Joakim Sæther

2021

Master's thesis

Master's thesis

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

NTNU
Norwegian University of
Science and Technology

Joakim Sæther

Improving alphanumeric selectivity estimation in MySQL using histograms of closed frequent itemsets

June 2021

NTNU
Norwegian University of
Science and Technology

NTNU

Norwegian University of
Science and Technology

Improving alphanumeric selectivity estimation in MySQL using histograms of closed frequent itemsets

Joakim Sæther

Computer Science
Submission date: June 2021
Supervisor: Norvald H. Ryeng

Norwegian University of Science and Technology
Department of Computer Science

NTNU

Abstract

Institute for computer science and informatics

Master of Science

Improving alphanumeric selectivity estimation in MySQL using histograms of closed frequent itemsets

by Joakim SÆTHER

As the data volume stored in databases increases by the day, it is becoming critical to have accurate approaches for estimating temporary results sizes for arbitrary queries. This thesis focuses on the estimation of result sizes for predicates using the LIKE operator in MySQL.

By using a set of three generated query workloads, a modified version of the SPH approach for selectivity estimation introduced in Aytimur and Çakmak, 2018 is thoroughly tested and compared to existing approaches in MySQL and PostgreSQL. Further, some more complex queries from the Join Order Benchmark are run to inspect the new approach's impact on the MySQL system in a practical setting.

The results show that the modified SPH approach gives an estimation accuracy comparable to PostgreSQL and superior to MySQL for the generated workloads. For the 73 complex queries of the Join Order Benchmark, the selectivity estimates provided by the modified SPH approach gives a 26% improvement to execution time compared to the current solution in MySQL.

NTNU

Sammendrag

Institutt for datateknologi og informatikk

Sivilingeniør i datateknologi

Forbedring av alfanumerisk selektivitetsestimering i MySQL ved hjelp av histogrammer av lukkede frekvente elementer

av Joakim SÆTHER

Mengden data som lagres i databasesystemer verden over er i konstant vekst. På grunn av dette blir det viktigere og viktigere å ha nøyaktige metoder for å estimere størrelsen av midlertidige resultater i vilkårige spørringer. Denne avhandlingen fokuserer på selektivitetsestimater for LIKE-operatoren i MySQL.

En modifisert versjon av SPH-algoritmen for selektivitetsestimering (presentert i Aytimur and Çakmak, 2018 blir nøye testet ved bruk av tre genererte sett med spørringer. Resultatene blir deretter sammenlignet med de nåværende løsningene for selektivitetsestimering i MySQL og PostgreSQL. Til slutt blir 73 komplekse spørringer fra 'Join Order Benchmark' kjørt for å observere den nye løsningens effekt på MySQL i en praktisk situasjon.

Resultatene viser at den modifiserte SPH-algoritmen gir en nøyaktighet som er sammenlignbar med PostgreSQL og langt bedre enn den nåværende løsningen i MySQL for de genererte spørringene. For de 73 mer komplekse spørringene viser resultatene en reduksjon i kjøretid på 26% sammenlignet med den nåværende løsningen i MySQL.

Acknowledgements

Throughout my work on this thesis, I have received a great deal of guidance and assistance.

First, I would like to express my appreciation to my supervisor Dr. Norvald Ryeng. Being both my contact at the MySQL team and my supervisor at the university, he has been an invaluable part of this research. It has been a pleasure working with you!

I would also like to thank the MySQL team in Trondheim for including the master students in their social events and supplying us with the equipment needed for our research. In particular, I would like to thank Steinar Gunderson for his assistance regarding the IMDb dataset.

Finally, I would like to thank my fellow students August Indal and Rolv-Arild Braaten, for their constant moral support through this entire Master's degree and for sharing their Python expertise to make my results look pretty. These five years would not have been the same without you!

Contents

List of Figures

Chapter 1

Introduction

Nowadays, a database management system (DBMS) is at the core of almost any thinkable information system, and the performance of such a DBMS is crucial to the performance of the information system that uses it. The amount of data stored is constantly increasing (Holst, 2021), and the algorithms and methods used to search this data are thus becoming increasingly important. Due to this, the exploration of methods to increase database performance has become a popular topic of recent research.

Among DBMS systems, relational databases (RDBMS) are found as a sub-group. In the architecture of an RDBMS, the most critical component concerning performance is the query optimizer that focuses on finding a good plan for executing any query received from a client application. An essential part of this task is to find a join order that reduces temporary result sizes early in the execution. This problem is far from trivial, and most query optimizers bear an unmistakable mark of this.

Leis et al., 2015 and Leis et al., 2018 examines the different parts of the query optimizer in five different DBMSs to understand the impact that the various components have on the overall performance of the system. The conclusion shows that cardinality estimates are the limiting factor for most optimizers and that the sophistication of the cost model makes a minimal impact as long as the cardinality estimates are wrong. Because of this, they conclude that research should be focused on cardinality estimation rather than cost models.

Further, Sæther, 2020 runs the same experiments as Leis et al., 2015 for the MySQL system to compare the cardinality estimates of the MySQL system against other DBMSs. The results show that the MySQL system generally has a larger error in its cardinality estimates than other DBMSs and that LIKE-predicates involving wildcards are especially problematic. Since the LIKE operator plays a crucial role with respect to substring searches in databases, this shortcoming of the MySQL system is quite significant.

Because the LIKE operator is a filtering operator it is simpler to talk about its selectivity than its cardinality. The selectivity of a LIKE operator is the percentage of rows that passes the filter, and can be computed as $Selectivity = \frac{output\ rows}{input\ rows}$. The focus of this thesis will be on selectivity estimates of the LIKE operator in MySQL, and specifically on predicates containing wildcards. It will aim to answer the following research questions:

- What research has previously been done on the subject of result size estimation for the LIKE operator containing wildcards, and which of these approaches are sensible to implement into the MySQL system?

- How is the accuracy and performance of the method from question one compared to the current approach of the MySQL and PostgreSQL systems?

- How will an improved selectivity estimate for the LIKE operator influence the optimizer's decisions for complex queries?

A literature review will be made of the relevant research to answer the first question, starting from the nineties before working towards the more recently proposed approaches. The MySQL architecture will then be discussed with this literature in mind to find a fitting approach for the MySQL system. This discussion will show that some methods are more practically fit than others to be implemented into the MySQL system.

The second research question will then be answered using the gained knowledge from the first question to find a suitable method and implement it into the MySQL system. The chosen method will be tested thoroughly using three query workloads. The three query workloads will each consist of one particular type of predicates in order to test the chosen method for a wide spectre of LIKE predicates. The results will be put into context by running the same experiment for the MySQL and PostgreSQL systems.

Finally, the last research question will be answered by running some more complex queries from the Join Order Benchmark that was introduced in Leis et al., 2015. The queries were produced to closely mimic a real-world query workload and uses a snapshot of the IMDb dataset. The results will be compared to the previous solution in MySQL to see the improvement both to pure selectivity estimates and to the system's overall performance. By examining the selectivity estimates supplied to the optimizer and the resulting execution plan, it will be clear whether small changes to selectivity estimates can impact the decisions made by the optimizer.

The remainder of this thesis will be structured as follows: Chapter 2 will give a relatively quick introduction to the process of selectivity estimation in DBMSs before moving on to the MySQL system in particular. Then, Chapter 3 will give an overview of previous work that will be relevant and ultimately answer the first research question. Moving on, Chapter 4 will discuss the implementation of a suitable selectivity estimation approach into the MySQL system. Chapter 5 will then give an overview of the experimental setup that will be used to conduct the experiments in Chapter 6. Finally, Chapter 7 will give answers to all the three research questions, as well as some suggestions for future work.

Chapter 2

Problem description

This chapter will introduce some relevant concepts before eventually describing the problem discussed through this thesis in more detail. The discussion will start from the top of a general DBMS before moving into the relevant components. At the end of the section, we will take a step back and look at the current situation of MySQL with the discussed components in mind.

2.1 The architecture of a DBMS

The complete structure of a DBMS is quite complex and is documented through entire books in the database literature, such as Hellerstein, Stonebraker, and Hamilton, 2007, where most of this discussion will originate. The focus of this discussion will be on relational database management systems (RDBMS). Even though these systems are only a tiny fraction of the database literature, this discussion can not go into detail on all its components.

Among all the more or less complex components of a relational database system, the most relevant one for this thesis will be the relational query processor. The relational query processor works as a middleman between the client and the storage system on which the DBMS is built. There are individual differences between the different RDBMSs for the processing steps that happen within the query processor, but Figure 2.1 shows an example workflow.

When a query arrives at the query processor, it is immediately parsed and authorized. Among other things, this involves checking that the tables and fields mentioned in the query exist in the database catalog and that the user is authorized to access these tables and fields. If the parsing completes successfully, the query is passed along to the query rewriter in an internal format.

The Query Rewriter is responsible for simplifying and normalizing the query without changing its semantics. Since the query processor is quite complex, it does not work on arbitrary queries but rather on a normalized version. Normalizing a query into a normal form is thus an important task. In addition to normalization, the rewriter is also responsible for simplifying the query. The simplification process can be compared to simplifying a mathematical expression involving unknown variables. By canceling out and expanding the terms, the expression can be made simpler to comprehend. An example of such a rephrasing is the rewriting of contradicting predicates (i.e x > 9 AND x < 8) as FALSE. When the query rewriter is done with a query, the query, still in an internal format, is handed to the query optimizer.

The Query Optimizer is one of the most complex components of a database system, and it will be at the core of the discussion throughout this thesis. Due to this, it will only be discussed in short here before going more into details in section 2.2. The optimizer's job is to transform the internal query representation received from

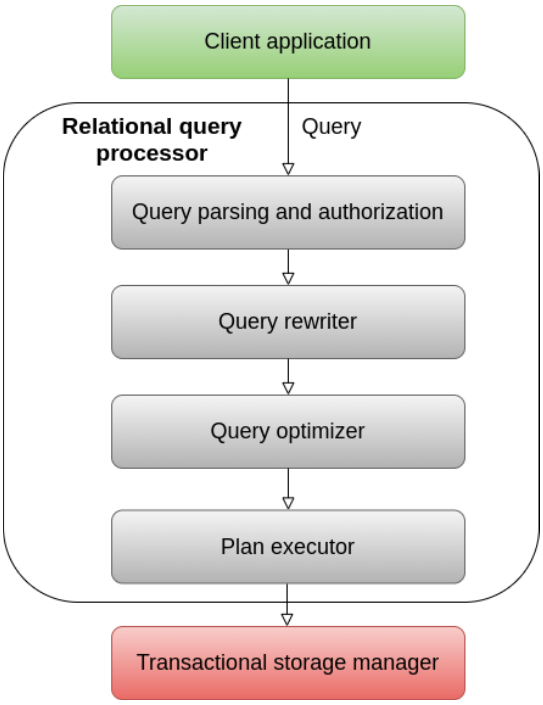

Client application
Relational query processor
Query
Query parsing and authorization
Query rewriter
Query optimizer
Plan executor
Transactional storage manager

FIGURE 2.1: An example workflow within a relational query processor. A query arrives from the client application, before moving through the different components, and eventually executes against the underlying storage system.

the query rewriter into an efficient query plan for executing the query. The most crucial part of this phase is reordering the query's operations such that the resulting query plan represents an efficient order of operations. This optimized query plan is usually represented as a tree graph with operators in the inner nodes and tables in the leaf nodes. Each operator connects to one or two tables depending on the nature of the operator. The plan is then sent forward to the query executor.

Finally, the fully specified plan arrives at the query executor, which executes the query against the underlying storage medium. Most modern query executors employ the iterator model, in which an iterator object is given to the receiver of the data, which can be the client (at the top of the query plan graph) or the succeeding operator. The execution is then initialized when the root node calls 'next' on its iterator object. This call is then propagated down the tree before eventually reaching the bottom node. The bottom node then starts executing until it has produced a single row. If this row is then filtered out somewhere on its journey to the top of the tree, a new 'next'-call will automatically be made by the node that filtered out the result such that the root node eventually receives a row or an end-of-execution signal. This workflow is then continued until the client has received all requested results.

The most prominent advantage of the iterator model is its simplicity. The use of a pull model rather than a push model avoids any need for rate-matching between the nodes since a new row will only be pulled when the node is ready to process it. The iterator model also has the advantage of allowing a single DBMS thread to execute an entire query graph. For systems that run on a single thread (which is quite common), this approach will not sacrifice any performance while still keeping execution clean and straightforward. However, for parallel execution, a push model could have performance-enhancing effects.

2.2 The query optimizer

As previously mentioned, the query optimizer is a critical component for the performance of a database system. Its job is to transform the internal query representation received from the query rewriter into an efficient query plan for executing the query by rearranging the operations. These operations could, among other things, be a join between two tables, filtering a table based on a given predicate, or simply sorting the contents of a column. The optimizer attempts to find the fastest possible execution plan based on the available information about the involved database columns. However, the most critical trait of an optimizer is to find safe plans such that any query completes in a reasonable amount of time.

There are two main types of query optimizers, and the difference between the two types is found in the way that execution plans are compared:

- In a **rule-based optimizer** there is no consideration to the cost of each operation. The optimizer is based on a set of rules that it goes through systematically to reorder the query's operations. An example of such a rule could be that an index should be used if it exists. This kind of optimizer is simpler to implement, but no pure rule-based optimizers are used in real systems today since quite a few crucial factors for query performance are ignored. Another downside to the rule-based optimizer is extending it to changes in access paths or changes in data distribution. A rule-based optimizer could (in theory) be created to choose the perfect plan for any query, but as soon as something about the data load changes, new rules must be added. This does not scale well in the long run.

- **Cost-based optimizers** on the other hand, involves a cost function that takes various relevant factors into account. All commercial systems today involve a cost-based optimizer to a more or less prominent degree. The most typical approach is to have a cost-based approach whenever statistics are available in the DBMS while using a rule-based approach when no such statistics are available. The nature of the cost-function is quite different from system to system, ranging from the most straightforward functions that only consider table sizes and the number of joins to the more sophisticated functions that consider many more factors such as the properties of the host (for example, available RAM and processor speed).

The set of possible query plans for a query is called the search space of the query. The size of the search space increases for each table involved in the query and for each available access path (for example, indices or table scans). For fairly complex queries involving multiple join conditions or indices on multiple keys, the search space is too big to be fully explored, meaning that the optimizer would not necessarily find the best available plan even if the cost model had been perfect. For a cost-based optimizer to consistently return good plans in such cases, it must have a decent strategy to explore the correct parts of the search space. The two main approaches to do this are dynamic programming (bottom-up search) and top-down search. Both of these approaches have shown promising results in practical settings and are both used in real-world systems.

Dynamic programming works by breaking the problem into smaller parts and taking advantage of the fact that the smaller parts will be part of the solution to the entire problem. By storing the cost of smaller parts of the solution, these parts can be used as building blocks when the final solution is built.

Top-down search works oppositely by first considering a complete solution. From this initial solution, small transformations of the query graph are made while monitoring the cost of new plans. The advantage of the top-down approach is that the optimization can be stopped at any time since the optimizer always has a complete plan. Having a plan available at any time allows the DBMS to set a bound on the time that is available for optimization.

The main goal of a query optimizer is to reduce the intermediate sizes of a query such that large and expensive join operations are avoided towards the end of the execution. If a large portion of the contents of a table can be filtered out at an early stage, it is often very profitable to do so compared to doing a join on the table and then filtering the (possibly huge) result of the join. For the optimizer to keep the initial results small, it needs accurate estimates for the result size of any predicate or join operation in the query. Estimating these result sizes is called selectivity estimation.

2.3 Selectivity estimation

Selectivity estimation is the process of estimating the filtering effect (indirectly the join size) of predicates on single columns or join operations between two columns. Since the set of columns in a database could involve a wide range of different data types (Integers, strings, timestamps), and a query could contain a wide range of different operators on these columns (>, <=, LIKE, BETWEEN), the optimizer needs to take into account both the operator and the data type in addition to the predicate value in order to give a reasonable estimate of the result size. Due to the differences between data types and operators, the selectivity estimation process is usually quite different based on the present data type and operator.

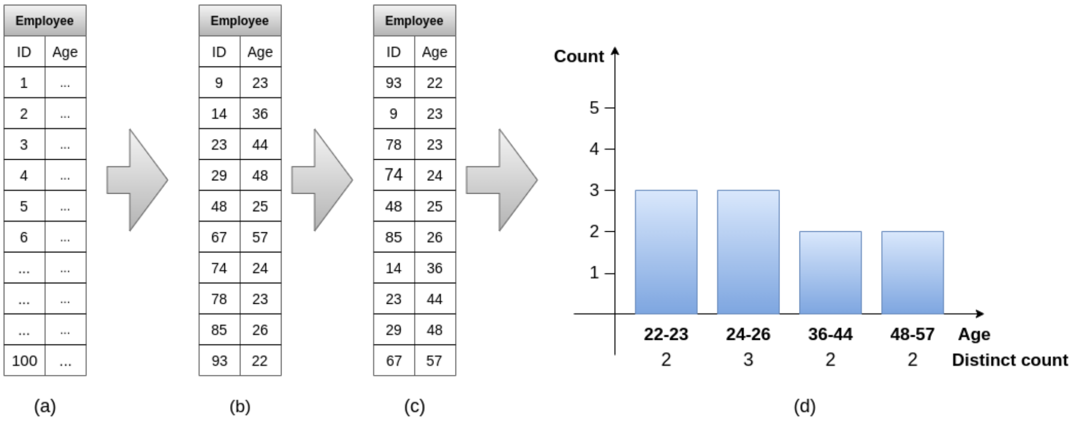
Before considering the LIKE operator that will be the main focus area throughout this thesis, the idea of selectivity estimation concerning numeric values should be visited. Estimation of numeric columns is the most widespread form of selectivity estimation in the existing literature and existing DBMSs and is thus essential in order to understand the particular problems encountered when considering the LIKE operator. When it comes to numeric selectivity estimation, the discussion provided in this thesis will barely scratch the surface of existing approaches. Sæther, 2020 does, however, provide a more thorough discussion of the topic.

2.3.1 Numeric selectivity estimation

There exist numerous papers written about selectivity estimation for numeric data types. This discussion will only involve the histogram and sampling approaches since these two classes of approaches cover a large portion of the literature in the field, and many more approaches are closely related to those two. No matter which approach is being used, the ultimate goal is to estimate the result size of a predicate using a small representation of the data in a column.

Histogram

A histogram consists of a set of buckets where each bucket contains a specific range of values in a column. A good estimation of the underlying value distribution can then be made using this compressed version of the data. Although this sounds simple enough, there exist loads of ways to distribute the data into the various buckets.



