Masteroppgave

NTNU
Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for datateknologi og informatikk

Lisa Willa

Pinning Multiple Plans in MySQL

Masteroppgave i Computer Science
Veileder: Norvald H. Ryeng
Juni 2023

Lisa Willa

Pinning Multiple Plans in MySQL

Masteroppgave i Computer Science
Veileder: Norvald H. Ryeng
Juni 2023

Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for datateknologi og informatikk

NTNU

Kunnskap for en bedre verden

Abstract

Most mature DBMSs provide some means of manipulating the query optimizer into making query execution plans with specific traits. This gives database administrators a powerful tool to handle the rare but existing cases where the optimizer produces bad plans that cause long execution times. Because bad plans can be disastrous to performance, it is common for a DBMS to provide at least one such manipulation tool. MySQL along with many other systems provides optimizer hints that can manipulate certain aspects of the plan. Many commercial systems provide features that manipulate queries automatically, called plan pinning. MySQL does not provide such features, and MySQL's new optimizer does not even provide optimizer hints.

This project aims to see if it is possible to implement plan pinning that can pin multiple plans at once in MySQL and whether these implementations have good enough performance to be considered for future use. To accomplish this a literature study is conducted to study existing techniques for plan manipulation and plan pinning. Secondly, two different prototypes were implemented in the new optimizer. Both of the prototypes are capable of pinning multiple plans for a query. These are join order hints that are capable of pinning multiple complicated join orders, and hash-pinning that uses hashes generated from the plans to pin the full plans. Then an experiment running JOB queries was conducted. The study of the optimize times of the JOB queries for the two prototypes shows that join order hints are better in the general case, but that the hash-pinning has more potential if the hash generation function gets better.

Sammendrag

De fleste etablerte databasehåndteringssystemr har måter å manipulere spørringsplanleggeren til å lage eksekveringsplaner med spesifikke egenskaper. Dette gir database administratorer et kraftig verktøy til å håndtere den sjeldne, men eksisterende begivenheten der planleggeren produserer dårlige planer som fører til lange eksekveringstider. Fordi dårlige planer kan være katastrofale for ytelse, er det vanlig for et databasehåndteringssystemer å ha minst et slik manipuleringsverktøy. MySQL, som mange andre systemer, tilbyr funksjonaliteten planleggingshint (optimizer hint) som kan manipulere aspekter ved planen. Mange kommersielle systemer tilbyr funksjonalitet som kan manipulere spørringer automatisk, kalt planlåsing (plan pinning). MySQL tilbyr ikke slik funksjonalitet, og MySQL sin nye planlegger har ikke engang planleggingshint.

Dette prosjektet vil forsøke å se om det er mulig å implementere plan låsing (plan pinning) som kan låse flere planer samtidig i MySQL, og hvorvidt disse implementasjonene har god nok ytelse til å kunne brukes i fremtiden. For å få til dette er det gjort en litteraturstudie for å studere eksisterende teknikker for plan manipulasjon og planlåsing. Etterpå ble to forskjellige prototyper implementert i den nye planleggeren. Begge prototypene er i stand til å låse flere planer for en spørring. De to prototypene er join order hint (skjøt rekkefølge hint) som er i stad til å låse flere kompliserte skjøte rekkefølger, og hash-pinning(evt. hash-låsing) som bruker hasher generert fra planer til å pinne hele planer. Så ble et eksperiment kjørt med JOB spørringer gjennomført. Studien av planleggingstidene for JOB spørringer for de to viser at join rekkefølge hintene er mer effektive i snitt, men at hash-låsing har mer potensiale hvis hash genereringsfunksjonen blir bedre.

Acknowledgements

I would like to thank Associate Professor Norvald H. Ryeng for excellent guidance while working on this master's Thesis. His knowledge of the MySQL server, his insights, and our many good discussions have been essential to this project.

I would also like to thank the amazing people at the Oracle office in Trondheim for the great environment there and the abundance of knowledge. Among the great people in Oracle I would especially like to thank Knut Anders Hatlen for his help in understanding the hypergraph optimizer's codebase, and Roy Lyseng for his essential help with re-runs of the hypergraph optimizer.

In addition, I would like to thank Randi Jacobsen Willa (the honorary rubber duck) for being a great sparring partner, proofreader and assistance in making great bar-graphs, Hanne Willa for her assistance with creating figures, and Jonas Brunvoll Larssen for an introduction to scripting.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Providers of modern database management systems strive to improve the performance of their systems to satisfy the demands of their clients. Performance is one of the most important, if not the most important metric for the end user of a database as it determines how fast a query is answered [1]. As such most database management systems (DBMS) have done a lot to improve performance. Among the more common measures taken we find indexes, performance metrics for query tuning[2][3], Write-Ahead logging and query rewrites. Despite this the most critical component for performance is the query optimizer[4][5][6]. The query optimizer's job is to create a query execution plan for the queries the database receives. The optimizer's query execution plan dictates how the system chooses to execute a query. Herein lies choosing things like access methods, join order, join algorithms, and more[7][8]. This means that the optimizer may change the complexity of a query execution and also the amount of I/O involved in it. This means that the execution time of a query depends heavily on the execution plan[7][9].

The query optimizer of a DBMS is highly optimized to find good plans as fast as possible. This is typically done using a combination of heuristics and cost models. Wherein heuristics are generally smart rules that are applied to the query, and the cost models use statistics to help it evaluate the cost (cost can be thought of as estimated execution time) of different possible plans in order to choose the cheapest plan[9][7].

The optimizer usually comes up with good query execution plans, but on occasion, the optimizer estimates wrong and produces a bad plan. This can drastically decrease performance and is considered a big problem. Bad plans can cause queries that usually would take seconds to execute to take minutes or more. Another related problem is when the optimizer produces a decent plan whose performance is worse than what the end users are used to. Users have little patience for degradation of performance. This phenomenon typically occurs after updates to the optimizer or statistics[5]. When the optimizer produces bad plans for a query, the database specialists(DBAs) start tuning the query[10][3].

In the cases where query tuning does not work well enough, or one worries that the plan will degenerate again, the powerful technique of plan pinning is utilized. It is a type of feature that exists in most mature DBMSs that allows the user to override the optimizer partially or completely to enforce/pin a desired query execution plan[2][11][5][12]. It is considered a last resort and warned against as it will often cause you to lose out on performance gains. This is because the optimizer is often smarter, and a pinned plan cannot adapt to changes[10]. However, it can be a good tool to ensure performance in some cases, and sometimes it is necessary to have predictable performance[5].

There are two primary types of plan pinning. The most common one uses something called optimizer hints. These nudges (hints) the optimizer to make certain decisions during optimization based on the hints. Optimizer hints can for instance enforce specific join algorithms, how tables are read, and in which order tables are joined. Optimizer hints are added to the query text (SQL statement) often within the comments and take effect for the query they come with. One can hint at so many parts of the plan that one (almost) pins a full plan, i.e. the optimizer does not decide

(nearly) anything and the hints dictate the entire plan[2][13][14], this is exploited in some systems to pin whole plans[14][5]. Some systems implement functionality that can make the DBMS apply the hints for you when it recognizes the query. The query is typically recognized by a query id that can be generated from the query text[14][13]. When the process is automatic it is formally called plan pinning.

The second type of plan pinning pins multiple full plans for a query and also adapts to changes by discovering and pinning new better plans. This is typically implemented by letting the optimizer work as usual first. If the query is pinned and the optimizer did not generate one of the allowed(pinned) plans it rejects it and generates the best alternative among the pinned plans for the query and executes this instead. Afterward, the DBMS will check if the rejected plan was actually better than the pinned ones by checking the performance of the plan. If the new plan is better it is pinned as well, this means that optimizing without pinning first allowed the DBMS to discover a better plan. Pinning multiple plans and updating the allowed plans makes this method able to negate most of plan pinning's weaknesses[5].

Plan pinning is a powerful tool provided by most mature DBMS to avoid plan regression and bad execution plans[2][15][16][12][5]. Many DBAs use these tools. MySQLs new hypergraph optimizer[17] does however not have significant hint functionality as of yet. And implementing pinning for this new optimizer opens up for new techniques. Especially an implementation of pinning that can pin multiple plans for one query. Most systems only allow one to pin a single plan pr query, which means that the plan that has been pinned has to work for all possible parameters for the query, which makes it less adaptable[10][18]. Pinning multiple plans can negate this[5].

This thesis will discuss plan pinning and its implementation in MySQL. This section will give a quick introduction to some of the most relevant topics and give an outline of the scope of the thesis. The next sections will give a more in-depth explanation of the relevant topics and the state of the art of optimizers, plan pinning, and plan caching. Then two implementations of plan pinning in MySQL will be presented and tested.

1.1 scope

This thesis will focus on the viability of implementing plan pinning for MySQL's two optimizers. Wherein the thesis will look into two approaches. One that utilizes optimizer hints, and one that coerces the optimizer to make the exact pinned plan by utilizing the hypergraph optimizers innate properties. The thesis will also study how the methods work if one wishes to pin multiple plans for the same query and check their performance. The prerequisite to understanding plan pinning and the state of the art of plan pinning and plan caching(overlapping properties) is also covered.

Research question 1: How can one implement plan pinning in an existing DBMS like MySQL?

Research question 2: Should pinning pin parts of an execution plan or the full execution plan?

Research question 3: Is it permissible to use less domineering pinning in order to discover potentially better plans?

Research question 4: Can pinning multiple plans for one query have benefits, and how costly would it be?

Chapter 2

Background

This chapter will cover the knowledge needed to understand the rest of this thesis. The first section 2.1 will cover the query processing pipeline that will illustrate how a DBMS processes a query and introduce important concepts about optimization and execution. The second section 2.2 will introduce query execution plans. The third section 2.3 will go into detail on the optimizers that make the query execution plans. The last section 2.4 will cover the bottom-up hypergraph optimizer that most of the prototypes introduced in chapter 4 are implemented in.

2.1 The Query Processing Pipeline

When a user issues a query to the DBMS, the query has to go through several steps before the DBMS can return the desired data. These steps are Lexing, parsing, resolve, rewrite, optimize, and execute[4]. In this thesis, the pipeline will be split the same way as MySQL and so the pipeline here will have these steps: parsing (lexing and parsing), prepare (resolve and rewrite), optimize and execute. This section will explain these steps.

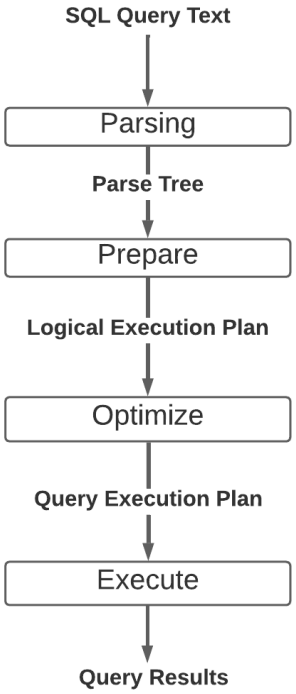


Figure 2.1: Illustration of the query processing pipeline

2.1.1 Parsing

When the query is sent to a DBMS it is written in some form of query language. Among these SQL is the most typical. However, SQL and other query languages are not written in machine code and as such needs to be interpreted by a "compiler". And because of this DBMSs implement the compiler components lexer and parser [19].

The lexer takes the SQL query text, turns it into tokens, and tags them. The lexer is in many ways a tokenizer that can tag the tokens with their symbol type. The types are keywords like SELECT, FROM, and so on. The second type is identifiers: table names, column names, and such. Literals are the values provided as parameters, like quoted strings and numbers [4].

When the lexer has produced a token it is fed to the parser. The parser takes the tokens as they come and determine whether the query has the correct syntax. It also removes comments. It does however parse optimizer hints! The parser also converts the tokens from the lexer into a parse tree which will be used for the next step in the pipeline. The parse tree is designed to be easy to process later.[19]

2.1.2 Prepare

This step is usually two steps, called resolve and rewrite. However, MySQL combines them into prepare and since MySQL is the system used in this thesis, the name Prepare is used.

The resolve step checks that the tables, views, columns, and other objects referenced in the query exist in the database. Resolve also takes locks for the tables, so that other processes cannot alter the table, during the query's lifetime. Here alter means removing the table or a column, or otherwise changing the metadata about the objects in the catalog. (Locks can work a little differently (snapshots), but that is not relevant here.) For all the objects (tables etc.) resolve also fetches metadata and statistics for later use.

It also does security checks, ensuring that the user that issued the query is allowed to perform the operations in the query (read/write/alter) on the tables and views referenced. If the security check fails it aborts the query. Lastly, it will bind metadata to the parse tree. The metadata is stored in a catalog, and now the parse tree refers to real table metadata in the catalog.

Rewrite is the step that prepares the parse tree for optimization. It utilizes rules and heuristics to make the tree take a form that is expected by the optimizer. It rewrites predicates to make them easier for the optimizer to use. Unpacks views and turn them into subqueries. Merge subqueries to have fewer subqueries for the plan, as each subquery has to be optimized separately. When the rewrite step is done the query tree is as optimizer-friendly as possible and is known as a logical execution plan. It can easily be made directly into a query execution plan, but usually a very bad one.

2.1.3 Optimize

This is easily the most complex part of the query processing pipeline. This is where the plan for how the DBMS will execute the query is made. The plan decides which algorithms will be used to read tables, join tables, and in what order the tables will be joined to name a few of the important decisions made by the optimizer. The plan that the optimizer generates is a query execution plan, but for simplicity, it will mostly be referred to as the plan or execution plan. A more detailed explanation of the query execution plan can be found further down in section 2.2.

The problem of finding an optimal execution plan is NP-Hard(exponential complexity) and cannot be completed quickly for a decent-sized plan space. Therefore the optimizers make plans that are good enough rather than squeezing the optimizer for the last drops of performance. After all, optimization takes time too, and if the optimization time becomes so long that optimization time exceeds the gain on the execution time it is simply not worth it. Therefore optimization focuses on finding a good plan rather than the best plan[8]. It is however important not to get a sub-optimal plan, aka. a bad plan. A bad plan can be the difference between a subsecond execution and one that takes minutes to execute, for the same query[5].

The process of making these execution plans is very complex, and this is because the optimal plan is not determined by the query text alone. It depends on many things like table size, data distribution in the tables, and whether there are indexes to name a few. Some optimizers ignore the dependence on data and use rules that have proved to work well most of the time. These have a lot of intelligent rules called heuristics, but cannot adapt to different data distributions. This makes for very fast optimizers, but they can lose out on a lot of performance when the data does not match the assumptions of the heuristic rules[20].

The alternative is cost-based optimization. The cost-based optimizers acquire statistics to be able to calculate the cost of the different options. The statistics were added to the parse tree in the prepare step. And some optimizers will also sample the tables to acquire more accurate statistics. The statistics allow the optimizer to make educated assumptions about table sizes and data skew (skewed distribution of data). Which again can be used to assign each operation a reliable cost, the least costly plans will be chosen. The statistics can be flawed as it only samples the tables, and wrong statistics are among the most common causes of bad plans. And minuscule differences in statistics or cost functions can cause the optimizer to make bad decisions[9][20].

Most optimizers combine heuristic rules with cost-based decisions. Typically the heuristic rules that are good regardless of data distribution and table size will be utilized. And the data-dependent decisions will be made based on statistics in a cost-based fashion[8][20].

Different types of optimizers will be described in greater detail further down in section 2.3.

2.1.4 Execute

The execute step executes the plan made by the optimizer. This means that it is the execute step that reads the tables or writes to them. Execute performs the algorithms to read and/or write data that the plan dictates. When execute is done it sends the results or confirmation to the client, completing the pipeline.

The executing step is usually done with interpreted plans instead of compiled code. Though both exist[4]. Compiling code in realtime has been shown to have good performance, especially the execution itself will be faster[21][22][23], but the compilation time can be too long to make it a better option for all cases[22][23]. This is a product of the higher capacity of RAM, the bottleneck has moved from disk to memory [24].

However, the most common way to execute code is still with an interpreter style, typically the volcano model.

Volcano model

Execution with an interpreter is typically done using the volcano/iterator model. It uses pipelining or stream-based processing to execute the operations of the plan in a pipeline rather than one at a time. Since executing the operations to completion would often force the DBMS to write temporary results to file, it would be very ineffective. So instead they make each sub-plan into an iterator. Each such iterator will produce one line of result(tuple) on command and will request rows from the iterators under it to fulfill its task/operation.

An iterator has an interface with three methods: *open*() which starts up the iterator with state, and *close*() which closes the iterator. An iterator will *close*() all its children before "closing" itself. And the most important method $next()/get_n ext()$, this method is the one that fetches rows. The iterator returns a single row when *next*() is called on it. Internally the iterator will call *next*() on its children to be able to produce the row if it is not a leaf [4].

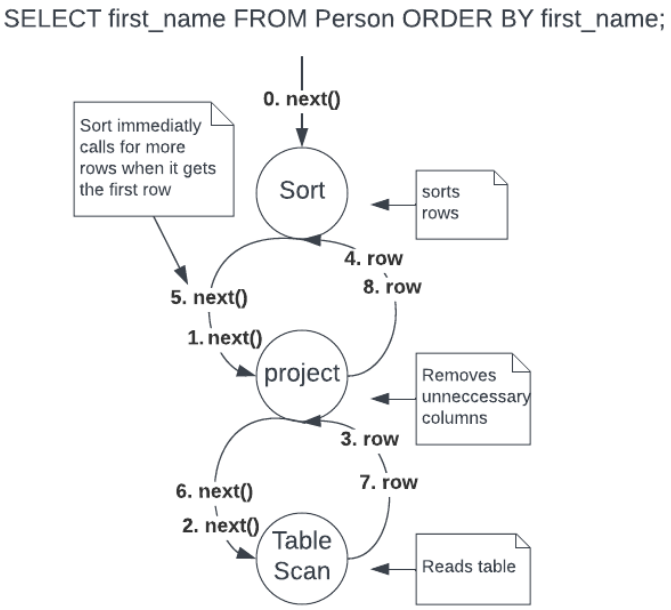


Figure 2.2: simplified example of volcano model

The iterators in the figure behave differently internally. For instance, the iterator whose operation

is sort needs to get all the rows from its children before it can complete its task, and so it will call for more rows until there are no more rows, then it will return the first row. This is therefore not a pipelining operation, but rather a blocking operation, making it rather expensive. After this all sort's children are closed and sort feeds its parent one row at a time. The project operation (removing unnecessary columns) can complete its operation for one row at a time and will call its child only once for each time it itself is called, a pipelining operation. Table scan reads from the table directly and therefore it does not call an iterator to perform its read. It is a pipelining operation due to producing rows without delay.

Note: a join operation would have two child iterators, and would call both as needed.

2.2 The Query Execution Plan

The most important decisions on the optimizer's part when making execution plans are access methods and join order. And so those are the ones covered here. However, there are many more sub-plan types, like join methods, sort, filter, projection, and more [4].

2.2.1 Access methods

The choice of access methods is one of the most crucial choices the optimizer makes, and it affects performance significantly. The most typical access methods found in most DBMSs are table scan, index range scan, and index lookup. To explain their traits one can use an old hardback encyclopedia, for this purpose assume that the encyclopedia is not sorted alphabetically.

Table Scan: is the equivalent of reading the encyclopedia from start to finish. Table scan reads the records in the order they are stored, and does this relatively fast. It does however read the entire table into memory. If one does not desire to read a large part of the table, a table scan reads far more than necessary and is thus slower than what is ideal. It is also worth noting that the data is not ordered when read by a table scan[4][7].

Index Lookup: is the equivalent of looking up desired words in the index of the encyclopedia, hence the name index lookup. This access method utilizes an index created for the table to find the records whose index matches a value or predicate. Index lookup reads only the desired records and thus saves time. The lookup itself takes a little time however and if one would read a large part of the table using Index lookup it would be ineffective. Same as looking up a very large number of words in an encyclopedia using the index. At some point reading the entire thing and filtering out the unnecessary data would be faster[4].

Index Range Scan: is the equivalent of using the index to find the first word beginning with a desired letter, and then read from there until the next first letter begins. However since the encyclopedia is unordered, this means multiple lookups, however instead of doing every lookup from scratch one reads all records in each relevant part of the index in succession. This is the only method that will give a sorted set of records [4][7].

2.2.2 Join Order

Join order is essential and the most complex part of query optimization, as the search space of join orders is huge. The join order dictates how large the intermediate results of the query execution are. This is very important because the size of the intermediate results determines how much data needs to be moved around during execution. And also how much data must be processed by subsequent operations. If the intermediate result is small and is joined on some predicate in the next operation, the number of checks would be small, and the resulting result from the join would likely be small. This means that subsequent operations would save processing, and fewer data would have to be in memory. It would be even worse if the results had to be written to disk in temporary files because of blocking operations too.

Long story short, The goal is to have as small intermediate results as possible for as long as possible. A join that causes a large result set would preferably be late in the execution order (aka. high in the plan tree). And joins with tiny results should be as far down as possible. This can also hinder large results from happening at all[4][25].

2.2.3 Examples of simple analysis

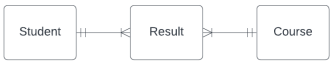


Figure 2.3: Simple ER model of the tables in the example

For this example, there are three tables: Student, which keeps track of students. Course, that holds all courses that a university provides. And the table Result holds the results of a course for a student (see 2.3). From this, a query is issued like this:

```
SELECT * FROM  Student
JOIN Result ON Student.id = Results.student_id
JOIN Course ON Result.course_id = Course.id
WHERE Course.subject_name = "Ex.Phil"
AND Result.score > 88;
```

The query looks at all students who have gotten the rare A grade (5% of all students) in the very common subject called Ex.Phil (80% of students have taken it). A manual analysis on access methods and join order for the query can go like this (bottom-up):

Step one: Find ideal access methods: Assuming an index on course_name in the table Course and only one record for Ex.Phil this is an obvious case for an index lookup. If no index had existed one would simply have to use a table scan.

Result is a typical case for an index range scan, where an interval in the values (88-100) is desired, especially when the range contains only a small limited part of the table. This holds true here as getting an A is rare. And an index lookup would be ineffective on the number of records that provide the score. If there is no index, a table scan is used. Note that if one was looking for a more common range a table scan might be better regardless.

The student table would use a table scan in this case, this is because most all students would need to be read to be evaluated as 80% have a result in the subject. Even when one looked at only the ones with the good grade, the index lookup could be ineffective, and a range scan makes no sense.

Step 2: look at join of two tables

Here there are several options, however, to simplify, the significance of join algorithms is removed and as such for this example, $Result \bowtie Student$ is equivalent with $Student \bowtie Result$, even though they usually would not be. Therefore for two tables the options$Student \bowtie Result$, $Student \bowtie Course$, and $Result \bowtie Course$ exist.

Firstly, one can safely eliminate $Student \bowtie Course$, this is because the two do not have any predicate that says how they should be joined together, meaning that the join would result in a cartesian product, which is not ideal [25].

Now the options $Student \bowtie Result$ and $Result \bowtie Course$ remain. The order that results in the smallest result set is considered the best. $Student \bowtie Result$ will produce a result set that join every student that has a high score. This will be a rather large number of students and results, most students will get an A grade within their time as a student, so the number of rows in the result would be at least as high as the number of students. On the other hand, $Result \bowtie Course$ will return the results for Ex.Phil that have a high score. This number will be about the same as our final result and will be considerably smaller than the number of students. Therefore $Result \bowtie Course$ is chosen as the first join.

step 3: join in the next table: simply join in the third table. And now the plan is done (for the intent of this example)

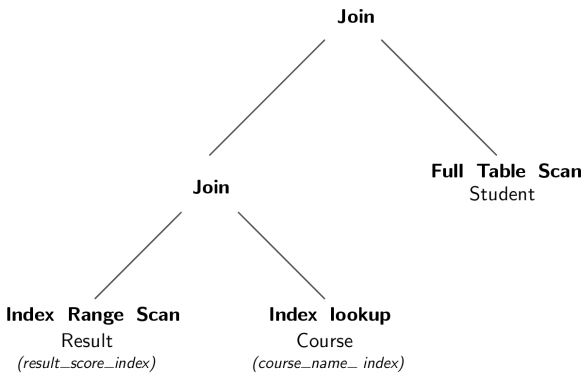


Figure 2.4: Plan from simplified example of query execution plan

2.2.4 Sort

The sort operation sorts the resultset it gets as input, and returns results sorted in a way specified by query through $ORDERBY$ and/or $GROUPBY$ (in an **interesting order**). The orderings specified by the query through $ORDERBY$ and/or $GROUPBY$ are called **interesting orders**.

The most logical way to sort the query result is to add a sort sub-plan at the top of the plan tree (last to be executed, see 2.2). The sort operation sorts the resultset it gets as input, and returns results sorted in the way specified by the query through $ORDERBY$ and/or $GROUPBY$. However, sort is a blocking operation, and as such it is very expensive for large data sets (see 2.2). Because the explicit sort operation is very expensive DBMSs try to avoid it if they can.

They try to avoid it by having intermediate results from the operations in the tree be sorted in an interesting order so that the final results are sorted in the specified way (the interesting order) without having to add a sort operation. All plans that contribute to this, aka those whose results are guaranteed to maintain or induce the interesting ordering, can help avoid an explicit sort operation.

If a sub-plan's results maintain the interesting order it is kept, in case there is a full plan that maintains the interesting orders so that it does not need to add an explicit sort sub-plan. If the plan without an explicit sort is cheaper it is chosen[7].

This is only possible if a form of range scan is possible for the table(s) with the relevant columns (or explicit sorting) or single value lookups like a lookup on the primary key. There are join algorithms that maintain ordering, and even some that exploit it[7]. But maintaining interesting orders in a plan does limit the number of possible operations that can be used in the plan. For example, table scan and index lookup on indexes that might return more than one value for one or more indexes and hash-join cannot be guaranteed to have/maintain an interesting order[4].

2.3 General Information About Optimizers in Databases

2.3.1 Bottom-up

A bottom-up optimizer uses dynamic programming to optimize. It will start building the first canonical part of the plan. The bottom is usually the access methods for the tables. And work upward before ending with a complete query execution plan. The advantage of this method is that it gets away with maintaining fewer plan variants. For instance, the best table access methods can be rather deterministic regardless of what happens later in the execution.

