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

Sevre Vestrheim

Automatic updating of histograms in MySQL

Master's thesis in Computer Science
Supervisor: Jon Olav Hauglid
December 2019

Sevre Vestrheim

Automatic updating of histograms in MySQL

Master’s thesis in Computer Science
Supervisor: Jon Olav Hauglid
December 2019

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

NTNU
Norwegian University of
Science and Technology

Abstract

When the database query-optimiser optimises a query, several factors are taken into consideration while attempting to decide on the query plan to use. One of these factors is the order in which tables are joined. When determining the order in which tables should be joined, the cardinality of each table is one of the essential characteristics investigated. If any predicates can be applied to tables before join operations begin that will reduce the execution time, and is as such almost always performed. The selectivity of a predicate is determined by how many rows in the table which satisfy the predicate. It is approximated either by using histograms or heuristics. Heuristics are constant values chosen by the developers of the query-optimiser, and these will seldom provide as accurate estimates as a histogram. However, a histogram has to be computed, which takes time and resources. If the data on which the histogram is based changes without the histogram being updated, the histogram will become stale and will not provide accurate selectivity estimates.

An evaluation of the effects of varying histogram accuracy levels and different strategies of maintaining accuracy is presented in this report. Along with designing the requirements for, and implementing, a use-case in which different levels of histogram accuracy affect query execution times. Initially, nine different updating methods are presented, of which four are implemented. later another four are suggested as a basis for future work. Through testing of the implemented updating methods and a set of base class methods, it is shown that in the given use-case stale histograms perform worse than accurate histograms, and also worse than no histograms. It is also shown that with relatively simple updating schemes performance can be significantly improved when compared to using stale histograms. Further, it is shown that the implemented updating scheme in two state-of-the-art systems performs very poorly in our use-case.

Sammendrag

Nr en sprring blir optimalisert av sprreoptimalisatoren i en database er det mange faktorer som underskes fr valg av sprreplan blir tatt. En av disse er rekkeflgen tabeller joines i. Rekkeflgen bestemmes i stor grad av kardinaliteten til tabllen. Dersom noen av predikatene kan benyttes til filtere vekk rader i tabellene fr join operasjonen begynner, vil det redusere kjretiden. Dermed blir dette nesten alltid gjort. Selektiviteten til predikatene blir bestemt av antall rader i tabellen som tilfredstiller predikatet. Dette approksimeres enten ved hjelp av et histogram eller ved bruk av heuristikker. Heuristikker er kontstante verdier som blir bestemt av de som utvikler sprreoptimalisatoren. Disse gir sjelden et like nyaktig estimat som et histogram. Det er dog slik at et histogram m beregnes, noe som tar opp tid og ressurser. Dersom dataen histogrammet er basert p endres uten at histogramet beregnes p nytt, vil histogrammet vre feil og gi et unyaktig selektivitets estimat.

Effekten varierende histogram nyaktighet og forskjellige oppdaterings startegier for histogram har p kjretiden til sprringer evalueres i denne rapporten. Sammen med design av kravene til ,og implementeringen av, en database hvor histogram nyaktighet har en effekt p kjrtiden til sprringer. Til begynne med presenteres ni forskjellige oppdateringsmetoder hvorav fire implementeres. Senere presenteres fire nye som basis for fremtidig arbeid. Gjennom testing av de implementerte oppdaterings metodene og en gruppe med base metoder, vises det at i den gitte databasen yter unyaktige histogram drligere enn nyaktige histogram, og ogs verre enn ingen histogram. Det vises videre at gjennom relativt enkle oppdaterings metoder kan ytelsen til databasen kes betraktelig sammenlignet med unyaktige histogram. Det vises og at oppdaterings metodene som benyttes i to moderne system yter drlig i vr database.

Contents

List of Figures

Chapter 1

Introduction

An SQL query usually has numerous distinct execution plans that all yield identical results but with varying execution times. There is no known way to compute which of these plans is the quickest; the only option is to evaluate all and then choose one, this is the job of the query-optimiser. How does the optimiser choose a plan without spending possibly hours trying every one? The answer is it does not, it cheats. Firstly the optimiser chooses what it *believes* to be the quickest/optimal plan, it gives no guarantee that it *is* the optimal plan, it also does not check every single plan, it only tests a subset of the plan space. Secondly, it employs one of its most powerful tools, the databases statistics. The optimiser approximates the cost of each plan using statistics and a cost function, once the selected subset of plans is evaluated, the cheapest plan is chosen.

Database statistics is an umbrella term; there are several different types of statistics ranging from simple maximum and minimum values of columns to histograms approximating the distribution of values within a specified column. It can be argued that along with the total number of rows, the most useful statistic for the optimiser is, in fact, the histogram. It approximates the distribution of values within one or more columns and is used by the optimiser to approximate the result size, *cardinality*, from different parts of a query. The ratio between the approximate return number and the maximum possible amount of returned rows is known as the *selectivity* of a predicate. The optimiser uses the selectivity measurement to predict the cost of the different plans better and choose accordingly.

In the widely used MySQL database system, the query-optimiser computes the selectivity of predicates using histograms if the predicate conforms to certain rules. When a histogram is created, it can be an arbitrarily accurate approximation of the distribution, meaning that it can be as accurate as we want it to be, with some restrictions such as size and the time allowed to spend computing the histogram. However, as the data changes, the histogram may no longer be accurate; it has gone *stale*. When a histogram goes stale that *should* mean that the query-optimiser no longer creates an accurate cardinality estimate since it is basing its estimate on inaccurate information. This *can* in-turn result in bad execution plans being chosen. However, it is not known *if* this is the case, neither how *much* better the execution plan would have been if the histogram used was very accurate.

In other words, it is not known to which degree the approximation accuracy of the histogram influences query execution times. Since MySQL's implementation of histograms currently does not support any automatic updating, it is an excellent candidate for testing how the approximation accuracy effects the query-optimiser and query execution times of a modern RDBMS.

To study how the accuracy of a histogram effects query execution, this report will be looking at a case where continuous distribution change is frequent, namely a time-series dataset. Time-series data is the term given to data which has a connection to a specific point in time. Examples of such data are weather and temperature recordings, bank transactions and financial records. Typical for this data type is that new rows are appended to the database at a steady and high rate. As the value of incoming rows changes, so does the distribution of the data as a whole. Queries on these types of datasets typically restrict the result to a particular period in the data, usually querying data from the last 24-hours, week or month. This enables the rows that are being queried to have a shifting distribution. In this project, a fictional time-series dataset with fictional queries is used, due to our need of control over attributes of both the dataset and workload. While some time-series datasets are stored in specially designed time-series databases such as Amazon's Timestream [1], and Influxdata's InfluxDB [2] due to their need for high write throughput and fast query response time. Many administrators of regular RDBMS' can find themselves needing to store and query time-series data at some point during the life cycle of their database system. The use of time-series datasets is almost always as a part of a larger data model stored in a DBMS. It is therefore not unreasonable to expect MySQL to be able to handle this kind of workload.

The goal of this project is to; define attributes that have to be met for database performance to improve or deteriorate based on histogram accuracy, investigate the effects of histogram accuracy on query execution times and query-optimisation and present guidelines on when and why histograms should be updated. Achieving these goals required the project to be split into two development efforts, the first part centred around the development and iterative testing of a use-case and its accompanying reactions to histogram accuracy. The second part focused on developing and presenting histogram updating guidelines by implementing them in MySQL and experimenting with different variations of updating rules and values using the use-case developed in the first part as the "base".

To explore these items the report starts with the theory required presented in chapter 2, it then continues onto presenting the current state of the art in chapter 3 before presenting the design of both the use-case and the histogram updater in chapter 4. Then the implementation and iterative testing of the use-case is presented in chapter 5, followed by a presentation of the implementation done in MySQL in chapter 6. Finally, the results of testing are evaluated in chapter 7 before the report closes in chapter 8 with conclusions and suggestions for future work.

Chapter 2

Theory

Query optimisation is complicated and combines several different parts of the database into making a decision on which query plan to use. Understanding the reasoning behind that decision, how it can be affected, and how it affects query execution time is described through introductions into different topics. We start by describing how database queries work and how the database optimises them. We then show a brief example of how a query plan is chosen. Followed by an introduction into different statistics types, then a section discussing cardinality estimation, before some examples of the most common join algorithms are presented along with how the join order plays a role. We close with introductions into how database systems are evaluated, an introduction of time-series databases and a summary of some statistical terms and theory. Large sections of this chapter are parts that were initially presented in the previous work we performed on this topic in [3]; it is included here as well for completeness sake.

2.1 Database queries

Relational Database queries are formulated using the structured query language (*SQL*) standard. SQL is generally divided into three different categories, data manipulation language(*DML*), data definition language (*DDL*) and queries. DML contains the update, modify and delete statements, although select statements are often included in this group as well. While DML statements also need optimisation, they are not optimised in the same way as normal queries. In this report, the term query is used about a select statement unless otherwise specified, while a statement can be any of the DML or DDL statements including select. SQL is a declarative language, meaning that it defines *what* the results of the query should be not *how* to obtain the result [4]. Meaning that when the relational database management system (*RDBMS*) receives a query, it has to decide how to calculate the results of the query. The different ways to execute queries are known as *execution plans* or *query plans* and even though they produce the same result, they can have massively varying execution times[4], [5]. Thus selecting efficient query plans is important for database performance; this is the task of the query-optimiser.

2.1.1 Choosing a query plan

Choosing the optimal plan is not a trivial problem and query-optimisers in modern RDBMS' are large, complicated and have been developed and refined through years of testing and research[5]. Their implementation will vary from one system to another, but they all have a few things in common; The plan believed to be the fastest of those evaluated is chosen, while no guarantee is given that faster plans do not exist. The search space is limited to only a subset of the available query plans, often choosing only to consider plans created by a particular set of join algorithms such as those created by a left/right deep join tree. Statistics about tables and columns are employed in an attempt to predict the cardinality of different parts of the query. Cardinality results are used along with heuristics and cost models to approximate the cost of each plan [4].

For the optimiser to choose a suitable plan it needs to estimate the execution times of the different plans in the plan space[1]. Estimating the time required to execute the plan is directly related to estimating the number of rows needed to process at the different steps of the plan. Each process has a different cost associated with it; reading rows from disk is relatively expensive while comparing values in the CPU is cheap. This is where the name cost-based optimiser comes from. The optimiser approximates the results of different parts of the query attempting to discover the most efficient combination of steps resulting in the most efficient execution plan. Initially, it starts with the cardinality of one of the tables included in the predicate as the result set. It then applies the different predicates to determine which one reduces the result set the most. The reduction factor from different predicates is termed the *selectivity* of the predicate[2]. Selectivity and cardinality are tightly linked, multiplying the selectivity of a predicate to the original cardinality results in the number of rows expected to remain after the predicate is applied. If we have multiple predicates we wish to evaluate, then the result of the first evaluation is the cardinality for the second and so on. It is important to remember that selectivity is a ratio, a number between 0 and 1, while cardinality is the number of rows, an integer larger than or equal to 0. How the plan reads data from disk also impacts its execution time, the problem with choosing a retrieval strategy is that there is not one which is always fastest, it depends on the number of rows that end up being fetched. There are three main strategies for retrieving data; a linear search which can be applied to any file, a binary search which requires the search attribute to be ordered and finally an index search which requires the appropriate index to exist on the search attribute [4]. Below we show how this works for a given query.

Let us try to apply this to choose a query plan for the following query.

```
SELECT * FROM T1 WHERE C1 < 100 AND C2 < 50;
```

Firstly we must limit our search space, in this case, there are no joins which make things a lot easier and makes the possible plan space quite small, because of this we can check all the possible plans and do not need to limit our search space. Secondly, we check the cardinality of table T1 and find it to be 100000 rows. Thirdly we try to determine which of the two predicates `C1 < 100` and `C2 < 50` have the highest selectivity, meaning which of the clauses cause the largest reduction in the result set. To do this we look at a

[1]The plan space is the set of all possible query plans generated by the optimisers different combinations of allowed join algorithms, the ordering of joins and predicates and different data retrieval strategies [4].

[2] The term predicate in this context means any operation that defines a part of the result. Thus `C1 < 100` is a predicate, but `T1.C1 = T2.C1`, commonly referred to as a join, is also a predicate.

histogram for each of the two columns `C1` and `C2` and find that the predicate `C1 < 100` has a selectivity factor of 0.1 and for `C2 < 50` it is 0.82.

Then we look at what retrieval strategy we should use for this data. In this case, we have already created a sorted index for C1 and one for C2. These indexes have been created for this table since columns C1 and C2 are often used as predicates in queries. We then choose to fetch the data using index search, since it is faster than fetching all rows using a linear search, and we have not stored table T1 sorted on the relevant columns. Thus using a binary search is not possible. We do have to consider whether to fetch the data using the C1 or C2 index. In this case, we check the selectivity and see, quite clearly, that the C1 predicate has a much higher selectivity, i.e. lower number, than C2, so naturally, we choose that one. After we have read all rows into memory, we have already satisfied the first predicate, and we can check that each row satisfies the second predicate and output the result.

In this example, knowing that the predicate `C1 < 100` would return approximately 10% of table T1's rows was vital when choosing which of the available retrieval strategies to use. We can see from the selectivity estimates that we expected the result to be about eight times larger if the other index is used. If we assume the computation time for checking the second predicate is sufficiently small to be disregarded entirely, the plan fetching based on C1 should execute about eight times as fast a plan fetching based on C2. This is not unheard of when considering query plans. The difference between the optimal and worst query plan can be several orders of magnitude, as shown by Leis et al. in [5]. Thus choosing the optimal plan is not only a convenient way to improve performance but necessary to answer even relatively simple queries promptly.

2.2 Histograms and other statistics

Modern RDBMS' create and maintain a host of different statistic types on almost any data in their tables. The creation and updating of these statistics are handled differently from system to system and is covered in chapter 3 of this report. In this section, however, we describe the different types of statistics available and then investigate the type most interesting to this report, the histogram.

In general the DBMS will store a number of different statistics about tables and columns [4].

For tables it is common to store:

- The number of rows in the table, *cardinality*.
- The width of the table.
- The number of blocks occupied by the table on disk.
- The blocking factor *bfr*, which is the number of tuples per block.

For columns the normal is:

- The number of distinct values, *NDV*.
- The *max* and *min* values.

– A histogram.

Not all of the above types, while all are important and have their usages, are the focus of this report, we will be concentrating on the histogram.

Histograms in databases are designed to *approximate* the distributions of values in columns. It is used by the query-optimiser when determining the cardinality of different parts of a query. The optimiser needs the approximation the histogram provides to approximate the result set of some predicates that can be used in a query [6]. Through the years, different types of histograms have been developed, with differing strengths and weaknesses but all with the intent of improving the approximation accuracy and dependability of the histogram[6]–[8].

When a histogram is created, a scan of the underlying data must be conducted first. The scan is usually done in one of two different ways. Either a scan of the entire table is performed, or only a portion is scanned, known as *sampling*. When sampling, one relies on the assumption that the sample is a large enough and random enough to be representative for the rest of the table. When done correctly, sampling does not necessarily reduce the accuracy of the histogram as shown by Chaudhuri et al. in [9] where they manage respectable results using their adaptive sampling algorithm. They also show that when tables grow beyond a specific size, the number of rows needed to sample stops increasing. Leveraging this attribute can, in some cases, drastically improve the execution time of the statistics gathering. After the data is fetched from disk, it is sorted into ”buckets”, the number of elements in each bucket is then stored in the histogram and can be accessed by the query-optimiser when needed [4]. If a histogram is said to be an approximate histogram that generally means the histogram is based on a sample, this notation is used exclusively in that manner in this report. In other words, the histogram is an approximation of the underlying data distribution, but that does not mean that the histogram was based on a sample. If the histogram itself is approximate, that means the histogram is based on a sample of the dataset. The types of histograms explained and listed in this report are by no means a comprehensive list; however, they are all relevant and in use in modern DBMS', and are for that reason described here.

2.2.1 Equi-width and equi-depth histograms

The difference between histograms is generally in how the data collected is sorted into the different buckets the histogram contains. There are two simple ways to sort the data into the different buckets; one can either keep each bucket ”filled” to the same level known as *equi-depth*, or each bucket can have the same boundary range, known as *equi-width*. E.g. could an equi-width histogram have ten buckets of width ten, which would span the domain from zero to 100. While an equi-depth histogram for the same domain could have six buckets, two of which could span almost the entire domain while the remaining four spanned a small range with higher accuracy. This describes the situation seen in figure 2.1.

The advantages of the equi-depth histogram are quite apparent; it dynamically increases or decreases granularity where it is needed. Thus it is better equipped to adapt to high data skew. As shown in figure 2.1 the equi-depth histogram adapts to the skew of the data and more accurately models the distribution of data than the equi-width histogram does. The disadvantage with equi-depth is increased space requirements, equi-width is

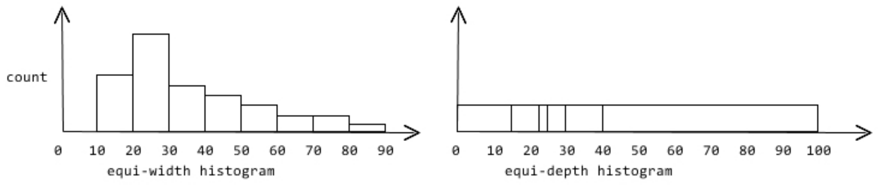

Figure 2.1: Equi-width and equi-depth histograms displayed on the left and right respectively. Figure from [15].

therefore preferred when storage space, and in particular when RAM space is at a premium. The price per megabyte has decreased substantially in the last twenty years, this along with improved accuracy means that the equi-depth histogram is preferred to equi-width in all modern systems [6], [10], [11]. The price of storage, especially primary (RAM), has decreased so much in the last years that several of the paradigms that shaped and defined the design of RDBMS' might no longer apply. An example of this is that many modern databases have an in-memory option, which means that the database and its tables are stored entirely in primary-memory. Secondary-memory is used to store backups of the tables and logs creating during operation. These databases are often termed as IMDB's, *in-memory database*, and include both standard RDBMS and their more recent counterpart NoSQL databases. Switching from disk-based storage to in-memory storage increases performance drastically. In fact, some systems such as Aerospike [12], Redis [13], and Oracle TimesTen [14] are even designed from the ground up to be in-memory systems.

2.2.2 Frequency- and top-n frequency-histogram

The frequency histogram, unlike other histograms, is not an approximation of a distribution; it is a compression method that allows a precise recreation of the distribution of columns. This type of histogram provides the greatest accuracy in selectivity estimation [6]. In a frequency histogram, each distinct value will have its bucket, and the histogram tracks the number of records in each bucket [10]. A histogram of this type allows us to know without any approximation how many rows in the column will satisfy the predicate. Creating this type of histogram is only possible if we have as many or more buckets as distinct values in the column (*NDV*).

When there are more NDV's than there are buckets in the histogram, we cannot use a standard frequency histogram. However, we still want to exploit a fact highlighted by Ioannidis in [6], namely that accurate data on the most common values has a significant impact on selectivity estimation. If the n-most frequent values occupy more than a threshold p of rows, then a top-n frequency histogram will provide a better approximation for the *popular* values. In this case, the least popular values can be discarded and not stored in the histogram at all if we assume they have a negligible impact on accuracy [10].

2.2.3 Hybrid histogram

In an equi-depth histogram, the number of items in each bucket is kept to a somewhat similar level. However, a value cannot span two buckets, unless they are singleton buckets[3], it must be entirely in one bucket. As such, values that are *almost* popular may be approximated very poorly by the equi-depth histogram in some cases [10]. The hybrid histogram is an attempt at achieving the "best of both worlds" from the frequency and equi-depth histograms, aiming to improve the accuracy in the cases of almost popular values. Firstly values are grouped so that none span buckets, secondly, for each bucket, a record of how many times the endpoint value gets repeated is stored. When using the hybrid histogram, one does not have to assume a uniform distribution *inside* the bucket, which is the norm for histograms, since the repeat count stores the number of times the last value appeared in the bucket. The repeat count reveals information about the buckets internal distribution and helps improve accuracy for almost popular values.

2.2.4 Multidimensional histograms

In the descriptions given up to this point, all histograms have been one-dimensional. Meaning that the histogram approximates the distribution of a *single* column. When a query includes multiple predicates, its selectivity can be computed by assuming that the columns are independent, meaning that the selectivity of each column can be multiplied to give the compound selectivity. This technique, known as attribute value independence (*AVI*), is employed by modern systems [16]. However, this assumption is often incorrect and can result in the DBMS' either over- or under-estimating the selectivity. [5]

Mit et al. show in [17] that creating and maintaining multi-dimensional histograms need not be expensive, slow or inaccurate. They achieve promising results with their dynamic summary data structure providing the basis for on-demand accurate histogram creation.

When examining figure 2.2, we can see that the two columns are not independent. They are dependent because observing the value of one of the variables influences our assessment of the probability distribution of the other. If we do not have a multi-dimensional histogram on these two columns, the optimiser would likely underestimate the number of rows returned by the query. If instead, we had created a two-dimensional histogram using the methods and algorithms outlined by Mit et al. in their report, the optimiser could find better approximations. These would, hopefully, achieve a much more accurate estimate of the predicate selectivity, which in turn could lead to choosing a better query plan [7].

To avoid these types of misestimations, we should not assume that columns are independent and rather employ some technique to deal with correlated columns. In [16] Poosala and Ioannidis explore a novel idea of using singular value decomposition (*SVD*) instead of the normal approach using multi-dimensional histograms. Through their research, they conclude that while the SVD approach has the positive trait that it only requires one-dimensional histograms to compute the selectivity, it does not provide sufficient accuracy when compared with their MHIST-2 algorithm. They show that when using AVI, they regard estimation accuracy on any multi-dimensional predicates to be inadequate, and

[3]A singleton bucket is a bucket containing only one value. If a value occurs more times than the depth of our buckets, it must span buckets and those buckets must then be singleton buckets.

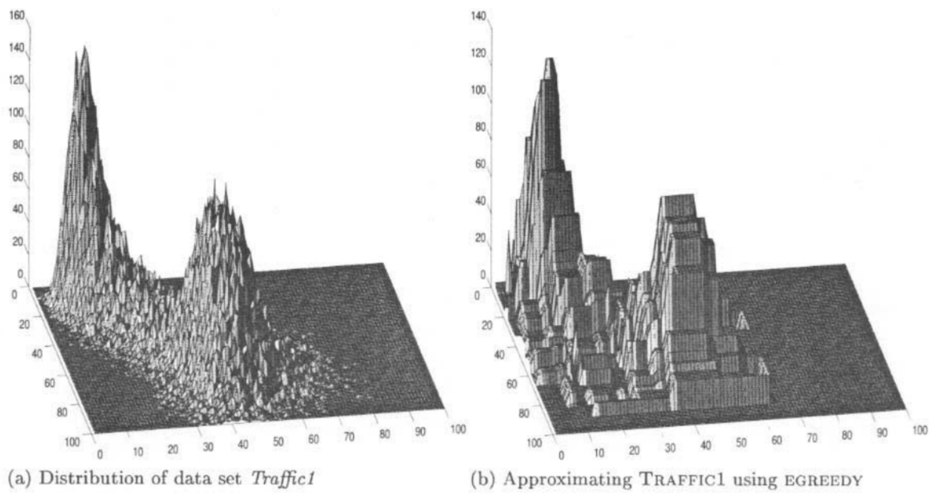

Figure 2.2: True distribution of the Traffic dataset is depicted on the left, with the two-dimensional histogram approximating its distribution on the right. Figure from [17].

recommend abandoning AVI in favour of multi-dimensional histograms computed with MHIST.

2.2.5 Updating histograms

If a histogram is created correctly then it should, when it is new, be as accurate of an approximation to the distribution of its column as possible, regardless of it being based on a sample or not. Meaning that when we create a histogram for a column, we want it to be an accurate approximation to that column even if it is an approximate histogram. However, what happens to this approximation as time progresses and the underlying data changes? In theory, if the data changes in such a way that the distribution of the underlying column changes then our histogram will at some point no longer be accurate, it has *drifted away* from its underlying data set. How long this takes is dependent on how much and what type of changes are made to the original column and what we define to be accurate [6]. No matter how long that takes, we should at some point be forced to update our histogram to keep it accurate, and it is at this time we say the histogram has gone *stale*. If the histogram has gone stale, we deem it to no longer be reliable for the optimiser. We claim that selectivity and cardinality estimates will be too erroneous. When the histogram has become stale, there are generally two ways to make it accurate and useful again.

A naive implementation is to delete the histogram and recompute it from the ground up. This approach is simple, and we can use the same algorithm we used to build the histogram in the first place and delete the old one when we want. However, there is a drawback; if histograms are quickly going stale, we could end up spending too much computation power on maintenance. We can mitigate this to an extent by applying the

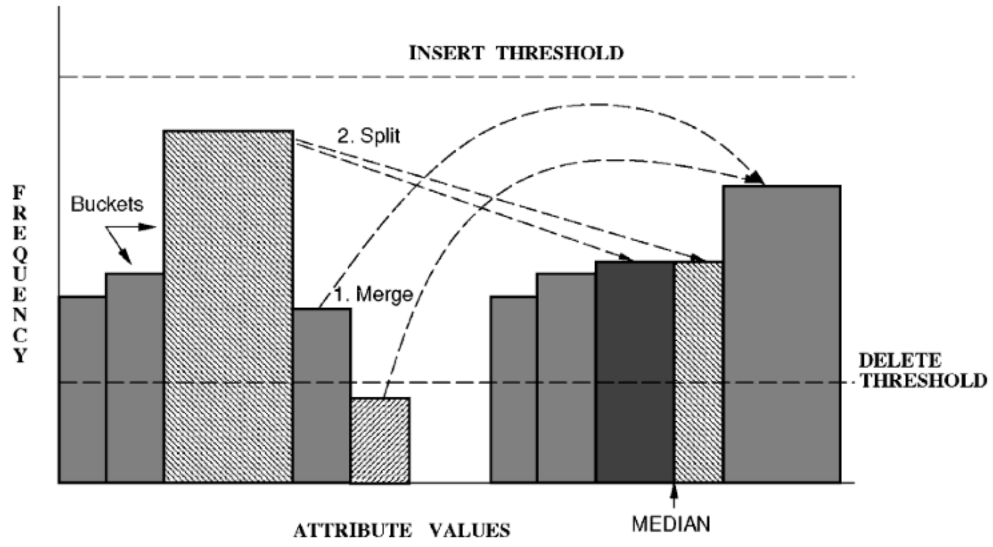

Figure 2.3: The split and merge tactic of Gibbons et al. in action. The smallest bucket just went below the delete threshold, as such it must be merged with one of its neighbours. Figure from [18].

algorithms Chaudhuri et al. develop and showcase in [9], which would lower the computational cost associated with a histogram update. These algorithms are not available in MySQL however, and implementing them is outside the scope of this project.

The other option is to keep the original histogram and change only the buckets which are no longer correct. In literature, this approach is termed *incremental updating*, and it is shown to retain accuracy under updates while improving computation time by Gibbons et al. in [18]. They combine a split and merge technique for buckets with a backing sample used to check the accuracy and recompute the entire histogram. The backing sample is a predetermined sample of the table kept up to date as updates arrive at the table. As buckets reach a threshold difference from their optimum size, the bucket is either split or merged. When the accuracy of the histogram in total is too far from their error threshold, in their paper termed γ, they recompute the entire histogram. Depicted in figure 2.3 we see how before the split and merge operations the frequency of items in the buckets are quite varying and after we see something which more closely resembles an equi-depth histogram with optimum item count in each bucket.

In the literature search conducted during this report, no theory could be found, nor do we believe there exists any theory, *defining* when a histogram should be marked as stale. We believe there to be several reasons that cause defining a theoretical optimum for staleness to be complicated. The two most prominent being the sheer number of variables which influence the accuracy of histograms, and what we define to be accurate. How quickly a histogram stops being sufficiently accurate will wary from one database to another. Some databases see large amounts of traffic and churn on tables while other databases may have stable data that rarely ever changes. Some databases can have value distributions that are continually changing due to updates and inserts. While others may

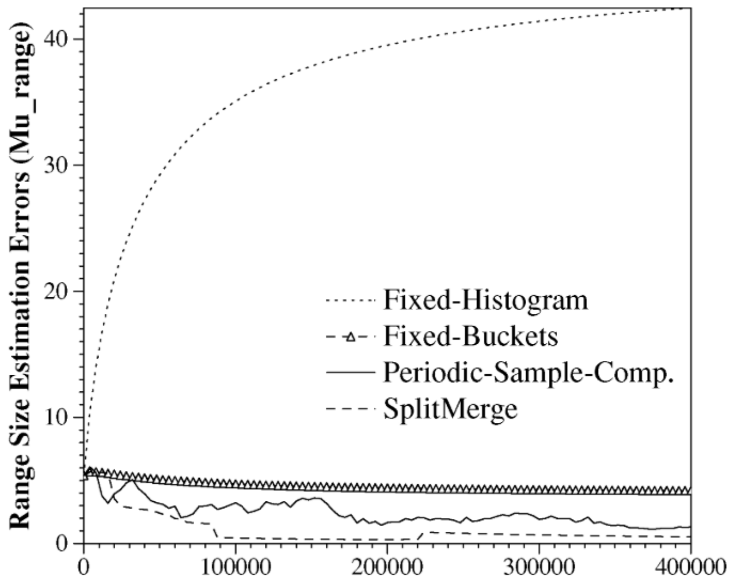

Figure 2.4: Cardinality error for range estimation after inserting rows. The algorithms of Gibbons et al. are compared with fixed buckets and fixed histograms. Figure from [18].

have just as much churn but stable value distributions within columns [4]. These types of uncertainties and differences mean that formulating any generally applicable theory is exceedingly tricky.

We did, however, discover that in [18] Gibbons et al. performed several tests comparing their algorithm to others. They compare their accuracy with that of other techniques, including non-updating, *fixed histograms*. We see in figure 2.4 that our hypothesis of histograms becoming stale after some undetermined amount of time seems reasonably accurate. The estimation error by the non-updating histogram is strictly increasing and would be providing poor cardinality estimates to a query-optimiser. Both algorithms proposed by Gibbons et al. maintain accuracy; in fact, they increase in accuracy after the first roughly 100 000 insertions. They also have the benefit of being tunable, meaning that we can tune γ to achieve a more efficient or more accurate histogram depending on needs.

While Gibbons et al. have done extensive testing comparing efficiency and accuracy between histogram updating methods. They have not analysed the effects of these differences in accuracy and efficiency in the context of query execution times in an RDBMS. Since we do not have any data on this particular situation, we could be forced to consider the state-of-the-art databases heuristics to be good choices of heuristic values, at least until we have performed our analysis.

[4]This can be caused by chance, e.g. two rows are updated in such a way that they swap values, this would not change the distribution of the column in question, but the table has still been updated.

2.3 Cardinality estimation

As pointed out by Christodoulakis and Ioannidis in [8], the join operator is the operator most sensitive to cardinality approximation errors. A query can often contain several join operators, the errors from the first approximation will propagate through all the joins. At each step, the small errors induced will compound into a possibly huge error in the final estimate. For this reason, inaccurate cardinality estimates for joins can lead to terrible execution plans. In modern systems accurate estimates for join operator cardinality is one of the largest contributing factors to reductions in execution time, especially for queries consisting of several joins [4], [5].

In MySQL, the query-optimiser does not attempt to estimate cardinality using histograms for any predicates that involve two variables/columns, e.g. predicates of the form `C1>C2` or `C1=C2` will be estimated with heuristics. It only uses histograms to estimate cardinality when a column is compared to a constant [19]. Since the predicate in any join always contains two variables, this means that the MySQL optimiser makes no attempts to compute cardinality of joining tables, and relies on heuristics and other statistics to determine the resultant size.

Towards better join cardinality estimates In [8] Christoduolakis and Ioannidis use histograms on the frequency distributions of join variables to approximate the cardinality of different join predicates. Their join variables are Zipf[5] distributed to more closely resemble a real-world database. They attempt to find optimal histograms which compute this cardinality while limiting the worst-case approximation error for any arbitrary query. They find that given their restrictions on the join type, namely *t-clique* queries, serial histograms are always the most optimal. They also conclude that while the most optimal histograms are always serial, which particular serial histogram it is, is dependent on the relations involved in the query and their particular frequency distributions. This translates to finding different histograms for the different join predicates that are common for the given relation.

In general, the accuracy of complex cardinality estimates is affected by; the ”correctness” of assuming AVI, the accuracy of histograms used to determine predicate selectivity and the accuracy of the estimated join selectivity. As we will discover later, modern systems are assuming AVI unless otherwise specified. This is often faulty as values that are joined together in RDBMS’ are seldom independent of another, which Leis et al. show in [5] by comparing the results of their tests with the IMDB dataset and using the common TPC-H benchmark.

2.4 Join algorithms

Many SQL queries require us to join two tables based on a specified relation R_i. The choice of join algorithm and the ordering of different joins often has the most significant influence on the execution time of the query plan. Indeed, sometimes the order *within* one

[5] In a Zipf distribution the distribution of each item is inversely proportional to its rank in the frequency table. Meaning that the most frequent item will appear twice as often as the second most frequent, three times as often as the third most frequent and so on.

specific join can have a profound effect as well. Below three principal join algorithms are described to help give a better understanding of why the query-optimiser is so dependent on appropriate cardinality estimates for these operators. We will not describe exactly how the optimiser decides which algorithm to choose, other than that it is mainly based on the size of tables and by extension the cardinality estimate of predicates [6].

2.4.1 Nested loop join

The simplest solution to joining two tables together is to check every possible combination of rows [7], this the nested loop algorithm (*NLJ*). It does not require any particular access paths on the table or the file and will always find all matches, and it is a generalisation of many of the algorithms we will explain later [4]. In NLJ we choose one of the two tables as the inner table and the other as the outer table, it makes a difference which table is chosen as the inner and outer if they differ in size. It is advantageous to use the smaller table, fewer blocks on disk, as the outer table as this reduces the total number of I/O operations to disk[4]. For each row in the outer table search for a match of the join predicate in the inner table, if there is a match then output the tuple. Once this is done for all rows in the outer table, we will have found all possible matches between the two tables. The disadvantage to NLJ is the number of times we have to read the tables to perform the join. For each row in the outer table, we must read all rows in the inner table resulting in a $n_o \cdot T_i + T_o$ size of reads. Where n_o is the number of rows in the outer table, T_i is the size of the inner table and T_o is the size of the outer table. If we are required to join more than two tables together, we have to chain our joins, meaning that we first have to join two tables together and then join the third table to the result of the first join. This result is often called the *intermediate result* [4], and when the second join is performed the same rules of size, outlined above, apply. The smallest table should be chosen as the outer table and the larger as the inner.

A variation of the regular NLJ algorithm is *index based* NLJ; it requires an index or a hash key for one of the tables. If the index exists for table T_i we read table T_o into memory and for each line use the access structure to immediately fetch every row in T_i which satisfies the join condition [4]. This type of join algorithm is typically quicker than the general NLJ algorithm but necessitates a particular access structure. The cost of this algorithm can be shown as $T_o + n_o \cdot x$ where T_o is the size of the outer table, n_o is the number of rows in the outer table, and x is the cost of using the index for lookup, which generally has a low cost.

In the case of the regular NLJ, we can see how a reasonable cardinality estimate will lead the optimiser to choose the optimum ordering of tables. Also, we see how a lousy estimate can cause it to choose wrong and select a suboptimal plan. Since the index NLJ requires a specific access structure, it might not always be possible to choose any ordering of tables, in those cases, the cardinality estimate is not essential for the ordering. However, it can still be relevant; a lousy cardinality estimate for T_o can be the reason an index NLJ was chosen when it was not the most optimal. It is also clear that a poor cardinality

[6]These modern optimisers are often called cost-based optimisers since they estimate what the cost of executing a query with each plan is [4].

[7]Checking every alternative is inefficient and is commonly known as the brute force or exhaustive search method.

estimate can cause the wrong two tables to chosen as the first ones to be joined when there are several to choose from. It is beneficial to choose joins such that the first one creates an intermediate result that is as small as possible and then join the smallest available table to the intermediate result. Which causes each subsequent intermediate result to be as small as possible, and therefore causes subsequent joins to require fewer computations.

2.4.2 Sort-Merge join

The most efficient join algorithm is the sort-merge join algorithm. If the prerequisites are in place, it only requires a single pass over both tables to find all pairs of rows satisfying the join condition. However, not only does it require both tables to be sorted, but they must be sorted on the join columns themselves. If this is the case, we can start by reading the first blocks of both tables into memory and choose one as the inner table. Then we look for a match for the first row in the inner with the outer table. Once located, we choose the second row in the inner table and continue from the point where we found our first match. We use $R(i)$ to refer to the i'th record in table R and likewise for $S(i)$. Because the tables are sorted on the join predicates $S(2)$ must be greater than $S(1)$ and so will the match in the outer table. E.g. if $R(4)$ matched with $S(1)$ then the match for $S(2)$ must be $R(5)$ or higher [8]. If the tables are not sorted, we can sort them beforehand and then perform the join using sort-merge join, but this may not always be faster than other algorithms [4].

