table with 16 columns evenly divided between integer-based columns and string-based columns. Since we want to test the performance of the jitted code, we want to ensure that we run the entire expression and avoid instances of short-circuit evaluation. Therefore, every expression is generated to always evaluate to false and only consists of the `=` operator, comparing the equality of a row-column and a constant value. The complexity of the expressions is also fairly low and only consists of a certain sized `OR` chain, where the size can be found within the collection of $\{1, 10, 25, 50, 100, 250, 500, 1000\}$. By running this experiment, we expected to achieve a greater understanding of the following concepts:

- An indication about if compiled expressions prove to be faster than the existing evaluation logic using `Item` trees.
- How large and complex does an expression need to be for JIT compiling to be beneficial for performance?
- How much time does it take to compile an expression, given its size and complexity?
- How much time does it take to optimize the LLVM IR generated for an `Item` tree, given its size and complexity?
- How many instructions are we removing by compiling the expression? Is the reduction at the level of something noteworthy?
- The impact of applying optimizations has on performance at expression and query level.

Measurements and setup

In this experiment, we chose to extract the following measurements:

- **Total query execution time**: the time it takes to execute an entire query, from the client's perspective.
- **Total time used on `::val_int` calls**: the sum of all time measurements to evaluate the expressions. To clarify, evaluating the expression for JIT compiled expressions is done by calling upon `Item_compiled`'s `val_int` method, which is the same behavior as for non-compiled `Item` trees.

- **Total codegen time**: The time it takes to generate the LLVM IR for a given `Item` tree.
- **Total compiled time**: The time it takes to compile the generated LLVM IR.
- **Total optimization time**: The time it takes to optimize the generated LLVM IR.
- **Total instructions used on `::val_int()` execution**: The number of hardware instructions executed on the CPU during the execution of the `::val_int()` methods.

To carry out these measurements, a MySQL Server with the JIT implementation and a MySQL client were run locally on the same machine. The total query execution time was extracted from the client side, timing millisecond precision. For the other measures, these require timing logic within the existing MySQL system. Fortunately, MySQL already provides a profiling tool to show in which parts of the execution MySQL spends time, called `EXPLAIN ANALYZE`. This tool supplies measurements for time and number of processed rows for varying parts of the execution plan, which is handed to the client after execution. We decided that extending this profiling tool to measure the metrics of our JIT implementation was a feasible approach for extracting the desired performance measurements. However, it is important to note that `EXPLAIN ANALYZE` includes extra logic to time different `Iterators` of the query plan, making the total execution time slower, and is therefore not a good approach for measuring the total execution time of the query. On that note, a large portion of this benchmark consisted of running `EXPLAIN ANALYZE` queries, and extracting the result from the query output. For example, to get measurements regarding `::val_int()` calls, we added timing logic within the `FilterIterator::Read()` method, between the call to `::val_int()`:

```
int FilterIterator::Read() {
    ...
    start = now();
    bool matched = item_condition->val_int();
    end = now();
    ...
}
```

(a) Pseudocode to illustrate how `::val_int()` calls are timed

```
-> Filter: Item_compiled[...](codegen=X compile=Y opt=Z)(<EXPR>)
   ... (time_spent_on_val_int_calls=W instruction_count=P) ...
       -> Table scan on <TABLE> ...
```

(b) Simplified output of `EXPLAIN ANALYZE`, containing time spent on `::val_int()` calls, `W`.

Figure 4.1: Showing how `::val_int()` calls are measured and outputted by the `EXPLAIN ANALYZE` profiling tool.

As shown in Figure 4.1, the time spent evaluating the expression is outputted by the `EXPLAIN ANALYZE` profiling tool, together with the other measurements relevant to the compilation process. For timing the total execution time, this internal timing logic was disabled to ensure all overhead introduced by timing was nonexistent.

All queries benchmarked in this experiment were run multiple times in different *execution mode* configurations: `nJIT`, `JIT`, and `JITOPT`. `nJIT` represents the *non-JIT* execution mode, which executes queries without JIT compilation of any expressions. `JIT` compiles all expressions in the query unless our implementation does not support the expression. `JITOPT` behaves similarly to `JIT` but also applies optimizations on the LLVM IR before compiling. These execution modes are set by applying a predefined group of runtime variables, which toggles different features regarding JIT. With this approach, we are easily able to collect comparable measurements that can be used to analyze the different aspects of our JIT system.

4.1.3 Introduction to the TPC-H experiment

This experiment consisted of running the TPC-H benchmark on our MySQL JIT implementation. The TPC-H benchmark is a popular decision support benchmark, which specifies a dataset and a suite of business-oriented ad hoc queries for organizational decision-making activities [45]. According to the specification, the benchmark is chosen to be of industry-wide relevance, illustrating decision support with a set of high-degree complex queries that answer critical business questions. We chose this benchmark because it provides a collection of realistic and diverse expressions, which would be a great help in understanding the possible performance benefits of our JIT system in real-world scenarios. The specification provides a set of 22 queries and is scalable to different scale factors. However, in this experiment, we have restricted ourselves to 17 queries, as some of these queries introduced some technical difficulties.

This benchmark was conducted by generating the TPC-H dataset in different scale factors to see how our JIT system impacts performance on different workloads. The benchmark indicates whether increasing workload makes expression compilation more beneficial, measuring how large workload is as a factor. However, since we restricted our JIT system to a few select data types, the data model specified in the benchmark was modified to only utilize data types based on `INT` and `STRING`. We had to convert dates and decimal types into integers to achieve this. Converting the data types also means that the 22 queries specified in the TPC-H benchmark were modified by transforming the dates and decimals to their integer equivalents. Furthermore, these queries also consist of other features that our JIT system does not support, such as functions as `EXISTS` and `SUBSTRING`. For the four queries, this involves, the system will automatically find that their `Item` equivalent is not supported and will not compile them. For the final mutation that we performed, the TPC-H queries also introduce some `IN` expressions, which we have transformed into a chain of `OR` operators. This change was done because we wanted to support most expressions as much as possible.

We expected to gain the following information about our JIT system by conducting this experiment:

- The impact of JIT compiling expression has on performance for realistic business-oriented queries. It is intended to provide an idea of what performance expectations can be made for a typical jitted business-oriented query in our system. This benchmark will also provide information about how speeding up expression evaluation will impact the total query execution time.

- The benchmark will also provide information on the performance impact of JIT scales with different workloads. Since we are testing with different scale factors within the interval of `0.1GB` to `30GB`, we might be able to get an indication of the workloads at which the compilation time is large enough to make the compilation of expressions disadvantageous.
- The impact of running optimizations on realistic business-oriented queries. Since the expressions specified in the TPC-H benchmark come with some complexity, it is interesting to see to what extent the optimizations can reduce the number of instructions to execute.

Measurement & setup

When running this benchmark, we decided to extract the same measurements as stated in the list in Section 4.1.2 regarding the *the expression experiment*. These measurements consist of the time used on `::val_int()` calls, the compilation time, the codegen time, the optimization time, and the total query time. All these measurements are also extracted in the same way, using `EXPLAIN ANALYZE` when applicable and evaluating the total query time on the client-side. Every query in the benchmark is also run in the same set of execution modes mentioned in Section 4.1.2, benchmarking MySQL without JIT, with JIT, and with JIT, including optimizations.

4.2 Evaluating the expression experiment

The following section will present and discuss the different results of *expression experiment*. First, we will focus on the performance of the expression evaluation, where we present the results of each expression type and discuss our observations. Afterward, we will examine our compilation time observations, considering factors like complexity, expression size, and how applying optimizations is affected by these factors. Lastly, we will show the results of the instruction count measurements and consider whether applying JIT affects the existing instruction overhead of `Items` in MySQL in a positive manner.

4.2.1 The performance of expression evaluation

In this experiment, the resulting data show promising results in terms of performance. This experiment consists of various test suites, such as different expression sizes, execution modes, and involving data types. We will start by introducing the result of `INT` based expressions due to their simple form and complete isolation from existing MySQL code. Afterward, we will provide the results of the rest of the expression types, involving the `STRING` expressions and a mixture of both. In this experiment, we ran on a table with approximately 18 million rows, equivalent to 3GB in size.

The results of the `INT` based expressions

By conducting the experiment on expressions that consisted only of `INT` based data types, we got the following results:

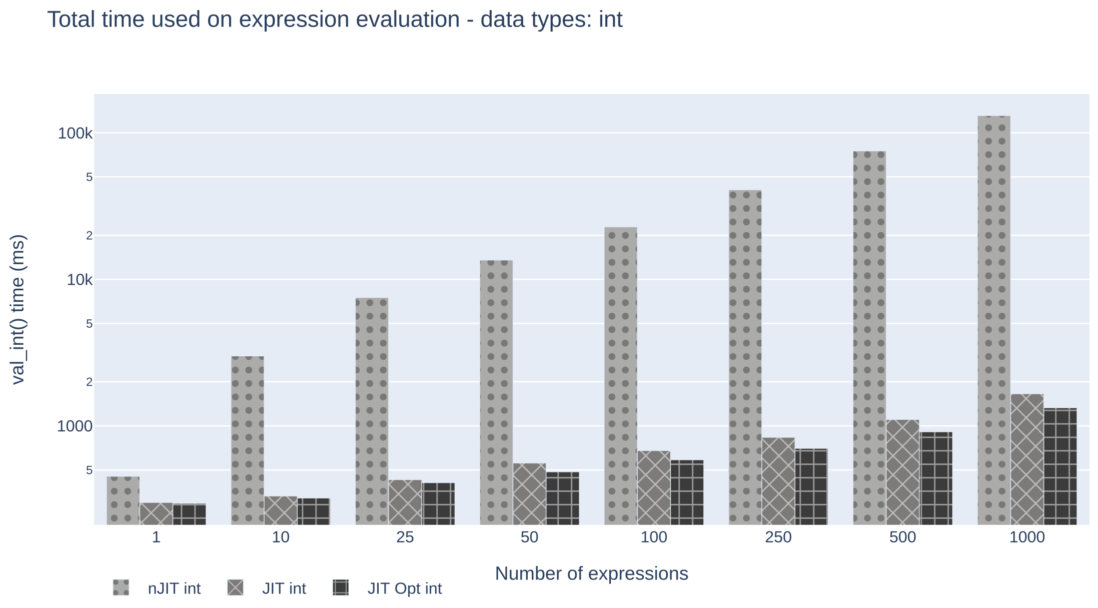


Figure 4.2: The time used on evaluation expressions on `INT` based exp ressions by size.

Expression Experiment - results in ms - Data type int											
	nJIT		**JIT**				**JITOPT**				
Size	total	val_int	total	val_int	codegen	compile	total	val_int	codegen	compile	opt
1	4194	452	4042	300	0.01	1.35	4045	296	0.01	1.33	0.24
10	7836	2998	5099	332	0.04	2.21	5098	321	0.04	2.01	0.46
25	13746	7508	6506	429	0.08	3.34	6554	409	0.08	2.92	0.73
50	19963	13541	6633	557	0.14	5.31	6547	485	0.14	4.32	1.21
100	30375	22805	7138	678	0.27	9.28	7060	586	0.27	7.02	2.17
250	49039	40832	7475	834	0.63	21.3	7336	702	0.63	15.03	4.94
500	83680	75244	7756	1105	1.41	43.17	7531	910	1.41	29.46	9.76
1000	139239	130878	8375	1656	3.69	94.4	8022	1328	3.67	61.93	19.12

Table 4.1: The results of all modes on `INT` based expressions.

Average `::val_int()` time per row (ms)					
Expr size	nJIT	JIT	JITOPT	JIT Speedup	JITOPT Speedup
1	0.0000251	0.0000166	0.0000164	50.88%	52.81%
10	0.0001666	0.0000184	0.0000178	802.12%	831.71%
25	0.0004172	0.0000238	0.0000227	1647.71%	1733.70%
50	0.0007524	0.0000309	0.0000269	2328.78%	2688.17%
100	0.0012671	0.0000376	0.0000325	3261.29%	3787.41%
250	0.0022689	0.0000463	0.0000390	4790.31%	5711.44%
500	0.0041810	0.0000614	0.0000506	6703.94%	8161.23%
1000	0.0072724	0.0000920	0.0000738	7802.00%	9749.56%

Table 4.2: Time used on expression evaluation per row for `INT` based expressions.

The results in both Figure 4.2 and Table 4.1 show a great performance benefit from JIT compiling expressions. Figure 4.2 shows the time used on `::val_int()` calls for all different expression sizes with a logarithmic y-axis, which measures the time of expression evaluation alone. This shows that evaluating expressions with existing MySQL `Item` logic is much slower than applying JIT. For a single `COLUMN = <INT>` expression, JIT seems to apply a 50% speed-up when evaluating the expression on all 18 million rows. For the largest expression of 1000 `COLUMN = <INT> OR ...` subexpressions, the speed-up of JIT seems to be around 7800%. This indicates that JIT can remove much overhead related to `Items` for these simple `INT` based expressions. We also observe that the performance benefit of JIT increases with the number of expressions to evaluate, evidently due to the accumulating overhead of the `Item` expressions. Looking at Table 4.2, this growth in speed-up provided by applying JIT seems to be growing in a logarithmic fashion for this simple `INT` based expression.

The results show that the expression evaluation process takes a more significant portion of the total query execution workload as the expression size grows. For example, an expression of size 1 without JIT consumes around 10% of the total execution time. When the expression size reaches the size of 25, the expression evaluation absorbs approximately 50% of the total execution time and reaches 90% with a size of 1000. This indicates that the expression size greatly impacts the total query time, which makes speeding up the expression evaluation process more critical to reduce the overall query execution. Therefore, by applying JIT compiling on these expressions, the total query execution time has been drastically reduced. For an expression of size 1000, JIT improves the performance of the query by around

17x. However, most realistic queries do not contain expressions of these sizes and empirically exist between 1 and 10 subexpressions. For these smaller sizes, our test results show an improvement in performance in the total query time between 4% and 53%, where expression evaluation has been accelerated between 50% and 800%, according to Figure 4.2. Since queries usually consist of joins and aggregations, expression evaluation typically takes a smaller portion of the execution time. These numbers are a much more realistic outcome of JIT compiling expressions.