Specifically, the bottom-up optimizer will first explore and evaluate all sub-plans with only one table, then all possible relations with two tables in them, three tables, and so on until the best plan for the entire query is found[26]. To see a more detailed example of a bottom-up algorithm look at Sellinger in 2.3.4

2.3.2 Top-down

The top-down optimizers use a variation of dynamic programming called memoization. It starts with a logical execution plan. It transforms the logical plan into a query execution plan from the top down. It evaluates sub-plans it traverses on its way downwards and eventually ends up with a working query execution plan. In this regard, it is the exact counterpoint to bottom-up who traverses upwards instead.

The top-down optimizer starts by considering the possible operations for the uppermost sub-plan of the query first and traversing downwards. It is worth noting that this does not mean that all the child sub-plans must have been generated for each alternative, just that the children can exist. And as such the top-down optimizer can prune its search space without having to generate all the potential plans, only the ones it evaluates. The plans that produce the same output from the same input are compared to each other. The children of plans that are discarded are never generated or evaluated, saving a lot of processing [26][25].

2.3.3 Genetic Optimization

A genetic algorithm for optimization is another option. A genetic algorithm uses the idea behind Darwinistic evolution. It creates a population, in this case, query execution plans, and uses the concept of survival of the fittest to discard the least "fitting" plans from the population. Afterward, it will merge two and two plans together (reproduction) and evaluate them. When the most "fitted" (cheapest) plan is cheaper than a given threshold or the population is too similar to bother to reproduce the most "fitted" plan is returned from the optimizer [27][28]. This strategy has the distinct advantage of being able to stop at any time and have a working execution plan. And it is therefore far easier to control the optimization time.

2.3.4 Sellinger: The Classic Join Ordering Algorithm

The Sellinger join order algorithm from the original system R[7] is the basis for most commercial optimizers and is the most typical example of a bottom-up optimizer[6][25][26]. The approach in Sellingers join order algorithm is simple.

(1) For i = 1, ...,N
(2) For each set S containing exactly i of the N tables
(3a) Generate all appropriate plans for joining the tables in S
(3b) considering only plans with optimal inputs
(3c) retaining the optimal generated plan for each set of interesting physical properties

It first finds the sub-plans with only one table (i=1) (access methods). For each table, it keeps the best-cost access method (and the best option giving an interesting order if the query result is to be in a particular order). Then it constructs all plans with two tables in them (i=2) and keeps the best-cost (and best interesting order option) for each table combination. Then it constructs plans with three tables (i=3) from the plans with two tables and an access method plan. Again it retains and prunes plans on the same criteria.

Next, it makes plans with 4 tables in them (i=4). These are either made from combinations of one access method and a three-table plan from the previous step or constructed from two two-table plans. When the four-table plans are made and evaluated it starts to make five-table plans and so on. The algorithm finishes when it has made plans that join together all the tables in the query. Then it picks the best one and sends it to execute[7][25][26].

This exhaustive search will generate and evaluate a large number of potential plans and maintains very many plans at a time. Therefore three heuristics have been proposed to prune the search space.

- Selection-projection heuristic: selection and projections(trim columns) are not really processed by the optimizer, instead it simply adds them where they fit. They do not affect plans in a negative way and should be put there anyway. This means that adding selections and projections does mean adding new plans to maintain for the dynamic programming algorithm, saving a lot of memory space[25].
- Cartesian product heuristic: Never consider Cartesian products first. This has two benefits. One, when there are no obligatory Cartesian products in the query, one avoids creating the near-always bad plans. The second is that when there is a Cartesian product in the query, it is automatically placed at the top, where it does the least amount of harm[25][26].
- Tree form Heuristic: Typically only create plans that are left-deep. This is a somewhat controversial heuristic because it will likely prune away the actual optimal plan. However one has found that there will typically be a left-deep plan with comparable performance to the best bushy plan. It is implemented in most commercial DBMS[25][26].

Keeping only left-deep plans in particular is a very powerful tool to reduce the processing significantly. It means only creating and maintaining plans that have the shape of a left-deep tree. In 2.5 one can see what a left-deep tree looks like, all left-deep trees follow the pattern shown in the figure. On the right side of the figure, one can see an example of a bushy tree.

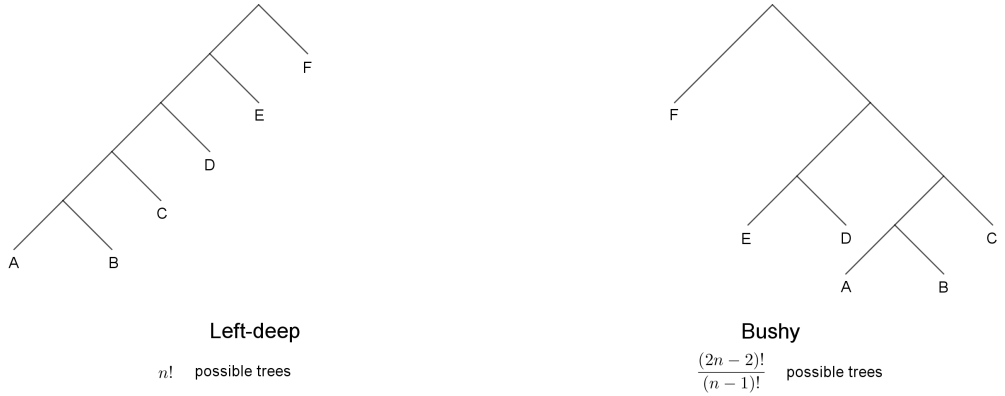


Figure 2.5: Results join order hint non-strict 1-5

Bushy trees can take any shape that a binary tree can. The bushy trees have significantly more possible combinations than a left-deep tree. This can be seen in 2.1, where for each table/leaf added the potential combinations explode. But far more for the bushy tree. Creating and processing every decent possible combination for a bushy tree is very costly when many tables are involved. Therefore the loss of optimality caused by only using left-deep plans is accepted to have an acceptable optimize time[25].

No. of Relations N	**No. of Left-Deep Trees N!**	**No. of Bushy Shapes S(N)**	**No. of Bushy Trees $\frac{(2N-2)!}{(N-1)!}$**
2	2	1	2
3	6	2	12
4	24	5	120
5	120	14	1,680
6	720	42	30,240
7	5,040	132	665,280

Table 2.1: Number of unique join trees for left-deep and bushy trees[8]

These heuristics can however, and especially the tree form heuristic[25], hinder the algorithm in finding the actual optimal plan (according to cost), but ensures that optimization takes far less time, and memory meaning it can function for a larger plan space[25][26][7]. It can still find situations where it reaches the capacity of memory and/or CPU. In those cases one has two options: limit the complexity of queries by allowing only a certain number of tables in a query. Or change strategy[6].

Iterative Dynamic Programming, for Sellinger

When Sellinger reaches the limits of the system, IDP(Iterative Dynamic Programming) is a way of coping with this. IDP uses the normal dynamic programming algorithm (for example Sellinger) until it threatens to exceed its resources (usually memory) on the next iteration. For example, if it has succeeded in generating all possible plans for join of 10/k tables, but making the plan for 11/k+1 tables will crash the system. This is When IDP steps in. IDP chooses the best available plan with 10/k tables as a base. Then it removes all plans that contain any tables in the chosen base plan. When this is done it restarts the dynamic programming algorithm, but now the chosen base plan is treated like a table access method. This effectively prunes the search space dramatically

and allows the optimizer to continue working. It does however remove a lot of potentially good plans from the search space[6].

It is worth noting that this approach naturally can be used for other optimizers using dynamic programming.

2.3.5 The Optimizer Failing at its Job

A lot of things can cause the optimizer to make bad decisions that result in bad plans. The main sinners are the statistics and the cost model. Wherein flawed cardinality estimates based on the statistics are the biggest sinner[29][30]. The effect of these factors failing can be plans with catastrophic performance[5][29][30].

Statistics Misguiding the Optimizer

If the statistics are wrong/misleading the optimizer will make decisions based on wrong assumptions. And since statistics is an estimate based on sampling, this can happen rather often[29][30][9][31].

The function of the statistics is to give the optimizer an idea of the size of the tables and the cardinality to be expected on different values in the table (rows that have the given property). It does this by sampling the table, and assuming that the trends it sees are representative of the rest of the query [30][29] or sampling in runtime during the optimization, this is done to find the cardinality of joins [29].

When the statistics are wrong, the optimizer will assume wrong about cardinalities. When that happens it can begin to make mistakes, as a wrong estimation in one place easily can propagate to other parts of the plans where that cardinality estimate is used to estimate the cardinality(number of rows) returned by joins and so on. And the smallest variation in statistics can cause issues[30][29].

For instance, if the optimizer assumes low cardinality for a value in table A, it might choose an index lookup for it, even if this is a bad choice. It would also be likely to join table A early and thus the under-estimation propagates up the tree. Optimizers are also notorious for their tendency to assume whether the join results will be large or small. They usually make very simplified assumptions on the cardinalities of joins when they do not have statistics on the join itself[29]. Which is not common as of now.

The optimizer does however rely heavily on statistics, and so flaws here will often lead to suboptimal or outright bad plans[30][29][9][3].

Cost Model Flaws

A faulty cost model is a big issue, but the importance of this is in practice overshadowed by the more critical cardinality estimates[30]. Having said that the cost model does factor in.

Typical flaws of the cost model are constants being calibrated wrong in regard to the actual hardware. An example of one such constant would be the cost of a single disc I/O. Another flaw is the cost models inability to actually calculate perfect estimation of time on the algorithms used in execute. Runtime parameters and cache hits are another factor the cost model cannot perfectly assume. And on top of that, concurrent running of multiple queries will affect runtime too(locking, contention on indexes and more)[9]. Or changes to the tables [31]

2.3.6 Skipping the Optimizer: Plan Caching

Despite optimize usually taking comparably little time when compared to execute however, it can induce a large CPU load, and use a lot of memory. And so it can be an advantage to skip the

optimizer when possible. This is done by maintaining the query execution plans from optimization and applying them when a query is issued again, skipping the optimizer entirely. This is called plan caching. Like any other cache, it only maintains a limited number of plans, but it can reduce the CPU load a lot [32].

The caching of plans is not necessarily as amazing as it seems at first glance. In some cases the plan that was cached can simply have stopped working because of changes in the database, if columns have been added or removed for instance, or if indexes have been deleted, and if data has changed so much the statistics are updated. This is usually handled well by the systems and does not pose a big problem, the system removes the outdated plan. On the next run of the query, the optimizer makes a new plan.

The bigger problem occurs when the optimality of a query plan is dependent on data distribution in its parameters. In this case, most systems will simply reuse the plan in the cache which can result in very long execution times. For example, if a query lists all employees of a given role and all their paychecks. The optimal plan for fetching the top leaders will not necessarily be optimal when fetching all factory workers. If the leader one is cached, it can use very ineffective strategies to process the many factory workers. The smarter systems implementing a plan cache compensate for this, by caching multiple plans for a query when there is significant skew in their parameters. This does not however mean that they find the optimal plan for every parameter combination, just that they ensure that it is not too bad[32][33][34][11].

2.3.7 Short Circuit the optimizer: Plan pinning

Plan pinning is forcing (pinning) parts of, or the entirety of a query execution plan for a specified query. It manipulates the optimizer into choosing a plan with the pinned qualities. In other words, it is a tool to override the query optimizer to make it provide plans with desired properties. Even when it would not make such a plan when not pinned. Plan pinning is utilized in cases where the optimizer makes badly performing plans to force the optimizer into providing a better plan.

Plan pinning is not an amazing silver bullet though, as it also stops the query from having plans that are adapted to different data distributions. A problem it shares with plan caching. However, unlike with caching the fix is not so trivial. Because plan pinning is already there because the optimizer did/or could do something stupid, one cannot simply re-optimize if the plan is bad for new parameters. The pinning is used in cases where one does not trust the optimizer's judgment. This is why plan pinning is dissuaded by most DBMSs and why some systems provide means of pinning multiple plans for a query, typically through SQL plan management.

The primary issue is that the optimizer will choose the pinned plan regardless of the parameters. This means that for the best possible performance with a single pinned plan, the DBA must find a plan that has acceptable performance for all possible parameters even when the data is skewed. This will often lead to sub-optimal plans for some parameters[34][5][11]. An undesirable consequence of plan pinning [5]. It is also a big factor that a pinned plan cannot adapt to changes in data distributions and as such the hints who worked well at one point might cause great performance loss when data changes[10][34][5][11]. This has not stopped DBAs from using plan pinning both to avoid bad plans and/or plan regression (degradation of plans), the trade-off is worth it in some cases[2].

Optimizer hints

Optimizer hints are mechanisms most mature DBMS provide to nudge the optimizer to make certain choices. Optimizer hints can for instance dictate join order, force the optimizer to use or not use certain indexes, and use certain access methods on specific tables. Hints have the advantage of only hinting parts of the plans, but simultaneously the weakness of being unable to pin an exact query execution plan (not enough hints to cover everything)[9].

Optimizer hints are also not too powerful in that if they do not work, they simply are not heeded.

In other words, if the hints do not work the optimizer simply overlooks the hint. This means that hints cannot crash the system in any way, but can be overlooked[35].

Optimizer hints are usually inserted into the query text itself (usually within comments) [9]. This can be done manually or by a plugin or by the DBMS itself when hint-based plan pinning is used[13][35].

Hint-based plan pinning

The simplest way to pin a plan is to provide optimizer hints along with the query. This is also the most commonly provided method. However, this method forces the system developers to provide the query hints with the query text, making it less than ideal for maintaining application code.

Some systems provide pinning tools that place the burden of applying hints on a plugin or the DBMS itself. In the cases where the DBMS applies the hints, it is explicitly called plan pinning [13].

Plan pinning with executable query plans

Most systems use hints in some way or form to pin, but there are those who keeps actual query execution plans for pinning. It is far more tricky to maintain as its dependencies can change. And in these cases, a re-optimize is more problematic than when caching. Because the reason for the pinning is that the optimizer produces un-desired plans. So if one is forced into a re-optimize the pinning stops working. (The alternative to a re-optimize is a crash.) However, This method has the distinct advantage of effectively caching the plan, and the plan pinned will be a one-to-one mapping to the plan the pinned optimizer produces. Several systems come close to this but do not do this exactly [36][14][5][37][12]

2.3.8 Plan pinning and Plan caching, two sides of the same coin?

Despite having two very different purposes plan caching and plan pinning has significant overlap. To begin with, caching a plan for a query effectively pins the plan to the query as a side effect. But plan pinning also shares other traits with plan caching. When plan pinning keeps a full execution plan, it effectively caches the plan, allowing it to skip optimization. With hint-based pinning, it does not keep the entire plan, but the optimization can be faster as the optimizer does not need to make as many decisions as normal, effectively "caching" the decisions [38].

The biggest difference between them is intent. The plan cache is there to make optimize go faster on average, so an occasional re-optimization is fine. Plan pinning on the other hand does not shy away from spending time optimizing (unless they pin full plans, stored in execution plan form), some plan pinning strategies even optimize multiple times[5]. Plan pinning is there to manipulate the finished execution plan and the goal is to improve performance on the execution step, not the optimize step. They do however share a weakness in data skew, and changes to metadata.

2.4 Hypergraph

2.4.1 Hypergraph the graph

A hypergraph is an undirected graph in which hyperedges may appear. A hyperedge is an edge that may connect more than two nodes. As seen in the image, the hyperedge connects more than one node. It is worth noting that a hypergraph can exist without a hyperedge (a simple hypergraph), though that means it works like a normal undirected graph until it gets a non-simple hyperedge.

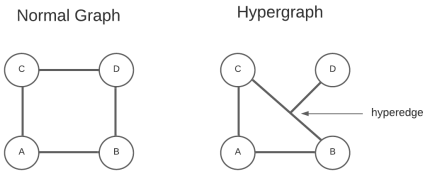


Figure 2.6: a demonstration of the difference between a normal graph and a hypergraph[17]

2.4.2 Hypergraph as a Representation of Joins

The usage of hypergraphs in optimizers comes from its ability to model join orders and table relations. The Nodes/vertices represent tables, or specifically the plan for table access like index lookup and table scan. And all edges represent join predicates that apply between the tables they connect. The image below illustrates an example of this.

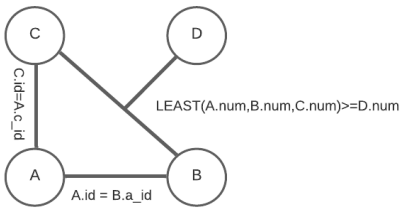


Figure 2.7: Example of possible join predicates for a hypergraph [17]

It is worth mentioning that any predicate can be presented as hyperedge. For example, $B.num + C.num = D.num$ could be the predicate for the hyperedge in the image and it would still be correct. $B.num = D.num - C.num$ is equivalent to the previous example, the article uses a more complex description of hyperedges to be able to use this. The complex hyperedges have three sets of nodes instead of two, where the third can be on either side. In other words, if the hyperedge has elements that can be on either side of a predicate through commutativity they have their own placement in the hyperedge to denote that they can be on either side. This is not shown in the pseudocode further down, but adding it is simple.

2.4.3 DPhyp

DPHyp is a new alternative to Sellinger that manages to optimize queries in an acceptable time without the left-deep heuristic that is used in most optimizer implementations. Meaning that this algorithm can create bushy tree plans within a reasonable time. [39]. It accomplishes this by utilizing a hypergraph to represent its joins.

Important Concepts

Connected Subgraph csg: This is a part of the full graph where all the nodes in the subgraph are connected by an edge and none of the edges connects the subgraph to a node outside the subgraph. In 2.7 possible connected subgraphs are $A, B, C, D, \{A, B\}, \{A, C\}, \{A, B, C, D\}, \{B, C, D\}$ (ordered list). Ones that would not work are $\{B, C\}, \{B, D\}, \{C, D\}$ This is because the hyperedge would reference a node that was not in the subgraph. Any variation including A and D without both B and C. This is because A cannot be connected to D without the other two.

Connected complement cmp: The connected complement of a connected subgraph S_1 is any

connected subgraph that includes none of the nodes of S_1 but is still a subgraph of the same graph as S_1. An example using the figure 2.7 would be $S_1 = \{A\}, S_2 = \{B, C, D\}$, or $S_1 = \{A\}, S_2 = \{B\}$.

csg-cmp-pair: This is two connected subgraphs (csg) that are each other's connected complements (cmp) that can be connected by a hyperedge (simple or otherwise) that only reference nodes in the cmp. If the cmp can be connected by a hyperedge without referencing any nodes outside the pair it is a csg-cmp-pair. These are significant in that when these are found the edges connecting them are where the algorithm will insert plans for joins.

Important Symbols and Details

Neighbourhood of node: The neighbourhood of a subset S of a hypergraph G is any allowed nodes that is not S. In DPhyp that means any node "larger" than $Max(S)$ is in the neighbourhood of S. Since

X is a set containing the forbidden nodes that DPhyp are not allowed to process yet. This forbidden set typically contains S_1 and any node 'smaller' than the smallest element in S_1, which means it is smaller than the last 'opened' node. (S_1 is typically the most recent set under consideration).

N or $N(S, X)$ is the neighbourhood of S excluding any nodes in X. Neighbours are nodes reachable from S.The neighbourhood of the subset S of a hypergraph G is any allowed node that is not S. In DPhyp that means any node "larger" than $Max(S)$ is in the neighbourhood of S.

Meaning that for the hypergraph $G = (V, E) = (\{v_1, v_2, v_3\}, \{(\{v_1\}, \{v_2, v_3\})\})$ the neighbourhood of the node v_1 will be v_2 and then enumeration is used to reach v_3 to get a valid connection. Note that N and N are separate things, where N is the entire neighbourhood N is used to denote one node in the neighbourhood.

This is simple for simple edges and troublesome with hyperedges. The strategy chosen here is to give the smallest node for the 'right' side of such hyperedges. Because everything is done in decreasing order, we get the smallest node when the hyperedge points to multiple nodes. This means that by the time DPhyp sees N in a neighbourhood and N is reached by a non-simple hyperedge all the other nodes referenced by the hyperedge are allowed nodes. This saves processing.

dpTable The dynamic programming table contains the plans of the nodes(table access) in the hypergraph and the plans for combinations of nodes (joins). The addressing in the table is based on a bitmap or bit vector, where each node(table) is represented by its own bit. Thus when two nodes n_1 and n_2 are combined(joined) the plan for that combination(join) will be at address $n_1|n_2$ in the *dpTable*.

Example: $G = (V, E) = (\{v_1, v_2, v_3\}, \{(\{v_1\}, \{v_2\}), (\{v_2\}, \{v_3\})\})$ where v_1 has the address 001 v_2 has the address 010 and v_3 has the address 100. Then the subplan/subgraph with v_1 and v_2 has the address $v_1|v_2 = 001|010 = 011 = 3$. And the address of the plan to join v_1 and v_2 is stored at $dpTable[3]$. This means that each combination will have its own unique address like this. If there exist multiple possible subgraphs with the same address (i.e. uses different edges to connect the subgraph of the same nodes) the best known plan for that combination will be stored in that spot in the table.

2.4.4 DPHyp: Bottom-up Hypergraph optimizer

This algorithm exploits the innate properties of a hypergraph. When the hypergraph representing the join options is made the DPHyp algorithm will start by first finding plans for all the nodes/access method plans and putting them into the dpTable at their designated spots (the bit value of the table (one bit): 100-¿4). Then from the last to the first table, the DPHyp will look at an expanding number of nodes, starting with the last node and including the first last. For each new node, DPHyp will try to make csg-cmp-pairs with the other nodes it has seen (its neighbourhood). Csg-cmp-pairs are valid connections that can have a valid join between them. Each part of the pair is a sub-plan (a single table or larger connected subgraphs). DPHyp ensures that the

plan for the sub-plans a csg-cmp-pair consists of already exists, and so the plan to join them can be made(, and the two elements in the csg-cmp-pair become its children).

Note that a subgraph can have multiple possible plans, joining tables A, B and C can be done in multiple ways (if the hyperedges allow it) and the best found plan for the subgraph will be in dpTable[A — B — C] (for example dpTable[001—010—100]=[dpTable[———]=dpTable[7]]). by the time tables A, B and C have been enumerated by DPHyp the subgraphs best possible plan lies in the dpTable. This also means that one does not need to create it again and can use the sub-plan as is for future enumerations in the graph. When the last table is enumerated the resulting plan will be at dpTable[sum_of_all_table_bits] and this is the finished full execution plan. When the enumeration is done the best plan for the entire graph lies there.

2.4.5 Top-down DPhyp

There has also been proposed a Top-down algorithm for hypergraphs. This algorithm is designed to be usable for any pre-existing partitioning algorithm for simple graphs. It does this by making the hyperedges appear like simple edges. They accomplish this by finding sub-graphs that cannot be separated because of obligatory hyperedges and compounding these into a single node. An obligatory hyperedge is a hyperedge that has no simple edge alternative, and where removing it stops one or more of the nodes connected by the hyperedge to be unable to connect to the rest of the graph. Then this obligatory hyperedge with all its nodes will be combined into a single node. When this process is done, only a simple graph is left, and there are fewer nodes to process. This way, when the hypergraph enters the partitioning algorithm it is a normal simple graph and can be processed by the algorithm[40].

2.4.6 DPhyp in MySQL

MySQL has in many ways implemented DPhyp directly as it is described in the paper [17]. The most notable difference is that MySQL chooses to maintain multiple plans for each entry in the *dpTable* (node (table access) and edge (join)). This is done to avoid discarding plans with traits that might open up for better plans further into the process. The system that decides which plans to keep is essentially a skyline.

Skyline

A skyline is a set of interesting values. Interesting values are values that are not worse than any other value in all ways(dimensions) and better in at least one dimension. The term skyline comes from the intuitive picture of the concept, a literal skyline, where everything you see is in the skyline and everything you don't see is not. Finding the skyline in data processing is also known as the maximum vector problem [41].

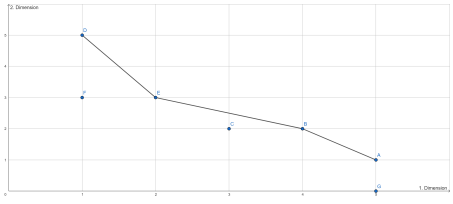


Figure 2.8: example of skyline

To simplify: any data point/value that is not dominated by any point/value in all dimensions. One point dominates another if it has the same or higher value in all dimensions and a higher value in at least one dimension. In the figure, one can see that E is in the skyline even when D has a higher value in the 2. dimension and C, B, and A has a higher value in the 1. dimension. This is because

no data point has a higher value in both dimensions simultaneously. On the other hand, C is not on the skyline. This is because the data point B has the same value in 2. dimension and a higher value in 1. dimension, which means that B dominates C. Similarly F is dominated by both D and E, and G is dominated by A.

Note that a higher value can also mean the smallest value, for instance, a dimension for price where the cheapest is the most desirable. Then a low cost will equal a high score/value for the skyline.[41]

Intuitive Example A skyline also has more intuitive uses. If one changes 1. dimension to tasty and 2. dimension to healthy one would have a way to evaluate cereal, dinner alternatives, and much more. The dimensions can be anything that can be quantified or confirmed (boolean value). And there can be any number of dimensions. The cereal example can have a third dimension cost, that prefers the cheaper ones. Dinners can additionally have a dimension quantifying preparation time, and so on.

Subsuming Dimensions as a Concept Another possibility though not commonly used is a super-dimension that subsumes the others. Using the dinner skyline from the last paragraph, one could add another dimension: allergen. If a dish contains an allergen that will cause harm, it is instantly excluded. This can be modeled such that any dish containing dangerous allergens would always be dominated by a dish without any of the allergens. In other words, any dish without allergens will always dominate those with them regardless of the other dimensions. The allergen dimension subsumes the other dimensions in that it dominates regardless of other dimensions. The skyline for the safe dishes will however behave as before, just that only safe dishes will appear in it.

Skyline in MySQLs Hypergraph Optimizer: The Tournament

The skyline in MySQLs hypergraph optimizer will use different sub-plan traits as dimensions. This skyline is implemented as a tournament of sorts where every sub-plan competes against all alternative plans for the same operation for the same tables. For example Table Scan on table A, Index Range Scans on table A and Index Lookups on table A will compete against each other. However, an Index Lookup on B will only compete against other access methods for B.