As stated at the beginning of this paragraph, the sort-merge algorithm is the most efficient join algorithm if the prerequisites are in place. If the tables in question are sorted on the join variables, the choice of algorithm is trivial and not dependent on any estimates. However, if for instance, only one is sorted, it may be beneficial to sort the other table and use sort-merge join, but we need cardinality estimates and cost models to determine that. Again, it is evident that even in cases which should be reasonably straightforward, the choice of join algorithm is not trivial.

2.4.3 Hash join

The last join algorithm to be presented here is the hash join; it consists of two phases, the partitioning phase and the probe phase. In the partitioning phase, we read the smallest of the two tables into memory/primary storage and create a hash table. The hash table is created by hashing the join variable(s) to a set of buckets; once the table has been created, we can start the probing phase. In the probing phase, we read as much of the larger table into memory as possible. Once we have read the parts of the table that will fit, we compute the hash value for the join column(s) and check if the value hashes to a bucket with any rows in it from the partitioning phase. If it does we check that the value of the row in the hash bucket matches to the one we just probed if it is a match we output the tuple, if not it is a false positive and we disregard it. We do this on a per-row basis and once done for all the rows of the larger table - the join is complete [4]. As seen in figure 2.5 two values, represented by colours, which are not the same can hash to the same bucket, also when

[8]Given that the tables do not contain duplicates. If they contain duplicates, then the match for $S(2)$ must be $R(4)$ or greater.

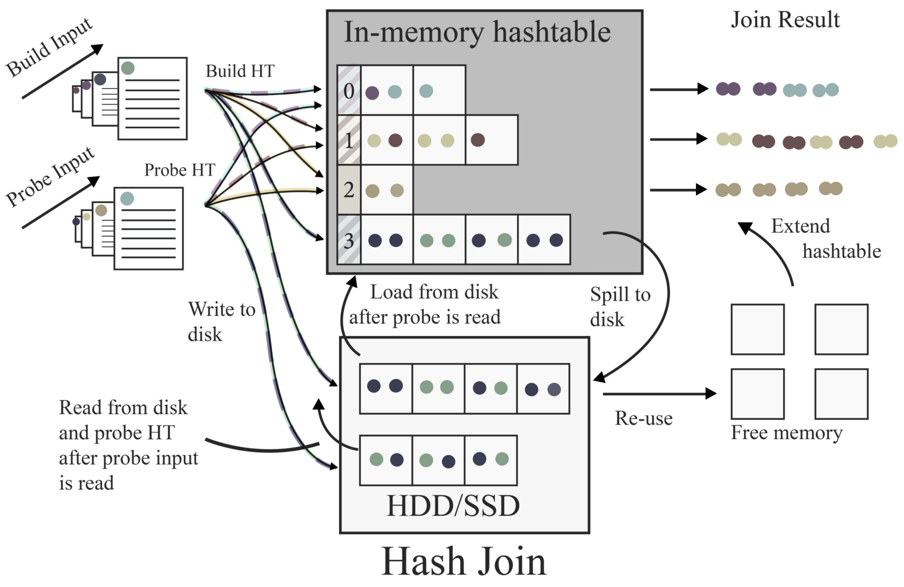

Figure 2.5: Overview of the structure of a hash join, the different colours represent different values. Different colours can hash to the same bucket, and buckets will overflow into disk if necessary.

rows cannot fit in the hash table we created, they spill to disk/secondary storage and must be handled in the next round.

This type of join requires us to set up the hash table, to do that we need some part of memory allocated to the table. If the part allocated to the table is not large enough, then hash buckets will start to spill into secondary storage until we are done with the partitioning phase. Any rows that hash to a bucket that has started spilling into secondary storage during the probing phase will have to be stored in secondary memory, and will not be evaluated until all rows have been probed. Only then will the rows that spilt into secondary memory be read into primary storage and processed again. This process will happen as many times as is necessary until all rows have been hashed and tested [4]. Spilling can have a significant impact on execution time, as shown by Leis et al. in [5]. They show how a query-optimiser wrongly estimates the size of the two tables to be joined due to inaccurate histograms. Which causes spilling to disk to occur multiple times over, resulting in an execution time much worse than what could have been achieved with a nested loop join. This is an example of how a poor cardinality estimate from the optimiser caused a sub-optimal execution plan. In this the case, it might be that the hash join was the optimal choice; the problem was that not enough space was allocated in memory for the hash table, causing it to spill to disk.

2.5 Evaluating database systems

Throughput, be that query, insert, update or delete throughput, of a database system is a measure of its performance, and different database systems, and indeed different databases using the same system will often have very different throughput levels. When a database is tested for throughput performance, the TPC benchmarks are often used. There are several different types, each designed to model different environments. The TPC-C benchmark *"simulates a complete computing environment where a population of users executes transactions against a database"*[20], and the performance of TPC-C is measured in *"new order transactions per minute"* the primary metric is the number of transactions per minute[20]. The TPC-H benchmark *"is a decision support benchmark. It consists of a suite of business-oriented ad-hoc queries and concurrent data modifications. The queries and the data populating the database have been chosen to have broad industry-wide relevance. This benchmark illustrates decision support systems that examine large volumes of data, execute queries with a high degree of complexity, and give answers to critical business questions."*[20]

There is a problem with the TPC benchmarks; however, as Leis et al. highlight in [5] where they show that in the TPC-H benchmark, the data in different tables is independent of another. This means that when Leis et al. examine cardinality estimates errors for queries in the TPC-H benchmark, they find that PostgreSQL optimiser has almost no error in its estimates. Which is visualised in figure 3.1, where we see that there is a significant difference in estimation error between the JOB and TCP-H benchmark, caused by the JOB benchmark using a dataset from IMDB in which data is not independent [5]. Thus, the PostgreSQL optimiser should choose good plans for the TPC-H benchmark, while for the JOB benchmark the chosen plan could be a poor one. If the PostgreSQL optimiser estimation accuracy was tested solely with the TPC-H benchmark one could argue that the optimiser predicts cardinality very accurately, however, from Leis et al. testing we now know that is not the case for the JOB benchmark.

Leis et al. show how important it is to carefully consider, not only the complexity of queries that are used to test the database, but also the attributes of the data being queried. We will try to keep this in mind when designing and developing both our queries and our dataset during the latter half of this report.

2.6 Some statistical terms and theory

Normally when describing statistical distributions, one thinks of them as representing the selection of elements one would end up with if one were to draw a large amount of them from a very large set [21]. For instance, if one rolls a dice and records the number, and do that enough times, one would expect there to be equal amounts of each number. Indicating that the distribution of rolling dice is uniform, rolling a six is as likely as rolling a one since after all those rolls there were roughly as many one's as there were six's. In the design and evaluation chapters of this report, statistical distributions and confidence intervals are employed, the required knowledge to understand their usage in those chapters is presented and reviewed here.

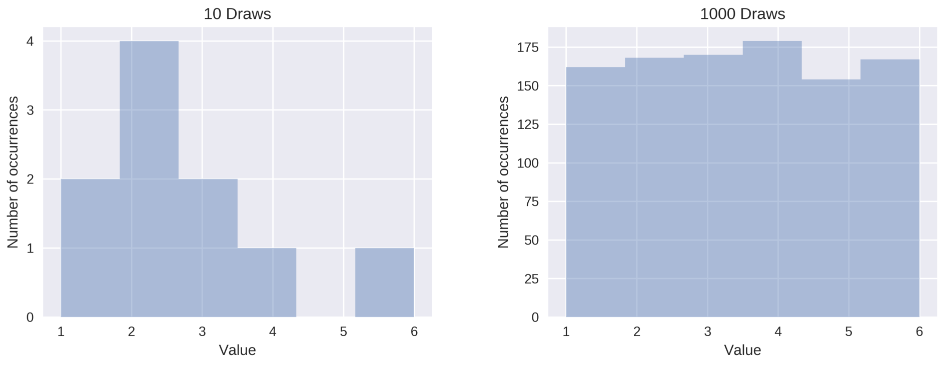

Figure 2.6: Box plots of values drawn from a uniform distribution

2.6.1 Uniform distribution

If an attribute is uniformly distributed, each possible *value* will have the same probability of happening for each *instance* of the attribute. A typical showcase of this is the dice-example from above; however, there are many other variables which are uniformly distributed as well. Most things that are uniformly distributed require there to be many instances before the distribution appears uniform. An example is a random number generator. They are designed to produce any number with an equal probability. Shown in figure 2.6 are two examples of values created by a random number generator plotted in graphs to visualise. In the left plot in figure 2.6, we see that not every number has been chosen, as it should have been since it was expected that there be the same probability for each number. Additionally, some numbers have been chosen way more than other numbers. This is because not enough samples have been drawn from the uniform distribution. If on the other hand, one looks at the right plot where 1000 samples have been drawn, it is easy to see that the distribution is much closer to an even distribution of numbers across the possible values. A common trait with distributions, and probability and statistics in general, is that if one does not draw enough samples from the distribution, then the distribution of the samples drawn tends not to be equal to the distribution drawn from [21]. It is therefore essential to draw enough samples so that one can be sure that the drawn samples are distributed in the same way as the set they were drawn from.

2.6.2 Normal distribution

One of the most well known and understood distributions is the normal distribution, also known as the gaussian distribution and the bell curve. It is used to model the probability distributions of many natural phenomena, such as height, blood pressure and IQ, among others [21]. In figure 2.7, a plot of values taken from a gaussian distribution with a bell curve plotted above is shown. It can be seen that the values closely match the bell curve with a mean of 0, which was the mean of the Gaussian distribution from which the values were drawn. Some traits about the gaussian distribution; it is mirrored about the mean, often denoted μ, the area below its curve when plotted on a graph is always equal to one.

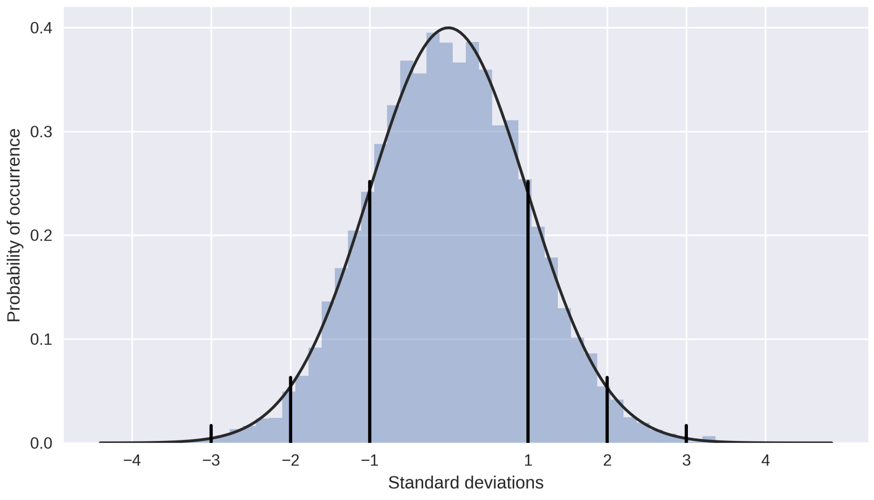

Figure 2.7: A sample of 10 000 values from a gaussian distribution with the bell curve of the distribution plotted ontop

There is roughly a 67% chance of drawing a value that is within one standard deviation of the mean, and roughly a 95% chance to draw one within two standard deviations of the mean. In figure 2.7 vertical lines are drawn at one, two and three standard deviations from the mean. In the figure that would then mean that roughly 67% of all values are within the two innermost vertical lines, and roughly 95% are within the two second most inner lines, finally roughly 99.7% of values lie within the two outermost lines.

In addition to accurately modelling many natural phenomena, several statistical theorems make use of the gaussian distribution. One such usage is shown below in the confidence interval example, where the sampling distribution follows roughly a Gaussian distribution when certain conditions are met. When the dataset is created later in this report, it will have to be populated with values. As mentioned earlier, many query-optimisers assumes a uniform distribution of values within a column if they do not have access to statistics about that column. If statistics on the column shows that values are uniformly distributed then the query plan will be the same whether or not database statistics were used during optimisation. For this project that is not desired, thus values used for the dataset will have to be drawn from something other than a uniform distribution. Instead, they will be drawn from a Gaussian distribution. There are other distributions than the gaussian to choose from that are also not uniformly distributed, such as the Poisson distribution and the Exponential distribution [21]. However, they do not provide any significant advantage in this case, and the Gaussian distribution will, as stated above, accurately model many natural phenomena, and can, therefore, be used in many different use-cases.

2.6.3 Confidence interval

Statistics give the ability to conclude on the attributes of a large set based on attributes and values of a relatively small subset. When doing so, it would be very nice to be able to say something about how certain one is about those conclusions - which is what confidence intervals do. A confidence interval has three key bits of information, first and second, it defines a starting point and an endpoint of its interval, lastly a value between zero and one which is a measure of how confident the interval is about the first two values. Consider the following example.

There is an election coming up, and there are two candidates, `A` and `B`, there are five million voters, and we are tasked with conducting a poll what proportion of them is going to vote for candidate `A`, let us call that proportion p. Conducting a poll such as this is common, and happens every year before all big elections. We can not ask all five million voters, that is simply not feasible, so what can be done instead? We do something which is ordinary in statistics, namely use a sample as a representation for the entire set.

Let us say we choose one thousand voters as our sample; then we ask them whom they would vote for. It is found that 58% of them would vote for candidate `A`, let us call this sampling proportion $\hat{p}$. If we were to choose another set of one thousand voters as our sample, we could find that $\hat{p}$ was not 0.58, let us say it was 0.52 in this second set. The critical thing to notice here is that the sampling proportion will vary between samples. However, if we were to draw enough samples, we would eventually find that our sampling proportion is roughly normally distributed. There are a few prerequisites for this to be true, however, for instance, each voter has to have an independent opinion to all others, and the actual proportion,p, can not be too close to zero or one [21].

In our case, we will assume these prerequisites are met[9], in that case, it is known that the mean of this sampling distribution is equal to the real proportion p [21]. The formula for standard deviation is given by $\sigma_{\hat{p}} = \sqrt{\frac{p(1-p)}{n}}$ where n is the sample size, in this case one thousand. The value found for $\hat{p}$ could lie anywhere within the sample distribution, it is not known where the distribution is because we do not know where the mean, p, of that distribution is, that is what we are trying to find out. Something else is known; however, it is known that the probability that our value $\hat{p}$ is within $2\sigma_{\hat{p}}$ of p is approximately 95% because the sampling distribution is normally distributed. In other words, if one were to draw such a sample 100 times, the value one would get for $\hat{p}$ would be within two standard deviations of the true value of p 95 times. Now that tells us a lot about the real value of p, now if it was only possible to work out the value for the standard deviation. Well, it is if $\hat{p}$ is used as a substitute for p in the formula for standard deviation. Why this is possible is not discussed here, but it can be shown that $\hat{p}$ is what is called an unbiased estimator for p[21], and when that is the case it can be used as an estimator for p.

If those values are then plugged in we get that $\sigma_{\hat{p}} = \sqrt{\frac{0.58 \times 0.42}{1000}} = 0.016$. Now the standard deviation is known, and there is just one last thing that needs to be realised. Consider what was said earlier about *the probability that our value $\hat{p}$ is within $2\sigma_{\hat{p}}$ of p is approximately* 95%, and we tweak that ever so slightly it can be said instead that "with 95% confidence between 0.548 and 0.612 of voters support candidate `A`", now that

[9]This is not unrealistic given the large size of our original set, the relatively small sample and how unlikely it is that almost all voters would vote for either `A` or `B`.

is almost exactly what we set out to find. Granted, the exact value of p is not found, but an interval in which we are 95% confident that p will reside is found, and that is equally useful. By altering how confident one is about the interval that will change its size, for instance, would a 68% confidence interval for p be between 0.564 to 0.596 which is quite a bit tighter than that of the 95% confidence interval.

Hopefully, this short example has shown what a confidence interval is, how to use it, and why they are so useful when trying to determine fundamental statistical values. In this project, they will be used in plots to show how confident we are about the real mean of query execution times for given queries, datasets and level of histogram accuracy.

2.7 Time-series databases

If one wants a system that monitors events and faults, every data point that the system collects has a connection to a particular point in time. As events and faults occur and data about them are stored, a log of what has been happening is being created. Later on, when technicians want to determine the cause of or results of, those events, they will query the system for information about what happened around the time the event occurred. To answer that question one will have to filter the log based on a time range, i.e. one might need to find all events that occurred before or after a given date or the events that happened between two points. This can be accomplished by any RDBMS that supports storing timestamps in a column. However, when it is needed to answer the query quickly and be capable of having a large write throughput, a regular RDBMS' may begin to struggle.

Time-series databases are designed to handle this type of load. They can record vast amounts of events every day while also answering queries in a timely fashion. These types of databases often form the foundation of many systems in operations environments where one is attempting to monitor the "health" of one or more other systems. A pioneering monitoring system in this field was Gorilla developed by Facebook. It was designed to monitor and record the status of a large set of servers and systems reporting on any faults that were occurring in real-time. While also storing information needed to diagnose and solve problems when the time was right to do so. One of the design goals of Gorilla required it to handle over 700 million insertions per minute, and store over two billion unique time series identifiable via a string key, which they managed to achieve [22].

Many systems are specifically designed to handle time-series data today. They do not all share the same architecture, but they do have common traits none the less. For instance, it is common for these types of systems to be highly optimised for range queries, which are very common when querying time-series. They are also generally implemented as append-only databases, meaning that once something has been inserted into the database it can not be altered. Simplifying the data structure of the databases and allowing for higher insert throughput. They are also often implemented as in-memory databases to increase performance.

2.8 Summary

In this chapter, we have presented the theory needed for the following chapters of this report. We have seen that determining good query plans is not a trivial task, indeed avoiding choosing poor query plans is not easy either. The speed of the plan is dependent on many variables; the join algorithm used, the retrieval strategy used, available statistics, statistical accuracy, statistic type, the ordering of operations, database hardware and the relationship between all these variables. Trying to approximate the execution time of queries while all these variables are influencing accuracy in unpredictable ways, and only have a second to do it, is what the query-optimiser of any RDBMS is doing for all queries. It has also been showed that there are several ways to measure how efficiently the query-optimiser is completing this task. Moreover, we saw that some benchmarks commonly used were not accurate measures of how the optimiser performed in real-world cases. We also gave a short introduction to statistical distributions and some of their most useful attributes, along with the powerful tool that is the confidence interval. In the next chapter, we will be looking at how a selection of database systems handle, utilise and manage some of the elements discussed throughout this chapter.

Chapter 3

State-of-the-Art

In this chapter, we start by exploring how the state-of-the-art is, with regards to histograms in DBMS' by examining how MySQL and three other systems handle their creation and maintenance. Nextly we discuss join cardinality estimation and its usage in these systems. The three other systems reviewed in this chapter should all be considered industry leaders. On the corporate side, both Microsoft SQL Server and Oracle database have hundreds of thousands of business customers and between them service most of the enterprise market. Spearheading the opensource DBMS market is PostgreSQL, who in later years have shown excellent performance, and in many cases rival that of the enterprise solutions. These systems provide an excellent basis for what can be considered state-of-the-art with regards to most things DBMS related. We wish to point out that, the features listed for each system in this chapter is not an exhaustive list, it is a short description of what we believe to be the most important for this project. This description of the state-of-the-art in terms of histogram invalidation and cardinality estimation, is originally from our previous work on the subject in [3]; it is included in this project with minor changes to reflect the change of scope.

3.1 Creating histograms

Deciding which columns warrant the creation and upkeep costs associated with histograms is a topic of debate, and the different systems considered here have different approaches to this problem. However, there are some ground rules which are followed by all of the systems. They all support creating and updating histograms on columns with numerical values, and they all have some form of the equi-depth histogram available as a statistic.

MySQL has no automatic creation of histograms but allows database administrators, *DBA's*, to create them using the `ANALYZE TABLE` statement with the `UPDATE HISTOGRAM` clause and then provide some columns on which to create the histograms[23]. The DBA can either use the default 100 buckets or specify a specific number of buckets to use in the histogram. MySQL implements two histogram types, the *frequency* and *equi-depth* histograms. Consistent with the definition of the frequency histogram in section 2.2, the frequency type is preferred when NDV is less than the number of buckets. Otherwise an equi-depth histogram is created [19].

Microsoft SQL Server is by default configured to handle histogram and statistics collection automatically by the `AUTO_CREATE_STATISTICS` parameter. This parameter gives the DBA the option of turning off automatic creation and perform it themselves, or the DBA can combine both. By allowing MSSQL to create its standard histograms and statistics while the DBA creates specific single- or multi-dimensional histograms on columns [11]. MSSQL implements its automatic creation by creating histograms for columns that do not already have histograms as the optimiser encounters them during query execution.

This automatic process will not attempt to restrict the number of histograms stored for each table in any way. MSSQL will automatically create multi-dimensional histograms if there exists an index on a set of columns. For these types of sets, MSSQL computes cross-column correlation statistics, which they call *densities*. A DBA can also create densities for column sets which are not part of any index, MSSQL will not do this automatically. MSSQL uses a form of the hybrid histogram, very similar to the one discussed in subsection 2.2.3. It also has the capability of creating histograms either based on the entire table or a sample[11].

Oracle database uses a procedure to gather statistics for the database. By default, the scheduler runs the `GATHER_STATS_JOB` every day between 22:00 and 06:00 and all day during weekends [24]. The job will execute a procedure in `DBMS_STAT` which will create and update histograms, and other statistics, for all columns that have an entry in `SYS.COL_USAGE$`. This way, Oracle database will not maintain or create histograms for columns that are not used by queries [10]. When creating histograms, the `ESTI MATE_PERCENT` parameter determines if the histogram is created using sampling, and what percentage of sampling to use. This parameter can be any value the DBA wishes or left to its default of `AUTO_SAMPLE_SIZE`. If the DBA wishes he can create new statistics for a table or column at any time using the `DBMS_STAT.GATHER_TABLE_STATS` procedure [24], and inserting a new row into the`SYS.COL_USAGE$` table. The `DBMS_STAT` package also gives a DBA the capability of creating *extended statistics* objects. These extended statistics objects inform the optimiser of the real-world relationships that exist between data, which causes it to use multi-dimensional histograms for its cardinality and selectivity estimation [1].

If the query-optimiser encounters a query with complex predicates and extended statistics are not available, the optimiser can use dynamic statistics. Dynamic statistics, originally termed Dynamic sampling, will when employed on complex queries execute the query on a sample data set and extrapolate its values as estimates for cardinalities [26], [27]. Whether or not dynamic statistics are used depends on the `OPTIMIZER_DYNAMIC_SAMPLING` parameter, which has to be set correctly[2].

[1]The procedures used for creating and maintaining histograms will also handle multi-dimensional histograms [25].

[2]Correclty, in this case, would require the parameter to be set to a value between two and 11 [24].

When Oracle database creates a histogram, it chooses between four types; the equi-depth, frequency, top-n frequency and hybrid -histograms, based on a set of parameters. If left to its default configuration, it will create a frequency histogram if the number of distinct values is less than the number of buckets. If the percentage of rows occupied by the top n frequent rows exceed a threshold p, it will create a top-n frequency histogram. Else a hybrid histogram is created [10]. A more in-depth discussion of how the different types of histograms work were given in section 2.2.

PostgreSQL uses the `ANALYZE` and `VACUUM ANALYZE` commands to create histograms on the columns of a table manually. For large tables the `ANALYZE` command will take a random sample of the table contents rather than examining every row, to achieve speedy collection even on large tables. The size of the sampled data is dependent on the number of buckets defined for the histogram, defaulting to 100. If left to its standard configuration PostgreSQL will have its autovacuum daemon enabled, and histograms will be created for new tables as they are loaded with data [28]. PostgreSQL creates the same types of histogram as MySQL namely the equi-depth [29]. PostgreSQL also has the capability of creating multi-dimensional histograms, although in PostgreSQL they are called *multivariate* statistics. Similarly to Oracle database and MSSQL, these types of histograms are not created by default. A DBA can specify a particular relation of interest, and PostgreSQL will create and maintain histograms for these as it does with other columns and tables [30].

3.2 Updating and maintaining histograms

Keeping statistics the DBMS has collected up-to-date and valid should be vital for approximation accuracy as shown in subsection 2.2.5. The usefullness of histograms is dependent on more than which histograms are available. It is also important when the histogram was last updated. The state-of-the-art systems examined in this report use different heuristics to determine when the histogram has gone *stale*. MySQL has no automatic updating of histograms as apparent from the introduction to this report. Meaning that any updating of histograms must be done explicitly by DBA's and that the only time histograms are guaranteed to be accurate is when they are first created [19].

MSSQL updates and maintains statistics on an as necessary basis, which means that when the query-optimiser receives a query, it checks whether the statistics for that table are *stale* or not. If the flag `AUTO_UPDATE_STATISTICS_ASYNC` is FALSE, which it is by default, the DBMS' will recompute the statistics there and then. Else the query will run, and statistics are computed in the background. There is an advantage here that if the flag is FALSE, a query plan will never be based on stale statistics [11]. However, if the computation of statistics is slow, the query might time-out, and the user might be forced to wait for longer than it would have taken the query to complete even with stale statistics.

Statistics are considered stale in MSSQL after $\sqrt{1000 \cdot R}$, where R denotes the number of rows for the table in question, updates or inserts to the table have been made since the last update. This rule means that larger tables will require more rows to change than smaller tables before the statistics are considered stale [11].

Oracle database uses the same job to update statistics as it used to create them, namely the `GATHER_STATS_JOB`. In the case of updating statistics, the job will recompute the statistics for all objects that are deemed stale. In Oracle database, statistics are considered stale when the underlying object has been updated significantly, meaning more than 10% of the objects rows have been altered [24]. Since the maintenance window is only open during the night by default, a highly volatile table could have stale statistics for most of the day. For these types of tables, Oracle database offers two solutions; Either the statistics for the table can be removed and the table-statistics locked, preventing them from being created again, in that case, the table will be treated as in section 3.3. Alternatively, the statistics for the table can be set to values that represent the *steady state* of the table and then locked.

PostgreSQL in its default configuration uses its *autovacuum* daemon to update the statistics of tables as they change during regular operation [28], [31]. This update is not triggered by query execution as in MSSQL but is rather a worker thread started by the persistent *autovacuum launcher* process. The update worker will be started every one minute by default if there are not too many workers running, defaulting to three [32]. Also, statistics can be gathered manually at any time by running the `ANALYZE` and `VACUUM ANALYZE` commands.

The workers started by the autovacuum daemon examine tables in the database sequentially and check whether the table statistics are stale or not. It considers statistics to be stale after $B + S \cdot n$ rows have been updated, inserted or deleted since the table last had its statistics updated. In this case, B is a lower threshold, S is a scaling factor both defined by parameters in `postgresql.conf` defaulting to 50 rows and 10% respectively, and finally, n is the table cardinality.

3.3 Missing histograms

When a query-optimiser encounters a column or table for which it has no histograms or other statistics, it has two options. Continue without any statistics and employ default values, called heuristics, or create new statistics. In the case of MSSQL creating new statistics is the standard operating procedure as it will compute histograms for each query if none exist or if they are marked as stale. For PostgreSQL, if a table has no histogram, it means there is no data in the table, and a histogram is not needed. In MySQL, if there is no histogram on the column, it will merely continue without one and use heuristics. However, in the Oracle database, something quite interesting happens.

Oracle database uses the job discussed in section 3.1 to create statistics for objects that do not have it. This job will only run during the maintenance window, however, and so if a new object is created and used before the window opens, there will not be statistics for the object. In this case, the optimiser will use *dynamic statistics* to avoid having to use heuristics. Dynamic statistics was introduced to help the query-optimiser avoid choosing substandard query plans. The statistics gathered by this job are not as accurate nor complete as the ones provided by the standard `DBMS_STATS` package. It does, however, give much more accurate estimates than the heuristics can. It is most effective when used on queries that query tables which do not have any stored statistics at all, or if histograms and statistics are very stale for the table in question [26], [27].

3.4 Cardinality estimation

During the literature search performed for this project, it was not discovered how Oracle database or MSSQL database's query-optimisers calculate join cardinalities. This is likely intentional since it is advantageous for a DBMS to approximate the join cardinality accurately. As discussed in section 2.3 and as shown by Leis et al. in [5]. E.g. if MSSQL is better at approximating join cardinality than Oracle database, then MSSQL should be better at choosing query plans on queries with many joins. Which, in turn, means that MSSQL is faster at answering queries which can cause customers to migrate to MSSQL.

Leis et al. describe in [5] how PostgreSQL estimates cardinality for joins, they show the cost-function the PostgreSQL query-optimiser uses but more importantly, they highlight the assumptions PostgreSQL makes in their query-optimiser.

- uniformity: it is assumed that all values, except the most-frequent, have the same number of tuples.
- independence: the assumption of attribute value independence is used to simplify the cardinality/selectivity estimation.
- inclusion: the domain of the largest key overlaps that of the smaller such that keys in the smaller are contained in the larger.

PostgreSQL makes these assumptions to ease the task of cardinality estimation. The independence assumption is especially useful for the optimiser since it means that complex predicates can be evaluated by merely multiplying the individual selectivity of each predicate. As discussed in subsection 2.2.4, these types of assumptions often lead to errors in estimates. In literature, these assumptions have been deemed too optimistic for a long time, and Leis et al. show how significant errors this assumption can cause.

Benchmarking cardinality estimation Leis et al. develop a benchmark which they argue more accurately represents real-world conditions when compared to the standard TPC-H benchmark. In their paper, they show the large discrepancy in cardinality estimate accuracy between the TPC-H and JOB benchmarks, depicted in figure 3.1. In the TPC-H benchmark attributes are independent, as such the assumption of attribute value independence PostgreSQL does is valid. However, this assumption is not valid when applied to the JOB benchmark, whose dataset is the IMDB database.

When Leis et al. run their benchmark on all the systems and gather the cardinality estimates, they discover that they all systematically underestimate the cardinality, which can be seen in figure 3.2. They conclude that all the systems assume attribute value independence and that assuming AVI is detrimental to the approximation accuracy. Their opinion of AVI is not theirs alone. It has been the opinion of research for a long time that AVI assumption is not doing estimation accuracy any favours. Both Poosala and Ioannidis

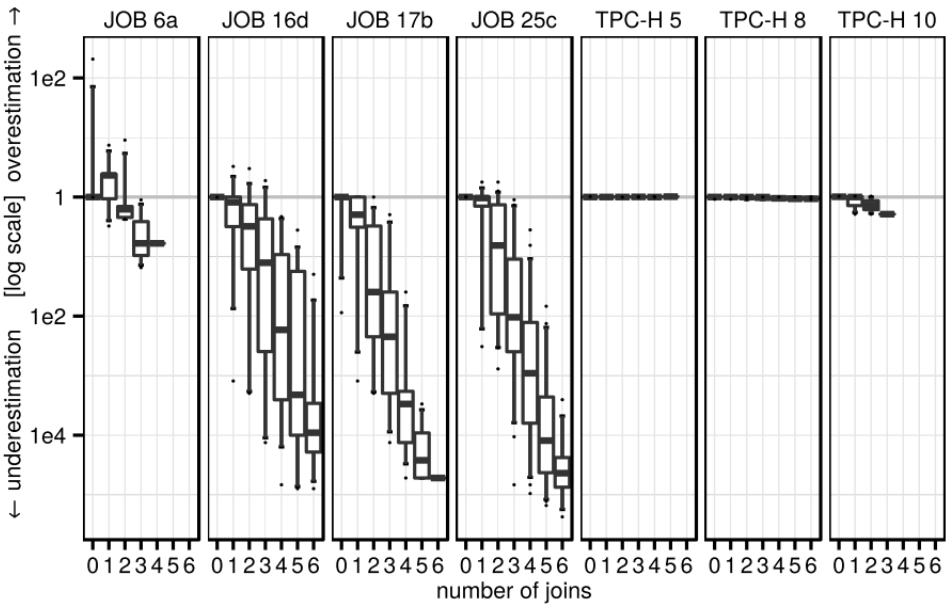

Figure 3.1: PostgreSQL cardinality estimate errors for JOB and TPC-H queries. Figure from [5].

in [16] and Swami and Schiefe in [33] conclude that assuming AVI should no longer be acceptable when estimating cardinality.

The effect of poor cardinality estimates is clearly shown in figure 3.3, where Leis et al. compare PostgreSQL's estimate of different plans execution times with their real execution time. One can see that the query-optimiser often believes plans will cost less than what they do and that many plans will cost the same. In the three rows, Leis et al. show the estimate created by three different cost models, in the left column, those cost models use estimates for cardinality, and in the right, they use real cardinality. One can also see that the difference in cost estimation accuracy is far more significant when switching from cardinality estimates to real cardinality compared to when switching from a crude cost model to a complex one. Cardinality estimates are often wrong, and when estimating cardinality for queries containing multiple join predicates, the error increases exponentially [5]. Leis et al. show that even with a straightforward cost model, the last row in figure 3.3, it is possible to create accurate cost estimates as long as the cardinality estimates used are accurate. They finish off their report by stating that ”cardinality estimation is much more crucial than the cost model”, based on what can be seen in figure 3.3 it would be hard to argue with them.

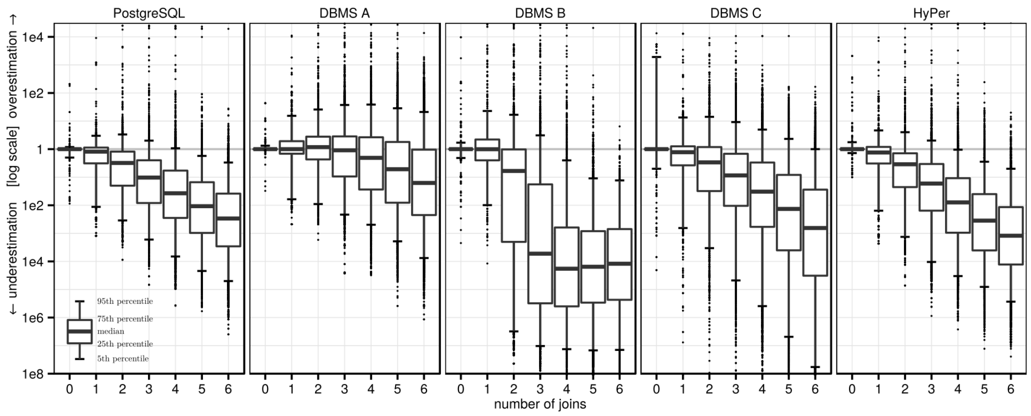

Figure 3.2: Error level of join cardinality estimation for each system. Each boxplot is a summary of all queries in the JOB benchamrk. Figure from [5].

3.5 Summary

It would seem like the significant modern DBMS' all agree that keeping histograms up to date is worth the overhead of computing them periodically, although they all use differing definitions for staleness without any explanation of their choice. For instance, Oracle database's definition of staleness as 10% of rows modified seems arbitrary; in their online documentation, they do not refer to why they chose 10%. We will, however, assume they have done some testing with different values and based on this testing, determined 10% to be a useful heuristic.

As discussed in subsection 2.2.5, we could not find any theory defining when a histogram is stale, and it was decided that the heuristics used by state-of-the-art systems can be considered good choices of heuristics, or at least to be a good starting point. We did; however, find that while Gibbons et al. do not define a value for staleness, they do find, through experimentation, that setting their γ variable to 0.5 is a *"reasonable value for limiting the number of computations as well as for decreasing errors"*. It is not attempted to speculate on how this can be translated into the heuristics that the above systems can use. During the tests Gibbons et al. performed, they were only attempting to achieve an optimal time for when to re-calculate their histograms. That time does not translate directly into the above systems; also a value that works well for Oracle database does not necessarily work well for MSSQL or PostgreSQL due to differences in internal structure and optimisation. Indeed, as we will show later a value that works well for one Oracle database might be a very poor choice for another Oracle database.