Another interesting topic to present is the `JITOPT` execution mode, which applies optimizations to the LLVM IR before compiling. Figure 4.2 shows that this execution mode provides a non-existent performance benefit for the smallest expression of size 1 but becomes more and more beneficial as the expression size grows. For example, Table 4.2 shows a speed-up almost identical to the `JIT` execution mode, with a speed-up around 50%. However, for larger expression sizes, such as 10 and greater, applying optimizations seems to speed up expression evaluation to a greater extent, ranging from 4% to 25% additional improvement to the `JIT` execution mode. This is not surprising, as larger expressions add more room for optimizations. As mentioned in Section 3.4.1, the most common optimizations to use on the generated LLVM IR is the removal of duplicated memory loads to the same column, removing unnecessary casts to `i64`, and removal of branching. During this test suite, we observed that most of the optimizations made were removing duplicated loads to the same column in memory, which we find to be the best optimization of the alternatives. This is also considered sensible, as the table in this experiment consists of only 7 `INT`-based columns, making the generated expressions consist of the same column more than once. We consider this the best optimization because memory accesses are slow. By removing many of these unnecessary memory accesses, more of the data can be found within the CPU registers, making execution generally more efficient. Furthermore, Table 4.3 shows that we were able to reduce the size of the generated code from 10% to 38% for the four smallest expressions, showing that the application of optimization can greatly benefit the performance of the expression evaluation process.

# of LLVM IR lines by execution mode			
Expression size	`JIT`	`JITOPT`	reduction (%)
1	10	9	10%
10	95	68	28%
25	230	152	34%
50	455	278	38%

Table 4.3: Caption

Despite these excellent results, it is worth mentioning that the performance boost of applying optimizations was achieved mainly due to the generated expressions consisting of multiple instances of the same column. For example, for all columns in the expression of size 25, the same column appears within $2 - 6$ times, which is an unlikely scenario for most realistic human-written queries. However, this might not be so uncommon for large machine-generated queries from business intelligence tools, as these queries can become extremely large and far from optimal (e.g., 1MB SQL of text, according to the authors of Hyper [23]). Therefore, this implies that running optimizations can provide a significant performance benefit for queries using the same columns multiple times but might not be that applicable for queries that contain few duplicated columns.

The results of the `STRING` based expressions

For the expressions consisting of a chain of `STRING` based comparisons, the JIT-compiled versions utilize function calls to the existing MySQL C++ codebase, which strongly differs from the `INT` based expressions of this experiment. With this said, we got the following results from the `STRING` based benchmark:

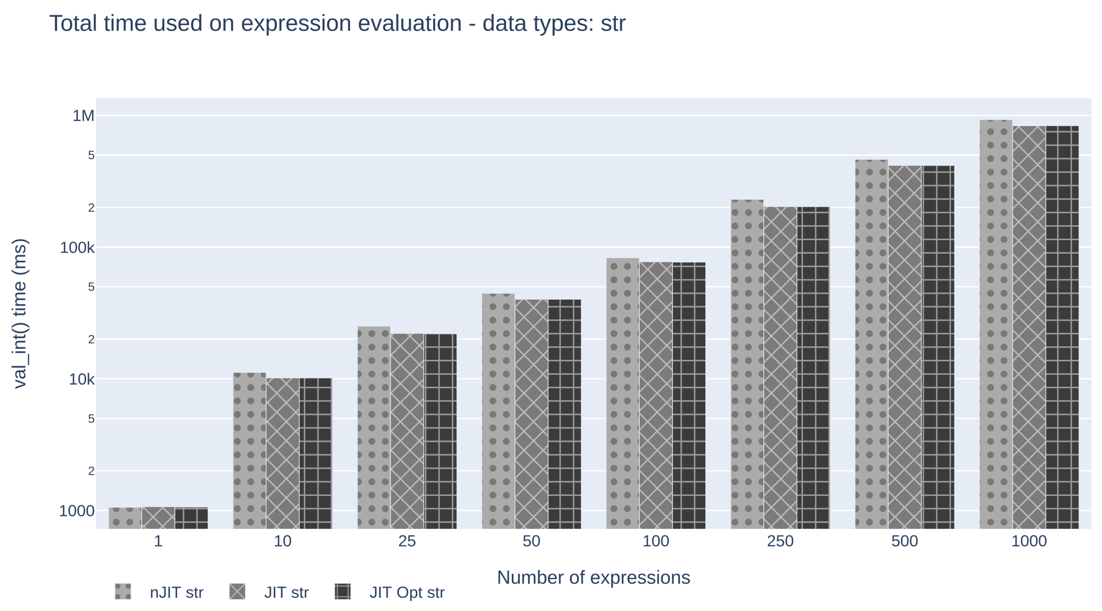


Figure 4.3: The time used on `::val_int()` calls for `STRING` based expressions

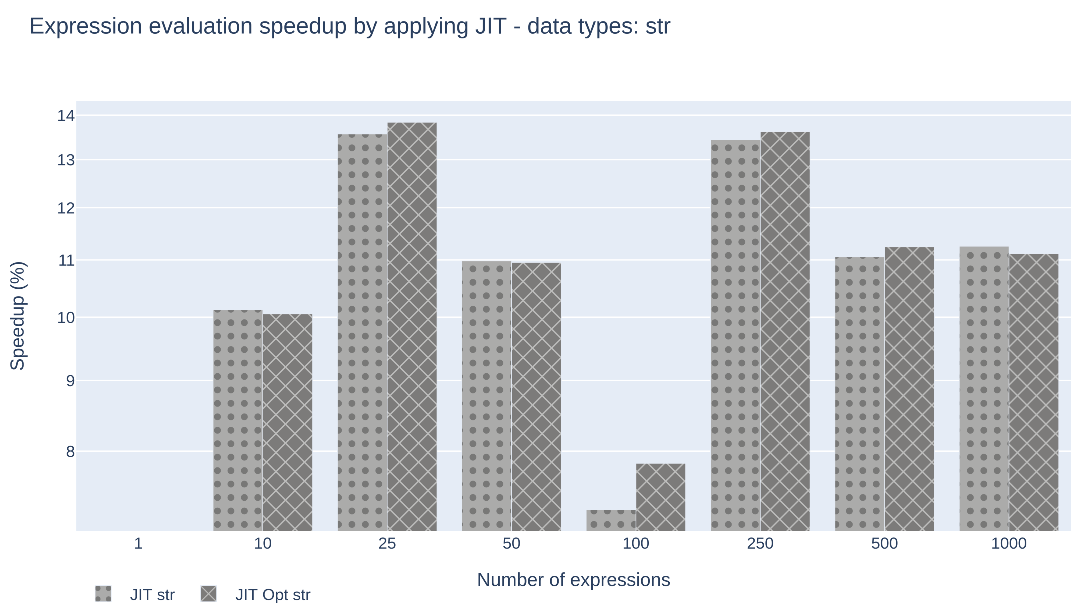


Figure 4.4: The speedup of applying JIT for `STRING` based expressions

Expression Experiment - results in ms - Data type str											
	nJIT		**JIT**				**JITOPT**				
Size	total	val_int	total	val_int	codegen	compile	total	val_int	codegen	compile	opt
1	5214	1056	5252	1067	0.01	1.31	5196	1061	0.01	1.26	0.19
10	16953	11172	15880	10144	0.03	2.3	15921	10151	0.02	2.27	0.28
25	31862	25111	28891	22111	0.04	3.57	28786	22059	0.04	3.69	0.38
50	51487	44566	46999	40156	0.07	5.68	46998	40166	0.07	5.62	0.6
100	90066	82932	84483	77321	0.13	9.86	84050	76903	0.13	9.82	1
250	238808	230462	211377	203148	0.3	22.63	210968	202845	0.31	22.37	2.19
500	472141	463549	426049	417398	0.77	44.34	425327	416697	0.76	44.32	4.32
1000	938135	929233	844213	835233	2.88	101	845274	836285	2.5	93.37	10.19

Table 4.4: The results of all modes on `STRING` based expressions.

Average `::val_int()` time per row (ms)					
Expr size	nJIT	JIT	JITOPT	JIT Speedup	JITOPT Speedup
1	0.0000586	0.0000593	0.0000589	-1.04	% -0.46%
10	0.0006207	0.0005637	0.0005640	10.12%	10.06%
25	0.0013953	0.001228	0.0012257	13.57%	13.84%
50	0.0024763	0.0022313	0.0022319	10.98%	10.95%
100	0.0046082	0.0042964	0.0042732	7.26%	7.84%
250	0.0128058	0.0112881	0.0112712	13.44%	13.61%
500	0.0257575	0.0231931	0.0231542	11.06%	11.24%
1000	0.0516338	0.0464106	0.0464690	11.25%	11.11%

Table 4.5: Time used on expression evaluation per row for `STRING` based expressions.

The results shown in Figure 4.3, Figure 4.4, Table 4.4 and Table 4.5 show that the performance impact of JIT differs greatly from the `INT` benchmark. For example, from Figure 4.3 we can observe that there is barely any difference between `nJIT` as the execution mode and JIT as the execution mode, especially compared to the INT-based expression benchmark. For a single STRING-based comparison expression, the time difference with or without JIT is considered negligible. However, for an expression of size 10, we get better results with a speed-up of around 10%. From there, an increase in the expression size does not affect the performance of expression evaluation in any way, keeping the speed increase at approximately 10%. These results are not surprising, as the code generation for the STRING based expressions mainly consists of function calls to existing C++ to deal with the string comparison. The LLVM IR consists of many function calls, similar to the behavior of the existing Item evaluation process. One of the main purposes of JIT compiling an expression is to reduce the number of virtual function calls, as a function call introduces additional instruction overhead and stack accesses. By making the generated LLVM IR call upon the existing C++ code, the extra overhead of function calls remains, making the point of compiling these types of expressions useless, as we execute the same instructions as the Item-tree equivalent. However, from this conclusion, one might further wonder why we still get a speed-up of approximately 10% for expressions with sizes larger than 10. This speed-up means that we can still reduce some overhead by compiling the expression. One possible example of this reduction is that we do not call upon the val_int() function of the string comparison Item, but instead directly call the internal comparison function, avoiding one extra function call. Furthermore, the ::val_int() call consists of additional logic to deal with NULL values and other edge cases, which we considered out of the scope of this thesis. This might explain why there is a speed-up with JIT as the execution mode, although the instructions to evaluate the expressions are nearly identical.

# of LLVM IR lines by execution mode			
Expression size	JIT	JITOPT	reduction (%)
1	5	5	0%
10	45	45	0%
25	105	105	0%
50	205	205	0%

Table 4.6: The number lines of the generated LLVM IR for the different modes for STRING-based expressions

When it comes to the effects of applying optimizations to these STRING-based expressions, the results show insignificant performance improvements. The performance of the expression evaluation process with the JITOPT execution mode is considered equal to the performance of the JIT execution mode. The JIT system could not significantly optimize the generated IR to any extent. Table 4.6 shows the number of lines of the final LLVM IR by the different execution modes, and, unlike the INT based results of Table 4.3, there is no change between the execution modes JIT and JITOPT. When manually comparing the generated LLVM IR for these two modes, the IR is completely identical as they only consist of function calls to existing C++ code. Meaning that for the data types of strings, there is nothing to optimize, and applying optimization is considered unnecessary. This further implies that, for all data types that our JIT system can support by calling existing C++ code, the application of optimizations would not be beneficial, in addition to not producing any significant performance improvements by JIT compiling the expression.

The results of the MIX based expressions

The MIX-based test suite is the benchmark of expressions with both data types, integers, and strings. The following results were obtained with these expressions:

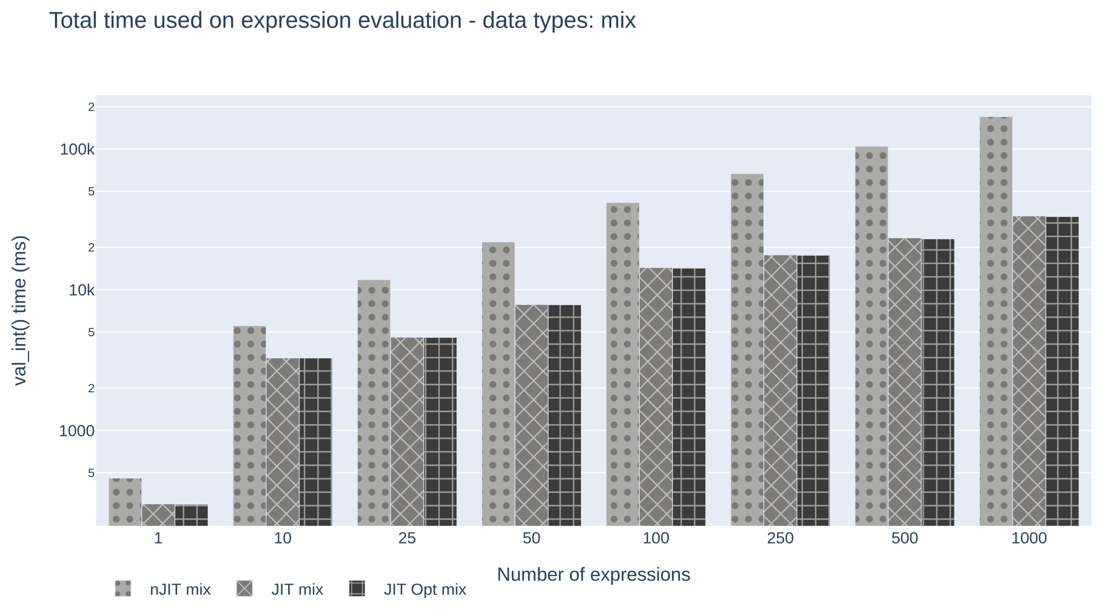


Figure 4.5: The time used on `::val_int()` calls for MIX-based expressions

Expression Experiment - results in ms - Data type mix											
	nJIT		JIT				JITOPT				
Size	total	val_int	total	val_int	codegen	compile	total	val_int	codegen	compile	opt
1	4206	458	4073	300	0.01	1.36	4045	298	0.01	1.32	0.24
10	11985	5552	9657	3272	0.04	2.36	9670	3269	0.04	2.28	0.45
25	18854	11780	11508	4591	0.08	3.57	11484	4578	0.08	3.37	0.75
50	31862	21866	16831	7845	0.13	5.64	16819	7816	0.14	5.26	1.23
100	51958	41662	23641	14358	0.25	9.91	23514	14234	0.25	9.1	2.22
250	78082	66797	27965	17680	0.57	21.88	27888	17563	0.56	19.17	5.19
500	116287	104655	33756	23365	1.35	43.26	33427	22973	1.34	38.04	10.44
1000	181427	169821	44010	33415	3.58	89.44	43659	33076	3.57	78.08	20.33

Table 4.7: The results of all modes on MIX based expressions.