MySQL has eight dimensions in its skyline and as such might maintain up to eight plans for every operation. This is however rarely the case as in many cases plans will dominate others as they have several of the traits and other times some traits do not come into play at all. But for each operation, MySQL's hypergraph optimizer will maintain between one and eight sub-plans in the *dpTable*. An important point for this thesis is that these skyline dimensions decide which plans are kept, and the number of dimensions determines the maximum number of plans maintained for each dimension.

One such dimension is "sorted". If a sub-plan has a trait "sorted" and another has a lower cost both will be kept as they do not dominate each other in both dimensions. Maintaining plans that maintain ordering ("sorted") is advantageous because plans built on these might avoid explicit sorting later on, possibly making a better plan despite initially having a higher cost than their non-sorted counterpart. And so an index range scan can pass through the tournament despite having a higher cost than a table scan.

The eight dimensions behave somewhat differently, four of the dimensions behave exactly as stated where they are either better than their competitor or not. The other four have an additional possible trait "slightly better" in addition to "better". If none of the two plans really differ much in these dimensions the "slightly better" flag is set rather than "better". First, they compete only with their "better" flags. If they are identical in those dimensions the "slightly better" trait will be tested and make the plan with the trait win. This means that the "slightly better" trait can only beat a plan that is otherwise exactly as good as the plan with the trait. If one of the competing plans dominates or they have different strengths without it, the "slightly better" trait does not come into play.

This skyline evaluation will for the most part be referred to as "the tournament" as it depicts how the sub-plans will be evaluated against each other and discarded when dominated/defeated. This is to make it more intuitive to read.

Chapter 3

Related Work

3.1 Plan Pinning in Existing Systems

This section will look at different approaches to plan pinning in existing DBMSs. It will start looking at optimizer hints in section 3.1.1, then look at hint-based plan pinning that utilizes such hints. Lastly the more intelligent SQL plan management solutions will be presented in section 3.1.3.

3.1.1 Hints to Pin

Optimizer hints allow one to pin parts of the query plan, but can usually not force a specific plan as the hints might not cover all possible decisions the optimizer might make during optimization. It does however nudge the optimizer to make desired decisions. Most systems that provide hints do so for the most critical parts of the query planning and so they are very effective.

Noteworthy Systems with Hints

MySQL, MariaDB, and EnterpriseDB are the databases that are covered that have hints, but not hint-based plan pinning[35] [42][43][44]. Otherwise, the systems that have hint-based plan pinning have hints.

Notewothy Systems Without Optimizer Hints PostgreSQL

PostgreSQL has chosen not to have hints or plan pinning as the PostgreSQL community considers it a bad idea to influence the optimizer in that way[45][18]. PostgreSQL does provide features that can do similar things to hints called planner method configurations, but not pr. query [46]. And people have found workarounds like using left join in place of inner join to get better row counts and many more[47].

There exist workarounds to get hints like pg_hint_plan made by NTT to create hints for PostgreSQL [48]. In addition, pg_hint_plan is in use in the enterprise version of a new system forked from postgreSQl called Postgres Professional Enterprise [49], and the PostgreSQL compatible database Aurora [50]. EnterpriseDB, another PostgreSQL-based system has also implemented hints[44].

Aurora has also found workarounds by utilizing for instance planner options (planner method configurations) to produce a desired plan they then make QPM pin that plan. They turn then on and of pr. query[3].

3.1.2 Hint-based Plan Pinning

Hint-based plan pinning, as the name suggests, is a pinning strategy that utilizes hints to pin plans. The basic idea is that a pinned query will have a set of hints stored in the system for the query. When a pinned query is issued to the system the system recognizes the query using a query id(, usually a query digest made from the query text). Then it applies the stored hint set corresponding to the query. These hints are applied to the query before optimizations so they influence the optimizer into making a plan that corresponds to those hints.

This method can only guarantee a 1 to 1 mapping between the intended plan and hinted plan when the system has enough different hints that they can force every aspect of the plan [13][37]. But is also safer in the sense that it will not fail to produce a plan if the optimizer overlooks hints if they do not work, meaning a breaking update will only limit the pinning, not make the queries fail[35][13][37].

A small beneficial effect is that it saves the optimizer from having to evaluate the cost and make decisions where the hints dictate its choices. This can save some optimize time when the plan space becomes large [38].

Oracle Plan Pinning

Early on Oracle implemented something they called stored outlines to pin queries[14]. A stored outline is a collection of hints stored in the DBMS that is applied to its corresponding query before optimization[36][14]. A query's outline is generated by the system based on the output from the optimization of the query at the time of generation. This means outlines have to be generated at a time that performance is good[36] and pins a whole query execution plan (almost). An outline can be rebuilt for the new conditions in the system at any time[16]. Stored outlines were deprecated in Oracle 11g and replaced with SQL Plan Management[5].

SQL Server

SQL Server currently has two plan pinning strategies, the first is a typical hint-based pinning and is called Query store hints. A big difference between Oracles stored outline and Query store hints is that query store hints are not generated but applied by the DBAs themselves. Meaning that they can pin a different plan than what the optimizer comes up with and that one can pin only parts of the query execution plan[13].

The second strategy pins an entire execution plan, however, it does not store the exact execution plan. It is actually an optimizer hint. The hint takes the query execution plan in XML format and recreates the plan from the XML. This is not a guaranteed 1 to 1 mapping either as the XML does not fully dictate a plan[37].

An honorable mention would be forceplan which is a SQL syntax that forces a specific plan, this plan is not very dynamic however and forces the join order to be the same as the order the SQL is written and forces nested loop join for all joins where it can be used. The modern methods are considered better as they have more options to ensure good plans[12].

PolarDB

PolarDB for MySQL has implemented hint-based plan pinning. They call their implementation statement outlines. Statement outlines apply hints to queries based on the query digest. The hints applied are the same hints that MySQL already provides so this can be considered a built-in rewrite plugin for the MySQL nodes. Because this system is built on top of MySQL it becomes particularly interesting for this thesis, however, they do not add much more than functionality to have the system add hints automatically when the query is issued[51].

Note that PolarDB for PostgreSQL does not have this feature as PostgreSQL does not have hints.

3.1.3 SQL Plan Management

SQL Plan Management (SPM) is a sophisticated way to pin plans that prevent plan regression while also allowing plans that perform better to be allowed. Plan regression is the term for optimizers producing different worse plans for a query than before, typically because of flawed statistics or updates to the cost model or the optimizer as a whole, or other changes in the database.

SPM is designed to avoid plan regression while simultaneously getting all gains from an updated optimizer and adapting to new data distributions. It does this by letting the DBA **pin multiple plans** for a query, and adding well-performing plans (often automatically) to the allowed plans for the query while stopping any other plans from being executed[5][31].

The primary idea is that the DBMS remembers which queries are pinned (using a query id/query digest)and store information about what plans the system allows for the pinned query. When the query is run the optimizer creates a plan as usual, this plan is called the best cost plan. When the best cost plan is generated the SPM goes to work. If the query is pinned the tests shown in 3.1 will be run deciding which plan it executes:

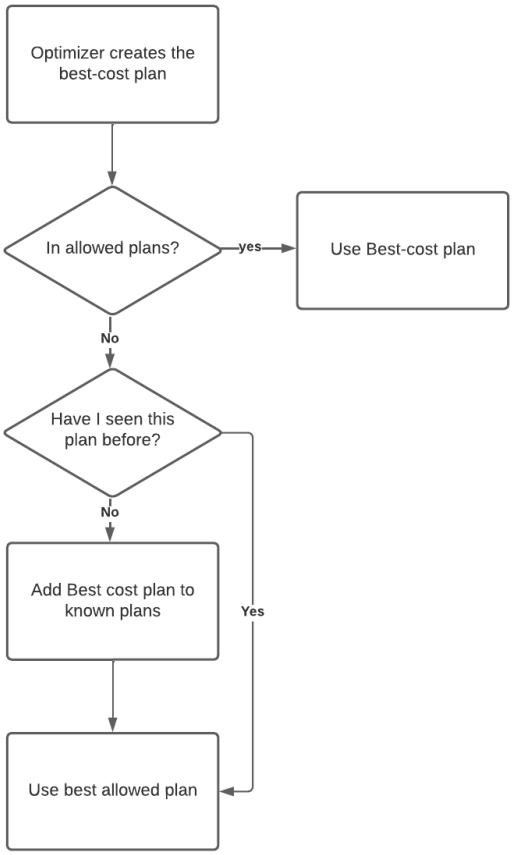


Figure 3.1: simplified figure for SPMs logic[5]

As the figure shows, unknown plans will be stored by the system. This is to be able to add new better plans to the allowed plans. To do this it stores new plans for consideration, in the figure these plans are known plans. At some point, a test is run (automatic or manual) to see whether the new plans perform better than the allowed plans. This allows for new better plans to be added to the allowed plans.

A notable trait here is that the optimizer works as usual, and SPM only takes effect after the initial optimization. This means that an optimization is always performed. The SPM is not meant to save time on optimization, but only to improve query plan performance while protecting from plan regression[5][31].

SPM pins multiple plans, making it particularly interesting as it sets precedence for how pinning multiple plans can be done and what trade-offs are accepted. SPM establishes that added overhead can be acceptable and that the quality of the plan takes priority over fast optimize times. SPM typically allows at least two optimize cycles, one ordinary cycle and another when the generated plan is not an accepted plan. SPM also stores whole plans (at least in principle) setting precedence for this when pinning multiple plans as well[5].

Oracle SPM

Oracle implemented their SPM for their 11g release 1. From then on SPM replaced stored outlines and became Oracle's new pinning strategy[52]. The stored outlines could be migrated into the baseline (allowed plans).

Oracles SPM has automatic baseline evolution which when enabled runs a test in the background when the system has extra resources to see if any of the new plans should become an allowed plan. The test runs the query with the new not-allowed plans and with the one the SPM chose, if the new plan is significantly better, i.e. has a significantly better actual execution time it is added to the allowed plans (baseline).

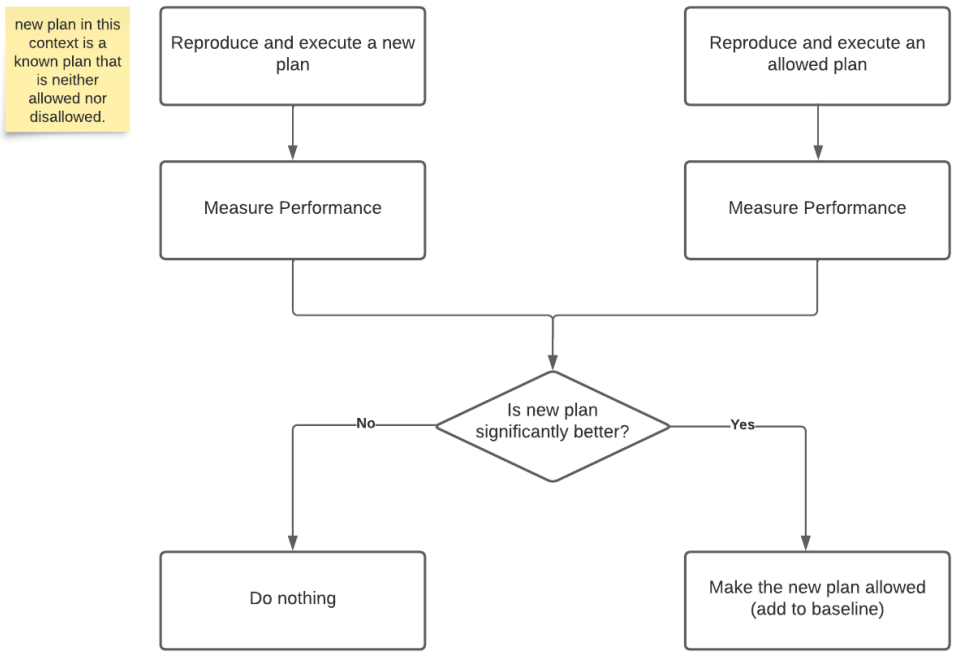


Figure 3.2: simplified figure for Oracles SPMs automatic plan evolution test[5]

It is also worth noting that Oracles SPM does not store the actual plan, but stores optimizer directives that allow the optimizer to reproduce it. And so when the SPM receives a not-allowed plan it has the optimizer reproduce all the allowed plans and cost them. Finally executing the cheapest of these reproduced allowed plans. If the reproduction fails it uses the best cost plan from the optimizer [5].

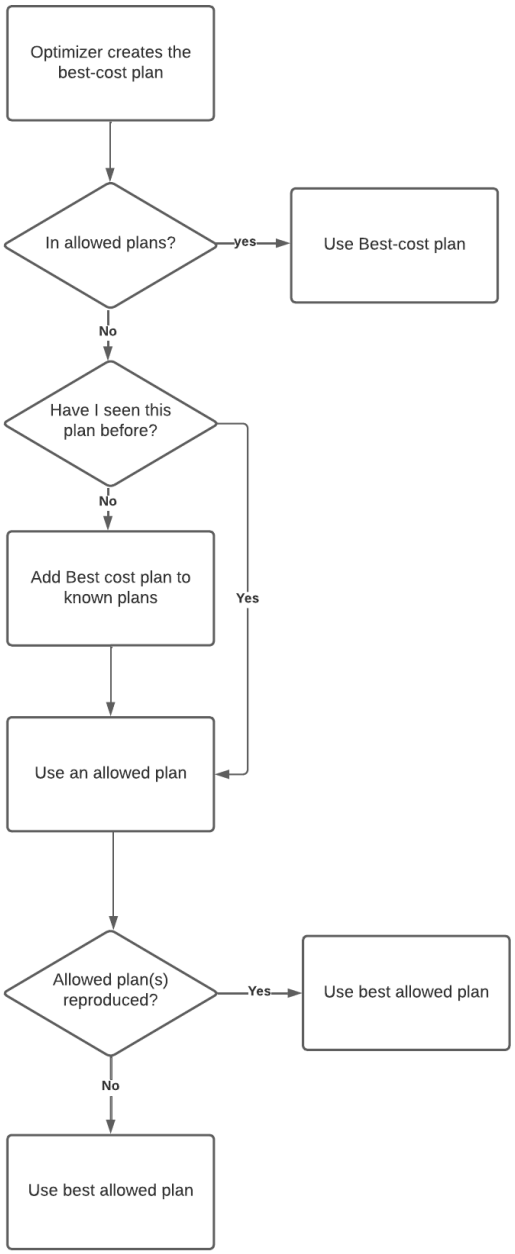


Figure 3.3: simplified figure for Oracles SPM logic[5]

Oracle even suggests its SPM as a preventive measure for plan regressions after updates. Set up plan management for all critical queries and capture baselines, or make stored outlines and then update the system. SPM then protects from plan regression upon system update. And automatic baseline evolution will ensure that the system gets all the gains of the update. It is shown to give good performance and so the tradeoff is acceptable[5]

3.1.4 Aurora QPM

Aurora's variation of SPM, known as QPM (Query Plan Management). It works much like Oracle's SPM [5][31]. This feature is only for Aurora PotgreSQL, (not MySQL), but does much the same as SPM. One can capture plans automatically or one can capture them manually to have lower granularity allowing one to only pin a few problematic queries or even problematic literals or

parameters[31]. Using pg_hints it also allows the DBAs to tweak plans and capture them. One can turn off the "hints" and still get the hinted plan.

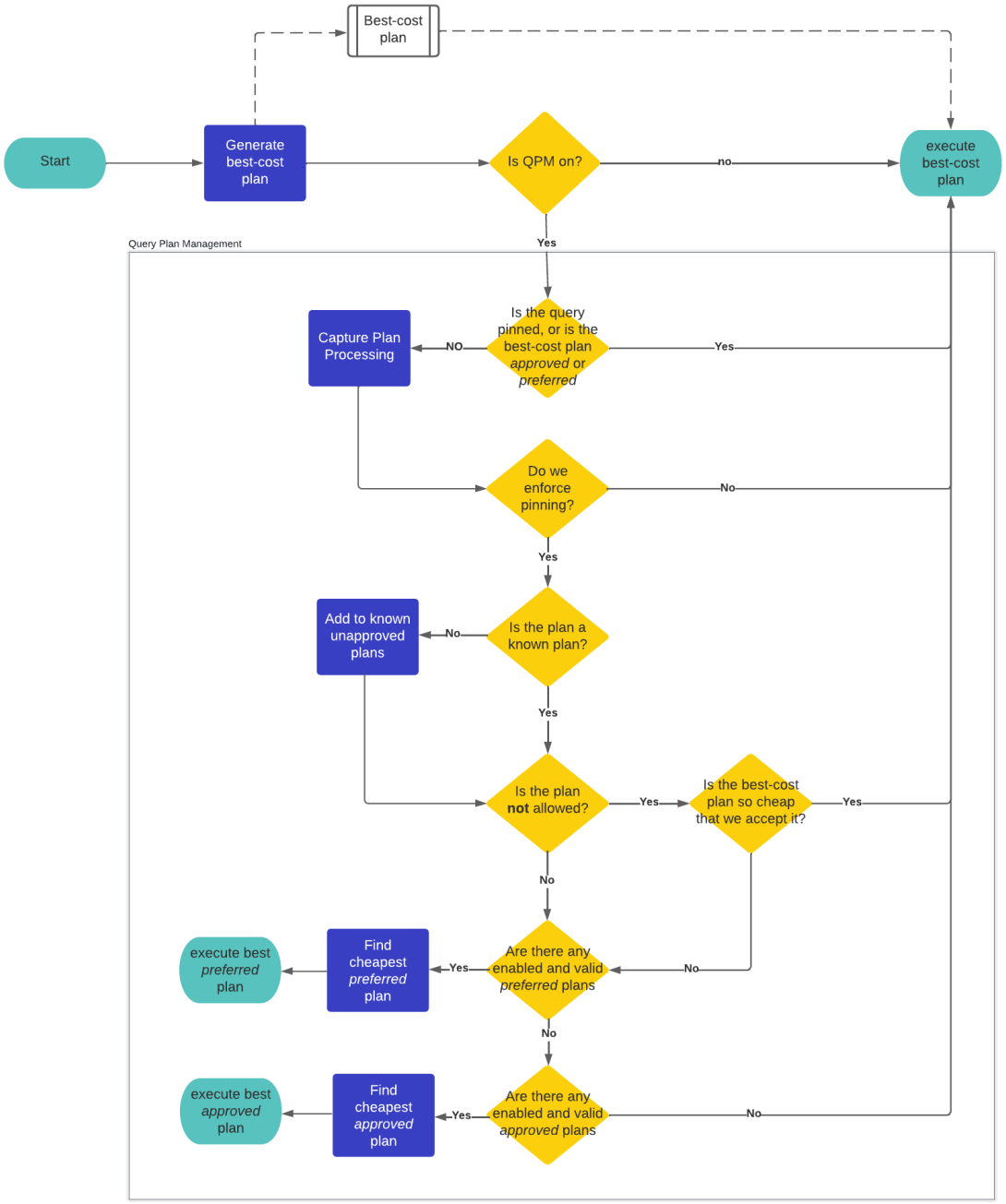


Figure 3.4: simplified figure for Auroras QPM logic[31]

3.2 Plan Caching

This section will cover plan caching in existing systems, and how they work. This section will be heavily inspired by the work done in the preparatory project in the fall of 2022 [38].

3.2.1 PostgreSQL

PostgreSQL has implemented plan caching for prepared statements. This implementation compensates for plan caching's issues with data skew, by having it be optimized the first five times, and then make a "generic plan" which is meant to work on most cases. When a generic plan is made, subsequent runs of the query will be able to use this generic plan from the cache. They do however cost it for new parameters so that if the new parameters cause the generic plan to be bad, in which case it optimizes for that run[53].

3.2.2 PolarDB

PolarDB effectivily inherits the plan cache capabilities of the systems it uses, so PolarDB for MySQL does not have a plan cache while PostgreSQL does[54].

3.2.3 PolarDB-x

In addition to the functionalities of PolarDB, it also caches the plans for distributed operations. This is related to the fact that PolarDB-x uses a cluster of PolarDB instances[55][56].

3.2.4 Oracle

The DBMS Oracle has a rather sophisticated plan cache that takes data skew into account. Firstly, the default behavior is that a plan is pinned with its literals, and so for non-parameterized queries, the same query with different literals will have different entries in the plan cache. It is possible to change this behavior, by enabling cursor sharing. This is not recommended for non-parameterized queries, however. It includes the DBMS turning the literals into system-generated bind variables during parsing.

Cursor sharing is essentially allowing a query with different parameters (system- or user-defined bind variables) to have multiple plans for the same query. This is done with data structures called cursors. For a cached query there is a parent cursor that matches a query, the parent has one or more child cursors containing plans. The children can be the same query for different bind variables or the same query across databases.

For parameterized queries, on the other hand, this is considered beneficial. Parameterized queries also have the power of adaptive cursor sharing. Adaptive cursor sharing is activated whenever bind value peeking is used. Bind value peeking is allowing the database to use statistics on the parameter values when making plans. When this is used the cached plan becomes parameter-dependent, and is marked as bind-sensitive. And that is where adaptive cursor sharing comes in, allowing Oracle to maintain multiple plans to fit the different bind variable values.

The process for making a query have multiple plans starts after Oracle caches a plan made with bind value peeking where the values peeked are skewed. It is then marked as bind-sensitive. When the query is issued with different parameters, the cursor cached is used. Afterwards, Oracle will check the execution statistics to see if the execution was comparably as good as the initial execution (the one where the plan was made). If the performance is not as good as expected Oracle marks the cursor (of the query) as bind-aware, meaning that Oracle will start considering parameters when it gets a cache hit on the query.

When a bind-aware cursor is found in the cache for a query, Oracle will bind value peek before it uses a query. If the peeked values have so different cardinalities from the original values the plan was made for Oracle will make a new plan for parameters with similar cardinalities to the new query. When there are more plans for a query it chooses the best one for the bind values of each run of the query. If it finds no good plans for a query in the cache it makes a new one to cover the new values. Note that the values can be very different, but have the same cardinality and therefore

be good for the same plan. The factor that decides which plan to choose or whether a new plan should be made is the cardinality of the bind values, not the values themselves.

When the parent cursor keeps multiple plans, it might start keeping identical plans, but for different cardinality ranges on the bind values. Cursor merging will then merge the two cursors with identical plans and mark them so that it is considered a good plan for the ranges in both the old cursors[34].

3.2.5 SQL Server

The solution developed for SQL server uses much the same strategy as Oracle[34][11]. SQL server identifies the most "at-risk parameterized predicates". In other words, predicates whose cardinality dictates the need for different plans the most in the query. SQL server chooses up to three such predicates. Each such predicate gets up to three query variants (plans) totaling up to nine plans pr. query. The plan is chosen based on cardinality as shown in the figure[11].

It is worth noting that SQL server does not store plans directly, but keeps what they call a compiled plan. The compiled plan can cheaply be turned into an executable query plan, so the difference is small[32].

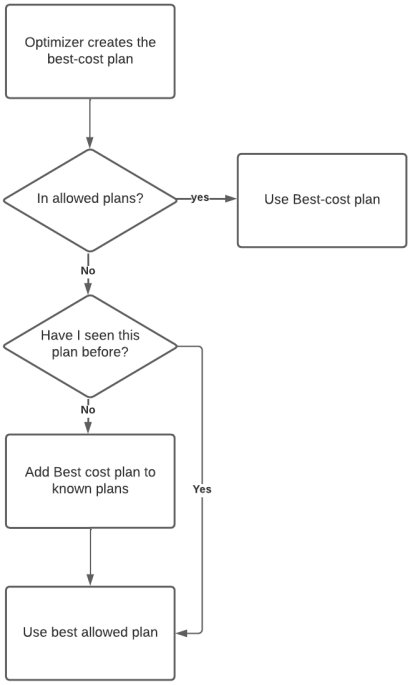


Figure 3.5: Cardinality split plan cache, with cars in country as example [11]

Chapter 4

Implementation

This chapter will look at how the different prototypes for pinning work. It will look at the idea behind the methods, their inherent strengths and weaknesses, and the details of how they are implemented into the MySQL optimizers. It will present two main methods, join order hints and hash-pinning.

4.1 Pinning in the Hypergraph Optimizer

Tournament (described in Background) Pinning in the hypergraph optimizer is done by "rigging" the tournament that selects which plans to toss or use. The non-strict versions effectively give a pinned plan absolute protection, meaning it cannot lose the tournament. The strict version will make it dominate all non-pinned plans, meaning every non-pinned plan outright loses and is discarded.

In more technical terms, the pinning represents its own dimension in the skyline used by the tournament. In the non-strict versions, it simply adds a new dimension equivalent to any preexisting dimension like sorted order. The strict version however practically adds a dimension that subsumes all the preexisting dimensions. If a plan is pinned in a strict implementation it will dominate all non-pinned plans it is compared to.

4.2 The Idea of Pinning Multiple Plans for a Single Query

One of the biggest issues with plan pinning is that a pinned plan does not adapt to changes in the database or to different parameters when there is a skew in the data. This is the primary reason plan pinning is advised against in the documentation of most databases that provide plan pinning. Allowing the pinning of multiple plans for a single query is the most intuitive way to allow a DBMS to both plan pin and work well with data skew. This is shown through the performance of SPM, where they pin multiple plans. And also through the fact that Oracle actually encourages the DBAs to use it as a preventative measure, even though it creates overhead.

The primary issue is that regardless of parameters the optimizer must choose the pinned plan. This means that for the best possible performance with a single pinned plan, the DBA must find a plan that has acceptable performance for all possible parameters even when the data is skewed. This will often lead to sub-optimal plans for some parameters. An undesirable consequence of plan pinning.

Pinning multiple plans for a query is an intuitive way to compensate for data skew, This effectively allows the optimizer to choose between a set of accepted plans, but not those that are not pinned. If one is careful one can successfully make the query avoid the bad plans, while choosing good

options with regard to parameters. This also allows the optimizer some choice between options giving it a limited ability to adapt to updates in data distribution and optimizer.

4.3 Multi-hint

The simplest solution would be to send in multiple hints (of the same type) and let the optimizer choose which one of the hints it uses based on cost. This is not necessarily easy to implement in practice. Since most optimizers generate a single plan and have to choose alternatives before it has the rest of the plan. If the system does not want to keep all hinted plans in their plan space the choice between which hints to heed will have to be taken without knowing the rest of the plan. This might be fine but is not explored in this thesis.

What is explored in this thesis is re-running the optimizer for every hint, letting the optimizer generate a plan for each, then choose the plan with the lowest cost among them.