Leis et al. showed us how essential cardinality estimates are for the query-optimiser. We saw examples of simple query-optimisers outperforming more complicated ones by merely having access to accurate cardinality estimates. We also saw that if given accurate cardinality estimates, the current query-optimiser in PostgreSQL performed admirably. Improving the accuracy of cardinality estimates is then a source of a performance increase for almost all databases, which supports the idea of maintaining the accuracy of histograms

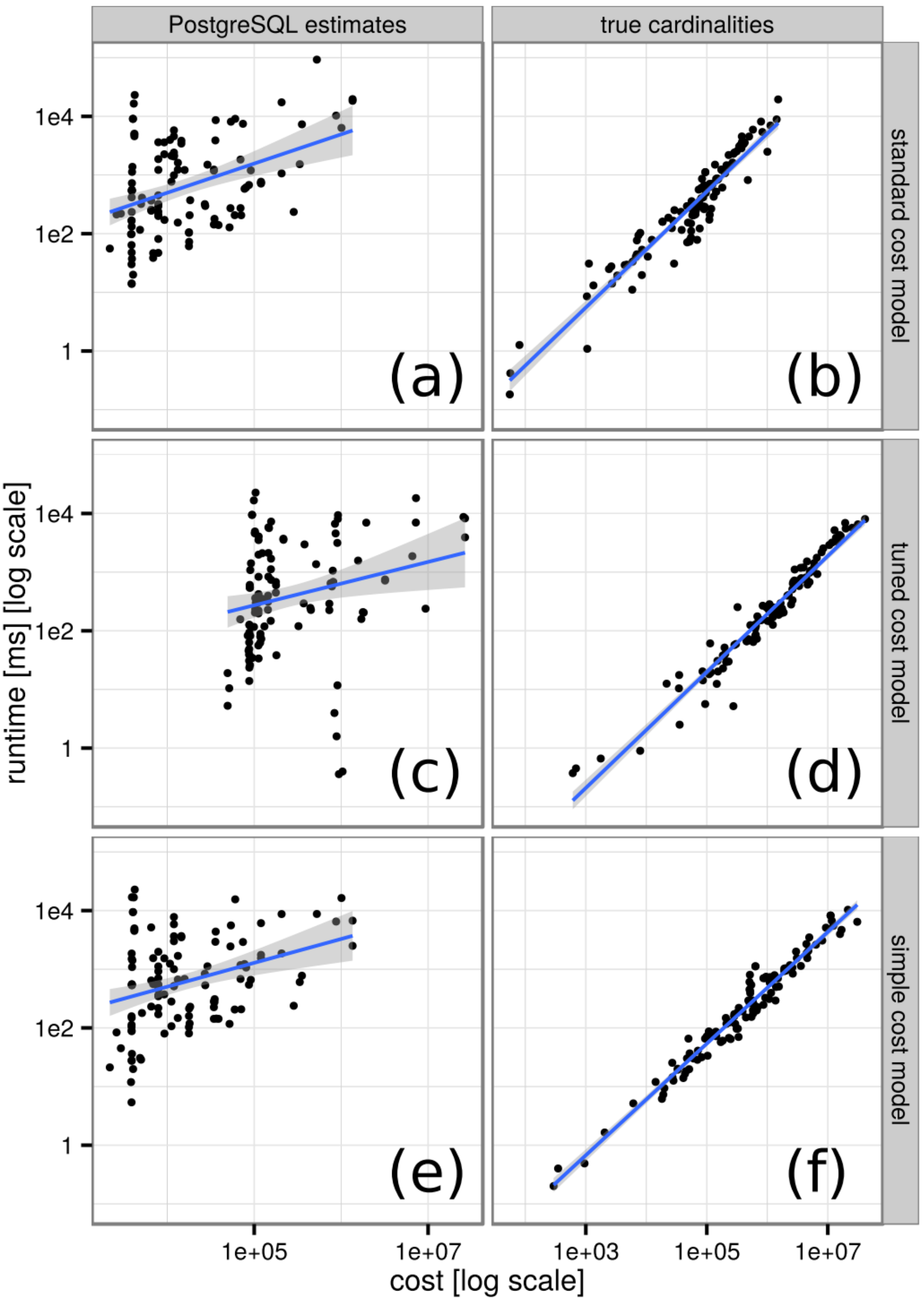

Figure 3.3: Estimate of query plan cost vs. runtime for different costmodels and cardinality estimates. Figure from [5].

such that they provide accurate selectivity approximations.

Chapter 4

Design

The goal of this report is to define; which attributes determine if a database will be affected by histogram accuracy, how that accuracy affects query-optimisation and use that to define guidelines for when and why histograms should be updated. To help us define these items, we will create a use-case and a histogram updater. The updater will handle updating of histograms, and the use-case defines the data model, queries and dataset in which the updater is tested. The updater will update histograms when they are determined stale by rules we create. In this chapter, we describe the design of; our use-case, the updater and the rules we have developed.

It was necessary to split our development efforts into two parts; first, we tried to discover in which scenarios histograms would affect execution times, and then in the second part, design and test updating rules for our histograms. In the first part, we test how the use-case, its data model, dataset and queries, will react to differences in histogram accuracy. The results from these tests are then used to draft the design requirements, which allows us to develop a use-case that would behave as required. The second part of development covers the development and testing of; the updater, the updating rules and the final tests.

4.1 Use-case

Development and testing has been centred around a use case to maintain a common thread to tie the testing, results and updating rules together, and link them to real-world cases. We want to determine how stale histograms affect databases in an actual real database. The use-case should, therefore, be designed in such a way that the accuracy of histograms will affect query execution times, while also remaining realistic. We do not want to design a use-case which feels contrived and is hard to justify. Achieving these requirements required some compromises, which will be covered later in this section along with different design decisions and requirements regarding the data model, dataset and queries.

4.1.1 Design requirements

The design requirements listed here are regarded to be more than just the design requirements for the use-case. They also list the requirements that must be met for systems, scenarios and queries if they are to be affected by histogram accuracy. Below, the requirements that have been discovered through evaluating the use-case of this project are presented. The list presented here serves as a summary only, and each point will be discussed further throughout this chapter.

- Histograms that are created but not maintained need to go stale at some point. We want to show how the accuracy of histograms affects the query execution time, that requires the accuracy of histograms to change when the underlying data is changed. The change in data must also cause the actual distribution of data to move away from the one approximated by the histogram, thus altering the accuracy.
- If we are to see any significant differences in query execution times due to histogram accuracy then the MySQL optimiser has to create different query plans based on the histogram. To do that the optimiser has to be able to utilise histograms in the optimisation. We discussed in section 2.3 that the MySQL optimiser can only use histograms when columns are compared to constant values. Meaning that if the optimiser is to use a specific histogram during optimisation, we need our `WHERE` predicates to use constant value comparisons against histogram columns.
- There has to exist multiple query plans for the query in question if the optimiser is going to have a choice of plan. More plans are preferred since more plans mean more potential for substantial differences in execution times. As discussed in section 2.3 having many different plans with different execution times is a characteristic of queries that contain several joins. Which means that if our queries have several join predicates that will be advantageous compared to having a single predicate.
- While the set of possible query plans to choose from clearly influence the choice the query optimiser will make, the size relationship between tables in those plans also influence the choice of plan. We uncovered that inter table size constraints had to be enforced if the optimiser was to choose different plans depending on query selectivity. This is caused by the heuristics the optimiser uses for different operators in `WHERE` predicates. Since there is a definite lower and upper limit to these heuristics, there are upper and lower bounds in table size difference required if we want every query plan to considered by the optimiser.
- We found that the optimiser was not able to use a histogram to determine the selectivity of the where predicate if it was also possible to use an index in the where predicate. This means that the design of our queries can not include; comparisons with values against columns that have indexes, in the where predicate.
- It was also discovered that simply having several query plans to choose from was insufficient; we needed to obtain query plans that resulted in large differences in execution times. Results showed that when the table with a histogram could be placed at either end of the join order that enabled plans with larger differences in execution times.

4.1.2 Use-case design

This section starts by designing a data model which will meet the design requirements laid out above, while trying to meet those requirements it is important to keep the model as realistic as possible. During use-case development, we tried a few different data models before we found one that would work. As stated in the design requirements, we found, among other things, that we had to carefully consider the join orders that the data model would support. The tests showed that if we organised the data model as a *star schema* that would produce plans with smaller differences in execution times when compared to a linear foreign key data model. Consider the following examples.

We create four tables, `A`,`B`, `C` and `D`, which have foreign key(s) from `A` to `B`, `A` to `C` and `A` to `D`. We then create a query that joins all tables together, the following join orders are then possible[1]. In the list(s) below a $\bowtie$ symbol means that we join the left operand with the right.

- $(((A \bowtie B) \bowtie C) \bowtie D)$
- $(((A \bowtie B) \bowtie D) \bowtie C)$
- $(((B \bowtie A) \bowtie C) \bowtie D)$
- $(((B \bowtie A) \bowtie D) \bowtie C)$
- $(((A \bowtie C) \bowtie B) \bowtie D)$
- $(((A \bowtie C) \bowtie D) \bowtie B)$
- $(((C \bowtie A) \bowtie B) \bowtie D)$
- $(((C \bowtie A) \bowtie D) \bowtie B)$
- $(((A \bowtie D) \bowtie B) \bowtie C)$
- $(((A \bowtie D) \bowtie C) \bowtie B)$
- $(((D \bowtie A) \bowtie B) \bowtie C)$
- $(((D \bowtie A) \bowtie C) \bowtie B)$

If we on the other hand create four tables, `A`,`B`, `C` and `D`, where the foreign key(s) are from `A` to `B`, `B` to `C` and `C` to `D`. And we then create a query which joins all tables together we will have the following possible join orders to choose from when optimising.

- $(((A \bowtie B) \bowtie C) \bowtie D)$
- $(((B \bowtie A) \bowtie C) \bowtie D)$
- $(((B \bowtie C) \bowtie A) \bowtie D)$
- $(((B \bowtie C) \bowtie D) \bowtie A)$
- $(((C \bowtie B) \bowtie A) \bowtie D)$
- $(((C \bowtie B) \bowtie D) \bowtie A)$

[1]Note that foreign key(s) are two way relationships, a key from `A` to `B` allows us to join `A` to `B` but also join `B` to `A`

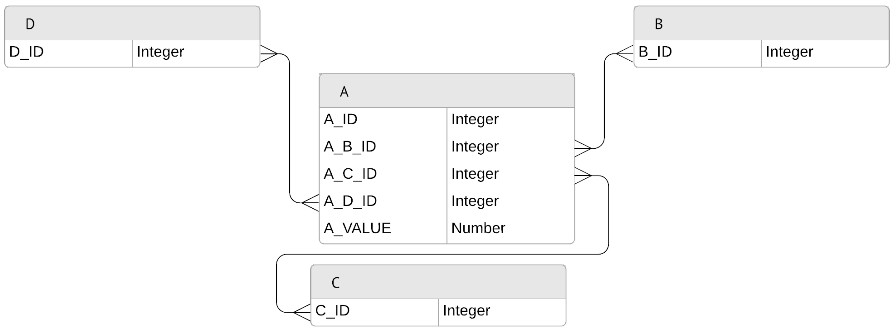

Figure 4.1: Star schema data model used in example

- $(((\texttt{C} \bowtie \texttt{D}) \bowtie \texttt{B}) \bowtie \texttt{A})$
- $(((\texttt{D} \bowtie \texttt{C}) \bowtie \texttt{B}) \bowtie \texttt{A})$

If we consider A as our ”central” table, i.e. the one that contains our time-series link and the column we wish to have a histogram on, we can see how these two examples are quite different. In the first one, A is either the first or the second table in all the possible join orders. There are no other options for when to join A; this would be the same if we had n tables. In the second example, A is the first in some orders, the second in others, the third and fourth in yet others again. Example one creates more plans than example two, but example two creates plans with the most substantial difference in join orders. Maximising the chances of having query plans with significant differences in execution times because, as we know from section 2.4, the order in which tables are joined together, is essential because it determines the size of intermediate results. The size of intermediate results in-turn determine the execution time of the join, and to some extent, the size of the following intermediate result, which determines the execution time of the next join and so on. It is therefore important that the choice of the first two tables is a good one, and even more important that we are *able* to make a good choice.

A linear foreign key data model allows the optimiser to start the join order with other tables than the A table and place it anywhere in the join order as long as the B table has already been included. ER-diagrams of the data models used in these two examples are shown in figure 4.1 and figure 4.2 respectively. When the models shown in these figures is reviewed, it is apparent that the star schema requires A to be in the first join, and that the linear foreign key model allows A to be in any of the joins executed so long as B is also present.

Our data model will be of the form shown in figure 4.2 where all foreign key(s) between tables are linear because this creates the largest difference in query execution time, which was part of the design requirements. We would like to point out that while it is possible to perform a join between two tables that do not have a foreign key relationship, we do not account for that behaviour in this project. Also when the term ”join” is used we are referring to inner-join, other variations of joins such as left-outer join and right-outer join

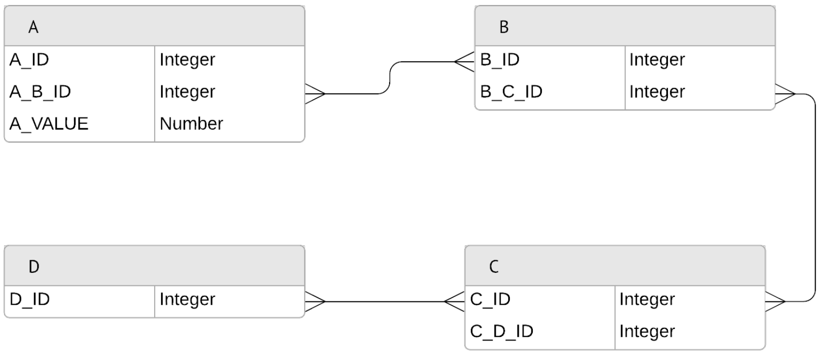

Figure 4.2: Linear foreign key data model used in example

are not considered in this project.

The queries in our use-case also need to meet the requirements presented in the preceding section. Primarily they need to be of a form which enables there to be several different query plans, and also the optimiser needs to be able to use histograms during query optimisation. Consider the following example.

We create four different queries which we will use to query data from the data model we chose above, these queries are;

```
SELECT * FROM A JOIN B ON A_B_ID = B_ID WHERE A_VALUE > SYSDATE;
SELECT * FROM A JOIN B ON A_B_ID = B_ID WHERE A_VALUE > X;
SELECT * FROM A JOIN B ON A_B_ID = B_ID JOIN C ON B_C_ID=C_ID JOIN D ON C_D_ID = D_ID WHERE A_VALUE > SYSDATE;
SELECT * FROM A JOIN B ON A_B_ID = B_ID JOIN C ON B_C_ID=C_ID JOIN D ON C_D_ID = D_ID WHERE A_VALUE > X;
```

The first and second queries join `A` with `B`, having only two tables there are only two possible join orders. Increasing the number of joins, such as in queries three and four, will increase the number of possible query plans and should increase the difference in execution time between the best and the worst plan. Queries one and three have a problem with their `WHERE` predicate; they are comparing our histogram column against a non-constant value, which means that the MySQL optimiser will not be able to use a histogram during the optimisation. Query four is the only one which meets all our design requirements, joining four tables together with the given data model gives eight possible join orderings during optimisation. Moreover, the value column is compared against a constant value `X`, which enables the use of a histogram during optimisation.

In the example above none of the queries included a column in the WHERE predicate that had an index, if any had done that would be in contradiction to the design requirements and would not be an acceptable query for us. We need to create queries that have several join predicates and at least one where predicate comparing a single column to a constant value and not have any columns in the predicate which have an index. Leaving us with having to create queries of the form of query four shown above.

Designing the dataset that will populate the use-case data model requires us to meet the design requirements outlined above, beyond those we also have to consider how the distribution of data should change over time. The inter table size relationship will, in some cases, impose restrictions on which query plans will be chosen. Consider the following example.

Let us fill the tables from the data model example above with an equal amount of rows, for instance, 10000 and let us set the value of A_VALUE to be 51 for all rows. If we then try to optimise, with and without histograms, the query shown below, the following will happen.

```
SELECT * FROM A JOIN B ON A_B_ID = B_ID JOIN C ON B_C_ID=C_ID JOIN D ON
 ↪  C_D_ID = D_ID WHERE A_VALUE > 50;
```

- When there is no histogram the optimiser will rely on heuristics to approximate the selectivity caused by the where predicate in our query. The heuristic that the MySQL optimiser will use for the $>$ operator is $1/3$, i.e. it assumes that the predicate will reduce the result set to be only 33% of the original data. Since all our tables are of the same size, this means that A becomes the smallest table and as such, it should be in the first join executed.
- If we perform the same optimisation again, but this time we have just created a histogram on the A_VALUE column we would hopefully choose another join order. The optimiser will use the histogram instead of its heuristic to estimate the selectivity of the predicate. Since all the rows in A have 51 in the A_VALUE column, the histogram is indicating that the cardinality of the result set will reduce by nothing, the filter leaves behind 100% of the original rows. Now all the tables are the same size after the predicate has been applied, and it becomes a bit difficult to predict which table the optimiser would choose as the innermost table in the first join. However, we know that if as little as one row had a value below 51, the histogram would indicate that some filtering would occur due to the predicate. In that case, A would be the smallest table, and it would be placed in the first join operation.

In this example, the optimiser chose the same two query plans, even when the histogram showed that the heuristic used in the first optimisation was off by a lot. This choice of query plan was because no matter what the selectivity of the WHERE predicate was, A was still the smallest table. To account for this sort of behaviour, we will have to take special care to create our dataset such that A can be both much larger and much smaller than the rest of the tables, based solely on the selectivity of the WHERE predicate. If we use the $>$ or $<$ operator in our queries that would then mean that our A table has to be smaller than the B table after we apply a filter of $1/3$ to A, or A has to be larger then B when a filter of $1/3$ is applied. Either way what we need is that the selectivity estimate of the histogram can cause a significant enough change in table size such that A goes from being the smallest to the larges, or vice versa.

There is one caveat to this we want to mention; however, it is not such that the query plan chosen by the optimiser based on the heuristic is always going to be the same as the one it would choose based on a histogram. That depends on the query being optimised just as much as it depends on the inter table size relationship. Nevertheless, in our case, we are only going to have a predetermined set of queries, which we will create and have full control over. As such, we need the optimiser to choose different query plans for *that* set as the filtering caused by the predicate changes. Therefore we need to carefully choose the size of tables such that we end up choosing different plans for the same query when we have, and when we do not have, access to histograms.

Further, we require the distribution of our histogram column to change over time, while also resembling a real-world scenario. To achieve the changing distribution we could, for instance, partition our data such that each subsequent partition has a slightly more skewed distribution than the previous. Allowing us to create our dataset such that the values in the WHERE predicate of our queries would match all rows at one point and then later match none of them. This would enable us to show how the accuracy of the histogram would affect the query plan chosen. A real-world scenario where the data is divided into such partitions is a time-series database. In a time-series database, queries are often concerned with only a subsection of the data; for instance, the data from the last ten hours, or last week. In that way, the entire dataset can be divided into partitions of ten hours or one week, and then queried on a per partition basis.

What we have outlined then is a dataset which adheres to the inter table size requirements we have laid out above, and also has a value in the histogram column which will change over time which gives the dataset a continuously variable distribution. When created in a computer program, we have fine-grained control over that distribution and its attributes, which will enable us to test how changing the distribution in different ways affects updating rules and histogram accuracy.

Closing remarks

This concludes the design phase of our use-case. In chapter 5 the actual implementation of the use-case is presented, we show how our data model ended up looking, what sort of queries we are going to use and what our dataset looks like. The actual values and attributes we have chosen for this were discovered in an iterative process, which we also present in that same chapter.

4.2 Different histogram updater architecture approaches

This report and the work presented in it is a continuation of [3], in which the histogram updater and accompanying theory were first presented. Since the updater used in this report is a continuation of the one developed in [3] the arguments for, and the decisions to, develop the desired functionality as it was, is the same as they were for the previous project. The discussion on this topic in [3] is included in this report as well for completeness sake. With necessary modifications made to reflect the change from a report centred around proving if automatic histogram updating in MySQL was required and indeed possible, to one primarily concerned with defining guidelines for histogram updating.

When trying to add features or functionality to vast and evolving software, several factors should be taken into account before we write any code. How challenging the implementation will be is, in many cases, influenced by how one attempts to implement the desired functionality into the system. We discovered four different ways in which we can implement automatic updating. We can either; create the desired functionality in SQL procedures, build the feature into the existing code, create a new program that works as an extension to the original, or create a plugin. Below, these four different possibilities are explored, and their pros and cons discussed.

4.2.1 Scheduled Job

Updating histograms in MySQL is done by a SQL statement. Since MySQL supports scheduled jobs, it would be possible to create the histogram updater as a SQL procedure stored on the server. This has the benefit of being a SQL procedure, which means that any MySQL database could "install" this new feature. The most significant drawback is the lack of access to statement metadata for the procedure. It would for instance not be possible to measure how many rows have received updates since the last histogram was created [2]. Nor can we measure the number of statements since the last update. Limiting the implementation to use a simple rule for staleness, such as a timer. I.e. could the procedure run every one hour and update all histograms, or maybe once every day and update all histograms. The lack of statement metadata makes it difficult to try out different rules for when to update histograms. Developing the histogram updater with that type of restriction intrinsic in the architecture is not a good idea.

4.2.2 Built-in

Developing the histogram updater as a built-in set of functions and procedures gives access to any function available in the MySQL internals. This level of access means that the updater, and by extension, the updating rules, can be as complicated as desired. It does, however, presuppose comprehensive familiarity with the already existing internals of MySQL. We will, for instance, have to choose where in the life cycle of a statement we execute the histogram updater statements and determine which histograms to update. This type of approach changes how MySQL parses and executes statements, an essential

[2] At least not without introducing significant overhead in the form of additional monitoring of the database, through temporary tables and the like.

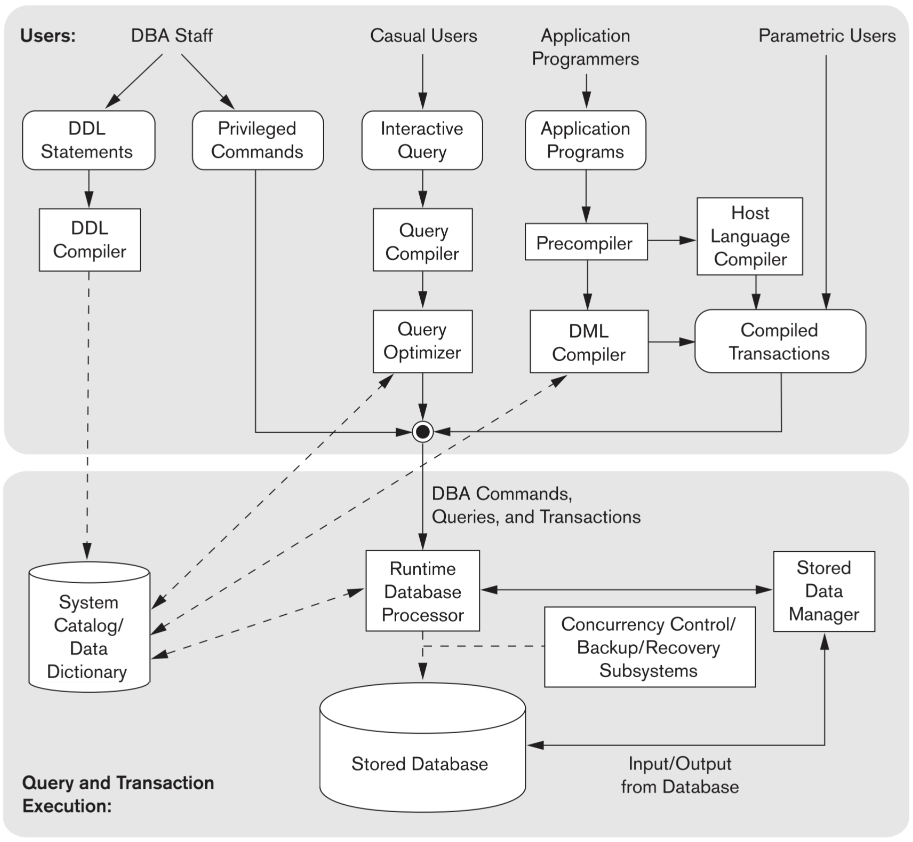

Figure 4.3: The general architecture of statement execution. In the middle we see how statements enter, are parsed and then optimised before they are executed on the database. Figure from [4].

part of an RDBMS, that we do not wish to disrupt or alter unnecessarily. In figure 4.3, the general architecture of a statement and its path through the database system is depicted. In our case, we would have to alter the query compiler and query-optimiser objects in this figure. The level of knowledge required to implement the updater in this fashion, without introducing errors or bugs, necessitates considerable training which is not possible in the short time available.

4.2.3 Extension

As an extension the updater is a completely standalone service running in conjunction with the standard MySQL server application. Developing in this fashion gives the developer a choice of programming languages; the developer could use any software they are comfortable with. The drawback is that the extension is not capable of accessing anything happening inside MySQL. The extension would only have access to data available through the MySQL API and through SQL commands on the server itself. However, if we

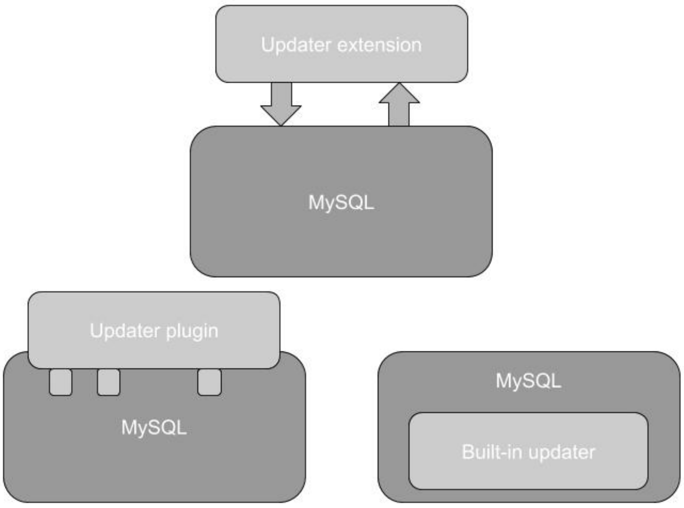

Figure 4.4: Visual representation of three of the different architectures discussed.

develop the extension as middleware and we let users connect with the extension rather than MySQL, that would enable the recording of statement metadata within the extension. For instance, the number of queries and active users on the database could be monitored and stored. This type of extension would require several features that exist in MySQL to also be present in the extension, such as session handling and user validation. Developing all these features again would require extensive work, and forcing all users to connect through a middleware can be unpopular and difficult to enforce. Meaning that a middleware type of extension can be difficult to achieve successfully. If it is not implemented as middleware, the extension alternative ends up sharing several problems with the scheduled job strategy with regards to statement metadata and information access.

4.2.4 Plugin

As a plugin, the histogram updater has access to functions and procedures within MySQL during execution. Meaning that we can use selected MySQL functions and procedures without having to know how the rest of MySQL works. As a plugin, the histogram updater acts as a service installed from inside the database by initialising the plugin using a procedure. MySQL invokes all installed plugins at different times during statement execution. The hooks in MySQL activate the histogram updater plugin so that it can execute before MySQL continues execution. There are hooks both before and after a statement is parsed, besides, we can add new hooks at almost any point during statement execution. Giving the option of choosing when during the execution of the statement, we wish the plugin to start.

4.3 Plugin workflow

In the end, we chose to develop the histogram updater as a plugin due to our need to create flexible rules for staleness and the limited time available for design and implementation. Also, recommendations and discussions with the supervisor and other groups of students at Oracle in Trondheim resulted in the plugin architecture winning out. For another relevant discussion of the different system types, the reader is referred to [34] by Hole and Eggen, a team of students at Oracle's office in Trondheim, that also developed a plugin for MySQL around the time we did.

The histogram updater plugin is intended to update histograms for columns when they are marked as stale by the plugin itself. Below we describe the proposed design for the plugin and its workflow.

The plugin is installed with its set of rules built-in, but can only use one at a time. We want to be able to choose which rule to use during runtime; we achieve that through the use of a global variable. During execution, the plugin determines which updating rule should be enforced based on that variable. One particular value is designed such that the plugin does nothing; it allows us to install the plugin and use the database as usual without the plugin enforcing any rules.

When a statement executes on the database, hooks in MySQL should activate the plugin after the statement has is parsed. When the plugin activates, it will determine the type of statement and the columns and tables effected by the statement the user sent. Different types cause different actions within the plugin. Statements that result in modification of data; inserts, updates and deletes, also known as DML statements, can be handled in two ways. Either the plugin checks the active rule before the statement has executed or after the statement has executed. In the first approach, the effects of the statement are not yet known and must be approximated by the plugin. If on the other hand, the plugin checks after the statement has executed the effects of the query can be known. For both alternatives, the columns the rule marks as stale should have their histograms updated by the plugin, and their status reset.

The second of these two approaches requires there to be a hook in the plugin API which will execute plugins after statements have completed, and provide the plugin with meta-data about the effects of the statement. At the time of writing this report, such a hook does not exist, and we have chosen not to implement it either because the functionality that it provides can be emulated. I.e. we know what the effects of inserts, updates and deletes will be before they execute. As such, we can emulate the behaviour of analysing the effects of DML statements by knowing the effects before they execute and then use that as if we had analysed the effects.

When the plugin receives a query and not a DML statement no action should be performed, and the plugin should return control to MySQL, there is however another way in which queries can be processed. If we let DML statements work as described above but with one small change, they should only *mark* histograms as stale, but not *update* them. That would allow us to use queries as triggers for when to perform the actual update for histograms marked as stale. This would save on computation since it would be possible to mark a histogram as stale multiple times and only compute a new histogram when it is needed. Marking a histogram as stale is in this case considered to be a very cheap operation, in most cases rules will only contain a few `IF` checks and some updating of counters.

We chose not to design the plugin in this way, because our tests will perform many inserts after another and then a bulk set of queries followed by another set of inserts followed by a set of queries. This goes on until we have no more data to insert; the test is then finished. In such a case, this alternative design would effectively remove the negative sides of several rules we plan to test. The goal of this report is not to create an effective plugin nor is it to create effective rules; it is to explore the attributes, both good and bad, and effects of different updating rules in general situations. As such, we do not want to remove the adverse effects of rules in our dataset, which might not be removed in the general situation.

4.4 Updating rules

Below we list the set of updating rules we have designed. With each rule, we present how it is intended to work, why we believe the rule is a good candidate for testing and implementation in the plugin, and its pros and cons. In this section, when we use the term histograms, we only mean those who would have been affected by a given operation. I.e. an update on a table, or column, with no histograms will not invalidate any histograms no matter the rule. Neither will a DML statement which can not cause a histogram to be invalidated. I.e. an update on columns `T1` and `T2` in table `A` will not affect the validity of a histogram on column `T3` in table `A`.

The ultimate goal of any of the updating rules is to maintain a sufficiently accurate histogram without invoking significant computational overhead in the process. It is trivial that the most strict rule would be to update histograms after every statement. There is, however, a significant overhead associated with this approach, so much in fact that it quickly becomes infeasible to enforce this rule. Any other rule than the trivial one must sacrifice the guarantee of accuracy in favour of reducing overhead; a good rule should avoid the histogram becoming or staying stale for long periods while also achieving substantial reductions in computational overhead. What this means then is that the rules presented below are actually varying *definitions* of histogram staleness. If one tried to create a theoretical definition of histogram staleness, it would be natural to state that, *once the distribution of the data has changed the histogram is stale*. While that definition works in theory, it does little good in practice. The ”strict rule” presented just above is, in fact, an approximation of this definition, ”once the data has changed the histogram is updated”. For our case, we can not use this definition; it defines histograms as stale too often, resulting in too many resources spent on computing histogram. Instead, we present a series of rules which define staleness as *when the histogram is causing the query-optimiser not to choose the optimal join order*. They approximate the point when the plan should have changed due to a significant change in distribution and define the histogram to be stale at that point.

Rule one, after each DML statement

Updating histograms after every DML statement is the trivial rule. As briefly mentioned above, it will keep a histogram perfectly accurate at all times, but with that accuracy, there is also a large computational overhead penalty. Not all DML statements lead to a row being updated, for instance, `UPDATE A SET A_VALUE = 0 WHERE 0 = 1;` will not update any rows of `A`, but it would still invalidate the rule and cause the histogram to be updated. The computational overhead associated with this rule is so large that enforcing such a rule is not possible. If such a rule could have been used, it would already have been implemented in MySQL. This rule is useful in another manner; however, since it ensures that all queries are optimised with accurate histograms, we can use the query execution times that this rule provides as the ”best case” times for queries. If we can manage to implement a which gurantees that all queries are optimised with perfect histograms, then that rule can be used as a baseline to compare performance against for all other rules.

Pros	Cons
Perfect histogram accuracy	Unnecessary resource consumption
Easy to implement	Large impact on DML performance
—	Large impact on database performance in general
—	Counting DML statements is an inaccurate measure of data change

Table 4.1: The pros and cons of rule one

Rule two, after n DML statements

A variation of rule one where instead of updating after every single DML statement we let n statements execute before we update the histogram. Reducing the overhead by a factor inversely proportionate to n, but also reduces the time in which the histogram is perfectly accurate by the same amount. As with rule one, this rule also suffers from the fact that a DML statement does not necessarily cause any data to change. Furthermore, perhaps even worse, a single DML statement can change all the rows of a table. Thus DML statements can change anything from zero rows to all the rows of the table.

Pros	Cons
Easy to implement	Difficult to choose the optimal value of n
Easy to tune updating performance by changing n	Possible large impact on DML performance, depending on choice of n
—	Counting DML statements is an inaccurate measure of data change

Table 4.2: The pros and cons of rule two

Rule three, after n DML statements also considering statement type

There are three different types of DML statements, insert, delete and update. One fact that is of particular interest to us is that both the insert and delete statements change the distribution of values in another way than the update statement does. Also, these two statement types alter the total amount of rows in the table; this might affect distribution in ways we might not consider originally. For instance, if a delete, or insert, statement affects the same amount of rows in each of the different buckets in the histogram, then the distribution is the same as it was before, but the number of elements has changed. This does not affect the accuracy of the histogram; however, as it computes percentages and does not return the actual number of rows. However deleting the same amount of rows in each bucket is quite unlikely, and we do not expect this to happen often. Since the statements are different and they affect the distribution in different ways there should be some difference in accuracy between a rule which accounts for this and one that does not. Rule three is meant to give different weights to different statement types and through that account for the differences between them.

Pros	Cons
Easy to implement	Difficult to choose the optimal value of n
Easy to tune updating performance by changing n	Possible large impact on DML performance, depending on choice of n
Increased tuneability as result of also considering DML type	Counting DML statements is an inaccurate measure of data change
—	Difficult to choose weighting values
—	More parameters to choose from increase possible number of combinations, making testing all combinations more time consuming

Table 4.3: The pros and cons of rule three

Rule four, after n DML statements, triggered by queries

Another relatively small change to rule two creates our fourth rule. We still count DML statements, but we do not update the histogram until the database receives a query which uses the histogram. We covered this sort of behaviour as a possible workflow design during our description of the plugin design above. Choosing to only mark histograms as stale and postpone the actual updating until a query needs the histograms removes the problems associated with bulk DML statements, it is also very similar to the MSSQL udpating scheme. However, it also means that the first query will have to be optimised with a stale histogram since we do not want to halt query execution to compute the histogram. Since this would mean that a user would be kept waiting for the amount of time it takes to compute the histogram before the user's query even begins execution.

Pros	Cons
Easy to implement	Difficult to choose the optimal value of n
Easy to tune updating performance by changing n	Counting DML statements is an inaccurate measure of data change
Negligable impact on DML performance since a query is required to cause histogram updating	—
Choosing a small value for n is no longer as bad as it used to be	—

Table 4.4: The pros and cons of rule four

***Rule five*, after n rows are updated**

If instead of measuring the *number* of DML statements, we measure the *effects* of DML statements, we remove the problems we have had with not knowing if rows change and how many were changed. This way, we are more accurate in our measurement and hopefully will not spend resources on updating histograms when no rows have changed. There are, however, still some issues; if we update two rows in such a way that they swap values, effectively making each row equal to the old version of the other one. That still counts towards the number of updated rows even though the distribution has not changed. While this presumably will not happen very often and should not be considered a big problem, it is a source of inaccuracy and is therefore mentioned here. The outlined approach for this rule also requires an analysis of the effects of DML statements, meaning that meta-data about statements needs to be available. Another problem is that the histogram will updated after n altered rows no matter how many rows there are in the table, meaning that updates may be performed when only a fraction of the table has changed.

Pros	Cons
Easy to tune updating performance by changing n	Difficult to choose a ”good” value of n
Direct link to rows changing	Possible large impact on DML performance, depending on choice of n
—	Requires meta-data about the effects of DML statements
—	Rows can be updated without changing the accuracy of the histogram but the rows will still count
—	The ratio of changed rows to table size is not considered

Table 4.5: The pros and cons of rule five

Rule six, after a ratio r between table size and updated rows is reached

The sensitivity of a histograms accuracy is very dependent on the number of entries in the underlying variable. If for instance, a column contained ten values and five of those were deleted 50% of the dataset the histogram is based on was just removed. That probably means that the histogram is no longer accurate. If however, the column contained 1000000 rows deleting five of these will only change 0.000005% of the dataset, now that is probably not enough to alter the accuracy of the histogram significantly. Rule six tries to account for this by weighting the number of updates against the size of the table. In other words, rule six checks the percentage of changed rows and triggers an update based on that exceeding a certain ratio of change r.