Average `::val_int()` time per row (ms)					
Expr size	nJIT	JIT	JITOPT	JIT Speedup	JITOPT Speedup
1	0.0000254	0.0000166	0.0000165	52.48%	53.51%
10	0.0003085	0.0001818	0.0001816	69.69%	69.83%
25	0.000654	0.0002551	0.0002544	156.58%	157.31%
50	0.0012150	0.0004359	0.0004343	178.71%	179.75%
100	0.0023150	0.0007978	0.0007909	190.17%	192.69%
250	0.0037116	0.0009824	0.0009759	277.80%	280.31%
500	0.0058153	0.0012983	0.0012765	347.90%	355.55%
1000	0.009436	0.0018567	0.0018379	408.22%	413.42%

Table 4.8: Time used in evaluating expressions per row for `MIX` based expressions.

Unsurprisingly, the results for this test suite reflect a combination of the previous test suites of `INT` and `STRING`. Figure 4.5 shows an improvement in performance for expression evaluation, much better than the `STRING` test suite, but significantly worse than the `INT` test suite. This is obviously due to the fact that these kinds of expressions are affected by the drastic reduction in instruction overhead by the `INT`-based expressions but, in contrast, do not get any notable benefits from the `STRING`-based expressions. However, in most realistic query expressions, there is often a combination of multiple data types, such as integers and strings, making the expressions from this test suite represent more common use cases to a greater extent. This means that the results from this test suite are a more accurate representation of what one can expect regarding performance by JIT compiling such simple expressions. From Tables 4.7 and 4.8, we observe an increase in speed from approximately 50% to 400%, where most of the speed comes presumably from the generated code for the `INT`-based expressions. This implies that the combination of external C++ function calls and generated IR logic works well together and that JIT compiling expressions consisting of such different data types is still worth doing in regards to performance, despite containing other data types and expressions that provide no significant benefit by being compiled.

# of LLVM IR lines by execution mode			
Expression size	`JIT`	`JITOPT`	reduction (%)
10	80	69	14%
25	210	161	23%
50	410	316	23%

Table 4.9: The number lines of the generated LLVM IR for the different modes for `MIX`-based expressions

When applying optimizations, the optimized IR behaves unsurprisingly similar to the previous two test suites. Looking at the optimized IR, the same optimizations as mentioned in the `INT` test suite are performed, where unnecessary memory accesses for reused columns are removed, and some unneeded casts are removed 64-bit integers. For the `STRING`-based subexpressions in this test suite, there are no optimizations to be applied, similar to the `STRING` test suite. In other words, the only optimizations performed are related to the logic of the `INT`-based subexpressions. Looking at Table 4.9, the optimizations performed have been able to reduce the IR generated from 14% to 23% for these medium-sized expressions. However, looking at Table 4.7, the `JITOPT` execution mode has not been able to provide any notable performance improvement compared to the `JIT` execution mode and the performance benefit of optimizations can be considered negligible. Therefore, from these results, it is possible to argue that applying optimizations is not necessarily worth doing in this scenario, where around 50% of the subexpressions generate unoptimizable code. For that reason, this shows that when deciding whether to apply optimizations, it is important to consider the data types used in the expressions and whether they are optimizable. The results of this test suite show that having expressions with 50% unoptimizable subexpressions does not provide any significant improvement for small and medium-sized expressions and barely provides any improvement for larger unrealistic-sized expressions. Altogether, this implies that the size of the expressions and the ability to optimize the different data types play a big role in whether applying optimizations is considered beneficial and should be used in the evaluation process of whether to run the `JITOPT` execution mode under these circumstances.

4.2.2 Compilation times

Regarding compilation times, the experiment measured the different aspects of transforming an `Item` tree into executable machine code. The different aspects consist of code generation, optimization, and compilation. The following results on compilation times were achieved during the different test suites of the experiment:

JIT compilation times (ms)				
X =(total compile time)/(`nJIT val_int - JIT val_int`) rows				
Expr size	codegen	opt	compile	X
int				
1	0.0124	0	1.3482	N/A
10	0.0425	0	2.2118	N/A
25	0.0798	0	3.3375	8687
50	0.1389	0	5.3093	7551
100	0.2743	0	9.2797	7770
250	0.6316	0	21.2998	9867
500	1.4139	0	43.1746	10823
1000	3.6921	0	94.4005	13661
str				
1	0.0094	0	1.3072	N/A
10	0.0264	0	2.3042	N/A
25	0.04	0	3.5678	21641
50	0.0695	0	5.6844	23478
100	0.1261	0	9.8625	32035
250	0.3008	0	22.6304	15109
500	0.7655	0	44.3444	17590
1000	2.8771	0	101.9389	20067
mix				
1	0.0125	0	1.3585	N/A
10	0.0382	0	2.3575	N/A
25	0.0794	0	3.5732	9143
50	0.1328	0	5.6428	7413
100	0.2479	0	9.9065	6692
250	0.5712	0	21.8823	8227
500	1.3476	0	43.2593	9875
1000	3.5788	0	89.4411	12272

Table 4.10: Compilation times of the different test suites in *the expression experiment* for the JIT execution mode

JITOPT compilation times (ms)				
X =(total compile time)/(nJIT val_int - JIT val_int) rows				
Expr size	codegen	opt	compile	X
int				
1	0.0125	0.2393	1.327	N/A
10	0.0422	0.4602	2.0086	N/A
25	0.0795	0.7281	2.9214	9452
50	0.1403	1.207	4.3154	7805
100	0.2665	2.169	7.0185	7657
250	0.6279	4.9422	15.0253	9236
500	1.4082	9.7557	29.4608	9835
1000	3.6653	19.1165	61.9303	11767
str				
1	0.0092	0.1907	1.2626	N/A
10	0.0246	0.282	2.2713	N/A
25	0.0395	0.3842	3.6872	24241
50	0.0694	0.6035	5.6185	25733
100	0.127	1.0043	9.825	32704
250	0.3095	2.1944	22.3717	16210
500	0.7606	4.3166	44.3155	18972
1000	2.4957	10.1851	93.3706	20533
mix				
1	0.0127	0.2388	1.3172	N/A
10	0.038	0.4504	2.2806	N/A
25	0.079	0.7496	3.37	10491
50	0.1366	1.2278	5.2633	8489
100	0.2474	2.2171	9.0959	7585
250	0.5619	5.1926	19.169	9110
500	1.3408	10.441	38.0382	10976
1000	3.5703	20.3337	78.0809	13421

Table 4.11: Compilation times of the different test suites in *the expression experiment* for the JITOPT execution mode

Code generation and compile time

Taking into account the results of Table 4.10, it shows significantly low compilation times, where most of these results are below a 10 millisecond mark. The time used on code generation is something to consider first, where we observe consistent performance across the different sizes and involving data types. Generating the LLVM IR typically takes less than 1 milliseconds, except for the two largest expressions with sizes of 500 and 1000. For both execution modes, `JIT` and `JITOPT`, the time of code generation seems to scale roughly linearly with the size of the expression. However, with an edge case for the expression size of 1 and 1000, and by adding code generation times for some larger sizes, we can see from Figure 4.6 that it scales superlinear, with small margins for the smallest expressions. Implying that the code generation process does not significantly impact the total execution time of a query until we reach unrealistic expression sizes, which are most likely only to be generated by business intelligence tools.

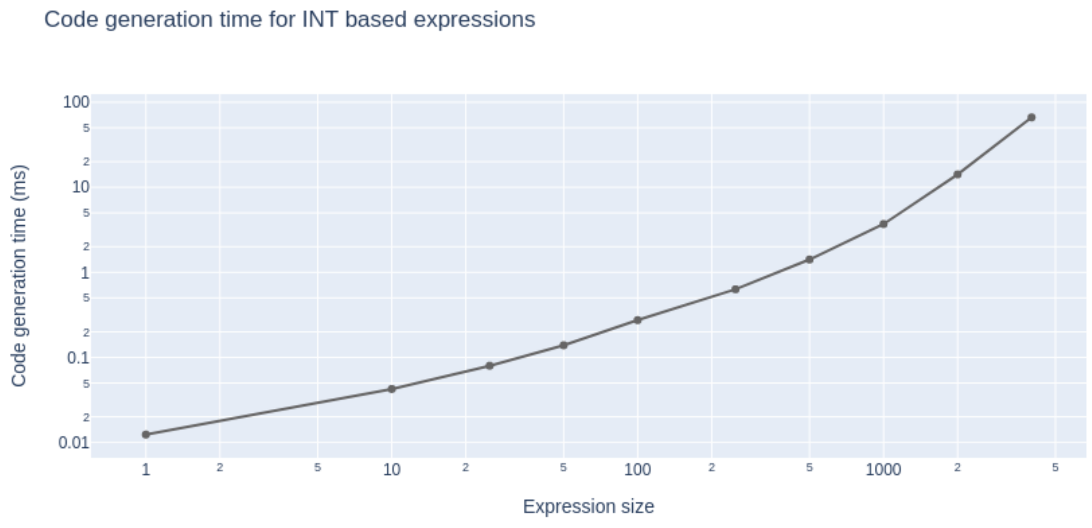


Figure 4.6: Time used on code generation based on expression size (LOG)

Figure 4.13 shows the geometric mean of the expression evaluation speed-up of all queries in the experiment. This figure makes it easy to observe a consistent speed-up among all the scale factors conducted. We can speed up the expression evaluation process by 60% for all query workload sizes. Suggesting that the size of the workload does not affect the potential speed-up, which is not unsurprising, as the time it takes to evaluate a single row is considered nearly constant. The expression evaluation time scales linearly with the number of rows in the table, which complies with the results of a previous project, where we prototyped a simplified replica of the MySQL `Item` tree, and JIT compiled the tree with LLVM [20]. This prototype showed that the expression evaluation time scaled linearly in proportion to the workload scale when evaluating TPC-H query 6 [20]. However, more importantly, this work also showed that the size of the workload affected the performance of the total execution time when applying JIT in Postgres. Compilation time introduces an extra delay to the total execution time, and it might be faster to execute the query without compilation if the workload is small enough.

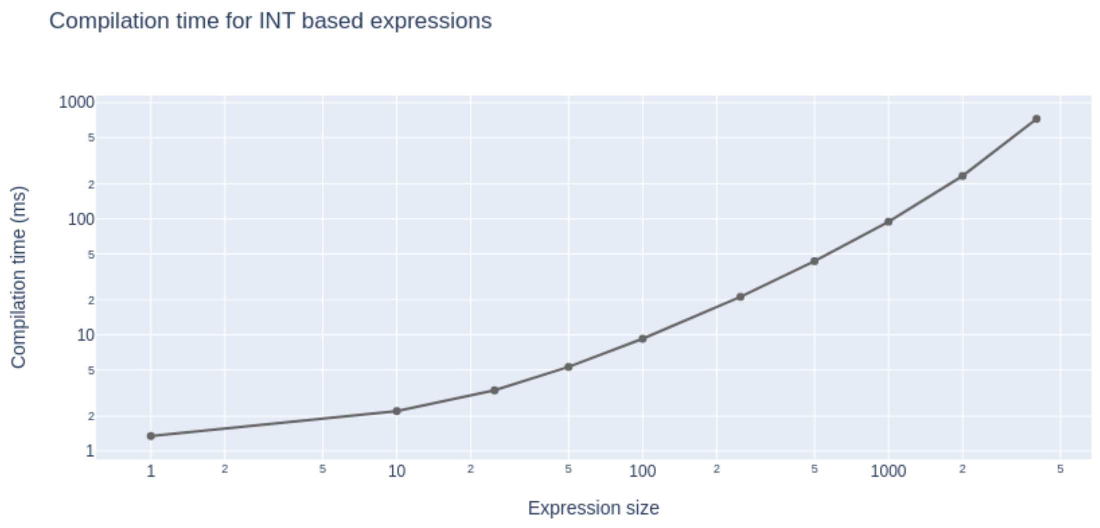


Figure 4.7: Time used on code compilation based on expression size (LOG)

Compilation time vs. workload

An essential aspect of manifesting is whether compiling a query's expressions is performance-wise beneficial for a given workload. How many rows must be processed before a compiled version exceeds the performance of a non-compiled version? In this experiment's context, queries are made up of a single, simple expression of a specific size. An estimation of how many rows are to be processed can be made by solving for X in this equation:

$$\text{JIT val\_int time per row} * X + \text{total compilation time} = \text{nJIT val\_int time per row} \tag{4.1}$$

This equation describes the number of rows X for the two different execution modes `nJIT` and `JIT` when their time used on expression evaluation becomes equal. Solving for X would give a rough idea of how many rows must be processed for the `JIT` execution mode to give any improvement. Tables 4.10 and 4.11 provide these values for X for each size, in the context of the very simple expressions from this experiment. Some of these values are marked as *N/A* due to the lack of accurate time measurement per row for low-sized expressions, as the time precision is not considered reliable enough. On the other hand, considering `INT` based expressions only, the results show values of X for the larger expressions that span an interval of $8-13$ thousands of rows. It is possible to observe that the larger the expression, the larger the workload needs to be for the `JIT` execution mode to become beneficial, which we believe is due to the superlinear increase in compilation time. On the other hand, for `STRING` based expressions, the values of X are unstable since compiling such expressions do not give any significant performance boost, as previously mentioned. Furthermore, for the expression consisting of both data types, we see similar growth as the `INT` based expressions, growing superlinearly in proportion to the size of the expression. These results imply that the larger the expression, the more time needs to be used to compile, which requires a larger workload for JIT compiling to be considered beneficial.

The effects of optimizations

An interesting observation is how applying optimizations has affected compilation times. Applying optimizations has provided extra time by optimizing the IR, shown as *opt* in Tables 4.10 and 4.11. As mentioned in Section 4.2.1, there are not many optimizations that can be made in the generated IR unless the expression contains frequent references to the same column. The lack of optimizations that can be applied is reflected in the results, where the optimization times are less than a 1 millisecond for the smaller expressions and grow from 2 milliseconds to 20 milliseconds for the larger expressions. These times are not significantly impactful and are a fraction of the compilation time. With these low compilation times, the threshold for applying optimization is considered low and might be applicable in most scenarios.