(a)

Employee	
ID	Age
1	...
2	...
3	...
4	...
5	...
6	...
...	...
...	...
...	...
100	...

(b)

Employee	
ID	Age
9	23
14	36
23	44
29	48
48	25
67	57
74	24
78	23
85	26
93	22

(c)

Employee	
ID	Age
93	22
9	23
78	23
74	24
48	25
85	26
14	36
23	44
29	48
67	57

FIGURE 2.2: The process of constructing a equi-height histogram for the column "Age" of the employee table. A sample of 10 rows is taken from the table in (a). The sample in (b) is sorted on the age column, before the sorted version in (c) is used to distribute the values across 4 buckets in (d).

The first (and simplest) approaches were proposed in Kooi, 1981 and Piatetsky-Shapiro and Connell, 1984, using buckets of equal width and equal height, respectively. The latter of the two quickly showed itself to be the most promising. The process of constructing an equi-height histogram is illustrated in Figure 2.2. Initially, the table is sorted by the key on which the histogram will be built. The sorted version of the table is then looped through to produce buckets of similar height. A single value will never span across more than a single bucket, meaning that the buckets can have quite different sizes. In addition to the total number of values within each bucket, the number of distinct values in the bucket is also kept. Keeping the number of distinct values enhances the accuracy of the estimates produced by the histogram.

The equi-height histogram has been extended in multiple ways since its introduction. Muralikrishna and DeWitt, 1988 extends the histogram into multiple dimensions, and Gunopulos et al., 2000 builds further upon this by introducing multi-dimensional overlapping buckets. The introduction of multi-dimensional statistics also introduced a problem related to sample size. Due to the curse of dimensionality, the space in which buckets and tuples are located grows exponentially with the number of dimensions, and thus the sample size needed for a statistically sound result also grows exponentially. Due to this, a multi-dimensional histogram is expensive to build and maintain, making it less used in modern applications than the single-dimensional histogram.

When a histogram has been built, producing selectivity estimates from it is usually quite simple. Figure 2.3 considers the histogram that was constructed in Figure 2.2, and the predicate "Age > 25". It is known that the two last buckets contain only tuples where the age is greater than 25. Thus these four values can be added to the result. Further, half of the second bucket satisfies the predicate and is added to the result as well. Adding up these values results in a selectivity estimate of 0.65, which could be more or less correct depending on how well the sampled tuples represent the data distribution of the underlying table.

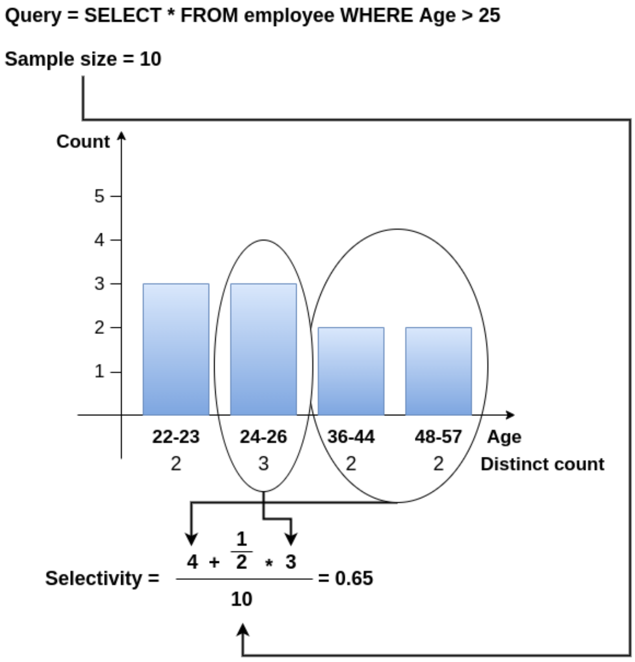


FIGURE 2.3: Producing selectivity estimates from the histogram generated in Figure 2.2. For the predicate "Age > 25", the two last buckets is added to the estimate. Additionally, half of the second bucket is added because 25 is in the middle of this bucket.

Sampling

Besides the histogram approach, using sampling is the most common selectivity estimation approach seen in real-world systems today. Compared to a histogram, the advantage of a sampling approach is its ability to detect correlations between columns. No matter how many columns a sample spans, it is always possible to detect correlations between any number of the spanned columns. Just like the histograms, however, samples suffer from the curse of dimensionality, making it infeasible to make accurate samples across many columns.

The two main classes of sampling are online and offline approaches. In an online approach, samples are made ad-hoc for each query. Ad-hoc sampling allows the sample to adapt to the columns that are involved in each query. The problem with an online approach, however, is that repeatedly taking an accurate sample is expensive compared to only taking a single sample. On the other hand, in an offline approach, the sample is taken only once, and the same sample is used for any query. The main problem with this is that any query using this sample needs to include exactly the same columns as the sample. Whenever the query workload is known, the samples can be generated to match the queries, but for selectivity estimation of arbitrary queries, this could involve storing hundreds of separate samples.

In addition to the two main approaches to sampling, it is also worth mentioning that there are many ways to take a sample. When sampling is used to estimate join sizes, tuples can be drawn from the two tables involved in the join in multiple ways. The Bernoulli sampling draws tuples randomly from both tables, while the correlated sampling used in Wang and Chan, 2020 draws tuples independently from one table before sampling independently from the semi-join between the sample and the second table involved. Even though Bernoulli sampling and correlated sampling are the most regularly used in literature, they are often combined into more exotic sampling schemes such as in Estan and Naughton, 2006, or Chen and Yi, 2017.

To conclude the discussion on numeric selectivity estimation, it should be mentioned that most approaches proposed in the research are not widely used in real DBMSs today. The discussion of Sæther, 2020 reveals a large gap between state-of-the-art approaches in the literature and approaches seen in real-world applications. DBMSs today tend to keep only simple statistics such as a single-dimensional equi-height histogram or a basic sample. There are many reasons for this gap, but the most prominent reasons are a lack of focus on the area from the vendors' side and the risk of deploying new solutions to a large user base without (very) thorough testing.

2.3.2 Selectivity estimation for textual data

Due to the increasing amount of textual data seen in databases today, it becomes more critical to estimate selectivities for string data efficiently. This turns out to be a non-trivial task due to the almost infinite number of strings that could be stored. Considering the histogram approach discussed in the previous section, one can imagine a bucket with a lower bound of "ABC" and an upper bound of "ABE". Even though the two bounds are relatively close together, the number of strings between them can still be enormous, including all strings starting with "ABC" and "ABD", such as "ABCDEFGH" and "ABDCABGD". For integer data, on the other hand, the available integers between, for example, 6 and 16 are very restricted. This difference is the reason that pure histogram approaches are unsuited for selectivity estimation on string columns.

Another difference between string and integer data is the number of duplicates. In a table with full names of employees, there will usually be a low number of duplicates and a large number of distinct values. Taking a sample of these names to use for either histogram construction or pure sampling would have no chance of giving a good representation of the data. Most vendors offer an enum type for string values with a low number of distinct values, and thus almost any string column will have a large number of distinct values. By comparing an imagined "Name" column to the integer type column "Age" in Figure 2.2, it can be seen that the "Age" column would have a significantly larger proportion of duplicate values and a much lower number of distinct values. In turn, this has the effect of simplifying selectivity estimation drastically for integer values.

Finally, there exists another operator for string data, namely the LIKE operator. The primary purpose of the LIKE operator is to aid in substring searches. The string data type is the only datatype for which it is interesting to search for only a part of the value. It is pretty rare to have a reason to search for all ages that end with a "1". However, it is not rare for textual data to make a query similar to "Find all actors with the first name 'Robert'". If one were to forget the actor's last name, for example, it would be a lot simpler to find the actor after filtering on the first name.

Krishnan, Vitter, and Iyer, 1996 considers the LIKE-predicate with wildcards to be the rule rather than the exception for string data, and thus it is vital to have a good way of estimating selectivities for such predicates. The topic of LIKE selectivity estimation will be examined more closely in Chapter 3.

2.4 MySQL

The optimizer of MySQL has been known to be relatively primitive compared to many other systems. The only forms of joins available to the system are the standard nested loop join and the hash join. This simplicity has the downside of giving the optimizer very few options from which to choose. On the other hand, however, this allows the optimizer to be relatively plain. A little bit simplified, one can say that picking the join order and the access paths are the only responsibilities of the optimizer. However, deciding on a good join order requires considering result sizes and available access paths such as table scans and indices. The most important part of this is selectivity estimation for the estimation of temporary result sizes.

For the task of selectivity estimation, the optimizer relies on a prioritized order as follows:

1. **Index dive.** If an index is available and the predicate can be rewritten as a combination of ">" and "<" operators, the optimizer makes an index dive. In an index dive, the optimizer dives into the index to retrieve the exact selectivity of the predicate. This index dive would also work for LIKE predicates with a constant prefix such as "name LIKE 'Robert%'".

2. **Index statistics.** This includes information about the maximum and minimum values of the index, the number of unique values, index type, and the index's handling of NULL values.

3. **Histograms.** As of MySQL 8.0, a simple equi-width histogram can be built to make more accurate selectivity estimates. An index is, however, always preferred to a histogram. Due to this, histograms should only be generated for non-index columns to have any effect.

4. So-called **"guesstimates"**. These are constant values inserted as an estimate for each operator when no other statistics are available. For example, the "=" operator is given a selectivity of 0.1 no matter how the underlying data is distributed. The approach of having these constant guesses was first introduced with System R in Astrahan et al., 1976.

One can see that this prioritization strongly favors the use of indices, which does not include any information about the data distribution. The main reason for this prioritization is that the histogram statistics must be manually updated and built from scratch each time. The need for manual updates of the histogram makes indices more precise and faster for dynamic data loads. Since a dynamic data load is the rule rather than the exception, a decision has been made to favor dynamic rather than static data.

By running the join order benchmark in MySQL, Sæther, 2020 finds that the MySQL selectivity estimations are struggling in many cases compared to other systems. The most prominent issue is related to the LIKE operator. For LIKE predicates with constant prefixes, an index dive is used to find an accurate selectivity estimate. However, if the predicate has a leading wildcard, the results are not necessarily located together in the index and thus an index dive cannot be used. Further, the information stored in the index statistics does not contain any information about substring frequencies and is thus not used. The next step in the list is the histograms. The current histogram in MySQL is not a good approach for string selectivity estimation, and using them produces vast underestimation in the order of 10^{-300} due to the enormous space within each bucket. Due to all of this, the only reasonable option for selectivity estimation for LIKE predicates with leading wildcards in MySQL is the guesstimate of $\frac{1}{9}$.

It can be seen that this estimate can be wrong by orders of magnitude. Single-letter predicates can have a selectivity close to 1 in some use-cases, while longer predicates tend to have a selectivity far closer to zero, giving vast miss-estimates. The most straightforward approach to improve upon these estimates is to store a statistic that targets the LIKE operator directly by keeping information about the selectivity of certain sub-strings and using this statistic during selectivity estimation. Chapter 3 will make a dive into the literature that is available on the topic of LIKE selectivity estimation and will give some clues as to which methods could be relevant in order to improve the situation in the MySQL system.

Chapter 3

Previous work

Selectivity estimation for LIKE predicates has gained most of its attention due to the increasing amount of alphanumeric data in databases. It has, nonetheless, been a field of research for quite some time, with the earliest approaches being proposed already in the 90s. It is, however, through the last decade that most of the research in the area has been conducted. This section will give a short overview of research on the related topic of fuzzy string queries before transitioning over to the problem of LIKE selectivity.

3.1 Approximate string matching

Approximate string matching concerns finding strings that match or partially match a query string predicate (often referred to as a fuzzy string predicate). The most regular formulation of the problem uses the edit distance to measure the equality of two strings. A common version of edit distance is the Levenshtein distance in which insert, delete and replace are the available operations, each adding one to the distance between two strings. For example, the string 'Haelro, world' needs three operations to form 'Hello, world!'. Thus it has a Levenshtein distance of three. These operations would be removing the a, replacing the r with an l, and inserting an exclamation mark at the end. The Levenshtein distance between two strings S_1 and S_2 will be referred to as ld(S_1, S_2).

Estimating the selectivity of a fuzzy string predicate such as 'edit distance < 3 from "hello"' is a known problem in the literature, and it does represent a more general approach to the LIKE operator. Jin and Li, 2005 introduces the technique SEPIA to solve this problem. SEPIA groups strings into clusters before building a histogram for each cluster, and finally, a global histogram for the entire database. Rather than using pure edit distance in the histograms, an edit vector is used. The edit vector keeps track of the number of inserts, deletes, and substitutions that need to be done individually. This division of operations allows a more fine-grained distinction between strings of equal edit distance. Each cluster contains a pivot string that is representative of the strings within the cluster. Rather than comparing the query string with each of the strings in the local histogram, it is only compared to the pivot string. A probability function is then used to estimate the number of matching strings within the cluster.

A similar approach is found in Mazeika et al., 2007, where selectivity for fuzzy string predicates is estimated using inverse strings of overlapping q-grams. In simpler terms, this means that each string in the database is decomposed into overlapping strings of length q (q-grams), which are then stored as an inverse index. A signature of this index is then computed and stored, giving a lower space complexity since this signature is relatively compact. Finally, the signatures are clustered together using k-means clustering. Using these clustered signatures to give selectivity estimates yields reasonably accurate estimates whenever the length of the query string is close to the average length of strings in the database. The results show that the resulting estimator, VSol, is efficient for large skewed databases of short strings but has weaknesses for arbitrary data loads.

Finally, Lee, Ng, and Shim, 2009 introduced a state-of-the-art approach using an N-gram table extended with wildcard characters ('?'), and ideas from set hashing signatures similar to Mazeika et al., 2007. The N-grams are collected in a table where they are grouped depending on which operations were needed to create these N-grams (for example, one deletion and one substitution). Only the minimal base substrings are kept in the table, while the remaining N-grams are pruned away. Based on this table, they propose one clustering approach and one sampling approach. The idea behind the extended N-gram table is the assumption that a string S usually has a substring S' with a selectivity estimate similar to S. This assumption was examined in Chaudhuri, Ganti, and Gravano, 2004, showing that 70% to 90% of predicates have substrings of length less than or equal to 5 that accurately determines the frequency of the predicate in its entirety. The experiments did, however, show a deviation across data loads.

Even though previous work regarding approximate string matching is not directly applicable to the SQL LIKE operator, the issues are closely related. Jin and Li, 2005 provides a discussion regarding an extension of their technique to support other similarity measures such as the Jaccard similarity. This extension does, however, require a continuous numerical measure of the similarity between two strings, and the LIKE operator can therefore not be applied.

Lee, Ng, and Shim, 2009 does also propose an extension to support other similarity measures for their approaches. The experimental results reveal that the clustering-based technique performs best for the Jaccard similarity. Additionally, the algorithm is extended to support the LIKE operator by estimating the size of unions between the substrings of the query string. The experimental results show that the clustering approach outperforms the random sampling approach for the LIKE operator.

3.2 Estimation of LIKE selectivity

Even though some of the techniques used for approximate string matching can be extended to include the LIKE operator, there exist better approaches that specialize in the issue of LIKE selectivity estimation.

3.2.1 Suffix tree

The first approach to the estimation of LIKE selectivity was proposed in Krishnan, Vitter, and Iyer, 1996, using a pruned count-suffix tree to keep track of frequently occurring strings in the database. This was later extended in Jagadish, Ng, and Srivastava, 1999. Both of these approaches build a suffix-tree that initially keeps track of all the strings in the database and their suffixes. The tree is then pruned as soon as a space threshold is reached. At the end of the construction, the tree consists of the n nodes with the highest frequencies. A threshold is then set to the frequency of the most frequent node that was pruned away, and stored for later use.

The difference between the two approaches lies in the assumptions the algorithms make about the relationship between substrings. Krishnan, Vitter, and Iyer, 1996 assumes that substrings occur independently of each other, while Jagadish, Ng, and Srivastava, 1999 assumes that the data exhibits the statistical property called "short memory". Short memory means that the probability of a substring occurring depends on the preceding substrings. To model this dependency, Jagadish, Ng, and Srivastava, 1999 uses Markov chains.

3.2.2 Short identifying substrings

Chaudhuri, Ganti, and Gravano, 2004 introduce the concept of short identifying substrings, which essentially means that the selectivity of a long string predicate is determined by a shorter substring called a "short identifying substring". The example used in the paper considers the predicate R.A LIKE '%seattle%'. In most cases, the selectivity of this predicate is roughly the same as the selectivity of R.A LIKE '%eatt%'. This happens simply because very few other words contain the substring 'eatt'. The argument for utilizing this pattern rather than the independence assumption or the Markov assumption is the severe under-estimations made by these assumptions. Estimators built on the independence assumption or the Markov assumption will over-compensate for the additional letters in 'Seattle' and give a far lower estimate for 'Seattle' than they would for 'eatt'.

The most crucial part of the estimation approach in Chaudhuri, Ganti, and Gravano, 2004 is finding the correct identifying substring. Initially, one candidate identifying substring is chosen for each length between a threshold q and the query string length. The decision on which substring to choose is based on the fact that the query string cannot have higher selectivity than any of its substrings. Therefore, the lowest available selectivity is chosen at each level. Finally, the selectivities of all the candidate substrings are combined using any selectivity estimation technique. The one chosen in the paper uses a learned decision tree to assign weights to the various lengths of candidate substrings. The decision tree is trained using a representative query workload that can typically be obtained from a profiling tool.

The only apparent disadvantage to the approach proposed in Chaudhuri, Ganti, and Gravano, 2004 is the requirement for a q-gram frequency table from which upper bound selectivities are derived. Such an exhaustive table could be expensive to compute and to store. The paper does, however, suggest some future work with regards to storing only important q-grams. In such a solution, the estimator would return a default constant value for q-grams that are not present in the table.

3.2.3 Sequential Pattern-based Histogram (SPH)

The most recent approach to LIKE selectivity estimation was introduced in Aytimur and Çakmak, 2018 and uses mining of frequent itemsets combined with a modified histogram to estimate selectivity. The approach first mines closed frequent itemsets in an offline fashion using the BIDE algorithm. Further, the resulting itemsets are divided into buckets in the same manner as other histogram approaches. The steps that are taken in the SPH algorithm are summarized in Figure 3.1.

An attractive property of the SPH approach compared to other histogram approaches is how the buckets are composed. Each bucket stores only a single frequent itemset as its endpoint value, in addition to the frequency of this itemset. This approach saves some space by only storing a single value (rather than an upper and a lower bound) and is quite similar to the buckets of a singleton histogram. The difference between the SPH algorithm and a singleton histogram is that the singleton histogram stores all values in the column, while the SPH histogram only store a spread-out subset of the frequent itemsets.