Pros	Cons
Easy to tune updating performance by changing r	Difficult to choose the optimal value of r
Direct link to rows changing	Possible large impact on DML performance, depending on choice of r
The ratio between changed rows and dataset size is considered	Requires meta-data about the effects of DML statements
The parameter r is self explanatory and easy to understand	Rows can be updated without changing the accuracy of the histogram but the rows will still count

Table 4.6: The pros and cons of rule six

Rule seven, after n rows are updated, comparing the change against histogram boundaries

The accuracy of a histogram is not equally sensitive to all changes to the base dataset. One could argue that a change to the dataset, which is outside the measurement range of the histograms affects the histogram relatively more than one which is inside the range. I.e. a change which causes a value to end up outside the range of the histogram is worse than one which only changes it to another value inside the histogram range. This is because the possible values of the underlying data increases and the histogram is not changing to reflect that. While a variable which is altered inside the histogram might move in such a way that it actually increase accuracy. If the rule tracks the boundary values of the histogram, changes to data compared against those boundaries can be made. If the change falls outside the range, the change should count as being more important. In this way, changes to the range of possible values becomes more important than changes to the proportion of values inside the range.

Pros	Cons
Easy to tune updating performance by changing n	Difficult to choose the optimal value of n
Direct link to rows changing	Possible large impact on DML performance, depending on choice of n
Some changes are considered more important than others	Difficult to choose a ”good” weighting value
—	Rows can be updated without changing the accuracy of the histogram but the rows will still count
—	The ratio of changed rows to table size is not considered

Table 4.7: The pros and cons of rule seven

***Rule eight*, comparing updates against a sample table**

It is difficult to know which changes a given statement will result in and how those changes, in turn, alter the distribution of the dataset. Until this point, the rules have made some simplifications which enable them to approximate that distribution change. A more accurate way to determine the change is by having a copy of the table and applying the incoming statement to that copy and then check if the distribution changed as a result. This is unfortunately not feasible as the associated overhead would be far too great; one would be in essence be doing every operation twice, once on the base table and once on the copy. However, it is possible to apply the statement to a sample of the table. On could then accurately measure what the effect of the statement(s) was on the sample table and extrapolate those effects out to the base table. Then determine whether or not the histogram requires updating by comparing the table distribution of the copy with that of the histogram. Gibbons et al. perform a similar sampling and updating technique to the one we described here in their *Fast incremental maintenance of approximate histograms* paper, [18]. They maintain what they call a backing sample by sampling the base table once, and then applying incoming statements to that sample. In their case, they are not only using the backing sample to determine when to update the histogram, but the backing sample also serves as the dataset from which the new histogram is computed, this significantly reduces histogram computation time.

Pros	Cons
Accurate rendition of changes made to the distribution	Complicated to implement
In theory histograms are only updated once they actually deviate *enough* from the real distribution	Can be difficult to define what ”*enough*” is

Table 4.8: The pros and cons of rule eight

Rule nine, after the estimated cost of an inaccurate histogram exceeds the limit I

We have seen how there can be several factors that should be taken into account when trying to determine if a histogram requires updating. Rule two showed how different DML statements could be weighted differently and rule six exemplified that table size should be a strong contender for determining the importance of a single DML statement or a single row changing value. It might be that a rule should account for these things together. Rule nine is meant to provide a function where each update or general data change is divided into several components where each has an associated weight parameter. These parameters can then be tuned to optimise the rule for different scenarios. We intend to include components such as; table size, statement type $w_{i,u,d}$, histogram boundaries w_{oh}, and the importance of accuracy I.

$$r \times (a \times w_i + b \times w_u + c \times w_d + d \times w_{oh}) > I \tag{4.1}$$

An example of the rule's formula is shown in Equation 4.1, the values, a,b,c and d correspond to the number of statements of a given type that have been recorded since the last histogram update. Where a is the number of insert statements, b is the number of update statements, c is the number of delete statements, and d is the number of rows that have values outside the current histogram range. The weighting given to these different counters is then; w_i, w_u and w_d to the three different types of DML statements, `INSERT`, `UPDATE` and `DELETE`, and w_{oh} signifies the importance of a data change happening outside the histogram boundaries, and it is the weighting given to d. r is the ratio between altered rows and table size, and finally, I is the importance of accuracy. As data is changed the sum of values on the left of the $>$ operator accumulate until it exceeds I. At which point the histogram will be invalidated, and the counters will reset to zero. A low value for I would then cause the histogram to be invalidated more frequently, in other words, I is the inverse sensitivity to change, the more insensitive we wish the rule to be the higher the value we have to choose for I. The weighting values and the inverse sensitivity to change are meant to be tuned to each specific scenario to maximise the efficiency and accuracy of the rule.

This rule retains some of the problems the previous rules had with regards to choosing values for its parameters. While gaining new ones because now we have to define values for much more than just one parameter, and we are not sure how these parameters affect each other or the performance in general. However, it combines what we believe to be advantages from some of the simpler rules into a single rule.

Pros	Cons
Can account for many different components which influence the importance of a data change	Time consuming to choose values for weights
Adjustable to different scenarios	Could be very difficult to choose optimal values for weights
Easy to implement	Difficult to predict how the values given to different parameters will influence each other

Table 4.9: The pros and cons of rule nine

4.5 Summary

In this chapter, we presented and discussed the design of; our use-case, our plugin and a set of rules. We showed that the design of our use-case is a data model with linear foreign key relations between tables, queries with varying numbers of `JOIN` predicates, simple `WHERE` predicates carefully constructed around critical values in the dataset. Moreover, it has a dataset with; strict inter table size constraints and a column with a varying value that changes as ”time” progress and spans the values in our queries `WHERE` predicates. We also showcased and exemplified a set of design requirements that have to be met if the accuracy of histograms is to have a profound effect on query execution time. Finally, we presented the designed workflow of the plugin we will implement before we discussed and showed a set of rules which can be used to maintain histograms.

In chapter 5 we present the implementation and evaluation of our use-case. We continue in chapter 6 with the implementation of the plugin and its associated rules

Chapter 5

Use-case implementation and evaluation

During the beginning of our development efforts, we evaluated and implemented a few different use-cases. In the first part of this chapter, we present the process of implementing, testing and evaluating those use-cases. In the second part, we summarise how those efforts lead to the final use-case and what that final use-case is.

5.1 Background and the first development efforts

When deciding on how to test the plugin and the different rules, what metrics we should measure, and how we should define our dataset and queries. We found that we needed a real-world use-case where we could try to improve performance rather than creating a case in which our solutions *would* improve performance. In section 4.1 we presented the design of this use-case, during the presentation we referred to results from use-case implementation leading us to reevaluate design choices. Those tests, their actual results, how those results influenced decisions regarding the use-case, and a general description of our testing suite is presented in this section.

5.1.1 Implementing different use-cases

Starting with tests as early as possible was important to us from the beginning, it would give us the ability to check early what would work and what would not. It proved to be critical to our development efforts, as these tests revealed several flaws in our initial use-case designs and enabled us to make the necessary changes. The design of our dataset, queries and data model was an iterative process, where the results derived from measuring the execution times of one set of queries caused us to change the design of one or more of our use-case's sub-components. These results and their corresponding tests and development efforts are grouped into this first implementation and evaluation chapter, and they are presented in the paragraphs below.

Creating the data model of our use-case involved experimenting with different table, column and foreign key structures. We knew from our discussion in section 2.3 that we would need to be able to join tables together to create as extensive a set as possible of different query plans to choose from. All the data models we experimented with had several tables with foreign key relationships between them. That would allow us to create numerous queries with several join predicates, they did not, however, result in the same number of interesting join orderings which would prove to be vital.

Initially, we created two different data models; the first was designed to model a payment system, where payments would arrive at different points in time moving money from one account to another in exchange for a product or service. And then a system which would measure and log temperatures across geographic regions and countries. Temperature measurements would arrive at different points in time, and they would be linked to different measuring stations in different locations. We did not see any significant differences in the results between the two models. We, therefore, went back and studied the join orderings the models allowed for. When we did that we uncovered that while it was possible to join the tables in different ways, it always involved joining the ”central” measurement table to the others in different orderings. Because our data structure was set up with the measurement table in the middle with one or more foreign key(s) to all the other tables[1]. There were no join orders which allowed us to place that table anywhere else than as either number one or two in the join order. Changing the join order of our central table from the first to the second place is not a massive change. As we had discovered, we were having trouble seeing any significant differences in execution time as a result of this plan change. What we needed was for the optimiser to be able to swap the ordering of the measurement table from the first to the last position - making as substantial a difference as possible. To achieve this flexibility in join order, we needed to have a model in which any of the selected tables could theoretically be the first one in the join order for that query. We found that a successive set of foreign keys gave us the best possibility for creating comprehensive sets of possible join orderings with regards to a ”main” table.

We found that even though a linear foreign key relationship provided *fewer* join orders than a star schema of equal size, it still provided us with more significant differences in execution times. That is because it is more critical where the ”central” table can be placed in the order than how many orders there are in general. The reader is referred to the example showcasing this behaviour in subsection 4.1.2 for a more thorough description of how those join orderings are generated and how they influence execution time.

Developing a dataset which would suit our needs turned out to take up a significant amount of our development efforts at the start. We first had to set up an environment in which we could create a dataset with any attributes we desired. We wanted to use Python[35] to do this since it is a language with which we were familiar, and there is a large number of extensions to it which provide the language with added functionality which we found very helpful. The NumPy[36] extension enabled us to create datasets of arbitrary size efficiently, and with distributions and attributes, we desired. Pandas[37]

[1]This is commonly referred to as a star schema.

provided us with data manipulation and ways to handle our data in frames, similar to how tables work in RDBMS'. It is also well integrated with the Seaborn[38] visualisation library, enabling us to present our results in lucid and informative plots. After we managed to create datasets of arbitrary size using Python, we had to figure out what attributes the dataset needed to have. We knew that we needed to be able to control the distribution of data at any point so that we could make histograms stale when we wanted to. However, there were yet some attributes that we did not know we needed.

To start with the dataset was intended to be loaded into the database and then swapped out as time progressed and we required the distribution of our histogram column to change. However, we realised that this was not very realistic and would not fit any use-case we tried to create. We, therefore, decided that the changing distribution should be caused by values that change over time. For instance, let us say we measured air temperature. During the day temperatures would rise and at a point reach maximum, probably in the evening once the sun has been warming all day. Then as the sun sets the temperature will decrease until the sun rises again in the morning. If we then partition the day into 24 periods of one hour each, we would see quite clearly that the distribution within those 24 different partitions is different. The example just described can be implemented as a time-series database, it would be a simple one but a time-series database none the less. If we then create our dataset such that the values change as time progresses just like in the example and then partition it up into bite-sized chunks of equal size, we could imitate the behaviour of the above example. We, therefore, created a sequential import of data where the dataset was created with a changing distribution as one progressed over the rows. In figure 5.1, we see a plot of how the values in the final dataset are meant to shift as we go from one measurement to the next. In the example shown below, 100 measurements are taken every second, and a total of 500 minutes pass from the time the first measurement is made to the last.

In figure 5.1 we can see that the recorded temperatures increase and decreases as "time" passes and load on the servers change. We predict that this property will cause histograms created on a previous set of data to be inaccurate and cause the optimiser to choose a poor plan.

We experimented with different ways of loading the different partitions into the database. We tried using the built-in `LOAD DATA IN FILE` MySQL function by outputting the partition in our dataset as separate CSV, *comma-separated-values*, files. This made loading data quick and easy, but also not very realistic in our scenario. Also, this sort of "bulk insert" would limit our possibilities when it came time to implement the updating rules. We, therefore, decided to create separate files for each partition, such that we could load one partition into the database at a time. Instead, of making them CSV files, we created insert statements in Python and exported that as .txt files. Using regular insert statements gives us more control and options for the implementation of updating rules. It is also more closely resembles how data in many systems is modified, added or deleted.

Next, we discovered that sometimes the optimiser was not changing query-plans when a histogram was available, but instead choosing the same query-plan no matter how we skewed the distribution. In figure 5.2 we can see that even though the values in the column are heavily skewed to one side, which should result in different query plans when the optimiser has access to histograms, we saw no significant difference in execution time

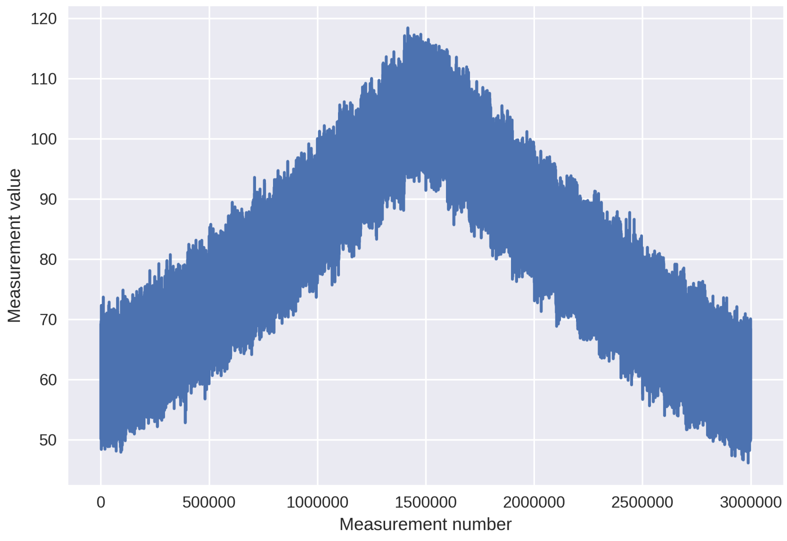

Figure 5.1: An example of how we shifted the temperature as ”time” progressed for some of the tests performed during use-case implementation.

when using this dataset. Our query-plan data also corroborated this, the plans were the same for the different runs[2]. It turned out this was caused by the sizes of the tables in our dataset, or to be more precise; the difference in size caused it. Because the `SERVER` table had as many rows, 10 000, as the `MEASUREMENT` table, that caused the `MEASUREMENT` table to be the smallest table when the predicate was applied, both when the optimiser could use a histogram and when it had to resort to heuristics.

The problem of table size ratios is one that can impact our queries and their execution plans in several ways and has caused us to implement rules regarding inter table size ratio. These rules make it ”possible” for the optimiser to choose different plans when a histogram is available. For instance, if the `MEASUREMENT` table is more than three times as large as the `SERVER` table, the optimiser will choose to join the `MEASUREMENT` table last when the query contains a $<$ or $>$ sign in the where clause. If the histogram then shows that the actual returned result from the measurement table is much larger than $1/3$, the optimiser will also place it last. Thus the histogram made no difference to the query-plan for this example query. In subsection 4.1.2 a general example showcasing this behaviour is explained.

It became clear that we had to restrict the size of the `MEASUREMENT` table as we were inserting more rows; otherwise, it would increase in size as our tests executed. The exe-

[2]The query used here was the same as the one showed in listing 1 but the plans are not the same as listing 2 or listing 3, however, they did look very similar.

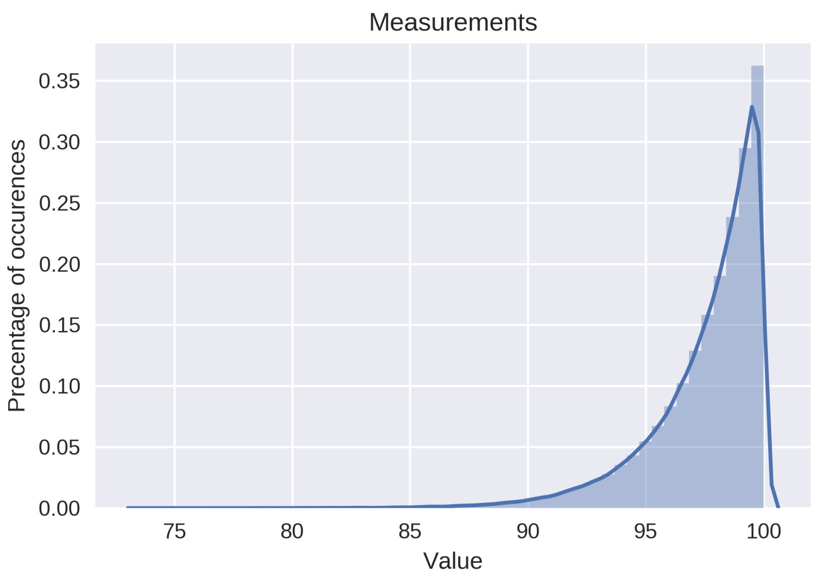

Figure 5.2: The distribution of values in the `MSM_VALUES` column used for some of the tests performed during use-case implementation.

cution time of our queries would then be influenced by other factors than the histogram[3]. To achieve this, we delete the oldest row in the `MEASUREMENT` table as we insert a new one once we have reached the table size we want. We omit the delete statement from the first partition of our data, of course, or our table would be empty. This means that the size of the `MEASUREMENT` table will stay constant at the size of our partitions. It might not seem like a very realistic scenario to delete the oldest data upon inserting new data, but consider the following situation; a time-series data warehouse has a database which stores all of the rows. What happens when the system has been running for a long time and continuously kept on receiving information? They can not possibly store everything forever; at some point, the data warehouse will have to start deleting old rows to make space for new ones. This is a common situation for many time-series data warehouses. In our case, we only start deleting old rows earlier. We choose to keep the number of rows constant to mitigate the number of unknown factors influencing query execution time. Therefore we only delete a row when we have inserted a new one to ”take its place”.

We tried skewing the distribution of the `MSM_VALUE` to various degrees to see how extreme the skew had to be for there to be a large discrepancy in execution time. We found, to no surprise, that when the selectivity provided by the histogram was large enough, or small enough, to change the size of the `MEASUREMENT` table noticeably. Namely such that when compared to the other tables in the query it is either smaller or larger than the others then the query plan would change to reflect that. Below is part of the results we found

[3]Reducing the number of unknown variables that could influence query execution times would turn out to be something we were considering continually throughout development efforts.

during the development of our use-case. When listing 1 was executed the MEASUREMENT table had 100000 rows while the two second largest tables had 33000.

```
select * from test.measurement join test.server on
 ↪  test.measurement.msm_serv_id = test.server.serv_id join test.rack on
 ↪  test.rack.rack_id = test.server.serv_rack_id join test.center on
 ↪  test.center.cent_id = test.rack.rack_cent_id join test.city on
 ↪  test.city.city_id = test.center.cent_city_id where
 ↪  test.measurement.msm_value >80;
```

Listing 1: An example of the queries we use in our tests. This query has four join predicates and a where predicate requiring msm_value to be greater than 80. Later in the report, we use a shorthand notation for our queries, we would format this to *query with four joins and WHERE MSM_VALUE > 80.*

```
-> Nested loop inner join
    -> Nested loop inner join
        -> Nested loop inner join
            -> Nested loop inner join
                -> Filter: (measurement.msm_value > 80.0000)
                    -> Table scan on measurement
                -> Single-row index lookup on server using serv_id
                 ↪  (serv_id=measurement.msm_serv_id)
            -> Single-row index lookup on rack using rack_id
             ↪  (rack_id=`server`.serv_rack_id)
        -> Filter: (center.cent_city_id is not null)
            -> Single-row index lookup on center using cent_id
             ↪  (cent_id=rack.rack_cent_id)
    -> Single-row index lookup on city using city_id
     ↪  (city_id=center.cent_city_id)
```

Listing 2: Result of using EXPLAIN on listing 1 when there is no histogram on MSM_VALUE.

We can see in listing 2 that the optimiser has chosen to join the MEASUREMENT table with the SERVER table first. That is because the optimiser assumes the WHERE predicate will cause only 33% of the rows to be selected. That is not correct; however, as what returns is closer to 99.8% of the rows. The predicate removes only 0.2% of rows, and the MEASUREMENT table should have been joined as late as possible. If we review the plan shown in listing 3 we see that now the MEASUREMENT table is being joined last because the optimiser made use of the histogram and since the histogram shows that nearly 100% of the rows will remain after the predicate is applied it joins that table as late as possible.

```
-> Nested loop inner join
    -> Nested loop inner join
        -> Nested loop inner join
            -> Nested loop inner join
                -> Filter: (`server`.serv_id is not null)
                    -> Table scan on server
                -> Single-row index lookup on rack using rack_id
                 ↪  (rack_id=`server`.serv_rack_id)
            -> Filter: (center.cent_city_id is not null)
                -> Single-row index lookup on center using cent_id
                 ↪  (cent_id=rack.rack_cent_id)
        -> Single-row index lookup on city using city_id
         ↪  (city_id=center.cent_city_id)
    -> Filter: (measurement.msm_value > 80.0000)
        -> Index lookup on measurement using msm_serv_id
         ↪  (msm_serv_id=`server`.serv_id)
```

Listing 3: Result of using EXPLAIN listing 1 when a histogram has just been created on the `MSM_VALUE` column

When experimenting with different queries we found that in addition to the optimiser constraints regarding usage of histograms that we already knew about, and which we discussed in section 2.3, indexes could affect the histogram usage of the optimiser. In MySQL, the optimiser first checks if there exists an index on any of the columns in a `WHERE` predicate, next it checks for a histogram on the same columns. If an index is found on any of the columns, the index is used instead of histograms. E.g. if the query was of the form `SELECT * FROM ... WHERE MSM_DATETIME > SYSDATE-1 AND MSM_VALUE < 80` it did not matter if there existed a histogram on the `MSM_VALUE` column as long as `MSM_DATETIME` had an index. This is a limitation with MySQL which we had to work around, which involved changing the design of our queries such that they no longer include columns with indexes in `WHERE` predicates.

We originally wanted queries of the form shown above because in time-series applications queries are often only concerned with data within a given period. For instance, the last hour or last day of data, not the entire set of measurements ever made. It had the advantage that as long as the measurements were being inserted at a steady rate, i.e. a given number of inserts every minute or similar, the dataset cardinality would not change much after the time had been considered. This would have been helpful since it would reduce the number of unknown variables that could be affecting query execution times [4]. However, as we could not have both indexes and histograms in the `WHERE` predicate, and since removing the index on the `MSM_DATETIME` column is unrealistic, and would cause significant slowdowns we were forced to remove it from the predicate altogether. Thus

[4]As it turned out we were able to perform that constraint in size by using the partitioning described in the dataset paragraph above.

our queries would have to be of the form SELECT * FROM {JOIN PREDICATES} WHERE MSM_VALUE {OPERATOR} {VALUE}.

Our tests confirmed that the queries with the most join predicates and join orderings[5] were consistently the ones which produced the most considerable differences in execution times. We created a few variances for each of our queries, they retained the same WHERE predicate form with different constant values, and the number of join predicates would increase from one to as many as possible. Together with the query shown in listing 1 the one in listing 4 are examples of some of our most promising and useful queries. They both contain as many join predicates as the model allows.

```
select * from test.measurement join test.server on
 ↪  test.measurement.msm_serv_id = test.server.serv_id join test.rack on
 ↪  test.rack.rack_id = test.server.serv_rack_id join test.center on
 ↪  test.center.cent_id = test.rack.rack_cent_id join test.city on
 ↪  test.city.city_id = test.center.cent_city_id where
 ↪  test.measurement.msm_value between 80 and 100;
```

Listing 4: This query utilises all the possible join predicates our data model allows for. Together with the BETWEEN 80 AND 100 predicate this has given us some quite interesting results

5.1.2 Test scenario

We wish to have a set of queries that we can run using different updating strategies, and which will give us reproducible and reliable results. Because of this, we will be using the built-in test framework in MySQL, namely *mtr* to set up and conduct our tests. This testing framework also handles server setup and teardown before and after each test run, *mtr* also allows us to run the entire server in-memory. We mentioned the advantages and emergence of in-memory systems briefly in section 2.2, in our case running the server in-memory gives a significant increase in insert speed which allows us to perform more tests during implementation and evaluation. The results we present will be with the server running in-memory. We ha verified by running our tests using the more conventional disk storage option that the results are similar between the two approaches, and that running in-memory does not affect the applicability of our results negatively. *Mtr* allows us to automate our testing procedures and makes the testing more robust. It also allows us to start testing early on in the development process, which will give us time to reevaluate our designs during implementation. As far as our test scenario design is concerned, based on what we have learned until this point, we will only be running our tests on inserted data and not modified data. From a database point of view, deleting old data and inserting new is equivalent to modifying data. Furthermore, since we are only concerned with the impact histogram accuracy has on query execution times whether we insert and then delete

[5]Join ordering is determined by the data model and the query in conjunction, both the set of join predicates involved and the join orderings the data model allow for, determine which orderings are possible. How join orderings affect query execution time is described previously in this section and subsection 4.1.2.

or modify is irrelevant. In light of that and with our goals in mind, we wish to benchmark these different cases:

- No histogram
- Histogram refreshed after every inserted partition
- Histogram created after the first insert and not updated again
- Histogram updated based on different updating rules

It is not necessary to perform an extensive set of tests to conclude that always updating relevant histogram(s) before query execution should provide the best query execution times. The only problem with this approach is the computational cost associated with creating or updating a histogram; if this cost is less than what we gain by having a histogram, then the decision is easy, always update histograms. However, this is not realistic; the overhead associated with computing histograms for every single query sent to the database will quickly become so large that all the database is doing is computing histograms. Thus we need to find a middle ground between not having histograms and updating histograms all the time.

The four different cases listed above will hopefully show; what should be the worst-case scenario, not knowing the distribution of values. Next, the best-case scenario, having perfect histograms before any query is executed. Then the worst-case scenario for histograms, a histogram that becomes progressively less accurate as the distribution of values in the dataset moves away from the original. Lastly, an updating scheme that will, hopefully, contain at least one rule which will be a middle ground between the best and the worst. A good compromise which sacrifices some histogram accuracy in favour of less updating overhead and an overall better performing database.

Deciding on how to gather and present results of our tests was something we paid particular attention to. We needed a reliable way of measuring the execution time of our different queries after the individual runs. Additionally, we also needed a way to record the overhead related to the different updating rules. The best way we discovered to do this was to let MySQL time the queries and statements for us. Fortunately, it stores a substantial amount of meta-data about queries and statements in the `EVENTS_STATEMENTS_HISTORY_LONG` table in the `PERFORMANCE_SCHEMA` schema. Using that we were able to retrieve the execution time of every query and statement we ran. We then processed that in Python and with Pandas *dataframes*, we were able to aggregate those result together into a more usable form, which we could then present using the Seaborn library.

We have chosen to run every query 10 times, then remove the first one, which is often much slower than the rest since different buffers and caches have to be filled during the first execution of a query. We then load the results into data frames and present those results with accompanying error bounds in our plots. The choice of 10 runs of each query at all partitions is an attempt to reduce the uncertainty and error bounds associated with each query. The error bounds of our measurements are presented in our line-plots as transparent areas coloured the same as the line itself. The area within the error bounds represents a 99.5% confidence interval for the observation[6].

[6]The reader is referred to section 2.2 for short introduction to confidence intervals.

With the line-plots, we can monitor how the different queries are affected by the histogram accuracy in our use-case; however, we do not have any way to know how much overhead is associated with each rule. That overhead consists of the amount of time spent checking and enforcing each rule. We consider the time spent checking each rule to be negligible since it consists of mostly evaluating and advancing counters in C++, which is several orders of magnitude quicker than actually calculating the histogram. To quantify this overhead, we will use the same table as we use to measure the execution time of queries. After each partition has been imported and the queries have been run, we collect the time spent enforcing the rule, namely calculating histograms, that time is then recorded, and our Python code collates that information together into something we can present.

Having a measure for rule effectiveness is something that will be useful to help evaluate how well different rules perform for a given dataset. If we assume that the best execution time we can get is when we time our queries with perfectly accurate histograms, we could define a measure as such. If the total execution time of all queries and the total time spent computing histograms after all partitions have been run for each rule is t_r. And the total execution time of all queries and histogram updates after all partitions have been run for the perfect histogram rule is p. We can then define a ratio from which each rule differs from the optimal by a ratio of q as $q_r = \frac{t_r}{p}$, we term this ratio the Q-ratio, and we will be using it in chapter 7 to measure how well a rule compares to the "optimal" rule.

5.2 Implementation results

Below we sum up our findings from the evaluation and development of the use-case. During the first part of our development efforts, we had to reevaluate the implementation of our use-case continuously. In reality, we were experimenting with different use-cases, changing different parts of them to see which ones would be affected by histogram accuracy and to what degree. How we chose to change the use-cases was not random; it was more like educated guesses as to how we should alter parts of the use-case to make it more susceptible to histogram inaccuracy. This was an iterative process, where each change, or set of changes, was tested and reviewed against previous results. In the previous section, we described this process in detail, and the following is a summary of those iterations.

In the first few iterations, our data model was designed as a *star schema*. As we discovered during our tests, and as we showed in subsection 4.1.2, such a model is not optimal when trying to create query plans with significant differences in execution times. We discovered this after a few iterations, changed the model to reflect that, and it was the first part of our use-case that we finished developing. We tried to think of real-world systems which would be organised in the linear foreign key relationship we described as that would enable many different join orderings for the primary table. We landed on a system which monitors the temperature of servers in data centres. Each measurement is performed at a specific time with a reading of the measurement. Each measurement is taken from a particular server in a given rack within a specific datacenter in a determined city. An ER-diagram of this data model is shown in figure 5.3, we see the linear foreign key relationship which allows the `MEASUREMENT` table to placed anywhere in the join

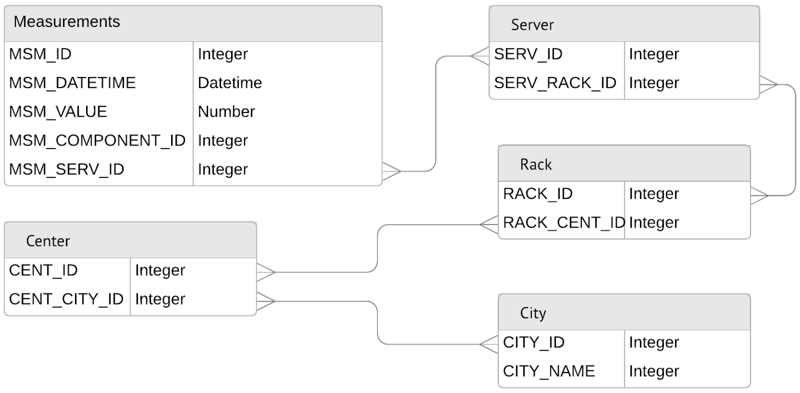

Figure 5.3: The data model in our use-case displayed in an ER-diagram.

order. This data model also supports the requirements of a time-series object, which we needed for the moving mean of our dataset.

After we finished developing the data model, we were able to finalise our set of queries together with the dataset. Our queries have two essential factors that, in no small degree, determine their design. First, the number of join predicates in a query significantly affects the variances we see in query execution times; this means that we would expect to see more significant differences when there are more join predicates and less when there is not. We want to be able to check this behaviour, to do that we wanted queries with varying numbers of join predicates in our set. We created five different ”levels” of queries, and each level has a set number of join predicates ranging from zero, when querying only the `MEASUREMENT` table, to four when all tables are joined together. Second, we know that values in the `WHERE` predicate determine the selectivity of the query. We require that the values used here would allow a range of filtering values from close to zero up to almost one. This would, in turn, mean that the result from the `MEASUREMENT` table could be huge at one point and be significantly smaller at another only due to filtering. Which is necessary to show off the effects of an inaccurate histogram as we showed in subsection 4.1.2. This requirement placed on `WHERE` predicates means that there is a close relationship between, the values used in the where predicate and the range of values we have to choose from when creating the continuously varying value in the dataset. This ties the attributes of our dataset together with the values used in where predicates in our queries. After testing, we settled on a range of values shown in figure 5.1 for the rest of our tests. We chose not to span a massive range so that the transition from matching many values in the predicate to very few would not be too abrupt.

5.2.1 Checking the use-case and the results interpreter

At the end of use-case development, we ran some bigger tests on our Linux machine to verify that our use-case behaved as intended. We created 3 million measurements, split those into 30 partitions and tested three different histogram updating rules. ”No histogram”, ”Perfect histogram” and ”Stale histogram”. We ran small scale tests continuously throughout the development of the use-case, it was as we have stated earlier an iterative process of development and testing. The results of these tests are not included, nor discussed here, that is left for chapter 7. However, as we stopped developing the use-case further after these tests, they did show us that this use-case was influenced by histogram accuracy to a satisfactory degree, and we were confident with continuing.

During the development of the use-case, we also developed a result interpreter. It parses the file created by our mtr test suite and collects the resulting timing data. The results interpreter uses the timing data to create the graphs we use to represent the results of our tests. Thanks to the interpreter we can display results that span hundreds of thousands of lines in a text file into graphs that are concise, easy to understand and enable us to conclude about different attributes regarding the use-case and the various rules. The source code for the results interpreter is included in the appendix of this report under the file name `Results.py`.

5.3 Summary

In this chapter, we have presented the testing and development efforts performed to create and verify our use-case. We also described what the use-case ended up looking like after the development finished. With a firm grasp of the plugin design and both the design and implementation of the use-case, we are ready to present the implementation of the plugin histogram updating rules. In chapter 6 we will do that before we continue to chapter 7 where we will evaluate everything we have implemented in this project.

Chapter 6

Implementing updating rules

During chapter 4 we discussed and reviewed several different designs of our use-case and plugin, and in chapter 5 we showcased the implementation of our use-case. In this chapter, we show how we implemented our plugin and the updating rules which it enforces. We split our development efforts in this part of the project into smaller iterations. This ensures progression throughout the project by achieving small increments continuously. Each iteration is meant to have a clear and attainable goal so that while we achieve little in each iteration, it takes us one step closer to a finished plugin and rule-set. As a rule of thumb, each iteration is concerned with implementing one rule. In this chapter, we describe these different iterations, and in the next chapter, we will be performing a thorough evaluation of the different rules and the effects of inaccurate histograms.

6.1 Rule overview

In section 4.4, the designs of different updating rules were described, not all of these rules have been implemented into the plugin. This was done as a scope restriction both to save time and due to their usability in the use-case. All parameters that the rules use will be implemented the same way as the *rule parameter* the plugin itself uses, meaning that it will be possible to alter the value of those parameters through a system variable. Below we describe each rule briefly and present a short discussion of whether or not it has been implemented.

- Rule one is the trivial rule, to update histograms after every data change. It was implemented, but due to the large overhead associated with this rule, it is not tested as the tests for this rule alone would take over a week.
- Rule two is to update histograms after n data changes. It is also implemented; however, this rule is also tested, as are all other implemented rules unless otherwise specified.

- Rule three was designed to update after n statements, but also consider which statement type executed. This rule is also implemented, however it is not tested, the reasoning for that decision is found in section 7.2.
- Rule four is not implemented because of how it would perform in our tests. The tests are laid out such that inserts and deletes, associated with the current partition are performed first. Then after all the DML statements are complete all queries are run, their execution times collected, and the number of, and total time spent on updating histograms collected. Before finally emptying the table containing the timing data and starting all over again with the next partition. This means that there are no queries that run during the insert/delete stage of tests, this would, in turn, mean that a rule which uses queries to trigger a histogram update and use the DML statements to only mark the histogram as stale would perform optimally. That is exactly how rule four is designed to work, *"after n DML statements, triggered by queries"*. Its results would be the same as our benchmark rule "Perfect histograms", and is as such dropped.
- Rule five is also not implemented as it was designed to update the histogram *"after n rows are updated"*. Now it might seem weird at first to omit that rule, but we control the load the database experiences, and therefore it is known that every DML statement the database executes effects *exactly* one row. Therefore as far as our testing is concerned, there is no difference between monitoring the number of statements or the number of rows those statements have affected. This means that rule five will be "implemented" when rule two is implemented, its results would be the same as that of rule two. There is; however, a distinction between the two based on what results the rule would achieve in all possible cases. Which means that while in this particular case rule two and five can be considered the same, in most cases they would not. This will affect how the results from our tests of rule two will apply in the general case. In most systems one would probably rather have rule five than rule two implemented.
- Rule six was designed to consider the table size when deciding on how many rows could be updated before a histogram update was needed. In its definition, it is meant to use rows updated as the measure, statements executed will be used instead for our implementation of this rule. This is based on the same arguments that were given for omitting rule five, namely that in our use-case, there is no difference between the number of statements and the number of rows updated.
- Rule seven is meant to give more importance to statements that affect values that are outside the range of the histogram. This rule is also implemented and use the same simplification that rule six uses regarding counting statements instead of rows.
- Rule eight keeps a sample of the table in memory and applies any incoming updates to that sample as they are applied to the main table. The sample table is then used to determine how the distribution of values has changed, and if a histogram update is required. This rule is not implemented in this project due to time constraints.
- Rule nine was intended to emulate a cost function, much like the one the optimiser uses, to determine if the cost of inaccuracy associated with the current histogram

warrants updating to a new one. This rule is implemented and use the same simplification that rule six and seven use regarding how to count updates.

6.2 Implementing rules

In the following section, the different iterations the plugin has gone through are described, problems encountered, and solutions implemented during these iterations are discussed continuously. How problems were solved and why they were solved in a particular way is also briefly discussed. The first iteration is meant to change the workflow of the plugin created in [3] to match the workflow designed for the new plugin in section 4.3.

6.2.1 Getting the plugin ready for the new rule architecture

Development of the plugin for this project has been a continuation of the work started we started in [3] the reader is referred to our previous work for mew information about the initial structure and development of the plugin. The plugin developed in [3] required some changes to adapt to the new data model used in this report as well as changes to the workflow.