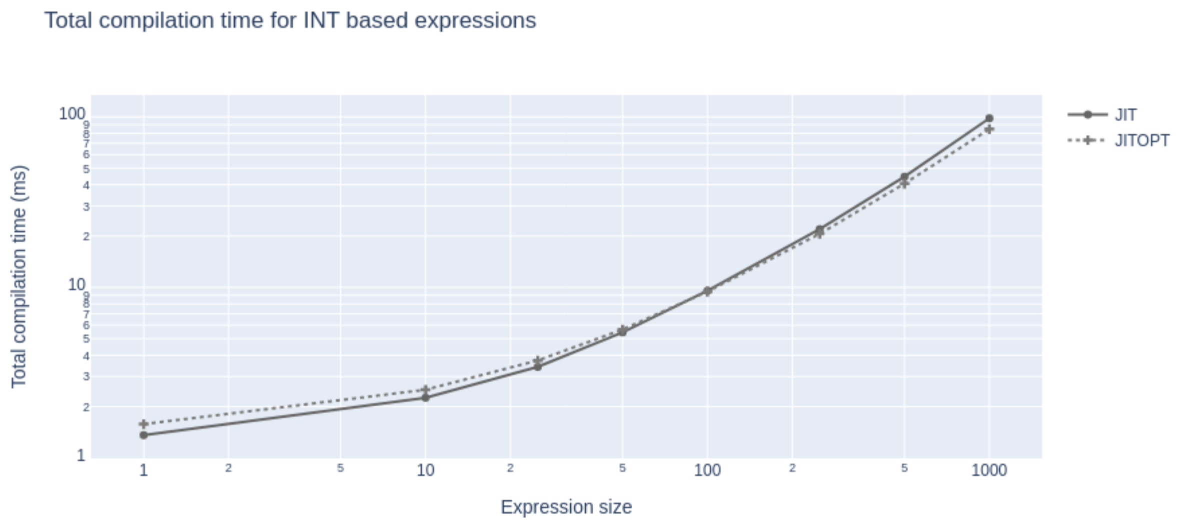


Figure 4.8: Time used on code compilation based on expression size (LOG) for `JIT` and `JITOPT` execution modes (LOG)

On the other hand, the optimization time becomes more impactful for the largest expressions, reaching up to 20 milliseconds. However, an important observation is how applying optimizations has affected the compilation time. The total compilation time without optimizations for the `INT` based expression with a size of 1000 is approximately 98 milliseconds. However, optimizations have reduced the total compilation times to 84 milliseconds! This 15% reduction is due to how optimizations can reduce the LLVM IR to the extent that the compilation time has been significantly reduced. As a consequence of this phenomenon, applying optimizations might not necessarily increase the total compilation time but reduce it. It can be beneficial to optimize, considering only the expression size as a factor. By comparing the total compilation times between the execution modes `JIT` and `JITOPT`, shown in Figure 4.8, we observe that the execution mode `JITOPT` achieves a smaller compilation time than the execution mode `JIT` for the larger queries, due to this phenomenon. This implies that for the expressions that our JIT system generates, there exists a point where the expression can become large enough that applying optimization is considered significantly more favorable regardless of other factors. In this experiment, this point exists around the size of 100, which is a size that is considered unrealistic for human written queries. However, suppose our JIT system was to expand to support more complex data types and reimplement them with LLVM IR. In that case, the generated code might grow large enough to reach this point with more common queries, as long as the IR is reducible in similarity to this experiment. In other words, these results indicate that this phenomenon of applying optimization to achieve reduced compilation times is currently only applicable to large queries created by business intelligence tools. Additionally, the most important implication is that this phenomenon mainly depends on the size of the expression and the compressibility of the generated IR. In contrast, the latter depends on the properties of the expression itself.

4.2.3 Reduced overhead

This section will provide the instruction count for the expression evaluation process from the different test suites. These results are obtained through a specific Linux system call, called `perf_event_open`, which causes the CPU to count the number of hardware instructions executed. By performing this system call during the different test suites of this experiment, we obtained the following results:

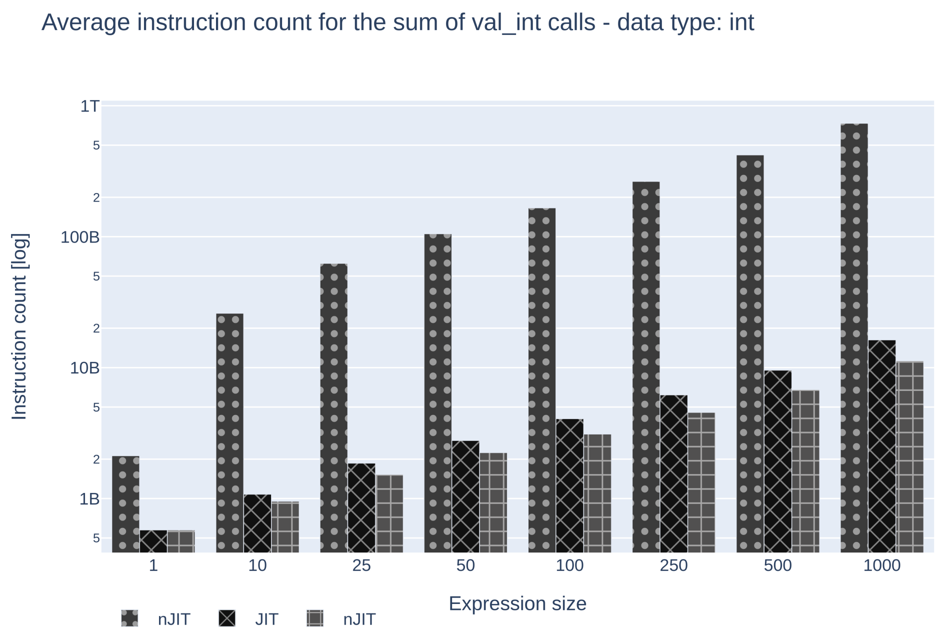


Figure 4.9: The average instruction count for expression evaluation by expression size, showing the data for `INT` based expressions

Average instruction count of `val_int` calls					
Expr size	nJIT	JIT	JITOPT	JIT reduction	JITOPT reduction
int					
1	2.12E+09	5.76E+08	5.76E+08	72.88%	72.88%
10	2.60E+10	1.08E+09	9.54E+08	95.85%	96.33%
25	6.26E+10	1.86E+09	1.53E+09	97.02%	97.56%
50	1.05E+11	2.78E+09	2.24E+09	97.36%	97.87%
100	1.66E+11	4.08E+09	3.11E+09	97.54%	98.13%
250	2.64E+11	6.19E+09	4.55E+09	97.66%	98.28%
500	4.21E+11	9.56E+09	6.80E+09	97.73%	98.39%
1000	7.34E+11	1.63E+10	1.13E+10	97.78%	98.46%
str					
1	1.00E+10	1.02E+10	1.02E+10	-1.62%	-1.62%
10	1.27E+11	1.19E+11	1.19E+11	5.84%	5.84%
25	2.79E+11	2.61E+11	2.61E+11	6.52%	6.52%
50	5.11E+11	4.75E+11	4.75E+11	7.08%	7.08%
100	9.77E+11	9.05E+11	9.05E+11	7.39%	7.39%
250	2.37E+12	2.19E+12	2.19E+12	7.61%	7.61%
500	4.69E+12	4.33E+12	4.33E+12	7.68%	7.68%
1000	9.32E+12	8.60E+12	8.60E+12	7.73%	7.73%
mix					
1	2.12E+09	5.76E+08	5.76E+08	72.88%	72.88%
10	5.66E+10	3.70E+10	3.69E+10	34.59%	34.72%
25	1.10E+11	5.55E+10	5.53E+10	49.64%	49.83%
50	1.87E+11	9.20E+10	9.16E+10	50.72%	50.93%
100	3.33E+11	1.68E+11	1.67E+11	49.57%	49.77%
250	4.71E+11	2.03E+11	2.02E+11	56.87%	57.12%
500	6.74E+11	2.64E+11	2.62E+11	60.82%	61.08%
1000	1.07E+12	3.72E+11	3.69E+11	65.12%	65.41%

Table 4.12: The instruction count of the different execution modes, divided by the `int`, `str` and `mix` expression types.

From the results shown in Figure 4.9 and Table 4.12, we observe a huge reduction in the number of instructions executed for the execution modes `JIT` and `JITOPT`. By simply JIT compiling a single integer-based comparison expression, the instruction count of the evaluation process has been reduced by approximately 73%, and for larger expressions converges towards a 98% reduction. This strongly implies that by JIT compiling an expression, one can remove a large amount of instruction overhead, given that all of the logic of the expression is compiled down. On the other hand, looking at the results for the `STRING` based expressions in Table 4.12, they do not reflect the same reduction as for `INT` based expressions. This is not a huge surprise, as these expressions execute injected C++ functions of the existing MySQL system and execute approximately the same amount of instructions compared to the `nJIT` execution mode. However, by skipping over some virtual function calls, our JIT system can still reduce roughly 5% to 8% of the instruction overhead, giving an idea of how large the overhead a function call provides. Moreover, the instruction reduction for the expressions containing both data types is smaller than that for the integer expressions due to the string-based subexpressions. For these kinds of expressions, the overall instruction overhead is reduced by 50 to 60 percent, showing that our JIT system can still reduce a huge amount of instruction overhead even by including injected MySQL code.

As mentioned earlier, these queries with mixed data types represent an everyday use case. Since we believe a complete JIT implementation would consist of a mixture of injected C++ functions and generated IR code. We consider the result of the `mix` test suite to be a more accurate representation of what instruction reduction can be expected for more ordinary queries. Overall, these results strongly indicate that compiling expressions is an effective way to remove unnecessary overhead introduced by the MySQL ecosystem, implying that MySQL would greatly benefit from conducting JIT compiling of their `Item` trees. Despite these promising results, it is essential to note that our JIT implementation does not consider all the different edge cases supported by MySQL, such as `NULL` values. Meaning that our JIT system might provide a smaller reduction in instructions when expanding to support all the features within MySQL. However, our results still indicate considerable potential for MySQL in eliminating the instruction overhead of the system and achieving a significant amount of speed-up.

Furthermore, the results from Table 4.12 comply with our time-measurement results for expression evaluation, where we achieved almost 100x speed-up with the `JITOPT` execution mode, at best. According to the authors of Hekaton, to go 90x faster, one must execute 90% fewer instructions [13]. To go 100x faster, one must execute 99% fewer instructions. Our JIT system seems to act according to these statements, where we achieved 98.5% fewer instructions compared to the `nJIT` execution mode, which allowed us to achieve the 100x speed-up. Looking further at our results, we obtained approximately 50% speed-up with 72% reduction of instructions and achieved 8x speed-up with 95%. Between an instruction reduction of 95% to 98%, our results show speed-ups between $8x$ and $100x$. With this, we observe that the speed-up scales are superlinear in proportion to the number of instructions reduced, where the growth starts to escalate around a reduction of 70%. This implies that a significant reduction in the number of executed instructions is essential to achieve notable speed-ups, which is why our `STRING` based expressions did not show any significant acceleration.

4.3 Evaluating the TPC-H experiment

In this section, we will show the results of the TPC-H experiment. The goal is to understand how our JIT system compiles `Item` expressions to impact the performance of more realistic business-oriented queries. To begin, we will first provide the results of the different queries in the TPC-H benchmark with a scale factor of 3, corresponding to roughly 3GB of data, and discuss our observations. Next, we will reveal the instruction counts produced by the different execution modes and discuss the relevant changes compared to the `nJIT` execution mode. Lastly, we will provide the results of the different scale factors and show how our system is affected by different workload sizes.

4.3.1 The performance of expression evaluation

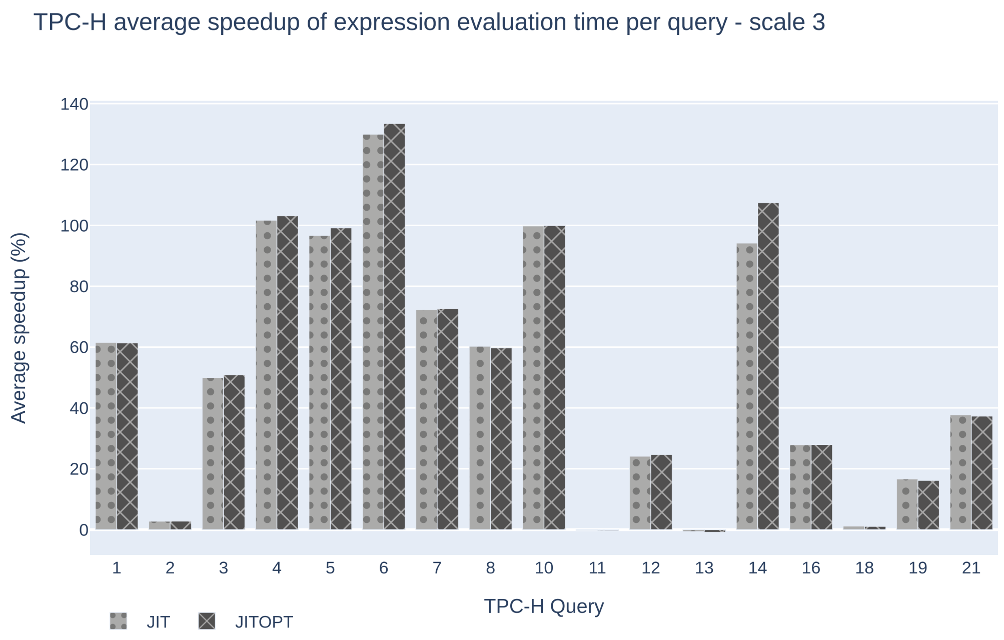


Figure 4.10: The average speedup of expression evaluation time by TPCH query - scale 3

These results show a positive impact on the performance of expression evaluation by JIT compiling the expressions. Figure 4.10 shows that most queries have improved the time to evaluate the rows from 20% to 130%. This indicates that our JIT system is capable of reducing the overhead of `Items` in many realistic query expressions. However, for some other queries, such as query 2, 11, 13 and 18, the evaluation processes have not really given any significant impact on performance. For these queries, we consider the benefit negligible, as the numbers show barely any or negative improvements. This implies that our JIT system affects queries differently and that JIT compiling the expressions of a query is not necessarily beneficial for all types of queries. This means that the queries need to be analyzed in more depth to truly know what is going on.