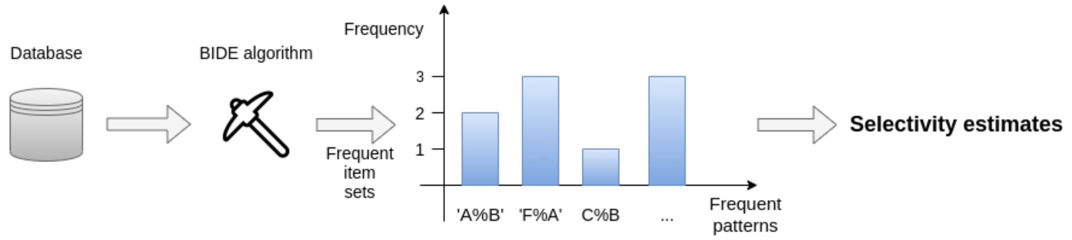


FIGURE 3.1: A rough overview of the SPH algorithm. The values of a database column is mined for closed frequent itemsets using the BIDE algorithm, before being divided into buckets of a histogram that is further used to give selectivity estimates.

When the resulting histogram is used to estimate selectivity, the approach distinguishes between an exact match and an encapsulated match. If a query string is matched precisely by a histogram bucket endpoint, the endpoint frequency is returned as an estimate. On the other hand, an encapsulated match happens if all letters of the query string show up in the bucket endpoint in the same order (the query 'AB' would have an encapsulated match for both 'ACEB' and 'CBACB'), but not adjacently. If no exact match is found, the average of all encapsulated match endpoint frequencies is returned. However, if no encapsulated matches are found, a constant value is returned.

The initial approach to histogram construction presented in Aytimur and Çakmak, 2018 did not consider the position of letters but just the order in which they appeared in a string. Ignoring the position of the letters, the string 'ABDE' and the string 'ADE' both contribute towards the total frequency of 'AD' even though the two letters are not adjacent in the first string. This issue was later addressed in Aytimur and Cakmak, 2020 which is an extension to Aytimur and Çakmak, 2018. In this version of the SPH algorithm, the position of the characters is considered as well. The consequence of this extension is that two letters that frequently appear adjacently are considered separately from the same two letters appearing separated ('ABC' and 'ACB' will give two occurrences of 'A%B' and one occurrence of 'AB'). In addition to this extension, Aytimur and Cakmak, 2020 introduces elimination of redundant patterns before constructing the histogram. This elimination is done by computing the information content of each frequent itemset and removing sets that give little additional information to the histogram.

Although the results presented in Aytimur and Çakmak, 2018 and Aytimur and Cakmak, 2020 reports state-of-the-art accuracy for selectivity estimation of the LIKE operator, there is one issue that makes the approach impractical in a real-world system. Aytimur and Çakmak, 2018 reports a construction time in the range of 10-20 hours with a space overhead of 2-5 GB. Further, Aytimur and Cakmak, 2020 reports a construction time of 20-60 hours with a space overhead of 5-10 GB. The argument for the immense time and space complexity is the fact that the histogram construction is done in an offline fashion. For use-cases with a static data load, this could indeed be very attractive. For the general use case (which is the main target of large DBMSs such as MySQL), however, a dynamic data load combined with the requirement to build histogram statistics for many columns would make the SPH approach infeasible.

The determining factors for histogram construction time for the SPH algorithm are the sample size and the minimum support threshold for an itemset to be frequent. The two papers have documented the effect of varying minimum support thresholds but not the sample sizes. By reducing the sample size from the 800 000 rows used in the experiments, the histogram construction time could be reduced along with the space requirement. This adjustment would, however, sacrifice some of the accuracy supplied by the larger sample.

3.3 PostgreSQL

So far, this chapter has only pointed out the theoretical work that has been done in the area of LIKE selectivity estimation. As pointed out by Sæther, 2020, however, practical solutions are often quite different from theoretical work when it comes to selectivity estimation. Due to this, this discussion would not be complete without visiting the solution that a real-world system has employed.

Apart from the MySQL system (which, as we have already seen, lacks a good solution to LIKE selectivity estimation), the most widely used open-source system available today is PostgreSQL (or simply 'Postgres'). Since the software is open source, the implementation details can be inspected to understand how Postgres has solved selectivity estimation for LIKE predicates.

There are two main components that are taken into account whenever LIKE selectivity is estimated in Postgres. These are histograms and the Most Common Values list (MCV list). Initially, this section will describe how these statistics objects are built in Postgres before moving on to how they are combined and used in practice to get a sensible selectivity estimate for almost any predicate.

3.3.1 Building column statistics

Whenever a table is created in Postgres, a procedure is run to build statistics for the various columns in the table. An overview of this procedure can be seen in Figure 3.2a. The statistics are built on a sample of at least 300 rows, and the MCV-list is always the first statistic to be produced. The Postgres documentation describes the process of building the MCV-list as follows:

1. Sort the data
2. Count distinct groups, decide how many to keep
3. Build the MCV list using the threshold determined in (2)
4. Remove rows represented by the MCV from the sample

The threshold that is found in step two of the above procedure is determined using statistical analysis. This threshold represents how often a record must show up in the sample to be kept in the MCV list. The bound is always at least 10, but never more than 25, and it is mathematically equivalent to demanding the standard deviation to be less than 20% of its mean. Having this bound ensures that the predictions made by the MCV list are reasonably accurate. In the last step of the construction process, all rows picked up by the MCV list are removed from the sample such that they will not be included in any other statistics built on the column.

In the next phase of the statistics building process, the histogram is built on the remaining values in the sample. The histogram is only built if at least two values are remaining after building the MCV-list. Ensuring that at least two values remain ensures that the histogram does not collapse into a single bucket. When building the histogram, the first and last values remaining after building the MCV-list are kept, as well as evenly spaced values in between. All these selected values are then used as histogram bucket endpoints in the resulting histogram.

The histogram constriction does not, in contrast to the construction of the MCV-list, take any measures to ensure a statistically sound result. Instead, this is handled whenever the histogram is used to get a selectivity estimate.

3.3.2 Using column statistics for LIKE selectivity estimation

Both the histogram and the MCV-list that was previously discussed are used actively in the Postgres selectivity estimation process. This section will only describe how this happens for LIKE selectivity, but the estimation process would happen similarly (with some adjustments) for all operators and data types.

The main point of the estimation process is to combine the MCV-list and the histogram. Figure 3.2b shows a simplified version of the Postgres statistics usage. The MCV list and the histogram do not have any common values, and their estimates can thus be added up. Since the MCV list only keeps relatively accurate values, those estimates are added to the selectivity estimate without further weighting. For the example in Figure 3.2, this result is 0.33 ($\frac{3}{9}$).

For the less reliable histogram statistic, however, this process is a bit more complicated. The weight given to the histogram estimate depends on how much the estimate can be trusted. A histogram with more buckets is more likely to give accurate estimates and can thus be trusted to a more considerable degree. In the Postgres system, a histogram with at least 100 buckets is treated as secure (Figure 3.2 uses ten buckets for the sake of brevity) and is used similarly to the MCV-list. When a histogram has less than 100 buckets, the estimate from the histogram is only partly trusted. A histogram with 50 buckets will be weighted 50%, while a histogram with 20 buckets will be weighted 20%. The remainder of the weighting comes from a heuristic approach computed by multiplying a 'prefix selectivity' with a 'rest selectivity'. The prefix selectivity is computed similarly to a MySQL index dive and concerns the part of the predicate that precedes the first wildcard. Since this thesis's focus is on predicates with a leading wildcard, the most interesting part is the 'rest selectivity' that concerns everything after the first wildcard.

The heuristic rest selectivity is computed as described in Algorithm 1. All characters that trail the first wildcard are looped over (ignoring the first wildcard since that is included in the prefix selectivity), letting encountered wildcards increase the estimated selectivity, and constant characters decrease the estimated selectivity. Although this approach is somewhat reminiscent of the magic values sometimes used in MySQL, they are only used when the histogram gives a limited amount of information and is unsafe to use.

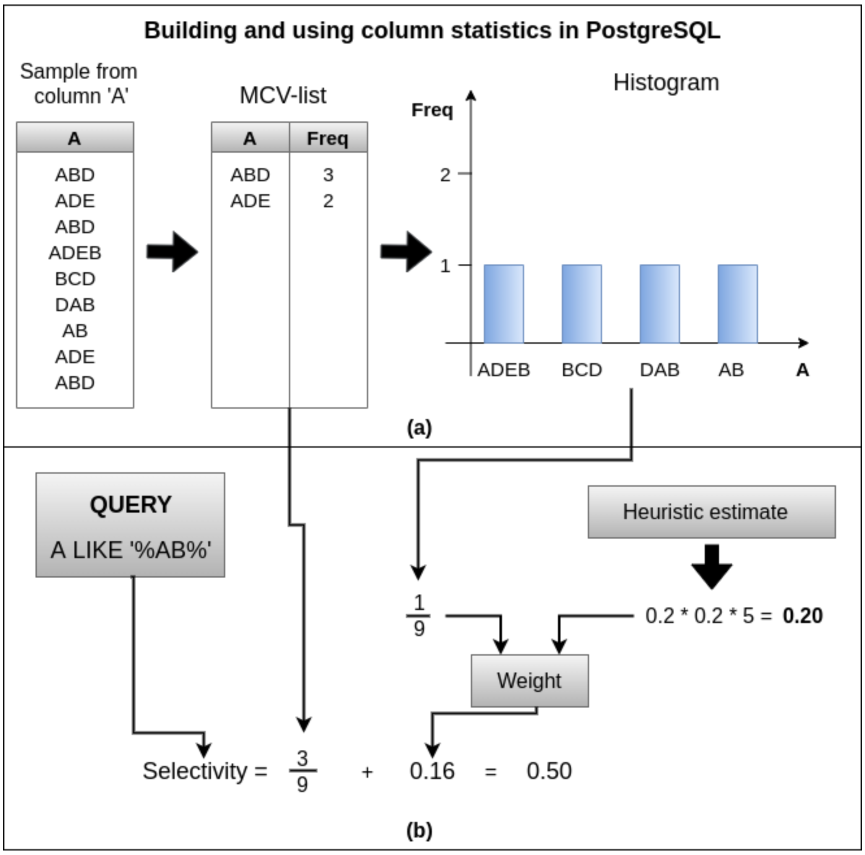


FIGURE 3.2: Building and using column statistics in PostgreSQL. In figure (a), a MCV list and a histogram is generated from a sample. Figure (b) uses these statistics to estimate the selectivity of a LIKE predicate. The histogram in the figure is weighted 10% for each bucket, while Postgres uses only 1% per bucket.

Algorithm 1: Rest selectivity in Postgres

```
Result: Estimated selectivity for the input predicate starting at the first
        wildcard.
$restSel = 1$;
$restPredicate = charactersAfterFirstWildcard()$;
for $char$ in $restPredicate$ do
    if $char == '\%'$ then
        $restSel* = 5.0$;
    else if $char == '\_'$ then
        $restSel* = 2.0$;
    else
        $restSel* = 0.20$;
    end
end
return $restSel$;
```

Chapter 4

Algorithm description and implementation

The previous chapter presented a few approaches that have been made for the related problem of selectivity estimation for approximate string queries. These approaches turned out to have a reduced accuracy compared to the specialized approaches for LIKE selectivity estimation. Since this thesis aims to improve estimates for the LIKE operator, these approaches are not relevant.

Further, the previous section presented some specialized approaches for the LIKE operator. The papers that presented these approaches have used different datasets, query workloads, and measures of accuracy. This makes them difficult to compare, and thus other factors must be considered to pick a method for the MySQL system. Among the three approaches discussed for LIKE selectivity estimation, the SPH approach is the most similar to what exists in MySQL today. Since it builds on a histogram approach, the existing MySQL histograms can be re-used such that more time can be used on experimenting with the algorithm while less time is used on implementing it. A bonus with choosing the SPH algorithm is that a sampling approach has not previously been attempted and could give the algorithm the performance boost needed to fit into a real-world system.

This section will first look at the two versions of the SPH algorithm to understand the details of the approaches. This overview will give some clues about which parts of the algorithms cause the big time and space complexity. Further, some modifications to the approaches will be discussed to reduce the complexity to an acceptable level. Finally, the details of a MySQL implementation of the modified algorithm will be discussed.

4.1 Sequential Pattern-based Histograms (SPH)

In short terms, the SPH approaches in Aytimur and Çakmak, 2018 and Aytimur and Cakmak, 2020 can be divided into two parts:

1. Histogram construction
2. Selectivity estimation

Before these parts can be described in detail, an understanding of closed frequent pattern mining needs to be established. Definitions 1 through 4 give an introduction to the relevant terminology.

Definition 1. (Frequent pattern): A pattern P consisting of the characters C_1, C_2, C_n is a frequent pattern in the database D if P occurs in more than T of the entries in D, where T is a minimum support threshold.

Definition 2. (Closed pattern): If a pattern P_1 contains a pattern P_2, then P_1 is a superpattern of P_2. A pattern P is a closed pattern if there exist no superpatterns of P with the same frequency as P.

Definition 3. (Closed frequent pattern): A pattern P is a closed frequent pattern if it is both a frequent and a closed pattern.

Definition 4. (Closed frequent pattern mining): The process of finding all closed frequent patterns and their respective frequencies in a database.

4.1.1 Histogram construction

Even though there exist many approaches to closed frequent pattern mining (FP-tree, AprioriClosed, CHARM), the SPH approaches have employed the BIDE algorithm for this purpose (presented in Wang and Han, 2004). In order to give a good introduction to the BIDE algorithm, the following definitions will be needed:

Definition 5. (Projected sequence): Given a pattern P and a sequence S, the projected prefix sequence is the remaining part of S when the first occurrence of P has been removed along with all characters preceding it. For example, the projected sequence of the sequence ABDBAF with respect to the pattern AD is BAF.

Definition 6. (Pseudo-projected database): Given a pattern P and a database D, the pseudo-projected database is the projected sequences of all tuples in D with respect to P.

Initially, the BIDE algorithm finds all frequent patterns of length 1. From this point, the algorithm calls itself with each discovered frequent pattern and its respective pseudo-projected database in order to incrementally grow the discovered patterns until all frequent patterns have been discovered. Additionally, the BIDE algorithm uses a backward-extension pruning step to narrow down the search space and speed up the mining process. The more detailed description of the algorithm is outside the scope of this thesis, but for further reading, the original paper, Wang and Han, 2004, can be recommended.

As an extension to the BIDE algorithm, Aytimur and Cakmak, 2020 introduces positional sequence patterns. With this extension, two letters that show up next to each other will be handled separately from letter combinations that appear apart. As an example, consider a database that contains the names "Mike" and "Michael". For this database, the pattern "Mi" is an example of a frequent positional pattern, while "Me" is an example of a regular frequent pattern. In this case, "Mi" will be stored as-is, while "Me" will be stored as "M%e". The "%" in the last pattern is the wildcard character and symbolizes one or more characters between the letters.

The main problem with using the BIDE algorithm without any further pruning is that it would undoubtedly return many pretty similar patterns. For example, if the pattern "ADAM" is frequent in the database, one would, in many cases, expect to see "ADA" as a frequent pattern as well. Storing both of these patterns would not add much information to the histogram. The extended version of SPH avoids storing such combinations of patterns by using a measure of information content (IC) to prune the result of the BIDE algorithm. The information content is defined as

$$IC(R) = -loglogP(R) \tag{4.1}$$

where R is a frequent pattern and P(R) is computed as the frequency of pattern R divided by the number of rows in the database. Mined patterns are then pruned away from the result by using the following definition:

Definition 7. (Redundant pattern): A pattern P_1 is considered redundant if there exists another pattern P_2 such that $P_1 \neq P_2$, P_2 contains P_1 and IC(P_1) - IC(P_2) < T, where T is a small threshold.

The threshold used for this pruning process was determined experimentally in Aytimur and Cakmak, 2020, resulting in a threshold of -0.00216. The bucket sizes are then determined by dividing the total frequency of the resulting patterns by the number of buckets desired for the histograms, which was determined experimentally to be 1024. Finally, a sorted version of the mined patterns is looped through, and a histogram bucket is added for each time the current bucket is full. Even though each bucket contains several frequent itemsets, only one frequent itemset will be stored for each histogram bucket. Distributing the frequent itemsets this way has the effect of spreading the bucket endpoints out across the patterns such that a diverse set of patterns are kept in the histogram. Additionally, a frequent pattern has a better chance of appearing in the histogram than a rare pattern.

4.1.2 Selectivity estimation

The process of producing selectivity estimates using the generated histogram is quite straightforward. For a predicate P, the buckets of the histogram are looped through while distinguishing between three different types of matches in this prioritized order:

1. **Exact match:** If a bucket endpoint B exactly matches a predicate P, then an exact match is found between P and B.

2. **Encapsulated match:** There is an encapsulated match between a bucket B and a predicate P if all letters in P show up in the same order in B. For example, the predicate "A%B%C" would have an encapsulated match for the bucket "A%D%B%F%C".

3. **No match:** If a predicate P has no exact or encapsulated match to any of the histogram's endpoints, it ends up with a 'No match' case.

In case of an exact match, the frequency of the matched bucket is returned. For encapsulated matches, however, a predicate can match with multiple buckets. In such a case, the lowest frequency of the matched buckets is returned. In cases where no matches are found, the most recent version of the SPH algorithm turns to a partitioning-based matching to decide the selectivity. In this approach, a slider is moved across the predicate. Both sides of the slider are then checked for an exact or encapsulated match at each step. The thought behind this approach is that no substring of the predicate can have a lower selectivity than the predicate itself. While sliding across the predicate, the minimum selectivity is kept and eventually returned as the selectivity for this predicate. In order to avoid large over-estimations, the minimum length of the substrings is set to be one character shorter than the predicate itself. Finally, in the case where the partition-based matching does not find a match, the returned selectivity is set to 10% of the minimum support threshold of the BIDE algorithm. Aytimur and Çakmak, 2018 evaluated the optimal threshold to be 1.5%, meaning that the "no-match" selectivity is 0.15%.

4.2 Algorithm modifications

It has previously been stated that the space and time requirements of the SPH algorithm are too large to be used in any practical database setting. This section will present the modifications that have been done to reduce these requirements to acceptable levels.