The prototype only implements join order multi-hints, but the method can be used for other optimizer hints. Since the optimize time is relatively short compared to the execution time in most all cases the re-optimization will typically be an acceptable cost. Especially when the data skew is significant so that the ideal plans for the different parameters differ significantly.

Strengths and Weaknesses of Multi-hint

It optimizes at least as many times as the number of hints (of the same type), and an additional time to re-produce the selected plan. As it does not store the plans themselves but merely their cost. If the last produced plan is the best one this extra optimize iteration is skipped as the last plan is still in the program state. If a single hint is used there is only one iteration.

Among its weaknesses we find all of MySQL hints's weaknesses. Among them is that if the optimizer does not work the optimizer will overlook them. Meaning that if one of the multiple hints does not take effect the optimizers cost model will decide the plan, not the hints. It is enough in this implementation for one of the hints to fail in one iteration of the optimizer, for a flawed cost model to override all other hints. This is because in the event of the optimizer not finding a hinted plan it chooses the best cost plan it finds. Which can easily be the plan one wanted to avoid in the first place.

Another weakness of hints is that the hints are unable to perfectly pin a plan, there is no guarantee of a one-to-one mapping between desired plan and hinted plan no matter how well one writes the hints (with the current hints in MySQL). In addition it demands that the one writing the hints know what to hint. Furthermore the method still has the cost model to choose which plan to execute, meaning that a faulty cost model will still be able to cause trouble.

This weakness of hints is however also its biggest advantage. It permits the forcing of only one or a few aspects of the plan while letting the mostly smart optimizer figure out the rest. Meaning that the optimizer has some freedom to adapt to its environment, while the bad aspects of the plan are removed with hints. This still requires the one writing the hints to know what they are doing, however.

4.3.1 Hint Syntax

The syntax implemented to allow the DBMS to parse a multi-hint is very simple. It allows a single hint (in this project, the join order hint) to take multiple sets of parameters pr. hint. This is demonstrated here, this example takes 5 join order sets, effectively pinning 5 join orders simultaneously.

```
/*+ JOIN_ORDER((join_order_set1),(join_order_set2),
```

```
(join_order_set3),(join_order_set4),(join_order_set5))*/
```

Single does not have the extra parenthesis and works like before.

```
/*+ JOIN_ORDER(join_order_set1)*/
```

4.4 Hint-based Multi-pin for the Current Optimizer

This implementation uses the current hint functionality in MySQL to pin plans, and specifically to pin multiple plans to deal with data skew in parameters. The implementation includes the extended syntax for JOIN_ORDER() which is capable of taking multiple sets of join orders. The Implementation loops over optimize and applies one hint set for every cycle and then chooses the best plan among them to be run. It must often recreate the chosen plan by running optimize with the corresponding hints again.

One of this method's greatest strengths is the low implementation cost, the only thing that has to be implemented is a loop around optimize, some new syntax, and logic to access the different hints in the different iterations. Everything else is implemented already. It has the great advantage of being possible to implement in any bottom-up optimizer. However, to be able to compare against the hypergraph optimizer exclusive hash-pinning in a good way the multi-hint was implemented and this thesis will focus on the hypergraph version.

4.5 Hint-based Multi-pin for the HyperGraph Optimizer

This section will look into how the multi join order hint is implemented in the HyperGraph optimizer. The hint will have most of the same properties as the one for the ordinary optimizer. But it differs in that it accomplishes the same functionality in the hypergraph environment. Instead of choosing the join order directly like in the current optimizer it will make the hint-compliant plans seem so good that they will never loose the tournament. It is also worth noting that this new hint is capable of forcing bushy plans.

4.5.1 Implementation

The general idea is the same as for the current optimizer, have multiple join order hints that will nudge the optimizer to make the desired choices when deciding the join order. Thereafter it will re-optimize with another hint if there are multiple. The implementation of the join order hints is where this implementation will truly differ from the one in the current optimizer. This is because the Optimizers are fundamentally different.

The difference occurs because of the hypergraph optimizer that will propose bushy plans. This complicates the layout of a join order hint. A hint: **JOIN_ORDER(t1,t2,t3)** will have a deterministic interpretation in the current optimizer. But when bushy plans are introduced, formats that can represent the possible join orders become necessary. An example of this is demonstrated in 4.1.

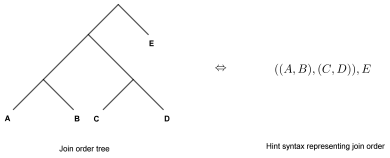


Figure 4.1: Tree to join order hint translation

The correct syntax for join order hints can be found in the appendix.

The chosen way to represent the hints is to generate binary trees. These trees have edges representing joins and nodes representing tables. This works well because that is the form the finalized plans will have too for its join too. The trees will exploit the optimizers representation of tables, that is a bitmap. This representation is universal to DPhyp implementations and allows the usage of fast subset enumeration, it is also a common format in other systems, so the method can be transferred easily. The logic is that if the path's join order perfectly matches a sub-tree in the join order binary tree, the plan is marked as hinted. Note that it needs to perfectly match, just having the right tables in the right and left sub-tree is not enough, the plan must have the exact same structure/order as a sub-tree in the join order binary tree. The hinted property then determines how it will fare in the "tournament" explained in section 2.4.6.

A hinted sub-plan will be marked as best in its skyline dimension if it is compared to a non-hinted sub-plan. If it meets another hinted sub-plan however the two will compete normally. The reasoning behind this is that if all hint-compliant plans are marked as best, then a great number of plans will be unbeatable in the tournament, most of which has no real value. For example, if the join order of a plan is hint compliant but the join algorithm is inexpedient for the query, the plan should be beaten by other hint-compliant plans that have a more appropriate join algorithm. This way the number of plans kept by the optimizer is minimized while still nudging the optimizer into being hint compliant by preferring hinted plans over non-hinted plans.

When the optimizer chooses the plan to send to execute it first chooses the best cost plan as backup, then it chooses the cheapest among the pinned plans if there is one. As long as there exists at least one pinned plan, a pinned plan will be chosen, if not the best cost plan is chosen, and a warning is issued (currently a print).

4.5.2 Non-strict vs Strict

The non-strict version will keep other good (according to the optimizer) plans in case it does not find a hint-compliant plan, e.g. the hints are wrong/ try to pin a plan that would not work. In the non-strict version, the optimizer keeps at least as many plans as it would without the hints, and the overhead would not be smaller however it maintains a safety net that ensures a backup plan unaffected by bad hints. This is the equivalent of adding a new dimension to the skyline. Note that any hinted plan will cause the optimizer to discard any "slightly better" plans it competes against even those that would normally be kept.

The strict one gives the hint-compliant plans absolute competitive strength, meaning that if a plan is hint compliant any non-pinned plan competing with it will be thrown out. That would result in keeping far fewer plans, as any non-pinned plans competing against a pinned plan would be discarded. However it would cause the system to toss potential good plans that one might want to keep in case the hints do not work, e.g. user hints a join order that cannot produce a working plan. If the good plans are gone by the time the hints stop working, the resulting plan is a partially hinted plan that could have very varying cost. If the hints are correct it would minimize the number of plans processed by the hypergraph optimizer significantly. This is equivalent to a

subsuming dimension in the skyline of the tournament.

4.5.3 Notable Strengths and Weaknesses of Implementation

The strengths of this implementation lie primarily in having join order hints in the hypergraph optimizer, and the option of pinning multiple different alternative join orders with the same hint. Meaning it can be tuned to work for multiple data distributions.

The loop over the optimization only works for prepared statements so far. This is because clearing out just enough state while keeping everything necessary is such a big task, with so much complex code around it that it was not plausible within the confines of this project. It is also worth noting that the hint is only functional during the first execute after prepare, so it is very impractical. The impracticality can be fixed with the help of a stored procedure that can both prepare and execute upon being called, though that is not likely to be very effective. It works as a prototype, but the loop over optimize will increase the implementation cost of a production-ready implementation significantly.

The strengths and weaknesses of this implementation are mostly the exact same as the implementation in the current optimizer, with the big difference being the implementation cost. An important weakness is that this plan pinning will increase the time the CPU load is increased up to sixfold, and the same with high memory usage.

It is worth noting that there is no internal appending of hints and so the ones writing the queries have to add the hints themselves.

Possible Implementations That Were Not Implemented

Another possible solution that was not been implemented in this project is to only allow the hint-compliant hyperedges when the hypergraph is generated. This would be more risky though as it would limit the system if the hints force a plan that cannot execute the query. One could also put some preference on the compliant edges, that method would be very similar to the current one but differs in that the hinted flag would be set before optimization. It would need checks to see if the rest of the chosen edges (joins) were also compliant, so it has overhead too. That solution was not explored within this project.

The most interesting ideas work on developing the current solution further. One of the most interesting ideas is to implement an approach that allows the optimizer to make all the hinted plans within one optimize cycle, and choose the best one among them. This is very much plausible with the implemented strategy but has a risk of having very high memory usage due to many plans maintained simultaneously. Unfortunately, there was not enough time to implement this within the bounds of this thesis.

4.6 Hash-pinning

The idea behind hash pinning is to identify specific predefined sub-plans for a query and make them seem like particularly good plans to the optimizer, by doing this during the building of the plan one can manipulate the optimizer into making and choosing the desired plan as long as it is a valid plan for the query, and the hashes help sufficiently to build the desired plan. As the hypergraph the implementation is in is a bottom-up optimizer the pinning can fail if the pinning does not ensure the pinning of the building blocks of the desired plan.

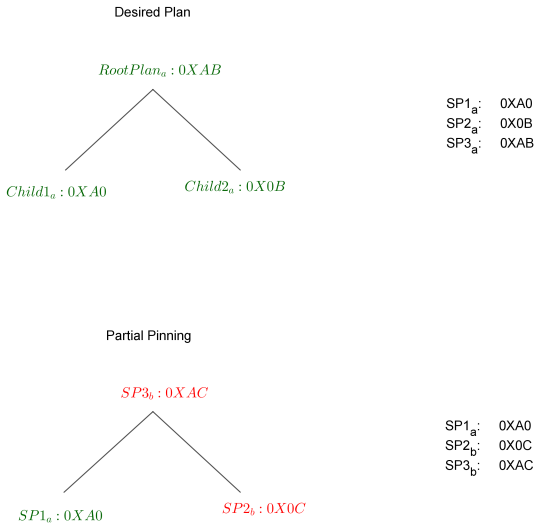


Figure 4.2: demonstration of only pinning parts of the plan

As an example, if one only pins the upper parts of the plan, like in 4.2. It can fail because the optimizer can discard the necessary bottom parts as the bottom parts are not pinned. If a necessary part of the plan is discarded the optimizer will be unable to produce the sub-plans over it in the tree, as the plans (and hashes) are produced using both the children sub-plans of the sub-plan and the sub-plan itself demonstrated in 4.2. Meaning that if one necessary child sub-plan is missing ($SP2_a$), it is impossible to produce the correct parent sub-plan ($SP3_a$). And so it is essential to get all the hashes of the plan one wishes to pin to have any real guarantee that the plan will be pinned. It is however possible to pin a lower part of the plan safely, without pinning the rest. Though that is not studied in this thesis.

4.6.1 Strengths and Weaknesses

The primary strength of the hash-pin solution is that it is perfect one-to-one pinning while still being parameter independent. This means that nothing can change the desired plan in any way(as long as the hashes do not change, which they should not!). If plan A is the only pinned plan for a query, any plan B that is marked as pinned by the optimizer is a perfect one-to-one match for plan A.

Another elegant advantage of this strategy is that it exploits the hypergraph optimizers traits, allowing it to pin multiple plans at once in a single optimize cycle. This is done by pinning multiple plans for the same query by storing all the hashes of all the pinned plans and then doing the same as for a single plan, only this will result in as many pinned plans as pinned. This is because unlike with the multi-hints, a hash-pinned plan is never allowed to lose the tournament even when its competitor is also pinned, and so all of the hinted plans "survive". This is because in the case of hash pinning it does not add any more plans than the ones pinned, and so the cost of maintaining all pinned plans is limited to the number of pinned plans.

Of course, because the solution exploits traits of the hypergraph optimizer Other optimizer types would not work well. This is because to pin multiple plans it is essential that it maintains multiple alternative plans while optimizing. An optimizer that does not do that will not be able to implement this effectively. And so this solution has usage only in optimizers similar to the hypergraph. And as such it has less usage for the common optimizer types.

Another weakness is calculating the hashes. The hashes of sub-plans are calculated using the sub-plan itself and all its children's hashes. This means that there will be a tree traversal for every time one needs to calculate a hash. This is not particularly great for speed. Additionally, the current hash function is ineffective itself as well.

There is also the issue of acquiring the hashes used to pin. The calculation of hashes has other small issues in that if the hash algorithm or the data used to calculate the hashes from sub-plans change, any hashes created before that will fail to match current hashes. So reliance on stable hashes is also a notable weakness.

Another important weakness is that when one pins the entirety of an execution plan like here, the pinning is very sensitive to changes in the database. New columns, changed names of columns or tables, and removal of indexes will easily break the pinned plan, resulting in no pinning at all, or partial pinning.

Like any pinning strategy, it is sensitive to data skew, especially when only pinning one plan.

4.6.2 Implementation

The implementation fetches the sub-plan hashes with its corresponding query digest (query id created from the query text) upon turning on the hypergraph optimizer(twice in a row), though many other options are possible. Then it reads them into a two-dimensional hash map (unordered_map typical lookup complexity of O(1) [57])and stores them in the thread handler. The reason for the two-dimensional hash map is that it allows one to first look upon the query digest (query id). When the query digest is found in the hash-map during optimize the hash-map for that query/digest is kept separately in the thread handler for easy access, and the rest of the optimize cycle knows itself to be pinned by hashes. The advantage here is that non-pinned plans will not have to do any of the things that are done to hash-pin like generate hashes for all its plans.

When the optimizer has made its choice of good plans, it chooses the best cost one among its plans. Then it tries to remove all non-pinned plans among the candidates and finds the cheapest one among the pinned ones if one or more pinned plans exist. If it does not find a pinned plan, it goes with the best cost plan found earlier and gives a warning (currently a print).

Pinning multiple plans is trivial with this method as it is simply to add the hashes for more plans for the same query, and it will by nature pin all of the plans whose hashes are stored for pinning. Making a multi-hint has a very low implementation cost when it is already implemented for a single hash-pinned plan. Which is very easy too.

Identifying pinned plans on query digests/query ids is very common in plan pinning where the DBMS pins without changes to the query text. Aka. hint-based plan pinning and SQL plan management. And it pins multiple exact plans like SQL plan management. PolarDB has implemented hint-based plan pinning practically in MySQL. But despite being implemented in the same system they share few similarities. Both identify the pinned queries on the query digest, but PolarDB adds hints and pins only one plan. Whereas hash-pinning does not add anything to the query, pins a full query execution plan, and pins multiple plans if requested. In that sense, it is more like SPM that pins multiple plans.

Non-strict vs. Strict

The non-strict method retains at least as many plans as an optimize cycle that does not have pinning. And as such it does not remove any overhead. It does however provide a means to inform the users about potentially good plans in the cases where the best cost plan is better than the pinned plans. Opening for adding the plan in case it actually is a better plan. This is especially useful when using plan pinning to prevent plan regression from updates to the optimizer, where one would want to know about any new plans with better performance. This is the equivalent of adding as many dimensions to the skyline as there are pinned plans. Note that any pinned plan will cause the optimizer to discard any "slightly better" plans it competes against even those that

would normally be kept.

The strict version will retain only as many sub-plans as pinned sub-plans. And so minimizes the overhead of both maintaining plans and generating hashes for proposed plans. Despite all table accesses being proposed, the following plans will only be proposed based on the plans that are kept from the previous tournament. Thus this becomes far more efficient. The biggest savings comes from the fact that only pinned plans will be retained, and processed minimizing the cost of optimization. At least on paper. This is the equivalent of adding a subsuming dimension to the skyline for every pinned query plan.

Light Hash Function

The hash function that was implemented before this project and used in these experiments is very inefficient. So when the optimize times for hash-pinning showed very bad performance. Measures where taken to find out why. The only part that would have higher overhead than join order hints when hash-pinned naturally bans more plans and the logic is as fast or faster than join order hint's, and without multiple optimize cycles. The only logical cause was the hash function. Therefore a more efficient hash function was made referred to as the light hash function to see if it would improve the situation.

The original hash function traverses through the sub-plan it creates the hash for twice. first, it generates a JSON object representing the entire sub-plan and its children. Then this JSON object is recursively traversed to generate the explain output with hashes for its children and then uses those hashes to generate the requested hash. The most inefficient part of this is that the function does a lot of things that are not necessary to generate the hashes. Only a tiny part of the JSON object is needed to make the actual hash. And the added overhead of first traversing to generate the JSON object and then traversing it right after is a waste, as it should be possible to do it within one traversing.

It would be too time-consuming to make a new hash function from scratch though that would likely be far more effective. And so the light hash function was made by taking the original hash function and removing unnecessary things. Specifically, minimizing unnecessary data in the JSON object, and making very little of the explain output. These measures also allowed some function parameters to be removed. The light hash function does nothing about the structure of the hash function, however, so there are still two traverses. Meaning that the hash function is still inefficient, but more lightweight than the original.

4.6.3 Strengths and Weaknesses

An important strength is the fact that multiple pinnings easily can be done within a single optimize cycle. This means that the implementation can be isolated to the hypergraph optimizer itself and does not require big invasive changes. It is not invasive in the optimizer either, using properties that already existed, and just adding some properties on matches in addition to adding another skyline dimension. In the case of the strict version, the pin dimension subsumes the other dimensions completely.

It does not demand any changes to the query text, and as such is more practical to use than hints.

A notable weakness is that when multiple plans are pinned, it is still the cost model choosing between them. So the pinned plans have to be chosen in a way that prevents the cost model from choosing a bad plan. As the cost model is typically the problem to begin with that can make it tricky.

Another weakness is that calculating the hashes for the sub-plans as they are proposed is not a particularly efficient strategy, As the calculation by nature includes a tree traversal with complexity $O(n)$. This can accumulate a lot of overhead if one uses a solution that keeps many plans while pinning in this way. A possible solution to that problem is to let the sub-plans themselves keep their own hash, reducing the traversal to at max a left and right child. This however increases

the size of the plans significantly, creating higher memory usage. In MySQL, the hashes would be stored as 18 chars, aka. 18 bytes, meaning a steep 12,5% increase from the original 144.

The complexity itself is not the only issue with the hash function as it also does a lot of extra work. The function's main function is to get hashes for debugging, so it is not implemented to be fast. It actually generates the entire explain output for the sub-plan for each run, Most of that explain output is not necessary to generate the hashes, and so it is a lot of extra unnecessary work. The hash function is made literal independent (parameters that are not ?) at the cost of allowing numbers in index names, so indexes used with has-pinning cannot contain numbers and remain effective.

Chapter 5

Experiment Setup

This chapter will detail how the experiments were conducted and some of the choices made in the process. The experiments primarily look at the optimizer performance when plan pinning is introduced. This is important to establish whether the pinning strategies are viable for production. If the performance of the optimization is so bad that it would outweigh the gain of a better plan in most cases it has little value. Therefore the experiments will study optimize time.

This is done with a timestamp before and after optimize in the code(, and one around execute) using the standard library's steady_clock[58].

5.1 Join Order Benchmark JOB

JOB(Join order benchmark) is a benchmark introduced to test the performance of database management systems on realistic data and realistic queries. It is also a benchmark with many joins. The dataset is a snapshot from the International Movie Database(IMDB) and contains 21 tables distributed on 3,6GB. The benchmark has 113 queries which are variants of 33 query structures[30].

This benchmark is chosen for its emphasis on join orders, since join order hints are the only hints made in this project. It also adds more data skew, which will demonstrate (though not visibly) the solution's ability to force plans despite the optimizer finding them to be bad.

For this thesis, only 10 of the queries are used. This is because there is considerable work to find valid join orders and to prepare the different experiments. The ten queries are chosen randomly and only one query of any structure. The queries used are:

1a, 2a, 8c, 13b, 14b, 16a, 18c, 21c, 27b , 31a

5.2 Experiment Setup

This section will detail the setup of the experiments and ...

5.2.1 Machine specs

The experiments were run on a Dell Inc. Latitude 7420 with an 11th Gen Intel®Core™i7-1185G7 @ 3.00GHz x 8 running Ubuntu 22.04.04 LTS (64bit). The machine has 32GiB RAM and 512,1GB disc space.

5.2.2 MySQL Configurations

The configurations that differ from the default are hypergraph for release builds, global_infile set to true. The hypergraph is currently not available in release builds, so this has been modified for the sake of these experiments. Global infile has been set to true, to allow the .sql files used in the experiments to run. The InnoDB Buffer is set to 20GB and the sort-buffer is set to 4GB.

5.2.3 Common Setup

All experiments are done by running every query a total of 102 times, measuring optimize time and execution time. The first two runs are removed as they are there to get all the data into memory and cache. This is important as the first run will be very different as it needs to get a lot of its data from disk, something consecutive runs do not. And so to get a meaningful average the first run has to be removed, to be absolutely sure two warm-up cycles are used here. The remaining 100 are used to calculate an average. It is these averages that will be studied as results.

Note that one experiment Includes 102 consecutive runs of query 1a, then 102 consecutive runs of 2a, and so on. Each query was duplicated 102 times in a .sql document. So each query would have its own .sql document, and these documents would be run in succession. They also switched the optimizer to hypergraph at the beginning of each file. (Twice actually, this has to do with the hash-pinning that only reads files on the second time.)

5.2.4 Setup for Baseline

No pinning or anything else is involved, but all queries are run in the hypergraph optimizer.

5.2.5 Setup for Join Order Hint Experiments

For the join order hints the setup was done by first finding the join order of the optimizer's chosen plan, and hinting that order with a join order hint, this effectively pins the plan the optimizer would choose without pinning. This is the setup for a single join order hint experiment.

The setup for multiple hints included trial and error on potential join orders. Join orders that were allowed (successfully pinned a join order) and had an acceptable execution time (not more than 5 minutes) were used. And so experiments for two, three, four, and five simultaneous hints were made for each query. Note that the plans pinned in five include all plans pinned in four and so on, and all include hints for the best-cost plan.

The files for multiples put the query in a prepared statement, as these allow for reruns of the optimizer, and therefore can handle multiple hints. Then the prepared statement is executed. Then the prepared statement is prepared again, executed, and so on 102 times. For the experiments with a single hint, the prepared statement tactic is unnecessary, and so the queries are not in prepared statements.

An important thing to note here is that the best-cost plan (the plan the optimizer normally chooses) is hinted in the first iteration, meaning that for most cases the plan found first will be chosen, which means there will be an extra optimize for the experiments that pin multiple plans with join order hints. Resulting in an expected total optimize time between 3-6 times the initial optimize time.

5.2.6 hash-pinning Setup

The hash-pinning experiments run the same .sql files as the baseline experiments, but the DBMS reads in hashes for the query beforehand. For the single experiment each query has only one hash,

for two there are two hashes for each query, and so on until each query has five hashes for the five-plan experiment.

The hashes are acquired by running the plan one wishes to pin and retrieving the hashes from the run. The first hash set for each query is the hash set for the best-cost plan for the query (just like for joh). And the second set and so forth are found by pinning the plans hinted by the join order hints. This means that the two experiments pin the exact same plans. The join order hint experiment with three plans pin the exact same three plans as the hash-pin experiment with three plans for example. Note that the hashes from explain are wrong for every sub-plan except access methods and the finished plan. The query digests are found by either printing it (similar to the hashes) when the query is run or by using .

Note that when the join order hints technically pin more than one plan, the hash-pin experiments pin only the plan that was ultimately chosen among the hinted plans. And so the hash-pin is a little more specific than the join order hint.

5.2.7 Non-strict vs. Strict

The distinction between the strict and non-strict is important here. All experiments except for the baseline are run in both the strict build (where pinned plans dominate) and the non-strict build (where pinned plans merely cannot be beaten). This is done using two different builds of MySQL, where the only difference is whether the pinning is strict or non-strict.

5.2.8 Normal vs. Light

When nothing else is specified the original hash function that existed in MySQL (though parameter independent) is used. When light is specified in the experiments it means that the hash-pinning utilized the more efficient light hash function. Light is run with both non-strict and strict.

5.2.9 List of Experiments

- baseline: no pinning
- non-strict join order hint 1,2,3,4, and 5
- strict join order hint 1,2,3,4, and 5
- non-strict hash-pinning 1,2,3,4, and 5
- non-strict hash-pinning light 1,2,3,4, and 5
- strict hash-pinning 1,2,3,4, and 5
- strict hash-pinning light 1,2,3,4, and 5

Chapter 6

Results and Discussion

This chapter will study the results from the experiments in reference. The first experiments presented are the join order hint experiments. This has two variants one non-strict and one strict. Afterward, the results of experiments done on hash-pinning are demonstrated. Also for hash-pinning, there is a non-strict and strict variant. In addition, there is an experiment that tests the same but with slightly more efficient hash generation called hash-pinning light.

The experiments are meant to show whether the plan pinning strategies proposed are viable ways to pin plans. An important factor in this is performance, as increased optimize time can reduce or remove the gain of a better pinned plan. And so the experiments show the optimize time of the different strategies compared to the baseline.

6.1 Non-strict Join Order Hints

This section will cover the results of the non-strict join order hint experiment. The join order hint experiments utilize the newly developed join order hint for the hypergraph to pin plans with a provided join order. In the non-strict variant, the plans that have a hint-compliant join order cannot lose the tournament unless it is beaten by another hint-compliant plan.

