Master's thesis

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

Jonas Brunvoll Larsson

Plan caching in MySQL

Master's thesis in MSIT
Supervisor: Norvald H. Ryeng
June 2023

NTNU
Norwegian University of
Science and Technology

Jonas Brunvoll Larsson

Plan caching in MySQL

Master's thesis in MSIT
Supervisor: Norvald H. Ryeng
June 2023

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

Abstract

The relational database management systems of today processes enormous amounts of data. Digital trends lead to increasing amounts of data that must be processed at an ever-increasing pace. As upgrading hardware is not a reliable solution in the long run, new smart solutions must continuously be introduced to make relational database management systems capable of meeting the performance requirements of today and for the future.

Plan caching is an approach used to increase the efficiency of a relational database management system. Normally when executing a query, a new query execution plan is created for every execution. When plan caching is applied, query execution plans are being stored in a cache to enable reuse when matching queries being executed at a later point in time. By using plan caching, both the execution time and the resource consumption of the relational database management system is reduced, as well as reusing plans can result in more predictable runtime, given that the plans are not outdated.

This thesis looks at the possibility of implementing plan caching as a feature to enhance the overall performance of MySQL. The thesis starts by closely examining existing solutions and related theory, before a plan cache prototype is implemented into MySQL. Further, two experiments are carried out to investigate how the performance of MySQL is influenced by the implementation when enforcing both *exact match policy* and *inexact match policy*. The findings show that plan caching results in both more predictable and faster executing queries when enforcing *exact match policy*, while plan caching when enforcing *inexact match policy* is too unpredictable to be used.

Sammendrag

Dagens relasjonsdatabasesystemer behandler enorme mengder med data. Digitale trender fører til at mengden med data som må behandles vokser i en stadig raskere hastighet. Siden oppgradering av maskinvare ikke er en pålitelig løsning på sikt, er det å komme opp med nye smarte løsninger for å gjøre relasjonsdatabasesystemer i stand til å møte dagens og fremtidige ytelseskrav.

Planbufring er en metode som brukes til å bedre ytelsen til et relasjonsdatabasesystem. Normalt når en spørring kjøres, opprettes en ny kjøreplan for hver gang spørringen gjennomføres. Når planbufring brukes, lagres kjøreplaner i en hurtigbuffer for å muliggjøre gjenbruk ved gjentgende spørringer. Gjennom bruk av planbufring reduseres både utførelsestiden og ressursforbruket til relasjonsdatabasesystemet, samt at gjenbruk av kjøreplaner resulterer i mer forutsigbar kjøretid, gitt at planene ikke er utdaterte.

Denne masteroppgaven ser på muligheten for å implementere planbufring i MySQL med mål om å bedre den generelle ytelsen. Masteroppgaven starter ved å undersøke eksisterende løsninger og relatert teori, før en planbuffer-prototype implementeres i MySQL. Videre blir to eksperimenter utført for å undersøke hvordan ytelsen til MySQL påvirkes av den implementerte prototypen når både *eksakt samsvarende gjenbruk* og *unøyaktig samsvarende gjenbruk* håndheves. Funnene viser at planbufring resulterer i både mer forutsigbare og raskere kjøretider ved bruk av *eksakt samsvarende gjenbruk*, mens *unøyaktig samsvarende gjenbruk* gir for uforutsigbare kjøretider til å kunne benyttes.

Preface

This report presents the work done in a master thesis from the Faculty of Information Technology and Electrical Engineering (IE) and the Department of Computer Science (IDI), at the Norwegian University of Science and Technology (NTNU). The presented work has been developed in collaboration with MySQL throughout the spring of 2023, and is based on my specialization project that was carried out in the autumn of 2022.

First, I would like to thank my supervisor *Norvald H. Ryeng* for his great support and highly appreciated guidance in the field of query optimization throughout the project. I would also thank *Knut Anders Hatlen* for his guidance in the MySQL repository and C++ development in general. Without both of you, this thesis would not have been the same.

Trondheim, 12.06.2023
Jonas Brunvoll Larsson

Contents

List of Tables

List of Figures

Listings

Chapter 1

Introduction

Nowadays, implementing digital systems to streamline manual work task is a well-known measure in most industries. New digital systems are daily being introduced, ranging anywhere from pioneering tech tools to social media platforms and games. For the majority of these systems, a database management system - (DBMS) serves as the core component to deal with data management. However, the ever-increasing number of digital services being introduced yearly have resulted in an explosion in the amount of data being generated[1]. As a consequence, the DBMS's of today must continuously introduce new improvements to be able to handle the challenges of processing increasing amounts of data, while still meet the performance requirements of today.

As of May 2023, the DB-Engines ranking lists MySQL as the second most popular DBMS in the world, only surpassed by Oracle[2]. MySQL is a relational database management system - (RDMBS), meaning that MySQL organizes data in tables and accesses data by passing Structured Query Language - (SQL) queries. Serving as the core component in several of the most frequently accessed applications of today such as Twitter, Netflix and Uber, MySQL has proven to be a both reliable and high performing system[3]. Like all other commercial RDBMS's, MySQL must continuously come up with new improvements not only to remain to be the RDBMS of choice for existing users, but also to attract new users by being a solid forward-looking system that can serve upcoming applications.

Plan caching is an approach used to reduce both the query execution time and the resource consumption of a RDBMS when executing the same query frequently. After the initial execution, the query execution plan is stored in a cache to enable it to be reused when the same query is executed at a later point in time. By storing the query execution plan, the RDBMS increase it efficiency by avoid having the need to repeat the process of developing a new executable query plan for every query execution. While various variants of plan caching have been implemented in commercial RDBMS's such as PostgreSQL[4], SQL Server[5] and Oracle[6], the feature have

yet to be implemented in MySQL.

This thesis investigates the possibility of implementing plan caching as a feature to enhance the overall performance of MySQL. The research presented in this thesis is based on the research that was carried out during the specialization project, in the autumn of 2022, where the main purpose was to investigate whether both plan caching and plan pinning could be adequate features to implement to increase the performance of MySQL. The findings of both approaches showed promising results that were worth further exploration. As the research was of the broader variety, no real implementation of either plan caching or plan pinning was carried out in the specialization project. The research presented in this thesis attempts to cover this shortcoming by implementing a plan cache prototype as a feature in MySQL and investigate how the implementation affects the performance.

1.1 Motivation

As stated in the previous section, the motivation behind implementing plan caching is to enhance the overall performance of MySQL. More specifically, this mean enhancing the performance of the query processor, the component that carries out both the processing and execution of incoming queries[7]. A query processor without a plan cache must create and optimize a new executable plan for every query execution, while a query processor with a plan cache only creates a new executable plan if none of the plans stored in cache are suitable for the query that is currently being executed.

Plan caching is not only appropriate for improving the performance of a RDBMS by reducing the total execution time of a query. Plan caching is also an approach to achieve predictable query response time, which makes it easier to develop applications that depends on response from a RDBMS within a given time frame[8]. As explained in more detail in section 2.1.5, there is a bit of randomness in how efficient a query execution plan ends up. In most cases, the query processor develops good-enough query execution plans which results in sufficient performance. However, outliers that lead to poor query execution do occasionally occur, which can cause major problems for applications that depends on predictable response times. After identifying whether a query execution plan is good-enough, plan caching is an approach to ensure that this plan will continuously be executed, without having to risk that a less efficient plan is to be executed.

1.2 Scope

The focus of this thesis is to investigate whether it is feasible to implement plan caching to enhance the performance of the query processor, as the source code of MySQL is structured

today. The thesis also investigates different policies to determine whether the query being executed can reuse any of the query execution plans stored in the plan cache. In this thesis, these policies are referred to as *cache match policies*.

To carry out experiments, a plan cache prototype is initially created and implemented as a feature into MySQL. The experiments involve executing various queries from the *Join Order Benchmark* against a MySQL database containing the *IMDB - dataset*. During the experiments, the time-consumption of the various steps within the query processor are logged, before the findings are later analyzed to compare the performance of the queries.

To extend the research, the experiments are also conducted by using both *exact match policy* and *inexact match policy*, to investigate whether there is reason to believe that a query execution plan can be reused by the same prepared statement, but with different sets of parameters. If that is feasible without harming the performance of the query processor too much, there might be an opportunity to utilize the plan cache more often even and ultimately making MySQL even more efficient. To better answer these areas of interest, the thesis aims to answer the following research questions:

- **RQ1:** Does the findings indicate that implementing plan caching is a measure that would enhance the overall performance of the MySQL query processor?
- **RQ2:** How is the performance of the MySQL query processor affected when using a plan cache enforcing *inexact match policy* compared to when enforcing *exact match policy*?

1.3 Structure of thesis

This thesis is divided into seven chapters. After the initial introduction in chapter 1, chapter 2 describes the fundamental knowledge about query optimization. Chapter 3 defines plan caching as a concept, and presents related work conducted by both commercial companies and academic communities. Chapter 4 describes the implementation of plan cache prototype in MySQL. Chapter 5 describes the experimental setup and the implementations of the experiments. In chapter 6, the experimental findings are presented and discussed, before chapter 7 finally summarizes the thesis and answers the research questions.

Chapter 2

Background

This chapter contains background information that the theory presented in chapter 3 is based on. Section 2.1 provides a detailed description of the query processor, the component in any RDBMS responsible for both processing and query execution. Section 2.2 describes prepared statements, as this feature serves an important part in the plan cach implementation presented in chapter 4.

2.1 The query processor

The query processor is the component responsible for both the processing and the execution of query statements in a RDBMS[9]. The workflow of the query processor is commonly referred to as the query processing pipeline, as the workflow is divided into several steps, each carried out by a designated internal component in a sequential order. As shown in figure 2.1, every component except the initial parser, processes the output of the previous component in the pipeline. The structure of the pipeline somewhat varies depending on the RDBMS, but essentially all RDBMS's end up solving the same tasks. A textbook example of how the components are organized is explained in the sections 2.1.2 to 2.1.6. Before further explaining these components, section 2.1.1 describes the query tree, the data structure that is passed between the components within the query processing pipeline.

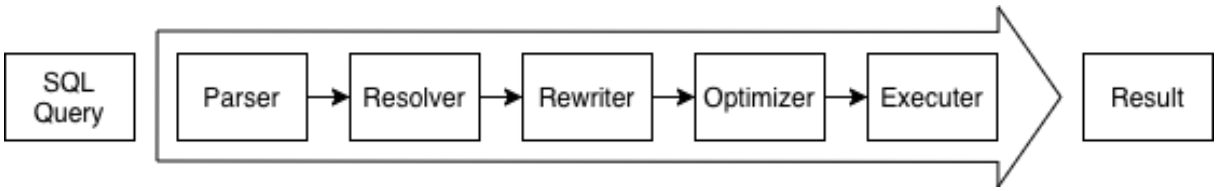


Figure 2.1: An illustration of the query processing pipeline. The query processing is initiated by the query parser and ends by returning the results back to the client after completing the query execution.

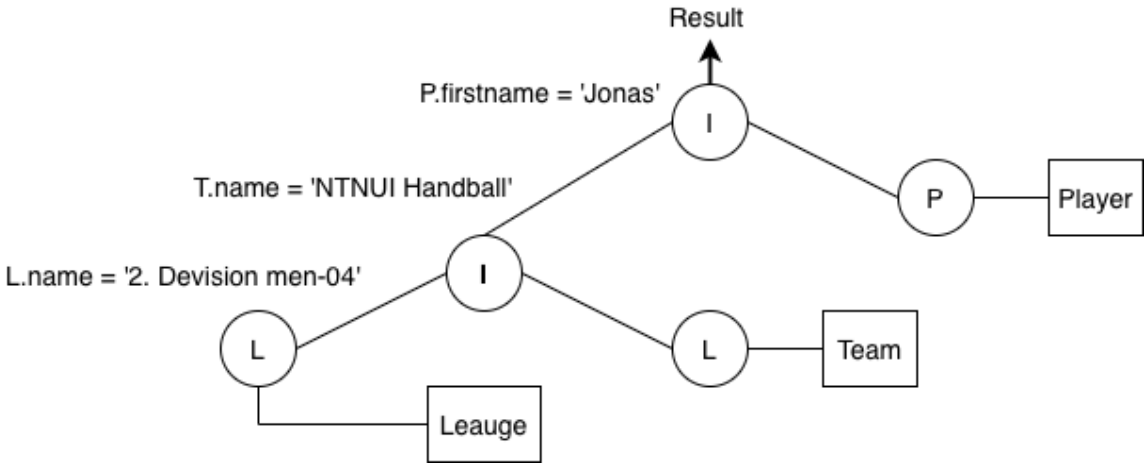
2.1.1 The query tree

The query tree is an in-memory representation of the query that enables the components within the query processing pipeline to interpret and further process the query. The data structure presents the query as expression tree. Each node represents an algebraic operation, which together forms one extended relational algebraic expression[10]. The leaf nodes of the tree represent the parameter conditions provided to the query, while the internal nodes represent algebraic operations that must be carried out in order to produce some sort of result based on the conditional terms provided by the two corresponding leaf nodes.

The query tree is created by splitting a query statement into units named blocks. Each block represents a single SELECT-FROM-WHERE statement, including the optional GROUP-BY and HAVING clauses[10]. As shown in figure 2.2, the query tree of single SELECT-FROM-WHERE statement is represented through one block. In cases where the query statement consists of a given number of nested SELECT-FROM-WHERE statements, the query tree is represented of equally many blocks.

```
1 SELECT P.firstname, T.name, L.name
2 FROM Player P, Team T, Leauge L
3 WHERE P.firstname = 'Jonas'
4 AND T.name = 'NTNUI Handball'
5 AND L.name = '2. Devision men-04';
```

(a) A simple query statement.



(b) The query tree of query (a).

Figure 2.2: An illustration of a query (a) and its corresponding query tree (b). The leaf nodes (L), represents the conditions of the query, while the internal nodes (I) represent the algebraic operations based on the provided conditions computes some type of result[10].

2.1.2 The query parser

The initial step of the query processing pipeline is to translate the SQL query statement into a query tree that the forthcoming components in the query processing can interpret and process[9][11]. This task is carried out by the query parser, through the combination of lexing and parsing operations.

Lexing involves splitting the query statement into distinct keywords. Each keyword is tagged with a token to represents the specific keyword type. As an example, a table name is tagged as an identifier, while a larger-or-equal-then condition is tagged as an operator.

Parsing involves organizing the stream of tokens that was found through the lexing operation into a query tree. To initialize this process, the parser requests a token from the lexer. The lexer responds by sending a single token which the parser places into the query tree. This process is repeated until there are none tokens left to place.

After the lexing and the parsing of the entire query statement is completed, the query parser ends by verifying that the syntax of the query statement is correctly written and remove unneeded comments. In case an error is encountered, the query process is cancelled, and an error message is sent back to the client.

2.1.3 The query resolver

The next step in the query processing pipeline is to ensure that the identifiers represented in the query tree actually exist in the database[9]. This task is carried out by the query resolver. After verifying that the identifiers exists, metadata is added to the query tree as proof.

Further, the query resolver verifies that the client is authorized to execute the query[9]. If all the identifiers exists and the client is authorized to execute the query, the resolver locks the identifier entries to ensure that the tables and columns referenced in the query statement will exist in the same state for the remainder of the query execution. Otherwise, if the client is not authorized to execute the query, the query process is cancelled, and an error message is sent back to the client.