4.2.1 Sampling

The simplest way to reduce the time and space requirements of the SPH algorithm would be to reduce the number of samples that are considered during item mining. The original implementation of the SPH algorithm uses the entire database of about 800 000 tuples, but one can usually get a good representation of a data set by considering far fewer tuples.

Thompson, 1999 looks closer at the effects that sampling has on item mining of association rules, which is a problem that is directly related to the process of mining closed frequent itemsets. The study examines mining of four different databases and concludes that sampling is an effective tool for data mining.

Chapter 5 and 6 will include experiments to evaluate the effects that sampling size has on the data set used in this thesis. These experiments will suggest the sample size needed to reach an acceptable time and space complexity for the BIDE algorithm and the impact this sampling has on the estimation accuracy.

4.2.2 Information content & positional sequences

The second version of the SPH algorithm (Aytimur and Cakmak, 2020) introduces two changes compared to the first version (Aytimur and Çakmak, 2018). These changes are the pruning of results based on information content and the extension to positional sequences. Introducing these two extensions increased the time complexity of the algorithm by a factor of up to three while only slightly increasing the accuracy of the model. For this approach to fit into a real-world system, the algorithm's time complexity needs to be reduced by a factor of more than 1000. Removing the extensions introduced in the second version of the algorithm is a small step towards that goal, but it will allow for a three times larger sample size, which could prove crucial for the estimation accuracy.

4.2.3 Exact and encapsulated matches

For a predicate to score an exact match in the original SPH approach, it needed to match the bucket endpoint exactly. However, if a predicate P is matched against the bucket endpoint s_1Ps_2, where s_1 and s_2 are arbitrary strings, it is known that P must have a frequency that is equal to or greater than the endpoint frequency. Due to this, the modified version of the SPH algorithm will accept this as an exact match and then return the maximum of all exact matches (if there are more than one). An example of such an exact match would be matching the predicate 'en' against the bucket endpoint "lent". In this case, "en" must have a sampled frequency that is at least equal to that of 'lent' in the sample.

For encapsulated matches, on the other hand, the original version of the SPH approach returned the minimum frequency of all matched endpoints. Even though this makes sense given that the frequencies in the histogram are correct, it turns out to introduce an additional risk when the frequency counts are imperfect. In these cases, returning the average frequency of all matches gives a far more stable selectivity estimate. Due to this, the new approach will return the average of all encapsulated matches rather than the minimum.

4.2.4 Estimation of long predicates

The main issue with histograms consisting of frequent character patterns is that only short strings are represented in the bucket endpoints. Having only short strings in the buckets has an unfortunate effect when longer predicates are estimated using the histogram. Since each bucket endpoint contains a pattern no longer than 5 or 6 letters, each predicate that exceeds this length would have no chance of being matched in the histogram.

The SPH algorithm introduces an approach using partition-based matching to handle this. However, the minimum length of the strings on each side of the slider is set to be one letter shorter than the predicate itself. This approach essentially means that only two long substrings of the predicate will be considered (excluding the first and the last character). This solution has little effect other than allowing predicates to be one letter longer than the bucket endpoints against which they are matched.

To solve this issue, the implementation of the SPH algorithm that is used in this thesis divides long predicates into multiple short strings of lengths less than or equal to three letters. The selectivity estimates of these substrings are then combined in different ways to estimate the selectivity of the predicate as a whole.

4.3 MySQL implementation

Evaluating the effect that the SPH algorithm has on the MySQL system would require a working implementation of the algorithm inside the code base of MySQL. The source code of the SPH algorithm with the previously discussed modifications can be found in Appendix B. The BIDE part of the algorithm is a C++ translation of this Python implementation by Robert Yu. Apart from this, the histogram feature introduced with MySQL 8 has been taken into use by simply overriding the old system's code for histogram construction and selectivity estimation. This re-cycling of older components allows most of the existing code base to be re-used without further modifications, including MySQLs approach to sampling.

Chapter 5

Experimental setup

In the following sections, some experiments will be presented before they will be conducted in Chapter 6 to evaluate the proposed solution. The initial experiments will be performed using a Python version of the modified approach discussed in Section 4.2. For the initial experiments a self-standing implementation is used in order to separate the proposed approach entirely from the MySQL system. Finally, the proposed approach will be implemented into MySQL to see the effect the new approach has on more complex queries.

5.1 Dataset

All experiments will be performed using the same snapshot of the IMDb database as in Leis et al., 2018. In particular, the 'name' column of the 'name' table has been chosen to stay as close as possible to the data load used in Aytimur and Çakmak, 2018 and Aytimur and Cakmak, 2020. The 'name' column contains 6.4 million records with an average length of 14.6 characters. The minimum and maximum lengths are 1 and 106 characters, respectively.

5.2 Query workload

The query workload used in the experiments consists of three separate sets of queries. The first two query sets have been generated similarly to Aytimur and Çakmak, 2018, but with a few minor tweaks. The last query set represents a more realistic set of queries that are more likely to be queried in a real-world system. All query generation and sampling have been done using a seeded random generator such that the results are reproducible.

5.2.1 Positive queries

The first set of queries are positive queries, meaning that all queries have a non-zero selectivity. Initially, 100 names are sampled from the database's 'name' column before one name (the first name, a middle name, or the last name) is chosen from each of these 100 sampled rows. If the sampled tuple contains the name 'Robert Downey Jr.', either 'Robert', 'Downey' or 'Jr.' would be picked uniformly at random. Making this adjustment is necessary to produce predicates with the desired variations in selectivity.

For each sampled name, a random number of letters are removed from the name such that at least one letter remains and at least one letter is removed. Next, between 2 and 8 wildcards are inserted into the reduced name at random locations in the string. Since the focus area of this thesis is MySQL's selectivity estimates in the presence of preceding wildcards, one additional wildcard is prepended to all predicates.

The result of this predicate generation is 88 predicates (12 queries had a selectivity of 0 or were duplicates) with an average length of 5.6 characters (including wildcards) and maximum and minimum lengths of 15 and 2 characters, respectively. The queries have a selectivity ranging from $1.10 * 10^{-4}\%$ to 15.2%, and an average selectivity of 1.6%.

As an example, the first sampled name was 'Karmarkar Saswati'. Initially, 'Karmarkar' was picked at random from the two names. The number of letters to be removed was decided to be 7, specifically the indices [0, 1, 2, 3, 5, 6, 7], resulting in the string 'ar'. Next, the number of wildcards to be inserted was randomly set to 2, specifically the indexes [0, 1]. The final predicate was thus '%a%r'.

5.2.2 Negative queries

The second set of queries contains predicates that have zero selectivity in the dataset. These predicates have been generated in the same fashion as the positive queries but with a few adjustments. The number of letters to be removed is kept low enough that the remaining string contains at least four characters, while the number of wildcards to be inserted is between 1 and 3. This procedure will yield a more significant proportion of zero-selectivity predicates than what was seen in the positive query generation.

Any use of indices is avoided by ensuring that all predicates have a preceding wildcard. This is ensured by moving the first wildcard in each predicate to the front of the predicate.

The generation of negative predicates resulted in 78 predicates (22 had a non-zero selectivity) with an average length of 8.9 characters (including wildcards) and maximum and minimum lengths of 18 and 3 characters, respectively.

One example from the generation of negative predicates is the case where 'Hrgovic Zdenka' was sampled. The indices of letters to be removed was [2, 4, 12], resulting in 'Hroic Zdena'. The number of wildcards to be inserted was one, specifically index 2. The result so far in the process was 'hr%oic zdena'. Finally, to avoid any use of indices for selectivity estimation, the first wildcard of the predicate is moved to the front of the predicate, meaning that the final predicate, in this case, is '%hroic zdena', which is indeed a zero-selectivity predicate.

5.2.3 Name queries

Although the two workloads presented so far do measure the system's general ability to store and use compressed statistics about the contents of a database column, they do not, in most cases, represent a realistic workload. Consider a case where a relatively short string that should be present in all the results is known. For example, the first name of an actor. The best approach to solve this problem would be to make a query for "name LIKE '%name%'". A third query workload is generated to represent such a use case. The workload consists of 100 queries with random names seen in the database. Some of the names are first names, some are last names, and some are middle names.

The 100 generated predicates in this set of queries have an average length of 7.8 characters (including wildcards) and maximum and minimum lengths of 13 and 4, respectively. The queries have a selectivity ranging from $1.6 * 10^{-5}\%$ to 12.5% and an average selectivity of 0.3%.

An example of a predicate in the third generated data set is '%allan%'. This predicate was generated by first sampling the name 'Allan Cook' and then randomly choosing the first name as the predicate.

5.3 Sample size

Aytimur and Çakmak, 2018 and Aytimur and Cakmak, 2020 showed in their experimental evaluation that the construction time of their histograms was in the range of 10 to 30 hours. This construction time makes the approach infeasible to implement into the MySQL system. Although the modified approach proposed here is a bit less computationally heavy, it can be assumed that building a histogram by using the entire database column is still far too computationally expensive to be used in a real-world database, even considering the fact that the construction is done in an offline fashion. Due to this, the initial experiments of Chapter 6 will concern the effects that sample size has on the modified SPH approach.

5.3.1 Histogram accuracy

One experiment will be performed for each of the three query workloads to evaluate the effect that sample size has on histogram accuracy. By evaluating this effect on three different query workloads, it will become clear how differences in query workloads affect the accuracy. The experiments will build 20 histograms for each sample size from 30 to 1000 (in steps of 10) before evaluating the query workloads in each set of histograms. The results will show the average estimation error for each sample size.

Since the current approach of the MySQL system provides a feeble comparison concerning the accuracy, the approach of PostgreSQL (described in Section 3.3) will be used as a baseline. PostgreSQL offers two separate operators for string matching, namely the LIKE and ILIKE operators. The Postgres LIKE operator is case sensitive while the ILIKE operator is not. Since the MySQL LIKE operator is insensitive to case, the Postgres ILIKE operator will be used throughout the experiments.

Similarly to Aytimur and Çakmak, 2018, both relative error and absolute error will be used to measure the accuracy of the selectivity estimates. For queries that return at least one row, the relative error will be used, while for queries that return zero rows, the absolute error will be used. Relative error is defined as $|S_{true} - S_{est}|/S_{true}$, where S_{true} is the true selectivity, S_{est} is the estimated selectivity. Absolute error is defined as as $|S_{true} - S_{est}|$, (for negative queries this will simply be S_{est}, since the true selectivity is always zero), where '|n|' is the absolute value of n.

5.3.2 Construction time

The next experiment will concern the construction time of the new histogram approach. The current histogram feature in MySQL will be used as a baseline to ensure that the proposed approach has a manageable computational complexity. The estimation process of the newly proposed approach happens in two steps. The initial step is the construction of the histogram, which is performed in an offline fashion. The second step is the actual selectivity estimation of ad-hoc queries, which must be performed in an online fashion. Similarly, the current histogram feature of MySQL uses an offline building step and then an online estimation step. As long as the new approach has a similar computational complexity for these two steps as the current system, it can be assumed that the proposed approach has a manageable complexity and can be introduced into the MySQL system.

The experiments concerning construction time are performed similarly to the experiments concerning estimation accuracy. The time spent to construct 20 histograms for each sample size from 30 to 1000 (in steps of 10) is recorded. These construction times are then compared against the current MySQL histograms.

5.4 Composite queries

Although the experiments presented this far goes a long way to show the effects of the new approach compared to the current solution of MySQL, they will not give any clues about the effects that a new selectivity estimation approach for string matching will have on the MySQL system as a whole. The algorithm implementation discussed in 4.2 will be used to see the effects of the new approach. The source code of the essential parts of the implementation can be found in Appendix B.

The query set used in this experiment will consist of 73 queries from the Join Order Benchmark (queries can be found here). The queries are hand-picked such that they include at least one LIKE predicate with a leading wildcard. Since execution times can vary quite a bit between executions, each query will be executed ten times in each version of the system, and the average of the execution times will be the basis for further analysis.

After the initial ten executions are completed, each query will be executed one additional time with the MySQL EXPLAIN ANALYZE command to get the plans used through the experiment. The MySQL sampling seed will remain fixed through the entire experiment, meaning that the same histogram will be generated for the same table at each execution. This consistency will allow the experiment to be reproduced and force the optimizer to pick the same plan for each execution (since the histogram will be equal). Comparing the average execution times across the ten executions for the two versions of the system will give some clues about how much the system relies on the selectivity estimates and how much the estimates impact the optimizer's decisions.

Finally, a handful of queries will be chosen based on the results this far in the experiment. The selected queries will then be examined more closely by comparing the generated plans, selectivity estimates, and execution times of the two versions of the system. This examination will reveal the cause of any difference in execution plans or execution times.

Chapter 6

Experimental results

The following sections will walk through the results from the experiments presented in Chapter 5. All experiments have been performed on a machine running Ubuntu 20.04 with an Intel Core i7-8655U 1.90GHz CPU and 16GB of RAM. The SPH method proposed in Aytimur and Çakmak, 2018 has been modified according to the description in Chapter 4. The minimum support threshold and the number of buckets have been kept to the optimal values found in Aytimur and Çakmak, 2018 except stated otherwise. Those values were 1.5% and 1024, for threshold and number of buckets, respectively. The MySQL and Postgres servers have been run locally with default settings.

6.1 Varying sample size

Aytimur and Çakmak, 2018 gives an experimental evaluation for finding the optimal threshold and number of buckets for their algorithm. The next step of this process is to find an optimal sample size such that the estimation error is as low as possible while keeping an acceptable construction time. Since the optimal sample size could, in theory, be quite different for various query workloads, this experiment will be conducted for each of the three workloads.

6.1.1 Positive and negative query sets

For an approach to yield good selectivity estimates for longer coherent predicates (such as those in the names query workload), the approach must first give good estimates for less complex predicates. Due to this, the experimental results for the positive and negative query workloads will be presented first because those contain shorter predicates than the name workload. Those results will then be built upon when considering the more complex predicates in the names workload.

Positive query accuracy

Figure 6.1 shows the average relative error for queries in the positive query workload as the number of samples used to build the histogram is varied. The main takeaway from this result is that there is very little to be gained using a sample size of more than 200 tuples in terms of a decrease in error. The variations seen for larger sample sizes are likely to be caused by the specific random seed used, and have little impact on the results.

Figure 6.2 focuses on the critical area between sample sizes 130 and 270 of Figure 6.1. The figure includes three different approaches for handling situations where multiple buckets partially match a predicate. The 'avg' line represents returning the average of all matching buckets, which is the same approach that was used in

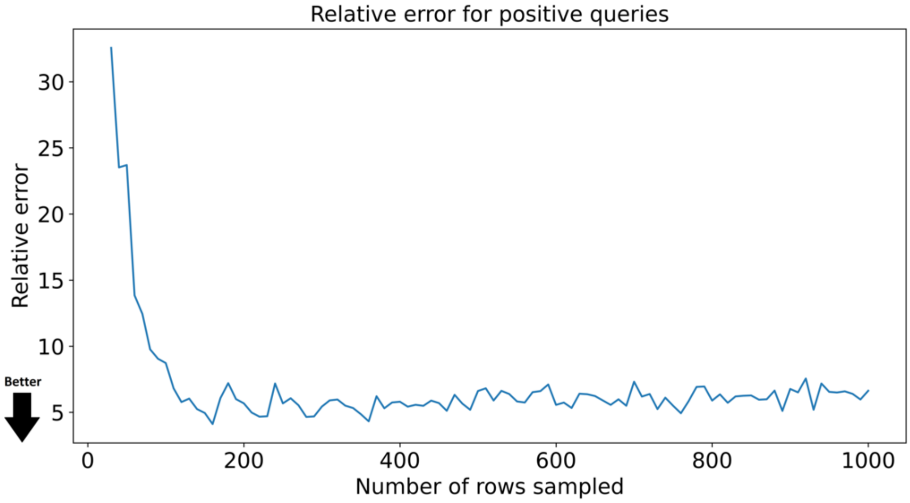


FIGURE 6.1: Relative error for queries with non-zero selectivity.

Figure 6.1. The ’min’ and ’max’ lines, however, represent picking the minimum and maximum bucket frequencies, respectively.

The fact that the average is consistently the best choice indicates that the predictions have a combination of over and underestimations. Further, since the ’min’ line closely mimics the ’avg’ line, it can be assumed that the average bucket frequency is close to the minimum bucket frequency. This indicates that the histogram contains a few high frequency buckets, but many more low frequency buckets pulling the average down.

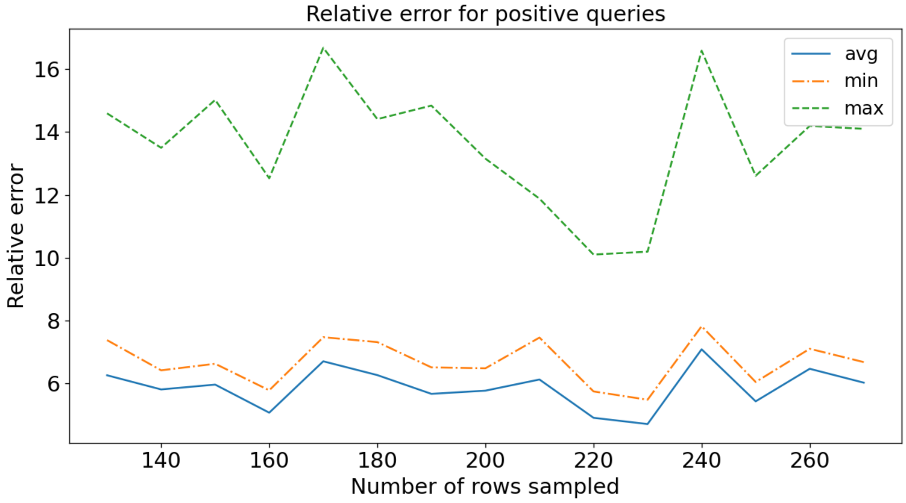


FIGURE 6.2: Relative error for queries with non-zero selectivity. The different lines represent picking the minimum, maximum and average selectivity whenever multiple buckets partially matches a predicate.