In the architecture described for the plugin in [3], which parts of were repeated in chapter 4, the plugin was intended to vary which tables and columns would have their histograms updated. However, the data model developed for the use use-case used in this project only has a single column which uses a histogram. The update string the plugin uses to update histograms was therefore changed so that it only updates the `MSM_VALUE` column. The development goals of this project is a plugin capable of updating histograms for a given column in a given table based on a chosen active rule. Since the data model is designed only to use histograms on a single column, as discussed in both chapter 4 and chapter 5, we do not want to spend unnecessary time on developing a feature that is not needed. This plugin is designed to; test the effectiveness of different updating rules, experimenting with varying measurements of staleness, and when an update is required, it does not warrant a general plugin capable of operating within different data models.

Before the plugin can be used, it must first be installed and initialised on the server. Upon initialisation, it reserves its memory location, declares internal variables and initialises a variable determining the currently active rule to a default value. Before the plugin will do anything this ”rule variable” must be changed, in its default state the plugin does nothing regardless of incoming statements. Meant to be a non-intrusive default version of the plugin so that it can be initialised without influencing any other tests, and be deactivated at any time without having to uninstall it. The internal rule variable is linked to a global system variable, `histogram_updater_rule`, this makes it easy to change, which rule the plugin is enforcing using SQL. Changing the value of the global variable to the corresponding rule number used to number rules in section 4.4 enables that rule.

After the plugin is initialised, it will be called by MySQL after a statement has been parsed. The first thing the plugin does is then to check what type of statement has been parsed. If the statement is either an insert, update or delete statement, then the plugin will check which rule is active. Then based on which rule is active, a different set of requirements are checked, if they are met, then a histogram update is performed on a separate MySQL connection. Currently, the execution of that separate connection is done within a single thread; this means that any DML statement that invalidates a rule will be halted until the separate connection has finished updating the histogram. If the plugin was to go into production, this is not how it would have been implemented since forcing the users to wait for the histogram update before their statement can execute probably is not acceptable for users. However, for this project users are of little importance, and the main point of interest is the effect on query execution time. Thus after the separate connection finishes updating the histogram, the plugin is finished executing and returns control to MySQL.

6.2.2 *Rule one and two*, after each DML statement and after n DML statements respectively

The goal of the second iteration was to get the plugin to work properly with rules enabled and to implement rules one and two from section 4.4. It was chosen to include both rule one and two in this iteration since they are very similar and one rule can easily be converted to the other with only small changes in code. In listing 5 and listing 6 we present the pseudocode for these two rules, it is easy to see how similar these two rules are.

Getting rules working with the plugin was quick and painless, once we discovered how to set and manipulate the value of global variables from within *mtr* it was time to implement the two new rules. To start with, we created variables to contain counters for the number of received statements. Then the rules themselves were implemented. Rule one simply checks if the incoming statement is either an insert, update or delete and executes a histogram update if it is. Rule two stores the count of incoming insert, update and delete statements, divides that count by a number n, to begin with, 1000 was chosen as a value for n, and checks if the remainder is zero. If it is, a histogram update is executed.

```
IF (update_rule == 1 AND (statement_type == INSERT OR statement_type ==
 ↪  UPDATE OR statement_type == DELETE)){
    update_histogram();
}
```

Listing 5: Pseudocode explaining the logic implemented for rule one.

We then attempted to test these rules, and we quickly discovered a problem with rule one. It never finished, it took such a long time to insert the data because of the constant histogram updating that the test had to be aborted. This result shows what we already expected, namely that updating the histogram after every DML statement is too costly and is not a feasible solution. This rule is therefore left out from all subsequent tests.

When rule two was implemented, a somewhat arbitrary choice of 1000 was chosen as a value for n. As mentioned in section 4.4 selecting a ”good” value for n can be difficult, yet it is essential since the choice of n will influence the efficiency of the rule. Consider the following example; if the partition size were 100000 rows, and the value chosen was 100000 then the queries would have perfect histograms, in our use-case, with

```
IF (update_rule == 2 AND (statement_type == INSERT OR statement_type ==
 ↪  UPDATE OR statement_type == DELETE)){
    increment rule_2_counter;
    IF (rule_2_counter modulo rule_2_no_between_updates == 0){
        update_histogram();
    }
}
```

Listing 6: Pseudocode explaining the logic implemented for rule two

only a single histogram update, which would be equivalent to the performance of the ”Perfect histogram” rule. While with the above choice, the histogram would be computed 100 times. Which alternative of n is a ”good” one is dependent on the type of load the server sees, which means that which value is the right choice for n will differ from database to database. This project will not try to define a global best choice of n because we believe there is none; however, trying to determine a suitable choice for n for this use-case will be done. And while doing so, it will be investigated how sensitive rule two is to changes in this value. To quickly test the sensitivity of the rule, the value of n is linked to a global variable which can be changed using SQL such that testing different values is easy.

6.2.3 *Rule three*, after n DML statements also considering statement type

The third iteration was intended to implement rule three into the plugin. The idea is that some statements affect the distribution of the value more than other types and that this can be accounted for by weighting the statement types differently. In the implementation system variables were used to weight the different statement types. E.g if the plugin receives a `INSERT` statement that will increment the `rule_3_counter` by $1 \times insert_type_weight$. Using system variables allows us to change the behaviour of the rule easily. Using system variables, one can define rule three to update the histogram three times as often if `INSERT` statements are the only type received when compared to receiving only `UPDATE` statements. Or one can choose other values for the system variables and weight deletes as more important. In listing 7 pseudocode for the implementation of rule three in the plugin is presented.

6.2.4 *Rule six*, after a ratio r between table size and updated rows is reached

Rule six is implemented through a single counter and a helper function which fetches the number of rows in the `MEASUREMENT` table. The counter is divided by the number of rows in the table and the ratio is checked, if it exceeds a value r the counter is reset and a boolean `TRUE` is returned indicating that a histogram update is called for. Pseudocode for this rule

```
IF (update_rule == 3 AND (statement_type == INSERT OR statement_type ==
↪  UPDATE OR statement_type == DELETE)){
    IF (statement_type == INSERT){
        increment rule_3_counter by 1*insert_type_weight;
    }
    IF (statement_type == DELETE){
        increment rule_3_counter by 1*delete_type_weight;
    }
    ELSE {
        increment rule_3_counter by 1;
    }
    IF (rule_3_counter modulo rule_3_no_between_updates <
    ↪  max(insert_type_weight,delete_type_weight)){
        rule_3_counter = 0;
        update_histogram();
        }
}
```

Listing 7: Pseudocode explaining the logic implemented for rule three

is included in listing 8; however, no code for the helper function `get_table_size()` is included.

6.2.5 *Rule seven*, after n rows are updated, comparing the change against histogram boundaries

The next rule to implement was rule seven; it is designed to account for changes that would affect the range of the histogram. Meaning that when a change happens, that is not captured by the histogram, updating or inserting a row outside the range, those changes will be weighted more than something that is happening inside the range.

To implement this, one needs to be able to determine what the range of the current histogram is, and then check if the update falls outside that range. The rule starts by checking the values for lower and upper bound are invalid; if they are these values must be fetched. To do that the database is queried for information about the histogram, using JSON operations both the upper and lower bounds of the histogram are found, and corresponding variables are updated. Each time the MySQL engine calls the plugin, it checks if the value to be inserted into the table falls within these histogram boundaries. If it does not, the insert is weighted more and counts more towards considering the histogram as stale. If it falls within, then the update is given an average weight, that weight is the same that given to `DELETE` statements. When the value of rule seven's counter becomes too high, signifying that the histogram is no longer accurate enough, the histogram is updated, and the boundary values that have been used are invalidated.

```
IF (update_rule == 6 AND (statement_type == INSERT OR statement_type ==
 ↪  UPDATE OR statement_type == DELETE)){
    IF (tables_size needs to be updated){
        table_size = get_table_size();
    }
    increment rule_6_counter by 1;
    ratio = rule_6_counter/table_size;
    IF (ratio > r){
        rule_6_counter = 0;
        invalidate table_size;
        update_histogram();
    }
}
```

Listing 8: Pseudocode explaining the logic implemented for rule six

6.2.6 *Rule nine*, after the estimated cost of an inaccurate histogram exceeds the limit I

The last rule to implement was rule nine. This rule is designed to be able to adapt to different database loads. To achieve that several weights were used, these can be altered to change the behaviour of the rule. In the implementation, this is achieved by using system variables as we did for the rest of the rules. A counter signifies the ”amount” of change that has happened to the table, this amount of change is calculated using different weights on the different operations that have happened since the last update, and is outputted as a single number. The ”amount of change” is multiplied with the ratio between the number of changed rows and table size, and compared with an update threshold. In listing 10 it can be seen how this functions in pseudocode, notice that the rule starts by gathering the values used to determine if updates are happening outside the histogram range, just like rule seven. Then the table size is fetched using the same function that was created for rule six. Next, the ”number of updates” counter is incremented by one times the given weight depending on what the DML statement is. Finally, the rule checks if the histogram should be updated using a simple function at the end. In many ways this rule acts as a cost function, attempting to weigh the cost of updating the histogram against the cost of using an ”outdated” histogram.

6.3 Summary

In this chapter, we have presented; which rules we implemented, why we implemented them, how we implemented them and the pseudocode related to each. We also showed how the different rules could be tuned using the system variables. In the next chapter, we will use these system variables to show how sensitive the rules are to changes in those values; furthermore, the results of testing these rules will be presented.

```
IF (update_rule == 7 AND (statement_type == INSERT OR statement_type ==
 ↪  UPDATE OR statement_type == DELETE)){
    extract value to be inserted or updated;
    IF (lower_bound and upper_bound need to be updated){
        fetch_histogram_boundaries();
    }
    IF (statement_type == INSERT && value to be inserted or updated is
     ↪  outside histogram boundaries){
        increment rule_7_counter by 1*insert_outside_range_weight;
    } ELSE {
        increment rule_7_counter by 1;
    }
    IF (rule_7_counter modulo rule_7_number_between_updates <
     ↪  insert_outside_range_weight){
        rule_7_counter = 0;
        invalidate lower and upper bound;
        update_histogram();
    }
}
```

Listing 9: Pseudocode explaining the logic implemented for rule seven

```
IF (update_rule == 9 AND (statement_type == INSERT OR statement_type ==
 ↪  UPDATE OR statement_type == DELETE)){
    increment altered_rows by 1;
    extract value to be inserted or updated;
    IF (table_size needs to be updated){
        table_size = get_table_size();
    }
    IF (lower_bound and upper_bound need to be updated){
        fetch_histogram_boundaries();
    }
    IF (statement_type == INSERT && value to be inserted or updated is
     ↪  outside histogram boundaries){
        increment rule_9_counter by 1*insert_outside_range_weight;
    } ELSE IF(statement_type == INSERT){
        increment rule_9_counter by 1*insert_type_weight;
    } ELSE IF (statement_type == DELETE){
        increment rule_9_counter by 1*delete_type_weight;
    } ELSE {
        increment rule_9_counter by 1;
    }
    ratio = altered_rows/table_size;
    IF (rule_9_counter*ratio > rule_9_update_threshold){
        rule_9_counter = 0;
        altered_rows = 0;
        invalidate table_size;
        invalidate lower and upper bound;
        update_histogram();
    }
}
```

Listing 10: Pseudocode explaining the logic implemented for rule nine

Chapter 7

Evaluation

In this chapter, we will be presenting the results of testing our implemented rules for automatic histogram updating. We start the chapter by presenting our metrics, why we designed them as we did, and why we are testing as we are. We then continue by presenting and discussing the results of each of our rules before we compare their performance. We then discuss possible errors and flaws in our use-case and testing design before we round off the chapter with a summary.

7.1 What, how and why

To evaluate the updating rules, we will investigate; how histogram accuracy affects query execution time, how the rules we have implemented will affect histogram accuracy and how those rules perform in our use-case and in general. We have chosen several metrics we will be monitoring through the tests we performed on both our Linux machine and an Oracle test server. These machines had 16GB of RAM and an Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz, and 512GB of RAM and two Intel(R) Xeon(R) Platinum 8160 CPU @ 2.10GHz respectively. To evaluate these elements we will be presenting a few different metrics, and these are; the q-ratio, the histogram updating overhead, the effect on the query execution time for different data partitions and rules, and the sensitivity to parameter values. In this section, we will discuss why we have chosen these metrics, what they are, how we use them, and how to interpret the graphs used to display them.

Measuring query execution time under different updating rules

We have chosen to use the execution times of queries as a measure of how well the histogram is performing because we believe that accurate histograms will improve the choice of query plan that the query-optimiser makes. If we did not think this was the case, we should not use histograms during query optimisation at all. If the choice of query-plan is better, that in turn means that queries will execute faster. Other metrics could be used to define if a query plan is good or not, however, for this project the measure of how good

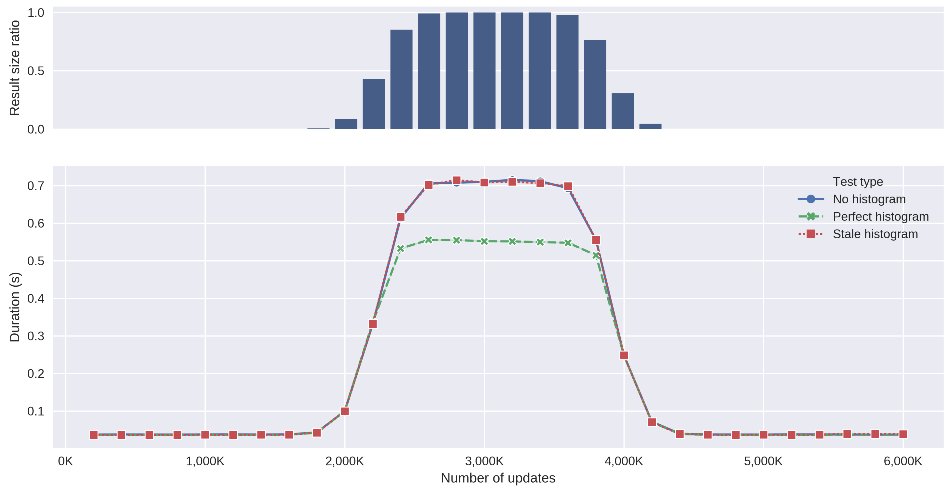

Figure 7.1: Results of timing a query with four joins and where `MSM_VALUE` $>$ 90 for the three base classes

plans are is the execution time of queries[1]. Thus measuring the execution time of queries with different levels of histogram accuracy will reflect how well the current histogram is performing at the given accuracy level. If the histogram is providing useful data to the optimiser, then the change in plan due to the histogram will be positive, if that is not the case, then the histogram is performing poorly. Once a measurement of execution times for one set of data is obtained, the data is altered and the execution times for queries is measured again. We can then see how the change in data has affected the query execution time by affecting the accuracy of the histogram and in turn causing the chosen plan to be either better or worse than it used to be. The results of this type of timing for each histogram rule we have implemented will be presented using graphs like those shown in figure 7.1

Figure 7.1 contains two graphs - the upper graph shows the size ratio of the query result compared to the table size, higher values indicate more returned rows and 1 indicates all rows were returned. The lower graph shows the average execution time of the query in question, one with ”four joins and a where predicate requiring `MSM_VALUE` $>$ 90”, measured at each inserted partition where the different coloured and marked lines represent a different updating rule. At each bar in the top plot and each marker in the bottom plot one partition of data has been inserted into the `MEASUREMENT` table, our set of queries has been run and the timing of queries collected. In other words, these two graphs share a common x-axis. If we first look at the upper plot, we see that around the middle of the graph, the ratio is equal to, or at least very close to one. Meaning that at that point the cardinality of the result is the same, or close to the same, as the cardinality of the `MEASUREMENT` table. This part of the figure is used to show how the size of the result affects the execution time of the query.

[1]Better plans will have lower execution times than poor plans when querying the same data under similar server load.

If we then turn our attention to the lower plot, we see that along the y-axis the execution time in seconds is plotted, and along the x-axis the ”Number of updates” is plotted. The number of updates tells us something about how the distribution of data has changed since the first row was inserted. While the ”Number of updates” is plotted along the x-axis, updates are not performed. Instead, inserts followed by deletes are performed. In theory, this can be regarded as the same thing, however, since by removing one row and replacing it with another that holds a different value one has essentially updated the row that used to exist. From a logical standpoint there is no difference between the two methods, they both result in the row containing the updated data. We showed an example of how the distribution of values could change as one progressed through the dataset in figure 5.1 in section 5.1, the actual distribution of data used for the tests conducted here is discussed in section 7.2. In that section, we present a figure which shows that until row three million is reached the mean of values in the distribution is strictly increasing. After that, the values are strictly decreasing back down to the starting point. Thus, at the middle point of the graph, the distribution of values in the `MSM_VALUE` column has been altered from the original distribution the most. We see that this is reflected in the top graph of figure 7.1, where, as we approach the middle of the graph more and more rows are returned, and in the middle, the largest amount of returned rows is reached, with a ratio of practically one. When the top plot and bottom plot shown in this figure are viewed together, they can then be used to determine how the updating rule performs. By measuring query execution time, as the distribution of data changes also taking into consideration how the size of the result set changes as one progresses.

In figure 7.1, the execution time for one query tested under three different updating rules is plotted. All queries are run ten times for each partition, and the first of the ten runs is excluded from the result. This is because, during the first execution of a query, MySQL fills different buffers with data that is used to complete the query. When the query runs for a second time these buffers help answer the query quicker, as such the nine runs that happen after the first have a much more consistent, and lower, execution time than the first. The average of these nine remaining query execution times is then calculated and plotted.

Around each line there is also a transparent area coloured the same as the respective line, this area signifies the range of a 99% confidence interval for the real average execution time for the query. We discussed the theory of, and how to interpret, confidence intervals in section 2.6. The reader is referred to that section for an explanation of how confidence intervals work. We have chosen to use a 99% interval instead of the more typical 95% interval because of the stability of measurements we have seen during development and testing. Increasing the percentage of our confidence interval results in a larger interval that we can be more confident contains the real value of the query's average execution time at each partition. Also, even with as a high a degree of confidence as 99% the interval is barely visible in most cases, meaning that the inter run query execution time is very stable when no other factors are influencing the database. If any factors are influencing the database such as; no more available RAM or processing power, the query times begin to fluctuate drastically. This fluctuation would then be visible in figure 7.1 as large confidence intervals surrounding each line. This was a problem during our testing. However, through careful management of computer resources and running our test several times we were able to achieve results that do not contain this uncertainty to any significant degree

Through our testing, we have uncovered that even with confidence intervals, there can be fluctuations in execution times that will not be transparent in our plots. This is only a problem for the sensitivity plots we present in section 7.1, where large fluctuations can occur without the plot showing the corresponding large confidence intervals which should surround each line. We believe this is caused by how the available processing power can be reduced for long periods in combination with how the sensitivity plots work. Namely, since the time shown in the sensitivity plots is an aggregate of the time spent computing all queries if all queries are computing slowly due to the machine being overloaded, there will be little variance between the slow results. While there can be significant differences between the slow results recorded at one parameter value and much faster results recorded at another parameter value when the machine is not overloaded. Thus it will look as though some rules perform much worse due to a change in parameter value when the reason they are performing worse is that the machine is overloaded. This behaviour has caused us to run our tests several times to gather results without this misleading discrepancy. There is one data point in figure 7.2 which shows this discrepancy in play. Around the middle of the graph, one point of ”Plugin rule X”, jumps to the level of ”Stale histogram” before jumping back down to the level of ”No histogram”. If everything was working perfectly, and there were no sources of noise or variance while testing, the blue line in this plot would follow the green line perfectly. However, we see that at one point during testing, the rule was slowed down for some reason. We argue that this does not affect the reliability of the tests we have performed since while this did happen for one point, the trend is clear and if we were to rerun the test with only that point we are highly likely not to see the discrepancy.

Histogram updating overhead

We showed in chapter 4 that the trivial way to obtain accurate histograms at any time is to update them whenever a data change happens on the column(s) the histogram is modelling. We also showed that the problem with such an approach is the overhead associated with computing a histogram that often. As an example we have seen during our tests on the Linux machine describe above, that computing a histogram for the `MSM_VALUE` column takes approximately 0.16 seconds when the table contains $200,000$ rows, while most of our queries take somewhere between 0.1 and 1 second to complete. It is clear that spending so many resources on continuously updating histograms is not viable for any MySQL database[2]. The goal of the rules we have created during this project is to reduce this overhead while still improving query execution time.

We categorise rules into ”good” and ”bad” rules based on how they impact the database, and in particular if they have a net-positive effect on available resources or a net-negative effect. I.e. if the time spent on ”enforcing” a rule is less than the time the rule saves through reducing query execution times, the rule has a net positive effect on available resources and would be categorised as a good rule by us. On the other hand, if we spend more time enforcing a rule than what we save through the reduced execution time of queries, that rule has a net negative effect on available resources and is categorised as a bad rule. It is es-

[2]Indeed as we have discussed earlier we do not have enough time even to record the results of such a rule since it would take approximately 100 days to do so with our dataset and queries.

sential to realise that the load the database sees, both through DML statements/change in data and query throughput, will influence which category a rule is placed in. E.g. a rule which updates histograms after every DML statement will be categorised as a bad rule in a database which has a large amount of DML statements and relatively few queries. On the other hand, if the database sees very few DML statements but a large number of queries, the same rule might be regarded as a good one. We discussed in the previous section how we would measure the query execution time, i.e. how much resources the database spends on executing a query under a given updating rule. Now we need to determine how much resources the database spends on enforcing that given rule, i.e. the overhead associated with a rule.

For each rule we will record the time spent computing histograms, the sum of all those updates is then considered to be the overhead associated with that rule. This means that no other operations are taken into consideration when determining the overhead, even though those operations might have an impact on the total overhead of the rule. While all rules have some degree of overhead in the form of checks performed in the plugin, we consider that to be negligible. Some of the rules; however, like six, seven and nine, query the database for information about table size and histogram boundaries at various points[3]. The impact of these queries is not visible in the measure of overhead used; however, we have measured the time spent computing these queries outside of the plugin and have found that the time spent is negligible, less than 0.01 second on average, when compared to the time spent computing the histogram.

We will present the overhead associated with a rule as we reach that rule in the evaluation section. There is a summarising table towards the end of our evaluation, which shows the overhead associated with the different rules together with their overall performance[4].

Sensitivity to parameter values

The rules we have defined and implemented in this project all have one or more parameters associated with them, be that the number of statements between updates, n, for rules two and three, or the required ratio of change,r, for rule six. The values given to these parameters will influence when the rule triggers a histogram update, and by so doing, those different choices of values will influence the efficiency and performance of the rule. In addition to that, different database loads will have different values that would be considered optimal for these parameters. We will not attempt to determine which value is the optimal choice for n or the ratio r or any other of the available parameters for all use-cases and database loads, that is simply not possible. Exactly as in the case of *when rules are considered "good" and "bad"* from the previous section, the load on the database can cause any choice of parameter value to be the worst, or among the worst possible. If we, for instance, consider the situation from above, there are two different databases, one has a large number of updates to data with little queries, and the other is the exact inverse with little updates and large amounts of queries. Let us say that we use the ratio of changed rows, r, as the trigger for when to update histograms, i.e. we are using rule six to keep

[3]The reader is referred back to the pseudo-code given for rules six, seven and nine in section 6.2 for information on when those rules will query the database for data.

[4]An example of such a table with just the base classes is shown in table 7.1.

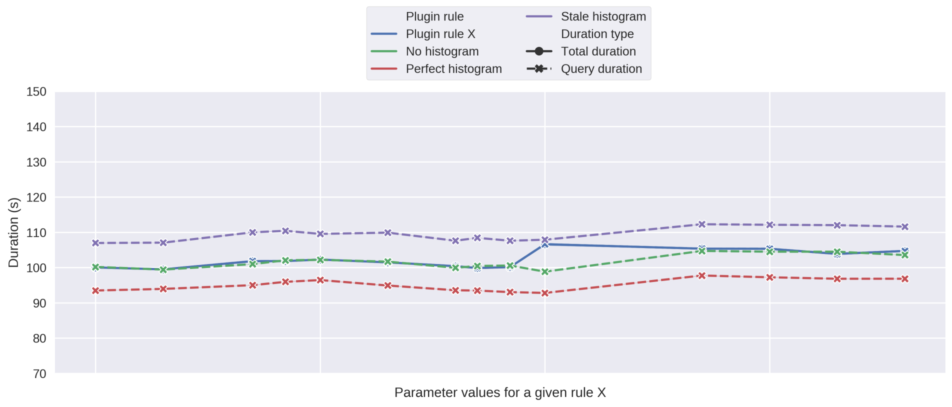

Figure 7.2: An example plot of how rule X responds to changes in its parameter. The base classes are plotted along side to serves as comparisons

histograms up to date. A low value for r can be considered advantageous for the second database since there are very few changes in data compared to the number of queries which will benefit from a good query plan. A low value for r is dire for the first database; however, since there are many updates which will cause vast amounts of resources to be spent om computing histograms compared to executing queries. As we can see, the only option available is to try to choose a value for these parameters that suits each particular use-case, and use that value when comparing the performance of the different rules. To provide some insight into the importance of this choice, we will try to determine how sensitive each of the different rules are to changes in their parameter(s), and how choosing a value which is not the optimal influences each rule.

To do that we will be running the entire test set over and over with different values for these parameters. We will then record the total time spent computing histograms, and the time it takes to execute all queries once for each of the different rules and plot that in a graph for each choice of parameter values. An example of the graph that will be used to present this data for each rule is shown in figure 7.2. For all the sensitivity plots that follow, the base classes will always have same coloured lines, namely green for ”No histogram”, red for ”Perfect histogram” and purple for ”Stale histogram”. The rule that is investigated in any given sensitivity plot will always be the blue line, and whole drawn lines always indicate the total duration. I.e. the time it takes to answer all queries once and the time spent on updating histograms, while dashed lines indicate the duration of just executing all queries once. The legends for all sensitivity plots is, therefore, the same as the one shown in figure 7.2 and are omitted from the other graphs.

We have chosen to invest time in determining the sensitivity of each rule because we believe that determining the sensitivity of a rule is some of the most useful information we can give about the rules. As an example, let us say that we have found through testing that $n = 1000000$ results in an optimal query execution time for rule two in our use-case. That does not give much useful information about how rule two will perform in any other use-cases with that value of n. If, however, we could say that values of n between 100000

and 10000000 have roughly the same query execution duration as 1000000, that is useful. Because that means that even if we choose a value of n that is ten times larger or smaller the optimal value, we still achieve a near-perfect query execution time in our use-case. While the query execution duration of a rule will fluctuate between different use-cases due to the differences in workload, datasets and data structure, it indicates how quickly the efficiency of the rule changes as we either increase or decrease the value of n away from the optimal. There is still one thing more to consider; however, the query execution time alone does not tell the whole story.

From our discussion in the previous section regarding histogram updating overhead, we can conclude that, knowing the time spent computing histograms is of vital importance when evaluating any updating rule. Thus we must also consider how the time spent on computing histograms differs with different parameter values for the rules we are testing. In figure 7.2, we show this difference by drawing the total time used for the base case rules and then drawing both the query execution time and total execution time for the rule. The overhead associated with each rule is then the distance between the dashed and whole drawn lines of the same colour.

We have chosen to omit the lines showing the query execution time for the base classes from the plot to increase readability, as the base class rules spend a negligible amount of time on computing histograms those lines would almost entirely overlap. Thus we will be able to see in the sensitivity plot how a rule changes behaviour as we apply different parameter values, and compare that to the behaviour of the base classes. It is important to realise, however, that the base classes do not use parameters and should, therefore, see no change in total duration based on the different parameters used. I.e. they should be relatively straight lines in the sensitivity plot with only minor variations in total duration caused by variances between the different timing runs.

Performance compared to the ”optimal”

Discussing the performance of the updating rules we have implemented makes no sense if we do not have something to compare that performance to. For that, we have chosen what we believe to the best and worst-case scenarios regarding histogram accuracy and query execution times. The three cases we have chosen can be seen in figure 7.1, namely; ”No histogram”, ”Stale histogram” and what we have called ”Perfect histogram”. The first case is straight forward. There are no histograms for the query-optimiser to use when attempting to approximate predicate selectivity it must use heuristics; therefore, it should result in sub-optimal query plans. For the ”Stale histogram” case a histogram is created when the first partition of data is loaded, that histogram is then never updated and will, as time progresses and new partitions of data are processed, become stale. After the middle partition is imported, the histogram will start to become more and more accurate. Until it is almost entirely accurate again when we reach the last partition, this is not necessarily how the stale histogram will behave in all scenarios of course. This means that as the underlying data changes and the query starts to either return more or fewer rows, depending on the predicate, the query plan that is being used will remain the same - because the histogram is the same. For the third case, a histogram is created right before the queries are timed. Meaning that the selectivity estimate the histogram provides is as accurate as it can be for every single query that we time. It also uses the least possible amount of histogram updates to achieve the optimal accuracy, namely one for each partition. This should then result in the most ”optimal” query-plans, and by extent the fastest query execution times, while computing the histogram as few times as possible, leading to the lowest total duration time. The advantage of the ”optimal” rule is then that the query-optimiser will be able to change the plan being used as the underlying data changes. This can be seen in figure 7.1 where, as the distribution of data in the `MSM_VALUE` column changes, the query plan is being altered to reflect that.

It is important to remember that in the MySQL query-optimiser, the only thing histograms will influence with regards to query plans is the join order. The selectivity estimate that histograms provide is used by the query-optimiser to determine how the cardinality of relevant tables will change as parts of the `WHERE` predicate is applied. This means that the only difference between sub-optimal and ”optimal” query plans is the order in which the tables are joined together.

The three base classes are used as something to which we can compare each rule we review. We have already used these base classes to show and explain the figures we will be using when evaluating each of the rules. The two first classes, ”No histogram” and ”Stale histogram” are used only as comparisons, both as baselines to compare the updating rules against, but also so that we can compare how these match up against the ”Perfect histogram” rule.

In addition to being used as the ”golden standard” in our plots, the third case, ”Perfect histograms”, will also be used to compute the Q-ratio[5]. The Q-ratio is a single number which indicates how well a rule has performed against the optimal. This is done by comparing the total time spent executing queries and updating histograms for each rule over all partitions of data to the time the ”Perfect histogram” rule used to do the same. Since we will be performing our entire test case over and over with different parameter values for the different updating rules, each rule will have several Q-ratio’s, one for each parameter we used. We will also have several runs of the base classes, these are not influenced by the changes in parameters, and we will, therefore, compute the mean of the different runs for the base classes. This is done to increase the reliability of the values the base classes have in the Q-ratio.

In table 7.1, we showcase how the Q-ratio and accompanying data will be presented in the final evaluation using only the base classes. When studying this table, it might become apparent that the values in the ”Total duration” column do not match up to the values we have presented for the total duration of the same rules in the sensitivity plot in figure 7.2. In the plot, the total execution times lie somewhere between 80 and 120 seconds for all the base cases, while in the table, they lie between 800 and 1000. This difference is intentional, and it is caused by us computing the ”Total duration” in table 7.1 as the aggregate of; all the times we ran all queries and all the histogram updates performed for each parameter value. While in the plot the ”Duration” is only the sum of running all queries once with a given parameter value and the time spent updating histograms. The other eight times each query has been run is used to create the confidence interval, because to create a confidence interval, we need to have several recorded values. I.e. if the same aggregation that is used in the Q-ratio table were also used for the sensitivity plot we would not be able to plot the confidence interval, and the duration values would be identical between the two.

[5]We first described the usefulness, and presented a formula, for the Q-ratio in section 5.1.2.

Test type	Total duration	Q-ratio	Query duration	Histogram duration	Number of histogram updates	Parameter values
Perfect histogram	823.9	1	819.9	4.0	30	Not applicable
No histogram	915.2	1.111	915.2	0	0	Not applicable
Stale histogram	984.8	1.195	984.7.0	0.1	1	Not applicable

Table 7.1: Example of how the Q-ratio will be presented in a table

Using the Q-ratio, we can see that the total execution time of all queries for the ”No histogram” rule is about 11% more than that of the optimal case. We also see that the execution time of the ”Stale histogram” rule is about 19% more than the optimal and by extent 8% more than not having a histogram. This shows that the ”Perfect histogram” outperforms ”No histograms” by about 11%, in our use-case. This also means that if the query-optimiser has access to accurate histograms, it will enable the query-optimiser to do a better job at choosing queries. Perhaps even more surprising is the fact that having a histogram that is accurate to begin with, that then becomes increasingly inaccurate before it starts to become accurate again, has a net negative effect on the available resources in the database. Based on the categories laid out above, we would categorise the ”Perfect histogram” rule as a good rule, and the ”Stale histogram” rule as a bad rule. The ”Stale histogram” rule is regarded as a *worse* rule than the ”No histogram” rule!

7.2 Results of rule testing

For the results presented in this section, a dataset consisting of six million rows for the `MEASUREMENT` table was created and divided into thirty partitions. The size of the tables was then 200000for the `MEASUREMENT` table, 70000 for the `SERVER` table, 70000 for the `RACK` table, 35000 for the `CENTER` and finally 35000 for the `CITY` table at each partition. The choice of table sizes for the other tables than `MEASUREMENT` was not arbitrary; we discussed how the inter table size relationship was important for the possible choices of query-plans in section 5.1. It is based on that discussion and our results from the first part of development efforts that we chose the sizes we did.

In figure 7.3 we have plotted the distribution of values for the `MSM_VALUE` column. We see that as we progress through the rows, split into thirty different partitions, the value for the `MSM_VALUE` column changes. It transitions from a mean of 60 at the first partition to a mean of 105 at the fifteenth partition, before moving back to the original mean of 60.

For the following test results, the dataset just described has been used. For each of the rules, we will discuss our choice of parameters and its effect on query execution times. Please note that as we stated in subsection 6.2.2 and earlier in this chapter, we are not at-

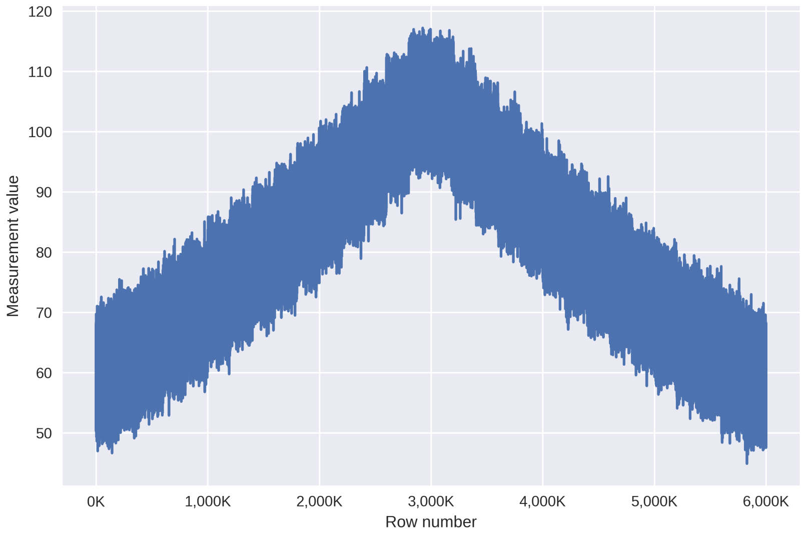

Figure 7.3: How the distribution of values changed for the `MSM_VALUE` column during our tests

tempting to define the optimal values for parameters in all possible cases. We will attempt to choose good values for our use-case, beyond that we will discuss how the sensitivity to parameter values for each rule will affect how important it is to choose correct parameter values for different use-cases. We will also present the results of the rule in both of the graph types shown in the previous section, and a summary regarding the Q-ratio and other important values. We will vary which query we present the results of for the different rules, so we can show the differences in how `WHERE` and `JOIN` predicates affect the execution times and graph shape. The choice of the query will not affect the sensitivity plot or the Q-ratio since those use aggregate values of all queries; the difference will only be visible in the line plot we showed in figure 7.1.

We will only be presenting results from the rules we have implemented in our plugin. Meaning that rules; one, four, five and eight are not presented below as we cannot test them. The reasons for why some rules have not been implemented differs and has been discussed in chapter 6. As the implemented rules are presented, we will also present the ranges chosen for parameter values. We believe the most important thing to consider when defining this range is to be relatively confident that the ”optimal” choice of the parameter in question lies within that range. With that in mind, one should consider what the different values of the different parameters result in. This will be covered for the different parameters as they are encountered.

It was stated earlier that tests would be conducted on two different machines, a desktop Linux machine and a test server. The results presented will be from the Linux machine exclusively, since it was quicker to run the tests on that machine due to the higher CPU clock speed. The tests that were developed for this project is not a multithreaded workload. As a result, there is a performance decrease when downgrading to the lower CPU clock speed on the test server. While the speed of the tests differed between the two machines,

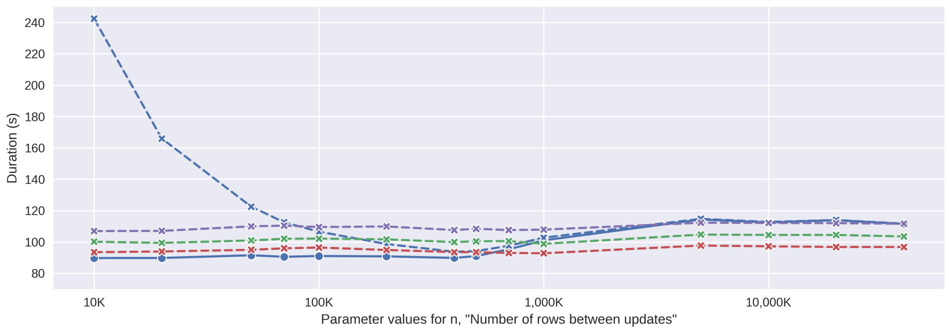

Figure 7.4: The sensitivity to change of parameter values for rule two with a logarithmic x-axis

the results were similar. Meaning that rules behaved similarly across the two machines, and we expect them to behave similarly on any other machine. It was therefore decided to continue testing on the fastest machine and present the results from that machine alone.