2.1.4 The query rewriter

The query rewriter simplifies and normalizes the query tree without changing the semantics[9]. In the same way as relational algebraic equations, the query tree can be rewritten to a simpler expression that requires less computation. This action is done as preparatory work to reduce the workload of the query optimizer in the next phase. By performing arithmetic evaluation and heuristics, simplifications such as sub-query flattening and constant folding are carried out until a logical optimal version of the query tree is finalized.

2.1.5 The query optimizer

Query optimization is the process of converting the logical plan developed by the query rewriter into a reasonably efficient query execution plan[9][10]. The query optimizer carries this process out by combining the two optimization approaches *heuristic optimization* and *cost-based optimization* to determine which rules and strategies to include in the query execution plan.

Heuristic optimization

Heuristic optimization is the process of implementing optimization rules that are known from experience to reduce the query execution time, ultimately increasing the performance of the query processor[10]. There exists several of these rules of thumb such as; *"Push selection as far down as possible"* or *"Push projection as far down as possible"*. Common to all the rules is that they aim to reduce the number of rows that are analyzed during the query execution to a minimum at the earliest possible stage.

The selection of rules to add to the query execution plan during heuristic optimization are done without analyzing the data associated with the query. Since no time is spent on data analyzation, heuristics optimization is a quick process. This also results in no query execution plans tailored for one specific query, but query execution plans that perform fairly well for most queries.

Cost-based optimization

Cost-based optimization is the process of selecting the strategy with the one single lowest estimated total cost out of all the available strategies[10]. The query optimizer accomplishes this by generating various strategies for carrying out the operations from one specific query tree, based on statistics provided form the database. Then, after all the strategies have been generated, the cost of each strategy is evaluated before the strategy estimated to be the least time-consuming is selected.

The strategy evaluation of cost-based optimization is both time-consuming and resource demanding. Therefore, the query optimizer usually terminates the evaluation after a certain time limit is reached, or if a strategy believed to be reasonably efficient is found. If allowed to continue the optimization process long enough, cost-based optimization should in theory find the strategy for the provided query tree. In practice, this is rather difficult to guarantee, as the statistics cost-based optimization is based on are usually not completely accurate.

2.1.6 The query executor

The query executor executes the internal operations in the query tree, ending up with a result ready to be sent back to the client[9][10]. Initially, the operations at the bottom of the query tree are executed, as these operations only takes parameter conditions from leaf nodes as input, thus making them executable from the beginning. After completed, the operations at the next level takes the output from the former operations as input.

This traversing process in the query tree continues from bottom to top until all operations have been executed. The query execution ends after the root operation terminates and the result is sent back to the client. When executing a nested query, the innermost query will first complete the execution before the result will be provided as an input condition in the second innermost query. This order of execution will continue until the outermost query is completed.

2.2 Prepared statements

Prepared statements is a feature used to repeatedly execute the same query statement with high efficency[12]. Initially, a prepared statement is created by completing the steps up until the query rewriter. Then, the query tree is stored as a query template, which will remain accessible for the reminder of the client-server session, or until the prepared statements is deallocated on purpose.

When a prepared statement is called to be executed, the query template is fetched and the query processing continues from where it ended when preparing. As shown on line 4 in listing 2.1, a prepared statement is executed by the command EXECUTE followed by the name of the statement. Any unspecified value ('?') in the query template must be specified before execution. This is done by adding the command USING, followed by a parameter to set the value.

```
1  Set @name = "Jonas";
2  Set @sql = "Select * From person p Where p.name = ?";
3  Prepare stmt From @sql;
4  Execute stmt Using @name;
```

Listing 2.1: A simple example of how a prepared statement is prepared and executed in MySQL. The parameter @name is used to set the unspecified value of p.name during the execution of stmt.

Chapter 3

Theory and related work

This chapter describes the theory behind the logic in the plan cache implementation presented in chapter 4. Section 3.1 starts by defining plan caching as a concept, before two separate groups of policies that affect the behavior of a plan cache named *Cache match policies* and *Cache replacement policies* are described in section 3.1.1 and section 3.1.2. Various existing plan cache implementations from other RDBMS's are presented in section 3.2. Lastly, section 3.3 describes some alternative approaches to plan caching to influence the decision-making of the query optimizer.

3.1 Plan caching

Plan caching, occasionally also referred to as query plan caching, is about reducing the total time and resource consumption of the query processor when processing frequently executed query statements. This is achieved by storing optimized query execution plans in a cache, enabling later matching queries to reuse the plans when executed. By doing so, the query processor avoid having to recompute a new query execution plan for every query execution[8][13].

There are two main reasons why caching plans may be performance enhancing for a RDBMS. First, plan caching ensures that the query processor does not have to create and optimize a new query execution plan for every query after the initial execution[8][13]. That being said, this is only performance enhancing when a match is found within reasonably short time. If no match is found, the plan cache effectively becomes a dead end, only consuming needless processing time and resources before the query processing is resumed. Also, a found match is only performance enhancing if the time spent on searching after and feeding the query executor with it, is less time-consuming than creating a new execution plan.

Second, by reusing the same query execution plan, the response time when executing a query should become more predictable compared to the alternative[14]. Occasionally, the

query optimizer ends up creating a different query execution plan than the ones created in previous executions, which can result in either equally fast, faster or slower response time when executing the query. In, theory, this uncertainty is eliminated when reusing a cached plan, as it should result in equally fast response time every time executed. In reality, this assumption can only be made given the condition that both the data that being accessed and the corresponding statistics in the RDMBS remain unchanged in between executions.

Figure 3.1, shows the interaction between a plan cache and the query processor when processing a query statement. A query statement is confirmed as executable after successfully passing the query resolver in the query processing pipeline. Before continuing the query processing, a match operation enforcing some type of *cache match policy* is carried out to check whether an eligible query execution plan already exists in the plan cache. If an eligible plan do exist, the plan is sent directly to the query executor, bypassing both the query rewriter and the query optimizer. Otherwise, if no matching query execution plan is found, the query processor continue processing the query as usual. After the optimization step is completed, the query execution plan is stored in the plan cache. If there is no space left to the new plan in the plan cache, a replacement operation enforcing some type of *cache replacement policy* is carried out to make space.

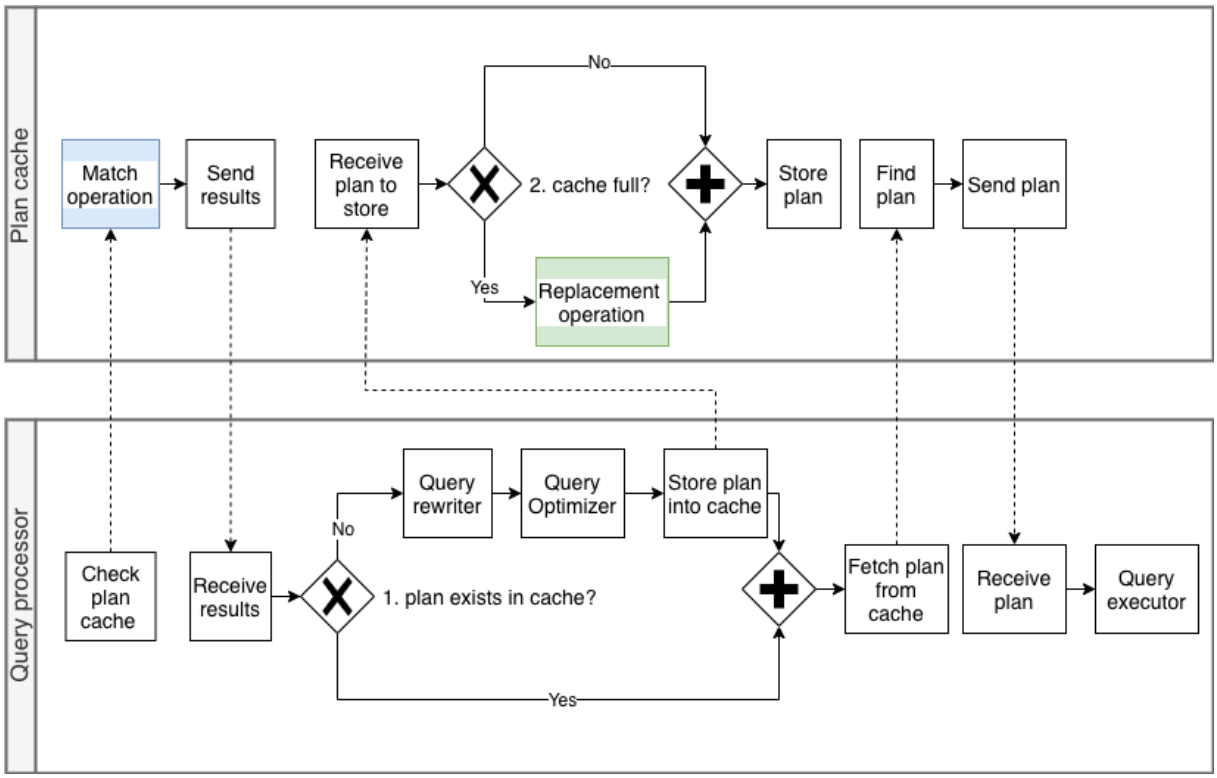


Figure 3.1: A simplified illustration of the major events in a plan cache when processing a query. The first major event is the match operation, where reusable plans are detected. The second major event is the replacement operation, which is carried out to free up space if needed to store a new query execution plan on the cache.

3.1.1 Cache match policies

A cache match policy is a similarity algorithm used to determine whether an incoming query statement can reuse any of the query execution plans stored in the cache[5][8][15]. Among others, estimating similarity by comparing the hash value between two query statements is a commonly used approach[5][6]. While this approach is both fast and uncomplicated to carry out, it is limiting as only exact matches are accepted.

Other more sophisticated solutions have also been researched over the years. In 2002, Gosh et al[15] presented PLASTIC - (PLAN Selection Through Incremental Clustering), a framework where query execution plans were organized in clusters. In PLASTIC, similarly between query statements is estimated based on query features, such as number of table join operations and presents of certain tables and query attributes to mention a few.

Parametric Plan Caching (PPC), is another cluster-based plan cache framework that was proposed by Aluç et al[8] in 2012. PPC map each query statement into a plan space based on the provided parameter values. Then, the similarity between two query statements is estimated by measuring the density between the query representations in the plan space. In addition, PPC makes use of locality - sensitive hashing (LSH) to avoid having to measure similarity between every possible combination of the query instances represented in the plan space, and instead focus on only measuring the similarity between query instances that are likely to be similar.

While both the mentioned frameworks have shown promising results, neither of the large commercial RDBMS's have yet to implement a cluster-based plan cache implementation. At the time of writing, some form of hash based plan cache implementation is still the most common approach.

3.1.2 Cache replacement policies

A replacement policy is an algorithm used to determine which plan out of all query execution plans stored on the cache that must be evicted to make space for a new plan. Arguable, the most commonly used replacement logic in any type of cache is the *Least Recently Used - (LRU)*, where the last accessed object in the cache is thrown out, thus ensuring that the most frequently accessed objects stays in the cache[16]. Other simpler replacement policies are the *First In First Out - (FIFO)* and the *Last In First Out - (LIFO)* algorithms, where the object that has been in the plan cache for the longest and the shortest time is thrown out.

3.2 Existing implementations

Plan caching has for a long time been accepted as an adequate approach to increase the efficiency RDBMS systems[5]. As a result, several commercial RDBMS's have introduced various implementations. This section describes a small selection of these implementations.

3.2.1 Microsoft SQL Server

The RDBMS Microsoft SQL Server have implemented a plan cache capable of caching both ad hoc queries and prepared statements. The first proficient version of the plan cache was introduced in the release of SQL Server 7.0 in 1998[17]. This version organized optimized query execution plans in key-value pairs with the hashed query statement as the key value[5]. Match operations were carried out by comparing hash values, which made the implementation both fast and uncomplicated.

The major drawback of this implementation was that because of the hash comparison, the match operations became case-sensitive which effectively resulted in that only exact matches were accepted. As a result, it was only possible to store one query execution plan per query statement. While not a problem when caching ad hoc queries, this made caching prepared statements problematic as the values of the parameters following the prepared statements were not taken into account when carrying out math operations. This is problem is illustrated in listings 3.1, where the hash value of *q1* and *q2* end up equal, although the set values of the parameters are unequal.

```
1  Set @name = "Jonas";
2  Set @q1 = "Select * From person p Where p.name = ?";
3  plan_cache_hash_function(@q1) = akcW...0ps
4
5  Set @name = "Oda";
6  Set @q2 = "Select * From person p Where p.name = ?";
7  plan_cach_hash_function(@q2) = akcW...0ps
```

Listing 3.1: A simplified example to illustrate the challenge of carrying out match operations by hash string comparison only. The hash function end up producing the the same hash value when provided both @q1 and @q2 as input, althrough the parameter @name is set to different values.

To accommodate the problem of caching prepared statements, SQL Server introduced *Parameter Sensitive Plan optimization - (PSP)* in SQL Server 2022[18]. PSP is as feature that makes it possible to store up to three query execution plans of a single prepared statement, each optimized to different set of parameters. Whenever SQL detects a prepared statement that can potentially

reuse one of the stored query execution plans, each plan is closer examined with regard to the parameter sets. If an exact match between the parameter sets are found, the query execution plan is reused. Otherwise, a new query execution plan is created. However, the new plan will not automatically replace either of the plans already stored in the cache unless specified.

3.2.2 PostgreSQL

The open-source RDBMS PostgreSQL have implemented a simple, yet interesting plan cache for caching prepared statements[4]. The PostgreSQL server distinguishes between two types of query execution plans. The first type, the custom execution plan, is re-optimized for every execution based on the parameters provided for the specific execution. The second type, the generic execution plan, is an optimized execution plan fetched from a cache that is the same across all executions, regardless of the provided parameters.

What makes the plan cache of PostgreSQL interesting is how the server determine whether a generic execution plan should be created and added to plan cache. By default, the server will issue custom execution plans for the first five executions of a prepared statement. After the fifth execution is completed, a generic execution plan is created by analyzing the previously issued plans and their corresponding parameters. Then, if the estimated cost of the generic execution plan is less than the average execution cost of the custom execution plans, the generic execution plan is stored in the plan cache.

3.2.3 Oracle

The RDBMS Oracle have implemented an intelligent plan cache capable of caching complete query execution plans of both ad hoc and prepared statements[19]. Likewise as SQL Server, after the initial parsing, the query statement is hashed and compared to the queries already stored in the plan cache. If a match is found, the query execution plan is reused. Otherwise, the query optimizer creates a new query execution plan and stores it in the plan cache. Also, the implementation have the ability to analyze parameter values when executing prepared statements. This process is termed *bind variable peeking*, which enables the query optimizer to read the parameter values following the prepared statements as if they were provided in an ordinary ad hoc statement with literal values. This makes it possible to determine whether a prepared statement should be executed using a cached query execution plan.

Arguable one the most interesting trait of Oracles plan cache implementation is the concept of *cursor sharing* which enables caching across multiple client-server sessions[19]. A cursor is best described as a pointer that points to a specific SQL area. A SQL area is classified as either private or shared, and is used to store some type of query related information. A private SQL area is used to store information about a parsed SQL statement and other

related session - specific information, and is pointed at by a cursor on the client side. A shared SQL area contains the parse three and the query execution plan of a query statement, and is pointed at by a cursor stored in private SQL areas. Cursors stored various private SQL areas are able to point at the same shared SQL area, thus enabling reuse of the same query execution plan by multiple separated client-server sessions. Figure 3.2, displays a simplified illustration of how the plan caching is achieved through cursor sharing.