Analyzing the queries

We start by looking at the queries without notable performance improvement, queries 2, 11, 13, and 18. What these queries have in common is that the MySQL optimizer forms a query plan where the `WHERE` clause is split into multiple expressions, creating a `FilterIterator` for each of them. Some of these expressions are not compilable by our system. They are instead evaluated by an `Item` tree, making the entire expression evaluation process consist of both JIT-compiled and uncompiled expressions. These queries show slight improvement due to having most of the work allocated to the uncompiled expressions, where the compiled expressions do only a tiny fraction of the entire evaluation process. For example, query 11 consists of three different `FilterIterators`, where two are compiled `STRING` expressions, and the last represents a sum filter, filtering by the sum of two columns from a subquery. Most of the evaluation process consists of evaluating these sums. Since the other two expressions have compiled `STRING` expressions, which do not benefit much from JIT compiling, the overall performance improvement is considered non-existent. Both queries 2 and 18 have similar stories, but query 13 is slightly different. It contains a single `FilterIterator`, where the `Item` condition has been compiled but represents a single expression based on the `STRING` data type. Since `STRING`-based expressions execute the same C++ code as the existing MySQL engine, the differences are minor and do not benefit JIT compiling to any considerable extent. With this information, these queries imply that not all `WHERE` expressions show favorable improvements when compiling, further indicating that analyzing the expression before compiling can help avoid compiling expressions that do not provide any benefit.

For the rest of the queries in the benchmark, we observe a significant performance improvement in the expression evaluation process. What all of these queries have in common is that all `FilterIterators` in the query plan contains a JIT compiled `Item` tree, making the entire expression evaluation process compiled, drastically reducing the number of virtual function calls. Moreover, an interesting question to manifest is why there is a huge difference in improvement between queries, as there is a speedup gap of more than 100% between the best and worst of these queries. Looking more closely, the queries providing the best performance consist heavily of `INT` based expressions, and the queries of lower performance consist of expressions based on a mix of both `INT` and `STRING`, where the worst leans towards containing more expressions based on `STRING`. This suggests that the expressions that are actually compiled by using generated LLVM IR are those that drive the performance boost. Therefore, from these observations, it is simple to argue that the more data types this JIT system is able to compile, without injecting existing C++ functions, the more likely we are to improve the performance of the expression evaluation process.

Evaluating the total exeuction time

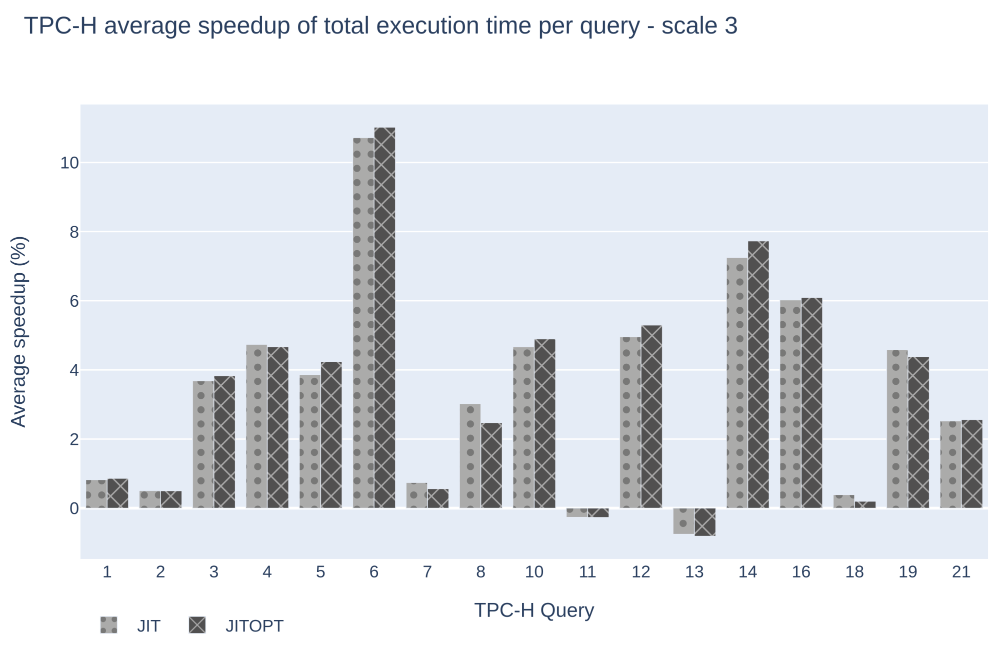


Figure 4.11: The average speedup of total execution time by TPCH query - scale 3

TPCH Benchmark Results - Scale 3 - in milliseconds											
Mode:	nJIT		JIT				JITOPT				
Query	total	val_int	total	val_int	codegen	compile	total	val_int	codegen	compile	opt
1	26810	523	26592	323	0.01	1.37	26581	324	0.01	1.38	0.25
2	337	153	335	148	0.02	3.7	335	148	0.03	3.66	0.56
3	6796	274	6555	182	0.03	3.46	6546	181	0.03	3.44	0.56
4	1952	156	1864	77.77	0.02	1.59	1865	77.22	0.02	1.47	0.3
5	1939	154	1867	78.76	0.04	3.86	1860	77.8	0.03	3.65	0.62
6	4881	838	4408	364	0.03	1.99	4396	358	0.03	1.75	0.38
7	3789	60.33	3761	35.01	0.04	5.13	3768	34.97	0.04	5.07	0.81
8	1903	125	1847	78.29	0.02	2.49	1857	78.55	0.02	2.43	0.4
10	1937	155	1851	77.97	0.02	1.6	1847	77.93	0.02	1.52	0.31
11	1137	515	1140	517	0.02	2.63	1140	517	0.02	2.54	0.38
12	9995	2337	9523	1883	0.04	2.22	9492	1874	0.04	2.09	0.41
13	8324	1735	8388	1746	0.01	1.51	8392	1750	0.01	1.36	0.25
14	5060	666	4718	343	0.02	1.58	4697	321	0.02	1.48	0.31
16	1301	350	1227	274	0.05	3.49	1226	274	0.06	3.22	0.64
18	22950	16064	22861	15879	0.02	1.46	22905	15895	0.02	1.41	0.25
19	8687	2664	8306	2284	0.15	10.06	8323	2293	0.15	9.26	1.74
21	8862	653	8644	474	0.05	6.86	8641	475	0.05	6.64	1.05

Table 4.13: The benchmark results of the TPC-H experiment for scale factor 3 in milliseconds.

Another interesting question to manifest is how the performance improvements in expression evaluation affect the total query time. `Figure` 4.11 shows the average speed-up in the total query execution time for the scale factor. Compared to *the expression experiment*, these numbers are less significant, but still considered satisfactory. Despite achieving a 100% speedup in expression evaluation, most of these provide a small $4 - 5\%$ speedup in total execution time, where the best provide a 11% speedup. The time used on compilation is also considered extremely low, as shown in Table 4.13, where the average compilation time is approximately 3 ms, the worst being 10 ms. These compilation times are considered negligible compared to the actual time used to evaluate the expressions. This further suggests that the process of evaluating expressions is not necessarily the most cost-intensive part of the query execution for these queries.

Expression evaluation % of total query time			
Q	val_int %	Q	val_int %
1	1.95%	11	45.36%
2	45.34%	12	23.39%
3	4.04%	13	20.84%
4	8.03%	14	13.18%
5	7.99%	16	26.95%
6	17.17%	18	70.00%
7	1.59%	19	30.67%
8	6.59%	21	7.37%
10	8.04%		

Table 4.14: The expression evaluation time % of total query time for each query in `JIT` execution mode.

Table 4.14 shows how much time of the total query time is devoted to evaluating expressions. These numbers confirm that the time used on `::val_int()` is not necessarily significant for most queries. Some exceptions are the queries 2, 11 and 18, where expression evaluation represents atleast 45% of the total execution time, but also does not benefit greatly from JIT compiling. The queries that actually benefit from JIT compiling devote only approximately 4% to 20% of the total query time to evaluating the expressions, which suggests that the total query time speed-up is relatively low due to the small workload. The queries that benefit the most from JIT compiling are very simple queries, where the `WHERE` clause is a bigger part of the query, since `GROUP BY`s and `ORDER BY`s are either very small or nonexistent. For example, queries 6, 14, and 19 consist mainly of a `WHERE` clause and an aggregated projection. Unsurprisingly, these results imply that the total speed-up of queries by JIT compiling expressions rely both on the speed-up of evaluating the expressions and the time used on the other parts of the query. This suggests that JIT compiling expressions do not necessarily provide more speed-up than 2% to 10% for many typical business-oriented queries, but is a result that we still consider satisfactory.

4.3.2 Reduced instructional overhead

Since our results in Figure 4.10 and Table 4.13 show clear performance improvements in expression evaluation, it is apparent that the acceleration will be reflected in the instruction count of `val_int()` calls. By measuring the instruction counts for each query, we got the following results:

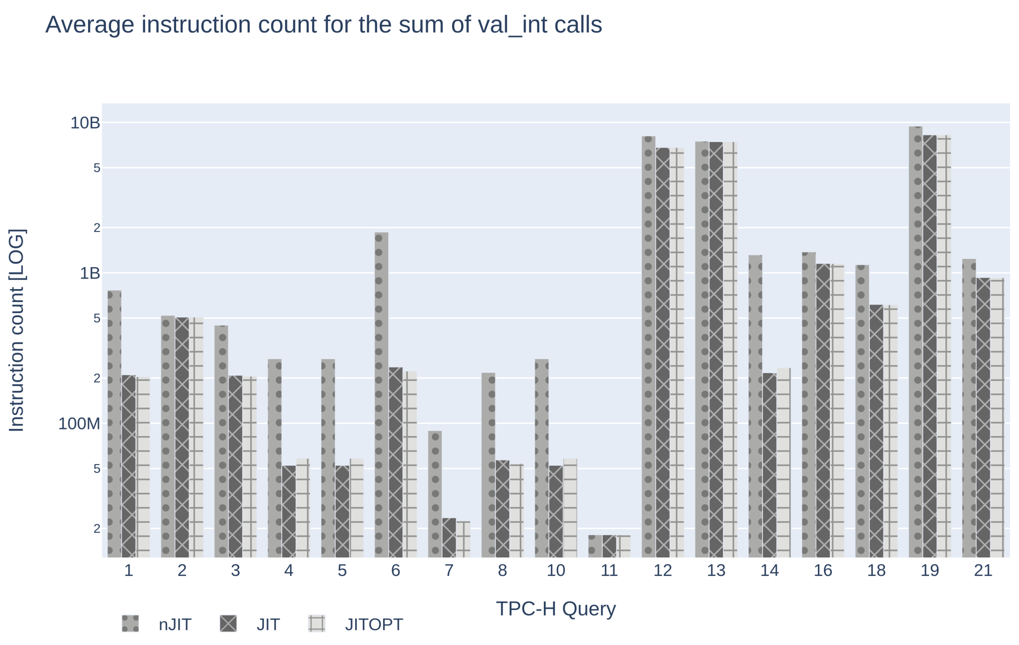


Figure 4.12: The average instruction count of the sum of `val_int` calls per TPC-H query.

Average instruction count for `val_int` calls					
Query	nJIT	JIT	JITOPT	JIT reduction	JITOPT reduction
1	7.68E+08	2.10E+08	2.04E+08	72.65%	73.43%
2	5.21E+08	5.08E+08	5.08E+08	2.67%	2.67%
3	4.48E+08	2.08E+08	2.06E+08	53.51%	54.11%
4	2.69E+08	5.25E+07	5.85E+07	80.45%	78.21%
5	2.69E+08	5.25E+07	5.85E+07	80.45%	78.21%
6	1.87E+09	2.36E+08	2.22E+08	87.34%	88.1%
7	8.94E+07	2.35E+07	2.24E+07	73.68%	74.9%
8	2.18E+08	5.70E+07	5.40E+07	73.79%	75.17%
10	2.69E+08	5.25E+07	5.85E+07	80.45%	78.21%
11	1.81E+07	1.81E+07	1.81E+07	0%	0%
12	8.13E+09	6.82E+09	6.81E+09	16.18%	16.23%
13	7.50E+09	7.45E+09	7.45E+09	0.68%	0.7%
14	1.32E+09	2.17E+08	2.34E+08	83.54%	82.22%
16	1.38E+09	1.15E+09	1.15E+09	16.21%	16.29%
18	1.13E+09	6.17E+08	6.14E+08	45.56%	45.83%
19	9.45E+09	8.27E+09	8.27E+09	12.51%	12.52%
21	1.24E+09	9.31E+08	9.28E+08	25.17%	25.42%

Most of the queries show an impressive reduction in the number of executed instructions by JIT compiling expressions. Many queries show a significant reduction of 70% to 86%, and others show a reduction of almost nothing up to 45%. As stated in *the expression experiment* 4.2.3, to achieve a speed-up of 10x, a reduction in the number of instructions of 90% must be achieved. The results of this mentioned experiment show a reduction in the number of instructions from 70% to 80%, which provided a speed-up around 50% to 100%. This complies with the speed-up achieved in this benchmark, where all queries with a reduction greater than 70% show a speed-up between 60% and 100%. A suggestion of why we are not achieving larger reductions is simply the fact that we are limiting ourselves when injecting calls to existing C++ functions for `STRING` based expressions. Most of these queries contain expressions that compare `STRINGS` in some way, and by using the same code to evaluate these expressions, the unwanted overhead from the MySQL ecosystem does not disappear. In addition, for queries containing only `INT` based expressions, we further suggest that the expressions are not large enough to exceed a greater reduction in the number of executed instructions. As shown in the previous experiment 4.2.3, the larger the expression, the more overhead there is to remove, the greater the reduction in the number of instructions. This implies that for a typical business-oriented query, the reduced overhead might not exceed much further than these results, given the typical expression sizes and the containment of unsupported data types. However, for larger queries, such as those created by business intelligence tools, we believe that the expressions could be large enough to exceed these results and could achieve similar speed-ups as *the expression experiment*.

4.3.3 Workload sizes

This section will provide test results for how our JIT system affects the performance of different workload sizes. During this experiment, we ran the TPC-H benchmark on different scale factors, ranging from approximately 100 MB to 30GB. The data were generated by a TPC-H data generation tool and inserted into the MySQL server instance as a separate database for each scale factor.