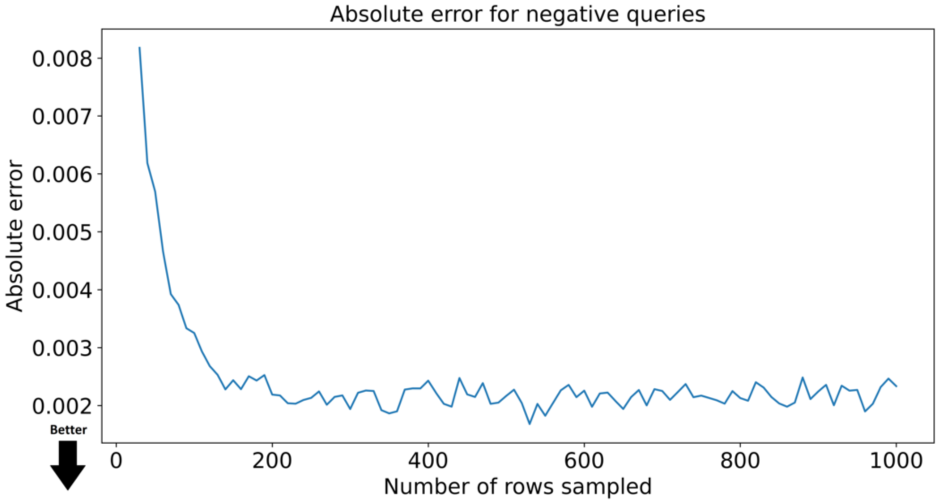


FIGURE 6.3: Absolute error for negative queries.

Negative query accuracy

The average absolute error for the zero-selectivity queries can be seen in Figure 6.3 as a function of the number of samples used during histogram construction. Similar to the results from the positive queries, it can be seen that the curve flattens from around 200 samples. These two observations combined suggest that 200 could be a good sample size for short string predicates.

6.1.2 Name query set

Although the results so far have shown low relative errors for non-zero selectivity predicates, those predicates have only included short strings (between wildcard characters) such that each substring in a predicate is likely to be found within a bucket in the histogram. The third query workload does, however, involve predicates that are longer than the strings stored in the buckets (which are typically 3-5 characters long), making it impossible to apply the predicate directly to the histogram.

As a solution to this problem, the following two approaches have been considered:

- **Independence assumption**: Long predicates are divided into substrings of three characters before using the histogram to get estimates for these substrings. Finally, the selectivities for these letter triplets will be multiplied together. As an example, the final selectivity estimate for the predicate 'String', $sel(string) = sel(str) \times sel(ing)$.

- **Short identifying substring assumption**: Whenever a predicate exceeds four letters, a window of three letters is moved across the string, returning the lowest seen selectivity. Using larger windows would not add any information since the length of the bucket endpoints is the limiting factor. As an example the selectivity of 'string' sel(string) will be estimated as MIN(sel(str), sel(tri), sel(rin), sel(ing)). This approach will work according to the theory of short-identifying substrings (introduced in Chaudhuri, Ganti, and Gravano, 2004) given that the identifying substrings have a length of three letters or less.

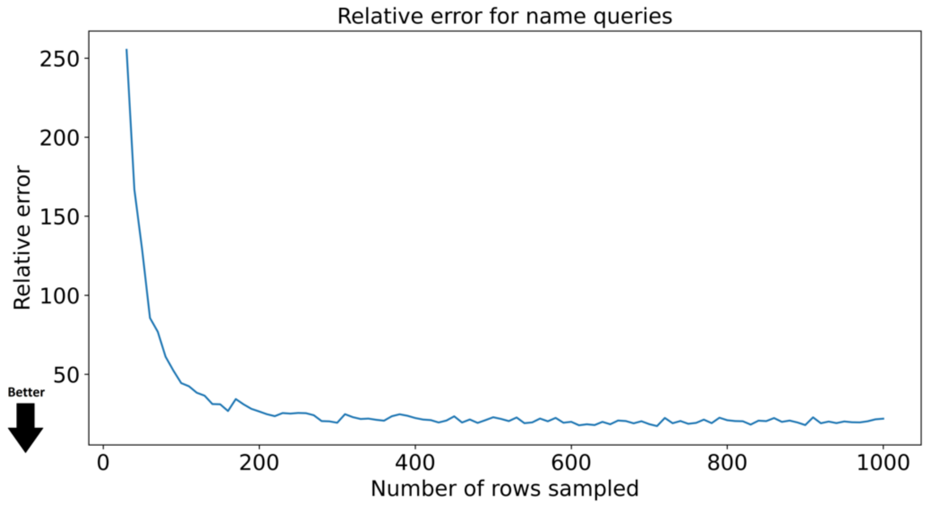


FIGURE 6.4: Relative error for long coherent predicates in the 'name' query workload.

An evaluation of these two approaches proved that the theory of short identifying substrings of length three or less did not hold for the 'name' column and yielded an error that was orders of magnitude worse than what was seen using the independence assumption. Due to this observation, the results of using short identifying substrings have been omitted throughout this section.

The relative error for the name queries is represented in Figure 6.4. Comparing the results to the positive and negative query workload results, one can see a similar accuracy curve for all experiments. Since the histograms are in all workloads used to estimate short strings (the name predicates were divided into substrings), it is not unexpected that the sample size has the same effect on the estimation accuracy for all workloads. A similar number of character combinations will be recorded and used in all three query workloads, the only difference being which combinations.

The same trend can be seen in the graphs of all three workloads regarding the optimal sampling size. This suggests that a sample size of 200 gives a reasonably good accuracy while keeping the sampling size (and thus the computation time) low. For the first experiment, the relative error flattened at about 5, whereas the third workload has a similar flat curve at a relative error of 25. This result indicates that the accuracy loss from using longer predicates is about 500%.

6.1.3 A comparison against previous approaches

A comparison of the estimation errors in PostgreSQL, MySQL, and the new histogram approach can be seen in Figure 6.5. The estimation error of PostgreSQL has been used as a baseline, represented by '1' on the y-axis. The y-axis is scaled logarithmically in order to fit the MySQL error within the same graph. As expected, both the modified SPH approach and PostgreSQL outperforms MySQL by orders of magnitude for all query workloads.

When it comes to comparing PostgreSQL and the proposed histogram approach, they both have their strengths and weaknesses. For the first query workload, the difference between the two approaches is only 18% in favor of PostgreSQL. The negative query set does, however, prove difficult for the modified SPH approach. Although the absolute error is only 0.2%, it is outperformed by PostgreSQL by one

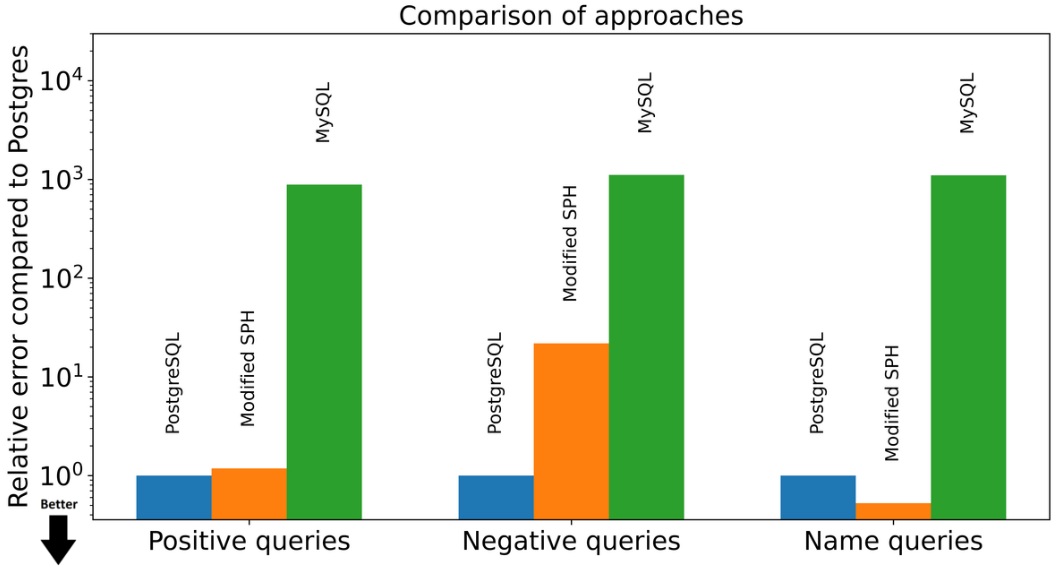


FIGURE 6.5: Relative error compared to PostgreSQL. A value of 1 corresponds to a similar error to PostgreSQL, while a value of 10 corresponds to an estimation error 10 times as large as PostgreSQL.

order of magnitude. Finally, for the more realistic third query set with longer coherent predicates, the relative error of 50 observed in PostgreSQL is twice that of the proposed approach.

Considering the vast spectrum of use-cases that can be seen for databases these days, it can be argued that a real-world system should handle any workload reasonably well in order to be a good solution. With this argument, the PostgreSQL solution might be a better solution than the proposed histogram approach. On the other hand, a case can be made for implementing features that are not optimal for all cases and then leaving the decision to the DBAs on whether to use the feature or not. The current version of histograms in MySQL is an excellent example of this. They can be pretty helpful in some specific cases, but since histograms are not automatically updated, index statistics are always preferred whenever they are available. This solution means that the DBA must decide whether to use a histogram or an index for each individual column.

6.1.4 Histogram construction time

Even though good estimation accuracy has been seen so far for the new approach, there is another factor that is equally important. For a new approach of selectivity estimation to be relevant to the MySQL system, the algorithm must not be too computationally expensive. In the current system (MySQL 8.0.23), a histogram can be computed for any column in a database schema. Creating a histogram with default sampling size on the 'name' column used in the experiments uses 80 000 samples and takes anywhere between 0.55 and 0.70 seconds. Computing a similar histogram on the integer column 'production_year' in the 'title' table uses 400 000 samples and takes anywhere between 1.80 and 2.00 seconds.

Considering the time it takes to compute the current histogram solution, it can be argued that a building time of 2 seconds is acceptable in the MySQL system. Thus

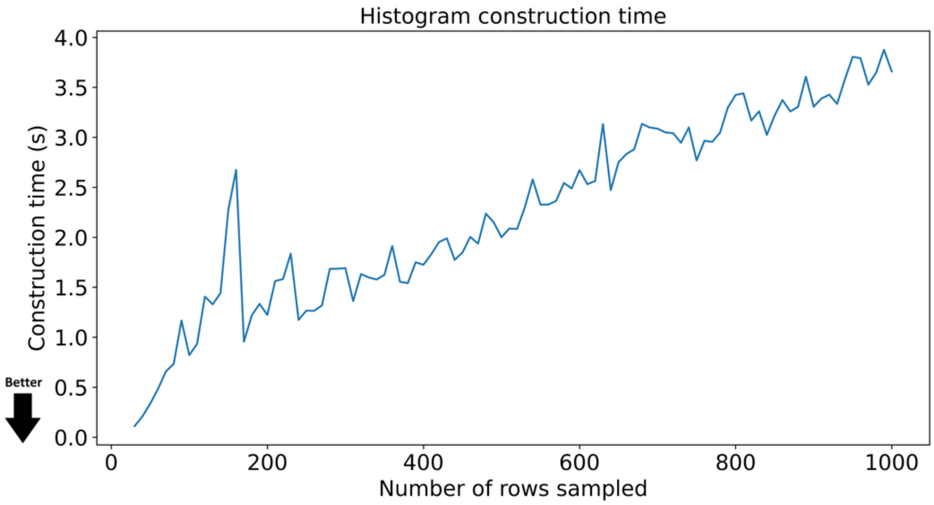


FIGURE 6.6: Construction time of new histogram approach with varying sample sizes.

any new approach should aim for a building time that does not exceed this limit. Figure 6.6 shows the average construction times for building a histogram on the 'name' column. For each sample size, ten separate histograms have been generated.

Figure 6.6 shows the contours of a linear relationship between construction time and the number of samples used during construction. Considering the relationship between estimation error and sample size seen in the previous section, one can conclude that a sample size of 200 is acceptable in terms of both error and construction time. At this point of the graph, the average construction time is 1.33 seconds, which is lower than the current construction time for integer histograms by a comfortable margin, and thus can be considered an acceptable construction time.

Another observation that can be made from Figure 6.6 is an immense growth in construction time between 150 and 200 samples before the construction time falls into linear behavior. This behavior can be accounted for by considering the threshold of 1.5% that was used throughout these experiments. At 160 samples, the threshold is set to 2.40, which is rounded down to 2. At 170 samples, however, the threshold is 2.55, which will be rounded up to 3. These results show that the threshold has an enormous impact on construction time when the sample size is small. Following these observations, one can see that the results from Aytimur and Çakmak, 2018 concerning the optimal threshold do not necessarily hold for smaller sample sizes. Due to this, the next section will have a closer look at the effects that the threshold has on both estimation error and construction time.

6.2 The effect of threshold

The observations made in the previous section suggested that the threshold has a more considerable effect on construction time when the sample size is smaller. Since acceptable error sizes and construction times have so far been seen for a sample size of 200, the following experiments will keep the sample size fixed at 200 tuples while varying the threshold from 0.5% to 5.0%. Ten executions will be done for each threshold.

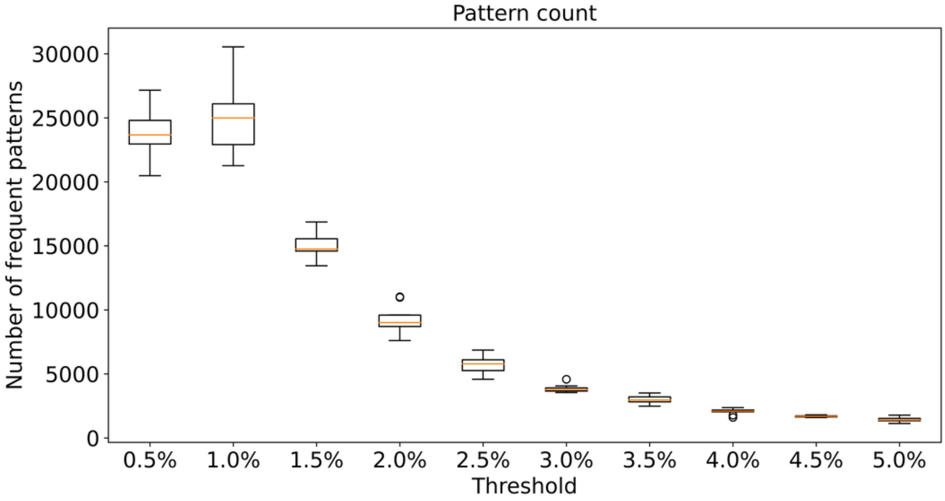


FIGURE 6.7: Number of frequent patterns with varying thresholds.

Figure 6.7 shows the number of patterns that exceed the various thresholds from 0.5% to 5.0%. Considering this result, it seems natural that the construction time will be higher for lower thresholds simply because more patterns need to be considered. Figure 6.8 shows that the average histogram construction time has a shape similar to the pattern count. Taking only these results into account, one would always prefer higher thresholds in order to save mining time. On the other hand, higher thresholds would record fewer patterns, and thus, the resulting histogram would represent a smaller share of the information stored in the column. This could, in turn, have a negative impact on estimation accuracy.

Another result of increasing the threshold is a reduction to bucket endpoint lengths. Shorter patterns (fewer characters) will usually, compared to longer patterns, have a greater chance of passing a higher threshold. Due to this, an increased threshold would result in a histogram consisting mainly of short bucket endpoints of two and three characters.

Figure 6.9, shows the estimation error as a function of the threshold for positive queries, negative queries, and name queries. The results show a steadily growing estimation error from a threshold of 1.0% for all three data sets. Looking at the estimation error isolated, one can conclude that the best threshold is at 1.0%, where the most information from the underlying column of data is available.

Both construction time and estimation error do, however, need to be considered as a whole to find the best threshold. Previously, the results showed that the mining time had a significant drop from 4 seconds to 2 seconds between a threshold of 1.0% and 1.5%. Additionally, it can be recalled that the current MySQL histogram has a construction time of about 2.0 seconds, meaning that a threshold of 1.0% would give a significant performance loss compared to the current solution. Considering these results as a whole, one can conclude that a threshold of 1.5% gives the best balance between construction time and error size. This conclusion is also supported by the observations in Aytimur and Çakmak, 2018.

Considering all the observations throughout this section, one can see that the extreme peak in construction time that was seen in Figure 6.6 does not pose any challenge to the algorithm's computation time. It is, however, essential to avoid the exact combinations of sampling size and threshold that give an increased computation time. Thus, through the rest of this thesis, the sampling size will be kept at

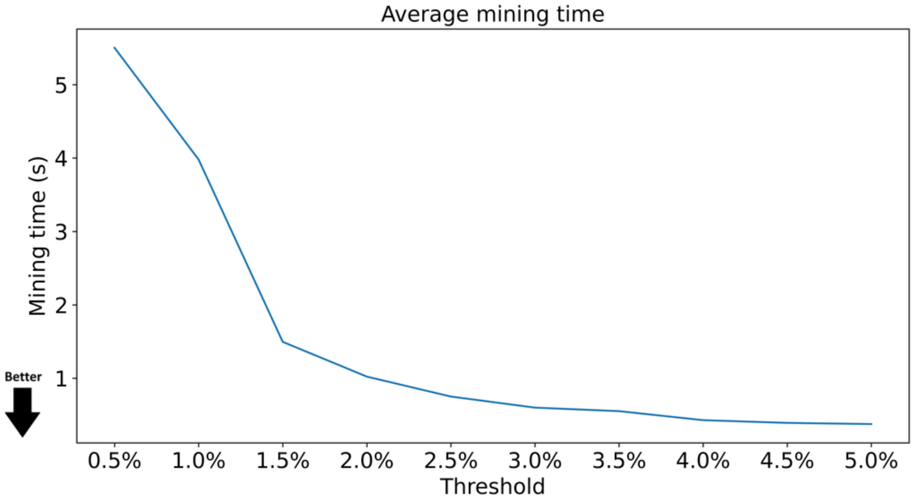


FIGURE 6.8: Average mining times for varying thresholds. A higher threshold will exclude more patterns early in the mining process, and thus have a quicker computation time.