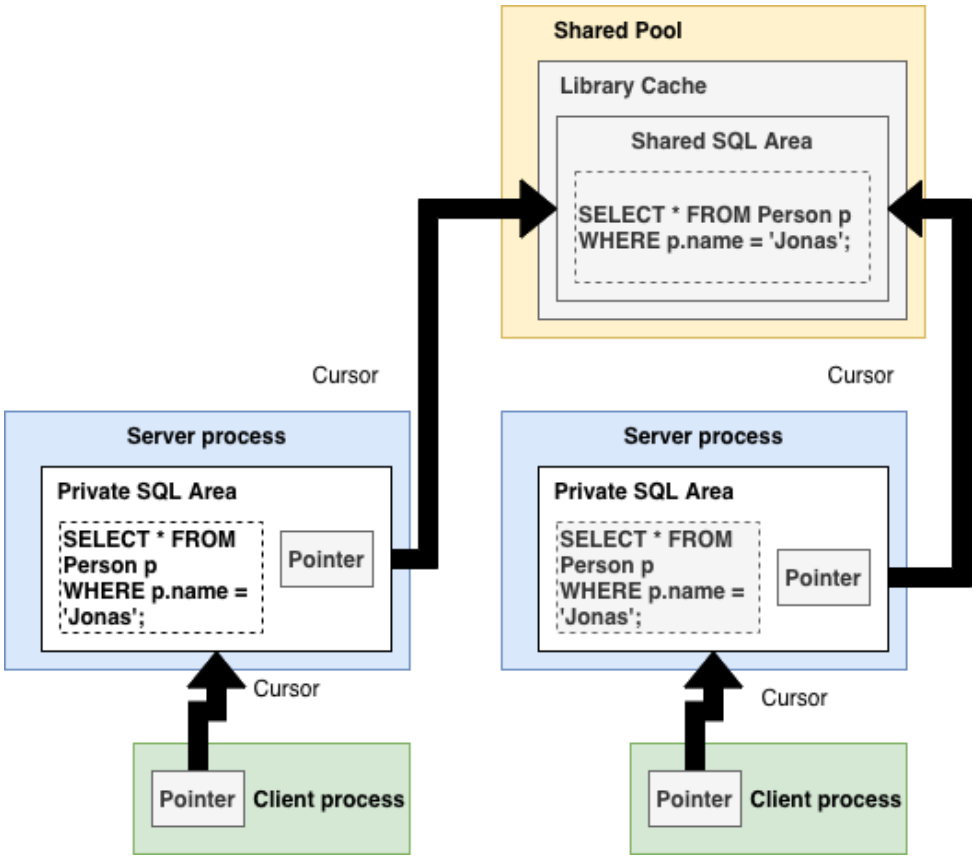


Figure 3.2: An simplified illustration of cursor sharing in Oracles plan cache. Both cursors from the two client processes each point at a separate private SQL area in server process. The cursors from the two separate server processes point at the same shared SQL area, enabling both client-server sessions to use the same query execution plan when execution the shown query statement[19]

3.3 Related optimization approaches

Plan caching is one of several approaches that are used to influence the choice making of the query optimizer. This section describes two related approaches named *plan hinting* and *optimizer hints*. While both approaches may resemble plan caching, there are some distinct differences that need to be addressed.

3.3.1 Optimizer hints

Optimizer hints are optional statements that can be included in a query statement to override the decision-making of the query optimizer[20]. When an optimizer hint is included in a query statement, the hint forces the query optimizer to either choose or avoid a specific strategy when creating a query execution plan[12]. Various types of optimizer hints exists, all overriding different parts of the decision-making of the query optimizer. Lastly, multiple optimizer hints can be included in a query statement. If enough hints are included, the decisions that the optimizer are able to make become so limited that the query optimizer is effectively forced to create a predetermined plan.

Unlike plan caching, optimizer hints does not result in a complete bypass of the query optimizer. The query optimizer must still create a query execution plan, but can potentially spend significantly less time than when optimizing both ad hoc and prepared statements due to the limited number of possible decisions to make. How much less time is saved by using optimizer hints depends entirely on how limited the number of strategies the optimizer can choose from is.

3.3.2 Plan pinning

Plan pinning involves forcing the query executor to execute a predetermined query execution plan, overriding the query optimizers decision-making completely[14]. How plan pinning is implemented varies from different RDBMS's. In some systems, after being recognized, query statements are being supplemented with optimizer hints to force the creation of a specific query execution plan[21]. Other systems store optimized query execution plans and forces execution whenever a corresponding query statement is recognized, completely bypassing the query optimizer.

Plan pinning may appear similar to plan caching in certain situations, but there are some important differences. A plan cache is continuously being updated, whether it is being filed up with new query execution plans or replacing old plans with new ones. Plan pinning prevents the query executor from creating new query execution plans entirely, and is most often applied to prevent the query optimizer from creating poor plans[14]. This is useful when executing query statements that for some reason frequently end up with poor query execution plans. Plan pinning is also applied to avoid plan regression, which leads to a sudden reduction in the performance of the query processor. For most applications, a sudden increase in response time from the RDBMS is much more critical than a significant speedup, and is therefore the most crucial outcome to avoid[14].

Chapter 4

Implementation

This chapter describes the implementation of a plan cache prototype with functionality based on the theory presented in chapter 3. Section 4.1 describes some of the most significant functionalities from the MYSQL-SERVER repository, as well as explaining how these functionalities were reused in the implementation. Thereafter, section 4.2 describes the plan cache implementation and specify design choices.

4.1 Reuse of existing functionality

The plan cache implementation is partly developed by reusing parts of the already existing functionality from MySQL's open-source repository, MYSQL-SERVER. Having a brief understanding of this functionality is a prerequisite to be able to understand the workflow and the design choices of the implementation that are later described in section 4.2. Therefore, this section is included to highlight the most significant reused functionality and explain how these functions are used in the implementation.

MEM_ROOT

The MEM_ROOT is a data structure that makes it easier to keep track of memory allocations[22]. In the *mysql-server* repository, a MEM_ROOT - object is initialized at the start of every client-server session and is used to store data properties that should not be deleted, unless specified, when preforming cleanup operation at the end of query executions. In addition to being fast, the MEM_ROOT - object automatically release all memory allocations when destroyed.

In the plan cache implementation, a new MEM_ROOT - object is created to store each query execution plan and all corresponding data properties that are generated during the optimization. This makes both match - and replacement - operations fairly straight forward to carry out, as stored query execution plans are clearly separated in dedicated MEM_ROOT - objects.

```
36      public:
37          PLAN_ROOT(std::vector<prepared_statement_parameter> _parameters)
        {
38              parameters = _parameters;
39              set_timestamp_created();
40              set_timestamp_last_accessed();
41          }
42
43          MEM_ROOT mem_root;
```

Listing 4.1: Displaying of the mem_root attribute inside the plan_root class, fetched from sql_plan_root.h. The attribute is public to make the query execution plan accessible.

Thread handler - (THD)

The thread handler - (THD) data structure is a central part of the logic in the MYSQL_SERVER. Each time a new session between a client and the MYSQL-SERVER is established, a new THD - object is created and dedicated to the session[23]. This THD - object is not destroyed until the session between the client and the MYSQL-SERVER is terminated. The THD - object points to the MEM_ROOT - object of the session. This makes it possible to store data within the THD - object throughout the session.

In the plan cache implementation, the ability to store information inside the THD object is exploited by adding an object of the class PLAN_CACHE as a local attribute inside the THD class. The PLAN_CACHE is the main data structure in the plan cache implementation. By doing so, a new PLAN_CACHE instance is created inside the THD object whenever a new session between client and the MYSQL-SERVER is established.

```
928 class THD : public MDL_context_owner,
929             public Query_arena,
930             public Open_tables_state {
931  public:
932   /**
933     Controlled memory stats for this session.
934     This member is the first in THD,
935     to initialize Thd_mem_cnt() before allocating more memory.
936   */
937   Thd_mem_cnt m_mem_cnt;
938   PLAN_CACHE plan_cache;
```

Listing 4.2: Displaying the empty plan_cache attribute inside the thd class, fetched from sql_class.h

Swap_mem_root_guard

The SWAP_MEM_ROOT_GUARD is a data structure used to temporarily swap out the MEM_ROOT - object pointed at by the THD with another specified MEM_ROOT - object[24]. This is operation is conducted by calling the function MEM_ROOT_GUARD, which updates the THD - object to point to a specified MEM_ROOT - object, while temporarily holding the pointer to the original MEM_ROOT - object. The SWAP_MEM_ROOT_GUARD destructor is automatically called at the end of the scope, making the THD point back to the original MEM_ROOT - object.

In the plan cache implementation, shown in 4.3, the SWAP_MEM_ROOT_GUARD swaps out the main MEM_ROOT - object with the MEM_ROOT - object dedicated to the query execution plan before the optimization takes place. After the execution, the main MEM_ROOT - object is swapped back, before the fixed cleanup operations are carried out. This avoids the query execution plan and other corresponding data properties from being deleted.

```
775     // Switch to plan roots mem_guard.
776     PLAN_ROOT* ptr_plan_root = thd->plan_cache.get_active_plan_root();
777     Swap_mem_root_guard mem_root_guard{thd, &ptr_plan_root->mem_root};
778
779     if (!thd->plan_cache.plan_root_is_optimized()) {
780       if (unit->optimize(thd, /*materialize_destination=*/nullptr,
781                          /*create_iterators=*/true, /*finalize_access_paths
      =*/true)){
782         return true;
```

Listing 4.3: Displaying the swap_mem_root_guard being used to make the thd point at the mem_root dedicated to the plan_root before optimizing the query execution plan. Fetched from sql_select.cc

Access_path

The ACCESS_PATH is the data structure in the MYSQL-SERVER that represents the internal nodes of the query tree[25]. As described in section 2.1.1, each ACCESS_PATH - object contains information about how the algebraic operation between the two child nodes is to be solved. The algebraic operation itself is not conducted by the ACCESS_PATH - object, but by an associated ITERATOR - object, which takes the information from the ACCESS_PATH - object as input.

In the plan cache implementation, all ACCESS_PATH - objects and the corresponding iterators are stored inside the MEM_ROOT of the dedicated PLAN_ROOT - object. When a stored plan is being reused, the executor is simply provided with the root node of the ACCEES_PATH - TREE without having to re-compute all the algebraic operations or having to regenerate matching iterators.

4.2 Implementation of the plan cache

The plan cache implementation is an extension of the MYSQL-SERVER VERSION 8.0. The centerpiece of the implementation is the PLAN_CACHE, which is responsible for maintaining an overview over all the cached query execution plans, and to carry out both matching- and replacement - operations whenever necessary. The behavior of PLAN_CACHE is determined by a set of properties, which are further described in section 4.2.1.

The second component of the implementation is the PLAN_ROOT, which serves as a container-object dedicated to store one query execution plan and additional related data properties. All PLAN_ROOT - objects are organized by the PLAN_CACHE - object of the session. Together, the PLAN_CACHE and the PLAN_ROOT make up the entire implementation, including the auxiliary logic to interact with the MYSQL-SERVER. The details and the workflow of both the classes are explained in section 4.2.2 and section 4.2.3.

A few simplifications are made to avoid making the implementation too advance. First, the implementation is only able to store query execution plans of prepared statements. This is done to simplify the match operations. As explained in section 2.2, after the initial preparation is completed, a prepared statement is stored as a template and remains accessible for the remainder of the session. To maintain an overview over the query execution plans the are stored in the cache, the implementation simply maintains a list of the pointers of the prepared statements.

Second, the implementation is session based. This means that a new instance of a plan cache is created whenever a new session between a client and the MySQL server is established. Likewise, a created plan cache instance is destroyed whenever the session terminated.

4.2.1 Plan cache properties

The behavior of the plan cache implementation is determined by a set of parameters that are being provided when a prepared statement enters the plan cache. As shown in listings 4.4, in addition to the pointer of the prepared statement and its corresponding parameters, three additional parameters are provided. These parameters represent the cache match policy (*match logics*), the cache replacement policy (*replacement logics*) and the number of query execution plans *entries* each prepared are able to store in the plan cache. By combining these properties in different combinations, various behaviors are specified for the plan cache.

```
1002    /* match_logic, entries, replacement_logic, pointer preared statement,
          corresponding parameters*/
1003    thd->plan_cache.enter_plan_cache("INEXACT_MATCH", "ONE_ENTRY", "LRU",
         stmt, parameters_prepared_statement);
```

Listing 4.4: Displaying the entry point of the plan cache. The parameters passed into the function enter_plan_cache determines the behavior of the plan cache by specifying the match logic, the number of entries and the replacement logic that should be enforced.

Match logic

As described in section 3.1.1, the cache match policy is used to determine whether two queries are matching. In the implementation, the match policy is used to specify what type of match algorithm that shall be used when carrying out matching operations. Table 4.1, provides an overview of the various match algorithms the plan cache implementation can be specified to use.

Parameter name	*Short description*
EXACT_MATCH	Match if the parameter sets are identical.
INEXACT_MATCH	Match as long as the numer of parameters and the parameter types in both sets are equal.

Table 4.1: Overview of the various match logics the plan_cache can be specified to enforce.

Replacement logic

As described in section 3.1.2, the cache replacement policy determines which query execution plan to remove in order to free up space for a new plan. In the implementation, the replacement policy is used to specify what type of replacement algorithm that shall be used to determine which PLAN_ROOT - object to remove to make space to a new PLAN_ROOT - object. Table 4.2, provides an overview of the various replacement algorithms the plan cache implementation can be specified to use.

Entries

The *entries* parameter specifies whether the plan cache can store multiple PLAN_ROOT - objects containing a query execution plan corresponding to the same prepared statement. Allowing multiple versions of query execution plans, enables the possibility of storing multiply query execution plans optimized for different sets of parameters. Table 4.3, provides an overview of the various entry options the plan cache implementation can be specified to enforce.

Parameter name	*Short description*
LRU	Enforces the *Least Recently Used* - logic.
FIFO	Enforces the *First In First Out* - logic.
LIFO	Enforces the *Last In First Out* - logic.
WORST_MATCH	The WORST_MATCH - logic rank all the PLAN_ROOT - objects stored in the plan cache based in their similarity between the parameter sets of the incoming prepared statement. The worst ranked PLAN_ROOT - object is thrown out of the cache.

Table 4.2: Overview of the various replacement logics the plan cache can be specified to enforce.

Parameter name	*Short description*
ONE_ENTRY	Strictly one version of each prepared statement is allowed in the cache.
N_ENTRIES	'N' number of versions of each prepared statement are allowed to be stored in the cache. The value 'N' is set during the initialization of the plan cache, at the beginning of the session.

Table 4.3: Overview of the various entry options the plan cache can be specified to enforce.