Our results from *the expression benchmark*, in Section 4.2.1, suggest a particular speed-up for the relevant queries introduced in that experiment. The experiment results were run on a dataset similar to the TPC-H scale factor 3, roughly a table with 3GB. If we were to run that experiment on different scale factors, where the time used in evaluating a single row is considered quite stable, we would presumably, and unsurprisingly, achieve the same speed-up regardless of the table size we are processing. The only thing that would affect the speed-up would be the time it takes to compile the expressions, which is considered fairly low in both experiments. We assert that this impression of how the speed-up is little affected by the table size also applies to the TPC-H experiment, which the following results can show:

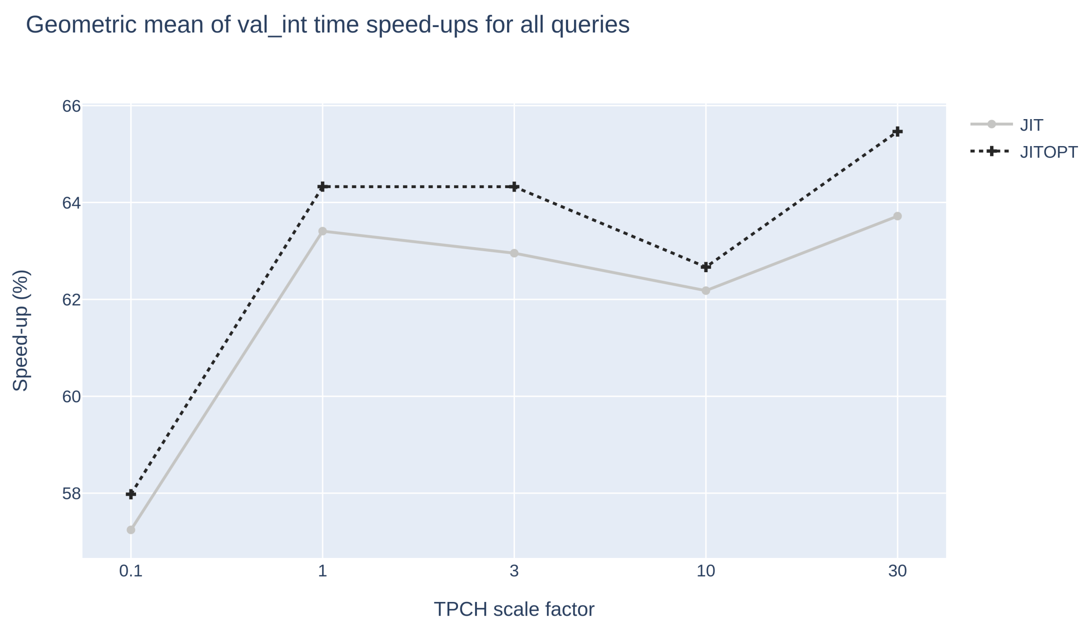


Figure 4.13: The geometric mean of the expression evaluation speed-up of all queries in the TPC-H benchmark.

Figure 4.13 shows the geometric mean of the expression evaluation speed-up of all queries in the experiment. This figure makes it easy to observe a consistent speed-up among all the scale factors conducted. We can speed up the expression evaluation process by 60% for all query workload sizes. Suggesting that the size of the workload does not affect the potential speed-up, which is not unsurprising, as the time it takes to evaluate a single row is considered nearly constant. The expression evaluation time scales linearly with the number of rows in the table, which complies with the results of a previous project, where we prototyped a simplified replica of the MySQL `Item` tree, and JIT compiled the tree with LLVM [20]. This prototype showed that the expression evaluation time scaled linearly in proportion to the workload scale when evaluating TPC-H query 6 [20]. However, more importantly, this work also showed that the size of the workload affected the performance of the total execution time when applying JIT in Postgres. Compilation time introduces an extra delay to the total execution time, and it might be faster to execute the query without compilation if the workload is small enough.

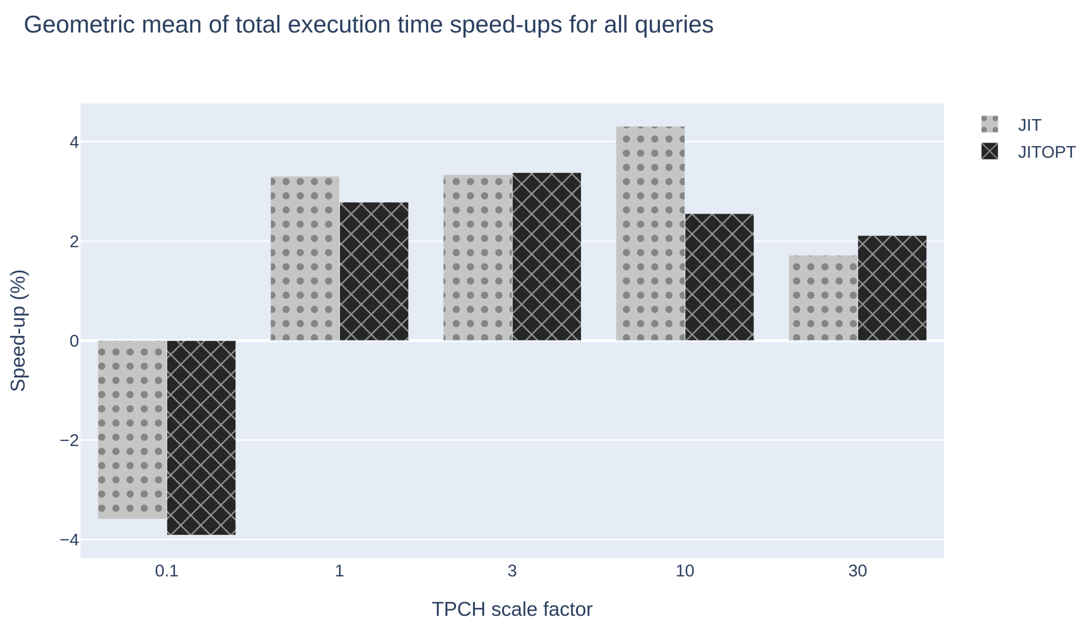


Figure 4.14: The geometric mean of the total query execution speed-up of all queries in the TPC-H benchmark.

As shown in Figure 4.14, despite the acceleration in expression evaluation, the performance of the total query execution time decreases for the scale factor 0.1, which corresponds to approximately 100MB. However, applying JIT shows positive performance impacts for the scale factor of 1, suggesting that the compilation time is too large for the scale factor of 0.1 compared to the total execution time, implying that applying JIT is not considered beneficial for this workload. Further indicating that concerning these types of queries, there is a point between the scale factor of 0.1 and 1 where applying JIT becomes beneficial. Moreover, from these results, it is easy to argue that when MySQL evaluates if it should compile an expression, the size of the workload plays a role in the final performance outcome and must be considered.

As shown in Table 4.10 from *the expression experiment*, there are not many rows needed in a table before the time saved by applying JIT exceeds the total compilation time. From these results, the number of rows where this typically occurs presumably seems to be around a couple of tens of thousands of rows. This result differs significantly from the earlier experiment with Postgres, where this point seemed to lie between scale factors 1 and 2 [20]. Postgres compiles much more than presented in this thesis and has much larger compilation times than shown here, which explains the difference. This is due to the assumption that a complete JIT implementation includes more of the different features and data types that MySQL provides and therefore increases the generated code's size, further increasing the compilation time. With this, we believe that the point in workload size where JIT becomes beneficial shifts towards larger sizes for a complete JIT implementation in MySQL. Consequently, this would decrease the performance for smaller workloads to a greater extent, which would be the TPC-H scale factor of

0.1 in this scenario. Therefore, when evaluating when to JIT compile expressions in MySQL, workload size must be considered carefully. This, again, intensifies the need for a good cardinality estimator, which has been mentioned as a critical factor in performance in several research studies [25] and is what Postgres depends on when evaluating whether to JIT.

4.4 Final evaluation and summary

At the beginning of this chapter, we focused on how we can use our JIT implementation to establish an answer to the research question. We summarized the three additional research questions that we believe are important to understand to conclude, in which our conducted experiments were to create the fundamental basis for that understanding. The first question asked about the performance expectations of JIT compiling expressions in MySQL, where our experiments indicate what to expect. Moreover, the second question raised the problem of understanding when it is considered beneficial to JIT an expression and what factors are essential. Our experiments have developed several implications for these factors and their significance under different circumstances. The third question concerns LLVM and whether it is a good choice for MySQL. We can now draw some conclusions from what we have observed in our experiments.

4.4.1 Performance impact of jitting expressions in MySQL

Reduced overhead

Our JIT system has shown great capabilities to remove the overhead introduced by the MySQL ecosystem, such as unnecessary virtual function calls and extra code to handle all functional cases. For simple queries containing only expressions based on integers, the system was able to reduce over 95% of the instructions to evaluate the expressions by utilizing the extra information provided at runtime. For more realistic business-oriented queries, the system was able to reduce over 80% of these instructions for the cases where the JIT system was used to its full potential. This suggests that there is great potential for MySQL to accelerate the expression evaluation process by eliminating a lot of the overhead that seems to exist. However, this reduction in the number of instructions does not seem to pay off until it reaches at least 80%, which is something we believe from our results to be a common achievement in a complete JIT implementation in MySQL for most typical business-oriented queries.

For very large expressions, we achieved a 100x speed-up in evaluating the expressions. For business-oriented queries, the speedup was more commonly found within $60 - 100\%$. The two experiments conducted suggest that larger queries introduces a lot more overhead than smaller queries, which can be further removed by JIT compiling and give greater reduction numbers. The TPC-H benchmark represents a group of smaller queries that is a more common and realistic use case, and therefore gives a more accurate view over the expectations by JIT compiling expressions in MySQL. For that reason, we believe that at least 80% reduction in the number of instruction is what we can expect from this JIT implementation on `WHERE` expressions. The results showing lower reduction numbers were due to the lack of using the JIT system to its full potential, where the existing `Item` system was used to evaluate larger parts of the expressions. We realize that injecting C++ functions that evaluate entire expressions is probably not the most optimal approach to satisfy this potential, as the overhead we want to eliminate persists. We also realize that we lose a lot of the potential for optimizing the generated code, missing out the possibility of achieving better performance.

LLVM templating

However, we recognize that there is an alternative approach for avoiding duplicating the logic of complex data types: LLVM templating. For instance, by generating the LLVM IR for string comparisons in advance as a template from the existing C++ code alone, we can represent the evaluation of `STRING` based expressions in the LLVM IR without having to maintain two different codebases for the same logic. Templating also allows us to optimize where we can remove more unnecessary overhead known at runtime. Similar JIT systems, such as Postgres [14, 30], Impala [49] and others [24, 44] already use this, where backend functions are precompiled and loaded into memory on start up, and then used to generate code for the specific expressions. By utilizing this dynamic template expansion approach to generate the IR, it is easier to support the different data types and features in MySQL and, at the same time, remove some of the overhead and preserve code maintainability. We decided not to do this due to time restrictions and early technical challenges. We see it as a missed opportunity for this thesis and a fascinating investigation for future work.

Speed-up of total exeuction time

An important observation from our result is to what extent the speed-up of expression evaluation affects the total query execution time. Speeding up expression evaluation by 100% seems to only accelerate the total execution time by 10% to be optimistic. The typical business-oriented queries provided by the TPC-H benchmark use a larger part of the execution time on other parts of the query, besides from expression evaluation, such as aggregations and sorting. This suggests that we are not compiling enough to achieve a greater reduction in the number of instructions. Other research papers experimenting with JIT compiling in Postgres have shown between $20 - 30\%$ speedup on TPC-H query 1, where they have compiled both filter expressions and aggregate projections (`SUM`, `COUNT`... etc.) [30, 5]. Our implementation barely achieves 1% speed-up for query 1, as we only compile filter expressions that consist of less than 2% of the entire execution, as shown in Table 4.14. Query 1 is very heavy on aggregations, which occupies 76% of the execution time in Postgres, according to one of these studies [5]. Since MySQL has an architectural nature similar to that of Postgres, it is therefore feasible to argue that MySQL has similar potential for performance improvement by compiling aggregations in runtime. Since MySQL implements aggregations as `Items`, proceeding with this idea is something we consider a natural expansion of our JIT implementation.

The performance of MySQL with the JIT

Altogether, this shows that MySQL has great potential in achieving sufficient speed-ups by JIT compiling expressions, and could achieve greater results than presented in this thesis by supporting aggregations and dynamic template expansion of LLVM IR. We acknowledge that our implementation is not complete and therefore would not be an accurate representation of what to actually expect performance-wise by a complete implementation. However, our implementation shows that MySQL introduces a lot of overhead created by the interpreter and the `Item` logic, which can be easily removed by JIT compiling the `Item` expressions into native machine code in runtime. When our JIT system is used close to its full potential, the total query time speed-up seems to be around $4 - 10\%$, which we find to be a satisfactory improvement for many long-running CPU-bound queries. For extremely large expressions, we were able to achieve a 100x speedup, which might be relevant for machine-generated queries, such as those created by business intelligence tools. We achieved these results by specifically compiling integer-based expressions into native machine code, which have shown positive results for most queries. We believe that there is much room for improvement for MySQL in terms of performance in this area, as the current `Item` implementation appears to be insufficient for many modern CPU-bound queries.

4.4.2 When to JIT in MySQL?

An essential aspect of JIT compiling is considering when it is beneficial to compile, as the overhead of compilation time might be higher than the time that can be saved. Since the compilation time also increases based on the size of the expression, we note that the size and complexity of the expressions are also important to consider. Additionally, our experiments imply that compilation might not be beneficial if any of the expressions consist of injected C++ function calls to the existing MySQL ecosystem, such as implemented for string operations in this thesis. In other words, depending on the implementation, the different types of expressions are essential to consider whether they contribute to reducing the number of instructions and whether the different data types drastically increase the compilation times due to extra complexity. Based on our results and observations from the experiments of this thesis, we suggest considering the following factors when determining whether to implement JIT compilation in MySQL:

- **Workload size** - Our experiments showed that JIT compiling expressions was not beneficial for small-running queries, as the table size was relatively small. The lack of benefit can be explained by the overhead of the compilation time being higher than what the system can speed up and is therefore not considered sufficient for such small sizes.

- **Estimated compilation time** - To consider what workload sizes are applicable, a rough estimate of the actual time used to compile is needed. This estimation should, for example, be based on expression sizes, as the results from *the expression experiment* clearly show that the compilation time grows superlinear in proportion to the expression size. Moreover, another factor that plays a role is the complexity of the different operators and data types.