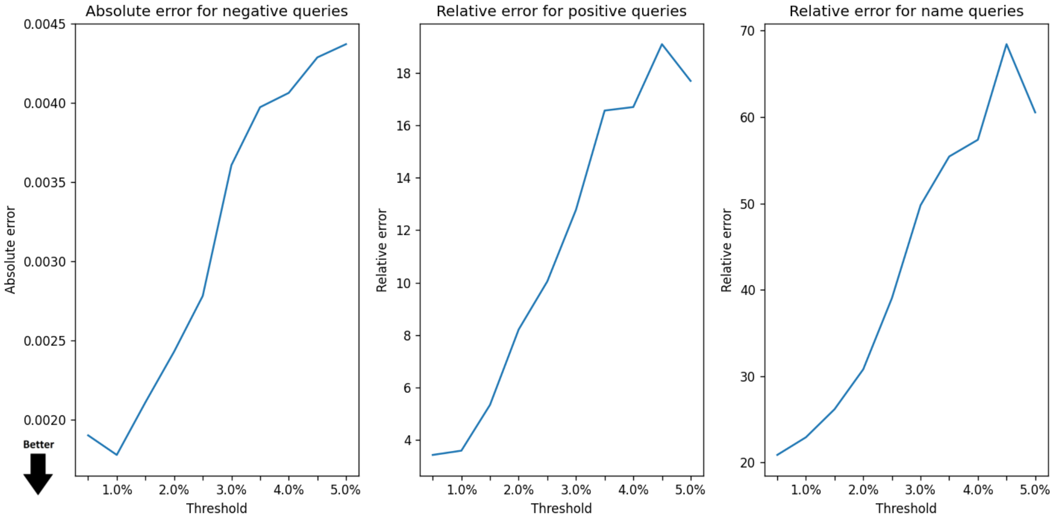


FIGURE 6.9: Relative and absolute error for all three data sets with varying threshold. A histogram built with a higher threshold will exclude more patterns, and thus contain less information about the underlying data, resulting in a higher error rate.

200 tuples, while the threshold will be kept at 1.5%, giving a stable mining time of slightly less than two seconds.

6.3 Composite queries

Moving on to considering the proposed approach within the MySQL system, Figure 6.10 shows the relative improvement in execution time for each of the 73 queries ordered by improvement. Each bar represents a single query, with the relative improvement being represented logarithmically along the y-axis. Among the 73 queries, the results show an improvement to execution time for 60 queries, while 12 queries executed slower with the new selectivity estimates (one query had the exact same execution time in both systems). Adding up the average execution time for all queries shows an improvement from 4249 seconds in the old system to 3164 seconds with the SPH implementation, giving a total improvement of about 26%. This improvement results from the fact that the optimizer has a more accurate view of the result sizes of temporary results, which, in turn, gives an execution plan closer to the optimal plan.

For quite a few of these queries, the change in execution time is so small that it can result from randomness. Many factors affect execution times in computers, such as the contents of the cache and the state of the host. These factors should be considered while interpreting the results. Considering only queries that have improved or worsened by a factor of more than 1.5, the results show 35 queries with faster execution times and six queries with slower execution times.

Even though the total execution time has decreased significantly, some queries are still slower in the new version of the system. In order to find the cause, these particular queries need to be examined further. Three queries will be picked from both ends of the scale (the three queries that improved the most and the three queries that worsened the most) and analyzed further through this section. For the selected queries, the output of the MySQL EXPLAIN ANALYZE command will be used to inspect the details of the plans that were selected by the optimizer.

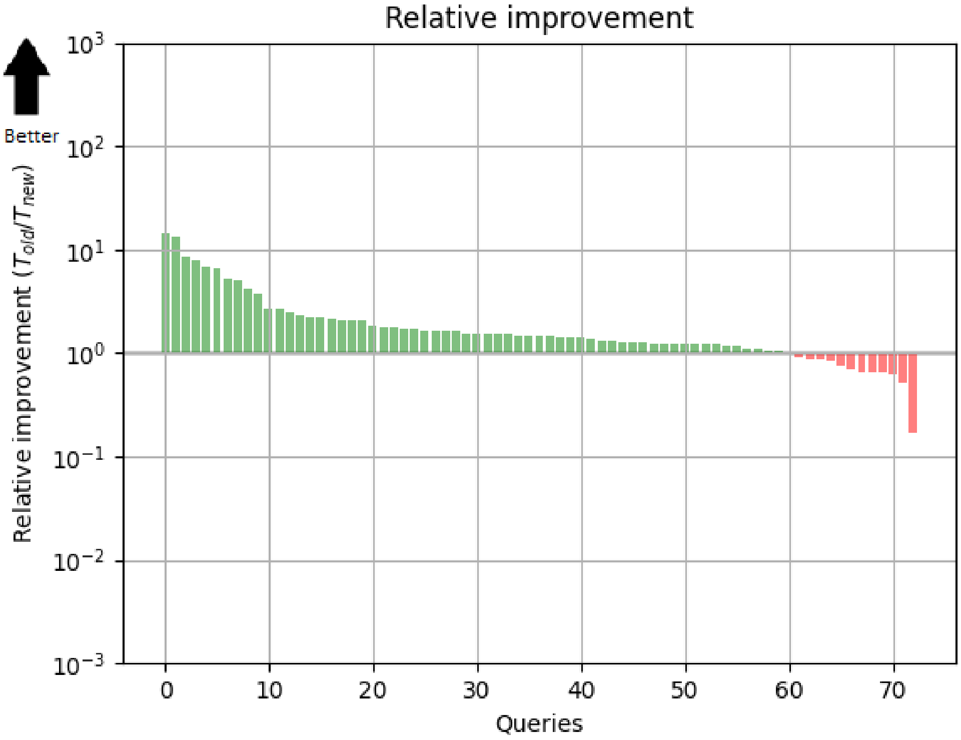


FIGURE 6.10: Relative improvement (old_time / new_time) for all selected queries. Each bar represents a single query, and the queries have been sorted from best to worst improvement. The scale of the y-axis is logarithmic with base 10.

6.3.1 Queries

The top three and bottom three queries of Figure 6.10 are listed in Table 6.1 with their respective execution times and relative improvement. This section will examine the generated execution plans for all these queries, and the differences between the old and the new system will be discussed. Whenever a query plan is expressed in a tree format, the following symbols will be relevant:

- $\Pi = Project$
- $\sigma = Select\ (with\ a\ filter)$
- $\bowtie = Join$

Query name	Old execution time (s)	New execution time (s)	Relative improvement
13B	40.25	2.81	14.325
5C	109.84	8.15	13.477
4B	3.16	0.37	8.485
24B	0.05	0.08	0.636
8B	0.01	0.02	0.632
10A	8.10	47.20	0.172

TABLE 6.1: The selected queries with their respective execution times and relative improvement. Execution times have been rounded to two decimal places, while relative improvements have been rounded to three decimal places.

Query 13B

Figure 6.11 shows the old (figure a) and new (figure b) execution plan for query 13B (found in Appendix A.1) in a tree format. In the old query plan, the tables "company_names" and "movie_companies" were joined at the bottom of the graph. This join produced a result size of just above 1 million rows that needed to be considered in the next join with the "title" table.

On the other hand, the updated query plan switched the locations of "title" and "company_names" in the tree. This change to the execution plan reduced the temporary result size of the first join to less than 1000 rows and allowed for a much cheaper join at the next step.

One can see that the new query plan is superior to the old, considering the temporary result sizes. To find the cause of this change of plans, however, the selectivity estimates of the two systems must be examined.

The old version of the MySQL optimizer estimated the LIKE predicates in query 13B to have a combined filtering effect of about 0.18. The correct filtering effect, however, was 9.4×10^{-4} resulting in a relative error of about 200. On the other hand, the new version of the optimizer estimated a selectivity of 9.4×10^{-5}, which gives a relative error of only 10. Due to this more precise estimate, the optimizer could push the join with the title table to the bottom of the execution tree without risking a large result size throughout the rest of the execution.

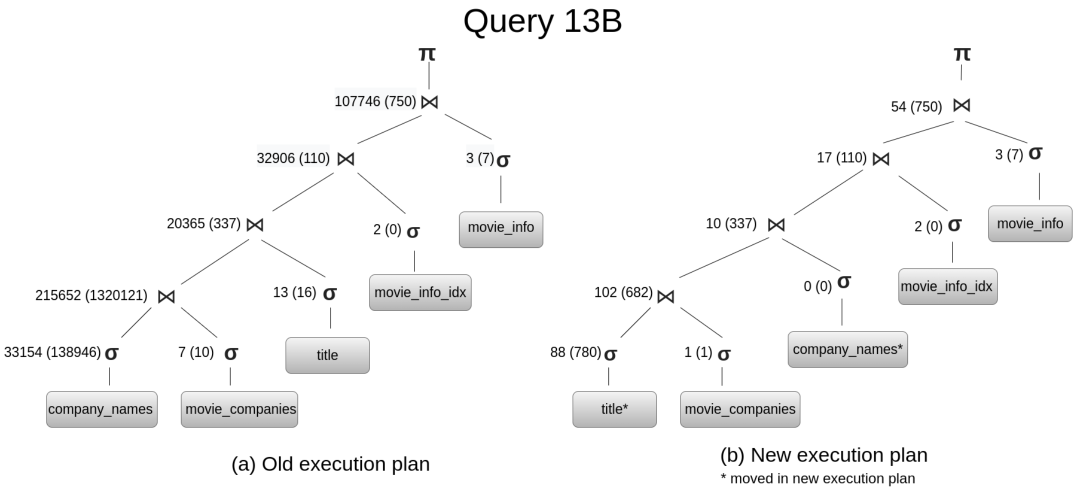


FIGURE 6.11: Old and new execution plan for query 13B of the imdb dataset. Result sizes are showed in the format 'Estimated (Actual)' beside each operator.

Query 5C

Figure 6.12 shows the old (figure a) and new (figure b) execution plan for query 5C (found in Appendix A.2) in a tree format. In the old query plan, the tables "info_type" and "movie_info" were joined at the bottom of the tree. This join produced a result size of about 1 million rows, which had to be further considered in the next join with the "movie_companies" table. The result size after the second join was greatly reduced compared to the first temporary result. On the other hand, the updated query plan pushed both the "title" and the "movie_companies" tables to the bottom of the execution graph. The join between these two tables resulted in just above 1000 rows that were further kept constant through the final two joins.

One can see that the new query plan is superior to the old, considering the temporary result sizes. To find the cause of this change of plans, however, the selectivity estimates of the two systems must be examined.

The old version of the MySQL optimizer estimated the LIKE predicates in query 5C to have a combined filtering effect of about 0.09. The correct filtering effect, however, was 0.08 resulting in a relative error of only 5%. The new version of the optimizer, on the other hand, estimated a filtering effect of 1.4×10^{-3}, giving a relative error of about 60.

One can see that the selectivity estimate of the new optimizer is much less accurate than the old version. Still, however, the new version picks a better execution plan for the query. The reason for the changed execution plan is a large underestimation on the "movie_companies" table, causing the optimizer to push it to the bottom of the graph. Luckily the "title" table is correlated with the "movie_companies" table in such a way that they produce only a tiny temporary result. Even though the new version of the optimizer found a better execution plan for query 5C, this must be mainly addressed to a lucky coincidence.

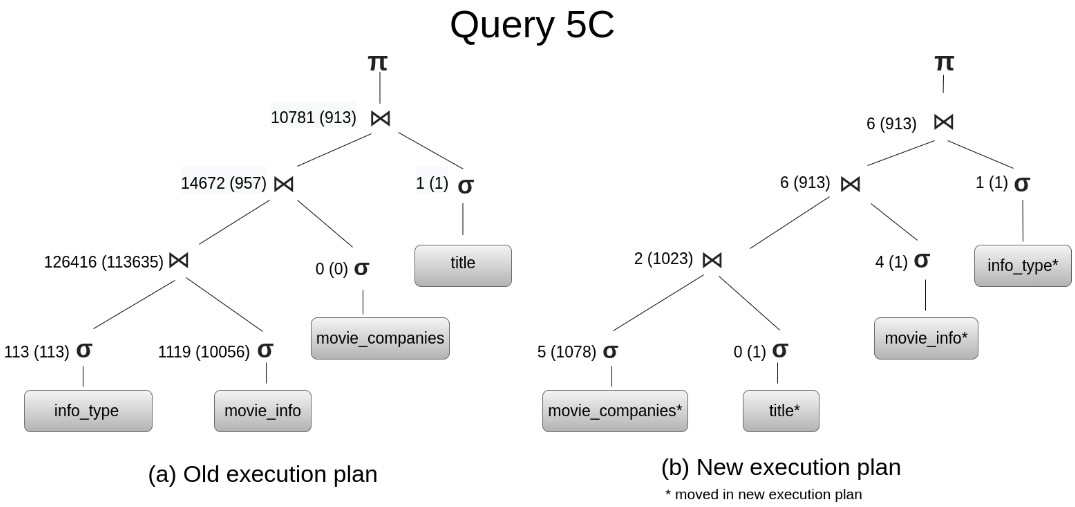


FIGURE 6.12: Old and new execution plan for query 5C of the imdb dataset. Result sizes are showed in the format 'Estimated (Actual)' beside each operator.

Query 4B

Figure 6.13 shows the old (figure a) and new (figure b) execution plan for query 4B (found in Appendix A.3) in a tree format. In the old query plan, the tables "movie_info_idx" and "title" are joined at the bottom of the graph. This join results in a temporary result of about 12 000 tuples that is further increased to 33 000 tuples after the join with the "movie_keyword" table. On the other hand, the updated query plan pushes the "keyword" and "movie_keyword" tables to the bottom of the graph. This change is similar to what was seen for query 5C, where the join order was also completely shuffled up. The new join order increased the size of the first temporary result by 2000 tuples but then reduced the second temporary result to about 1000 tuples. Thus, in the last join, the new plan executes far quicker than the old.

In the old version of the MySQL system, the filtering effect of the LIKE predicate in query 4B was estimated to be 0.11. The actual filtering effect, however, was 1.7×10^{-4}, resulting in a relative error of about 640. In the new version of the optimizer, the filtering effect is estimated to 4.2×10^{-6}. The relative error of this estimate is about 40, which is far better than the old optimizer but still far from the truth. The cause of the underestimation made by the histogram is that none of the substrings in "sequel" found a match in the generated histogram. The change from an overestimation in the old optimizer to an underestimation in the new optimizer suggests to the optimizer that the result size of the first join would be even smaller than it was. It is, therefore, natural that the "keyword" table is pushed to the bottom of the tree.

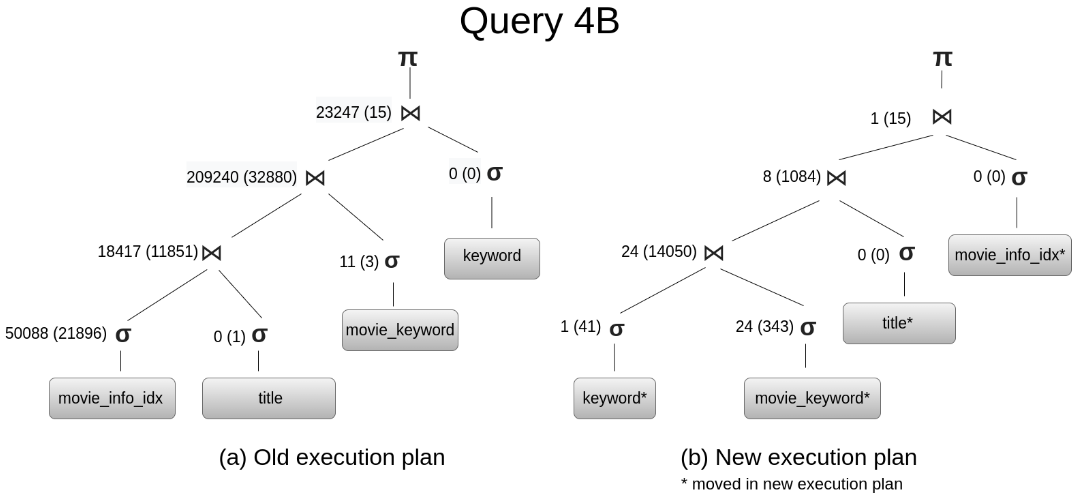
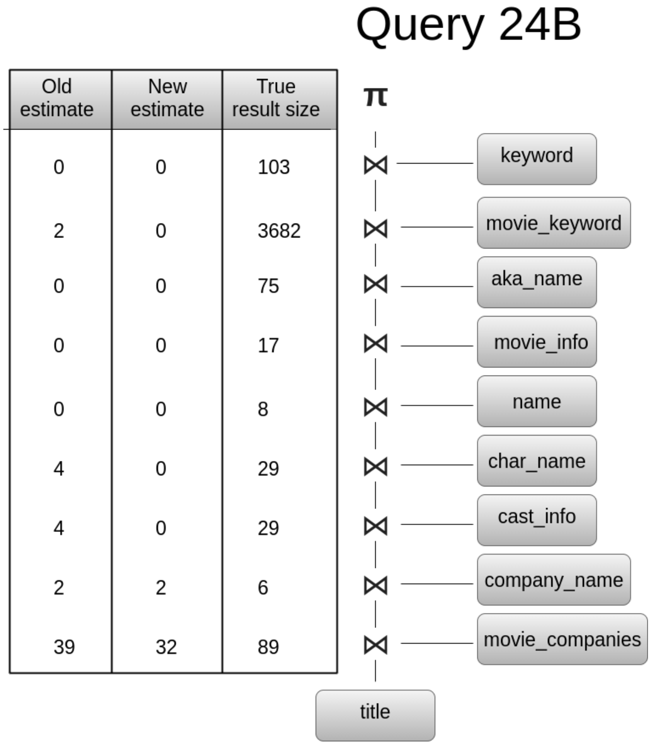


FIGURE 6.13: Old and new execution plan for query 4B of the imdb dataset. Result sizes are showed in the format 'Estimated (Actual)' beside each operator.

Query 24B

Figure 6.14 shows the execution plan for query 24B (found in Appendix A.4) in a compressed tree format. All filtering operations have been omitted due to the complex nature of the query. Since the plan returned from the optimizer was the same for both versions of the system, only a single execution tree is included. Even though the execution plans are the same, the old MySQL version executes the query almost twice as quickly as the new version. This observation makes it clear that execution times are unpredictable and must be used with care as a benchmark.

Query 24B contains only a single LIKE predicate that could change the execution plan chosen by the optimizer. For this predicate, the two system's estimated filtering effects only differ by about 1% (both having a relative error of about 3), which explains why the two execution plans are equal.