4.2.2 PLAN_ROOT

The PLAN_ROOT is the data structure where query execution plans are stored to enable reuse. Each PLAN_ROOT - object is dedicated to a single prepared statement. The lifespan of a PLAN_ROOT - object goes as following: A new PLAN_ROOT - object is created by the PLAN_CACHE to store a new query execution plan in the cache. During the rewrite step and the optimization step, the attributes of the PLAN_ROOT - object are being populated by the ACCESS_PATH - tree, timestamps to keep tack of when the PLAN_ROOT - object was both created and last accessed, and other data necessary properties that were generated during the optimization step. Also, it must be pointed out that all the attributes are allocated inside the MEM_ROOT - object of the PLAN_ROOT - object, making it possible to access the attributes in forthcoming query executions.

As described in section 3.1, when a match is later found, both the rewrite step and the optimization step of the query processing pipeline are bypassed. Instead of preparing a new query execution plan, the query executor is feed with a pointer pointing at the PLAN_ROOT - object containing the MEM_ROOT - object where the query execution plan is stored. Lastly, if a prepared statement at some point is re-prepared, the corresponding PLAN_ROOT - object is destroyed and a new PLAN_ROOT - object is created and dedicated to the new prepared statement.

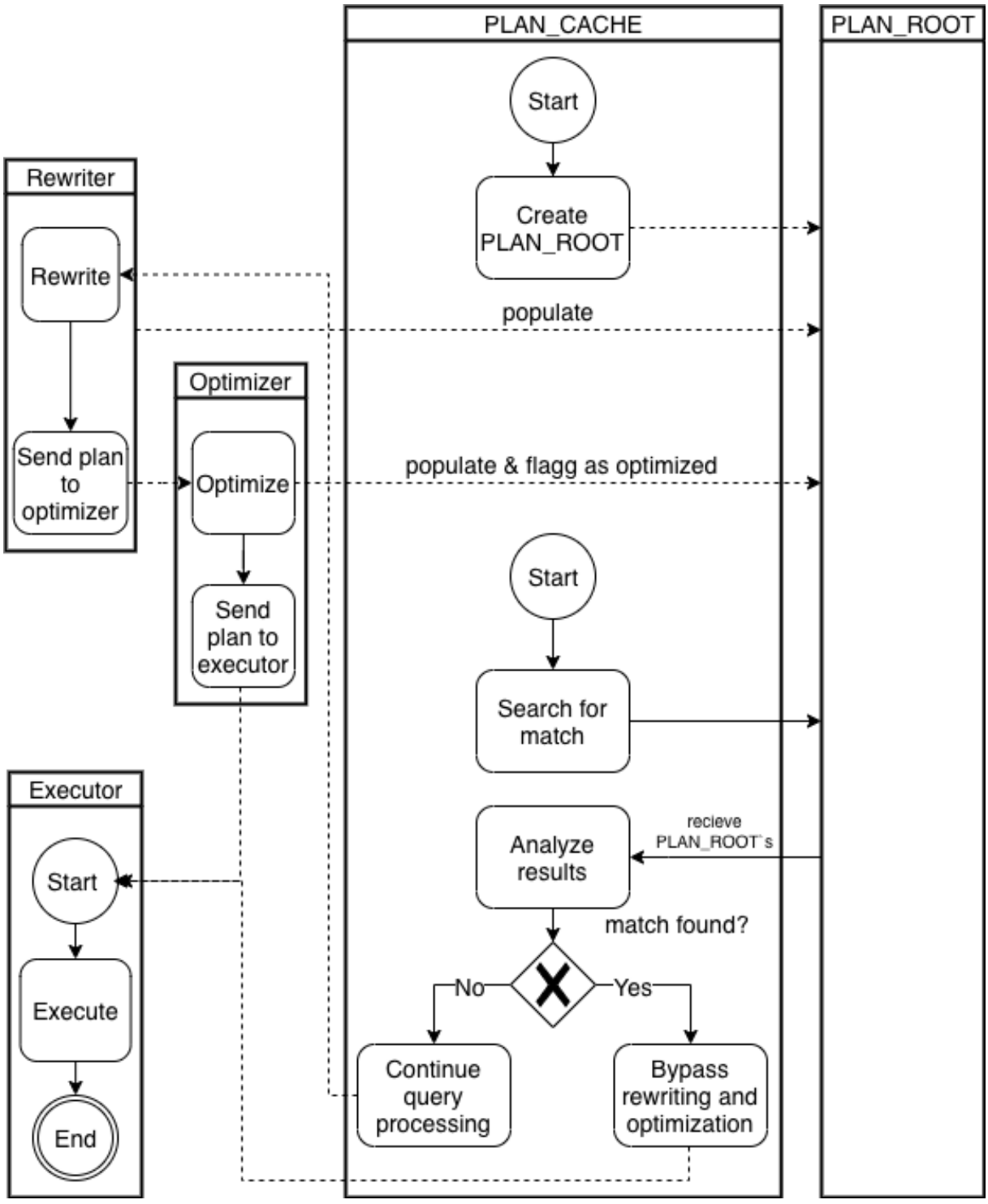


Figure 4.1: BPMN diagram illustrating the workflow when a new plan_root - object is created and populated, before later being used to bypass the optimization process.

4.2.3 PLAN_CACHE

The PLAN_CACHE is the main data structure in the implementation and where the majority of the logic is handled. The three main tasks of the class is to maintain an overview of the query execution plans stored in the cache, to conduct match operations between incoming prepared statements and the prepared statements stored in the cache, and lastly to conduct replacement operations in order to free space for new query execution plans.

In short, the workflow of the PLAN_CACHE goes as following: The execution of a prepared statement is initiated. After completing the preparation phase, the pointer of the prepared statement along with the predetermined properties of the plan cache is sent to the PLAN_CACHE. The PLAN_CACHE start by examine whether the prepared statement being pointed at is already stored in the PLAN_CACHE - object.

If it turns out that the pointer is not stored in the cache, the PLAN_CACHE continues by examine whether are any available spots in the plan cache. A spot is available of the number of cached query execution plans is not greater than the predetermined attribute GLOBAL_LIMIT. If a spot is available, the PLAN_CACHE created a new PLAN_ROOT - object dedicated to the prepared statement. Otherwise, if there are no available spots, the PLAN_CACHE first carries out a replacement operation, before a PLAN_ROOT - object is created and dedicated to the prepared statement.

If it turns out that the pointer is already stored in the cache, the examination of the corresponding query execution plans continuous according to the specified logic. At any point if a match is found, the process is terminated as there is no need to continue the examination. If no match is found due to too different parameter sets, a replacement operation is carried out and a new PLAN_ROOT - object is dedicated to the prepared statement.

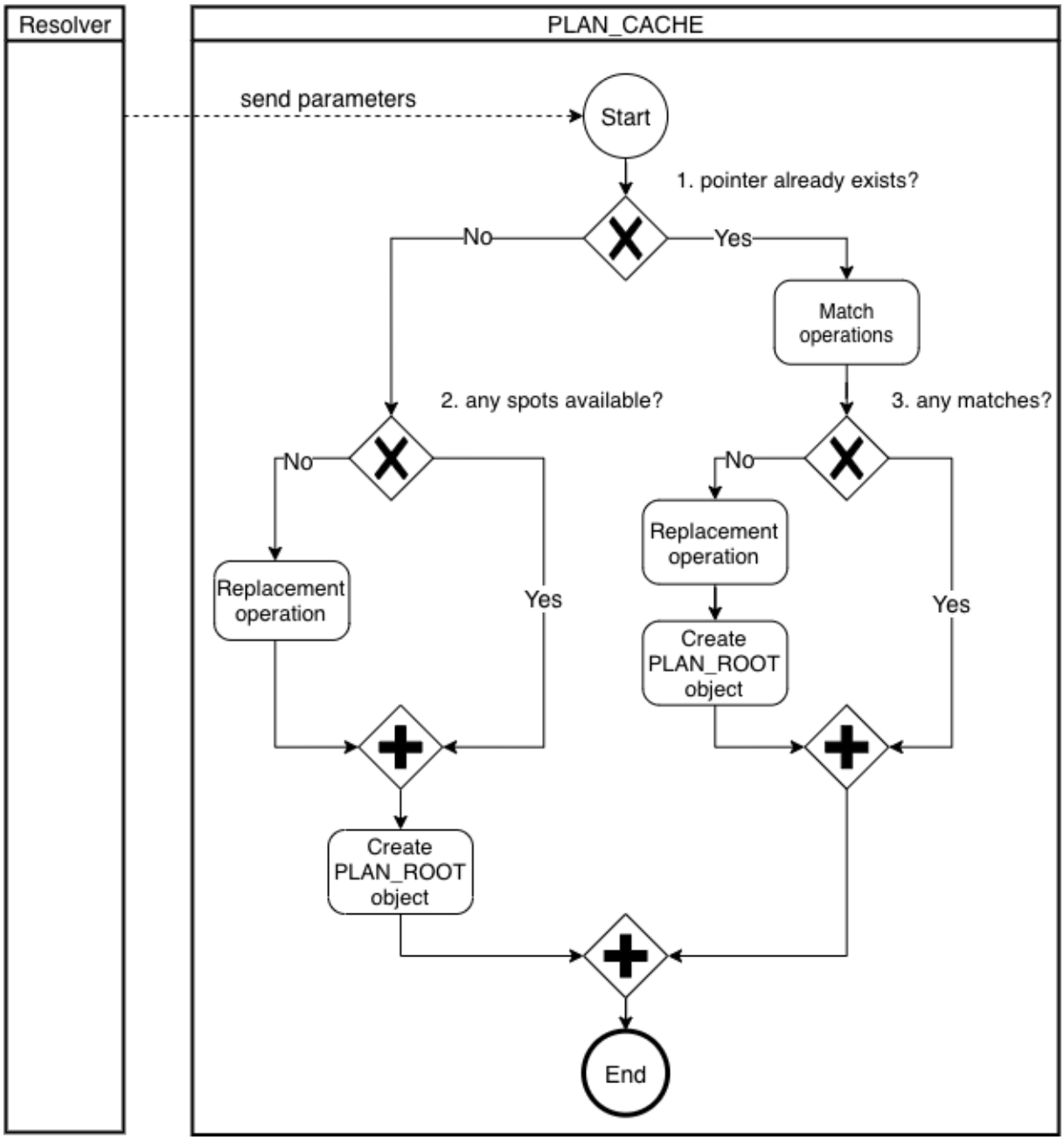


Figure 4.2: BPMN diagram illustrating the workflow of the plan_cache when searching after a plan_root - object containing a matching query execution to reuse. If none match is found, a new plan_root - object is created. If space must be freed before storing a new plan_root - object on the cache, a replacement operation is carried out.

Chapter 5

Experiments

Three experiments have been conducted to evaluate the performance of the plan cache implementation. The experiments all involved executing various prepared statements against a MySQL database to obtain insight about how the performance of the query processor is affected by the plan cache prototype. In addition, ad hoc versions of the query statements were also executed to enable performance comparing.

The chapter starts by describing the experimental setup in section 5.1. The benchmark used when carrying out the experiments is justified in section 5.1.1 before both environmental and hardware details during of the experiments are described in section 5.1.2 and section 5.1.3. Lastly, the implementation of each experiment is described in section 5.2.

5.1 Experimental setup

This section describes the preparatory work that was carried out before the experiments were conducted. The choice of benchmark used during the experiments is justified and both environmental details and hardware details of the laptop used to carry out the experiments are described.

5.1.1 Benchmark

The *Join Order Benchmark - (JOB)* is the benchmark that was used when carrying out the experiments. JOB is based on the *Internet Movie Data Base - (IMDB)* dataset and contains an enormous amount of movie-related information, distributed between 21 tables containing anywhere between 4 rows to 36 million rows[26]. The benchmark consists of a total of 113 individual SELECT - query statements, divided into 33 different query structures, each structure having between 2-6 versions with slightly different parameters[26].

With an average of 8 join operations, the queries forces the query optimizer to spend time on finding an appropriate join-strategy. This trait made JOB the ideal benchmark for evaluating the impact on the query processor when caching query executing plans, as selection of the join-strategies is arguable the most time-consuming operation performed by the query optimizer. For this reason, JOB was selected instead of other benchmarks such as Sysbench[27], TPC-C[28] and TPC-H[29], as these benchmark were more adequate when evaluating time spent on query execution, rather than time spent on query optimization.

The JOB benchmark is written using PostgreSQL syntax. Although the SQL - statements of MySQL and PostgreSQL are fairly similar, the syntax does not always translate. When preparing for the experiments, it was discovered that the syntax if query *10a* and *10b* did not translate and resulted in errors. Rather than rewriting the queries to MySQL syntax, they were not included in the experiments described in section 5.2.

5.1.2 Environment

InnoDB is the default storage engine of MySQL[30]. To fit the entire JOB dataset inside the main memory during the experiments, the InnoDB configuration *innodb-buffer-pool-size* was set to 12GB. This measure was done to avoid potentially having to carry out slow read operations during the experiments towards disk, which would have had an impact on the measured query execution time. All other default configurations of MySQL remained unchanged during the experiments.

5.1.3 Hardware and OS

All experiments were conducted on Dell laptop with the listed specifications:

- **Hardware Model**: Dell Inc. Latitude 7420.
- **Memory**: 32GB.
- **Processor**: 11th Gen Intel® Core™ i7-1185G7 @ 3.00GHz × 8.
- **Graphics**: Mesa Intel® Xe Graphics (TGL GT2).
- **Disk Capacity**: 512GB.
- **OS Name**: Ubuntu 22.04-2. LTS.
- **OS Type**: 64-bit.
- **Gnome version**: 42.5.

The laptop was restarted before each experiment. Also, all non-relevant applications were terminated to reduce the number of applications sharing the hardware to a minimum.

5.2 Implementation of experiments

The experiments involve testing the performance of the query processor when executing both cached and ad hoc queries with the plan cache implementation enforcing either *exact match policy* or *inexact match policy*. The experiments started by carrying out a preparatory examination of all the 111 included queries from the JOB benchmark to detect the queries that are the most interesting to include in the subsequent experiments. After the initial examination followed two experiments, on for each of the aforementioned match policies.

The following experiments were conducted in the following order:

- **Experiment 1:** Benchmark query selection.
- **Experiment 2:** Plan cache enforcing exact match policy.
- **Experiment 3:** Plan cache enforcing inexact match policy.

To evaluate the performance, the time-consumption when processing the query of both the query optimizer and the query executor were timed and written to a local log file. All write operations to the log file was carried out after each the query execution had completed to avoid interfering with the timing.

All experiments where carried out by initiating a bash script which automatically feed a running instance of the MYSQL-SERVER with a .sql file, containing all query statements corresponding to the specific experiment. Afterwards, bash and python scripts were used to format and analyze the data, as well as presenting the findings.

5.2.1 Benchmark query selection

The initial experiment involved executing the ad hoc - version of all 111 queries against a running MYSQL-SERVER. The goal of the experiment was to pinpointed which queries that were most likely to provoke a significant change in the performance of the query processor when caching queries, and would thus be the most interesting queries to include in the forthcoming experiments.

After warming up the memory, all the queries were executed in a sequential order against the *mysql-server*. To avoid that any query with unusual query processing distribution should shape the results too much, the execution process was repeated 10 times before the average durations of both the query optimizer and the query executor was computed.

5.2.2 Plan cache - exact match policy

The second experiment involved executing cached queries the MYSQL-SERVER when the plan cache implementation enforced *exact match policy*. The goal of this experiment was to investi-

gate how the performance of the query processor was affected when caching queries with exact matching parameter sets. Afterwards, the findings were compared against the performance of the query processor when executing ad hoc versions of the same query statements.