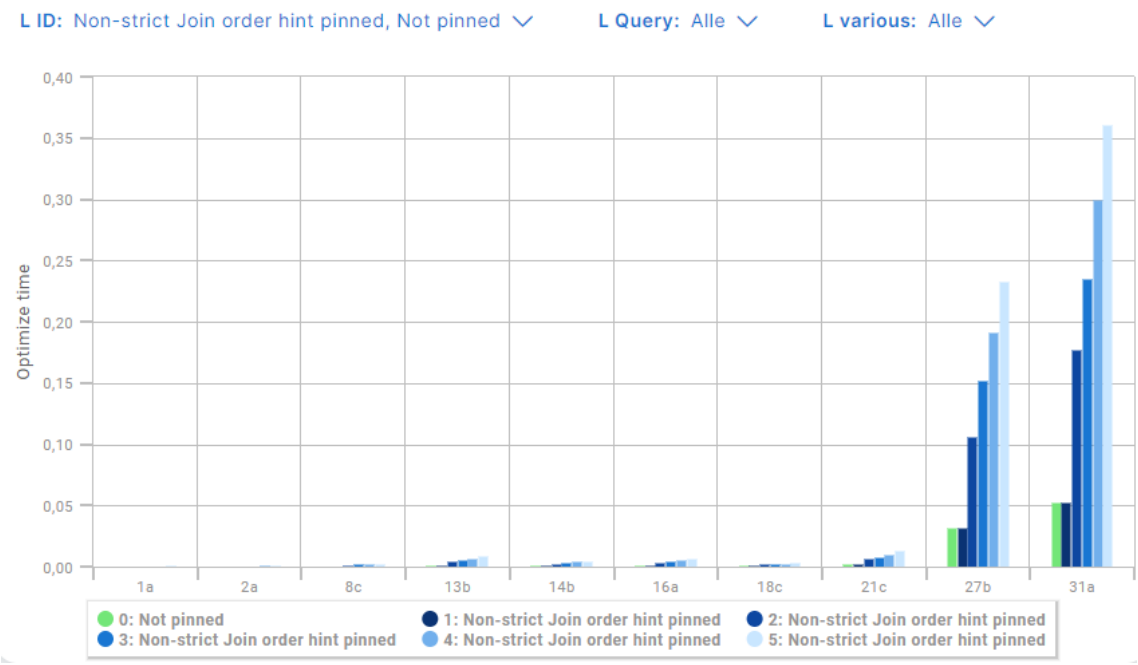


Figure 6.1: Results non-strict join order hint 1-5, optimize time in seconds pr. query

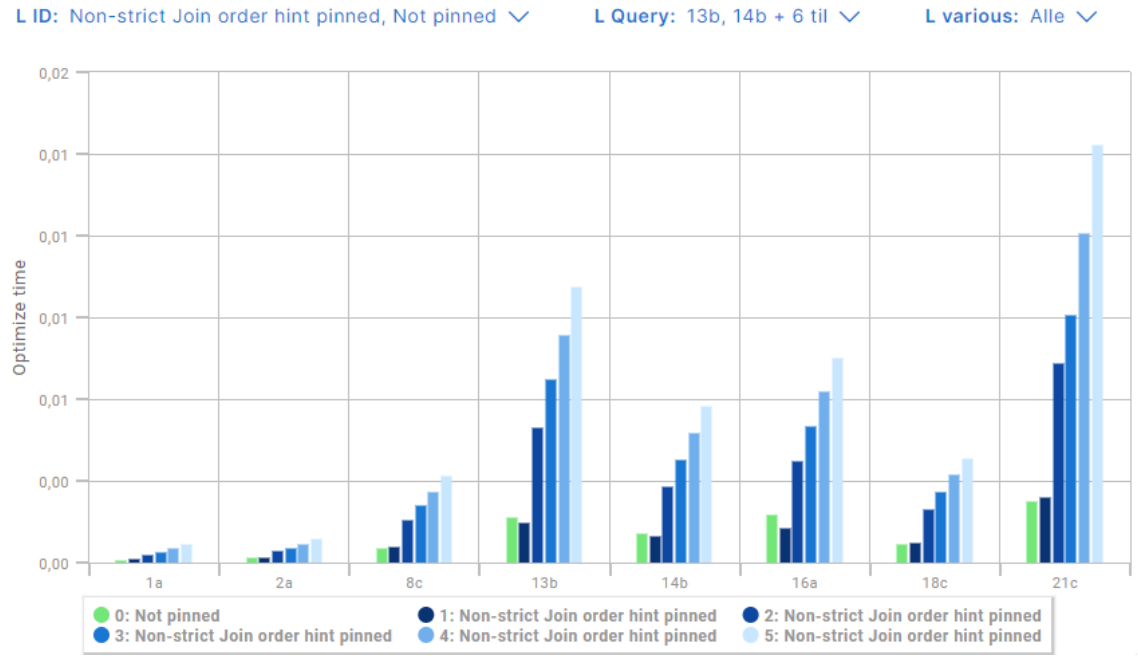


Figure 6.2: Results non-strict join order hint 1-5 with 27b and 31a removed, optimize time in seconds pr. query

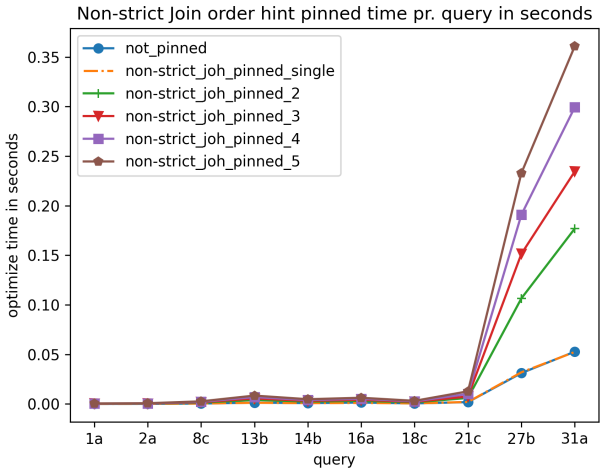


Figure 6.3: Results strict join order hint 1-5, optimize time in seconds pr. query

The results show that join order hints will slow down optimize for most cases, and this effect increases as more plans are pinned. The version that pins only one plan has an optimize time very close to the baseline, and even a little better for some queries. However, the optimize time can be far longer than the baseline for multiple plans. This is especially pronounced for queries 27b and 31a.

The distinct peaks at query 27b and query 31a also naturally correspond to higher optimize time on the baseline. This is because the optimizer is run as many times as the number of pinned plans (plus one for those that pin more than one plan) in the experiment. So a longer baseline optimize time would naturally mean that the cost of re-running the optimizer would be higher. This explains why the optimize time for two pinned plans is about three times the optimize time of the baseline and single one.

A notable trait is that when the optimize time is low, the accumulated optimize time is rather low, but when the optimize time is high, the re-optimization can be very time-consuming, possibly resulting in cases where the cost is too high to defend using it. Despite this, in this experiment no average total optimize time has exceeded 0.40 seconds, meaning that the time usage is not extreme. The optimize times are not disastrous and so this can be a viable pinning strategy.

6.2 Strict Join Order Hints

The strict join order hints experiment was run with join order hints like the one covered above but here the hint-compliant plans dominate any plan that is not hint-compliant. This will naturally reduce the plan space of the strict variant a bit more than the non-strict option. The results do however show that they perform very similarly.

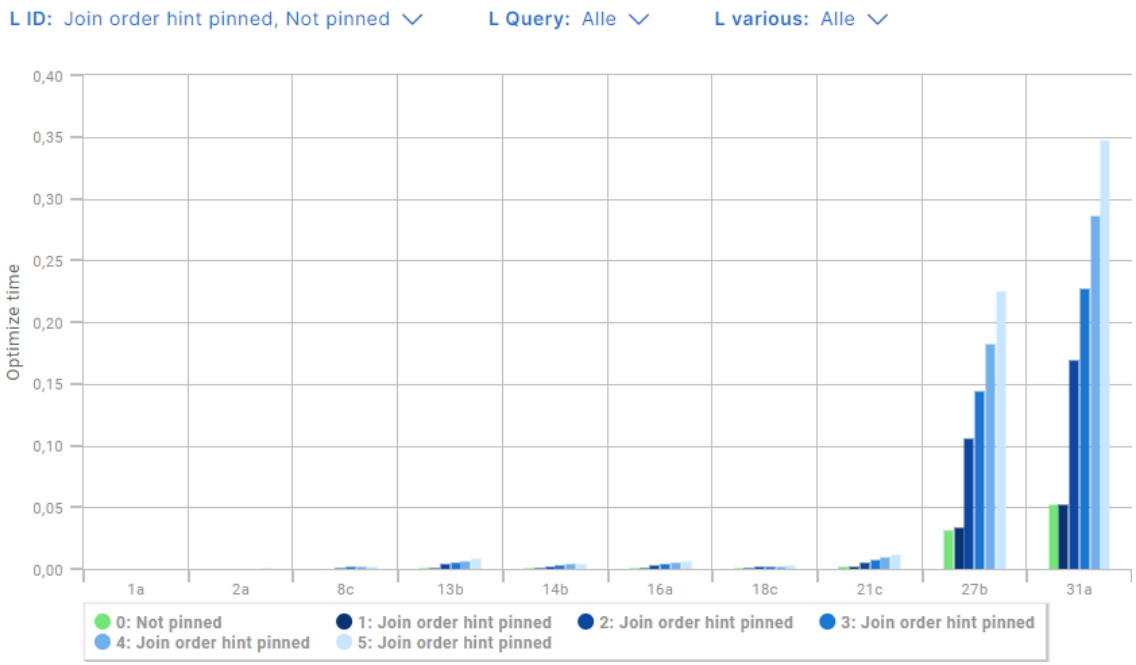


Figure 6.4: Results strict join order hint 1-5, optimize time in seconds pr. query

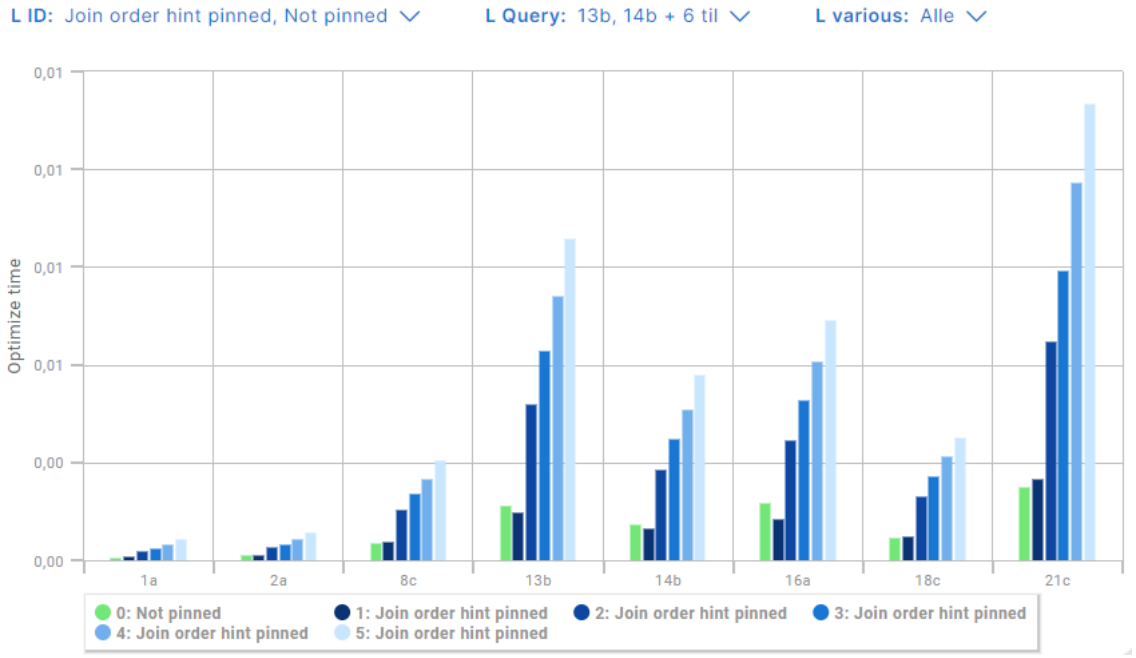


Figure 6.5: Results strict join order hint 1-5 with 27b and 31a removed, optimize time in seconds pr. query (y-axis 0,0025 increments on squares)

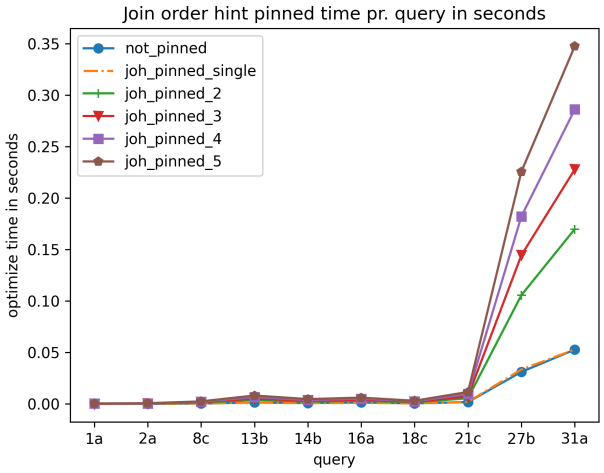


Figure 6.6: Results strict join order hint 1-5, optimize time in seconds pr. query

The results for strict in 6.4 look very similar to the non-strict ones, but do show marginally lower optimize times, for the cases where one hints more than one plan. This indicates that the difference between the two is marginal, but that the strict one saves some time over multiple iterations. This indicates that there is something to be gained from maintaining fewer plans also here.

The fewer plans compared to the non-strict join order hint implementation come from the fact that this version dominates any non-hinted plans. The reduced search space improves the optimize time somewhat but is still unable to compensate for the optimize loop or really speed up the optimize time much compared with the non-strict join order hints. This means that join order hints are a good option for exploiting the non-strict implementations' ability to find new good plans without significant added cost. The difference between the strict and non-strict versions are insignificant compared to the overhead of the re-optimizations.

6.3 Non-strict Hash-pinning

These are the results of the experiments run with the non-strict hash-pinning variant. This variant utilizes hashes to exactly pin plans, however, it does not have the pinned plans dominate other plans automatically. The results show very high optimize times for this particular strategy.

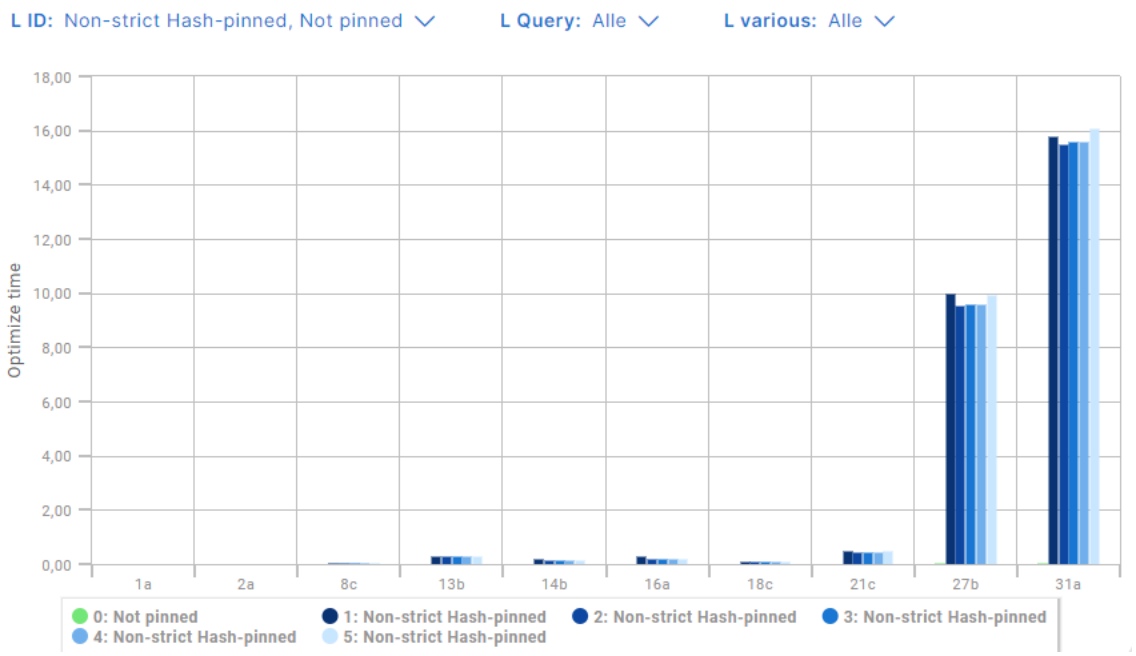


Figure 6.7: Results non-strict hash-pinning 1-5, optimize time in seconds pr. query

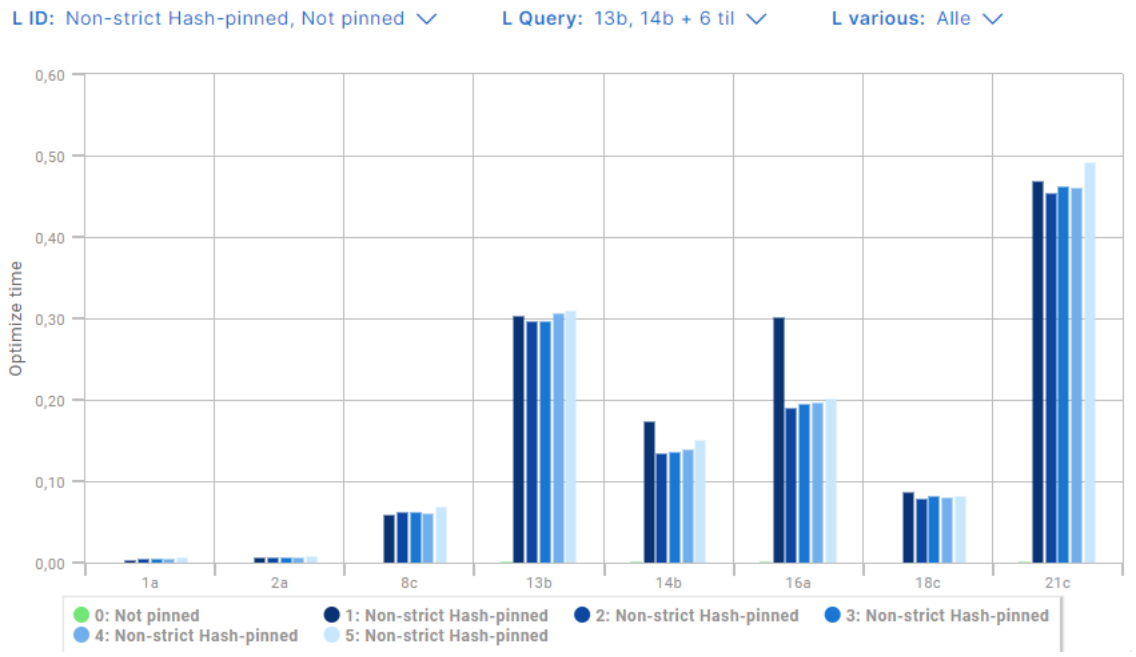


Figure 6.8: Results non-strict hash-pinned 1-5, with 27b and 31a removed, optimize time in seconds pr. query

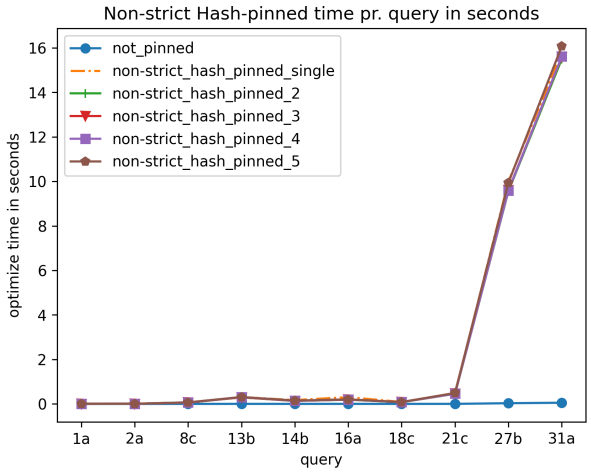


Figure 6.9: Results hash-pinned non-strict 1-5, optimize time in seconds pr. query

The results in 6.7 show that all the pinned experiments have far higher optimize times than the baseline including when only one plan is pinned. This implies that some part of the implementation is a significant bottleneck. The logic for the skyline/tournament is just as easy/complex as the join order hint implementation, even a little stricter than the join order hint non-strict version. which should cause fewer plans to be considered than for non-strict join hints. The most likely culprit is the current hash function, which is already stated to be ineffective.

Another surprising observation is that the single hash-pin is often slower than the versions that pin multiple plans. This can mean that the search space is not really minimized much. Considering that the plan pinned for single in these experiments is the best cost plan, it would never lose the tournament regardless. And so ensuring that this particular plan never loses does not really minimize the plan space by any other plans than those identical or "slightly better". The consequence of this is that all the plans in the not-pruned plan space must have their hashes generated. The best-cost plan is also typically a plan that would dominate a lot of other plans.

When a second plan is pinned, this one might pin a plan that would normally be discarded by the optimizer which means that it could make plans it otherwise would not, but it could also mean that this plan can begin to dominate plans it normally would not. Especially the plans that usually were kept because they were "slightly better" or identical, as they cannot beat a pinned plan unless they are pinned themselves.

Continuing this hypothesis the plan space might start to grow again as more plans are pinned or not. The graphs in 6.7 show that the optimize times typically start high for the single and then low for two, three, and four and get higher again for five, but there are significant variations in this pattern. This might indicate that the optimize time depends a lot on how the plan space is and how the pinned plans fit into it. And that multiple pinned plans dominate more of the plan space, but can also open more of the plan space when the number of pinned plans reaches a certain threshold or a very different plan is added, opening a new part of the plan space that was not explored before it was pinned.

When compared to join order hints these results are disastrous. optimize times above a second is bad already, but when one sees the low optimize times from both join order hint variants the comparison is not good for non-strict hash-pinning. And so far this is the only pinning strategy with prohibitive optimize times. A better hash function might correct this.

6.4 Non-strict Hash-pinning Light

This experiment aims to see if the non-strict hash-pinning above can be improved by a significant amount by making the hash function more efficient. The optimize time of the hash-pinning with a lighter hash function is presented in addition to a comparison of the optimize times of the original hash-pinning presented above and the light version.

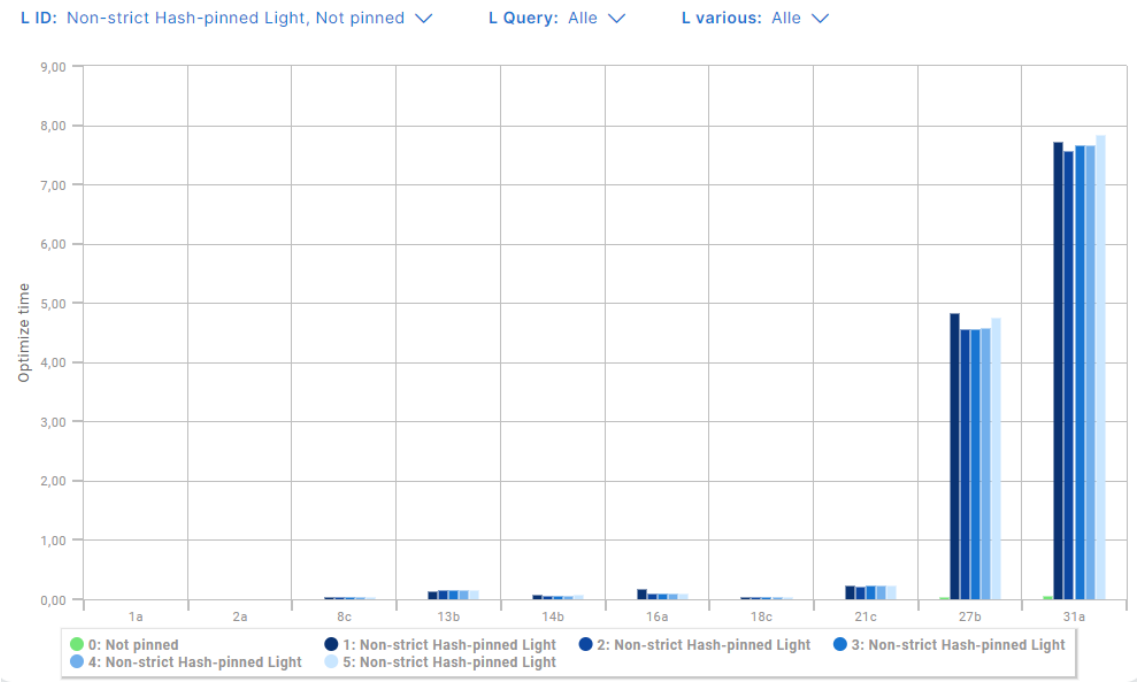


Figure 6.10: Results join order hint non-strict 1-5, optimize time in seconds pr. query

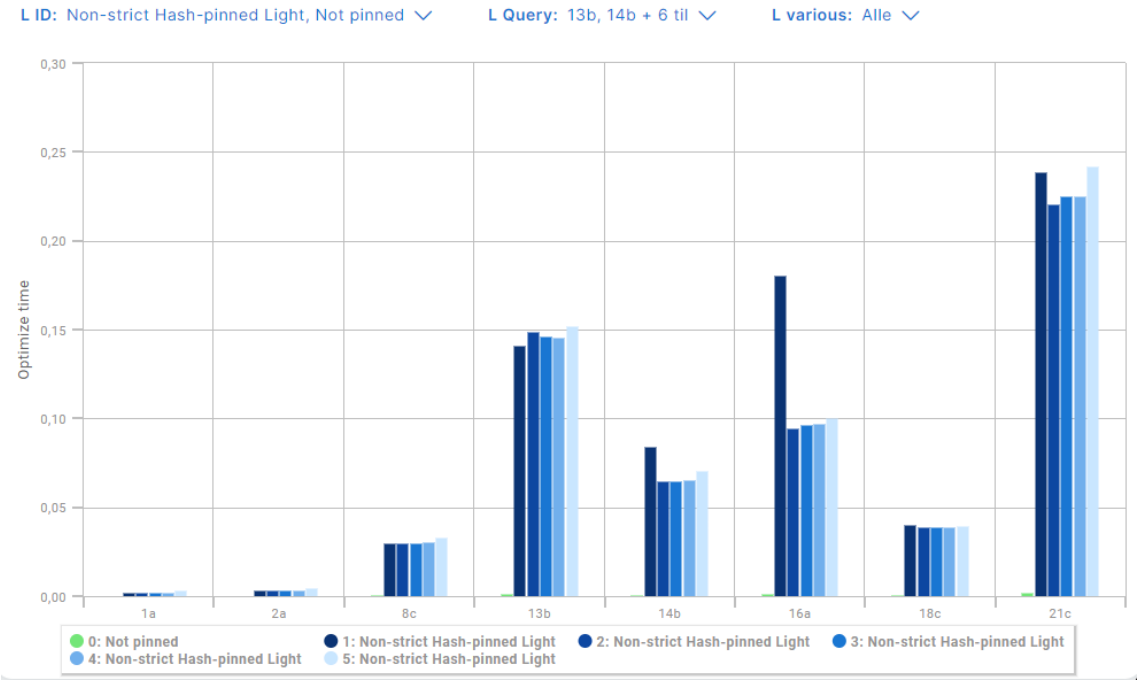


Figure 6.11: Results join order hint non-strict 1-5, with 27b and 31a removed, optimize time in seconds pr. query

Graph 6.10 and 6.11 show that the optimize time compared to the baseline is disastrous. And that 27b and 31a still have optimize times of multiple whole seconds, which is very bad as the execute times of most queries are not this high. However, the optimize times have become dramatically shorter with the light hash function compared to the original hash function.