Old estimate	New estimate	True result size
0	0	103
2	0	3682
0	0	75
0	0	17
0	0	8
4	0	29
4	0	29
2	2	6
39	32	89

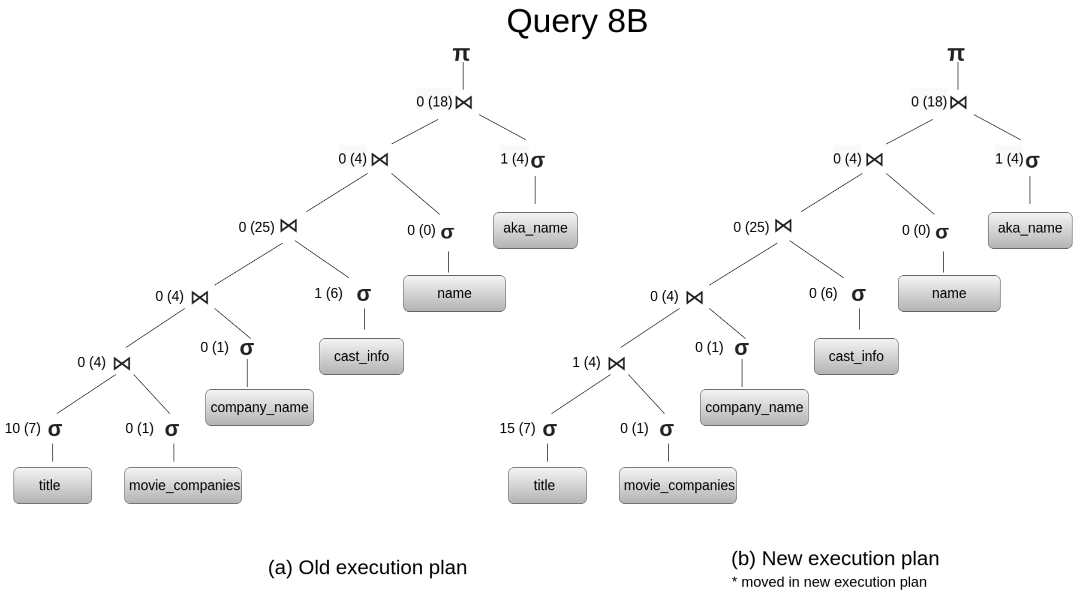
FIGURE 6.14: A compressed execution graph for query 24B. Due to the large nature of the query all filter operations has been omitted. Estimated result sizes and true result sizes are listed in the left table.

Query 8B

Figure 6.15 shows the old (figure a) and new (figure b) execution plan for query 8B (found in Appendix A.5) in a tree format. One can see that the join order is the same in both versions of the system. The fact that the execution times between them differ by a factor of two shows that the execution times that are observed throughout this section are decided by factors that cannot be controlled.

Regarding the selectivity estimates for query 8B, those are quite different between the two systems. In the old version of the system, the estimated filtering effect of the four LIKE predicates on the "movie_companies" table is overestimated by a factor of 10. With the new histogram approach, on the other hand, the four predicates yield an underestimation of a factor of about 500. For the predicates on the "name" column, both approaches give a relative error of about 10, but the new approach underestimates while the old approach overestimates.

Given the significant variations to the selectivity estimates, one would expect the query plan to change at least slightly. The old version of the optimizer overestimates the filtering effect of the "name" table and thus places this table towards the top of the execution tree. Changing this estimate to an underestimation would usually have the effect of moving the table down in the tree such that the result size can be reduced earlier in the execution. A likely explanation for not pushing the table down in this situation is that the result size is relatively small already. The small result sizes combined with the fact that filtering effects from the other tables are fetched from index statistics, and are thus more trustworthy, makes the risk greater than the reward for pushing the "name" table down the tree.



(a) Old execution plan

(b) New execution plan
* moved in new execution plan

FIGURE 6.15: Old and new execution plan for query 8B of the imdb dataset. Result sizes are showed in the format 'Estimated (Actual)' beside each operator.

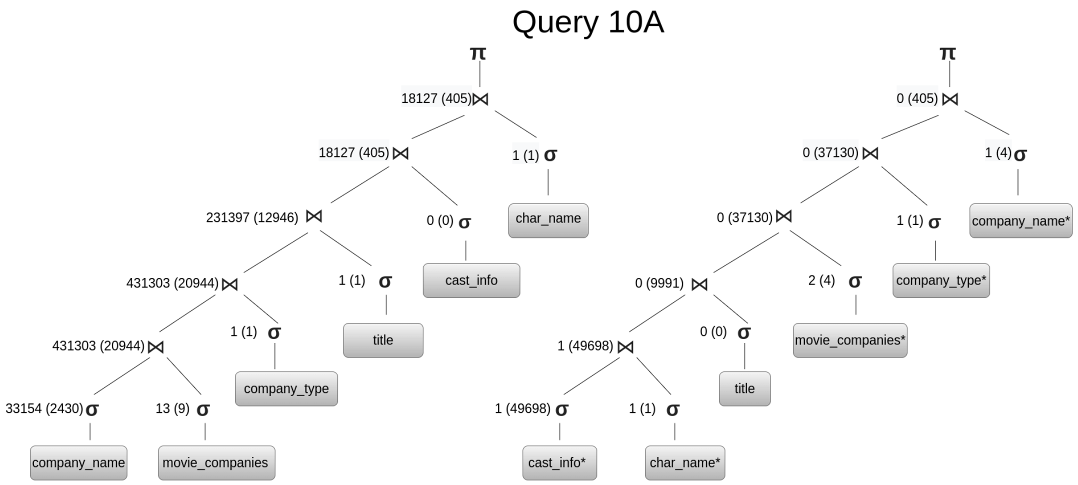
Query 10A

Figure 6.16 shows the old (figure a) and new (figure b) execution plan for query 10A (found in Appendix A.6) in a tree format. Both LIKE predicates in the query refers to the "note" column of the "cast_info" table. Still, almost all tables are shuffled around in the execution tree when the selectivity estimate for cast_info is moved. The new execution plan initially joins the "cast_info" table with "char_name", producing a temporary result almost three times larger than in the old execution tree. Towards the top of the tree, the temporary results of the new plan become much larger than in the old plan, which is the cause of the slower execution time in the new version of MySQL.

As one can see from the graph in Figure 6.16, the new query graph is changed due to a considerable underestimation for the cast_info table. This estimate makes the optimizer believe that the result size can be almost entirely filtered after a single join. By inspecting the contents of the histogram on the "note" column of "company_name", it becomes clear that the cause is a (very) unlucky sample made in the construction process. The 200 sampled rows were precisely equal, leaving the histogram with only a single bucket containing "(voice)". When this histogram was used to estimate the filtering factor of "note LIKE '%(uncredited)%'", the estimated result size was one row.

By further inspecting the "cast_info" table, it can be seen that only 2% of the tuples have "(voice)" in the note column. The probability of sampling only such rows is extremely low, and the reason this happens is a combination of two factors. Firstly, the contents of the IMDb dataset are not shuffled, meaning that rows that appear close together could be correlated in some way. Secondly, MySQL uses a block-level sampling approach, causing many consecutive rows to be sampled. Due to this sampling approach, many correlated samples are fetched, and for the "cast_info" column, this is problematic.

A new histogram was built with a different seed to observe how big of an impact the sampling seed has on this table. For the new histogram, the estimated result



(a) Old execution plan

(b) New execution plan
* moved in new execution plan

FIGURE 6.16: Old and new execution plan for query 10A of the imdb dataset. Result sizes are showed in the format 'Estimated (Actual)' beside each operator.

size was 6325 instead of 1. Although this is still far from the original estimates of MySQL, it is a significant improvement. This observation shows the enormous impact that the random seed has and that using block level sampling for the histogram construction is somewhat unstable.

6.3.2 Summary

Through the inspection of these six queries, many observations can be made. The following points will attempt to summarize the most important of these observations:

- The modified SPH approach has proved to be prone to severe underestimation. Even though the initial experiments showed good results, the tables and predicates present in the JOB queries have proven to be tougher.
- The MySQL block level sampling technique is not optimal when the data in the tables are ordered. In these cases, block sampling will not give a sample representative of the contents of the table. The effect of this sampling strategy becomes increasingly visible when the samples are as small as 200 tuples.
- The MySQL system relies heavily on selectivity estimates. Changing only a single selectivity estimate within a complex query such as those in the join order benchmark can completely shuffle the entire execution graph, and give large variations in execution time.
- Query 5C showed an example of a situation where good selectivity estimates produced worse plans than bad selectivity estimates. This observation is not uncommon in research regarding the query optimizer. It is not uncommon for "two wrongs to make a right" or even for "two rights to make a wrong" in such a complex component. Another example of this was observed in Leis et al., 2015 where supplying the PostgreSQL optimizer with correct distinct

counts for all columns resulted in larger estimation errors than estimating the number of distinct values.

- Execution times within a complex database system are not reliable to measure the performance of queries. Two of the six queries examined through this section proved to use the same execution plan despite having very varying execution times. Moving along on this observation, one can assume that many of the queries that were executed quicker in the new system did so despite using the same execution plan.
- Among the 73 queries executed through this section, 35 queries had a reduction to the execution time of a factor of at least 1.5. Even though execution times have shown to be an imperfect performance indicator, this is a significant result indicating that the MySQL selectivity estimates can be improved without very high risk, and give a great increase to query performance.

Chapter 7

Conclusions and future work

7.1 Conclusions

The main goal of this thesis has been improving the selectivity estimates for the LIKE operator in the MySQL system. This section will repeat the three research questions that were stated in Chapter 1 and give answers to these questions based on the observations made throughout the thesis. Finally, the conclusions of all three research questions will be summarized.

1. **What research have previously been done on the subject of result size estimation for the LIKE operator containing wildcards, and which of these approaches are sensible to implement into the MySQL system?**

Chapter 3 started with an overview of approaches used to estimate the result sizes of approximate string matching. Approximate string matching is a generalization of the LIKE operator that has been more widely researched than the LIKE operator. In this general problem, any similarity measure can be used to match similar strings. The LIKE operator differs slightly from this problem due to the complex nature of its similarity measure. For many approaches to selectivity estimation of approximate string matching, the LIKE operator cannot be applied since the LIKE similarity of two strings is binary. They either match, or they do not.

Among the approaches that have been proposed specifically for the LIKE operator, there are three main types. The first proposed approach was the suffix tree, in which the database builds a tree that is continuously pruned to the wanted size. The next type of selectivity estimation for LIKE predicates was the SEPIA approach. This approach builds on the idea that a long string cannot have higher selectivity than any of its substrings. Further, such a long string usually has a substring with similar selectivity. The most recent approach, however, is the Sequential Pattern-based Histograms (SPH) introduced in Aytimur and Çakmak, 2018. SPH mines frequent itemsets before building a histogram from those itemsets that can further be used to generate selectivity estimates.

There are multiple considerations to make when picking an approach to implement into an existing system. Chapter 2 showed that the proposed approaches for selectivity estimation of approximate string queries are not relevant to the LIKE operator as they are tough to extend to support the LIKE operator and offer no benefits when it comes to accuracy. Further, the approach using short identifying substrings needs a large amount of storage for the q-gram table. Using this approach would thus increase the space complexity of MySQL, which is undesirable. The suffix tree approach and the Sequential Pattern-based Histogram are both excellent candidates for the MySQL system. However, the SPH approach is more similar to the current solution of MySQL, giving it a slight edge against the suffix tree.

2. **How is the accuracy and performance of the method from question one compared to the current approach of the MySQL and PostgreSQL systems?**

Chapter 5 presented a set of experiments that were bound to give some answers to how good the SPH approach is. These experiments were then conducted through Chapter 6. Firstly, the effect of sampling size on the SPH algorithm was explored. The accuracy of the estimates provided by the histograms increased drastically between 50 and 200 samples for all three query workloads. The estimation error then flattened out for larger sample sizes, showing that 200 samples are enough for accurate selectivity estimates.

The comparison of these results to the current approaches of MySQL and PostgreSQL gave varying results. The modified SPH algorithm was outperformed by PostgreSQL for predicates with a high number of wildcards while beating PostgreSQL for longer predicates with fewer wildcards. However, compared to the current MySQL system, the new approach is superior for all three query workloads.

Providing accurate selectivity estimates is, however, not enough for an approach to be used in a real-world DBMS. It is also essential to have a low time and space complexity such that estimates can be computed ad-hoc. Regarding space complexity, the new approach reuses the MySQL histograms and even stores one value less for each bucket. Thus, the space complexity should not be an issue. When it comes to construction time, the current version of the MySQL histogram was found to have a construction time of around 2 seconds. Chapter 6 showed that the SPH histogram built on 200 samples had an average construction time of 1.33 seconds and is thus well within the limit. However, some considerations are to be made when the sample size and threshold for the SPH algorithm are decided. For all sample sizes over 100 samples, it is advised to have a threshold of at least 1.5% to avoid large spikes in computation time.

3. **How will an improved selectivity estimate for the LIKE operator influence the optimizer's decisions for complex queries?**

The final experiment of Chapter 6 showed the results of running 73 complex queries from the Join Order Benchmark. 60 of these 73 queries showed an improvement to execution time, and some queries had a relative improvement of more than 10. On the other side, a slower execution time was seen for 12 queries, and one of those queries had a significantly slower execution time. The total execution time across all queries was, however, seen to be reduced by 26%. The main takeaway from these results is the significant impact of a single selectivity estimate on a complex query with multiple joins.

On closer inspection of some of the most extreme changes in execution time, it was seen that the execution plans were drastically changed for some of the queries. Among the queries that had the best improvement to execution time, some changes to the join order had reduced the temporary result sizes and thus reduced the execution time. For two of the queries, this change was the result of an improved selectivity estimate. However, for the third query, the selectivity estimate was worse in the new version of MySQL, but due to a lucky join correlation, an improved execution time was seen nonetheless.

Among the bottom three queries, only one query had seen a change to the execution plan. From this result, it can be seen that execution time is not a perfect indicator of the success of an execution plan since it can vary quite a bit from factors that are hard to control.

As for the query that slowed down with a factor of 6, it did point out a significant weakness with the modified SPH approach in MySQL. Due to the MySQL block-level sampling, the sample taken was correlated and did not in any way represent the underlying data distribution. This weakness is something that could make the SPH approach unfitted for the MySQL system.

Summary

This thesis found the SPH approach proposed in Aytimur and Çakmak, 2018 to be the most promising approach to implement for selectivity estimation of the LIKE operator in the MySQL system. Some experiments on a lone-standing version of the algorithm found the time and space complexity acceptable even with a low sampling rate. Further, the algorithm showed comparable accuracy to the PostgreSQL solution and superior accuracy to the current MySQL solution for LIKE predicates with leading wildcards.

Finally, a slightly modified version of the SPH approach was implemented into the MySQL system and some complex queries from the Join Order Benchmark were run using this implementation. The queries were seen to have a 26% speed-up compared to the old version of MySQL as the optimizer could find a better execution plan. Despite this improvement to execution time, the SPH approach was seen to be fragile to the MySQL sampling schema, causing significant estimation errors for specific queries. Due to these observations, the proposed approach should be used with care and is not recommended to be used alongside a block-level sampling schema.

7.2 Future work

This thesis has answered some questions related to selectivity estimation of the LIKE operator and to the MySQL system specifically. It has, however, left behind a few new unanswered questions. The following points make some suggestions to research that can be done to answer some of these questions, and to a smaller or bigger degree, extend the topics discussed in this thesis:

- **Use tuple-level sampling** for the SPH approach. Since the MySQL system is not made for tuple-level sampling, this experiment will likely worsen construction times. It would, however, be interesting to see the quality of execution plans generated by such an approach. Further, it would be interesting to see if this could be successfully implemented in a system with better support for tuple-level sampling.

- **Supply perfect selectivity estimates** to see the effect that this has on execution plans. The generated plans can then be compared to the plans generated in this thesis to see whether the selectivity estimates are the limiting factor of the MySQL system. This experiment has already been performed for other systems in Leis et al., 2015.

- **Employ other approaches to selectivity estimation of LIKE predicates.** There are currently very few approaches available that fit into a real-world system. It is, however, likely that more approaches will be available in the future. Selectivity estimation of numeric data is still, after 40 years, seeing new approaches being proposed, and the field of machine learning has just recently been introduced. When more approaches are available for the LIKE operator in the future, these approaches should be evaluated within the MySQL system since there is undoubtedly much to be gained from improving selectivity estimates.

- **Using the extensions to the SPH approach.** Through this thesis, only the first proposed version of the SPH approach has been used. Including positional patterns and pruning based on Information Content to the implementation would impose an additional overhead but could boost estimation accuracy. Since these statistics are built offline, this additional overhead could be worth it, given that the accuracy boost is significant.

Bibliography

Astrahan, M. M. et al. (June 1976). "System R: Relational Approach to Database Management". In: *ACM Trans. Database Syst.* 1.2, 97–137. ISSN: 0362-5915. DOI: 10.1145/320455.320457. URL: https://doi.org/10.1145/320455.320457.

Aytimur, Mehmet and Çakmak, Ali (2018). "Estimating the selectivity of LIKE queries using pattern-based histograms". In: *Turkish Journal of Electrical Engineering & Computer Sciences* 26.6, pp. 3319–3334.

Aytimur, Mehmet and Cakmak, Ali (2020). "Using Positional Sequence Patterns to Estimate the Selectivity of SQL LIKE Queries". In: *arXiv preprint arXiv:2002.01164*.

Chaudhuri, Surajit, Ganti, Venkatesh, and Gravano, Luis (2004). "Selectivity estimation for string predicates: Overcoming the underestimation problem". In: *Proceedings. 20th International Conference on Data Engineering*. IEEE, pp. 227–238.

Chen, Yu and Yi, Ke (2017). "Two-Level Sampling for Join Size Estimation". In: *Sigmod '17*, pp. 759–774. URL: https://dl.acm.org/doi/epdf/10.1145/3035918.3035921.

Estan, Cristian and Naughton, Jeffrey F. (2006). "End-biased Samples for Join Cardinality Estimation". In: pp. 20–20.

Gunopulos, Dimitrios et al. (2000). "Approximating multi-dimensional aggregate range queries over real attributes". In: *Acm Sigmod Record* 29.2, pp. 463–474.

Hellerstein, Joseph M, Stonebraker, Michael, and Hamilton, James (2007). *Architecture of a database system*. Now Publishers Inc.

Holst, Arne (2021). *Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2024*. URL: https://www.statista.com/statistics/871513/worldwide-data-created/.

Jagadish, HV, Ng, Raymond T, and Srivastava, Divesh (1999). "Substring selectivity estimation". In: *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pp. 249–260.

Jin, Liang and Li, Chen (2005). "Selectivity estimation for fuzzy string predicates in large data sets". In: *Proceedings of the 31st international conference on Very large data bases*, pp. 397–408.

Kooi, Robert Philip (1981). "The Optimization of Queries in Relational Databases." In:

Krishnan, P, Vitter, Jeffrey Scott, and Iyer, Bala (1996). "Estimating alphanumeric selectivity in the presence of wildcards". In: *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pp. 282–293.

Lee, Hongrae, Ng, Raymond T, and Shim, Kyuseok (2009). "Approximate substring selectivity estimation". In: *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, pp. 827–838.

Leis, Viktor et al. (2015). "How good are query optimizers, really?" In: *Proceedings of the VLDB Endowment* 9.3, pp. 204–215.