Before conducting this experiment, the 8 queries that spent proportionally the most time on optimization were detected and written as prepared statements. Further, both the ad hoc versions and the prepared statement version of all the queries were copied into a separate .sql filed, before each file was executed against the MYSQL-SERVER in order. For the same reason as explained in 5.2.1, the experiment was repeated 100 times before the experiment was completed.

5.2.3 Plan cache - inexact match policy

The third experiment was carried out in the same manner as the experiment 2, except that the plan cache implementation was set now to enforce *inexact match policy*. The goal of this experiment was to investigate how the performance of the query processor was effected when caching queries with only inexact matching parameter sets.

Before conducting the experiment, 1 out of the 8 query statements from experiment 5.2.2 was selected and rewritten to a prepared statement with 5 unspecified parameter values. During the initial 10 executions, the parameters were set to the original values. The next ten executions, 1 out of the parameters was set to a value different to the original value to investigate how the query processor would be affected. This experiment was repeated for 5 separate parameter values. Like the experiment 2, the experiment was then repeated 100 times to ensure a good foundation to discuss results.

Chapter 6

Results and discussion

This chapter evaluates the findings from the experiments that were described in chapter 5. First, section 6.1 presents and discusses the queries that were found to effect the performance of the query processor the most when caching query execution plans. Then, section 6.2 presents and discusses the performance of the query processor when caching queries provided with exact matching parameters. Lastly, section 6.3 presents and discusses the performance of the query processor when caching a query provided with inexact matching parameters.

6.1 Benchmark query selection

The goal of this experiment was to detect the queries that were most likely to noticeably affect the performance of the query processor when caching query execution plans, and would therefore be the most interesting queries to include in the subsequent experiments. To ensure good readability, only the most interesting results are presented in this section. A complete overview of the results from all 111 query executions are included in appendix B.

6.1.1 Results

The stacked bars in figure 6.1 displays the average time spent by the query optimizer and the query executor when processing query *29b* to query *33c* inclusive. Two clear observations emerges. First, the total execution time varies greatly from query to query. The total execution time of the queries *29b* and *29c*, both takes more than 10 seconds to complete, while the queries *33a*, *33b* and *33c*, takes less than a second to complete. As a result of the large difference in total execution time, some bars in the figure are hardly visible. Second, based on the queries with visible bars, it appears that the query executor consumes more of the total execution time than the query optimizer.

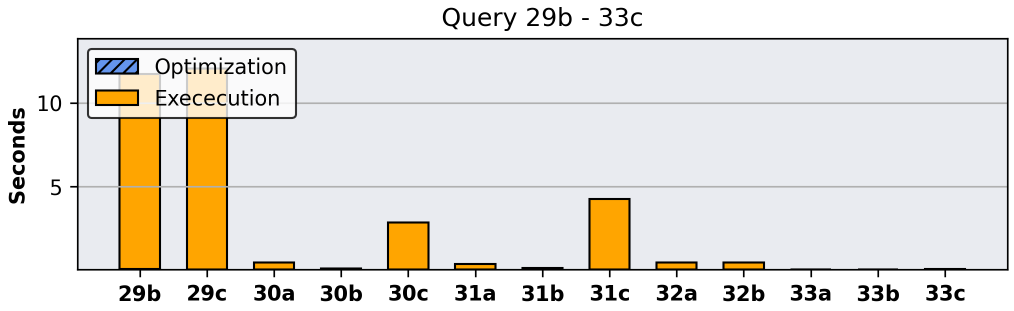


Figure 6.1: The total execution divided between time set to optimization [blue] and time set to execution [orange], when carrying out ad hoc executions of query 29b to query 33c inclusive.

The large differences in total execution time makes it problematic to visualize the results from multiple queries in the same graph, as some bars simply disappears. To accommodate this, figure 6.2 displays the average proportional distribution between the query optimizer and the query executor when processing the queries. Two new observations clearly emerges. First, a small subset of the JOB queries spend a significant amount of the total execution time on query optimization. Second, all the detected queries that spend a significant proportion of the total execution time on query optimization are fast-executing queries. Query *33a*, *33b* and *33c*, are all clear examples of this, as they are barely visible in figure 6.1, and clearly spend a significant proportion of the total execution time on query optimization, as shown in figure 6.2.

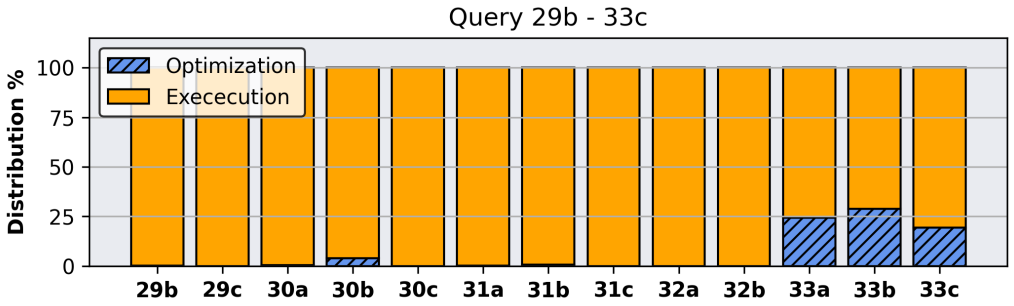


Figure 6.2: The proportional distribution divided between time set to optimization [blue] and time set to execution [orange], when carrying out ad hoc executions of query 29b to query 33c inclusive.

6.1.2 Discussion

As described in section 3.1, a match in the plan cache causes the query processors to bypass the query optimizer. In theory, this means that the time spent on query optimization should be deducted whenever a match is found, ultimately reducing the total execution time. In practice, some time will be spent on overhead, such as carrying out match operations and possible remote operations. Still, it is likely that the queries that make the query processor spend proportionally the most time on optimization, are also the queries that most likely will

noticeably affect the performance of the query processor the most. Figure 6.3 displays the 8 query statements were the query optimizer was found to consume proportionally the most time out of all the 111 queries. These queries are therefore included in the forthcoming experiments.

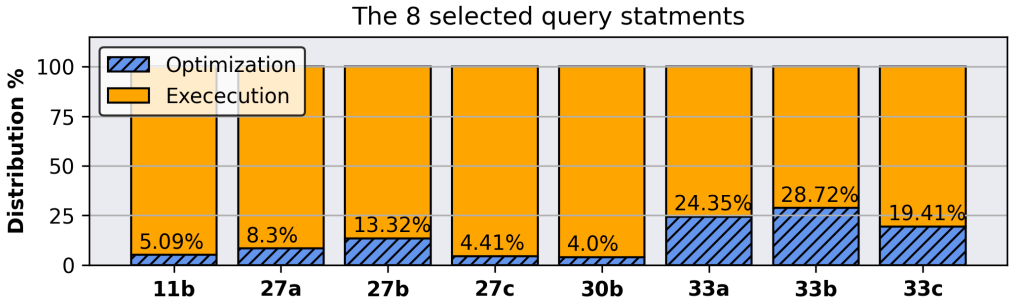


Figure 6.3: The proportional distribution divided between time set to optimization [blue] and time set to execution [orange], of the 8 queries that proportionally spends the most of the total execution time on optimization.

6.2 Plan cache - exact match policy

The goal of this experiment was to investigate how the performance of the query processor was affected when caching queries with exact matching parameters. Likewise as in experiment 1, only the most interesting results are presented. A complete overview of the actual time consumption in milliseconds of both cached and ad hoc executions are attached in appendix C.

6.2.1 Results

Table 6.1 displays the recorded performance of the query processor when comparing the execution of the cached queries and the ad hoc queries. The results show that the average time spent on optimization for all queries when caching the query execution plan is close to 100% faster compared to when executing the ad hoc queries. Further, for all queries, the average execution time was somewhere between 0% and 5% faster when caching the query execution plan. Interestingly, the results shows that some queries, like *11b*, had a max execution time 17.41% faster than the ad hoc executions, while query *33b*, had a far worse execution time 41.15% slower than the ad hoc executions. Lastly, the relative standard derivations in execution time varied less when executing most of the cached queries compared to the ad hoc queries. The only exception to this was query *33b*, were the relative standard derivations in execution time varied 2.56% more when executing the cached query.

Execution comparison between the cached queries and the ad hoc queries					
Query	*Average optimization*	*Min execution*	*Max execution*	*Average execution*	*Relative standard derivation*
11b	-99.48%	0.16%	-17.41%	-4.93%	-47.3%
27a	-99.93%	0.18%	-4.35%	-0.2%	-13.86%
27b	-99.92%	-0.33%	-9.21%	-0.19%	-2.03%
27c	-99.91%	2.4%	1.08%	-0.07%	-2.28%
30b	-99.97%	0.03%	-10.18%	-0.07%	-8.85%
33a	-99.97%	-1.0%	-0.69%	-4.0%	-10.59%
33b	-99.97%	-1.07%	41.15%	-3.19%	2.56%
33c	-99.99%	-1.36%	-7.98%	-2.84%	-3.53%
AVG	-99.89%	-0.12%	-0.95%	-1.94%	-10.73%

Table 6.1: Performance comparisons between execution of cached queries and ad hoc queries. Overall, cached queries appears to execute slightly faster and with less variation in time spent on execution.

6.2.2 Discussion

The observations from table 6.1 are interesting and reveal a lot of information about the current plan cache implementation. First, the short time spent on query optimization indicates that the query execution plans are successfully being fetched from the plan cache. With close to 100% less time spent on optimization on average, caching plans is clearly faster than creating new ones when carried out as in this experiment. The result may be slightly colored by the fact that the cache always contained one matching plan when the experiment was carried out. If the cache had contained more plans, more match operations must have been carried out, which probably would have resulted in a somewhat increased average time spent on optimization than recorded.

The reduced variation in relative standard derivation support that query execution plans are successfully being cached, as reused plans should lead to more predictable execution time. As stated, the only exception with recorded higher relative standard derivation is query *33b*. This is most likely the result of the multiple executions with poor performance such as the recorded max execution of the query.

Interestingly, the average execution time is only slightly reduced when caching and not as much as expected. As shown in figure 6.3, the query optimizer was found to on average consume approximately 24%, 29% and 19% of the total execution time when executing query *33a*, *33b* and *33c*. With these results as the starting point, one would expect a deduction in the total execution time close to these percentages when caching.

There could be several reasons to why this happened. First, there is the possibility that solution for feeding the query executor with cached query execution plans is not sufficient. As described in section 4.2, the actual match operation and the way the query optimizer is fed with a query execution plan, are two separate operations. This means that even if the first performs well, the latter does not necessarily have to. This may also explain why the max execution time of some cached queries are slower than the max execution time when executed as ad hoc queries.

Second, the performance of a cached query is entirely dependent on how good the execution plan the query optimizer made during the first execution. As described in section 2.1.5, there is a bit of randomness in which rules and strategies that end up being included in the query execution plan. Therefore, the recorded execution times may be colored by that the query execution plans created during the initial executions were on average less efficient plans compared to the ones created during the ad hoc executions, and therefore resulted in a poorer execution.

Analysis of the fastest cached queries

As described in section 3.2.2, the plan cache implementation of PostgreSQL attempts to avoid poor execution plans from being stored in the plan cache by delaying creating a generic execution plan. First after at least five executions, a generic plan is created and stored in the cache, given that the plan preforms provably better than the average custom execution plan. Such logic does not exist in the current plan cache implementation, but the concept of only storing provable efficient plans in the plan cache is interesting. A similar feature could potentially be added to improve the plan cache implementation. This section takes a closer look at the possible outcome of experiment 2, if only the query execution plans that resulted in the fastest executions of each query were stored in the cache.

Results

Table 6.2 displays the recorded performance of query processor when only comparing the executing of the fastest cached queries and the ad hoc executions. The results show a significant improvement among all queries. The recorded max execution time and the average execution time have both been reduced for all queries. Also, the found relevant standard derivation shows that the execution time is now even more predictable compared to the ad hoc executions.

Execution comparison between the fastest cached queries and the ad hoc queries					
Query	*Max optimization*	*Min execution*	*Max execution*	*Average execution*	*Relative standard derivation*
11b	-	1.86%	-51.68%	-6.56%	-76.25%
27a	-	1.29%	-40.14%	-5.38%	-76.18%
27b	-	-0.33%	-60.42%	-24.97%	-50.72%
27c	-	4.35%	-42.41%	-4.92%	-82.84%
30b	-	0.03%	-24.75%	-3.81%	-94.6%
33a	-	0.21%	-38.58%	-8.91%	-44.69%
33b	-	-0.2%	-41.51%	-10.4%	-56.93%
33c	-	0.2%	-39.15%	-6.72%	-49.51%
AVG	-	0.93%	-42.33%	-8.96%	-66.46%

Table 6.2: Performance comparisons between the fastest cached queries and the average ad hoc executions. *Max optimization* is not included as the recorded performance include both the initial and the cached executions of the queries.

Discussion

Not surprisingly, the execution time is significantly better results when only comparing the fastest executing cached queries. Arguably more important, the results show that the query execution times varies less. As described in section 3.3.2, predictable execution time is for most applications more important than fast execution. Based on these results, there are several indications that the implementation would benefit from being more careful about which query execution plans to be stored in the cache.

6.3 Plan cache - inexact match policy

The goal of this experiment was to investigate how the performance of the query processor was affected when caching queries provided with inexact matching parameters. As described in section 5.2.3, one out of the 8 queries used in experiment 2 was selected for this experiment. Using a randomizer, query *33c* ended up as the selected for the experiment.

As described in section 5.1.1, the queries from the *Join Order Benchmark* are all used to return data from the enormous *Internet Movie Data Base - (IMDB)* dataset. Exactly what rows of data that is returned when executing 33c is not relevant for this experiment, but how the execution time is affected when the query executor is feed with a query execution plan optimized for different parameter values. Table 6.3, displays the parameter values and provides a short description of how the new parameter values affects search area of query 33c.

Specified parameters in query 33c			
Parameter	*Original value*	*New value*	*Short description*
p0	-	-	No change in parameter values. Added for reference.
p1	!=’[us]’	!=’[de]’	Query 33c must not return content of Danish origin, rather to the original parameters where content of American origin where excluded.
p2	’rating’	’budget’	Query 33c must return content containing the search word ’rating’ instead of the original search word ’budget’.
p3	’tv series’	’movie’	Query 33c must return content of type ’movie’ instead of the original content type ’tv series’.
p4	’3.5’	’6.0’	Query 33c must return content width weighted individual average of all user ratings equal ’6.0’ instead of the original rating ’3.5’.
p5-1	2000	1995	Query 33c must return content where production period start sometime between 1995 and 2010 instead of the original timeline between 2000 and 2010. (Increased search space)
p5-2	2000	2005	Query 33c must return content where production period start sometime between 2005 and 2010 instead of the original timeline between 2000 and 2010. (Reduced search space)

Table 6.3: The parameter values of query 33c that were changed between the executions.