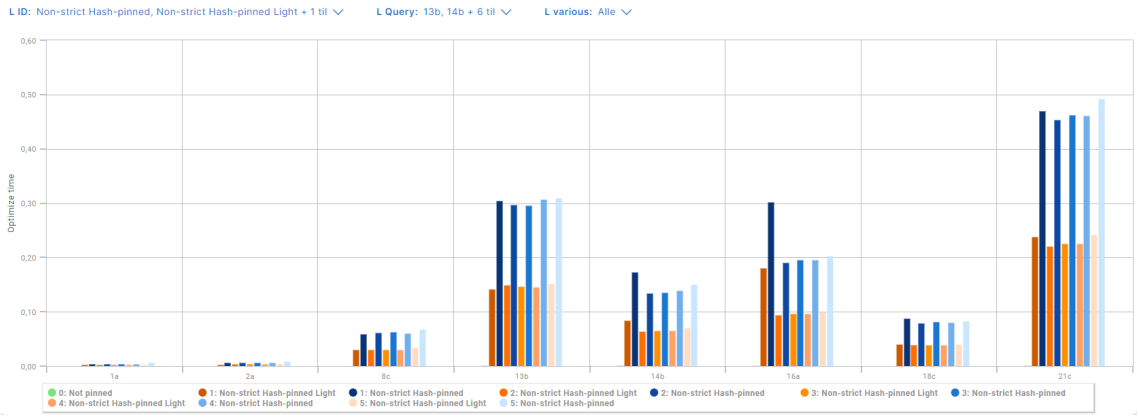


Figure 6.12: Results non-strict hash-pinning light compared against original, optimize time in seconds pr. query, 1a-21c

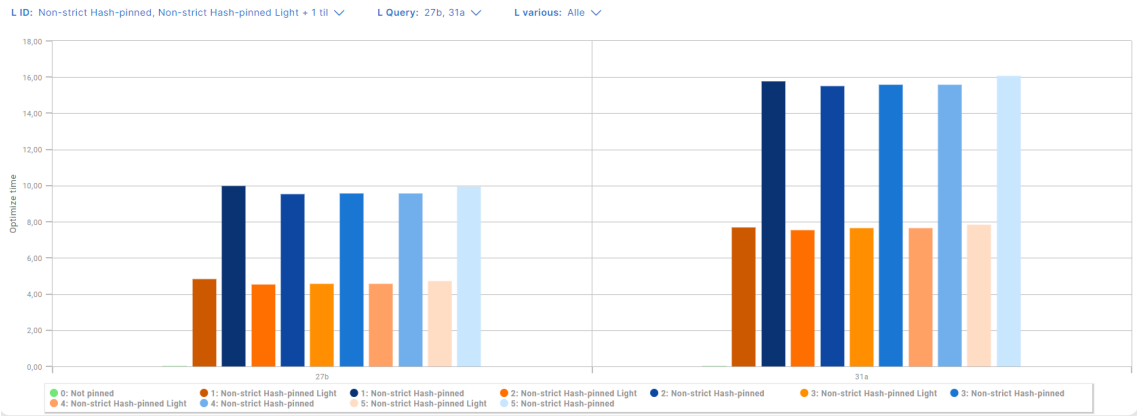


Figure 6.13: Results non-strict hash-pinning light compared against original, optimize time in seconds pr. query, 27b and 31a

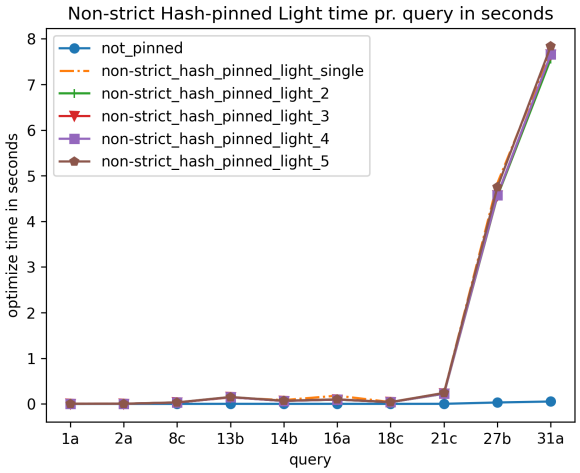


Figure 6.14: Results non-strict hash-pinned light 1-5, optimize time in seconds pr. query

The optimize time of the light non-strict hash-pinned strategy has far better optimize times than with the original. This is very apparent in figures 6.126.13 that demonstrate the difference in execution time between the experiments run with the original hash function compared to the new lighter one. The optimize times with the light hash function are actually around half the original optimize time.

This improvement proves that improving the hash function even a little improves the optimize times significantly, which indicates that the bottleneck is the hash function rather than other factors in the system. And also that the optimize times might be passable if an actually efficient hash function is introduced.

Despite having far better optimize times than with the original hash function this is still not a good option, optimize times of over five seconds are not acceptable. Especially when the join order hint implementation is subsecond for all queries despite the looping.

6.5 Strict Hash-pinning

This section will study the results of the experiments run with strict hash-pinning, meaning that pinned plans dominate any not pinned plans. The experiments try to find whether strict hash-pinning is a viable way to pin plans in the hypergraph optimizer by studying the optimize time when hashes are used to pin 1,2,3,4 or 5 plans. And the results show that the strict version is far faster than its non-strict counterpart.

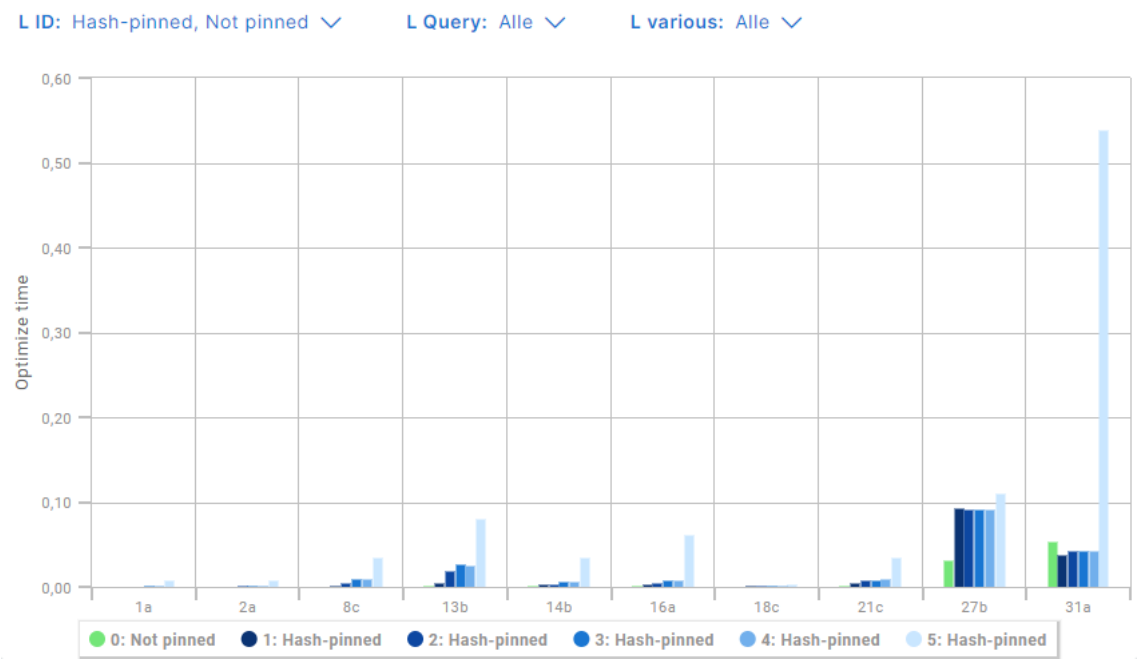


Figure 6.15: Results hash-pinning strict 1-5, optimize time in seconds pr. query

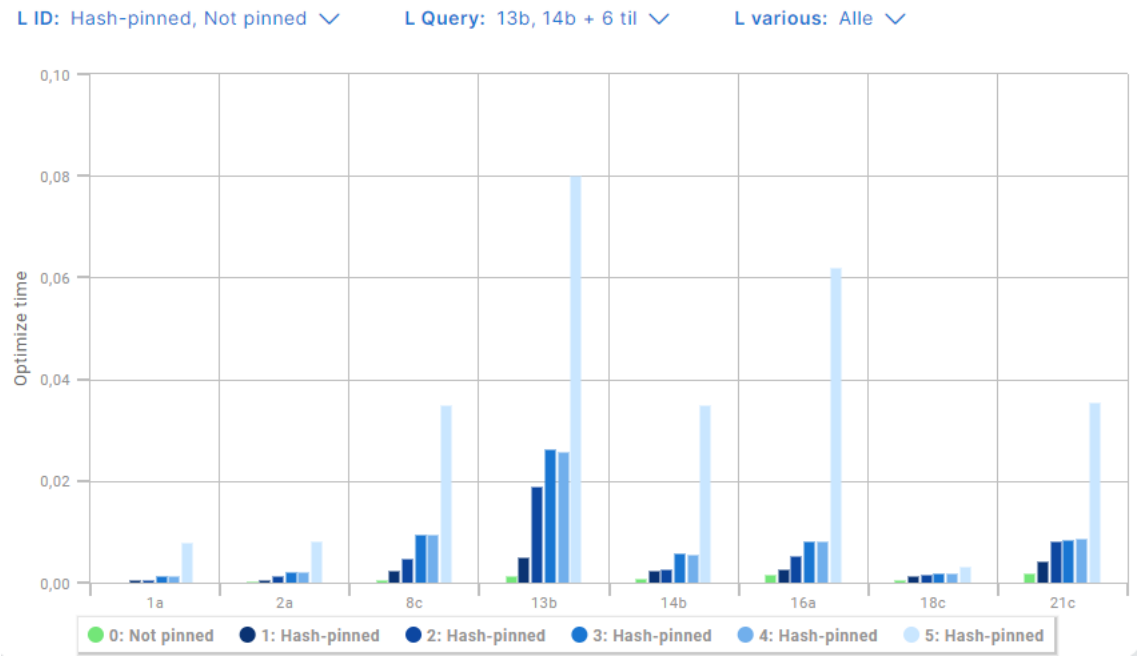


Figure 6.16: Results hash-pinning strict 1-5, with 27b and 31a removed

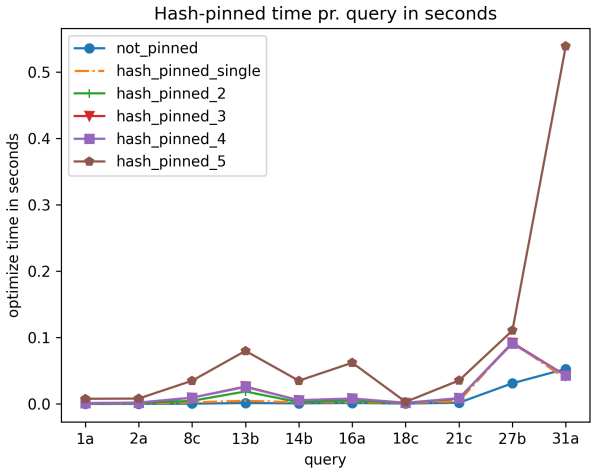


Figure 6.17: Results hash-pinned strict 1-5, optimize time in seconds pr. query

The results in figure 6.15 show that the optimize times do not get remotely as catastrophic as they do in the non-strict version. The times stay well below a second. They are however worse than for join order hints. There is one thing that it seems to do better, however. The peaks are not as pronounced for query 27b (where the optimize time is far lower for hash-strict) and only really appear when adding the fifth hash for 31a. Before adding the fifth 31a with strict hash-pinning is actually faster than the baseline. The spike when adding the fifth hash set implies that strict hash-pinning effectively keeps the number of plans down unless a pinned plan happens to expand the plan space drastically.

After a quick study of the hashes for 31a (see appendix), it was clear that the fifth plan for 31a pinned very different join orders than the other hashes, meaning that more join orders (built from the pinned join order) were made and evaluated when the fifth plan was pinned, causing the explosion. This shows that the plans one pin can affect the execution time drastically, and that similar plans pinned together will cause a far smaller plan space, making hash-pinning more affordable in those cases.

The optimize times are well below a second and for cases where pinning is necessary the expected gain should be far more than half a second, making this perfectly acceptable optimize times. It is still worse than join order hints for simpler optimizations but does actually show good performance for the cases where the plan space is normally high, despite this it is still somewhat unpredictable also here where 31a with five plans pinned has a higher optimize time than join order hints despite 1-4 beating the baseline.

Strict hash-pinning still suffers from bad hashes and thus faces similar performance issues despite its ability to reduce the plans significantly. In fact more than any other method. This is because for join order hints even the strict version only dominates those plans that do not have its join order. The optimizer still evaluates access methods and join algorithms independently on the hints, though some join algorithm plans will be avoided through pinning it is still no match for the strict hash-pinning that discards all plans that are not pinned explicitly pinned. This prunes access methods, join orders and join algorithms, and everything else. And so even with a single optimize cycle and a far smaller plan space than the other methods it still underperforms. This is the fault of the ineffective hash function. (the only other implementation detail that differs are hash-maps, that have lookup complexity at O(1))

6.6 Strict Hash-pinning Light

This experiment introduces the light hash function to the strict hash-pinning strategy. This section will study the optimize times for the light version and then compare them with the corresponding times for strict hash-pinning with the original hash function. The results show a large improvement when introducing the light hash function.

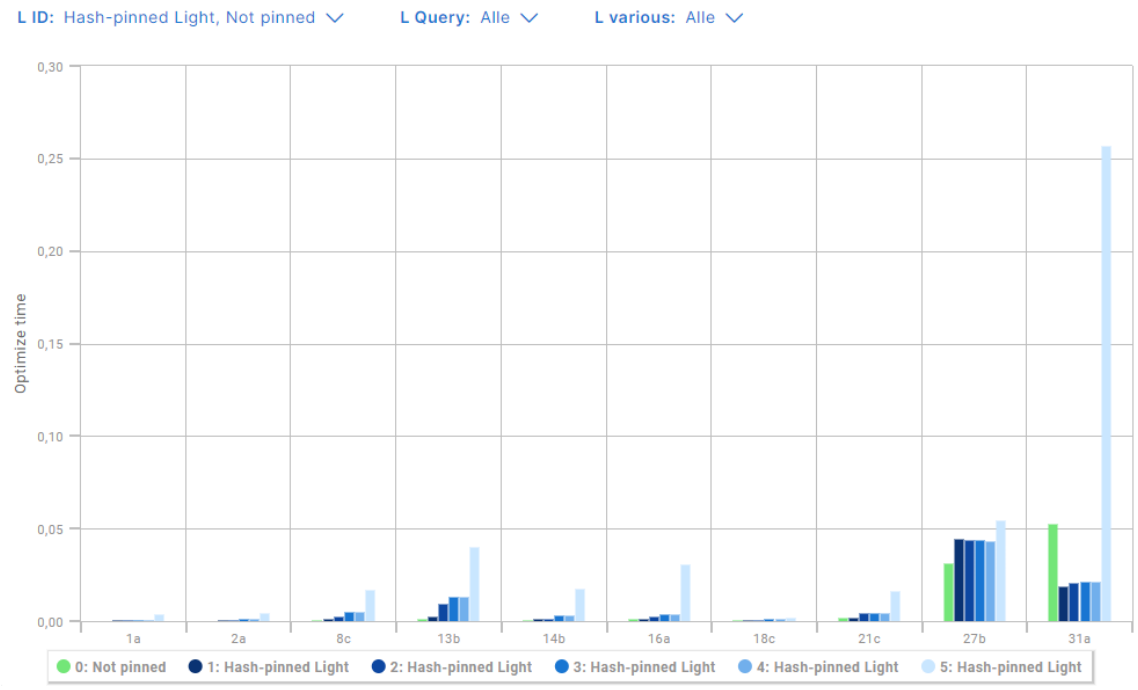


Figure 6.18: Results hash-pinning strict light 1-5, optimize time in seconds pr. query

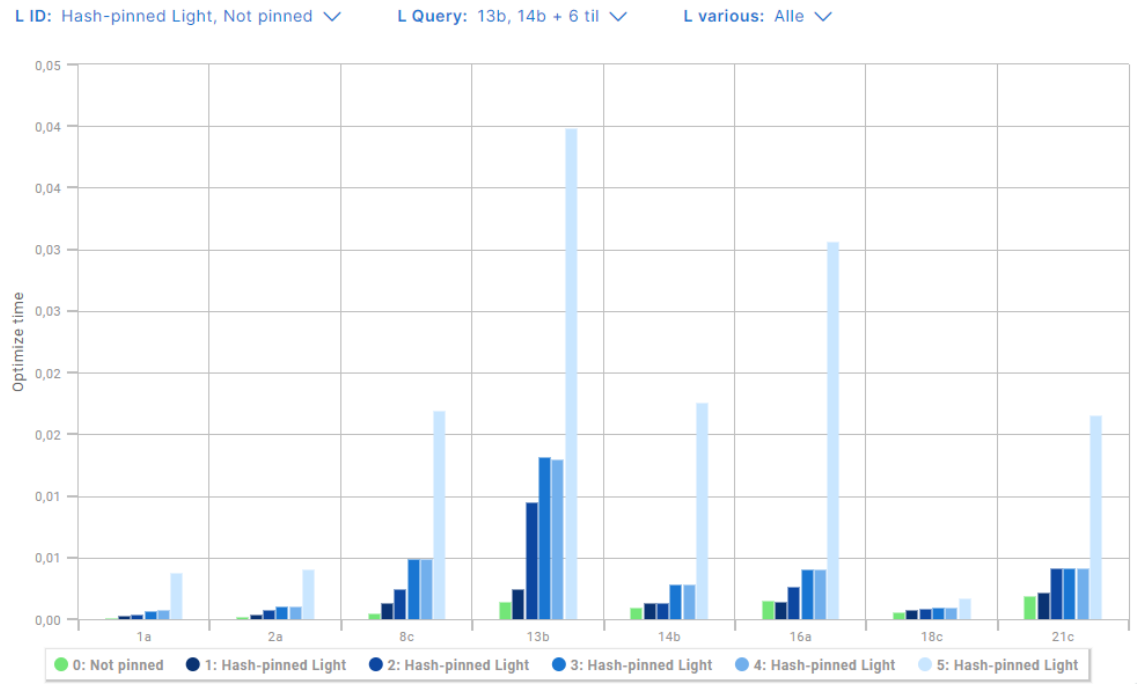


Figure 6.19: Results hash-pinned strict light 1-5, with 27b and 31a removed, optimize time in seconds pr. query

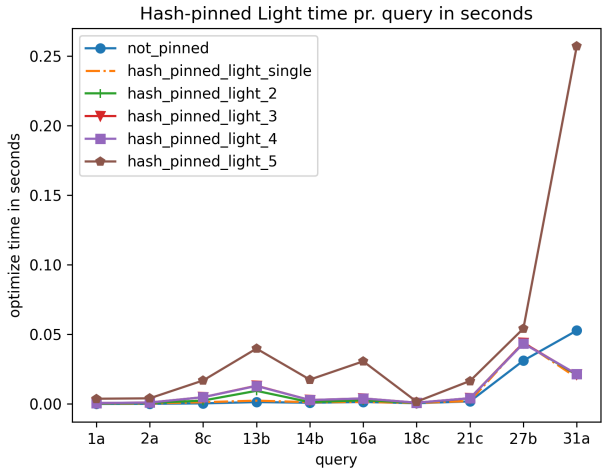


Figure 6.20: Results strict hash-pinned light 1-5, optimize time in seconds pr. query

The results with the strict hash-pinning with the light hash function in figures 6.186.19show rather great performance where even the worst case in 31a with five plans has an optimize below 0,3 seconds. which beats the optimize time for five pinned plans for 31a with strict join order hints. Optimize times well under 0,3 seconds are perfectly acceptable for most cases where the pinning is necessary.

There is also an interesting occurrence on 31a where the pinning of 1-4 plans actually decreases the optimize time significantly, indicating that for large plan spaces, hash-pinning can improve upon optimize times. In other words, this trait can be used to improve optimize times when the plan space becomes too large. Almost inducing a caching effect, where the plan is made faster than baseline because the plan space is tiny, which again is because the plan in some ways is premade by being pinned. They have to be optimized, but the optimizer does minimal work, as the pinning has alrady made them. Meaning that hash-pinning can be used as a light hash i the hash function becomes fast enough.

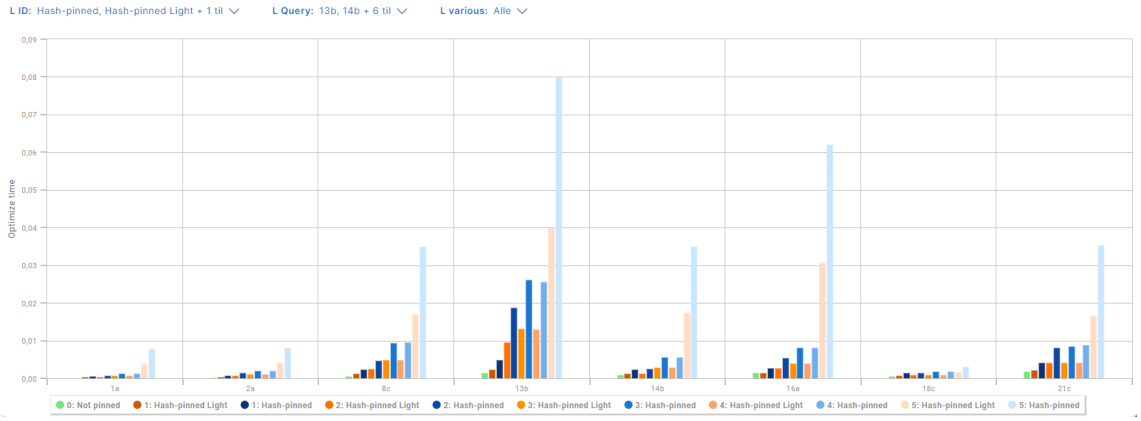


Figure 6.21: Results strict hash-pinned light 1-5 1a-21c

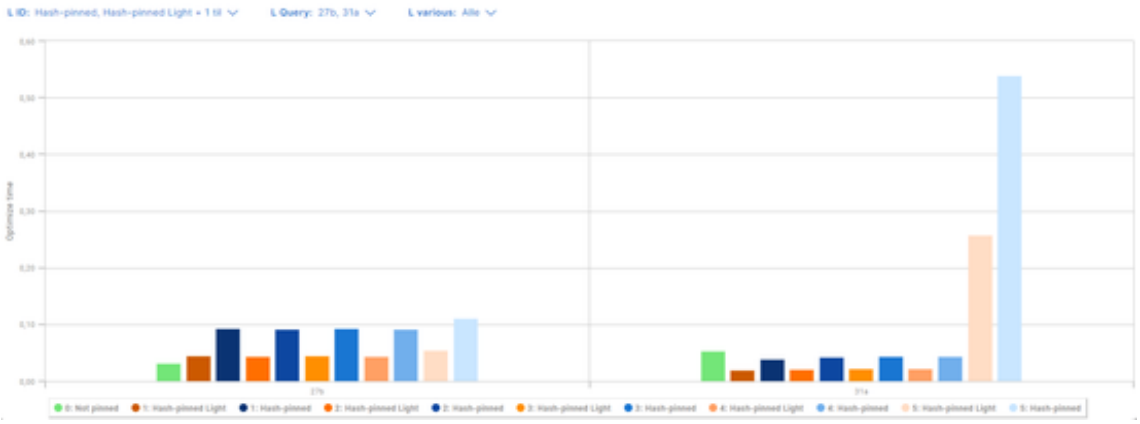


Figure 6.22: Results strict hash-pinned light 1-5 27b and 31a

The comparison in figures 6.216.22 shows that also for strict the faster hash function improves the optimize times significantly, almost cutting it in half on average. This shows how much the hash function has improved the optimize time. And also indicates that the bottleneck is indeed in the hash function. Seeing as the hash function is still ineffective, an actually efficient hash algorithm could make the hash-pinning method perform far better possibly also allowing the non-strict hash-pinning to be a viable plan pinning strategy.

The join order hint implementations have both shown that they scale predictably as the number of pinned plans increases. They also display acceptable optimize times for both the strict and non-strict versions. Comparatively, the hash-pinning is falling short, with one notable exception. When the baseline optimize times become high the strict hash-pinning actually performs very well. This implies that join order hints perform badly when the initial optimize times are high as it does optimize multiple times. The hash-pinning on the other hand will seem far better, as it makes many of the time-consuming decisions fast through hashes. This means that for now, they have different strengths. If an efficient hash function is introduced there is a good chance that hash-pinning will outperform join order hints for more cases. And possibly even the baseline, inducing a plan pinning that has a plan cache-like trait in that a plan is cached/pinned and quickly remade by the optimizer.

6.7 Comparison

The times for the single join order hint are very close to the baseline. The multiples for join order hints do however suffer from the implementation in that it cannot ever be faster than ”n” times its hinted optimize time, where n is the number of plans pinned with join order hints. The hash-pinning however suffers from far worse performance. For non-strict pinning the optimize times can be several seconds for the worst cases using the current hash function. The strict version is much faster as it has to make far fewer hashes but is still far worse than the baseline on average and performs far worse than join order hints overall.

Despite the ineffective hash function, the times for 27b (2-5) and 31a (1-4) on hash-pinned strict were actually lower than join order hints. And for 31a (1-4) strict hash-pin actually outperformed the baseline. This shows that strict hash-pinning has potential if one can find a more efficient hash function.

The hash-pinning with the light hash function is significantly faster than the versions using the original hash function. Hash-pinning is still not better than join order hints in optimize time with the exception of queries 27b and 31a where strict hash-pinning beats both non-strict and strict join order hints.