Leis, Viktor et al. (2018). "Query optimization through the looking glass, and what we found running the Join Order Benchmark". In: *The VLDB Journal* 27.5, pp. 643–668.

Mazeika, Arturas et al. (2007). "Estimating the selectivity of approximate string queries". In: *ACM Transactions on Database Systems (TODS)* 32.2, 12–es.

Muralikrishna, M and DeWitt, David J (1988). "Equi-depth multidimensional histograms". In: *Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, pp. 28–36.

Piatetsky-Shapiro, Gregory and Connell, Charles (1984). "Accurate estimation of the number of tuples satisfying a condition". In: *ACM Sigmod Record* 14.2, pp. 256–276.

Sæther, J (2020). *An overview of methods in cardinality estimation, and running the Join Order Benchmark in MySQL*.

Thompson, Carl (1999). "If you could just provide me with a sample: examining sampling in qualitative and quantitative research papers". In: *Evidence-Based Nursing* 2.3, pp. 68–70.

Wang, Jianyong and Han, Jiawei (2004). "BIDE: Efficient mining of frequent closed sequences". In: *Proceedings. 20th international conference on data engineering*. IEEE, pp. 79–90.

Wang, TaiNing and Chan, Chee-Yong (2020). "Improved Correlated Sampling for Join Size Estimation". In: pp. 325–336.

Appendix A

Selected JOB queries

A.1 Query 13B

```
SELECT MIN(cn.name) AS producing_company,
       MIN(miidx.info) AS rating,
       MIN(t.title) AS movie_about_winning
FROM company_name AS cn,
     company_type AS ct,
     info_type AS it,
     info_type AS it2,
     kind_type AS kt,
     movie_companies AS mc,
     movie_info AS mi,
     movie_info_idx AS miidx,
     title AS t
WHERE cn.country_code ='[us]'
  AND ct.kind ='production companies'
  AND it.info ='rating'
  AND it2.info ='release dates'
  AND kt.kind ='movie'
  AND t.title != ''
  AND (t.title LIKE '%Champion%'
       OR t.title LIKE '%Loser%')
  AND mi.movie_id = t.id
  AND it2.id = mi.info_type_id
  AND kt.id = t.kind_id
  AND mc.movie_id = t.id
  AND cn.id = mc.company_id
  AND ct.id = mc.company_type_id
  AND miidx.movie_id = t.id
  AND it.id = miidx.info_type_id
  AND mi.movie_id = miidx.movie_id
  AND mi.movie_id = mc.movie_id
  AND miidx.movie_id = mc.movie_id;
```

A.2 Query 5C

```
SELECT MIN(t.title) AS american_movie
FROM company_type AS ct,
     info_type AS it,
     movie_companies AS mc,
     movie_info AS mi,
     title AS t
WHERE ct.kind = 'production companies'
  AND mc.note NOT LIKE '%(TV)%'
  AND mc.note LIKE '%(USA)%'
  AND mi.info IN ('Sweden',
                'Norway',
                'Germany',
                'Denmark',
                'Swedish',
                'Denish',
                'Norwegian',
                'German',
                'USA',
                'American')
  AND t.production_year > 1990
  AND t.id = mi.movie_id
  AND t.id = mc.movie_id
  AND mc.movie_id = mi.movie_id
  AND ct.id = mc.company_type_id
  AND it.id = mi.info_type_id;
```

A.3 Query 4B

```
SELECT MIN(mi_idx.info) AS rating,
      MIN(t.title) AS movie_title
FROM info_type AS it,
     keyword AS k,
     movie_info_idx AS mi_idx,
     movie_keyword AS mk,
     title AS t
WHERE it.info ='rating'
  AND k.keyword LIKE '%sequel%'
  AND mi_idx.info > '9.0'
  AND t.production_year > 2010
  AND t.id = mi_idx.movie_id
  AND t.id = mk.movie_id
  AND mk.movie_id = mi_idx.movie_id
  AND k.id = mk.keyword_id
  AND it.id = mi_idx.info_type_id;
```

A.4 Query 24B

```
SELECT MIN(chn.name) AS voiced_char_name,
      MIN(n.name) AS voicing_actress_name,
      MIN(t.title) AS kung_fu_panda
FROM aka_name AS an,
     char_name AS chn,
     cast_info AS ci,
     company_name AS cn,
     info_type AS it,
     keyword AS k,
     movie_companies AS mc,
     movie_info AS mi,
     movie_keyword AS mk,
     name AS n,
     role_type AS rt,
     title AS t
WHERE ci.note IN ('(voice)',
                '(voice: Japanese version)',
                '(voice) (uncredited)',
                '(voice: English version)')
  AND cn.country_code ='[us]'
  AND cn.name = 'DreamWorks Animation'
  AND it.info = 'release dates'
  AND k.keyword IN ('hero',
                  'martial-arts',
                  'hand-to-hand-combat',
                  'computer-animated-movie')
  AND mi.info IS NOT NULL
  AND (mi.info LIKE 'Japan:%201%'
      OR mi.info LIKE 'USA:%201%')
  AND n.gender ='f'
  AND n.name LIKE '%An%'
  AND rt.role ='actress'
  AND t.production_year > 2010
  AND t.title LIKE 'Kung Fu Panda%'
  AND t.id = mi.movie_id
  AND t.id = mc.movie_id
  AND t.id = ci.movie_id
  AND t.id = mk.movie_id
  AND mc.movie_id = ci.movie_id
  AND mc.movie_id = mi.movie_id
  AND mc.movie_id = mk.movie_id
  AND mi.movie_id = ci.movie_id
  AND mi.movie_id = mk.movie_id
  AND ci.movie_id = mk.movie_id
  AND cn.id = mc.company_id
  AND it.id = mi.info_type_id
  AND n.id = ci.person_id
  AND rt.id = ci.role_id
  AND n.id = an.person_id
  AND ci.person_id = an.person_id
  AND chn.id = ci.person_role_id
  AND k.id = mk.keyword_id;
```

A.5 Query 8B

```
SELECT MIN(an.name) AS acress_pseudonym,
      MIN(t.title) AS japanese_anime_movie
FROM aka_name AS an,
     cast_info AS ci,
     company_name AS cn,
     movie_companies AS mc,
     name AS n,
     role_type AS rt,
     title AS t
WHERE ci.note ='(voice: English version)'
  AND cn.country_code ='[jp]'
  AND mc.note LIKE '%(Japan)%'
  AND mc.note NOT LIKE '%(USA)%'
  AND (mc.note LIKE '%(2006)%'
      OR mc.note LIKE '%(2007)%')
  AND n.name LIKE '%Yo%'
  AND n.name NOT LIKE '%Yu%'
  AND rt.role ='actress'
  AND t.production_year BETWEEN 2006 AND 2007
  AND (t.title LIKE 'One Piece%'
      OR t.title LIKE 'Dragon Ball Z%')
  AND an.person_id = n.id
  AND n.id = ci.person_id
  AND ci.movie_id = t.id
  AND t.id = mc.movie_id
  AND mc.company_id = cn.id
  AND ci.role_id = rt.id
  AND an.person_id = ci.person_id
  AND ci.movie_id = mc.movie_id;
```

A.6 Query 10A

```
SELECT MIN(chn.name) AS uncredited_voiced_character,
      MIN(t.title) AS russian_movie
FROM char_name AS chn,
     cast_info AS ci,
     company_name AS cn,
     company_type AS ct,
     movie_companies AS mc,
     role_type AS rt,
     title AS t
WHERE ci.note LIKE '%(voice)%'
  AND ci.note LIKE '%(uncredited)%'
  AND cn.country_code = '[ru]'
  AND rt.role = 'actor'
  AND t.production_year > 2005
  AND t.id = mc.movie_id
  AND t.id = ci.movie_id
  AND ci.movie_id = mc.movie_id
  AND chn.id = ci.person_role_id
  AND rt.id = ci.role_id
  AND cn.id = mc.company_id
  AND ct.id = mc.company_type_id;
```

Appendix B

Source code

This appendix only contains the important parts of the source code from the C++ implementation. For a complete view of the Python and C++ source code, including random seeds for reproduction, the attached zip folder can be visited.

B.1 BIDE algorithm

```
static bool islocalclosed(char previtem,
                          std::vector<std::tuple<int, int>> &
                              matches,
                          std::vector<std::vector<char>> db) {
  std::set<char> closeditems;
  int k = 0;
  std::vector<std::tuple<int, int>>::iterator it;
  for (it = matches.begin(); it != matches.end(); it++) {
    std::set<char> localitems;
    int i = std::get<0>(*it);
    int endpos = std::get<1>(*it);
    for (int startpos = endpos - 1; startpos >= 0; startpos--) {
      char item = db[i][startpos];
      if (item == previtem) {
        *it = std::tuple<int, int>(i, startpos);
        break;
      }
      localitems.insert(item);
    }
    if (k == 0) {
      closeditems.insert(localitems.begin(), localitems.end());
    } else {
      std::vector<char> newcloseditems;
      std::set_intersection(closeditems.begin(),
        closeditems.end(), localitems.begin(),
        localitems.end(), std::back_inserter(newcloseditems));
      closeditems.clear();
      std::copy(newcloseditems.begin(), newcloseditems.end(),
                std::inserter(closeditems, closeditems.end()));
    }
    k++;
  }
  return closeditems.size() > 0;
```

```
}


static bool canclosedprune(std::vector<std::vector<char>> &db,
                           std::vector<char> patt,
                           std::vector<std::tuple<int, int>>
                               matches) {
  std::vector<char> newpatt;
  newpatt.push_back('\0');
  std::copy(patt.begin(), patt.end(),
    std::back_inserter(newpatt));
  return reversescan(db, newpatt, matches, false);
}

static bool isclosed(std::vector<std::vector<char>> &db, std::
    vector<char> patt,
                     std::vector<std::tuple<int, int>> matches) {

  // Creating the matches input to the reversescan method.
  std::vector<std::tuple<int, int>> newmatches;
  std::vector<std::tuple<int, int>>::iterator it;
  for (it = matches.begin(); it != matches.end(); it++) {
    int i = std::get<0>(*it);
    newmatches.push_back(std::tuple<int, int>(i, db[i].size()));
  }

  // Creating the patt input to the reversescan method.
  std::vector<char> newpatt;
  newpatt.push_back('\0');
  std::copy(patt.begin(), patt.end(),
    std::back_inserter(newpatt));

  // Pushing pack a dummy char at the end. This is an
  // alternative to pushing back a NULL.
  newpatt.push_back('\0');

  return reversescan(db, newpatt, newmatches, true);
}


static void invertedindex(
    std::vector<std::vector<char>> seqs,
    std::vector<std::tuple<int, int>> entries,
    std::map<char, std::vector<std::tuple<int, int>>> &index) {
  int k = 0;
  std::vector<std::vector<char>>::iterator it;
  for (it = seqs.begin(); it != seqs.end(); it++) {
    int i, lastpos;
    std::vector<char> seq = *it;
    if (entries.size() > 0) {
      std::tuple<int, int> currtuple = entries[k];
      i = std::get<0>(currtuple);
```

```
        lastpos = std::get<1>(currtuple);
      } else {
        i = k;
        lastpos = -1;
      }

      int p = lastpos + 1;
      std::vector<char>::iterator it2;
      for (it2 = seq.begin(); it2 != seq.end(); it2++) {
        char item = *it2;
        std::vector<std::tuple<int, int>> l = index[item];
        if (l.size() > 0 && std::get<0>(*--l.end()) == i) {
          p++;
          continue;
        }

        l.push_back(std::tuple<int, int>(i, p));
        index[item] = l;
        p++;
      }
      k++;
    }
  }


  static void nextentries(
      std::vector<std::vector<char>> &data,
      std::vector<std::tuple<int, int>> entries,
      std::map<char, std::vector<std::tuple<int, int>>> &index) {
    std::vector<std::vector<char>> newdata;
    std::vector<std::tuple<int, int>>::iterator it;
    for (it = entries.begin(); it != entries.end(); it++) {
      int i = std::get<0>(*it);
      int lastpos = std::get<1>(*it);
      std::vector<char>::iterator data_it;
      std::vector<char> new_entry;
      std::copy(data[i].begin() + lastpos + 1, data[i].end(),
                std::back_inserter(new_entry));
      newdata.push_back(new_entry);
    }
    invertedindex(newdata, entries, index);
  }


  // The recursive BIDE algorithm. The following
  // inputs are necessary:
  // patt: The pattern that is currently being examined.
  // matches: The places in the database that this
  // pattern is observed.
  // result: The patterns that has so far been reported to be
  // closed and frequent. Patterns that are already recorded
  // can later be removed from this list.
```

```
// db: All strings that has been sampled.
static bool bide_frequent_rec(
    std::vector<char> patt,
    std::vector<std::tuple<int, int>> matches,
    std::vector<std::tuple<std::vector<char>, int>> &result,
    std::vector<std::vector<char>> &db) {
  int MINSUP = 3;
  long unsigned int MINLEN = 2;
  long unsigned int MAXLEN = 10;

  int sup = matches.size();
  // If pattern's length is greater than minimum length,
  // consider whether it should be recorded.
  if (patt.size() >= MINLEN) {
    // If pattern's support < minsup, stop.
    if (sup < MINSUP) {
      return false;
    }

    // if pattern is closed (backward extension check),
    // record the pattern and its support.
    if (isclosed(db, patt, matches)) {
      result.push_back(std::tuple<std::vector<char>, int>(patt,
          sup));
    }
  }

  // If pattern's length is greater than maximum length,
  // stop recursion.
  if (patt.size() == MAXLEN) {
    return false;
  }

  std::map<char, std::vector<std::tuple<int, int>>> occurs;

  // Find the following items
  nextentries(db, matches, occurs);

  std::map<char, std::vector<std::tuple<int, int>>>::iterator it
      ;
  for (it = occurs.begin(); it != occurs.end(); it++) {
    char newitem = it->first;
    std::vector<std::tuple<int, int>> newmatches = it->second;

    // Set the new pattern
    std::vector<char> newpatt;
    std::copy(patt.begin(), patt.end(),
        std::back_inserter(newpatt));
    newpatt.push_back(newitem);

    // forward closed pattern checking
    if (matches.size() == newmatches.size()) {
      std::tuple<std::vector<char>, int> tuple_to_find =
         std::tuple<std::vector<char>, int>(patt, sup);
      result.erase(
         std::remove(result.begin(), result.end(), tuple_to_find)
             ,
                   result.end());
    }

    // Can we stop pruning the new pattern?
    if (canclosedprune(db, newpatt, newmatches)) {
      continue;
    }
    bide_frequent_rec(newpatt, newmatches, result, db);
  }

  return false;
}
```

B.2 Histogram construction

```
float NUMBER_OF_BUCKETS = 1024.0;
int total_freq = 0;
std::vector<std::tuple<std::vector<char>, int>>::iterator it2;
for (it2 = result.begin(); it2 != result.end(); it2++) {
  total_freq += std::get<1>(*it2);
}

int bucket_size = ceil(total_freq / NUMBER_OF_BUCKETS);
int count = 0;
int bucket_number = 0;
const CHARSET_INFO *charset_info = &my_charset_latin1;
for (it2 = result.begin(); it2 != result.end(); it2++) {
    count += std::get<1>(*it2);
    if (count > bucket_number * bucket_size &&
        bucket_number < NUMBER_OF_BUCKETS) {
        bucket_number++;
        std::vector<char> curr = std::get<0>(*it2);
        if (value_map->add_values(
          String(&(curr[0]), curr.size(), charset_info),
          std::get<1>(*it2))) {
            return true;
        }
    }
}
```

B.3 Selectivity estimation

```
int find_match(std::vector<char> predicate, std::vector<char>
    bucket_endpoint) {
  bool PREDICATE_IS_LONGER = predicate.size() >= bucket_endpoint
      .size();

  // If the predicate is longer than the bucket endpoint,
  // we won't find a match.
  if (PREDICATE_IS_LONGER) {
    return 0;
  }
  long unsigned int matched_letters = 0;
  for (long unsigned int i = 0; i < bucket_endpoint.size(); i++)
      {
    for (long unsigned int j = 0; j < predicate.size(); j++) {
      if (i + j > bucket_endpoint.size()) {
        continue;
      }
      if (predicate[j] == bucket_endpoint[i + j]) {
        matched_letters++;
      }
    }
    // The predicate is found within the endpoint value.
    // Exact match.
    if (matched_letters == predicate.size()) {
      return 2;
    }
    matched_letters = 0;
  }

  matched_letters = 0;
  long unsigned int i = 0;
  for (char letter : predicate) {
    for (; i < bucket_endpoint.size(); i++) {
      if (letter == bucket_endpoint[i]) {
        matched_letters++;
        break;
      }
    }
  }
  // All letters are found in the correct order. Partial match.
  if (matched_letters == predicate.size()) {
    return 1;
  }
  return 0;
}
```

```
double Equi_height<String>::get_individual_selectivity(
    std::vector<char> &predicate) const {
  std::vector<double> partial_matches;
  std::vector<double> exact_matches;
  for (auto it = m_buckets.begin(); it != m_buckets.end(); ++it)
      {
    auto bucket = *it;
    String endpoint = bucket.get_lower_inclusive();
    size_t length = endpoint.length();
    int freq = bucket.get_num_distinct();
    std::vector<char> localstr;
    for (size_t i = 0; i < length; i++) {

      // Since the MySQL LIKE operator is case insensitive,
      // we only deal with lower case.
      localstr.push_back(tolower(endpoint[i]));
    }

    // Get exact, partial or no match.
    int match = find_match(predicate, localstr);

    // Keeping this fixed according to the
    // experimental results.
    double sample_size = 200;
    // Exact match.
    if (match == 2) {
      double new_sel = (double)freq / sample_size;
      exact_matches.push_back(new_sel);
    }

    // Keeping the partial match only
    // if no exact matches are found yet.
    if (match == 1 && exact_matches.size() == 0) {
      partial_matches.push_back((double)freq / sample_size);
    }
  }
  // The paper has experimentally evaluated that returning 10%
  // of the minimum support threshold is a good solution
  // whenever no matches are found. For our application
  // this would be 10% of 3/200, or 3/2000.
  double return_value =
       (double) 3 / (double) (sample_size * 10);

  // Exact matches have first priority.
  if (exact_matches.size() > 0) {
    auto it = max_element
       (std::begin(exact_matches), std::end(exact_matches));
    return *it;
  }
  if (partial_matches.size() > 0) {
    double sum = 0;
    std::for_each(partial_matches.begin(), partial_matches.end()
        ,
                  [&](double n) { sum += n; });
    return_value = sum / (double)partial_matches.size();
  }

  return return_value;
}
```