The results from the experiments show that simple integer expressions introduce short compilation times. However, we realize that for more complex data types, such as string expressions, there is a lot more complexity and code in hand that needs to be compiled, which can increase the compilation time to different levels. The increased compilation time due to more complex expressions can be reflected in a Postgres research paper for JIT, showing compilation times almost 40 times higher for TPC-H query 1 [5] due to the added complexity by aggregation expressions.

- **Estimated code reduction by optimizations** - As shown in *the expression experiment*, Section 4.2.2, applying optimizations might be able to reduce the number of the generated code lines significantly. To the extent where the actual compilation time for that code is much lower than the original compilation time, even with the extra time optimizing. The result of this is that applying optimizations might be a way to reduce the total compilation time and therefore make the compiling of the expression worth doing. For instance, in the context of our JIT implementation, if an expression repeatedly references the same set of columns, reducing the number of times the column is read from memory will reduce the number of instructions that need to be compiled. However, we recognize that this might be a difficult task to do right, especially with the various operations and data types. Furthermore, this phenomenon may not be accurate for more complex filter expressions, as several research studies have implied a more rapid exponential growth for optimization times in the number of instructions compared to compilation time [23, 21]. Therefore, this factor needs to be thoroughly studied in a complete implementation before it can be applied. However, our test results show that this phenomenon exists and could be considered for sizeable integer-based filter expressions.

With these factors, a further question is how can we apply them to MySQL? The results of our JIT implementation show that JIT is sufficient in most scenarios, as the compilation time is considered extremely low. Unsurprisingly, this is because the system hardly compiles any code, as most of the compiled code is native integer operations and function calls. Therefore, we recognize that our thesis cannot provide an accurate description of how MySQL should decide to compile since our JIT system does not represent a complete implementation. A complete system would presumably introduce more considerable and more various compilation times, making it difficult to suggest any specific numbers in this thesis.

On the other hand, on the basis of our observations, we are able to elaborate on which approach MySQL should take in this regard. The simplest and most natural approach for MySQL is to make this decision at plan time, before the actual execution. The existing optimizer engine (Hypergraph) already utilizes a cost model based on cardinality estimation, estimating the total number of rows to be processed at the different levels of the query plan, and other factors. These estimates are useful for determining the potential workload size, which we believe is the most crucial factor to consider when deciding whether to JIT. The JIT implementation in Postgres (as of version 14) determines whether JIT compilation should be applied by using the total estimated cost of the query and a configurable threshold [50].

This cost value is based on different aspects of the query, such as disk reads, filter operation costs, and cardinality estimates. For this reason, using the total estimated cost value to decide whether to JIT is something we consider a suboptimal choice, as the final cost value does not necessarily directly reflect the cost of what the JIT is actually compiling, and therefore can not be considered a 100% reliable measurement. Furthermore, many database systems suffer from poor cardinality estimates, producing poor query plans and large errors between the estimated cost and the actual cost[25]. From this, incidents have occurred where Postgres JIT has caused many queries to run significantly slower [19], especially due to added optimization times. Therefore, adopting the same model in MySQL is something we consider an unreasonable approach.

Alternatively, we suggest using a separate cost model for MySQL, where the factors mentioned above are the focal components of the model. This model focuses on the actual cost of the expressions and the potential cost of JIT. Such a model still relies on cardinality estimates to measure workload sizes but also requires a foundation for estimating compilation time for different operations and data types. The latter requires a study of the potential cost that the different operations and data types might introduce. An estimated time for optimizations is also helpful in calculating the total potential compilation time, which might be challenging to estimate. However, an empirical analysis is something we consider a reasonable approach. However, several research studies imply that relying on cardinality estimates can lead to poor query plans. A more reliable approach is to determine the query plan parameters at runtime [25, 12, 23, 31]. For instance, Hyper's adaptive execution engine decides whether to JIT compile at runtime, where runtime statistics is the basis of the decision, achieving low latency and high throughput for most queries[23]. A research study on adaptive execution in NoisePage does a similar approach, performing metric sampling to modify the execution behavior of JIT-compiled components at runtime[31]. Such a runtime analysis requires allocating multiple threads for executing a single query, which currently does not fit within the MySQL thread model and implementation [38, 33], where there is one thread per connection. For this reason, we believe the best approach for MySQL in its current state is to make this decision ahead of execution, analyzing the expressions of the query and making a decision based on cardinality estimates, compile-time estimates, and estimated evaluation time.

Based on what we have observed from our JIT implementation, for simple integer filter expressions, JIT compiling is also considered sufficient for short-running queries, as the compile time is significantly low. However, when introducing more complex operations and data types, we state that the compile times will increase, making JIT sufficient only for long-running CPU-bound analytical queries, which Postgres [50] and Impala [49] already suggest. The more features, operations, and data types in MySQL we compile, the more accurate we find this statement.

4.4.3 Is a good choice LLVM for MySQL?

The results of our experiments show that LLVM is an excellent candidate to help MySQL achieve performance gains and reduce some of the overhead of the MySQL ecosystem. LLVM provides very efficient compilation times, which have been roaming around a couple of milliseconds in this thesis. The observed fast compilation times are a strong argument for utilizing LLVM as the compiler, as it achieves low latency and makes compilation more applicable for small workload sizes. Furthermore, from our experience, the LLVM target-independent code generator integrates exceptionally well with the existing `Item` system and has not introduced any substantial complications to the existing codebase. The nature of `Items` fits very well for a code generation process, where each `Item` can generate its code independently of others. The process of generating IR and ending up with the native code equivalent works seamlessly, as all the complications get offloaded to the LLVM toolchain, where MySQL only needs to focus on generating the IR and calling the compiled output function. All of this shows that LLVM is highly suited for the job where the goal is performance, and we state that LLVM is a perfect candidate in this regard. From work done with LLVM and MySQL throughout this thesis, we have found the following advantages of utilizing LLVM for JIT compiling expressions in MySQL:

1. **Maintainability** - By using a library such as LLVM, MySQL can offload a large amount of compiler complexity. We believe it would be unwise for MySQL to maintain its compiler library for JIT, as it would require maintaining a complex codebase for different machine targets, architectures, and specialized optimizations. Maintaining such a compiler system requires much attention, and we do not see it worth spending time on for a database system. We suggest that offloading the work of JIT to a library is the best approach for MySQL in this regard, where we believe that LLVM is the library that best fits MySQL, as other large-scale alternatives are not as flexible and efficient[35].

2. **Performance** - Research indicates that LLVM provides slightly better-optimized code performance and lower compilation times compared to GCC, the second most goto open-source alternative that fits the C-like code environment of MySQL [35]. Their findings also comply with the results of this thesis, where LLVM shows low compilation times and provides good performance.

3. **Flexibility** - The LLVM provides a large set of flexibility concerning JIT, especially for optimizations. Having control over which optimizations to apply is a substantial benefit for LLVM, whereas applying optimizations can be time-consuming and applicable at different levels depending on the cost of the expressions. Moreover, not all optimizations are considered relevant for the generated IR and are not worth spending time on. Therefore, we state that LLVM is a good choice for a query compiler, as the various queries require different optimization levels depending on the circumstances.

However, there are also some drawbacks to using LLVM in MySQL. LLVM receives major updates every six months, which might introduce breaking changes requiring extra attention and maintenance. In addition, we have found that LLVM is a huge dependency, where the binary size of the MySQL Server used in this thesis increased by 10x when statically compiled. The increased binary size might not be acceptable for all systems, and there may be some platforms that MySQL supports but LLVM does not support, which means that there are other challenges to using LLVM in addition to the functional aspects. Such as support and licensing, which we find out of the scope of this thesis to make any conclusions regarding.

We believe that LLVM is the most obvious choice for MySQL when adopting JIT compiling of expressions, as it provides good performance, low compilation times, flexibility, and low maintenance. MySQL might achieve better performance by creating its JIT system but must pay a considerable maintenance cost and spend time supporting different target platforms and architectures. For this reason, offloading the complications of JIT to an external library would be considered reasonable, where we find LLVM to be a very suitable candidate.

Chapter 5

Conclusion & Future Work

In this last chapter, we summarize our findings and contributions from this thesis and give a final answer to the research question, RQ1, *"Is it possible to improve query performance in MySQL through JIT compiling expressions?"*. To support our answer, we will also provide our answer to RQ2, RQ3, and RQ4 based on our findings in this thesis. Finally, we will suggest future work on JIT compiling expressions in MySQL.

5.1 Conclusion

Throughout our work in this thesis, we have revealed that the internal workings of evaluating expressions in MySQL introduce excessive instruction overhead. Fortunately, we have also shown that a large portion of this overhead can be significantly reduced by compiling the `Item`-trees into native machine code at runtime, substantially increasing the evaluation process's performance.

This instruction overhead shows that it is possible to improve query performance in MySQL by JIT compiling expressions, which answers the main research question, RQ1. However, to better understand why this is the case, we also need to conclude our answers to the other research questions, where RQ2 answers the performance question, RQ3 addresses when it should be applied, and RQ4 addresses the usage of LLVM. Therefore, the following sections will answer these questions one by one, and we will also provide our conclusion at the end.

5.1.1 RQ2: The Impact on Performance

In the expression experiment, in Section 4.2.1, we found that for simple integer expressions, it is possible to achieve a 100x speed-up for massive and long-running CPU-bound queries. However, for more typical queries, we have deduced that a compiled expression can double the evaluation performance, given the typical size and complexity.

Moreover, in the TPC-H experiment in Section 4.3, we found that while gaining a speed-up of about 10% by just evaluating filter expressions, we believe that there is potential for even greater speed-ups. Compiling filter expressions alone still leaves much overhead, which could be further improved by JIT compilation. Filter expressions are only a tiny part of the total execution for most queries. We consider the 10% speed-up a satisfactory result, as MySQL can still compile a more significant part of the query execution, such as aggregate expressions. The ability to compile a more significant part of the query execution implies that MySQL has excellent potential to reduce the number of instructions executed and improve the query performance by JIT compiling expressions.

We also found that injecting function calls to existing C++ evaluation code into the compiled expressions does not provide any notable performance benefits, as the instruction overhead remains. However, we recognize that this technique is a way of supporting complex operations and data types that are difficult to compile and contributes to avoiding situations of having to compile an `Item`-tree into multiple programs. However, for complex operators and data types, we suggest that MySQL attempt to support them through dynamic templating techniques, where the IRs of these operators are precompiled and optimized before the start of the query engine. Supporting operators through dynamic templating incentivizes better support for operators, maintainability, optimization, and performance.

There might be cases where templating is considered insufficient due to high code complexity, such as complex math functions. In these cases, we consider injection of function calls to existing C++ functions to be a more reasonable approach.

Our answer to RQ2 on the performance impact of JIT compiling expressions boils down to the fact that it can vary. The expression's size, complexity, and scale of the filter compared to the rest of the query affect the total query performance. Our results reveal that 10% speed-up in total query performance is likely for most business-oriented queries but can also reach 16x for large, long-running machine-generated analytical queries. Since there is also more that can be compiled, this suggests that more performance improvements be made, as more of the overhead provided by unsupported operators can be reduced.

5.1.2 RQ3: Considering when to JIT

During the TPC-H experiment from Section 4.3, we verified that JIT compiling is not necessarily sufficient for performance under all circumstances due to the added time delay of compiling. We observe that the compilation time varies according to workload size, expression size, and operator and data type complexity; due to the inflated number of instructions these factors supply. In Section 4.2.3 we have shown that expression sizes and complexity strongly influences compilation time.

Consequently, we propose estimating the compilation time before execution and using these factors to predict whether JIT compilation would provide performance benefits that outweigh the cost. This proposal is based on the information obtained from the expression experiment. We demonstrated that simple integer expressions provide extremely low compilation times but also scaled superlinearly in proportion to the expression size. Furthermore, previous research has also stated that the compilation time increases superlinearly with the number of instructions [23], which will be the case when expanding to support more complex operations and data types beyond integers.

To answer RQ3, we want to stress that a complete JIT implementation on expressions in MySQL should attempt to estimate the compilation time to ensure improvement and avoid harming the performance. This estimation time should also be considered together with the workload size to predict whether the performance benefits of JIT outweigh the cost. We recognize that this thesis cannot provide exact numbers but instead reveal the different factors that affect whether JIT compilation is beneficial for a given query.

5.1.3 RQ4: LLVM and MySQL

Based on experiences and observations throughout this thesis, we claim that LLVM is a good candidate for MySQL when adopting JIT compilation, considering performance and simplicity. LLVM provides MySQL with low compilation times, fast execution, a feasible integration process, and the automatic benefit of future compiler improvements with little effort. However, we recognize that there are some other drawbacks to adopting LLVM, such as providing years of technical support for MySQL, and we find drawing any conclusion in regards to this outside the scope of this thesis. Despite this, based on what has been mentioned, our answer to RQ4 is that we consider LLVM a promising approach for JIT compiling expressions in MySQL.

5.1.4 RQ1: Final conclusions

The work done throughout this thesis provides a significant first step towards achieving runtime compilation in MySQL. Our work has revealed that MySQL has excellent potential to increase its performance by adopting JIT for expressions due to the interpreted nature of `Items`. From this statement, we want to answer RQ1 and conclude that MySQL can indeed improve performance by JIT compiling expressions.

We believe that JIT compilation would make MySQL a stronger competitor in terms of performance and help MySQL keep up with modern database research. We propose that MySQL adopt JIT compiling expressions with the LLVM compiler framework to obtain better query performance. We also suggest using dynamic templating to support the different operators and data types in MySQL in order to get the most of the performance benefits of JIT compiling. Lastly, to avoid performance loss for simple short-running queries, evaluating the circumstances and estimating the compilation time are crucial and go hand in hand with a JIT compiling database system.

5.2 Future work

This thesis has contributed to one of the first steps for achieving a JIT-supported query engine in MySQL. In the short time we had, we managed to implement a system for incrementally introducing JIT to the various `Item` classes. We also used this system to implement JIT on codeItem classes concerning integers and strings. The `Item` class JIT implementations we made allowed us to demonstrate the benefits MySQL can see from a full JIT implementation. Therefore, in this section, we suggest future work and further research on this topic that we see as further relevant steps to advance our research.

5.2.1 Complete the implementation

The implementation presented in this thesis was restricted to a few simple data types and operators, where only integer expressions were fully supported. Some string expressions were also supported through injection of function calls to existing MySQL code, which we found was not an optimal solution performance-wise. Therefore, it would be interesting to expand the support to more operators and data types where the level of complexity is a little higher than integers. Expanding the support and introducing more complexity to the generated code can affect overall performance and compilation time. We find this to be the next step of this work, as we can better understand how JIT compilation can affect performance in MySQL.