Rule two, after n DML statements

The first tests that do not involve a base class rule are those performed for rule two. This rule has only a single parameter that we can vary, namely the number of statements n between each histogram update. It is obvious that the lower the value of n, the more frequently the histogram will be updated. As stated earlier, we will be importing a total of six million rows each time we test one of the rules. Meaning that if we choose a value of say 10000 for n that would cause the histogram to be updated 1180 times for our test since in total the test will execute 11800000 DML statements, 6000000 insert statements and 5800000 delete statements. While we would want to test this rule with very different values of n, say for instance from $n = 100$ to $n = 1000000$ that is not feasible for us with the given time constraints. A value of 100 for n would cause 100 times more updates than the example value we presented above. We know that on the test machine computing the histogram for one partition of 200000 rows took approximately 0.16 seconds. It would take roughly 5 hours to compute the histograms alone if that value was tested for rule two. Moreover, we would still have to test the rest of the parameters for this rule, as well as all other rules. That is simply not feasible for us. Therefore the range of values we will test for this rule is from 10000 up to 40000000. We will start with the most sensitive parameter values and then progress up becoming more and more insensitive to change with the final parameter value being 4000 times less sensitive than the first value.

We were confident even before we saw the results in figure 7.4 that the optimal choice of n would be 400000 in our use-case. This is due to the way our test is structured; performing all our data manipulation, then a set of queries then a new set of manipulations and a new set of queries, continuing until all partitions have been used. We have previously stated that the ”Perfect histogram” rule is optimal in our use-case because it gives perfect histograms for all queries with the least amount of histogram updates. If n is exactly equal to the partition size then rule two is equivalent to the ”Perfect histogram” rule, in other words, it is at its optimal when n is equal to the partition size.

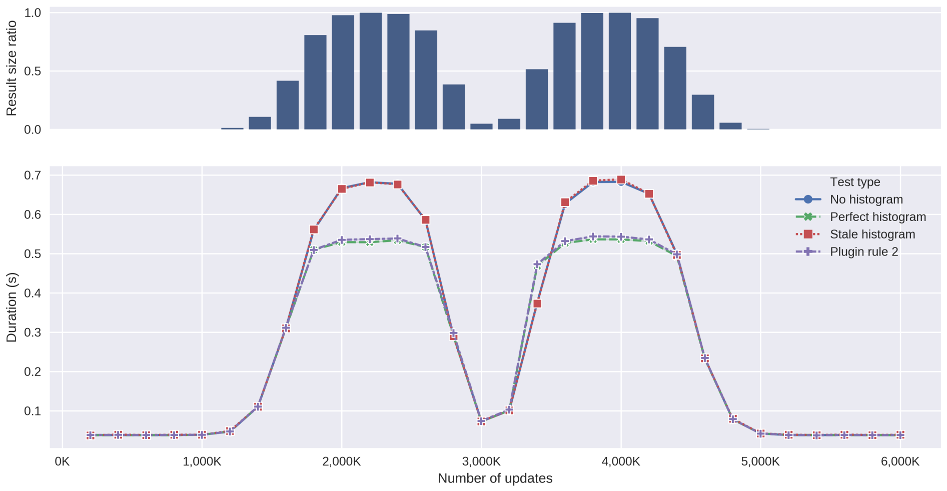

Figure 7.5: Results of timing a query with four joins and where `MSM_VALUE` is between 80 and 100 for rule two using $n = 400000$ as parameter value

In figure 7.4 we can see that our predictions about optimal values for n were correct. When n has a value of 400000 then the total duration, shown as dashed lines, for rule two, the blue line, comes very close to the red line of the ”Perfect histogram” rule, and no other values seem to be quite as close. Thus we have confirmed that $n = 400000$ is optimal in our use-case, we will use this information to reduce the space we have to search when trying to find optimal values for rules that have several parameters. If we also inspect the Q-ratio of rule two we find that the run of rule two with the lowest Q-ratio is the one with $n = 400000$ and a Q-ratio of 0.986, which confirms the results we see in the sensitivity plot nicely. Perhaps more interesting than finding the optimal value in our use-case, is what happens to the efficiency of rule two when we start to move away from that value.

As we decrease the value of n away from the optimal, we see that the total duration increases while the query duration closely follows that of the ”Perfect histogram” rule. In other words, as we decrease n, the database spends an increasing amount of time on enforcing the rule without saving any additional time on query execution. If we instead increase the value of n above the optimal we see the query duration, indicated by a whole drawn line, start to increase while the total duration also increases as an effect but with a shallower angle. This shows that the database is spending less time computing histograms but more time executing queries and that the time saved on computing fewer histograms is less than the time penalty incurred in the form of increased query duration. We also see that when n becomes way too large, the rule ends up never updating the histogram and we achieve a total duration that is equal to that of the stale histogram - which makes sense since the histogram is not updated.

In figure 7.5 we have plotted the execution time at each of the thirty partitions with $n = 400000$ as the parameter value. From the graphs in that figure, we see that as the query starts to return more rows the execution time increases for all the rules which we would expect. However, there are four distinct points where the execution time of the ”Stale histogram" and "No histogram" is considerably higher than that of the "Perfect histogram" and rule two. Indicating that as the selectivity of the `WHERE` predicate in the query changes the two updating rules can "keep up" with this change and cause the query plan to change to account for the increase/decrease in return cardinality of the `MEASUREMENT` table. This is corroborated by the query plans which are different between the two sets of lines. In the graph, this behaviour happens twice since the predicate is a range predicate, once when the value in the column rises, and once on its way back down.

The conclusion we can draw from these two figures is then, too low values of n will cause slowdowns on DML statements but not affect query duration to any significant degree. Furthermore too large values will increase query duration beyond that which was achieved when no histograms were available but not beyond that of stale histograms. Which means; it is better to choose a value of n that is too high rather than one that is too low since there is an upper bound on total duration when n becomes very large. We see that in the graph, the duration of rule two does not exceed that of the "Stale histogram" rule. However, too low values appear to have a very high upper bound, i.e. that of computing histograms for every data change. Also, when suitable values of n are chosen, the rule manages to keep up with the "Perfect histogram" rule by maintaining a similar total duration.

***Rule three*, after n DML statements also considering statement type**

Rule three has three different parameters that we can tune to affect its behaviour under different conditions. These three are; the number of DML statements between updates, n, the weighting given to insert statements, w_i and finally the weighting given to delete statements, w_d. We do not have to determine which weight we will give to update statements since that is implicit in the weighting given to inserts and updates.

We are not well equipped to test rule three in our use-case because of how changes to the data are implemented. As has been discussed earlier, the dataset used in this project is split into partitions. Each of those is then inserted into the database and queries are run querying data from each partition. Once the queries are done, the next partition is inserted while the old one is removed. This gives the `MEASUREMENT` table a constant cardinality throughout the tests and enables the distribution of values in the `MSM_VALUE` column to change as the test progresses. These were attributes that were required of our use-case if we were to be able to measure how stale histograms influence the execution time of queries. However, these attributes are problematic when attempting to measure the effectiveness of rule three.

The problem with these attributes regarding rule three is two-fold, first, only inserts and deletes are performed; there are no updates in the load the database sees. Meaning that it is not possible to test the rule thoroughly since we can not see how the different weight of inserts and deletes influence the inferred weight of updates. Second, at all partitions, except the first one since there is no previous partition to remove, the same number of inserts and deletes are performed. This conflicts with one of the critical assumptions of rule three, namely that different types of statements have different degrees of importance to the histogram accuracy. Now since there are precisely as many deletes as there are inserts, and since each of them modifies precisely one row, it is not visible to the rule if they have different levels of importance. If we can not determine which is the most important, then that means that we should regard them as equally important, and w_i should be equal to w_d. Also, since there are no update statements that means that every statement this rule encounters is given the same weight. When we do that there is no longer any difference between rule three and rule two. They both use n to determine how many statements there should be between updates, and they use the same weighting for all types of statements. The only difference is that for rule three, it is possible to change the weighting given to statements, while for rule two that weighting is always one. However, that behaviour can be replicated in rule two by changing n to increase/decrease the weighting given to each statement implicitly.

There would then have to be made changes in the use-case used throughout this project if rule three was to be adequately tested. One of the first things that would have to be done would be to include update statements so that all three types of statements are included. Then, during the data update portion of each partition, the number of each statement type would have to be varied, so that there is not $1/3$ updates, $1/3$ deletes and $1/3$ inserts. While the number of statements is varied the cardinality of the `MEASUREMENT` table would have to be consistent, or at least close to consistent between the different partitions. This is required to reduce the number of variables influencing query execution time to a minimum. Due to the time constraints of this project, we have not attempted to achieve this in our use-case. Because this has not been done rule three will have the same results as rule two in our use-case. As such, we will not present any results from testing rule three.

***Rule six*, after a ratio r between table size and updated rows is reached**

Rule six disconnects itself from updating the histogram based on only the number of changed rows. Instead, it uses the ratio of updated rows to table size when determining when a histogram becomes stale. In subsection 6.2.4 we presented how this rule was implemented, namely by comparing the current ratio of changed rows, r_c, to the ratio required for update, r, by requiring that when $r_c > r$ the histogram must be updated, if not, it is not updated. The question that arises from this is then, which values of r should be tested, and which value will be optimal for this use-case?

In chapter 3 how state of the art systems handled histogram invalidation and updating was reviewed, and it was discovered that both Oracle and PostgreSQL update histograms after 10% of rows are changed[6]. In the summary of chapter 3, we stated that since these are the values used by systems we consider to be industry-leading that those values should be regarded as good heuristics, at least until proven otherwise. For our tests then, we most definitely want to make sure that a ratio r of 0.1 is included. Ideally we want to start a bit lower than that, and we also want to try higher values. We, therefore, chose to test at different intervals from $r = 0.05$ up to and including $r = 100$ and focusing on values below and around 1. In figure 7.6 we show how rule six performs with different values for its parameter

We can see from the sensitivity graph that lower values of r result in a more noticeable difference between the query execution time and the total execution time. This is because the lower the value of r is, the quicker the histogram will be invalidated and a new one computed. Thus the rule spends more time on computing the histograms. As we increase

[6]PostgreSQL also required that a minimum of 50 rows were updated.

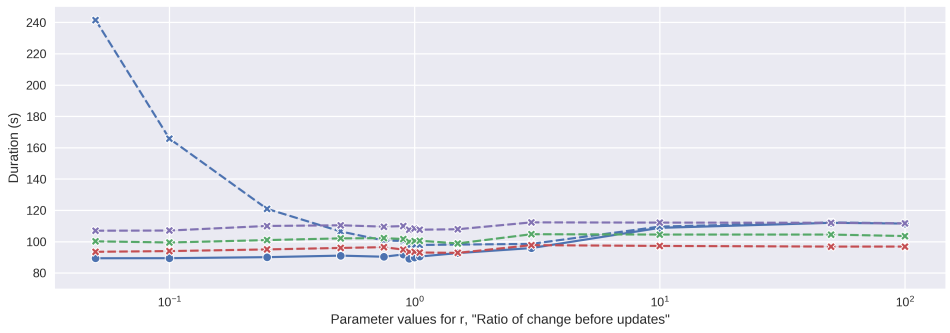

Figure 7.6: The sensitivity to change of parameter values for rule six

the value of r we see that the total duration starts to decrease, and as r exceeds a value of 0.5 we can see that the performance is improved compared to the stale histograms. As we continue to increase r and by extent, the time between updates, the query duration starts to increase as well. In the end, we obtain the same total duration for rule six as we do for stale histograms, indicating that the histogram is never updated.

To determine which parameter value is the most optimal for our use-case, we examine the Q-ratio for rule six with the different parameters. We then find that the lowest Q-ratio is 1.024 with a parameter value of 0.95. If we examine figure 7.6 at that point we see that it supports the results form the Q-ratio nicely, i.e. it seems like the rule is performing very well at that point, being as close to the red line as it gets. We plot the results of one of our queries using rule six as the histogram updating rule using that same parameter value in figure 7.7. Several things are interesting about the lower graph in this figure. First, we see that all three histogram types are outperforming the ”No histogram” rule in the beginning. Second is that ”Plugin rule 6” or rule six, is keeping up with the ”Perfect histogram” rule. That is not very surprising, since with a r value of 0.95 the histogram will on average be updated more than twice[7] during each partition. Thus the query-optimiser always has access to accurate histograms. Third, we see one of the reasons why the ”Stale histogram” rule was performing worse than the ”No histogram” rule during our presentation of table 7.1 previously in this chapter. As the result size of the query decreases, the query-plan that the stale histogram is causing the optimiser to choose, is becoming increasingly poor compared to the other plans. The advantage gained during the first roughly one million updates by creating a stale histogram is lost when the distribution changes. We see from the graph that the advantage gained by the stale histogram over no histogram in the beginning, is significantly less than the penalty incurred when the histogram goes stale between the first million and five million updates. It is clear how the accurate histograms are outperforming the stale, the optimiser can choose good plans for every query, and change to other plans as the distribution of values in the `MSM_VALUE` column warrants it.

For rule six we will then conclude that; too low values of r are detrimental to database

[7]Remember that each partition performs 200000 insert operations and 200000 delete operations, meaning that with $r = 0.95$ and a table size of 200000 the histogram will be marked as stale a little more than twice for each partition.

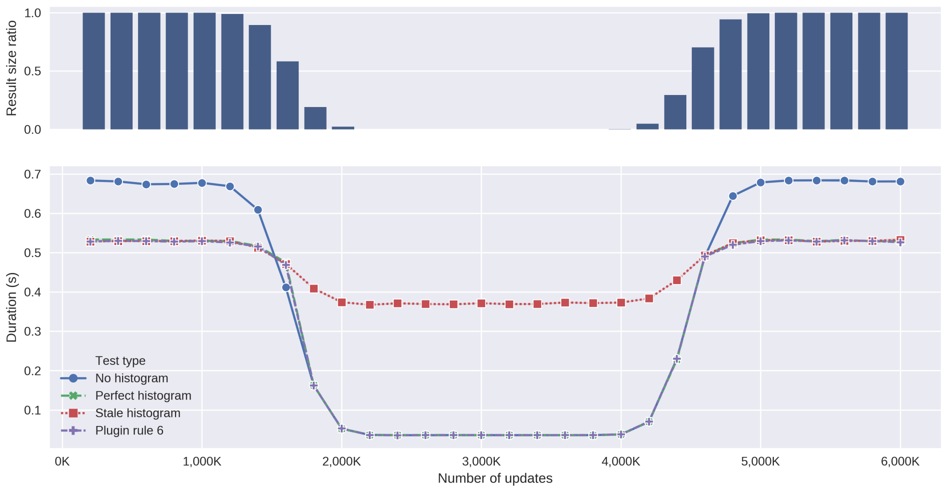

Figure 7.7: Results of timing a query with four joins and where `MSM_VALUE` < 80 for rule six using $r = 0.95$ as the parameter value

performance since there is little restriction on how long time the rule can spend updating histograms. As a result, it is better to choose too high values of r since the worst performance that is achieved then is that of the ”Stale histogram”. Further, there is a broad range for r where the rule can improve on performance compared to ”Stale histogram” and a good choice of r allows rule six to keep up with the ”Perfect histogram” rule throughout the testing suite. An additional benefit of rule six is that it normalises the amount of change by always comparing the change performed to the size of the table. Making the rule more generally applicable since it will handle tables of differing size more gracefully than for instance rule two will.

***Rule seven*, after n rows are updated, comparing the change against histogram boundaries**

Rule seven is designed to weight changes to data that result in values outside the range of the histogram more than changes which result in values inside the histogram range. To do that it uses two parameters, n, the number of statements between updates and w_{oh}, the weight given to a statement which results in a value outside the histogram. As a statement arrives, it is checked to see if it will change data such that the final value is inside or outside the histogram, if outside then the statement counts as w_{oh} statements instead of just one.

When deciding which range of values to use for n and w_{oh}, we will, in essence, be creating a two-dimensional grid of values we will have to experiment with using as parameters. Ideally, we would like to test as many values as we can, and also with as significant differences as we can. However, that is not possible given the time constraints of this project. If we wish to test for more values in one dimension of the grid, we have to reduce the number of values on the other axis. If we consider for a moment the two different

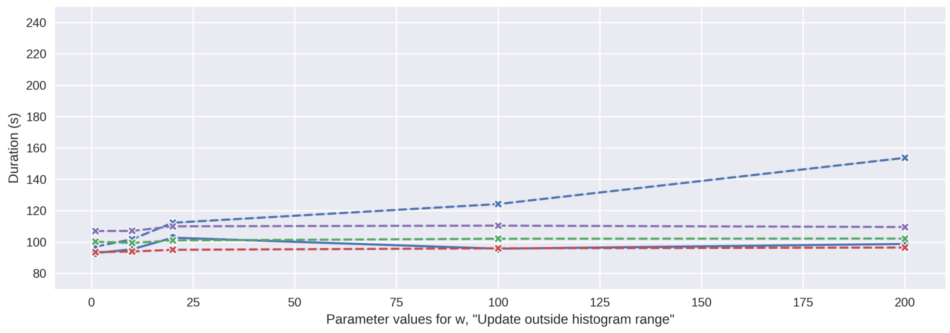

Figure 7.8: The sensitivity to change of parameter values for rule seven

parameters used by this rule it can be argued that the most interesting parameter to test for in this use-case is w_{oh}. That is because we have already determined that for rule two the optimal choice of n was 400000, now since rule seven uses that parameter in the same way as rule two does it makes sense that rule sevens optimal choice of n might be the same as rule two's.

With that in mind, we tested rule seven with different values of w_{oh} ranging from 1 to 200, while locking n to 400000 for all the tests. We are thus reducing the number of dimensions to test and allowing more testing of the interesting parameter of rule seven. Of course, with a constant value of n for all tests, there might be properties of rule seven that will not be discovered through this testing. This is something we choose to accept; however, as a trade-off which allowed us to test better the parameter of rule seven which we believe is the most interesting. In figure 7.8 the results of testing rule seven with our selection of different parameter values for w_{oh} are shown. Note that this graph also has a single data point that conflicts with the others. At point three on the x-axis, the query duration suddenly jumps up to that of the "No histogram" rule, before coming back down again to where we expect it to be at the "Perfect histogram". Point three is an outlier since it makes no sense for the query duration to increase as we update the histogram more often. Moreover, the point does not affect the trend we can see clearly in the graph. It has been challenging to achieve "perfect" results for every data point in all our rules. Our tests take more than 48 hours to complete, and all it takes to create one or two of the inconsistent data points is a small error of a few seconds during a few partitions for a rule. The data point discussed here is, unfortunately, an example of such an event happening. However, we strongly believe that this has *no* effect on the validity and reliability of the results of this project, and argue that with more time for testing it would be no problem to achieve results that do not contain these outliers.

In figure 7.8 we can see that the optimal value for w_{oh} seems to be one. Now that is not as surprising as it might seem at first, remember that we chose to use $n = 400000$ as the other parameter for all the rule seven tests since it was optimal for rule two. Now if rule seven weights statements that change data outside the histogram the same as it weights statements inside, then it is, in essence, a glorified version of rule two. And we already knew that rule two was at it best when $n = 400000$, and the weight of statements

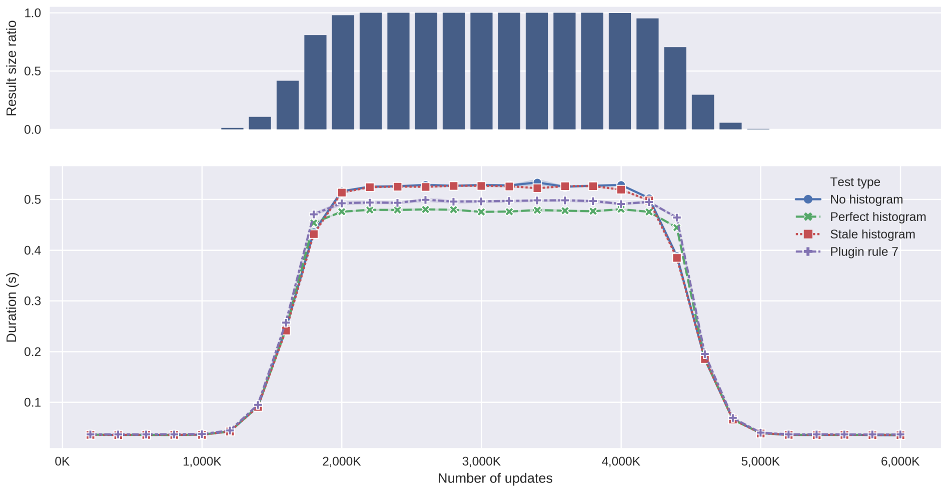

Figure 7.9: Results of timing a query with three joins and where `MSM_VALUE` > 80 for the base classes and rule seven. Using $w_{oh} = 1$ as parameter value.

was one, that was why we chose that n. Which is what happens with rule seven when $w_{oh} = 1$, but if we look at what happens when $w_{oh} = 10$, the results become a bit more interesting. If the weight of all statements were multiplied by ten then that would result in ten times as many histogram updates, but it has not, there has only been performed 15 additional updates. Furthermore, even when we double that to $w_{oh} = 20$, there is only a slight increase again of 18 more updates. It is not before we start to turn up the weight that the execution times begin to increase drastically. Indicating that most data change happens within the histogram, and not outside it.

In figure 7.9 the execution time of one query with a parameter value of $w_{oh} = 1$ is plotted. As our sensitivity plot indicated when the correct parameter value is used rule seven performs as well as the "Perfect histogram" rule. Following it closely as the distribution of values changes throughout the test. In the plot shown here, the parameter value used resulted in a Q-ratio of 1.020 for rule seven, showing yet again that with correct choices of parameter values our rules are performing well.

Regarding rule seven, we conclude that choosing a value for w_{oh} that is too high is worse than choosing one that is too low. The reasoning for this is the same as it was for both rule two and six, there are little restrictions on the possible amount of time that can be spent computing histograms if the rule is too sensitive. On the other hand, if it is too insensitive, at least the situation is not any worse than what it was when no updating was performed. We also see that when n is kept constant and w_{oh} is the only varied parameter, weighting updates outside the histogram significantly more than updates inside the histogram has little impact on the performance of the rule. This could, in some cases, lead to a potential increase in accuracy without much extra overhead.

Rule nine, after estimated cost of inaccurate histogram exceeds a limit I

Rule nine bases the decision of updating the histogram or not on a function which calculates the "cost" of using the current histogram by using different weights which can be tuned using parameters and comparing that to an inverse sensitivity of change parameter. There are four parameters in total used for the function implemented for rule nine, the insert weight w_i, the delete weight w_d, the weight of statements updating rows outside the histogram w_{oh} and finally the inverse sensitivity of change I.

Attempting to find the optimum combination of these values for the use-case presented in this project would involve searching in a four-dimensional space for a single optimum value. If we attempted to test only five values for each of these four parameters that would mean we would have to test 625 different combinations. As with rule seven, this is not feasible with the given time constraints. To achieve a satisfactory level of testing for rule nine, the number of dimensions has to be reduced to something more manageable, in our case one.

For rule nine, the thinking behind the two statement type weights, w_i and w_d is to give different statement types different levels of importance. The reasoning behind that is the same as it is for rule three, however, when evaluating rule three above we decided that having different weights for these two types of statements did not make sense in our use-case. The same arguments can be applied to rule nine, and these two weights should be the same for this rule as well. Also, we argued for rule three that experimenting with different values of these weights made little sense as well, since the n parameter could emulate the change to updating schedule that would be caused by that. To a certain degree that change can be emulated by the I parameter in this rule. Thus we will not be testing different values for w_i or w_d and will be using a weight of one as a default value. Next, w_{oh} is the weighting given to statements that affect data outside the histogram boundaries, just as it does in rule seven. In an attempt to further reduce the number of dimensions that require testing, we wish to avoid testing this parameter as well. In rule seven, we saw that even with relatively significant changes in w_{oh}, it resulted in few histogram updates, which indicates that in our use-case updates do not happen outside the histogram exceptionally frequently. We, therefore, argue that in this use-case changing this value will have relatively little effect on the number of updates caused by rule nine. Therefore this value is also set to a default value of one for the testing conducted. We then only have one remaining parameter, namely I, which we need to test different values for. In our implementation of rule nine, the cost function is as such:

$$r \times rule\_9\_counter > I \tag{7.1}$$

Where r is computed as the ratio between the number of altered rows and table size since the last histogram update. And $rule\_9\_counter$ applies the weights w_i,w_d and w_{oh} to each statement and computes the sum. As an example, if the plugin receives an insert statement that is within the histogram that would increase $rule\_9\_counter$ by w_i. Which values we choose for the weights w_i, w_d and w_{oh} is, as we have discussed previously, not visible in our case, what is interesting is the choice of value for I. However, since the value of I is not something which can be related to the number statements as easily and directly as n and r can choosing its range is more difficult. Its range is therefore decided

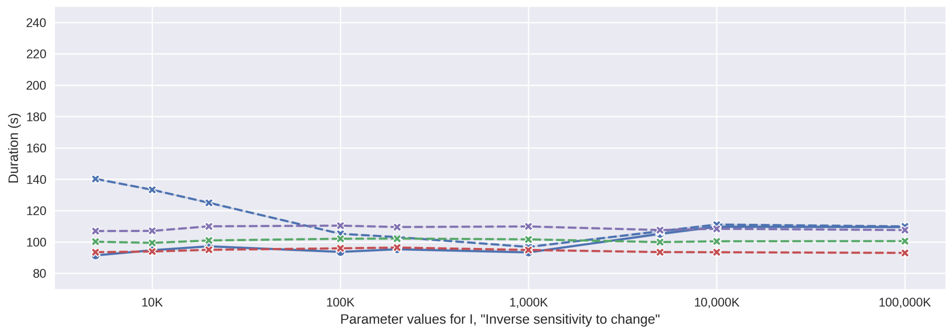

Figure 7.10: The sensitivity to change of parameter values for rule nine

upon through experimentation, meaning that we continued to increase the range of values in both directions until it performed worse than the base class rules or we reached a steady state where it did not change anymore. This resulted in a range of values from 5000 to 100000000, which can be seen in the x-axis of figure 7.10.

When studying figure 7.10 it is clear that it behaves similarly to rule two and six. Too low values of I, i.e. a high sensitivity to change, cause updates of histograms to occur too often, resulting in long total execution time and a low query execution time. As the sensitivity to change is decreased, by increasing the value of I, fewer and fewer updates are performed, until so few are performed that queries are often optimised with a stale histogram, causing the execution time to increase. In figure 7.10 we see this has started to happen around $I = 5000000$. By studying the graph, it can be seen that the best value for I we found in our use-case is 1000000, upon also investigating the Q-ratio we find that a parameter value of 1000000 gives the lowest Q-ratio, which supports the results from the figure nicely. However, upon closer inspection of the Q-ratio table, we find that the rule only executed 28 histogram updates. Which means that it is, in fact, a bit too insensitive with that choice of I since it needs to perform 30 updates to always run with perfect histograms. Which means that we could have increased the performance of this rule slightly if we had managed to find the optimal value of I. However, we did not have time to experiment further with more values for I, and the possible performance increase is not substantial. In figure 7.11 we have plotted how this rule behaves for a given query with the chosen value of I.

We see in figure 7.11, that when correct parameter values are used the histogram accuracy is maintained to a sufficient level such that the query-optimiser can choose the ”optimal” query plans for the queries in the test set. Thus the execution time of the query shown here using rule nine as an updating scheme closely follows the time achieved with the ”Perfect histogram” scheme, which is precisely how the other rules also performed when sufficiently good values of the parameters were chosen. We stated in the previous paragraph that the optimal value we have found for I is a bit too insensitive, we can see that around update number 2.5 million in figure 7.11. The rule updates the histogram one partition after the ”Perfect histogram” rule updated the histogram. As a consequence, the database was using a stale histogram for and extra partition before it was updated and the

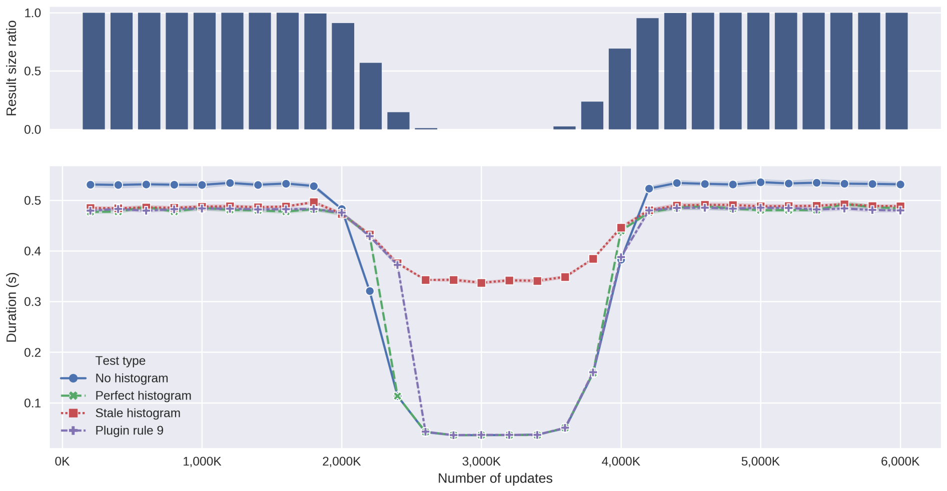

Figure 7.11: Results of timing a query with four joins and where `MSM_VALUE` < 90 for rule nine using $I = 1000000$ as parameter value

query plan changed.

7.3 Discussion

In this section, we start by reviewing some choices we made during our evaluation, and the influence they might have on the validity of our results. We also highlight the reasoning behind some of our choices. We then compare the rules we have implemented to discover advantages with each rule and determine if any performed significantly better in our use-case.

In chapter 4 we used a set of design requirements to define critical attributes of our use-case such that it would be susceptible to changes in the accuracy of histograms. We also explained that if the use-case did not meet these requirements, it was not possible to see how the change in histogram accuracy was having an effect execution time of queries. The use-case that forms the basis on which these tests are conducted is then of vital importance when attempting to decide to which degree the results seen here can be applied to other databases. In other words, is the use-case presented during this project so quirky that the results we have seen here can not be applied generally? We would argue both yes and no. No, because it is clear that no matter the data structure or dataset used in any database if the selectivity estimate that the histogram provides is wrong, then the query plan chosen may be sub-optimal. And sub-optimal query plans result in a decrease in database performance. However, there is no way to guarantee that databases with different data structures, datasets and workloads will experience the same results when histograms go stale as we have shown here. They will not necessarily see that no histograms are outperforming stale histograms, or that a r value of 0.95 is optimal for rule six. So yes, some results presented from this use-case can not be applied generally, and therefore it can be considered too quirky in those aspects. However, we argue that those aspects are not very important. What is important is that; we have shown that maintaining histogram accuracy has its merits, that it does not have to be exceedingly difficult and that whatever one chooses as a parameter value for any of the rules we have implemented can be made into a poor choice. To do that, all that is needed is to change the workload in a particular manner. We will get back to the implications of parameter choice sensitivity in our conclusion.

During the evaluation presented, it was necessary to reduce the number of unknown factors influencing different rules. In particular, this pertains to rule three, seven and nine where we removed some or all of the parameters in the rule to allow us to find ”optimum” values for the remaining parameter. While we believe our reasoning for the choices of parameter values to avoid testing is sound and that the effects of doing so are relatively small in our use-case, this can not be guaranteed. There may be elements we have not sufficiently taken into consideration or which we do not fully understand the impact of. We do, however, still believe the results of testing the implemented rules and the conclusions drawn about the different rules and how they are affected by a given parameter to be reliable and accurate. Having to reduce the number of parameters used for testing does highlight one key issue with rules using parameters, and especially those with several parameters. Namely that having many parameters to tune makes it difficult know how a change in value will influence when the histogram is updated, and in turn make it more challenging to choose good parameter values and test a broad range of those values. The fewer parameters a rule has, the easier it is to relate a change in value to a change in when the histogram is updated, which hopefully should result in better choices of values.

During the evaluation of the different rules we have presented the partition plots, the first graphs we explained and showed in figure 7.1, we have been presenting them with different queries. This was intentional as we wished to showcase how the different amount of join predicates affected the overall time, and how the different `WHERE` predicates affected the shape of the graph. We see that in figure 7.7, we start with ”No histograms” being the slowest by a little margin, but that the ”Stale histogram” becomes much slower than all the other rules as the distribution changes drastically towards the middle of the graph and no rows are being returned. In figure 7.1 however, all rules start off being equally fast and the ”Stale histogram” never performs worse than ”No histogram”, it is only ”Perfect histogram” which performs better than the two when the distribution changes drastically towards the middle. This shows how the predicate influences the execution time as the distribution changes and that in some queries stale and no histograms perform equally well, while in others no histograms are substantially better.

7.3.1 Comparing tested rules

In table 7.2, we have listed the best performing versions of each rule as categorised by the Q-ratio. Higher values indicate less efficient rules and lower values indicate more efficient rules. The table also contains a row for rule zero, this rule uses no histograms, and the plugin performs no other action than checking whether or not it should update the histogram, which it never does for rule zero. It serves to validate that there is little to no penalty in overall performance by having the plugin active, which it does nicely since there is very little difference between the total duration of it and the ”No histogram” rule[8]. In this table, we see that there are two rules which are performing better than the ”Perfect histogram” rule, which should not be possible if that rule is optimal. However, it is important to note that the two rules only perform better because they have a lower query duration, which is caused by fluctuations during our testing. We stated earlier that the query duration given for the base classes is the mean of several runs since each run is not affected by the different parameters supplied. This means that we are very confident that the values we see for ”Perfect histogram”, ”No histogram” and ”Stale histogram” are representative. While for the rules, there is some fluctuation in the query duration since we do not perform several runs of each rule with the same parameter values. It is clear, however, that only a handful of seconds are separating the ”Perfect histogram” rule from the other two, indicating that the variance between runs is low enough to be disregarded. Also, these tests have been performed several times during the project, and the results from the different runs have not been contradicting. Further indicating that the results we present here are reliable.

Test type	Total duration(s)	Q-ratio	Query duration(s)	Histogram duration(s)	Number of updates	Supplied parameters
Rule 6	809.7	0.983	801.6	8.2	62	0.95
Rule 2	812.5	0.986	808.6	4.0	30	400000
Perfect histogram	823.9	1.000	819.9	4.0	30	Not applicable
Rule 7	840.5	1.020	836.4	4.1	30	1.0
Rule 9	843.8	1.024	840.3	3.5	26	1000000
No histogram	915.2	1.111	915.2	0.0	0	Not applicable
Rule 0	921.4	1.118	921.4	0.0	0	Not applicable
Stale histogram	984.8	1.195	984.7	0.1	1	Not applicable

Table 7.2: Table with Q-ratio and parameter data for the slowest and fastest version of each rule

We see that rules two and six, and nine seven have very similar Q-ratios, indicating that with optimal choices of parameter values, all four rules are very close to equally efficient. While several factors constitute what makes an optimal choice of parameter values for all the tested rules, both rule two and seven have the table size as a factor. Since they only compare the number of updated rows, or equivalently in this use-case the number of statements, to a fixed number, a table with many rows will have its histogram updated just as frequently as a table with very few rows[9]. This is not a desired attribute since the ratio

[8]Rule zero was also the rule used to plot the blue line in figure 7.2, which was used to present that graph type and the legend.

[9]Assuming that the same number of rows are changed in both tables.

of change is of critical importance when attempting to decide whether or not a histogram has gone stale.

In other words, rule six and nine are more tolerant of changes in the database than what rule two and seven are without sacrificing any performance when optimal values are chosen. Beyond that we have seen in the sensitivity plot for rule nine shown in figure 7.10 that even though the values for the I parameter span an extensive range from 5000 up to a value more than five orders of magnitude larger, the total duration for rule nine is quite stable. If we compare this with the range of values used when testing rule six, we see that range is quite a bit smaller - from 0.05 to a value only roughly three orders of magnitude larger. This indicates that while all rules perform equally well with good parameter values, rule nine is the most forgiving in terms of the range of ”good” values, as well as the extremes that have to be approached to obtain inferior total duration times.

We, therefore, conclude that while our rules perform equally well with optimal parameters, we would recommend either rule nine or six as the ones to implement. With rule six having the advantage of being a more simple rule to both implement and understand, which makes it easier to tune the parameter to a good value. The advantage of rule nine, on the other hand, is an increased range of acceptable values resulting in the importance of choosing a value close to the optimal not being as significant.