6.3.1 Results

The recorded performance when executing query *33c* with both exact and inexact matching parameters is displayed as milliseconds in figure 6.4 and as percentage difference in figure 6.5. Two clear observations emerges. Most noticeably, when setting the weighted individual average of all user ratings (*p4*), from '3.5' to '6.0', the average query execution time increases from 12.1ms to 23.7ms, approximately a 96% increase in execution time. Further, increasing the search space by setting the start year of the production period (*p5-1*) from 2000 to 1995, resulted in an approximate 16% increase in execution time. When setting reducing the search space by setting the start year to 2005 (*p5-2*), the execution time was reduced with 25%. The execution time was also reduced with approximately 17% when the content type was changed from 'tv series' to 'movie'. Otherwise, the query executed with pretty much the same performance when provided with new values for the parameters *p0*, *p1* and *p2*.

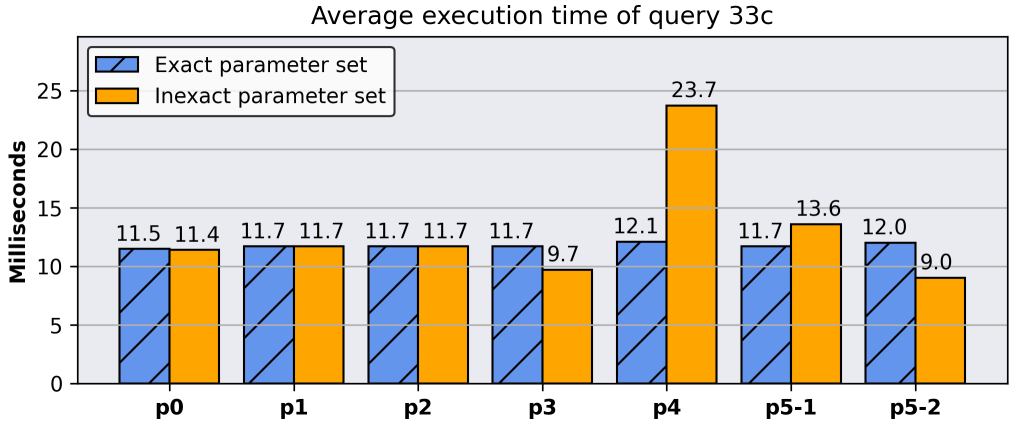


Figure 6.4: Execution time of query 33c when provided with both exact [blue] and inexact [orange] matching parameters. The execution time is measured in milliseconds.

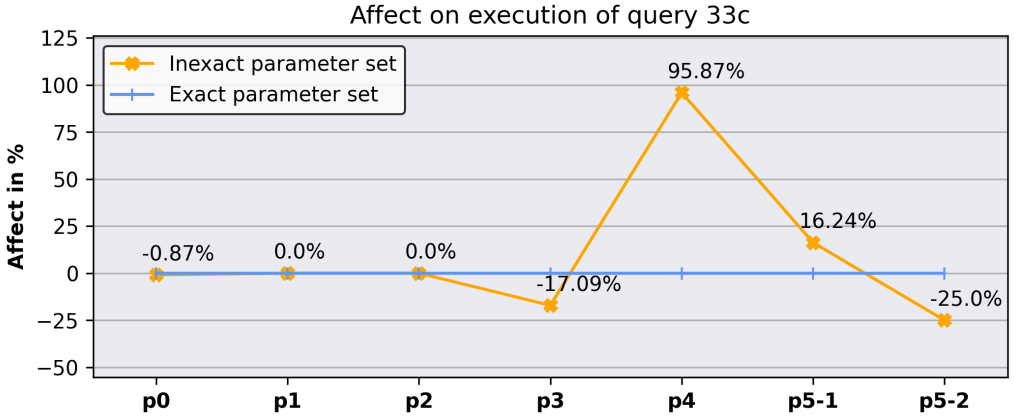


Figure 6.5: Difference of execution time of query 33c in percentage when provided with exact [blue] and inexact [orange] matching parameters.

6.3.2 Discussion

The observations from figure 6.4 and figure 6.5 showcases the three possible outcomes when executing a query in terms of performance. As described in section 3.1, repeated query execution can either result in equally fast execution (*p0*, *p1* and *p2*), faster execution (*p3* and *p5-2*), or slower execution (*p4* and *p5-1*).

The outcome of changing the parameters (*p4*) and (*p5-2*) went as one might expect. Both values specify the same parameter, which is used to limit or increase the search area of the query executor by specifying the production period. When increasing the search (*p4*), the query execution takes longer time to complete, as the query executor is forced to parse more rows of data than previously. Similarly, when reducing the search space, the execution completes faster, as the requirements limits the number of possible rows of data for the executor to parse.

The outcome of the parameters (*p4*) and (*p5-2*) could quite possibly been found through some type of quick preparatory analysis of the parameters values, as the end of the production period is specified in the query. In contrast, it had presumably been far more difficult to predict the outcome when changing parameter (*p5-1*) from '3.0' to '6.0', as there is are no indicators in the query statement that suggest that the search space would increase.

These results highlight a major weakness of allowing execution of inexact matching queries without any deeper analysis of the parameters. It is very difficult to predict how the change in parameter values will end up affecting the performance of the query processor. As stated previously in section 3.1 and 3.3.2, unpredictable performance is arguable the most critical outcome to avoid for the majority of application. Therefore, the results show that executing caching queries with inexact matching parameters is not only unwise, but also cause major problems for applications relaying on predictable response time from the RDMBS.

Chapter 7

Conclusion

The main goal of this thesis was to investigate whether it was feasible to implement plan caching to enhance the performance of the MySQL query processor. So far, fundamental knowledge about the query optimization was presented in chapter 2. In chapter 3, plan caching as concept and different implementations, in addition to various related approaches to influence the decision-making of the query optimizer were described. Then, the implementation of a plan cache prototype in MySQL was described in chapter 4, before three experiments described in chapter 5 were conducted, and the findings were presented and discussed in chapter 6. Based on the observations made throughout the stated chapters, this chapter attempts to answer the research questions that were presented in chapter 1. The following questions were asked:

RQ1: Does the findings indicate that implementing plan caching is a measure that would enhance the overall performance of the MySQL query processor?

RQ2: How is the performance of the MySQL query processor affected when using a plan cache enforcing *inexact match policy* compared to when enforcing *exact match policy*?

To answer **RQ1**. Based on the findings, it appears that plan caching can contribute to enhance the overall performance of the MySQL query processor. The experiments show that cached queries on average completes slightly faster than the ad hoc queries. More importantly, the execution time also appears to vary less when executing cached queries compared to ad hoc queries. However, the experiments have also revealed that the current plan cache implementation is prone to occasionally cause poor query performance. Before MySQL can consider implementing the plan cache implementation as a feature, this problem must be eliminated.

To answer **RQ2**. Based on the finding, caching queries with inexact matching parameters is a too unpredictable to be enforced. Whether the performance of the query processor increases, remain the same or decreases, when allowing uncritical inexact matching parameters to be included, depends entirely on what parameter value that is provided. To make caching of

query execution plans with inexact matching parameters feasible with predictable performance, more sophisticated methods to carry out match operations must be implemented than the one experimented with in this thesis.

7.1 Future work

There are several aspects of plan caching that would be interesting to closer investigate in the future. First, it would be interesting to experiment with more sophisticated solutions to closer investigate whether it is possible to reuse query execution plans when enforcing inexact matching policy with predictable performance. Solutions such as PLASTIC[15] and Parametric Plan Caching[8], have shown promising results, so a cluster based implementation could be a possible solution. Further, it would be interesting to closer examine how the size of the plan cache influences the search time when carrying out match operations. Being able to establish a limit to the number of query execution plans possible to store in the cache before the number of match operations becomes too time-consuming and ends up harming the overall performance of the query processor, would be a valuable contribution. Lastly, it would also be interesting to closer investigate how the logic for carrying out replacement operations affects the performance of the query processor.

Bibliography

[1] P. Taylor, *Total data volume worldwide 2010-2025*, en. [Online]. Available: https://www.statista.com/statistics/871513/worldwide-data-created/ (visited on 05/25/2023).

[2] *DB-Engines Ranking*, en. [Online]. Available: https://db-engines.com/en/ranking (visited on 05/14/2023).

[3] *What is MySQL?* en. [Online]. Available: https://www.oracle.com/mysql/what-is-mysql/ (visited on 05/14/2023).

[4] *PREPARE*, en, May 2023. [Online]. Available: https://www.postgresql.org/docs/15/sql-prepare.html (visited on 05/26/2023).

[5] G. Low, A. Prout, C. Fraser, *et al.*, *Plan Caching in SQL Server 2008*, en-us, Aug. 2009. [Online]. Available: https://learn.microsoft.com/en-us/previous-versions/sql/sql-server-2008/ee343986(v=sql.100) (visited on 05/31/2023).

[6] H. Baer, B. Bolltoft, A. Cakmak, *et al.*, *Improving Real-World Performance Through Cursor Sharing*, en-US, topic, Publisher: December2021. [Online]. Available: https://docs.oracle.com/en/database/oracle/oracle-database/19/tgsql/improving-rwp-cursor-sharing.html#GUID-8CC2E0D8-4C67-4795-93A8-9F563E7F27C7 (visited on 05/26/2023).

[7] *SyBooks Online*. [Online]. Available: https://infocenter.sybase.com/help/index.jsp?topic=/com.sybase.help.sqlanywhere.12.0.1/dbusage/plan-caching-queryopt.html (visited on 01/19/2023).

[8] G. Aluç, D. E. DeHaan, and I. T. Bowman, "Parametric Plan Caching Using Density-Based Clustering," in *2012 IEEE 28th International Conference on Data Engineering*, ISSN: 2375-026X, Apr. 2012, pp. 402–413. DOI: 10.1109/ICDE.2012.57.

[9] J. M. Hellerstein, M. Stonebraker, and J. Hamilton, "Architecture of a Database System," en, *Foundations and Trends® in Databases*, vol. 1, no. 2, pp. 141–259, 2007, ISSN: 1931-7883, 1931-7891. DOI: 10.1561/1900000002. [Online]. Available: http://www.nowpublishers.com/article/Details/DBS-002 (visited on 04/13/2023).

[10] "Query Optimization," en, in *Fundamentals of database systems*, Seventh edition, OCLC: ocn913842106, Hoboken, NJ: Pearson, 2016, pp. 691–727, ISBN: 978-0-13-397077-7.

[11] "Strategies for Query Processing," en, in *Fundamentals of database systems*, Seventh edition, OCLC: ocn913842106, Hoboken, NJ: Pearson, 2016, pp. 655–687, ISBN: 978-0-13-397077-7.

[12] *MySQL :: MySQL 8.0 Reference Manual :: 13.5 Prepared Statements*. [Online]. Available: https://dev.mysql.com/doc/refman/8.0/en/sql-prepared-statements.html (visited on 02/28/2023).

[13] I. Fernandez, "Establishing a baseline," en, in *Beginning Oracle Database 12c Administration: From Novice to Professional*, I. Fernandez, Ed., Berkeley, CA: Apress, 2015, pp. 324–328, ISBN: 978-1-4842-0193-0. DOI: 10.1007/978-1-4842-0193-0_3. [Online]. Available: https://doi.org/10.1007/978-1-4842-0193-0_3 (visited on 05/31/2023).

[14] M. Ziauddin, D. Das, H. Su, Y. Zhu, and K. Yagoub, "Optimizer plan change management: Improved stability and performance in Oracle 11g," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1346–1355, Aug. 2008, ISSN: 2150-8097. DOI: 10.14778/1454159.1454175. [Online]. Available: https://doi.org/10.14778/1454159.1454175 (visited on 12/18/2022).

[15] A. Ghosh, J. Parikh, V. S. Sengar, and J. R. Haritsa, "Plan selection based on query clustering," in *Proceedings of the 28th international conference on Very Large Data Bases*, ser. VLDB '02, Hong Kong, China: VLDB Endowment, Aug. 2002, pp. 179–190. (visited on 06/03/2023).

[16] W. Stallings and K. Goutam, *Operating systems: internals and design principles*. Pearson New York, 2012, vol. 9.

[17] *Chapter 1 - What's New in SQL Server 7.0*, en-us, Jan. 2010. [Online]. Available: https://learn.microsoft.com/en-us/previous-versions/cc917537(v=technet.10) (visited on 06/01/2023).

[18] R. West and D. Wilson, *Parameter Sensitive Plan optimization - SQL Server*, en-us, May 2023. [Online]. Available: https://learn.microsoft.com/en-us/sql/relational-databases/performance/parameter-sensitive-plan-optimization (visited on 06/01/2023).

[19] H. Baer, B. Bolltoft, A. Cakmak, *et al.*, *Improving Real-World Performance Through Cursor Sharing*, en-US, topic, Publisher: December2021. [Online]. Available: https://docs.oracle.com/en/database/oracle/oracle-database/19/tgsql/improving-rwp-cursor-sharing.html (visited on 06/03/2023).

[20] *17 Optimizer Hints*, Dec. 2003. [Online]. Available: https://docs.oracle.com/cd/B12037_01/server.101/b10752/hintsref.htm (visited on 06/05/2023).

[21] *IBM Documentation*, en-US, Mar. 2023. [Online]. Available: https://www.ibm.com/docs/en/db2/11.1?topic=plans-optimization-profiles-guidelines (visited on 06/05/2023).

[22] *MEM_root Struct Reference*. [Online]. Available: https://dev.mysql.com/doc/dev/mysql-server/latest/structMEM__ROOT.html#details (visited on 04/19/2023).

[23] *THD Class Reference*. [Online]. Available: https://dev.mysql.com/doc/dev/mysql-server/latest/classTHD.html (visited on 04/19/2023).

[24] *Swap_mem_root_guard Class Reference*. [Online]. Available: https://dev.mysql.com/doc/dev/mysql-server/latest/classSwap__mem__root__guard.html#details (visited on 04/19/2023).

[25] *AccessPath Struct Reference*. [Online]. Available: https://dev.mysql.com/doc/dev/mysql-server/latest/structAccessPath.html#details (visited on 05/14/2023).

[26] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann, "How good are query optimizers, really?" *Proceedings of the VLDB Endowment*, vol. 9, no. 3, pp. 204–215, Nov. 2015, ISSN: 2150-8097. DOI: 10.14778/2850583.2850594. [Online]. Available: https://dl.acm.org/doi/10.14778/2850583.2850594 (visited on 05/01/2023).

[27] A. Kopytov, *Sysbench*, original-date: 2015-03-07T08:27:40Z, May 2023. [Online]. Available: https://github.com/akopytov/sysbench (visited on 05/10/2023).

[28] *TPC-C Homepage*. [Online]. Available: https://www.tpc.org/tpcc/ (visited on 05/10/2023).

[29] *TPC-H Homepage*. [Online]. Available: https://www.tpc.org/tpch/default5.asp (visited on 05/10/2023).

[30] *Configuring InnoDB Buffer Pool Size*. [Online]. Available: https://dev.mysql.com/doc/refman/5.7/en/innodb-buffer-pool-resize.html (visited on 05/10/2023).

Appendix

A Source code

The source code of the implementation can be found at: https://github.com/jonasbrunvoll/mysql-server. The open-source mysql-server repository, can be found at MySQL's Github repository: https://github.com/mysql/mysql-server.