The light hash function reduces the optimize by about half when compared to its counterparts that use the original hash function.

Chapter 7

Conclusion

This thesis has introduced several key concepts to understand plan pinning and optimizers. Then the implementation of join order hints for both the current optimizer and the new hypergraph optimizer both can hint multiple join orders. Secondly hash-pinning was introduced, which is also capable of pinning multiple plans and is implemented for the hypergraph optimizer. The implementations in the hypergraph optimizer were then tested to see how they affected performance when pinning 1-5 plans. This final chapter will draw conclusions by answering the research questions and listing some possibilities for future work based on the work done in this thesis.

Research question 1: How can one implement plan pinning in an existing DBMS like MySQL?

Pinning has been implemented in multiple systems, and with multiple different techniques. The established techniques mentioned in 3 related work are optimizer hints, hint-based pinning that use the optimizer hints, and SQL plan management that adapts to changes. In 4 implementation, there are two additional methods introduced. These are multi-join order hints for the current and hypergraph optimizer, and hash-pinning for the hypergraph optimizer which also allow the hinting of multiple plans.

Research question 2: Should pinning pin parts of an execution plan or the full execution plan?

As stated in chapter 3 pinning the entire plan means more control over what plan one gets, similar to what SPM aims for. While pinning partial plans allows the optimizer to optimize the non-pinned parts of the plan which is more adaptive to change. This project implemented both (see chapter 4) and has established that to minimize the plan space as much as possible pinning the entire plan is better, however, it is not particularly fast as of now because the inefficient hash function makes the optimize time explode, especially for non-strict hash-pinning. Join order hints show far better optimize times overall, and with the current hash function options, using hints and pin partial plans is the fastest option.

Research Question 3: Is it permissible to use less domineering pinning in order to discover potentially better plans?

For the major implementations introduced in chapter 4 of this thesis, there has been made a strict and a non-strict version. The non-strict version allows the optimizer to mostly plan as if no pinning was present, but never discarding pinned plans. This has the benefit of allowing the system to discover better plans and notify the users. The join order hint implementation incurs little extra overhead compared to its strict version. But for the current implementation for hash-pinning the cost of calculating hashes (both original and light) has made a non-strict solution prohibitively expensive in terms of optimize time, especially when the plan space is big.

Research question 4: Can pinning multiple plans for one query have benefits, and how costly would it be?

Chapters 3 4 establishes that the benefit of being able to pin multiple plans for one query is that it becomes possible to pin plans that are good for different cases, like for different data distributions. The results in chapter 6 show that pinning multiple plans scale differently for different approaches. The join order hints naturally scale badly because of the multiple re-optimizations, but the cost is predictable and it is not particularly high as long as the ordinary optimize is low. For hash-pinning the scaling when going from single to multiple plans depend heavily on whether it is a strict or non-strict version. The non-strict version is slow but scales very well. The optimize times for multiples are about the same as or better than when pinning only one plan. With a better hash function, non-strict hash-pinning could perform very well in cases where one wants to pin multiple plans. The strict hash-pinning behaves more randomly in when comparing multiples to single plan pinning, but often badly for the fifth plan, and okay for 1-4.

7.1 Future Work

This thesis showed that the somewhat faster hash function improved the optimize time dramatically. This implies that large improvements can come from making the hash function even more efficient than the light one presented here. If one makes such a hash function the hash-pinning can become far faster. This is very much possible considering how inefficient both the original and the light one is.

The current implementation cannot pin partial orders, this should be possible to do with little trouble. It simply means that a sub-plan with exactly the hinted partial order has to propagate its hinted flag to its parents.

Both join order hints and hash-pinning are currently not in a state where they are practical to use. For multiple join order hints in the hypergraph one has to remake a prepared statement for every execution, which is neither practical nor efficient. For hash-pinning one would have to find practical ways to acquire hashes and to add syntax that allows users to use the hash-pinning effectively.

Plan Banning:

Hash pinning has a lot of potentials that was not studied within this project. One of the most interesting ones is the possibility to ban specific plans. By rigging the tournament in the other direction it is possible to make any plan that matches a hash to lose the tournament. By doing this one can remove the problematic part of a query with very high precision while still allowing the optimizer to have every other option. One can also remove any sub-plan that is banned before the tournament, this is probably faster.

One efficient example is to ban a finished plan. With this method, all plans will be made as usual and then hashes will be generated for the finished options, and then any banned plans are removed from the options before the optimizer chooses the best cost plan in the collection. The advantage is that one does not manipulate the optimizer while still being able to remove bad plans, and one only has to calculate hashes for the finished plans, far fewer than calculating them before the tournament like the current solution. Note that this option simply removes the banned option(s) when the tournament has made its selection. There is no guarantee that there are options left after banned plans are removed, and there is no guarantee that any remaining plans are good. Researching how well that would work is work for a future project. Banning only specific parts of the plan (access methods, full plans, or joins) also means that one would have to check hashes for only that type of plan, i.e. access method, full plan, or joins not all at once.

Plan banning would be very precise. One can remove the option of joining table_scan(A) and index_lookup(B)(some_specific_index) with a hash join for instance, where A is left of B. This still allows B to the left of A for a hash-join, or any other join algorithm independent of order, and any other access methods, with a hash-join. Only that one very specific option is removed.

If one has a query that is run often with the same parameters, where specific parameters cause problems, then one can ban a specific plan or sub-plan for those parameters. This can be done by

making the hashes parameter dependent (which they originally were). Then when one calculates the hashes based on parameters then one can ban the plan for only those parameters, allowing the banned plan to be chosen for any other parameters. One can ban index lookup on one table on one parameter. And one can ban a finished plan for the parameters. Or ban another type of sub-plan for the given parameters. This method can ban any sub-plan finished plan as well, but only for a specific combination of parameters. This is most likely too fine-grained for most uses, but the scenario of a query performing badly only for one or a few parameters very much exists. In those cases, this fine-grained method could give DBAs the ability to prevent problematic edge cases while still getting the normal performance for all other parameters. And also manipulating the optimizer as little as possible (the banned plan works for all other parameters).

Banning with hashes removes fewer optimizer options and can be made very specific to remove only one very specific sub-plan combination, or one operation for only a given parameter. This will allow the optimizer to utilize almost all its options while avoiding the problematic plan. Allowing for plan pinning that does as little to hinder the optimizer from finding new better plans as data changes, and for other parameters. Studying the potential of this is a very interesting future project. Additionally, which cases lend themselves better to banning or hinting is an interesting thing to study in the future. Banning can be a less intrusive type of pinning than what is used now. If the hash function remains ineffective the overhead of hash calculation can be a bottleneck causing the hash-based banning to have notable overhead. Join order hints can also ban specific join orders pr. definition, but it is not likely to be such an elegant implementation as with hashes and is far less precise. That does not mean that it is not a viable way to implement a negative join order hint.

Bibliography

[1] K. Ksiazek, *How to Measure Database Performance*, en-US, Jun. 2021. [Online]. Available: https://severalnines.com/blog/how-measure-database-performance/ (visited on 30th Jan. 2023).

[2] *Oracle database performance tuning guide*, Dec. 2003. [Online]. Available: https://docs.oracle.com/cd/B12037_01/server.101/b10752/hintsref.htm.

[3] C. Kiehl, *How to influence query planning in postgresql*, Sep. 2022. [Online]. Available: https://chriskiehl.com/article/query-plan-management.

[4] R. A. Elmasri and S. B. Navathe, 'Chapter 18 strategies for query processing', in *Fundamentals of database systems: Global Edition*. Pearson, 2017, pp. 655–687.

[5] 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, 2008.

[6] D. Kossmann and K. Stocker, 'Iterative dynamic programming: A new class of query optimization algorithms', *ACM Transactions on Database Systems (TODS)*, vol. 25, no. 1, pp. 43–82, 2000.

[7] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie and T. G. Price, 'Access path selection in a relational database management system', in *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '79, Boston, Massachusetts: Association for Computing Machinery, 1979, pp. 23–34, ISBN: 089791001X. DOI: 10.1145/582095.582099. [Online]. Available: https://doi.org/10.1145/582095.582099.

[8] R. A. Elmasri and S. B. Navathe, 'Chapter 19 query optimization', in *Fundamentals of database systems: Global Edition*. Pearson, 2017, pp. 691–728.

[9] N. Bruno, S. Chaudhuri and R. Ramamurthy, 'Power hints for query optimization', in *2009 IEEE 25th International Conference on Data Engineering*, 2009, pp. 469–480. DOI: 10.1109/ICDE.2009.68.

[10] R. Buda, *Five reasons why i avoid database query hints*, Mar. 2021. [Online]. Available: https://www.budaconsulting.com/five-reasons-why-i-avoid-database-query-hints/.

[11] Thesqlsith, *Parameter sensitive plan optimization - sql server*, Nov. 2022. [Online]. Available: https://learn.microsoft.com/en-us/sql/relational-databases/performance/parameter-sensitivity-plan-optimization?view=sql-server-ver16.

[12] WilliamDAssafMSFT, *Set forceplan (transact-sql) - sql server*, Sep. 2022. [Online]. Available: https://learn.microsoft.com/en-us/sql/t-sql/statements/set-forceplan-transact-sql?view=sql-server-ver16.

[13] WilliamDAssafMSFT, *Query store hints - sql server*, Nov. 2022. [Online]. Available: https://learn.microsoft.com/en-us/sql/relational-databases/performance/query-store-hints?view=sql-server-ver16.

[14] *Stored outlines and plan stability*, Jun. 2005. [Online]. Available: https://oracle-base.com/articles/misc/outlines.

[15] *Polardb-x:manage execution plans*, Nov. 2021. [Online]. Available: https://www.alibabacloud.com/help/en/polardb-x/latest/advanced-sql-tuning-manage-execution-plans.

[16] *Alter outline*, 2003. [Online]. Available: https://docs.oracle.com/cd/B14117_01/server.101/b10759/statements_2004.htm.

[17] G. Moerkotte and T. Neumann, 'Dynamic programming strikes back', in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '08, Vancouver, Canada: Association for Computing Machinery, 2008, pp. 539–552, ISBN: 9781605581026. DOI: 10.1145/1376616.1376672. [Online]. Available: https://doi.org/10.1145/1376616.1376672.

[18] Dec. 2022. [Online]. Available: https://docs.aws.amazon.com/dms/latest/sql-server-to-aurora-postgresql-migration-playbook/chap-sql-server-aurora-pg.tuning.queryplanning.html.

[19] J. A. Farrell, Aug. 1995. [Online]. Available: https://www.cs.man.ac.uk/~pjj/farrell/comp3.html.

[20] D. Taniar, H. Y. Khaw, H. C. Tjioe and E. Pardede, 'The use of hints in sql-nested query optimization', *Information Sciences*, vol. 177, no. 12, pp. 2493–2521, 2007. DOI: 10.1016/j.ins.2006.12.015.

[21] T. Neumann and V. Leis, 'Compiling database queries into machine code.', *IEEE Data Eng. Bull.*, vol. 37, no. 1, pp. 3–11, 2014.

[22] A. Rayabhari, *Compilation-based execution engine for a database*, Dec. 2020. [Online]. Available: https://www.cs.cornell.edu/courses/cs6120/2020fa/blog/db-compiler/.

[23] A. Hallem Iversen and S. Øverland, *Compiling expressions in mysql*, Sep. 2022. [Online]. Available: https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/3020760.

[24] S. Manegold, P. A. Boncz and M. L. Kersten, 'Optimizing database architecture for the new bottleneck: Memory access', *The VLDB Journal*, vol. 9, no. 3, pp. 231–246, 2000. DOI: 10.1007/s007780000031.

[25] S. Vellev, 'Review of algorithms for the join ordering problems in database query optimization', *Information Technologies and control*, vol. 1, pp. 32–40, 2009.

[26] L. Shapiro, D. Maier, P. Benninghoff *et al.*, 'Exploiting upper and lower bounds in top-down query optimization', Feb. 2001, pp. 20–33, ISBN: 0-7695-1140-6. DOI: 10.1109/IDEAS.2001.938068.

[27] P. Butey, S. Meshram and R. Sonolikar, 'Query optimization by genetic algorithm', *Journal of Information Technology And Engineering*, Jan-June, vol. Vol.3, no. No. 1, pp. 44–51, Jun. 2012. DOI: ISSN:2229-7421.

[28] K. Bennett, M. Ferris and Y. Ioannidis, 'A genetic algorithm for database query optimization', Feb. 1970.

[29] V. Leis, B. Radke, A. Gubichev, A. Kemper and T. Neumann, 'Cardinality estimation done right: Index-based join sampling.', in *Cidr*, 2017.

[30] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper and T. Neumann, 'How good are query optimizers, really?', *Proc. VLDB Endow.*, vol. 9, no. 3, pp. 204–215, Nov. 2015, ISSN: 2150-8097. DOI: 10.14778/2850583.2850594. [Online]. Available: https://doi.org/10.14778/2850583.2850594.

[31] S. Malik and J. Finnerty, *Introduction to aurora postgresql query plan management*, May 2019. [Online]. Available: https://aws.amazon.com/blogs/database/introduction-to-aurora-postgresql-query-plan-management/.

[32] D. Korotkevitch, 'Plan caching', *Pro SQL Server Internals*, pp. 491–515, Dec. 2016. DOI: 10.1007/978-1-4842-1964-5_26.

[33] 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*, 2012, pp. 402–413. DOI: 10.1109/ICDE.2012.57.

[34] *Sql tuning guide*, Dec. 2021. [Online]. Available: https://docs.oracle.com/en/database/oracle/oracle-database/19/tgsql/improving-rwp-cursor-sharing.html#GUID-971F4652-3950-4662-82DE-713DDEED317C.

[35] *Mysql 8.0 reference manual - 8.9.3 optimizer hints*, Apr. 2018. [Online]. Available: https://dev.mysql.com/doc/refman/8.0/en/optimizer-hints.html.

[36] *Oracle database performance tuning guide*, Dec. 2003. [Online]. Available: https://docs.oracle.com/cd/B14117_01/server.101/b10752/outlines.htm.

[37] rwestMSFT, *Query hints (transact-sql) - sql server*, Oct. 2022. [Online]. Available: https://learn.microsoft.com/en-us/sql/t-sql/queries/hints-transact-sql-query?view=sql-server-ver16.

[38] J. L. Brunvoll and L. Willa, *Hinted caching and pinning of query execution plans in mysql*, 2023. [Online]. Available: non-published%20project%20from%20TDT4501%20in%202023.

[39] M. Pang, *Fast query performance with mysql hypergraph optimizer for heatwave*, Jan. 2023. [Online]. Available: https://blogs.oracle.com/mysql/post/fast-query-performance-with-mysql-hypergraph-optimizer-for-heatwave.

[40] P. Fender and G. Moerkotte, 'Counter strike: Generic top-down join enumeration for hypergraphs', *Proc. VLDB Endow.*, vol. 6, no. 14, pp. 1822–1833, Sep. 2013, ISSN: 2150-8097. DOI: 10.14778/2556549.2556565. [Online]. Available: https://doi.org/10.14778/2556549.2556565.

[41] S. Borzsony, D. Kossmann and K. Stocker, 'The skyline operator', in *Proceedings 17th International Conference on Data Engineering*, 2001, pp. 421–430. DOI: 10.1109/ICDE.2001.914855.

[42] *Index hints: How to force query plans*, Jun. 2022. [Online]. Available: https://mariadb.com/kb/en/index-hints-how-to-force-query-plans/.

[43] I. Gilfillan, Nov. 2022. [Online]. Available: https://mariadb.org/wp-content/uploads/2022/11/MariaDBServerKnowledgeBase.pdf.

[44] J. Reock, *Enterprisedb vs. postgresql (postgres): Availability, dr, security, & amp; performance*, Feb. 2020. [Online]. Available: https://www.openlogic.com/blog/enterprisedb-vs-postgres.

[45] *Optimizerhintsdiscussion*, Jul. 2015. [Online]. Available: https://wiki.postgresql.org/wiki/OptimizerHintsDiscussion.

[46] May 2023. [Online]. Available: https://www.postgresql.org/docs/current/runtime-config-query.html.

[47] V. Markovtsev, *How we optimized postgresql queries 100x*, Mar. 2022. [Online]. Available: https://towardsdatascience.com/how-we-optimized-postgresql-queries-100x-ff52555eabe.

[48] 2012. [Online]. Available: https://pghintplan.osdn.jp/pg_hint_plan.html.

[49] 2021. [Online]. Available: https://postgrespro.com/docs/enterprise/9.6/pg-hint-plan#pg-hint-plan-seealso.

[50] 2023. [Online]. Available: https://catalog.us-east-1.prod.workshops.aws/workshops/098605dc-8eee-4e84-85e9-c5c6c9e43de2/en-US/lab3-query-plan-mgmt.

[51] *Polardb for mysql:statement outline*, Sep. 2022. [Online]. Available: https://www.alibabacloud.com/help/en/polardb-for-mysql/latest/statement-outline.

[52] *Stored outlines and plan stability*, Jun. 2015. [Online]. Available: https://oracle-base.com/articles/misc/outlines.

[53] *Prepare*, Nov. 2022. [Online]. Available: https://www.postgresql.org/docs/current/sql-prepare.html.

[54] *Polardb for mysql:polardb for mysql 8.0.2*, Oct. 2022. [Online]. Available: https://www.alibabacloud.com/help/en/polardb-for-mysql/latest/802-release-note1.

[55] F. Li, 'Cloud-native database systems at alibaba: Opportunities and challenges', *Proc. VLDB Endow.*, vol. 12, no. 12, pp. 2263–2272, Aug. 2019, ISSN: 2150-8097. DOI: 10.14778/3352063.3352141. [Online]. Available: https://doi.org/10.14778/3352063.3352141.

[56] *Polardb-x:execution plan management*, Nov. 2022. [Online]. Available: https://www.alibabacloud.com/help/en/polardb-x/latest/sql-tuning-guide-manage-execution-plans.

[57] U. Trivedi, *Unordered_map in c++ stl*, Jan. 2023. [Online]. Available: https://www.geeksforgeeks.org/unordered_map-in-cpp-stl/.

[58] Jul. 2022. [Online]. Available: https://en.cppreference.com/w/cpp/chrono/steady_clock.

Appendix

A Experiment Results Tables

Non-strict Join Order Hinted

	Not Pinned	joh 1	joh 2	joh 3	joh 4	joh 5
1a	0.00007604	0.00010274	0.00025756	0.00033237	0.00045022	0.00056953
2a	0.00014610	0.00015635	0.00035720	0.00044504	0.00058657	0.00073982
8c	0.00045834	0.00049130	0.00132514	0.00176561	0.00219042	0.00266949
13b	0.00138965	0.00122070	0.00415596	0.00560174	0.00694722	0.00846031
14b	0.00091054	0.00084000	0.00235256	0.00315822	0.00395999	0.00479940
16a	0.00146480	0.00106015	0.00311053	0.00419428	0.00522761	0.00626599
18c	0.00059179	0.00060703	0.00165691	0.00217325	0.00269151	0.00317916
21c	0.00187566	0.00202388	0.00609915	0.00759475	0.01007656	0.01277353
27b	0.03118467	0.03195418	0.10651393	0.15144161	0.19091376	0.23294231
31a	0.05278417	0.05247117	0.17694275	0.23457192	0.29936609	0.36092272

Table 1: Non-strict join order pinning with average optimize time pr. query in seconds (8 decimals)

Strict Join Order Hinted

	Not Pinned	joh strict 1	joh strict 2	joh strict 3	joh strict 4	joh strict 5
1a	0.00007604	0.00009949	0.00024386	0.00031625	0.00041687	0.00054441
2a	0.00014610	0.00015230	0.00034389	0.00041458	0.00054235	0.00071458
8c	0.00045834	0.00048121	0.00129497	0.00170399	0.00208885	0.00255190
13b	0.00138965	0.00122624	0.00400297	0.00537705	0.00676312	0.00822362
14b	0.00091054	0.00083150	0.00232534	0.00310435	0.00386782	0.00474699
16a	0.00146480	0.00106847	0.00306559	0.00409862	0.00510308	0.00616363
18c	0.00059179	0.00061406	0.00164674	0.00215110	0.00266982	0.00313659
21c	0.00187566	0.00209930	0.00558998	0.00742758	0.00965249	0.01168477
27b	0.03118467	0.03353248	0.10570520	0.14450057	0.18215694	0.22566842
31a	0.05278417	0.05292088	0.16986657	0.22785344	0.28606308	0.34746816

Table 2: Strict join order pinning with average optimize time pr. query in seconds (8 decimals)

Non-strict Hash-pinning

	Not Pinned	non-s-hash1	non-s-hash2	non-s-hash3	non-s-hash4	non-s-hash5
1a	0.00007604	0.00388667	0.00415602	0.00433070	0.00428960	0.00599643
2a	0.00014610	0.00625113	0.00598066	0.00631617	0.00634709	0.00829979
8c	0.00045834	0.05945830	0.06186619	0.06220038	0.06044961	0.06808670
13b	0.00138965	0.30401604	0.29686966	0.29616864	0.30634463	0.30920206
14b	0.00091054	0.17297635	0.13449225	0.13551452	0.13924437	0.15020424
16a	0.00146480	0.30189054	0.19034940	0.19576994	0.19597356	0.20235722
18c	0.00059179	0.08722871	0.07924498	0.08160004	0.08040512	0.08237805
21c	0.00187566	0.46954787	0.45346856	0.46163540	0.46069898	0.49257645
27b	0.03118467	9.99535660	9.56010080	9.59806890	9.58130820	9.94075650
31a	0.05278417	15.80094800	15.50187300	15.59926200	15.61298800	16.07739900

Table 3: Non-strict hash-pinning with average optimize time pr. query in seconds (8 decimals)

Non-strict Hash-pinning Light

	Not Pinned	hash no-li 1	hash no-li 2	hash no-li 3	hash no-li 4	hash no-li 5
1a	0.00007604	0.00199457	0.00208431	0.00212945	0.00212607	0.00296173
2a	0.00014610	0.00310152	0.00313752	0.00330964	0.00330465	0.00430977
8c	0.00045834	0.02994652	0.02961822	0.02964869	0.03047846	0.03328481
13b	0.00138965	0.14110447	0.14892312	0.14619651	0.14568697	0.15190486
14b	0.00091054	0.08375054	0.06437757	0.06474811	0.06531035	0.07046171
16a	0.00146480	0.18025339	0.09434670	0.09624349	0.09692447	0.10036085
18c	0.00059179	0.03999774	0.03878960	0.03863573	0.03866504	0.03964348
21c	0.00187566	0.23860699	0.22078784	0.22525415	0.22532161	0.24151040
27b	0.03118467	4.83572530	4.54897360	4.56765290	4.57309840	4.75026680
31a	0.05278417	7.71531010	7.56049520	7.65268180	7.66271560	7.83876460

Table 4: Non-strict hash-pinning light with average optimize time pr. query in seconds (8 decimals)

Strict Hash-pinning

	Not Pinned	hash strict 1	hash strict 2	hash strict 3	hash strict 4	hash strict 5
1a	0.00007604	0.00054476	0.00063859	0.00124513	0.00126082	0.00779753
2a	0.00014610	0.00064959	0.00136516	0.00201839	0.00203198	0.00812444
8c	0.00045834	0.00234548	0.00471576	0.00945266	0.00951371	0.03496707
13b	0.00138965	0.00486011	0.01881755	0.02621126	0.02561079	0.07995531
14b	0.00091054	0.00240917	0.00254836	0.00564390	0.00560503	0.03501410
16a	0.00146480	0.00273839	0.00535778	0.00814875	0.00805072	0.06206146
18c	0.00059179	0.00139715	0.00147734	0.00172151	0.00174511	0.00312051
21c	0.00187566	0.00417573	0.00804167	0.00848080	0.00879001	0.03536810
27b	0.03118467	0.09278502	0.09186636	0.09201908	0.09082227	0.11058203
31a	0.05278417	0.03793993	0.04235144	0.04297838	0.04271438	0.53887797

Table 5: Strict hash-pinning with average optimize time pr. query in seconds (8 decimals)

Strict Hash-pinning light

	Not Pinned	hash str li 1	hash str li 2	hash str li 3	hash str li 4	hash str li 5
1a	0.00007604	0.00031870	0.00036901	0.00066557	0.00071822	0.00378897
2a	0.00014610	0.00037403	0.00072473	0.00105508	0.00105367	0.00408212
8c	0.00045834	0.00128344	0.00247641	0.00488171	0.00487731	0.01689346
13b	0.00138965	0.00241830	0.00947388	0.01313452	0.01297358	0.03979770
14b	0.00091054	0.00127523	0.00135029	0.00283668	0.00283972	0.01753249
16a	0.00146480	0.00143421	0.00266592	0.00400731	0.00400299	0.03065149
18c	0.00059179	0.00077125	0.00081664	0.00094274	0.00094550	0.00167227
21c	0.00187566	0.00212463	0.00411581	0.00416367	0.00417912	0.01656290
27b	0.03118467	0.04440889	0.04386156	0.04393015	0.04345496	0.05420129
31a	0.05278417	0.01886361	0.02084633	0.02120092	0.02148645	0.25700843

Table 6: Strict hash-pinning light with average optimize time pr. query in seconds (8 decimals)

B Results With Execute Time

ID	queryname	optimized time	execution time
Not pinned	1a	7.603681e-05	0.010360216100000002
Not pinned	2a	0.00014610060000000002	0.3177275
Not pinned	8c	0.00045834177000000004	63.134231
Not pinned	13b	0.0013896496000000002	2.2337808999999997
Not pinned	14b	0.0009105420899999997	0.9214193500000001
Not pinned	16a	0.0014648007999999999	2.0733659000000007
Not pinned	18c	0.00059178673	15.398271
Not pinned	21c	0.0018756566	0.013377074
Not pinned	27b	0.031184673	0.0058865016
Not pinned	31a	0.05278417	2.6160987
Hash-pinned	1a	0.0005447600800000001	0.010791052
Hash-pinned	2a	0.0006495884899999999	0.31715695
Hash-pinned	8c	0.0023454747	63.211162
Hash-pinned	13b	0.0048601137	1.2469494
Hash-pinned	14b	0.0024091708	0.9225325999999999
Hash-pinned	16a	0.0027383849000000003	2.0603129
Hash-pinned	18c	0.0013971528999999997	15.307312999999999
Hash-pinned	21c	0.0041757324	0.013867138
Hash-pinned	27b	0.092785017	0.011441366000000001
Hash-pinned	31a	0.037939934	2.6282389999999998
Hash-pinned	1a	0.0006385943600000002	0.010208270399999999
Hash-pinned	2a	0.0013651619	0.31798919999999997
Hash-pinned	8c	0.0047157627	63.19897000000001
Hash-pinned	13b	0.018817553	1.2379904000000002
Hash-pinned	14b	0.0025483551999999996	0.9151960399999999
Hash-pinned	16a	0.005357778900000001	2.0546416
Hash-pinned	18c	0.0014773411	15.293283
Hash-pinned	21c	0.0080416657	0.013540234
Hash-pinned	27b	0.09186635599999998	0.011487074000000002
Hash-pinned	31a	0.042351437	2.6568433
Hash-pinned	1a	0.0012451329999999998	0.010363814999999998
Hash-pinned	2a	0.0020183895	0.31348864
Hash-pinned	8c	0.0094526604	62.905871
Hash-pinned	13b	0.026211255	1.252225
Hash-pinned	14b	0.005643902700000001	0.9179790300000001
Hash-pinned	16a	0.008148744800000001	2.0845026
Hash-pinned	18c	0.0017215091999999998	15.38616
Hash-pinned	21c	0.0084808005	0.013636057000000002
Hash-pinned	27b	0.09201908199999999	0.011636548
Hash-pinned	31a	0.04297837499999999	2.622735
Hash-pinned	1a	0.0012608183	0.010405946400000001
Hash-pinned	2a	0.0020319796000000004	0.31427297000000004
Hash-pinned	8c	0.0095137124	62.890532999999984
Hash-pinned	13b	0.025610789999999994	1.2408658
Hash-pinned	14b	0.0056050295000000016	0.9169208799999998
Hash-pinned	16a	0.008050714899999999	2.0697010000000002
Hash-pinned	18c	0.0017451059	15.257406999999999
Hash-pinned	21c	0.0087900143	0.013731699
Hash-pinned	27b	0.09082226800000001	0.011391070999999997
Hash-pinned	31a	0.042714384	2.6171534
Hash-pinned	1a	0.007797527299999999	0.011375662
Hash-pinned	2a	0.0081244444	0.31742117999999997
Hash-pinned	8c	0.03496707	63.11442499999999
Hash-pinned	13b	0.07995530599999999	1.2529032
~~Hash-pinned~~	~~14b~~	~~0.035014097~~	~~0.91737412~~
Hash-pinned	16a	0.062061460000000006	2.0586998
Hash-pinned	18c	0.0031205046000000003	15.350115000000002
Hash-pinned	21c	0.035368095	0.014141009
Hash-pinned	27b	0.11058203000000001	0.011303187000000003
Hash-pinned	31a	0.5388779699999999	2.6629175000000007

C Join Order Hints Experiment Setup

Format of JOIN_ORDER() hints when pinning 5 plans:

```
/*+ JOIN_ORDER((join_order_set1),(join_order_set2),
(join_order_set3),(join_order_set4),(join_order_set5))*/
```

Format of JOIN_ORDER() hints when pinning 4 plans:

```
/*+ JOIN_ORDER((join_order_set1),(join_order_set2),
(join_order_set3),(join_order_set4))*/
```

And so on.