7.4 Summary

In this chapter, the results of testing the implemented rules have been presented, and we would like first to point out that our testing shows without a doubt that in our use-case a stale histogram performs significantly worse than not having a histogram at all. It has also been shown that while the choice of parameter value is important for the different rules, there is often av large interval of values which will improve performance compared to stale histograms at a particular database with a particular workload. We have also discussed how different data structures, datasets and workloads influence which choice of parameter values are good and which are poor. Also, we have seen that at one end of a parameter value the influence on database performance is bound by an upper limit, while on the other end there is virtually no limit to how negative the impact of the different rules can be. We also saw that rule six and nine have the largest range of values in which updating histograms improve database performance. In the next chapter we will be using what we learned from this chapter to apply some of the results of our tests to more general cases, and present our recommendations for how to deal with histogram accuracy and what we believe warrants research in the future.

Chapter 8

Conclusions and future work

In this project, we have been aiming to define attributes that must be met for database performance to be affected by histogram accuracy, and define rules for when to update said histogram. As the final chapter in this report, the following pages will be used to present the following three items. The required attributes of a database if it is to be affected by histogram accuracy, the conclusions we have come to on the histogram updating rules we have tested, and some of our thoughts and insights regarding histogram updating. The chapter concludes with proposing subjects we believe warrant further study.

8.1 Effects of histograms on query execution times

Based on the results presented from our testing in the previous chapter and the design requirements discovered in chapter 5, our conclusions on the effects of histograms on query execution times are two-fold. First, we present our list of requirements for a database to be affected by histogram accuracy, then we present how histograms affect query execution times and what we believe should be done to handle those effects.

When are query plans effected by histogram accuracy

In chapter 5 we presented how we discovered, and in chapter 4 we listed and adhered to, what we labelled as design requirements for databases if they are to be affected by histogram accuracy. We found that there are requirements both to the data which populates the database, the workload and the data model. These requirements are listed below.

- The distribution of values within a column which has a histogram approximating its distribution must change in such a way that the approximate distribution the histogram provides will become incorrect if the histogram is not maintained. In other words, the distribution of underlying values for all histograms must change and must change such that each particular histogram becomes *stale* at some point. How the distribution(s) change also affects query execution times. We saw during development and testing of the use-case that as distributions became very skewed and started moving towards other skewed distributions the difference in execution times could become quite high as the stale histogram predicted the selectivity of the predicate poorly. When we instead started with a uniform distribution and moved towards a skewed one, or the other way around, the differences were not as large. This is caused by the histogram predicting the distribution more accurately than it did with very skewed distributions. This behaviour is not that surprising and indicates that it is the degree of error in the selectivity approximation that determines, to a large degree, how significant the influence of the histogram is.

- The queries that the database receives must be such that the optimiser can utilise histograms when creating the query plans. In the case of MySQL, this means that the `WHERE` predicate must contain one, or more, evaluations between a single column and a constant and that there are no indexes on any columns used in the predicate. For other RDBMS' other requirements may apply.
- There has to exist query plans with different join orderings for queries being optimised such that as the cardinality of tables changes, their position in the join order may change as well. This also requires that the differences in table sizes are such that differences in selectivity can cause different join orderings of tables. The change of order is caused by the histogram providing selectivity estimates for predicates, which are in turn used to determine how the predicate influences the cardinality of a table. Another important realisation regarding join orderings is that while having different join orderings that can be chosen based on selectivity determines *if*, a histogram will influence query execution time. It is how far the histogram causes tables to move in the join order which determines to what *extent* the histogram will influence execution time. The data model of the database in question determines the possible join orderings. Through our testing, we have found that the linear join model, while enabling fewer possible join orders, enables orders where the movement is more varied and larger than that of a star schema model.

Using these requirements, we were able to create a use-case in which the accuracy of histograms affected query execution times to a noticeable degree.

How will histogram accuracy affect query execution times

We wish to begin by pointing out that, in specific use-cases, like the one we have presented in this report, we have shown that histograms *will* affect query execution times. The extent of this effect and whether it can be regarded as positive or negative will depend on the accuracy of the histogram and database attributes. We saw from the graphs in the previous chapter and the data in table 7.2 that; having a histogram is not always better. We have managed to create a use-case in which stale histograms perform significantly worse than accurate histograms and even no histograms! It might be surprising to some that we saw the total execution time of all our queries increased by 19.5% when using a stale histogram and only 11.1% with no histograms compared to perfect histograms. Meaning that in our use-case it is better not to have them than to have them and not maintain them. We have also shown this behaviour in action in several of our graphs, e.g. figure 7.7 where we can see that the stale histogram is causing the query-optimiser not to change the join order of tables even though the distribution has changed significantly - resulting in a longer execution time for the query when compared to both the ”Perfect histogram” and ”No histogram” rules.

We have been using the word *stale* as a description of histogram state throughout this chapter and the project, but what does it mean to be a stale histogram? In subsection 2.2.5 we stated that we had not been able to find a general definition for histogram staleness during our literature search. However, in section 4.4 we said that a possible definition of staleness was *once the distribution of the data has changed the histogram is stale* and that this definition does not do us much good since it defines histograms as stale too frequently. We then continued with saying that our rules define staleness as *when the histogram is causing the query-optimiser not to choose the optimal join order*, and as we have seen during our testing, the rules need to be able to adapt to different databases. This reinforces our reasoning from subsection 2.2.5 of why a perfect general definition of staleness does not exist, namely that no one definition is optimal in all cases. Our rules *are* definitions of staleness, each of them is different, and each of them can be altered, and *must* be altered to adapt to different databases, by changing the supplied parameter.

Stating something as general as ”in all cases, a stale histogram will be detrimental to database performance compared to not having a histogram” is not possible. There are too many variables that influence how different join orders affect the execution time of the query to do so. However, we can state that there does *exist* a situation in which a stale histogram is worse than not having one, and in which maintaining histograms is beneficial to overall database performance. We conclude, therefore that it is not a question of whether or not histograms should be updated, but rather when they should be updated.

What should be done about histogram accuracy?

One of the primary objectives of this project has been to investigate how we can or should handle histogram accuracy. In chapter 4, we presented different rules for how we can go about maintaining histogram accuracy. Moreover, in the previous chapter, we studied how those rules, and a set of base rules, termed base cases, performed in our tests. Based on those tests, it became clear that histogram accuracy is *not* something we can dismiss and choose to not handle. Our tests showed as we stated in the previous section, that a neglected histogram with reduced accuracy, i.e. that is feeding the query optimiser false information about predicate selectivity, can cause the query-optimiser to choose sub-optimal join orderings when compared to accurate histograms and even when compared to heuristic values.

Our tests also showed that the rules implemented in our plugin work! When correct parameter values are used all rules improve the accuracy of the histogram to such a degree that the query plan chosen is the same as that which is chosen when the ”Perfect histogram” rule is active. We also showed that if poor values are chosen, the rules will impact overall database performance negatively, causing large amounts of resources to be spent on updating histograms when it is not necessary. As a result, the choice of the parameter value is vital for all rules we have presented, and choosing the optimal value is difficult. However, it is not required that the optimal parameter value be chosen, the range of possible values in which our implemented rules improve database performance is, in many cases, significant. We showed this with our sensitivity plots, in figure 7.6 values of r ranging from 0.25 up to 100 all perform as well or better than the ”Stale histogram” rule.

Moreover, for rule two we were able to reason that the optimal parameter would be 400000 in our use-case, but as we saw in figure 7.4 values of n from 100000 up to 1000000 improved upon the performance of stale histograms. While it is not possible to predict what the optimal value of n or r should be for all cases, it is not needed. Just choosing a value which is in the ballpark of the optimal will result in a performance gain. Even better, as long as the value that is chosen is not causing the rule to be too sensitive, the worst possible performance achievable by these rules is the same as what is achieved by not having the rules - namely that of the stale histogram. I.e. it is better to choose a value that causes the rule to be too insensitive than one which causes the opposite since there is a limit to how bad the rule can be when it is too insensitive, while for too sensitive rules there is almost no limit to how significant the performance impact can be[1].

We would also to like to point out that when MySQL creates a histogram, it uses a sample of the table to compute that histogram if the table size is too large. However, the sampling used is not actual sampling, since it is currently not possible for the underlying storage database of MySQL, *InnoDB*, to return samples of data. The entire table must be fetched from disk and MySQL then uses only a portion of the returned data. This means that updating histograms likely is slower than what it can be in MySQL if proper sampling from disk was performed, in other words, it might be possible to increase the range of acceptable values for parameters further if the sampling in MySQL is improved. In turn, this means that for database systems which can compute histograms faster than what MySQL has been capable of in this project, the range of acceptable parameter values is also increased.

We conclude that if histograms are used, then they should be updated at some point. We have shown that with quite simple measures, the performance of databases can be increased by a significant amount, the updating rule used need not be a complicated one. Using a rule as simple as updating after a percentage of rows has been changed, which is what Oracle and PostgreSQL use as their updating rule, with a reasonable parameter value yielded significant performance increases in our use-case. We can also see from figure 7.6 that the value Oracle and PostgreSQL use is too sensitive in our use-case, letting $r = 10\%$ results in a net performance loss for us. Which was quite surprising, how can it be that the standard values used by what can only be regarded as world-leading state-of-the-art systems perform so poorly in our use-case? This is caused by our use-case having too many DML statements when compared to queries, which results in too sensitive parameters performing exceedingly poorly. The workload used in our use-case is meant to exaggerate the cost of choosing poor values for rules, and as such, it shows that the choice of $r = 0.1$ is a poor one which results in relatively large net performance losses. Exemplifying how different data models, datasets and workloads respond very differently to different parameter values[2], we have also argued that any choice of parameter can be

[1]The limit being when every data change triggers a histogram update. However, at that point, the cost of maintaining the histogram is so tremendous that using the database becomes difficult.

[2]We are quite confident that both Oracle and PostgreSQL performed some sort of evaluation before deciding on their default value of r, which indicates that the differences between their database and ours is significant enough to cause a large difference in what can be considered good values for r.

made optimal by changing datasets, data models or workloads correctly. This enables us to make an important conclusion and outline some of our recommendations regarding updating rules. Rules must either be configurable by users or adaptive, meaning that the rule must either be such that it can be tuned by a user changing a parameter or it must tune itself by adapting to the database and its workload automatically. If rules with parameters are used, then they should have as few and as simple parameters as possible, allowing users to make an informed decision on their choice of parameter. We experienced first hand how difficult it can be to determine correct combinations of parameters when there are too many to test. An excellent choice of a simple rule in our opinion is number six, which only uses the proportion of changed rules and gives users an intuitive parameter to adjust. Alternatively, we would recommend a more complicated rule which hides the complexity from the user and allows them to adjust only one parameter. Our rule nine can be regarded as such a rule, where we have already chosen values for several weights based on our expertise, and allow the users to change a single parameter which determines how sensitive the rule is to change, the I parameter in the case of rule nine - combining the added flexibility of multi-parameter rules with the ease of tuning that single-parameter rules display. In the next section, we present one more variation of a rule with parameters and suggestions for three adaptive rules that can be used.

8.2 Future work

During the development of this project we have been creating and adding rules to our set as we have been progressing, we have however come up with four additional rules during the closing phases of this project which we believe warrant further research. These four rules are presented below, along with what we believe will be their strengths and weaknesses. We also outline possible avenues of research regarding use-cases and when histograms influence query execution time.

***Rule ten*, comparing query execution times.** The goal of histograms in the context of this project is to increase the accuracy of the selectivity estimate used during query optimisation; this is based on the idea that if the query-optimiser has access to more information about a query, it can make a better choice of a query plan. This plan will in-turn result in reduced execution time for the query in question. The rules outlined in this project so far have tried to approximate how events that have occurred since the last histogram update have affected the accuracy of the histogram, and how the change in accuracy is affecting the current execution time of queries. Rule ten is designed to circumvent this approximation by executing a query with the current histogram and register the time the query took. Then by using an optimiser directive, or something similar, rerun the query but this time without the query-optimiser using histograms. The time difference between the two runs of the same query querying the same data can then be compared and used as a basis on which to decide if the histogram should be updated or not. While this approach ensures that histograms are only updated when there is a benefit to doing so, the advantage the histogram provides is compared to not having a histogram, not to a perfect histogram. Meaning that there could be situations in which there would be an advantage to updating the histogram, but it is not done because the current stale histogram is performing as well as no histograms. In such a situation, there could still be that a perfect histogram would result in a better query plan than that of no histogram; however, this rule does not achieve that type of performance increase.

***Rule eleven*, after a ratio r between table size and updated rows is reached also considering the proportion of updated rows versus the number of queries.** In the previous chapter, we argued that one of the significant problems with deciding upon a parameter value for rules is that as the load on the database changes so does the impact of choosing too high or too low values. This is also true for rule two and six, who are in many ways very similar. The key difference being that rule six abstracts away the actual number of updated rows to be more general and perform more uniformly in a variety of use-cases with the same parameter value. The problem remains for rule six however that as the proportion of updated rows becomes very large choosing low values for r becomes increasingly worse, with the inverse also being true. With rule eleven, this behaviour is considered, as the proportion of updated rows increase, the value used for r also increases, and as the proportion decreases, so does the value of r. It would not be an insurmountable task to test how quickly r would have to change with the proportion of updated rows to be able to adapt efficiently to such shifting proportions.

***Rule twelve*, comparing histogram selectivity estimate with actual selectivity.** We know that the query-optimiser uses histograms to provide an estimate for the selectivity of predicates and that this selectivity is used to determine what the cardinality of tables will be after the rows that do not satisfy the predicate are removed. Since any query being optimised will also be executed, the *actual* selectivity of the predicate can be determined. It would then be possible to compare the actual selectivity of the predicate with the approximation given by the histogram, and based on the difference between those two values determine if the histogram requires updating. Such a rule would require changes to both the query-optimiser and the query-executor of MySQL. This would not be natural to implement using a plugin but instead may require implementing into the base of MySQL itself. Nevertheless, it would enable histograms to be checked frequently with little cost, and it also enables us to check the accuracy of complex predicates. It would even be possible to use this rule not only update already existing histograms but create new ones. One could use the difference between the selectivity provided by the heuristic values when no histograms exist and the actual selectivity to determine if a histogram should be created for a particular column. This tactic can be applied to any predicate which has an accompanying selectivity estimate and would allow us to evaluate arbitrarily complex predicates and create multi-dimensional histograms where they are needed.

A potential improvement of rule nine was discovered at the end of the project by changing the ”cost function” used. It was the realisation that actions happening outside the histogram boundaries need not always result in a less accurate histogram. Deleting a row that is outside the histogram will improve the accuracy of the histogram. Changing the ”cost function” to account for this behaviour is achieved by splitting the actions that happen on data into more pieces and adding weight parameters to each of them. A possible new function is presented below.

$$\sum \left(\frac{i_{ih}}{c} \times w_{i\_ih} + \frac{i_{oh}}{c} \times w_{i\_oh} + \frac{u_{ih->ih}}{c} \times w_{u\_ih} + \frac{u_{ih->oh}}{c} \times w_{u\_oh} + \right.$$
$$\left. \frac{u_{oh->oh}}{c} \times 0 + \frac{u_{oh->ih}}{c} \times -w_{u\_oh} + \frac{d_{ih}}{c} \times w_{d\_ih} + \frac{d_{oh}}{c} \times -w_{d\_oh} \right) > I \quad (8.1)$$

In the equation above the variable i_{ih} indicates the number of inserted rows inside the histogram, while i_{oh} indicates the number outside the histogram. Equivalently d_{ih} is the number of deleted rows inside the histogram and d_{oh} is the number of deleted rows outside the histogram. Furthermore, $u_{ih->ih}$ is the number of updated rows in which the value monitored by the histogram was initially within the histogram and after the update is still within the histogram. Also $u_{ih->oh}$ is the number of updated rows which initially had a value withing the histogram but were updated to have a value outside the histogram, the converse is true for $u_{oh->ih}$ and $u_{oh->hh}$. Additionally, c is the cardinality of the table on which the histogram in question exists; all the variables are divided by this number to normalise the number of changed rows. The w variables are the weights given to each data modification, while I is still the inverse sensitivity to change. There are several interesting aspects about how data change affects the histogram accuracy, which we are trying to capture in Equation 8.1, we list them below.

- Inserting rows with values both inside the histogram range and outside will affect the histogram accuracy negatively.
- Updating a row with a value that was inside the histogram to a new value that is still inside the histogram is negative because it can cause the number of elements in buckets to be wrong. Alternatively, if the histogram is sparse[3] the new values might be outside buckets but inside the total range of the histogram.
- Moving the value of a row from inside the histogram to outside harms accuracy, and we consider the effect to be larger than that of updating to a value inside. This is because it is guaranteed that the value outside the histogram is not represented in the histogram. However, if the value is updated to another value inside the histogram, that value might already be represented in a bucket, resulting in only the number of instances of that value being wrong. Which one can argue is not as wrong as not representing the value at all.
- Updating a row with a new value outside the histogram range which already had a value outside the range has essentially no effect on the accuracy.
- Moving a value from outside the histogram to inside has a positive effect on accuracy, and it is as large of an effect as the action of moving a value out.
- Deleting a row with a value that is within the histogram boundaries affects the accuracy negatively.
- Deleting a row with a value outside the histogram has a positive effect on histogram accuracy.

[3] A sparse histogram is a histogram which allows there to be ranges of values not covered by buckets, meaning that there can be a range of values between two neighbouring buckets which is not in any of the two buckets.

In the presented equation, both $u_{oh->ih}$ and d_{oh} are multiplied by negative weights, that is because they increase the accuracy of the histogram. Therefore they reduce the inaccuracy ”cost” of the current histogram. For the equation presented all weights used are meant to be positive numbers. This extended version of the original rule nine has the advantage that it should more accurately model the change of the histograms accuracy since last it was updated. However, it uses six weights to do, which means that finding good parameter values for this rule can be difficult.

The first three rules above and rule eight, which we presented along with our other rules, have a distinct advantage over the ones we have implemented and the potential improvement we suggested for rule nine. Namely that they do not need parameter values to be changed to adapt to different databases, the ability to adapt is built into the rule. By creating rules with the ability to dynamically determine when histograms are stale under different databases, and indeed in the same database but under different workloads, we believe we can create more general and robust rules. Which we think is the most significant advantage of these rules.

Exploring different use-cases. We know now that the data model, dataset and workload determine when a histogram is stale, and in-turn determines which choice of parameter value is right for each of the different updating rules presented. The use-case developed and implemented in this project has proven this; however, there are weaknesses and faults with our use-case, which limited which rules it made sense to test. We believe further experimentation with developing and implementing use-cases is warranted to answer what we believe to be important questions, such as. *Supposing that the data model allows histograms to influence performance, is the workload the defining factor of which definitions of staleness are good and not?* And, *is it possible to alter the use-case presented in this report such that the requirements for databases to be effected by histogram accuracy can be relaxed?*

What would happen for instance, in a use-case which had the same data model and dataset as the one used in this report but different workloads? We have shown that if the relationship between resources spent on queries and number of updated rows changes from what they are in our use-case the results of our tests would be different, but how different would they be? How important is that ratio? Alternatively, what if the ratio was kept the same, but instead of changing the distribution by only using inserts and deletes, we instead used updates? And what if the ratio between the different DML statements varied as we suggested for testing rule three. These variations and many more will influence the rules we have presented, that we are sure of, what we are unsure of is how much.

We believe that experimenting with differing use-cases can help create a more robust and well-defined set of requirements that better define in which situations the histogram accuracy affects database performance, and may also enable us to create more efficient and accurate rules for when to update them.

Bibliography

[1] Amazon, *Overview of Amazon Web Services Databases*. [Online]. Available: https://docs.aws.amazon.com/whitepapers/latest/aws-overview/database.html.

[2] InfluxData, *InfluxDB 1.7 documentation — InfluxData Documentation*. [Online]. Available: https://docs.influxdata.com/influxdb/v1.7/.

[3] S. Vestrheim, "Exploring statistical accuracy in relational database systems", Norwegian University of Science and Technology, Tech. Rep., 2019, p. 49.

[4] R. Elmasri and S. B. Navathe, *Database Systems SEVENTH EDITION*. Pearson, 2016, p. 1242, ISBN: 978-0-13-397077-7. [Online]. Available: http://noahc.me/FundamentalsofDatabaseSystems%287thedition%29.pdf.

[5] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann, "How good are query optimizers, really?", *Proceedings of the VLDB Endowment*, vol. 9, no. 3, pp. 204–215, Nov. 2015, ISSN: 21508097. DOI: 10.14778/2850583.2850594. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2850583.2850594.

[6] Y. Ioannidis, "The History of Histograms (abridged)", in *Proceedings 2003 VLDB Conference*, 2003, pp. 19–30. [Online]. Available: http://www.madgik.di.uoa.gr/sites/default/files/vldb03_pp19-30.pdf.

[7] ——, "Approximations in Database Systems", in *Lecture Notes in Computer Science*, Springer, Berlin, Heidelberg, 2003, pp. 16–30. DOI: 10.1007/3-540-36285-1{_}2. [Online]. Available: http://link.springer.com/10.1007/3-540-36285-1_2.

[8] S. Christodoulakis and Y. E. Ioannidis, "Optimal Histograms for Limiting Worst-Case Error Propagation in the Size of Join Results", *Article in ACM Transactions on Database Systems*, 1993. DOI: 10.1145/169725.169708. [Online]. Available: https://www.researchgate.net/publication/220225465.

[9] S. Chaudhuri, R. Motwani, V. Narasayya, S. Chaudhuri, R. Motwani, and V. Narasayya, "Random sampling for histogram construction", *ACM SIGMOD Record*, vol. 27, no. 2, pp. 436–447, Jun. 1998, ISSN: 01635808. DOI: 10.1145/276305.276343. [Online]. Available: http://portal.acm.org/citation.cfm?doid=276305.276343.

[10] Oracle, *Histograms*, 2019. [Online]. Available: https://docs.oracle.com/en/database/oracle/oracle-database/19/tgsql/histograms.html#GUID-BE10EBFC-FEFC-4530-90DF-1443D9AD9B64.

[11] Microsoft Team, *Statistics - SQL Server — Microsoft Docs*, 2017. [Online]. Available: https://docs.microsoft.com/en-us/sql/relational-databases/statistics/statistics?view=sql-server-2017.

[12] Aerospike, *Aerospike Database Documentation*. [Online]. Available: https://www.aerospike.com/docs/.

[13] Redis, *Redis Database Documentation*. [Online]. Available: https://redis.io/documentation.

[14] Oracle, *Oracle TimesTen In-Memory Database Documentation*. [Online]. Available: https://docs.oracle.com/database/timesten-18.1/.

[15] P. G. Selinger, Astrahan Morton M., D. D. Chamberlin, R. A. Lorie, and T. G. Price, *Access Path Selection in a Relational Database Management System*, 1979. [Online]. Available: https://pages.cs.wisc.edu/~zuyu/summaries/cs764/queryOpt.

[16] V. Poosala and Y. E. Ioannidis, "Selectivity Estimation Without the Attribute Value Independence Assumption", in *Pvldb*, Athens, 1997, pp. 486–495, ISBN: 1-55860-470-7. [Online]. Available: http://dl.acm.org/citation.cfm?id=645923.673638.

[17] N. T. Mit, P. I. Mit, S. Guha, and N. Koudas, "Dynamic Multidimensional Histograms", in *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, 2002, pp. 428–439. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.92.1201&rep=rep1&type=pdf.

[18] P. B. Gibbons, Y. Matias, and V. Poosala, "Fast incremental maintenance of approximate histograms", *ACM Transactions on Database Systems*, vol. 27, no. 3, pp. 261–298, Sep. 2002, ISSN: 03625915. DOI: 10.1145/581751.581753. [Online]. Available: http://portal.acm.org/citation.cfm?doid=581751.581753.

[19] MySQL team, *MySQL :: MySQL 8.0 Reference Manual :: 8.9.6 Optimizer Statistics*, 2019. [Online]. Available: https://dev.mysql.com/doc/refman/8.0/en/optimizer-statistics.html.

[20] TPC, *TPC - Benchmarks*. [Online]. Available: http://www.tpc.org/information/benchmarks.asp.

[21] R. Walpole, R. Myers, S. Myers, and K. Ye, *Probability and Statistics for engineers*, 9th Editio. Pearson, 2016, p. 816, ISBN: 9781292161365. [Online]. Available: `https://www.amazon.com/Probability-Statistics-Engineers-Scientists-Global/dp/1292161361`.

[22] T. Pelkonen, S. Franklin, J. Teller, P. Cavallaro, Q. Huang, J. Meza, and K. Veeraraghavan, "Gorilla: A fast, scalable, in-memory time series database", in *Proceedings of the VLDB Endowment*, Kohala Coast, Hawaii: Association for Computing Machinery, 2015, pp. 1816–1827. DOI: `10.14778/2824032.2824078`.

[23] MySQL team, *MySQL :: MySQL 8.0 Reference Manual :: 13.7.3.1 ANALYZE TABLE Syntax*, 2019. [Online]. Available: `https://dev.mysql.com/doc/refman/8.0/en/analyze-table.html`.

[24] Oracle, *Database Performance Tuning Guide — Managing Optimizer Statistics*, 2008. [Online]. Available: `https://docs.oracle.com/cd/B19306_01/server.102/b14211/stats.htm#g49431`.

[25] M. Colgan, *Extended Statistics — Oracle Optimizer Blog*, 2011. [Online]. Available: `https://blogs.oracle.com/optimizer/extended-statistics`.

[26] T. Kyte, *On Dynamic Sampling — Oracle Magazine*, 2009. [Online]. Available: `https://blogs.oracle.com/oraclemagazine/on-dynamic-sampling`.

[27] N. Bayliss, *Dynamic sampling and its impact on the Optimizer — Oracle Optimizer Blog*, 2010. [Online]. Available: `https://blogs.oracle.com/optimizer/dynamic-sampling-and-its-impact-on-the-optimizer`.

[28] Postgre Team, *PostgreSQL: Documentation: 11: ANALYZE*, 2017. [Online]. Available: `https://www.postgresql.org/docs/current/sql-analyze.html`.

[29] ——, *PostgreSQL: Documentation: 11: Row Estimation Examples*, 2017. [Online]. Available: `https://www.postgresql.org/docs/current/row-estimation-examples.html`.

[30] ——, *PostgreSQL: Documentation: 11: How the Planner Uses Statistics*, 2017. [Online]. Available: `https://www.postgresql.org/docs/current/planner-stats-details.html`.

[31] ——, *PostgreSQL: Documentation: 11: 19.10. Automatic Vacuuming*, 2017. [Online]. Available: `https://www.postgresql.org/docs/current/runtime-config-autovacuum.html#GUC-AUTOVACUUM-NAPTIME`.

[32] ——, *PostgreSQL: Documentation: 11: 24.1. Routine Vacuuming*, 2017. [Online]. Available: `https://www.postgresql.org/docs/current/routine-vacuuming.html#AUTOVACUUM`.

[33] A. Swami and K. B. Schiefer, "On the estimation of join result sizes", in *Advances in Database Technology EDBT '94. EDBT 1994*, Springer, Berlin, Heidelberg, 1994, pp. 287–300. DOI: `10.1007/3-540-57818-8{\_}58`. [Online]. Available: `http://link.springer.com/10.1007/3-540-57818-8_58`.

[34] K. A. Hole and H. O. Eggen, “Cross-shard querying in MySQL”, Norwegian University of Science and Technology, Tech. Rep., 2019, p. 46.

[35] Python, *3.8.0 Documentation*. [Online]. Available: `https://docs.python.org/3/`.

[36] NumPy, *Overview NumPy v1.18.dev Manual*. [Online]. Available: `https://numpy.org/devdocs/`.

[37] Pandas, *pandas: powerful Python data analysis toolkit pandas 0.25.3 documentation*. [Online]. Available: `https://pandas.pydata.org/pandas-docs/stable/`.

[38] Seaborn, *Statistical data visualization Seaborn 0.9.0 documentation*. [Online]. Available: `https://seaborn.pydata.org/`.

Appendix

The appendix included below contains the files created during this project which we consider to be most interesting, this includes the source code to the plugin and rules themselves, the mtr code used to perform our tests, and the python code that generates the data used and which interprets the result file generated by mtr and creates our graphs. These files and all other files created during this project are also available on github[4].

histogram_updater.cc

```
/*  File for the histogram updater plugin containing the system variables
    and corresponding update functions
 *  Also containts the plugin defition and calling functions, as well as
    the actual histogram updating query  */

/*  Copyright (c) 2015, 2017, Oracle and/or its affiliates. All rights
    reserved.

    This program is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License, version 2.0,
    as published by the Free Software Foundation.

    This program is also distributed with certain software (including
    but not limited to OpenSSL) that is licensed under separate terms,
    as designated in a particular file or component or in included license
    documentation.  The authors of MySQL hereby grant you an additional
    permission to link the program and your derivative works with the
    separately licensed software that they have included with MySQL.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License, version 2.0, for more details.

    You should have received a copy of the GNU General Public License
    along with this program; if not, write to the Free Software
    Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301
    USA */

#include <ctype.h>
#include <mysql/components/services/log_builtins.h>
#include <mysql/plugin.h>
#include <mysql/plugin_audit.h>
#include <mysql/psi/mysql_memory.h>
#include <mysql/service_mysql_alloc.h>
#include <string.h>
#include <thread>
```

[4]https://github.com/Vestrheim/mysql-server/tree/testcases

```cpp


#include "my_inttypes.h"
#include "my_psi_config.h"
#include "my_thread.h"  // my_thread_handle needed by mysql_memory.h

#include <iostream>

#include "plugin/histogram_updater/distributed_query_rewriter.h"
#include "plugin/histogram_updater/distributed_query.h"
#include "plugin/histogram_updater/query_acceptance.h"
#include "plugin/histogram_updater/internal_query/internal_query_session.h"
#include "plugin/histogram_updater/internal_query/sql_resultset.h"
#include "plugin/histogram_updater/helpers.h"
#include "plugin/histogram_updater/histogram_updater.h"

#define PLUGIN_NAME "Histrogram Updater"

//Declare internal variables


/// Updater function for the status variable ..._rule.
static void update_rule(MYSQL_THD, SYS_VAR *, void *, const void *save) {
    sys_var_update_rule = *static_cast<const int *>(save);
}

/// Updater function for the status variable ...statements_between_updates.
static void update_statements_between_updates(MYSQL_THD, SYS_VAR *, void *, const void *save) {
    sys_var_statements_between_updates = *static_cast<const int *>(save);
}

/// Updater function for the status variable ...insert_weight.
static void update_insert_weight(MYSQL_THD, SYS_VAR *, void *, const void *save) {
    sys_var_insert_weight = *static_cast<const double *>(save);
}

/// Updater function for the status variable ...delete_weight.
static void update_delete_weight(MYSQL_THD, SYS_VAR *, void *, const void *save) {
    sys_var_delete_weight = *static_cast<const double *>(save);
}

/// Updater function for the status variable ...ratio_for_update.
static void update_ratio_for_updates(MYSQL_THD, SYS_VAR *, void *, const void *save) {
    sys_var_ratio_for_update = *static_cast<const double *>(save);
}

/// Updater function for the status variable ...outside_boundary_weight.
static void update_outside_boundary_weight(MYSQL_THD, SYS_VAR *, void *, const void *save) {
    sys_var_outside_boundary_weight = *static_cast<const double *>(save);
}
```

```

/// Updater function for the status variable ... inverse_sensitivity_to_change.
static void update_inverse_sensitivity_to_change(MYSQL_THD, SYS_VAR *, void *, const void *save) {
    sys_var_inverse_sensitivity_to_change = *static_cast<const int *>(save);
}


static MYSQL_SYSVAR_INT(rule,              // Name.
                        sys_var_update_rule,      // Variable.
                        PLUGIN_VAR_NOCMDARG,  // Not a command-line argument.
                        "Tells " PLUGIN_NAME " what updating rule should be followed",
                        NULL,           // Check function.
                        update_rule,  // Update function.
                        0,              // Default value.
                        0,              // Min value.
                        10,              // Max value.
                        1               // Block size.
                        );

static MYSQL_SYSVAR_INT(statements_between_updates,              // Name.
                        sys_var_statements_between_updates,      // Variable.
                        PLUGIN_VAR_NOCMDARG,  // Not a command-line argument.
                        "Tells " PLUGIN_NAME " how many statements there should be between updates in rule 2, 3 and 7",
                        NULL,           // Check function.
                        update_statements_between_updates,  // Update function.
                        10000,              // Default value.
                        0,              // Min value.
                        INT_MAX,              // Max value.
                        1               // Block size.
                        );

static MYSQL_SYSVAR_DOUBLE(insert_weight,              // Name.
                        sys_var_insert_weight,      // Variable.
                        PLUGIN_VAR_NOCMDARG,  // Not a command-line argument.
                        "Tells " PLUGIN_NAME " how important we think insert statements are",
                        NULL,           // Check function.
                        update_insert_weight,  // Update function.
                        5,              // Default value.
                        0,              // Min value.
                        INT_MAX,              // Max value.
                        1               // Block size.
                        );

static MYSQL_SYSVAR_DOUBLE(delete_weight,              // Name.
                        sys_var_delete_weight,      // Variable.
                        PLUGIN_VAR_NOCMDARG,  // Not a command-line
    argument.
                        "Tells " PLUGIN_NAME " how important we think delete statements are",
                        NULL,                // Check function.
                        update_delete_weight,  // Update function.
                        5,                   // Default value.
                        0,                   // Min value.
                        INT_MAX,                   // Max value.
                        1                    // Block size.
                        );

static MYSQL_SYSVAR_DOUBLE(ratio_for_update,              // Name.
                        sys_var_ratio_for_update,      // Variable.
                        PLUGIN_VAR_NOCMDARG,  // Not a command-line argument.
                        "Tells " PLUGIN_NAME " what the ratio of updated rows in a table must be to force an update",
                        NULL,                // Check function.
                        update_ratio_for_updates,  // Update function.
                        0.05,                   // Default value.
                        0.00001,                   // Min value.
                        10000,                   // Max value.
                        1                    // Block size.
                        );

static MYSQL_SYSVAR_DOUBLE(outside_boundary_weight,              // Name.
                        sys_var_outside_boundary_weight,      // Variable.
                        PLUGIN_VAR_NOCMDARG,  // Not a command-line argument.
                        "Tells " PLUGIN_NAME " how important updates that are outside the boundary of a histogram are regarded",
                        NULL,                // Check function.
                        update_outside_boundary_weight,  // Update function.
                        5,                   // Default value.
                        0,                   // Min value.
                        INT_MAX,                   // Max value.
                        1                    // Block size.
                        );

static MYSQL_SYSVAR_INT(inverse_sensitivity_to_change,              // Name.
                        sys_var_inverse_sensitivity_to_change,      // Variable.
                        PLUGIN_VAR_NOCMDARG,  // Not a command-line argument.
                        "Tells " PLUGIN_NAME " how important updates that are outside the boundary of a histogram are regarded",
                        NULL,                // Check function.
                        update_inverse_sensitivity_to_change,  // Update function.
                        5000,                   // Default value.
                        0,                   // Min value.
                        INT_MAX,                   // Max value.
                        1                    // Block size.
                        );