B Experiment 1 - Benchmark query selection

B.1 Distribution of query optimization and execution

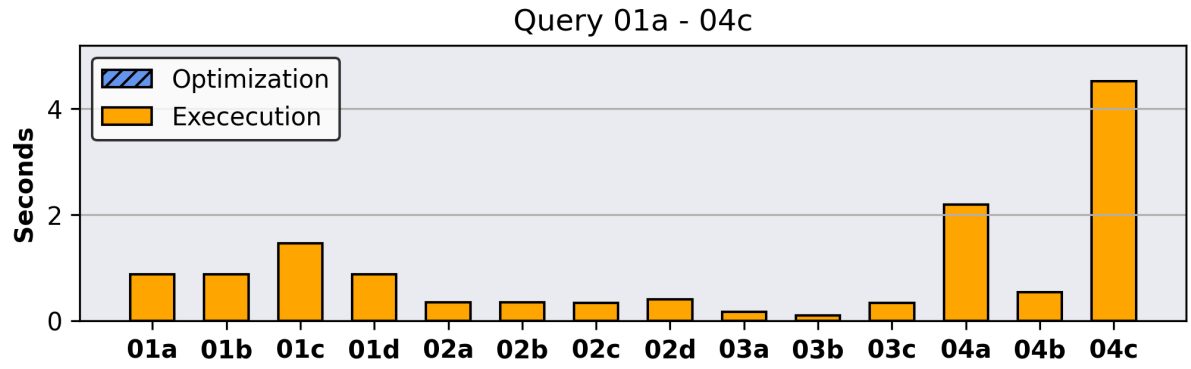


Figure 1: The total execution divided between time set to optimization [blue] and time set to execution [orange], when carrying out ad hoc execution of query 01a to query 04c inclusive.

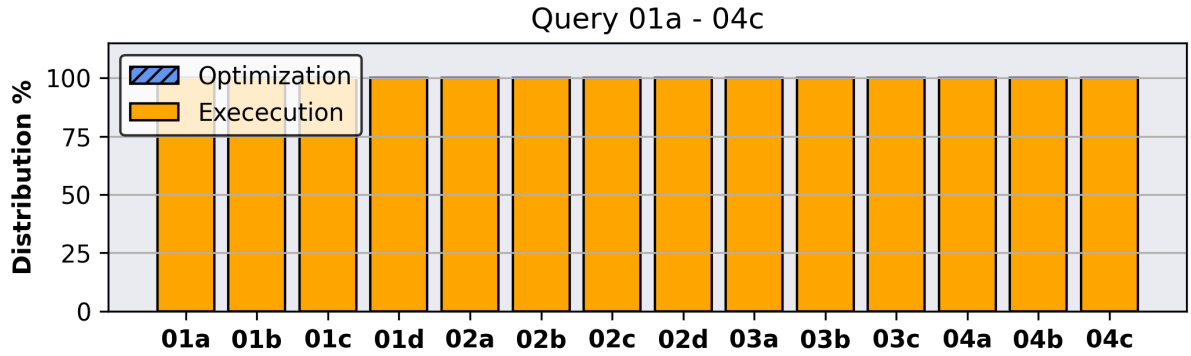


Figure 2: The proportional distribution divided between time set to optimization [blue] and time set to execution [orange], when carrying out ad hoc execution of query 01a to query 04c inclusive.

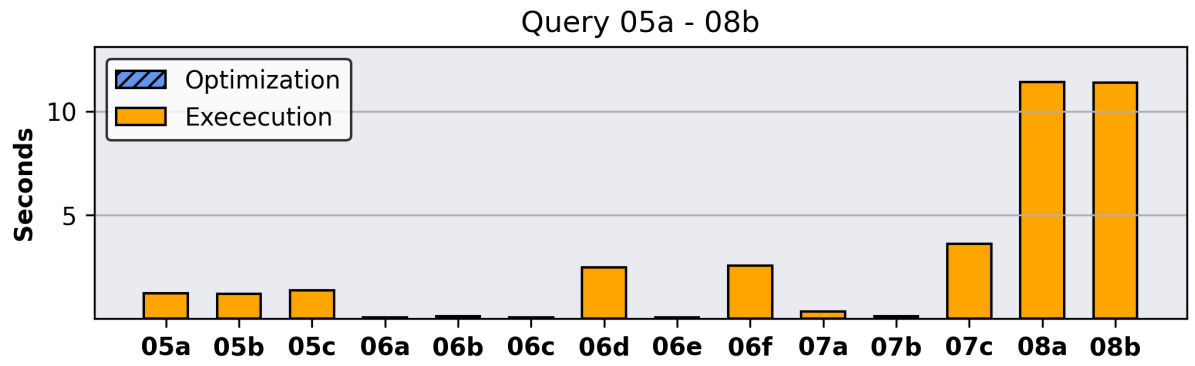


Figure 3: The total execution divided between time set to optimization [blue] and time set to execution [orange], when carrying out ad hoc execution of query 05a to query 08b inclusive.

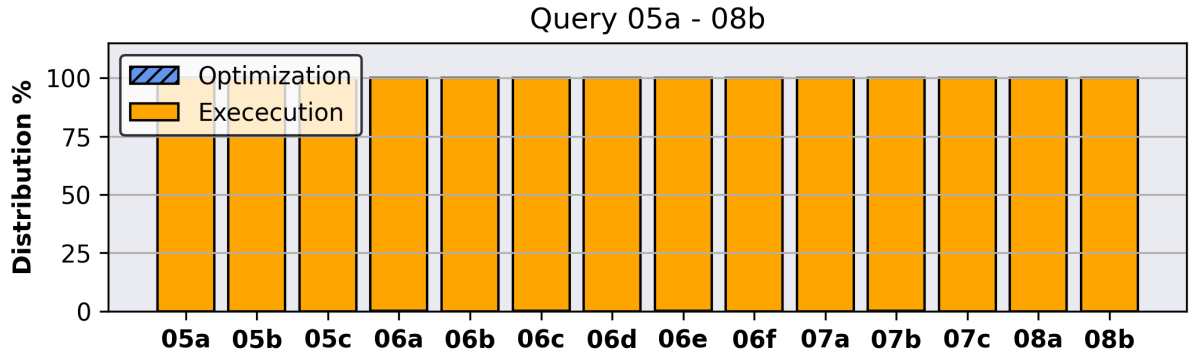


Figure 4: The proportional distribution divided between time set to optimization [blue] and time set to execution [orange], when carrying out ad hoc execution of query 05a to query 08b inclusive.

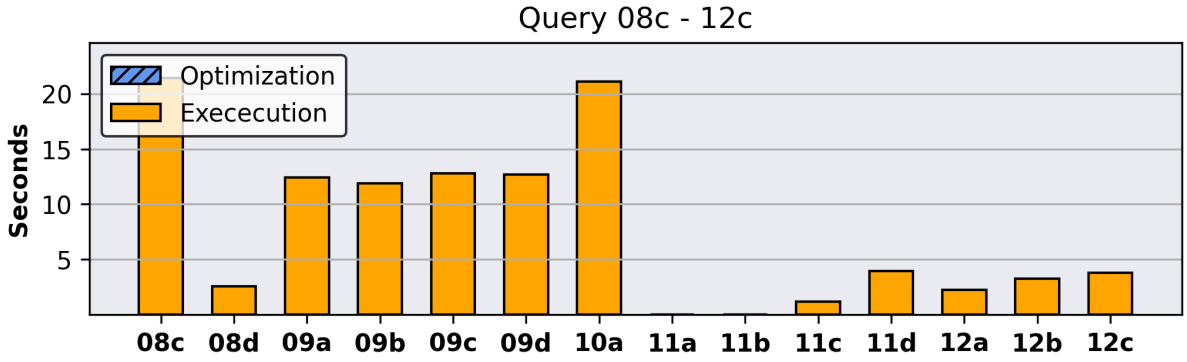


Figure 5: The total execution divided between time set to optimization [blue] and time set to execution [orange] , when carrying out ad hoc execution of query 08c to query 12c inclusive.

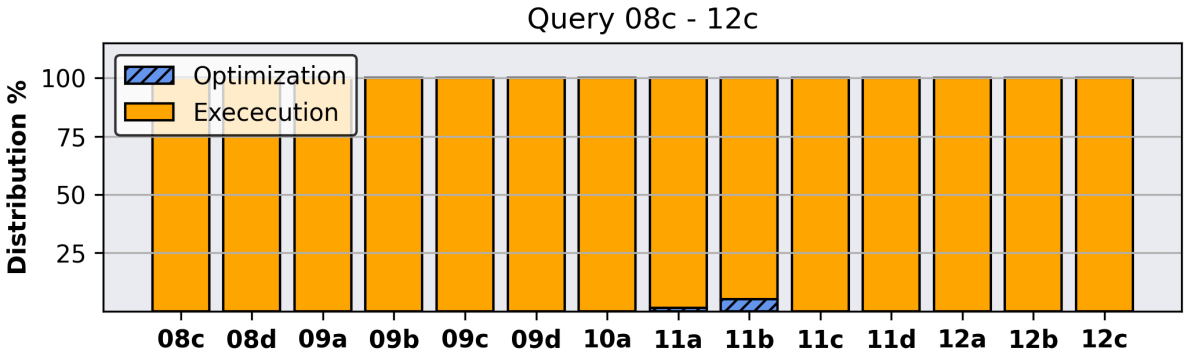


Figure 6: The proportional distribution divided between time set to optimization [blue] and time set to execution [orange], when carrying out ad hoc execution of query 08c to query 12c inclusive.

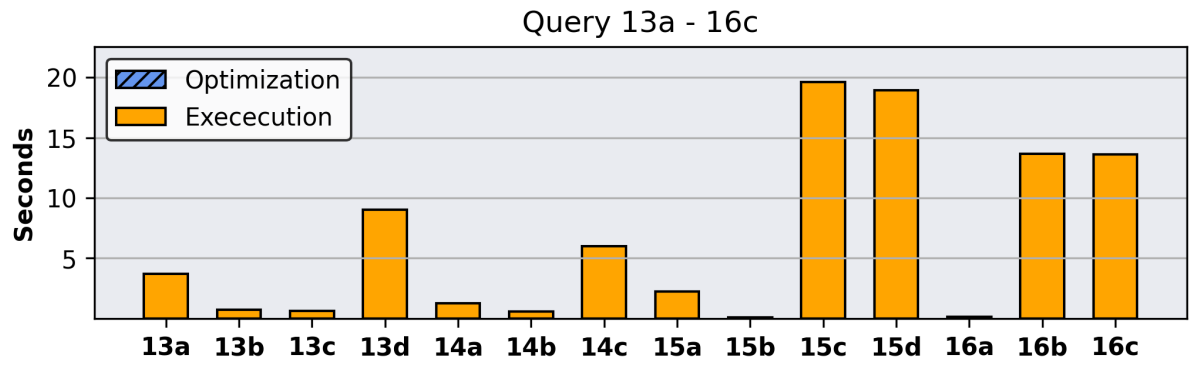


Figure 7: The total execution divided between time set to optimization [blue] and time set to execution [orange], when carrying out ad hoc execution of query 13a to query 16c inclusive.

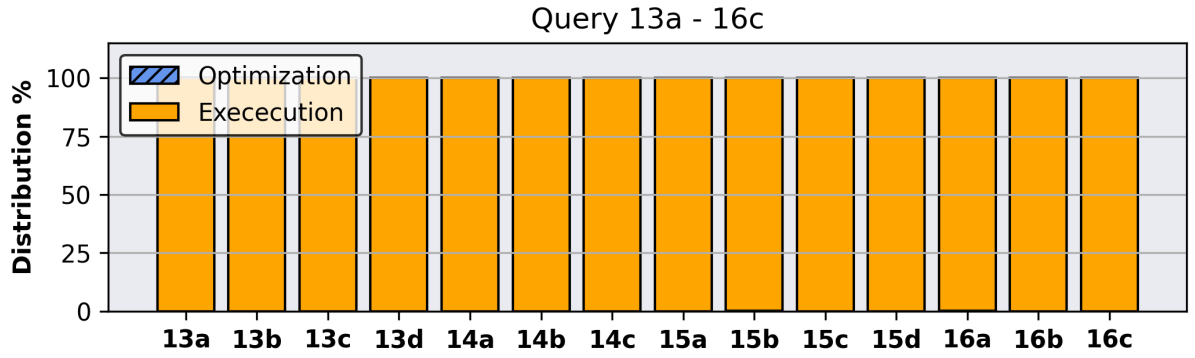


Figure 8: The proportional distribution divided between time set to optimization [blue] and time set to execution [orange], when carrying out ad hoc execution of query 13a to query 16c inclusive.

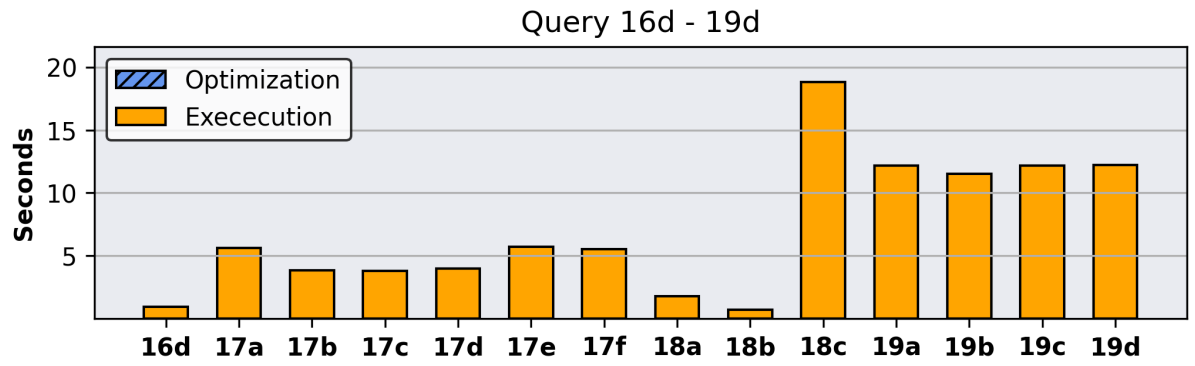


Figure 9: The total execution divided between time set to optimization [blue] and time set to execution [orange], when carrying out ad hoc execution of query 16d to query 19d inclusive.

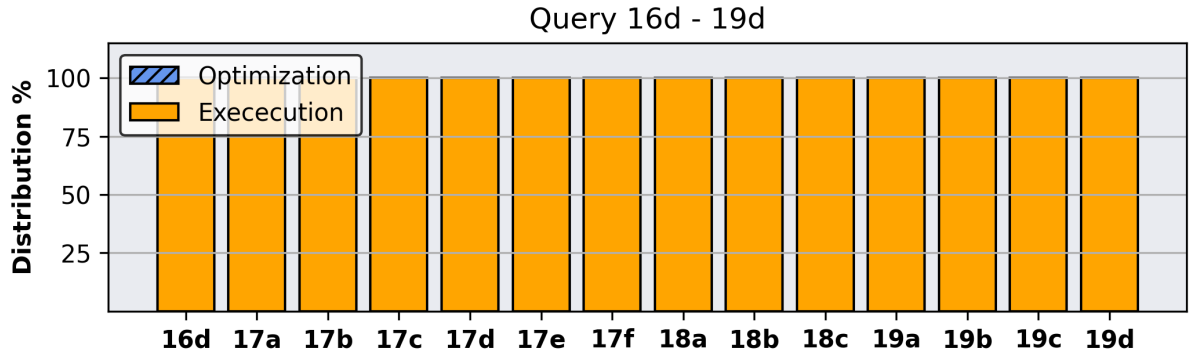


Figure 10: The proportional distribution divided between time set to optimization [blue] and time set to execution [orange], when carrying out ad hoc execution of query 16d to query 19d inclusive.