Here is a real example:

```
/*+ JOIN_ORDER((_lp, _lp, _lp, it, mi_idx, rp_, mc, rp_, ct, rp_, t),
(_lp, _lp, _lp, it, mi_idx, rp_, mc, rp_, ct, rp_, t),
(t,_lp, _lp, _lp, it, mi_idx, rp_, mc, rp_, ct, rp_),
(t,_lp, _lp, _lp, mi_idx, it, rp_, mc, rp_, ct, rp_),
(t,_lp, _lp, mc, _lp, it, mi_idx, rp_, rp_, ct, rp_))*/
```

The syntax uses "_lp" to denote a left parenthesis and rp_ to denote right parenthesis.

Here is how this one would be prepared for the join order hint experiment (query 1a)s:

```
PREPARE joh_1_5 FROM "SELECT
/*+ JOIN_ORDER((_lp, _lp, _lp, it, mi_idx, rp_, mc, rp_, ct, rp_, t),
(_lp, _lp, _lp, it, mi_idx, rp_, mc, rp_, ct, rp_, t),
(t,_lp, _lp, _lp, it, mi_idx, rp_, mc, rp_, ct, rp_),
(t,_lp, _lp, _lp, mi_idx, it, rp_, mc, rp_, ct, rp_),
(t,_lp, _lp, mc, _lp, it, mi_idx, rp_, rp_, ct, rp_))*/
MIN(mc.note) AS production_note,
MIN(t.title) AS movie_title,
MIN(t.production_year) AS movie_year
FROM company_type AS ct,
info_type AS it,
movie_companies AS mc,
movie_info_idx AS mi_idx,
title AS t
WHERE ct.kind = 'production companies'
AND it.info = 'top 250 rank'
AND mc.note NOT LIKE '%(as Metro-Goldwyn-Mayer Pictures)%'
AND (mc.note LIKE '%(co-production)%'
OR mc.note LIKE '%(presents)%')
AND ct.id = mc.company_type_id
AND t.id = mc.movie_id
AND t.id = mi_idx.movie_id
AND mc.movie_id = mi_idx.movie_id
AND it.id = mi_idx.info_type_id";

EXECUTE joh_1_5;

PREPARE joh_1_4 FROM "SELECT
/*+ JOIN_ORDER((_lp, _lp, _lp, it, mi_idx, rp_, mc, rp_, ct, rp_, t),
```

```
(_lp, _lp, _lp, it, mi_idx, rp_, mc, rp_, ct, rp_, t),
(t,_lp, _lp, _lp, it, mi_idx, rp_, mc, rp_, ct, rp_),
(t,_lp, _lp, _lp, mi_idx, it, rp_, mc, rp_, ct, rp_))*/
MIN(mc.note) AS production_note,
MIN(t.title) AS movie_title,
MIN(t.production_year) AS movie_year
FROM company_type AS ct,
info_type AS it,
movie_companies AS mc,
movie_info_idx AS mi_idx,
title AS t
WHERE ct.kind = 'production companies'
AND it.info = 'top 250 rank'
AND mc.note NOT LIKE '%(as Metro-Goldwyn-Mayer Pictures)%'
AND (mc.note LIKE '%(co-production)%'
OR mc.note LIKE '%(presents)%')
AND ct.id = mc.company_type_id
AND t.id = mc.movie_id
AND t.id = mi_idx.movie_id
AND mc.movie_id = mi_idx.movie_id
AND it.id = mi_idx.info_type_id";

EXECUTE joh_1_4;


PREPARE joh_1_3 FROM "SELECT
/*+ JOIN_ORDER((_lp, _lp, _lp, it, mi_idx, rp_, mc, rp_, ct, rp_, t),
(_lp, _lp, _lp, it, mi_idx, rp_, mc, rp_, ct, rp_, t),
(t,_lp, _lp, _lp, it, mi_idx, rp_, mc, rp_, ct, rp_))*/
MIN(mc.note) AS production_note,
MIN(t.title) AS movie_title,
MIN(t.production_year) AS movie_year
FROM company_type AS ct,
info_type AS it,
movie_companies AS mc,
movie_info_idx AS mi_idx,
title AS t
WHERE ct.kind = 'production companies'
AND it.info = 'top 250 rank'
AND mc.note NOT LIKE '%(as Metro-Goldwyn-Mayer Pictures)%'
AND (mc.note LIKE '%(co-production)%'
OR mc.note LIKE '%(presents)%')
AND ct.id = mc.company_type_id
AND t.id = mc.movie_id
AND t.id = mi_idx.movie_id
AND mc.movie_id = mi_idx.movie_id
AND it.id = mi_idx.info_type_id";

EXECUTE joh_1_3;


PREPARE joh_1_2 FROM "SELECT
/*+ JOIN_ORDER((_lp, _lp, _lp, it, mi_idx, rp_, mc, rp_, ct, rp_, t),
(_lp, _lp, _lp, it, mi_idx, rp_, mc, rp_, ct, rp_, t))*/
MIN(mc.note) AS production_note,
MIN(t.title) AS movie_title,
MIN(t.production_year) AS movie_year
FROM company_type AS ct,
info_type AS it,
```

```
movie_companies AS mc,
movie_info_idx AS mi_idx,
title AS t
WHERE ct.kind = 'production companies'
AND it.info = 'top 250 rank'
AND mc.note NOT LIKE '%(as Metro-Goldwyn-Mayer Pictures)%'
AND (mc.note LIKE '%(co-production)%'
OR mc.note LIKE '%(presents)%')
AND ct.id = mc.company_type_id
AND t.id = mc.movie_id
AND t.id = mi_idx.movie_id
AND mc.movie_id = mi_idx.movie_id
AND it.id = mi_idx.info_type_id";

EXECUTE joh_1_2;


SELECT
/*+ JOIN_ORDER((_lp, _lp, _lp, it, mi_idx, rp_, mc, rp_, ct, rp_, t))*/
MIN(mc.note) AS production_note,
MIN(t.title) AS movie_title,
MIN(t.production_year) AS movie_year
FROM company_type AS ct,
info_type AS it,
movie_companies AS mc,
movie_info_idx AS mi_idx,
title AS t
WHERE ct.kind = 'production companies'
AND it.info = 'top 250 rank'
AND mc.note NOT LIKE '%(as Metro-Goldwyn-Mayer Pictures)%'
AND (mc.note LIKE '%(co-production)%'
OR mc.note LIKE '%(presents)%')
AND ct.id = mc.company_type_id
AND t.id = mc.movie_id
AND t.id = mi_idx.movie_id
AND mc.movie_id = mi_idx.movie_id
AND it.id = mi_idx.info_type_id;
```

note how the last join order set was removed each time, and that the single one does not need to be prepared.

D Hash Experiments Setup

The format for the hash file rows is the following: statement digest, some_number, comma-separated-hashes

The number currently has no real function.

Here are the five hashes for pinning five plans for 1a.

f6cd75876db7be097f85efb643fb334ef145bf5e054cb05f3780055ef1cdc622, 1, 0x7afdc39222d0009b, 0xd5be7582b166ce1e, 0x829cc967309b7158, 0xf0f1f9f814450838, 0x83503d40fca467e9, 0x112d0686767921ce, 0x455bc36bca164ac4, 0xb82f68c30c71032a, 0x7794c2dfe10e6984, 0x452f407d54d55650, 0xb9439d37762679d9

f6cd75876db7be097f85efb643fb334ef145bf5e054cb05f3780055ef1cdc622, 1, 0xd5be7582b166ce1e, 0x829cc967309b7158, 0xf0f1f9f814450838, 0x83503d40fca467e9, 0x112d0686767921ce, 0x7afdc39222d0009b, 0xd173019b03bef7db, 0xb82f68c30c71032a, 0x79f57cc2c15deeb3, 0xa13b9876a6eca299, 0x12d23e682a326ecd

f6cd75876db7be097f85efb643fb334ef145bf5e054cb05f3780055ef1cdc622, 1, 0x5ca7771dda5afc14, 0xd5be7582b166ce1e,

0x829cc967309b7158, 0xf0f1f9f814450838, 0x83503d40fca467e9, 0x112d0686767921ce, 0x7afdc39222d0009b, 0xd173019b03bef7db, 0x15beecb05dd35569, 0x71c2767159d1a88f, 0x13b242f300c51116

f6cd75876db7be097f85efb643fb334ef145bf5e054cb05f3780055ef1cdc622, 1, 0x5ca7771dda5afc14, 0x2ecf86f0c276c308, 0xd5be7582b166ce1e, 0x1de15e357f696706, 0x83503d40fca467e9, 0xd0e09ba4367f75f3, 0x7afdc39222d0009b, 0xb040dd0c248ecdf7, 0xbfed6446d3bf80fc, 0x962afc6f42bcfd26, 0x4e843cdbbf0a9dc4

f6cd75876db7be097f85efb643fb334ef145bf5e054cb05f3780055ef1cdc622, 1, 0x5ca7771dda5afc14, 0x057e13f683cbea95, 0xd5be7582b166ce1e, 0x829cc967309b7158, 0xf0f1f9f814450838, 0x2f8008f074324e62, 0x7afdc39222d0009b, 0xc9377032ccc84994, 0xa542183b2c58d7cd, 0x6bbb8042812e41b5, 0x57eb1bdcf661dba1

for 4 plans

f6cd75876db7be097f85efb643fb334ef145bf5e054cb05f3780055ef1cdc622, 1, 0x7afdc39222d0009b, 0xd5be7582b166ce1e, 0x829cc967309b7158, 0xf0f1f9f814450838, 0x83503d40fca467e9, 0x112d0686767921ce, 0x455bc36bca164ac4, 0xb82f68c30c71032a, 0x7794c2dfe10e6984, 0x452f407d54d55650, 0xb9439d37762679d9

f6cd75876db7be097f85efb643fb334ef145bf5e054cb05f3780055ef1cdc622, 1, 0xd5be7582b166ce1e, 0x829cc967309b7158, 0xf0f1f9f814450838, 0x83503d40fca467e9, 0x112d0686767921ce, 0x7afdc39222d0009b, 0xd173019b03bef7db, 0xb82f68c30c71032a, 0x79f57cc2c15deeb3, 0xa13b9876a6eca299, 0x12d23e682a326ecd

f6cd75876db7be097f85efb643fb334ef145bf5e054cb05f3780055ef1cdc622, 1, 0x5ca7771dda5afc14, 0xd5be7582b166ce1e, 0x829cc967309b7158, 0xf0f1f9f814450838, 0x83503d40fca467e9, 0x112d0686767921ce, 0x7afdc39222d0009b, 0xd173019b03bef7db, 0x15beecb05dd35569, 0x71c2767159d1a88f, 0x13b242f300c51116

f6cd75876db7be097f85efb643fb334ef145bf5e054cb05f3780055ef1cdc622, 1, 0x5ca7771dda5afc14, 0x2ecf86f0c276c308, 0xd5be7582b166ce1e, 0x1de15e357f696706, 0x83503d40fca467e9, 0xd0e09ba4367f75f3, 0x7afdc39222d0009b, 0xb040dd0c248ecdf7, 0xbfed6446d3bf80fc, 0x962afc6f42bcfd26, 0x4e843cdbbf0a9dc4

for 3 plans

f6cd75876db7be097f85efb643fb334ef145bf5e054cb05f3780055ef1cdc622, 1, 0x7afdc39222d0009b, 0xd5be7582b166ce1e, 0x829cc967309b7158, 0xf0f1f9f814450838, 0x83503d40fca467e9, 0x112d0686767921ce, 0x455bc36bca164ac4, 0xb82f68c30c71032a, 0x7794c2dfe10e6984, 0x452f407d54d55650, 0xb9439d37762679d9

f6cd75876db7be097f85efb643fb334ef145bf5e054cb05f3780055ef1cdc622, 1, 0xd5be7582b166ce1e, 0x829cc967309b7158, 0xf0f1f9f814450838, 0x83503d40fca467e9, 0x112d0686767921ce, 0x7afdc39222d0009b, 0xd173019b03bef7db, 0xb82f68c30c71032a, 0x79f57cc2c15deeb3, 0xa13b9876a6eca299, 0x12d23e682a326ecd

f6cd75876db7be097f85efb643fb334ef145bf5e054cb05f3780055ef1cdc622, 1, 0x5ca7771dda5afc14, 0xd5be7582b166ce1e, 0x829cc967309b7158, 0xf0f1f9f814450838, 0x83503d40fca467e9, 0x112d0686767921ce, 0x7afdc39222d0009b, 0xd173019b03bef7db, 0x15beecb05dd35569, 0x71c2767159d1a88f, 0x13b242f300c51116

for 2 plans

f6cd75876db7be097f85efb643fb334ef145bf5e054cb05f3780055ef1cdc622, 1, 0x7afdc39222d0009b, 0xd5be7582b166ce1e, 0x829cc967309b7158, 0xf0f1f9f814450838, 0x83503d40fca467e9, 0x112d0686767921ce, 0x455bc36bca164ac4, 0xb82f68c30c71032a, 0x7794c2dfe10e6984, 0x452f407d54d55650, 0xb9439d37762679d9

f6cd75876db7be097f85efb643fb334ef145bf5e054cb05f3780055ef1cdc622, 1, 0xd5be7582b166ce1e, 0x829cc967309b7158, 0xf0f1f9f814450838, 0x83503d40fca467e9, 0x112d0686767921ce, 0x7afdc39222d0009b, 0xd173019b03bef7db, 0xb82f68c30c71032a, 0x79f57cc2c15deeb3, 0xa13b9876a6eca299, 0x12d23e682a326ecd

and for one plan

f6cd75876db7be097f85efb643fb334ef145bf5e054cb05f3780055ef1cdc622, 1, 0x7afdc39222d0009b, 0xd5be7582b166ce1e, 0x829cc967309b7158, 0xf0f1f9f814450838, 0x83503d40fca467e9, 0x112d0686767921ce, 0x455bc36bca164ac4, 0xb82f68c30c71032a, 0x7794c2dfe10e6984, 0x452f407d54d55650, 0xb9439d37762679d9

Note that also here one removes the last one each time. All the hashes pin a valid plan, but for the sake of comparability, the same plans are pinned in both for all multi-runs. Also, the spaces have to be removed, they are there for latex formatting and should not be there when the hashes are used.

E Attachments

The attachments contain two important things, these are:

- The diff.txt can be used to get the code from the experiments.
- The experiment_utilities folder contains queries, hashes, and experiment scripts that will allow one to reproduce the experiments conducted here.
- The ReadMe.txt file contains a link to GitHub and some other useful information to help reproduce the experiments.

F Hashes for 31a

plan 1:

81409a63a42c076dc4b892cbeacfa4ca980a9e9235a4437ba8a6602cd68b2ca6, 31, 0x891748ea4e3af00a, 0x6c1ce0d703bf57a5, 0xa29d3c79e7af4874, 0x83503d40fca467e9, 0x2e7d9725569e9160, 0x338e03d42c97b4ab, 0x7041a6defed1ea77, 0x6ed654d4c0a69bf0, 0xac70e7697292e22d, 0xb6427aa802d078bc, 0x8610c642c5fc5a11, 0xcd9f5b4e61e5d2ae, 0xe668460ceeb77ab2, 0x72ff6c0b661fe089, 0x4cd69ecd9212b2ca, 0x377b5ee16dc4a358, 0x12bc24e3510d774d, 0xffefd63862ac2513, 0x2faf874e00be15ec, 0x7e0c78bbb903f9d4, 0xd509bdf79765f260, 0x5996c893d9b75e42, 0x18d996dca03567f2

plan 2:

81409a63a42c076dc4b892cbeacfa4ca980a9e9235a4437ba8a6602cd68b2ca6, 31, 0x338e03d42c97b4ab, 0x891748ea4e3af00a, 0x6c1ce0d703bf57a5, 0xa29d3c79e7af4874, 0x83503d40fca467e9, 0x2e7d9725569e9160, 0x42a346ce6ae33ecc, 0x6ed654d4c0a69bf0, 0x9091c1ad8f3bca91, 0xe8acf05d0c4d59c1, 0x45beda81d0db04f2, 0xcd9f5b4e61e5d2ae, 0x10a5c9791ee52b19, 0x72ff6c0b661fe089, 0x6df904849567a371, 0x377b5ee16dc4a358, 0x60831656ddc97c71, 0xffefd63862ac2513, 0xc9f03d5fb0353106, 0x7e0c78bbb903f9d4, 0x3579b5a485b621ca, 0x62878be5e6eab19e, 0x772817d4be910d13

plan 3:

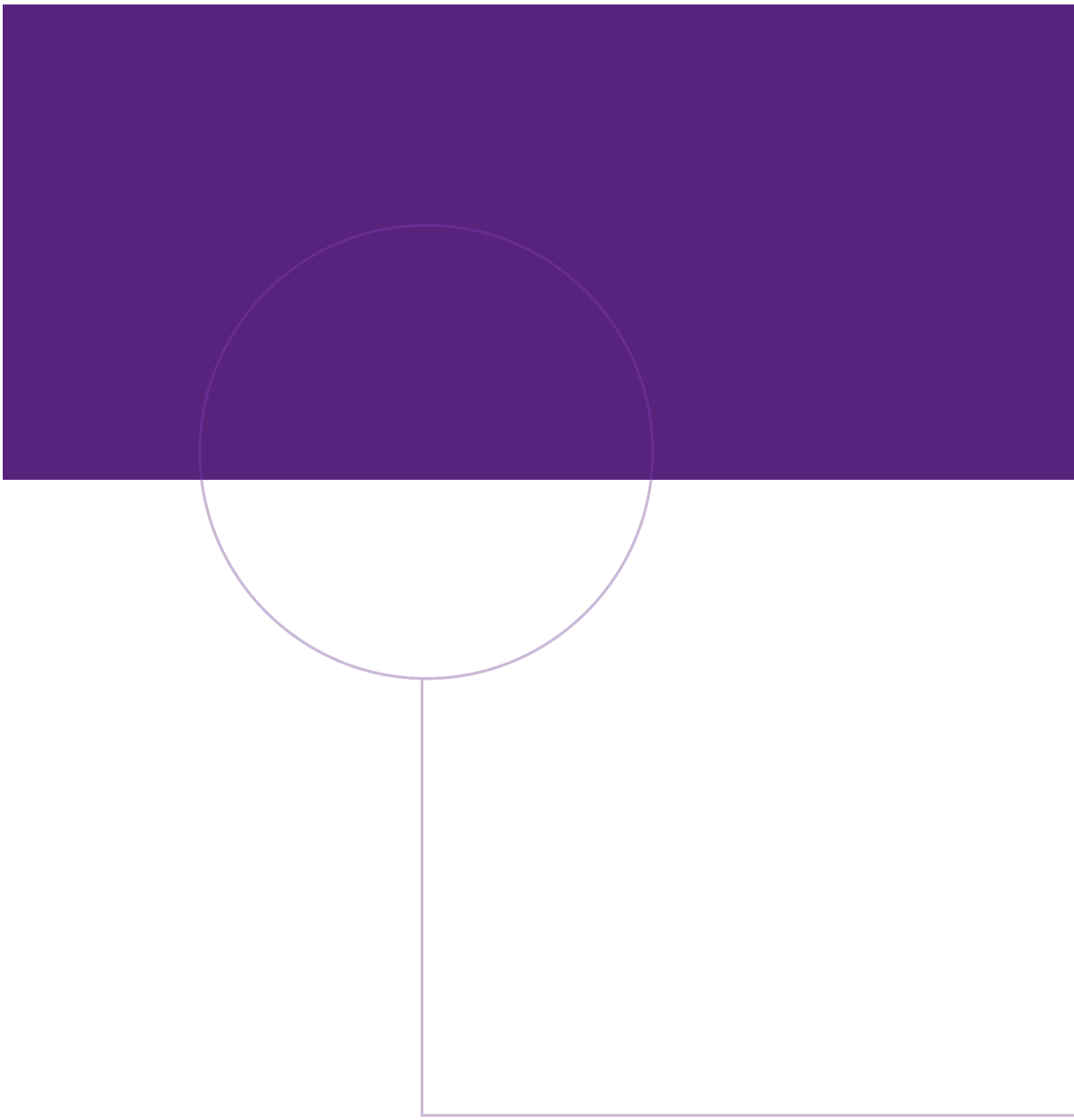
81409a63a42c076dc4b892cbeacfa4ca980a9e9235a4437ba8a6602cd68b2ca6, 31, 0xb6427aa802d078bc, 0x891748ea4e3af00a, 0x6c1ce0d703bf57a5, 0xa29d3c79e7af4874, 0x83503d40fca467e9, 0x2e7d9725569e9160, 0x338e03d42c97b4ab, 0x7041a6defed1ea77, 0x6ed654d4c0a69bf0, 0xac70e7697292e22d, 0x36cbd12e5440ff2b, 0xcd9f5b4e61e5d2ae, 0x1a37958c4eb82d26, 0x72ff6c0b661fe089, 0x12485a344ee52e52, 0x377b5ee16dc4a358, 0xfedb91009fa70145, 0xffefd63862ac2513, 0x7f48a76e28ac9dc0, 0x7e0c78bbb903f9d4, 0x1897c678c7937a45, 0x8ef3e4533eb910d2, 0x9cf6770d85e5624d

plan 4:

81409a63a42c076dc4b892cbeacfa4ca980a9e9235a4437ba8a6602cd68b2ca6, 31, 0xb2f9f9d4d507ba1c, 0x891748ea4e3af00a, 0x6c1ce0d703bf57a5, 0xa29d3c79e7af4874, 0x83503d40fca467e9, 0x2e7d9725569e9160, 0x338e03d42c97b4ab, 0x7041a6defed1ea77, 0x6ed654d4c0a69bf0, 0xac70e7697292e22d, 0xb6427aa802d078bc, 0x8610c642c5fc5a11, 0xcd9f5b4e61e5d2ae, 0xe668460ceeb77ab2, 0x72ff6c0b661fe089, 0x4cd69ecd9212b2ca, 0xdf2b0d287be45cee, 0xffefd63862ac2513, 0x21be4c53ee4e6486, 0x7e0c78bbb903f9d4, 0xd297c5f7f160e911, 0x1bd9829a2de8c738, 0xa7a3c71a0116cd59

plan 5:

81409a63a42c076dc4b892cbeacfa4ca980a9e9235a4437ba8a6602cd68b2ca6, 31, 0x338e03d42c97b4ab, 0x057e13f683cbea95, 0x891748ea4e3af00a, 0x6c1ce0d703bf57a5, 0xa29d3c79e7af4874, 0x953b04907659fd08, 0x4b9d3a4b59cdbb65, 0x6ed654d4c0a69bf0, 0x79622ea1149a8d40, 0xe8acf05d0c4d59c1, 0x18ff43ca6bad461b, 0xcd9f5b4e61e5d2ae, 0x2d459f03e08698c2, 0x72ff6c0b661fe089, 0xc8ef1c8436db65dc, 0x377b5ee16dc4a358, 0x658da4ce48c27766, 0xffefd63862ac2513, 0x25ac2a525f9a7b2c, 0x7e0c78bbb903f9d4, 0x13f73972c751cc56, 0xb0baba3fc2ecec66, 0x297a184ee6319e6b



NTNU

Kunnskap for en bedre verden