Here, it would be logical to investigate the usage of dynamic templating, where the different operations and data types in MySQL can be supported through precompiled IR templates. It is interesting to see what would be a good implementation model for dynamic templating in MySQL, for instance, if these templates should be linked into the generated program lazily or prior to the execution.

Furthermore, it would be sensible to expand to support aggregate expressions and compile the entire `SELECT` clause to increase the total query performance. By expanding to compile aggregate expressions, more of the interpreted overhead of `Items` can be removed, affecting performance and reducing the number of executed instructions. The interesting aspect to investigate here is the level of overhead produced by these expressions and further compare it to the overhead of filter expressions. It is also interesting to see at what scale this overhead can be removed by JIT compiling at runtime.

5.2.2 Measure compilation costs

This thesis stresses the importance of evaluating the compilation cost of the different operators and data types in a complete JIT implementation in MySQL. Therefore, a future study can address the cost of compiling the different components of the expression and further build a cost model that can be used to estimate the compilation time of the different expressions in the query. Such a model would be crucial to determine whether compiling an expression would exceed the cost of compilation time and benefit performance and should therefore be carefully studied.

Moreover, it would be interesting to investigate the cost of utilizing dynamic templating and applying optimizations for the different components in an expression versus the cost of simply injecting function calls. For instance, there might be cases where the complexity of an operator makes it too costly to compile and should instead be supported through function calls to existing C++ code. These cases also apply to optimizations, where the optimization cost can differ significantly between the different operators and should therefore be studied to understand the optimization time better.

5.2.3 Removing state in expressions

In Section 2.3, we argue that MySQL faces a hurdle when it comes to achieving query parallelism, where state in `Items` are considered one of the difficulties. In this regard, it would be interesting to investigate how the JIT compilation of expressions can contribute to removing this state and making `Items` a stateless entity. As mentioned, state is, for example, used to manage `NULL` values of rows or other similar behaviors, which would require refactoring of the existing `Item` behaviors to remove.

With this said, when generating code for an `Item`, it is possible to generate code that makes the evaluation of each row stateless, which can further open up the possibilities of parallelizing this process. We believe that MySQL can achieve extreme performance improvements by parallelizing the table scan and the evaluation of expressions. Therefore, it would be an interesting study to investigate how JIT compiling can be used to eliminate the state in `Items` and achieve parallelism in expressions.

5.2.4 Investigate adaptive execution

Our last suggestion for future work on this subject is the possibility of adopting an adaptive execution strategy in MySQL. In Section 2.5.4, we introduce the adaptive query execution strategy employed in Hyper, where they achieve low latency and high throughput by transitioning between different execution modes at runtime [23]. When compiling ahead of execution, the system sacrifices low query latency, which they avoid in Hyper by starting the compilation process while the query is already running. A JIT implementation on expressions in MySQL would also suffer from the latency of the ahead-of-execution compilation. It could also benefit from employing a similar adaptive strategy for the compilation of expressions.

For this reason, we suggest researching the possibilities of employing such a strategy in MySQL for expression-based compilation. What are the possibilities in MySQL for compiling an `Item`-tree while the existing interpreter is already running and switching to the compiled code when the compilation is complete? This execution model would require additional worker threads for a given query, which is currently restricted to one, as of MySQL version 8.0.28. Investigating the possibilities for increasing the thread count for a single query to adopt such an adaptive execution strategy can be a worthwhile investment to reduce the latency cost of compilation and, at the same time, improve query performance.

References

[1] Sameer Agarwal, Davies Liu, and Reynold Xin. *Apache Spark as a Compiler: Joining a Billion Rows per Second on a Laptop.* Accessed on 08.06.2022. May 2016. URL: https://databricks.com/blog/2016/05/23/apache-spark-as-a-compiler-joining-a-billion-rows-per-second-on-a-laptop.html.

[2] Yanif Ahmad and Christoph Koch. "DBToaster: A SQL compiler for high-performance delta processing in main-memory databases". In: *Proceedings of the VLDB Endowment* 2.2 (Aug. 2009). DOI: 10.14778/1687553.1687592.

[3] Anastassia Ailamaki et al. *DBMSs On A Modern Processor: Where Does Time Go?* Sept. 1999. URL: https://www.vldb.org/conf/1999/P28.pdf (visited on 06/01/2022).

[4] Peter Boncz, Thomas Neumann, and Orri Erling. "TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark". In: *Performance Characterization and Benchmarking* (2014). DOI: 10.1007/978-3-319-04936-6_5. (Visited on 05/17/2022).

[5] Dennis Butterstein and Torsten Grust. *Precision Performance Surgery for PostgreSQL.* 2016. URL: http://www.vldb.org/pvldb/vol9/p1517-butterstein.pdf.

[6] Jack Chen et al. *The MemSQL Query Optimizer: A modern optimizer for real-time analytics in a distributed database.* Sept. 2016. URL: https://www.vldb.org/pvldb/vol9/p1401-chen.pdf.

[7] Shun Yan Cheung. *CS554 - Advanced Database Systems Syllabus and Progress.* Apr. 2020. URL: https://www.mathcs.emory.edu/~cheung/Courses/554/Syllabus/syl.html.

[8] *Clang 15.0.0git documentation.* Accessed on 20.06.2022. URL: https://clang.llvm.org/docs/UsersManual.html.

[9] T. Cramer et al. "Compiling Java Just In Time". In: *IEEE Micro* 17.3 (1997). DOI: 10.1109/40.591653.

[10] Cristian Diaconu et al. "Hekaton: SQL Server's Memory-Optimized OLTP Engine". In: *ACM International Conference on Management of Data 2013.* June 2013. URL: https://www.microsoft.com/en-us/research/publication/hekaton-sql-servers-memory-optimized-oltp-engine/.

[11] Sangeeta Sen; Moumita Ghosh; Animesh Dutta; Biswanath Dutta. *Hypergraph based query optimization.* Jan. 2015. URL: https://ieeexplore.ieee.org/document/7218100.

[12] Wenchen Fan, Herman van Hövell, and MaryAnn Xue. *Adaptive Query Execution: Speeding Up Spark SQL at Runtime*. Accessed on 10.06.2022. Mar. 2020. URL: `https://databricks.com/blog/2020/05/29/adaptive-query-execution-speeding-up-spark-sql-at-runtime.html`.

[13] Craig Freedman, Erik Ismert, and Per-Ake Larson. *Compilation in the Microsoft SQL Server Hekaton Engine*. 2014.

[14] Andres Freund. *Postgres JIT Source Code*. May 2022. URL: `https://github.com/postgres/postgres/blob/master/src/backend/jit/README`.

[15] Andres Freund. *WIP: Faster Expression Processing and tuple deforming (including JIT)*. Dec. 2016. URL: `https://www.postgresql.org/message-id/20161206034955.bh33paeralxbtluv@alap3.anarazel.de`.

[16] Andres Freund, Peter Geoghegan, and David Rowley. *Postgres JIT: Source code README*. Accessed on 10.05.2022. URL: `https://github.com/postgres/postgres/blob/master/src/backend/jit/README`.

[17] Henning Funke and Jens Teubner. *Low-Latency Compilation of SQL Qeries to Machine Code*. July 2021.

[18] Goetz Graefe. *Volcano-An Extensible and Parallel Query Evaluation System*. Feb. 1994.

[19] Pouria Hadjibagheri. "Cascade of doom: JIT, and how a Postgres update led to 70% failure on a critical national service". In: (Nov. 2021). Accessed on 05.06.2022. DOI: `https://dev.to/xenatisch/cascade-of-doom-jit-and-how-a-postgres-update-led-to-70-failure-on-a-critical-national-service-3f2a`.

[20] Anders Hallem Iversen and Sveinung Øverland. *Compiling expressions in MySQL*. 2021.

[21] Timo Kersten, Viktor Leis, and Thomas Neumann. "Tidy Tuples and Flying Start: Fast compilation and fast execution of relational queries in Umbra". In: *The VLDB Journal* 30 (June 2021). DOI: `10.1007/s00778-020-00643-4`.

[22] Yannis Klonatos et al. "Building efficient query engines in a high-level language". In: *Proceedings of the VLDB Endowment* 7 (June 2014). DOI: `10.14778/2732951.2732959`.

[23] Andre Kohn, Viktor Leis, and Thomas Neumann. "Adaptive Execution of Compiled Queries". In: *2018 IEEE 34th International Conference on Data Engineering (ICDE)* (Apr. 2018).

[24] Konstantinos Krikellas, Stratis D. Viglas, and Marcelo Cintra. *Generating code for holistic query evaluation*. 2010. URL: `https://15721.courses.cs.cmu.edu/spring2016/papers/krikellas-icde2010.pdf`.

[25] Viktor Leis et al. "How good are query optimizers, really?" In: *Proceedings of the VLDB Endowment* 9 (2015). DOI: `10.14778/2850583.2850594`.

[26] *LLVM Language Reference Manual*. Accessed on 12.06.2022. URL: `https://llvm.org/docs/LangRef.html`.

[27] *LLVM Sponsors*. Accessed on 05.05.2022. URL: `https://foundation.llvm.org/docs/sponsors/`.

[28] *LLVM Users.* Accessed on 12.06.2022. URL: https://llvm.org/Users.html.

[29] *LLVM's Analysis and Transform Passes.* Apr. 2022. URL: https://llvm.org/docs/Passes.html#introduction.

[30] Dmitry Melnik et al. *JIT-Compiling SQL Queries in PostgreSQL Using LLVM.* May 2017. URL: https://beta.pgcon.org/2017/schedule/attachments/467_PGCon%202017-05-26%2015-00%20ISPRAS%20Dynamic%20Compilation%20of%20SQL%20Queries%20in%20PostgreSQL%20Using%20LLVM%20JIT.pdf.

[31] Prashanth Menon et al. *Permutable Compiled Queries: Dynamically Adapting Compiled Queries without Recompiling.* Oct. 2020. URL: https://db.cs.cmu.edu/papers/2020/p101-menon.pdf.

[32] *My First Language Frontend with LLVM Tutorial.* Apr. 2022. URL: https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/index.html.

[33] *MySQL Source Code.* Accessed in 01.05.2022. URL: https://github.com/mysql/mysql-server.

[34] *MySQL Standards Compliance.* Accessed on 01.06.2022. URL: https://dev.mysql.com/doc/refman/8.0/en/compatibility.html.

[35] Thomas Neumann. *Efficiently Compiling Efficient Query Plans for Modern Hardware.* 2011. DOI: 10.14778/2002938.2002940. URL: https://www.vldb.org/pvldb/vol4/p539-neumann.pdf.

[36] Thomas Neumann and Michael Freitag. *Umbra: A disk-based system with in-memory performance.* Jan. 2020. URL: http://cidrdb.org/cidr2020/papers/p29-neumann-cidr20.pdf.

[37] *ORC Design and Implementation.* Accessed on 24.04.2022. URL: https://llvm.org/docs/ORCv2.html.

[38] Sasha Pachev. *Understanding MySQL Internals.* O'Reilly Media, Inc., Apr. 2007. ISBN: 9780596009571.

[39] Norvald H Ryeng. *Refactoring Query Processing in MySQL.* https://www.youtube.com/watch?v=u7J0invbMxc&t=38s&ab_channel=CMUDatabaseGroup. Accessed on 01.06.2022. Nov. 2020.

[40] Norvald H. Ryeng. private communication. June 2022.

[41] Juliusz Sompolski. *Just-in-time Compilation in Vectorized Query Execution.* Aug. 2011. URL: https://homepages.cwi.nl/~boncz/msc/2011-JuliuszSompolski.pdf.

[42] Richard M. Stallman. *GNU Compiler Collection Internals.* Free Software Foundation, Inc., May 2004.

[43] K. V. N. Sunitha. *Compiler construction [electronic resource] / K.V.N. Sunitha.* eng. 1st edition. Pearson, 2013. ISBN: 93-325-2459-9.

[44] Ruby Y. Tahboub, Gre´gory M. Essertel, and Tiark Rompf. *How to Architect a Query Compiler, Revisited.* 2018. URL: https://www.cs.purdue.edu/homes/rompf/papers/tahboub-sigmod18.pdf.

[45] *TPC-H Standard Specification.* Feb. 2021. URL: https://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v3.0.0.pdf.

[46] Mihai Varga. *Just-in-time compilation in MonetDB with Weld.* July 2018.

[47] Stratis D. Viglas, Gavin Bierman, and Fabian Nagel. *Processing Declarative Queries Through Generating Imperative Code in Managed Runtimes.* May 2014. DOI: `10.1109/icde.2017.234`.

[48] Skye Wanderman-Milne. *Building a Modern Database Using LLVM.* Nov. 2013. URL: `https://llvm.org/devmtg/2013-11/slides/Wanderman-Milne-Cloudera.pdf`.

[49] Skye Wanderman-Milne and Nong Li. *Runtime Code Generation in Cloudera Impala.* 2014.

[50] *When to JIT?* Aug. 2021. URL: `https://www.postgresql.org/docs/11/jit-decision.html`.

Appendices

Appendix A

Source code of MySQL JIT implementation

The source code of our JIT implementation described in Chapter 3 can be found in two ways. A repository containing the source code can be found at the following URL: `https://github.com/Andorr/mysql-server-jit`. It can also be found in the form of a *git patch* in the file *jitimpl.patch*, added as an attachment to this thesis.

Appendix B

Source code of experiments

The source code of the conducted experiments can be found in the attachments, in which we have provided the following Python and SQL files:

- `main.py` - The main script for running the experiments. Handles the connection to the MySQL server, runs the benchmarks, and performs all client-side time measurements.
- `utils.py` - Utility file for extracting, visualizing, and transforming measurements.
- `gen_query.py` - A script to generate large queries that were used in *the expression experiment.*
- `convert.py` - Script to convert the unsupported columns of the raw TPC-H data into integers.
- `tables.py` - Utility file that contain metadata information about TPC-H tables.
- `tests.py` - Script to verify that the benchmark results and `EXPLAIN ANALYZE` output is consistent between runs and execution modes.
- `queries_int` - The converted TPC-H queries that were used in *the TPC-H experiment.*
- `queries_where` - The queries generated that were used in *the expression experiment.*