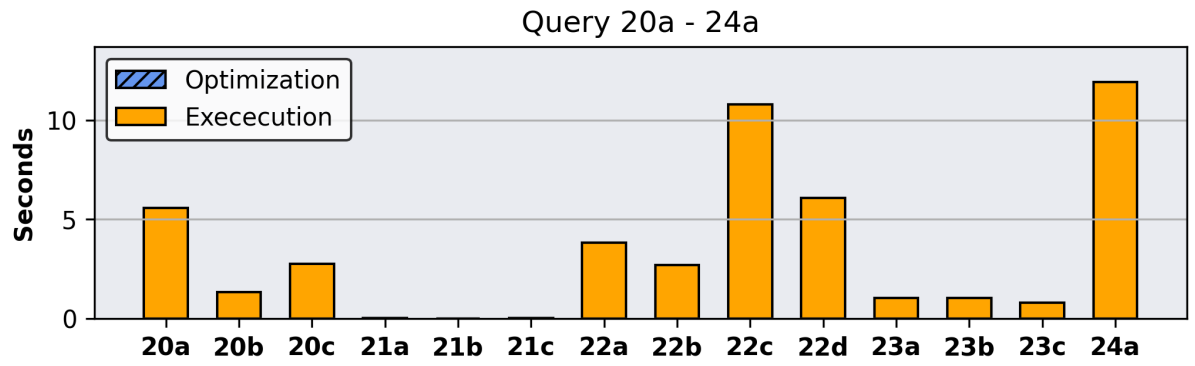


Figure 11: The total execution divided between time set to optimization [blue] and time set to execution [orange], when carrying out ad hoc execution of query 20a to query 24a inclusive.

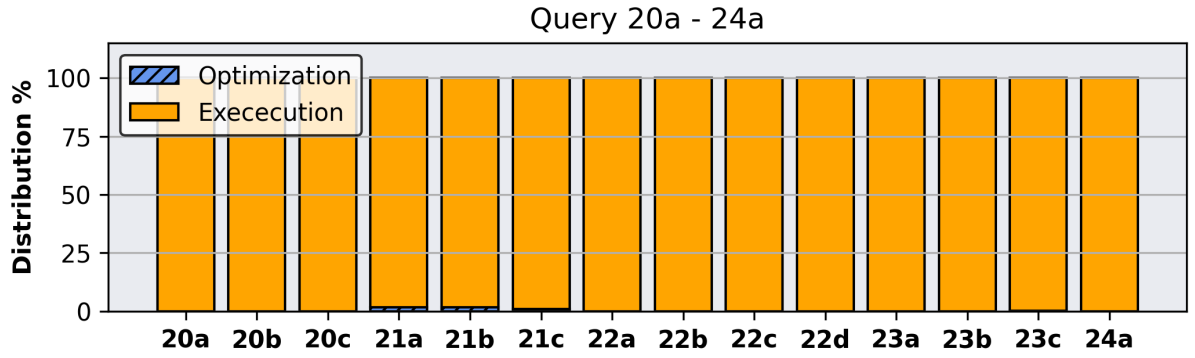


Figure 12: The proportional distribution divided between time set to optimization [blue] and time set to execution [orange], when carrying out ad hoc execution of query 20a to query 24a inclusive.

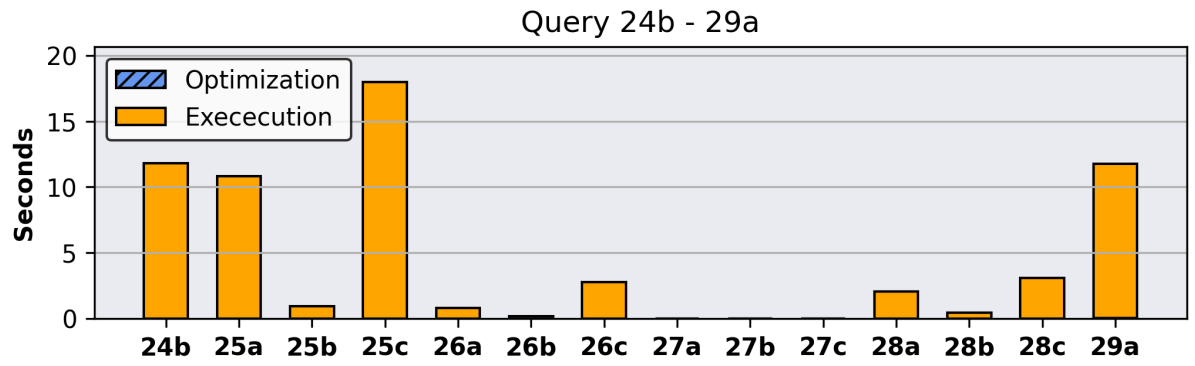


Figure 13: The total execution divided between time set to optimization [blue] and time set to execution [orange], when carrying out ad hoc execution of query 24b to query 29a inclusive.

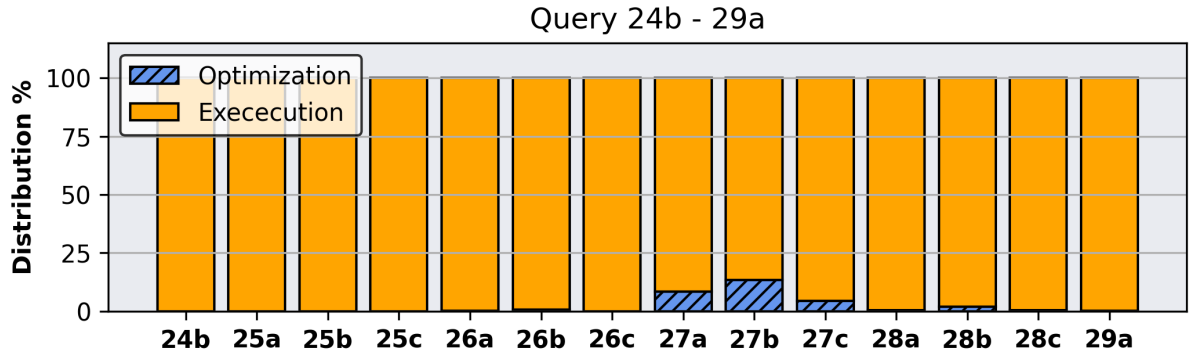


Figure 14: The proportional distribution divided between time set to optimization [blue] and time set to execution [orange], when carrying out ad hoc execution of query 24b to query 29c inclusive.

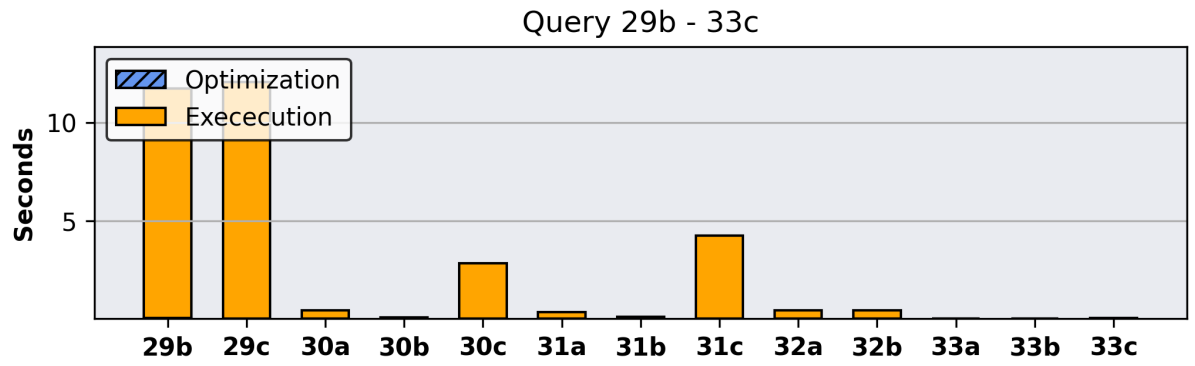


Figure 15: The total execution divided between time set to optimization [blue] and time set to execution [orange], when carrying out ad hoc execution of query 29b to query 33c inclusive.

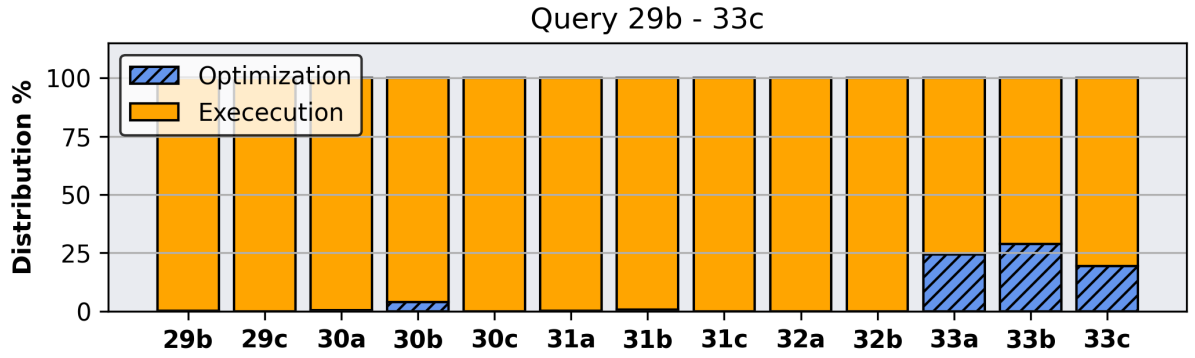


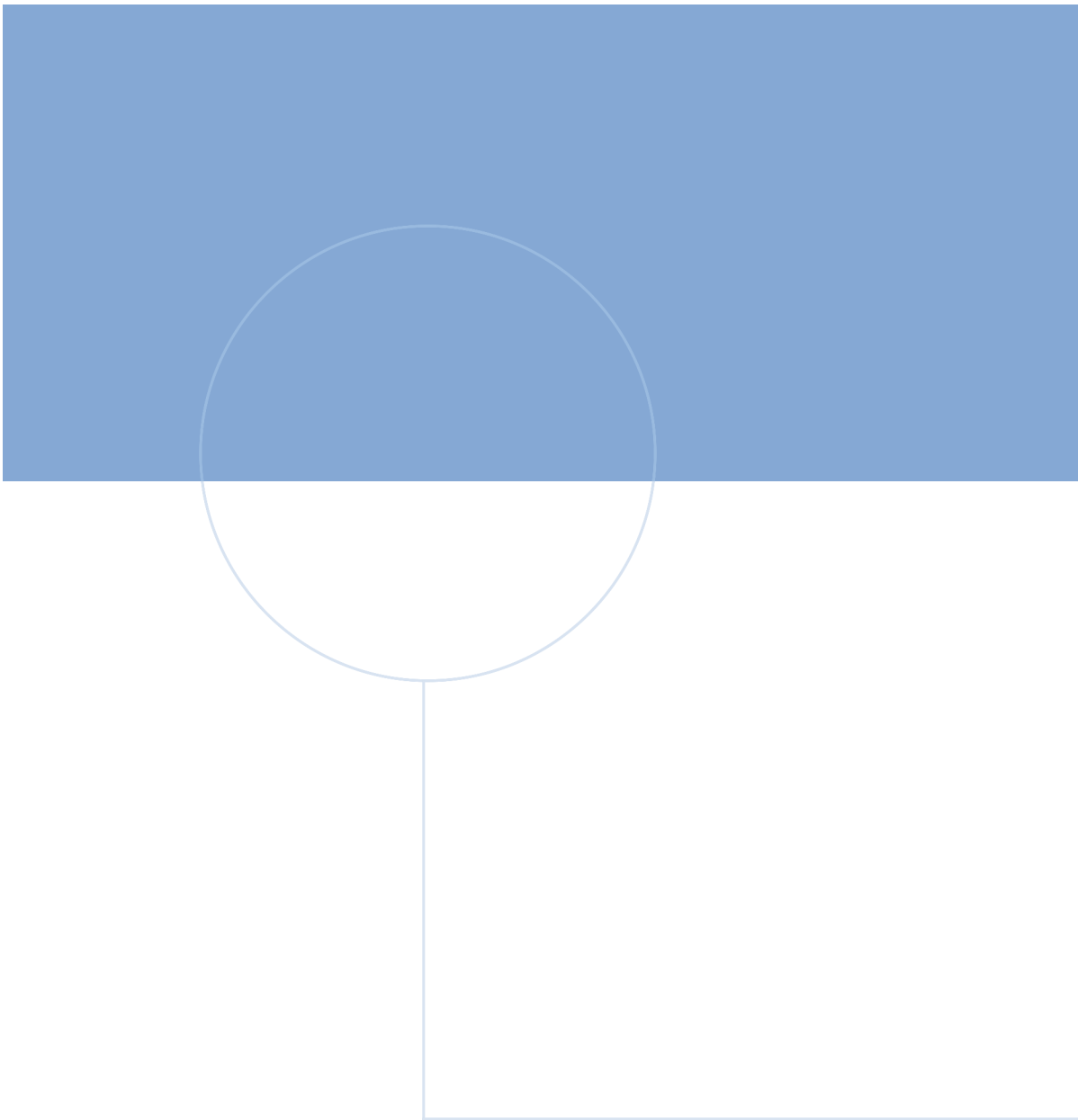
Figure 16: The proportional distribution divided between time set to optimization [blue] and time set to execution [orange], when carrying out ad hoc execution of query 29b to query 33c inclusive.

C Experiment 2 - Plan cache - exact match policy

C.1 Recorded performance - tables

Performance ad hoc executions					
Query	*Max optimization - (ms)*	*Min execution - (ms)*	*Max execution - (ms)*	*Average execution - (ms)*	*Average standard derivation - (ms)*
11b	0.605	1.235	2.941	1.4309	0.2501
27a	1.448	4.335	7.910	4.8573	0.4127
27b	1.565	2.761	8.078	3.9253	0.3588
27c	1.232	7.926	14.896	8.8534	0.6142
30b	2.667	46.205	61.750	48.2223	1.5712
33a	3.807	3.784	7.354	4.4850	0.4742
33b	3.989	3.451	6.788	4.0711	0.4372
33c	6.223	10.164	18.196	11.4917	0.6910
Performance cached executions					
11b	0.003	1.237	2.429	1.3603	0.1318
27a	0.002	4.343	7.566	4.8475	0.3555
27b	0.005	2.752	7.334	3.9180	0.3515
27c	0.004	8.116	15.057	8.8473	0.6002
30b	0.003	46.217	55.461	48.1875	1.4322
33a	0.003	3.746	7.303	4.3057	0.4240
33b	0.002	3.414	9.581	3.9411	0.4484
33c	0.002	10.026	16.744	11.1659	0.6666
Performance fastest cached executions					
11b	0.122	1.258	1.421	1.3370	0.0594
27a	0.688	4.391	4.735	4.5961	0.0983
27b	0.683	2.752	3.197	2.9452	0.1768
27c	0.573	8.271	8.578	8.4176	0.1054
30b	1.942	46.217	46.469	46.3844	0.0849
33a	1.855	3.792	4.517	4.0854	0.2623
33b	1.777	3.444	3.970	3.6476	0.1883
33c	3.397	10.184	11.072	10.7200	0.3489

Table 1: Experiment 2, recorded performance of the ad hoc, the cached and the fastest cached query executions, measured in milliseconds.



NTNU

Norwegian University of
Science and Technology