```

```
SYS_VAR *histogram_rewriter_plugin_sys_vars[] = {MYSQL_SYSVAR(rule),
                                                 MYSQL_SYSVAR(
    statements_between_updates),
                                                 MYSQL_SYSVAR(
    insert_weight),
                                                 MYSQL_SYSVAR(
    delete_weight),
                                                 MYSQL_SYSVAR(
    ratio_for_update),
                                                 MYSQL_SYSVAR(
    outside_boundary_weight),
                                                 MYSQL_SYSVAR(
    inverse_sensitivity_to_change),NULL};


static int histogram_updater_notify(MYSQL_THD thd, mysql_event_class_t
    event_class,
                                const void *event);

/* instrument the memory allocation */
#ifdef HAVE_PSI_INTERFACE
static PSI_memory_key key_memory_lundgren;

static PSI_memory_info all_rewrite_memory[] = {
    {&key_memory_lundgren, "histogram_updater", 0, 0, PSI_DOCUMENT_ME}};

static int plugin_init(MYSQL_PLUGIN) {
  const char *category = "sql";
  int count;
  table_size  = -1;
    current_bounds.lower_bound = -1;
    current_bounds.upper_bound = -1;
  count = static_cast<int>(array_elements(all_rewrite_memory));
  rule_2_counter = 0;       //Init counters to 0
  rule_3_counter = 0;

  sys_var_update_rule = 0;      // Intialise with the rule set to don't
    update histograms.

  /*
  MYSQL_THD thd;
  mysql_memory_register(category, all_rewrite_memory, count);
  std::string create_table = "CREATE TABLE IF NOT EXISTS tester (\n"
                               "    test_id INT AUTO_INCREMENT,\n"
                               "    text VARCHAR(255) NOT NULL,\n";
  char *init_query;
  strncpy(init_query, create_table.c_str(), sizeof(create_table));
  MYSQL_LEX_STRING new_query = {init_query, sizeof(init_query)};
  mysql_parser_parse(thd, new_query, false, NULL, NULL);

   */
  return 0; /* success */
}
#else
#define plugin_init NULL
#define key_memory_lundgren PSI_NOT_INSTRUMENTED
#endif /* HAVE_PSI_INTERFACE */
```

```



/* Audit plugin descriptor */
static struct st_mysql_audit lundgren_descriptor = {
        MYSQL_AUDIT_INTERFACE_VERSION, /* interface version */
        NULL,                          /* release_thd()     */
        histogram_updater_notify,      /* event_notify()    */
        {
                0,
                0,
                (unsigned long)MYSQL_AUDIT_PARSE_ALL,
        } /* class mask        */
};

/* Plugin descriptor */
mysql_declare_plugin(audit_log){
        MYSQL_AUDIT_PLUGIN,    /* plugin type                   */
        &lundgren_descriptor, /* type specific descriptor      */
        "histogram_updater",           /* plugin name                   */
        "Sevre Vestrheim", /* author */
        "Histogram updater plugin", /* description                   */
        PLUGIN_LICENSE_GPL,         /* license                       */
        plugin_init,                /* plugin initializer            */
        NULL,                       /* plugin check uninstall        */
        NULL,                       /* plugin deinitializer          */
        0x0001,                     /* version                       */
        NULL,                       /* status variables              */
        histogram_rewriter_plugin_sys_vars, /* system variables
       */
        NULL,                       /* reserverd                     */
        0                           /* flags                         */
} mysql_declare_plugin_end;


void connect_and_run(const char* table)
{
    Internal_query_session *session = new Internal_query_session();
    int fail =session->execute_resultless_query("USE test");
    char query[200];
    strcpy(query,"Analyze table ");
    strcat(query,table);
    strcat(query," Update histogram on msm_value");
    // std::string builder = "Analyze table ";
    // builder += table;
    // builder += " Update histograms on text";
    // const char* query = builder.c_str();
    Sql_resultset *resultset = session->execute_query(query);   //Analyze
    table measurement Update histograms on msm_value
    delete session;

}

/**
  Entry point to the plugin. The server calls this function after each
    parsed
  query when the plugin is active.
```

```
*/

static int histogram_updater_notify(MYSQL_THD thd, mysql_event_class_t
    event_class,
                                    const void *event) {
  if (event_class == MYSQL_AUDIT_PARSE_CLASS) {
    const struct mysql_event_parse *event_parse =
        static_cast<const struct mysql_event_parse *>(event);
    if (event_parse->event_subclass != MYSQL_AUDIT_PARSE_POSTPARSE ||
    sys_var_update_rule == 0) {
        return 0;       //if we don't match our event class or the update
    rule is 0 then don't do anything.
    }

    if (!update_histograms(thd, event_parse->query.str)) {
        return 0;
    }
    // connect_and_run(table_name); we used to do this, but we only want
    to update historams on one table, so let's simplify this for ourselves
    connect_and_run("measurement");

  }

  return 0;
}
```

histogram_updater.h

```
/* Header file with the needed variables for internal counters and system
    variables  */

#ifndef MYSQL_HISTOGRAM_UPDATER_H
#define MYSQL_HISTOGRAM_UPDATER_H

//Struct
struct  histogram_bounds{
    double lower_bound;
    double upper_bound;
};

//System variables
int sys_var_update_rule;    //Rule variable
int sys_var_statements_between_updates;  //rule 2,3 between updates
double sys_var_insert_weight;
double sys_var_delete_weight;
double sys_var_ratio_for_update;  //Ratio for updates to use for rule six
    and nine
double sys_var_outside_boundary_weight;
int sys_var_inverse_sensitivity_to_change;

//Rule 2 variables
int rule_2_counter;

//Rule 3 variables
double rule_3_counter;


```

```
//Rule 6 variables
int rule_6_counter;
double rule_6_ratio;

//static int rule_6_base_no_between_updates = 200000;
int table_size;


//Rule 7 variables
double rule_7_counter;
double insert_value;
histogram_bounds current_bounds;


//Rule 9 variables
double no_of_statements = 0;
double rule_9_counter;
double rule_9_ratio;
#endif //MYSQL_HISTOGRAM_UPDATER_H
```

query_acceptance.h

```
/* File containing the updating rules themselves and supporting functions
    to query database for information about
 * histograms and table size  */

#include <string.h>
#include <mysql/service_parser.h>
#include "plugin/histogram_updater/constants.h"
#include "plugin/histogram_updater/histogram_updater.h"
#include "plugin/histogram_updater/internal_query/internal_query_session.h
    "
#include "plugin/histogram_updater/internal_query/sql_resultset.h"
#include "plugin/histogram_updater/internal_query/sql_service_context.h"
#include "my_inttypes.h"
#include <regex>
#include <tuple>


#ifndef LUNDGREN_QUERY_ACCEPTANCE
#define LUNDGREN_QUERY_ACCEPTANCE

int fetch_measurement_table_size()
{
    Internal_query_session *session = new Internal_query_session();
    int fail =session->execute_resultless_query("USE test");
    char query[200];
    strcpy(query,"select concat('',count(*)) from measurement;");
    Sql_resultset *resultset = session->execute_query(query);   //Analyze
    table measurement Update histograms on msm_value
    if (resultset->get_rows()>0){
        resultset->first();
        int number_of_rows = atoi(resultset->getString(0));
        delete session;
        return number_of_rows;
    }
    else {
        delete session;
        return -1;
    }
}


histogram_bounds fetch_histogram_boundaries() {
    histogram_bounds new_histogram_bounds;
    Internal_query_session *session = new Internal_query_session();
    int fail = session->execute_resultless_query("USE test");
    char query[2000];
    strcpy(query, "SELECT TABLE_NAME,COLUMN_NAME,CAST(JSON_EXTRACT(HISTOGRAM,'$.buckets[0][0]')AS CHAR) AS LOWER_BOUND, CAST(JSON_EXTRACT(HISTOGRAM,CONCAT(\"$.buckets[\",JSON_LENGTH(`HISTOGRAM` ->> '$.buckets')-1,\"][1]\"))AS CHAR) AS UPPER_BOUND FROM INFORMATION_SCHEMA.COLUMN_STATISTICS WHERE SCHEMA_NAME = \"test\";");
    Sql_resultset *resultset = session->execute_query(query);   //Analyze table measurement Update histograms on msm_value
    if (resultset->get_rows()>0) {
        resultset->first();
        new_histogram_bounds.lower_bound = atof(resultset->getString(2));
        new_histogram_bounds.upper_bound = atof(resultset->getString(3));
    }
    else{
        new_histogram_bounds.lower_bound = 0;
        new_histogram_bounds.upper_bound = 0;
    }
    delete session;
    return new_histogram_bounds;
}


static bool update_histograms(MYSQL_THD thd, const char *query) {

    int type = mysql_parser_get_statement_type(thd);

    //RULE 1
    if (sys_var_update_rule == 1 && type == STATEMENT_TYPE_INSERT) { //Rule 1 means update for every insert.
        return true;
    }

    //RULE 2
    else if (sys_var_update_rule == 2 && (type == STATEMENT_TYPE_INSERT || type == STATEMENT_TYPE_DELETE || type == STATEMENT_TYPE_UPDATE)){ //Rule 2 has a set number of runs between each update, defined in histogram_updater.h
        rule_2_counter++;
        if (std::fmod(rule_2_counter,sys_var_statements_between_updates) < 1){
            return true;
        }
        else {
            return false;
        }
    }

    //RULE 3
    else if (sys_var_update_rule == 3 && (type == STATEMENT_TYPE_INSERT ||
     type == STATEMENT_TYPE_DELETE || type == STATEMENT_TYPE_UPDATE)){
        if (type == STATEMENT_TYPE_INSERT){
            rule_3_counter += 1*sys_var_insert_weight;
        }
        if (type == STATEMENT_TYPE_DELETE){
            rule_3_counter += 1*sys_var_delete_weight;
        }
        if (type == STATEMENT_TYPE_UPDATE){
            rule_3_counter += 1*1;
        }
        if (std::fmod(rule_3_counter,sys_var_statements_between_updates) <
     std::min(sys_var_delete_weight,sys_var_insert_weight) ){
            //printf("HEIHEI %d\n", rule_3_counter);
            rule_3_counter = 0;
            return true;
        }
        else {
            return false;
        }
    }

    //RULE 6
    else if (sys_var_update_rule == 6 && (type == STATEMENT_TYPE_INSERT ||
     type == STATEMENT_TYPE_DELETE || type == STATEMENT_TYPE_UPDATE)){
        if (table_size == -1 || table_size == 0){
            table_size = fetch_measurement_table_size();
        }
        rule_6_counter++;
        rule_6_ratio = (table_size!=0&&table_size!=-1)? (double)
    rule_6_counter/double(table_size) : 0;
        if (rule_6_ratio > sys_var_ratio_for_update){
            rule_6_counter = 0;
            table_size  =-1;
            return true;
        }
        else{
            return false;
        }
    }

    //RULE 7
    else if (sys_var_update_rule == 7 && (type == STATEMENT_TYPE_INSERT ||
     type == STATEMENT_TYPE_DELETE || type == STATEMENT_TYPE_UPDATE)){

        if (type == STATEMENT_TYPE_INSERT){//Get the inserted value
            std::string result;
            std::regex re("(\\d{2}\\.\\d{3,4})");
            std::cmatch match;
            if (std::regex_search(query, match, re) && match.size() > 1) {
                result = match.str(1);
            } else {
                result = std::string("");
                insert_value = -1;
            }
            if (result.length()>0){
                insert_value = atof(result.c_str());
            }
        }
        if (current_bounds.lower_bound==-1 && current_bounds.upper_bound == -1){ //Intialize/ fetch new data if required
            current_bounds = fetch_histogram_boundaries();
        }
        if (type == STATEMENT_TYPE_INSERT && (current_bounds.upper_bound < insert_value || insert_value < current_bounds.lower_bound)){ // statement outside range
            rule_7_counter += 1*sys_var_outside_boundary_weight;
        }
        else  {//Statment inside range or delete statement
            rule_7_counter++;
        }
        if ((sys_var_statements_between_updates-rule_7_counter) < sys_var_outside_boundary_weight){//Do we need to udpate?
            rule_7_counter = 0;
            current_bounds.lower_bound = -1;
            current_bounds.upper_bound = -1;
            return true;
        }
        else{
            return false;
        }
    }

    //RULE 9
    else if (sys_var_update_rule == 9 && (type == STATEMENT_TYPE_INSERT || type == STATEMENT_TYPE_DELETE || type == STATEMENT_TYPE_UPDATE)){
        no_of_statements += 1;
        if (table_size == -1 || table_size == 0){
            table_size = fetch_measurement_table_size();
        }
        if (type == STATEMENT_TYPE_INSERT){//Get the inserted value
            std::string result;
            std::regex re("(\\d{2}\\.\\d{3,4})");
            std::cmatch match;
            if (std::regex_search(query, match, re) && match.size() > 1) {
                result = match.str(1);
            } else {
                result = std::string("");
                insert_value = -1;
            }
            if (result.length()>0){
                insert_value = atof(result.c_str());
            }
        }
        if (current_bounds.lower_bound==-1 && current_bounds.upper_bound == -1){ //Intialize/ fetch new data if required
            current_bounds = fetch_histogram_boundaries();
        }
        if (type == STATEMENT_TYPE_INSERT && (current_bounds.upper_bound < insert_value || insert_value < current_bounds.lower_bound)){ // statement outside range
            rule_9_counter += 1*sys_var_outside_boundary_weight;
        } else if (type == STATEMENT_TYPE_INSERT){
            rule_9_counter += 1*sys_var_insert_weight;
        } else if (type == STATEMENT_TYPE_DELETE){
            rule_9_counter += 1*sys_var_delete_weight;
        } else {
            rule_9_counter ++;
        }

        rule_9_ratio = (table_size!=0&&table_size!=-1)? double(
    no_of_statements)/double(table_size) : 1;

        if (rule_9_counter*rule_9_ratio >
    sys_var_inverse_sensitivity_to_change){
            rule_9_counter = 0;
            no_of_statements = 0;
            table_size = -1;
            current_bounds.lower_bound = -1;
            current_bounds.upper_bound = -1;
            return true;
        }
        else {
            return false;
        }
    }


    else {      //Rule is not handled, don't update
        return false;
        }

}


#endif
```

execute_timing.test

```
#The master file for executing our tests. This file configures the
    paramters and calls the handlers for different rules to execute and
    test, it also reports the testing progress using the append_file
    function to write to a specified file.

--echo #
--echo #Master test for timing:
--echo #

--let $param_run_no = 1
--let $no_of_measurements = 30
--let $histogram_plugin_initialized = 0

--disable_query_log
--disable_result_log

#Run a full set of tests against all rules.

#initialise all weight params first to default values
--let $weight_param_1 = 10000
--let $weight_param_2 = 0.05
```

```
--let $weight_param_3 = 5
--let $weight_param_4 = 5000
--let $weight_param_5 = 5
--let $weight_param_6 = 5


--write_file suite/histogram_plugin/$reporting_while_running_location
Starting test
Starting single param rules
EOF

#Test the single param rules first !!!
--let $weight_param_1 = 10000
--let $weight_param_2 = 0.05
--source suite/histogram_plugin/include/Single_param_result_set.include
append_file suite/histogram_plugin/$reporting_while_running_location;
First set of single params - Complete
EOF

#Test the single param rules first !!!
--let $weight_param_1 = 20000
--let $weight_param_2 = 0.1
--source suite/histogram_plugin/include/Single_param_result_set.include
append_file suite/histogram_plugin/$reporting_while_running_location;
Second set of single params - Complete
EOF

--let $weight_param_1 = 50000
--let $weight_param_2 = 0.25
--source suite/histogram_plugin/include/Single_param_result_set.include
append_file suite/histogram_plugin/$reporting_while_running_location;
Third set of single params - Complete
EOF

--let $weight_param_1 = 70000
--let $weight_param_2 = 0.5
--source suite/histogram_plugin/include/Single_param_result_set.include
append_file suite/histogram_plugin/$reporting_while_running_location;
Fourth set of single params - Complete
EOF

--let $weight_param_1 = 100000
--let $weight_param_2 = 0.75
--source suite/histogram_plugin/include/Single_param_result_set.include
append_file suite/histogram_plugin/$reporting_while_running_location;
Fifth set of single params - Complete
EOF

--let $weight_param_1 = 200000
--let $weight_param_2 = 0.9
--source suite/histogram_plugin/include/Single_param_result_set.include
append_file suite/histogram_plugin/$reporting_while_running_location;
Sixth set of single params - Complete
EOF

--let $weight_param_1 = 400000
--let $weight_param_2 = 0.95
```

```
--source suite/histogram_plugin/include/Single_param_result_set.include
append_file suite/histogram_plugin/$reporting_while_running_location;
Seventh set of single params - Complete
EOF

--let $weight_param_1 = 500000
--let $weight_param_2 = 1.00
--source suite/histogram_plugin/include/Single_param_result_set.include
append_file suite/histogram_plugin/$reporting_while_running_location;
Eight set of single params - Complete
EOF

--let $weight_param_1 = 700000
--let $weight_param_2 = 1.05
--source suite/histogram_plugin/include/Single_param_result_set.include
append_file suite/histogram_plugin/$reporting_while_running_location;
Ninth set of single params - Complete
EOF

--let $weight_param_1 = 1000000
--let $weight_param_2 = 1.50
--source suite/histogram_plugin/include/Single_param_result_set.include
append_file suite/histogram_plugin/$reporting_while_running_location;
Tenth set of single params - Complete
EOF

--let $weight_param_1 = 5000000
--let $weight_param_2 = 3
--source suite/histogram_plugin/include/Single_param_result_set.include
append_file suite/histogram_plugin/$reporting_while_running_location;
Tenth set of single params - Complete
EOF

--let $weight_param_1 = 10000000
--let $weight_param_2 = 10
--source suite/histogram_plugin/include/Single_param_result_set.include
append_file suite/histogram_plugin/$reporting_while_running_location;
Elleventh set of single params - Complete
EOF

--let $weight_param_1 = 20000000
--let $weight_param_2 = 50
--source suite/histogram_plugin/include/Single_param_result_set.include
append_file suite/histogram_plugin/$reporting_while_running_location;
Twelveth set of single params - Complete
EOF

--let $weight_param_1 = 40000000
--let $weight_param_2 = 100
--source suite/histogram_plugin/include/Single_param_result_set.include
append_file suite/histogram_plugin/$reporting_while_running_location;
Thirteenth set of single params - Complete
EOF

append_file suite/histogram_plugin/$reporting_while_running_location;
Single param rules completed

```

```
Starting multiparam with rule 7
EOF

###WE DO NOT TEST RULE THREE ANYMORE SINCE OUR USE-CASE DOES NOT PROPERLY
    SUPPORT IT
##Now we move on to testing the multi param rules
#Reset param run counter
#--let $param_run_no = 1
##Starting with rule 3
#--let $weight_param_1 = 400000
#--let $weight_param_5 = 10
#--let $weight_param_6 = 1
#--source suite/histogram_plugin/include/testing_rule_3.include

#--let $weight_param_1 = 400000
#--let $weight_param_5 = 10
#--let $weight_param_6 = 2
#--source suite/histogram_plugin/include/testing_rule_3.include

#--let $weight_param_1 = 400000
#--let $weight_param_5 = 10
#--let $weight_param_6 = 4
#--source suite/histogram_plugin/include/testing_rule_3.include

#--let $weight_param_1 = 400000
#--let $weight_param_5 = 10
#--let $weight_param_6 = 8
#--source suite/histogram_plugin/include/testing_rule_3.include

#--let $weight_param_1 = 400000
#--let $weight_param_5 = 1
#--let $weight_param_6 = 10
#--source suite/histogram_plugin/include/testing_rule_3.include

#--let $weight_param_1 = 400000
#--let $weight_param_5 = 2
#--let $weight_param_6 = 10
#--source suite/histogram_plugin/include/testing_rule_3.include

#--let $weight_param_1 = 400000
#--let $weight_param_5 = 4
#--let $weight_param_6 = 10
#--source suite/histogram_plugin/include/testing_rule_3.include

#--let $weight_param_1 = 400000
#--let $weight_param_5 = 8
#--let $weight_param_6 = 10
#--source suite/histogram_plugin/include/testing_rule_3.include

#--let $weight_param_1 = 400000
#--let $weight_param_5 = 5
#--let $weight_param_6 = 10
#--source suite/histogram_plugin/include/testing_rule_3.include

#--let $weight_param_1 = 400000
#--let $weight_param_5 = 10
#--let $weight_param_6 = 5
```

```
#--source suite/histogram_plugin/include/testing_rule_3.include

#--let $weight_param_1 = 400000
#--let $weight_param_5 = 5
#--let $weight_param_6 = 5
#--source suite/histogram_plugin/include/testing_rule_3.include
#append_file suite/histogram_plugin/$reporting_while_running_location;
#Rule 3 - Completed
#Starting multiparam with rule 7
#EOF


#Moving on to rule 7
#Reset param run counter
--let $param_run_no = 1
#Then set first run of params
--let $weight_param_1 = 400000
--let $weight_param_3 = 1
--source suite/histogram_plugin/include/testing_rule_7.include


--let $weight_param_3 = 10
--source suite/histogram_plugin/include/testing_rule_7.include


--let $weight_param_3 = 20
--source suite/histogram_plugin/include/testing_rule_7.include


--let $weight_param_3 = 100
--source suite/histogram_plugin/include/testing_rule_7.include


--let $weight_param_3 = 200
--source suite/histogram_plugin/include/testing_rule_7.include
append_file suite/histogram_plugin/$reporting_while_running_location;
Rule 7 - Completed

Starting multiparam with rule 9
EOF



#Moving on to rule 9
#Reset param run counter
--let $param_run_no = 1
#Then set first run of params
--let $weight_param_3 = 1
--let $weight_param_4 = 10000
--let $weight_param_5 = 1
--let $weight_param_6 = 1
--source suite/histogram_plugin/include/testing_rule_9.include

--let $weight_param_4 = 20000
--source suite/histogram_plugin/include/testing_rule_9.include

--let $weight_param_4 = 100000
```

```
--source suite/histogram_plugin/include/testing_rule_9.include

--let $weight_param_4 = 200000
--source suite/histogram_plugin/include/testing_rule_9.include

--let $weight_param_4 = 1000000
--source suite/histogram_plugin/include/testing_rule_9.include

--let $weight_param_4 = 5000000
--source suite/histogram_plugin/include/testing_rule_9.include

--let $weight_param_4 = 10000000
--source suite/histogram_plugin/include/testing_rule_9.include

--let $weight_param_4 = 100000000
--source suite/histogram_plugin/include/testing_rule_9.include
append_file suite/histogram_plugin/$reporting_while_running_location;
Rule 9 - Completed

Test Complete - All rules complete
EOF

#Cleanup
--source suite/histogram_plugin/include/restart_server.include
```

queries.test

```
#MTR file which handles running all queries for one partition and
    gathering the timing and histogram results

--echo #
--echo #running queries:
--echo #

--let $i = $no_of_runs
--disable_query_log
--disable_result_log

use test;

while ($i)
{

# select * from test.measurement join test.server on test.measurement.
    msm_serv_id = test.server.serv_id where test.measurement.msm_value
    >80;
# select * from test.measurement join test.server on test.measurement.
    msm_serv_id = test.server.serv_id where test.measurement.msm_value
    <80;
# select * from test.measurement join test.server on test.measurement.
    msm_serv_id = test.server.serv_id join test.rack on test.rack.rack_id
    = test.server.serv_rack_id where test.measurement.msm_value >80;
# select * from test.measurement join test.server on test.measurement.
    msm_serv_id = test.server.serv_id join test.rack on test.rack.rack_id
    = test.server.serv_rack_id where test.measurement.msm_value <80;
# select * from test.measurement join test.server on test.measurement.
    msm_serv_id = test.server.serv_id join test.rack on test.rack.rack_id
    = test.server.serv_rack_id where test.measurement.msm_value between 80
     and 100;
select * from test.measurement join test.server on test.measurement.
    msm_serv_id = test.server.serv_id join test.rack on test.rack.rack_id
    = test.server.serv_rack_id join test.center on test.center.cent_id =
    test.rack.rack_cent_id where test.measurement.msm_value >80;
select * from test.measurement join test.server on test.measurement.
    msm_serv_id = test.server.serv_id join test.rack on test.rack.rack_id
    = test.server.serv_rack_id join test.center on test.center.cent_id =
    test.rack.rack_cent_id where test.measurement.msm_value <80;
select * from test.measurement join test.server on test.measurement.
    msm_serv_id = test.server.serv_id join test.rack on test.rack.rack_id
    = test.server.serv_rack_id join test.center on test.center.cent_id =
    test.rack.rack_cent_id where test.measurement.msm_value >90;
select * from test.measurement join test.server on test.measurement.
    msm_serv_id = test.server.serv_id join test.rack on test.rack.rack_id
    = test.server.serv_rack_id join test.center on test.center.cent_id =
    test.rack.rack_cent_id where test.measurement.msm_value <90;
select * from test.measurement join test.server on test.measurement.
    msm_serv_id = test.server.serv_id join test.rack on test.rack.rack_id
    = test.server.serv_rack_id join test.center on test.center.cent_id =
    test.rack.rack_cent_id where test.measurement.msm_value between 80 and
     100;
select * from test.measurement join test.server on test.measurement.
    msm_serv_id = test.server.serv_id join test.rack on test.rack.rack_id
    = test.server.serv_rack_id join test.center on test.center.cent_id =
    test.rack.rack_cent_id where test.measurement.msm_value between 90 and
     100;
select * from test.measurement join test.server on test.measurement.
    msm_serv_id = test.server.serv_id join test.rack on test.rack.rack_id
    = test.server.serv_rack_id join test.center on test.center.cent_id =
    test.rack.rack_cent_id join test.city on test.city.city_id = test.
    center.cent_city_id where test.measurement.msm_value >80;
select * from test.measurement join test.server on test.measurement.
    msm_serv_id = test.server.serv_id join test.rack on test.rack.rack_id
    = test.server.serv_rack_id join test.center on test.center.cent_id =
    test.rack.rack_cent_id join test.city on test.city.city_id = test.
    center.cent_city_id where test.measurement.msm_value <80;
select * from test.measurement join test.server on test.measurement.
    msm_serv_id = test.server.serv_id join test.rack on test.rack.rack_id
    = test.server.serv_rack_id join test.center on test.center.cent_id =
    test.rack.rack_cent_id join test.city on test.city.city_id = test.
    center.cent_city_id where test.measurement.msm_value <90;
select * from test.measurement join test.server on test.measurement.
    msm_serv_id = test.server.serv_id join test.rack on test.rack.rack_id
    = test.server.serv_rack_id join test.center on test.center.cent_id =
    test.rack.rack_cent_id join test.city on test.city.city_id = test.
    center.cent_city_id where test.measurement.msm_value >90;
select * from test.measurement join test.server on test.measurement.
    msm_serv_id = test.server.serv_id join test.rack on test.rack.rack_id
    = test.server.serv_rack_id join test.center on test.center.cent_id =
    test.rack.rack_cent_id join test.city on test.city.city_id = test.
    center.cent_city_id where test.measurement.msm_value between 80 and
    100;
select * from test.measurement join test.server on test.measurement.
    msm_serv_id = test.server.serv_id join test.rack on test.rack.rack_id
    = test.server.serv_rack_id join test.center on test.center.cent_id =
    test.rack.rack_cent_id join test.city on test.city.city_id = test.
    center.cent_city_id where test.measurement.msm_value between 90 and
    100;
dec $i;
}


--enable_result_log
#COLLECTING QUERY RUN TIMES
--eval select sql_text Query,truncate(((timer_end-timer_start)
    /1000000000000),6) Duration,rows_sent "Returned rows",concat($param_1,
    '') "Test type",concat('$weight_params','') "Supplied parameters",
    concat('$param_run_no','') "Parameter run no", (select max(msm_id)
    from measurement) "Number of Inserts" from performance_schema.
    events_statements_history_long where sql_text like 'select * from test
    .%' and event_id not in (
                                select ev.event_id from(select base.
    sql_text,max(base.duration) duration from (select event_id,sql_text,
    timer_end-timer_start duration from performance_schema.
    events_statements_history_long where sql_text like 'select * from test
    .%')as base group by base.sql_text) as max_values join (select
    event_id,sql_text,timer_end-timer_start duration from
    performance_schema.events_statements_history_long where sql_text like
    'select * from test.%')as ev on ev.sql_text=max_values.sql_text and ev
    .duration=max_values.duration
                                ) order by sql_text;

#COLLECTING HISTOGRAM UPDATING TIMES
--eval select sql_text "Analyse statement",truncate(((sum(timer_end-
    timer_start))/1000000000000),6) Duration,count(*) No_of_executes,
    concat($param_1,'') "Test type", concat('$weight_params','') "Supplied
     parameters", concat('$param_run_no','') "Parameter run no", (select
    max(msm_id) from measurement) "Number of Inserts" from
    performance_schema.events_statements_history_long where lower(sql_text
    ) like 'analyze table measurement update histogram on msm_value%'
    group by sql_text;


#SHOWING THE CURRENT PLANS FOR QUERIES, SOME PLANS ARE DISABLED
--enable_query_log
#--eval explain format = tree select * /*$param_1 $param_2*/ from test.
    measurement join test.server on test.measurement.msm_serv_id = test.
    server.serv_id where test.measurement.msm_value >80;
#--eval explain format = tree select * /*$param_1 $param_2*/ from test.
    measurement join test.server on test.measurement.msm_serv_id = test.
    server.serv_id where test.measurement.msm_value <80;
#--eval explain format = tree select * /*$param_1 $param_2*/ from test.
    measurement join test.server on test.measurement.msm_serv_id = test.
    server.serv_id join test.rack on test.rack.rack_id = test.server.
    serv_rack_id where test.measurement.msm_value >80;
#--eval explain format = tree select * /*$param_1 $param_2*/ from test.
    measurement join test.server on test.measurement.msm_serv_id = test.
    server.serv_id join test.rack on test.rack.rack_id = test.server.
    serv_rack_id where test.measurement.msm_value <80;
#--eval explain format = tree select * /*$param_1 $param_2*/ from test.
    measurement join test.server on test.measurement.msm_serv_id = test.
    server.serv_id join test.rack on test.rack.rack_id = test.server.
    serv_rack_id where test.measurement.msm_value between 80 and 100;
#--eval explain format = tree select * /*$param_1 $param_2*/ from test.
    measurement join test.server on test.measurement.msm_serv_id = test.
    server.serv_id join test.rack on test.rack.rack_id = test.server.
    serv_rack_id join test.center on test.center.cent_id = test.rack.
    rack_cent_id where test.measurement.msm_value >80;
#--eval explain format = tree select * /*$param_1 $param_2*/ from test.
    measurement join test.server on test.measurement.msm_serv_id = test.
    server.serv_id join test.rack on test.rack.rack_id = test.server.
    serv_rack_id join test.center on test.center.cent_id = test.rack.
    rack_cent_id where test.measurement.msm_value <80;
#--eval explain format = tree select * /*$param_1 $param_2*/ from test.
    measurement join test.server on test.measurement.msm_serv_id = test.
    server.serv_id join test.rack on test.rack.rack_id = test.server.
    serv_rack_id join test.center on test.center.cent_id = test.rack.
    rack_cent_id where test.measurement.msm_value between 80 and 100;
--eval explain select * /*$param_1 $param_2*/ from test.measurement join
    test.server on test.measurement.msm_serv_id = test.server.serv_id join
     test.rack on test.rack.rack_id = test.server.serv_rack_id join test.
    center on test.center.cent_id = test.rack.rack_cent_id join test.city
    on test.city.city_id = test.center.cent_city_id where test.measurement
    .msm_value >80;
#--eval explain select * /*$param_1 $param_2*/ from test.measurement join
    test.server on test.measurement.msm_serv_id = test.server.serv_id join
     test.rack on test.rack.rack_id = test.server.serv_rack_id join test.
    center on test.center.cent_id = test.rack.rack_cent_id join test.city
    on test.city.city_id = test.center.cent_city_id where test.measurement
    .msm_value >80;
--eval explain select * /*$param_1 $param_2*/ from test.measurement join
    test.server on test.measurement.msm_serv_id = test.server.serv_id join
     test.rack on test.rack.rack_id = test.server.serv_rack_id join test.
    center on test.center.cent_id = test.rack.rack_cent_id join test.city
    on test.city.city_id = test.center.cent_city_id where test.measurement
    .msm_value <80;
#--eval explain select * /*$param_1 $param_2*/ from test.measurement join
    test.server on test.measurement.msm_serv_id = test.server.serv_id join
     test.rack on test.rack.rack_id = test.server.serv_rack_id join test.
    center on test.center.cent_id = test.rack.rack_cent_id join test.city
    on test.city.city_id = test.center.cent_city_id where test.measurement
    .msm_value <80;
--eval explain select * /*$param_1 $param_2*/ from test.measurement join
    test.server on test.measurement.msm_serv_id = test.server.serv_id join
     test.rack on test.rack.rack_id = test.server.serv_rack_id join test.
    center on test.center.cent_id = test.rack.rack_cent_id join test.city
    on test.city.city_id = test.center.cent_city_id where test.measurement
    .msm_value <90;
#--eval explain select * /*$param_1 $param_2*/ from test.measurement join
    test.server on test.measurement.msm_serv_id = test.server.serv_id join
     test.rack on test.rack.rack_id = test.server.serv_rack_id join test.
    center on test.center.cent_id = test.rack.rack_cent_id join test.city
    on test.city.city_id = test.center.cent_city_id where test.measurement
    .msm_value <90;
--eval explain select * /*$param_1 $param_2*/ from test.measurement join
    test.server on test.measurement.msm_serv_id = test.server.serv_id join
     test.rack on test.rack.rack_id = test.server.serv_rack_id join test.
    center on test.center.cent_id = test.rack.rack_cent_id join test.city
    on test.city.city_id = test.center.cent_city_id where test.measurement
    .msm_value >90;
#--eval explain select * /*$param_1 $param_2*/ from test.measurement join
    test.server on test.measurement.msm_serv_id = test.server.serv_id join
     test.rack on test.rack.rack_id = test.server.serv_rack_id join test.
    center on test.center.cent_id = test.rack.rack_cent_id join test.city
    on test.city.city_id = test.center.cent_city_id where test.measurement
    .msm_value >90;
--eval explain select * /*$param_1 $param_2*/ from test.measurement join
    test.server on test.measurement.msm_serv_id = test.server.serv_id join
     test.rack on test.rack.rack_id = test.server.serv_rack_id join test.
    center on test.center.cent_id = test.rack.rack_cent_id join test.city
    on test.city.city_id = test.center.cent_city_id where test.measurement
    .msm_value between 80 and 100;
#--eval explain select * /*$param_1 $param_2*/ from test.measurement join
    test.server on test.measurement.msm_serv_id = test.server.serv_id join
     test.rack on test.rack.rack_id = test.server.serv_rack_id join test.
    center on test.center.cent_id = test.rack.rack_cent_id join test.city
    on test.city.city_id = test.center.cent_city_id where test.measurement
    .msm_value between 80 and 100;
--eval explain select * /*$param_1 $param_2*/ from test.measurement join
    test.server on test.measurement.msm_serv_id = test.server.serv_id join
     test.rack on test.rack.rack_id = test.server.serv_rack_id join test.
    center on test.center.cent_id = test.rack.rack_cent_id join test.city
    on test.city.city_id = test.center.cent_city_id where test.measurement
    .msm_value between 90 and 100;

--disable_query_log
--disable_result_log

--echo #Truncating the performance schema so that it's ready for the next
    run.
truncate table performance_schema.events_statements_history_long;
```

Dataset_notebook.py

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:


#The file which creates the datasets and statements used for our tests.
get_ipython().run_line_magic('matplotlib', 'inline')
import numpy as np
import pandas as pa
import seaborn as sns
import matplotlib.pyplot as plt
import matplotlib.style as style
import csv
import glob, os
import re
from scipy.stats import uniform
import math
plt.style.use('seaborn')


# In[2]:
```

```python


no_of_measurements = 6000000
no_of_partitions = 30

temp = no_of_measurements//no_of_partitions*0.33

no_of_servers = int(math.ceil(temp/5000)*5000)
no_of_racks = no_of_servers
no_of_centers = no_of_racks//2
no_of_city = no_of_centers
no_of_components = 4

multi = 1

print(temp)
print (no_of_servers)
print (no_of_racks)
print(no_of_centers)
print(no_of_city)


#Data used for the preliminary dataset
#no_of_measurements = 200000
#no_of_partitions = 20
#no_of_racks = 10000
#no_of_racks = no_of_servers
#no_of_centers = no_of_racks//2
#no_of_city = no_of_centers
#no_of_components = 4


# In[4]:


#Functions to create the datasets
float_formatter = lambda x: "%.4f" % x

def move_distribution(total,no_of_partitions):
    i=0
    strides_up = np.geomspace(60,105,no_of_partitions/2)
    strides_down = np.flip(strides_up-1)
    strides= np.append(strides_up,strides_down)
    partitions = np.ones((no_of_partitions,total//no_of_partitions))
    for partition in partitions:
        partition*=np.random.normal(strides[i],3,len(partition))
        i +=1
    return partitions.flatten()#Output 1d array.

def create_measurement (multiplier):
    msm_id = np.arange(no_of_measurements*multiplier)+1 #Shifting to first
     id beeing 1
    msm_datetime = np.round(np.linspace(start=1546300800,stop=1546300800+(
    no_of_measurements*multiplier/100),num=no_of_measurements*multiplier)
    ,1)
    msm_value = np.round(move_distribution(no_of_measurements,
    no_of_partitions),4)#(np.random.normal(30,5,no_of_measurements*
    multiplier),4)
    msm_component_id = np.random.randint(1,no_of_components+1,
    no_of_measurements)
    msm_serv_id = np.random.randint(1,no_of_servers+1,no_of_measurements)

    df = pa.DataFrame({
   'msm_id':msm_id,
    'msm_datetime':msm_datetime,
    'msm_value':msm_value,
    'msm_component_id':msm_component_id,
    'msm_serv_id':msm_serv_id
    })
    df = df.astype({'msm_id':int,'msm_datetime':float,'msm_value':float,'
    msm_component_id':int,'msm_serv_id':int})
    return df


def create_server(multiplier):
    serv_id = np.arange(no_of_servers*multiplier)+1
    serv_rack_id = np.random.randint(1,no_of_racks+1,no_of_servers)
    df = pa.DataFrame({
        'serv_id':serv_id,
        'serv_rack_id':serv_rack_id
    })
    df = df.astype({'serv_id':int,'serv_rack_id':int})
    return df


def create_rack(multiplier):
    rack_id = np.arange(no_of_racks*multiplier)+1
    rack_cent_id = np.random.randint(1,no_of_centers+1,no_of_racks)
    df = pa.DataFrame({
        'rack_id':rack_id,
        'rack_cent_id':rack_cent_id
    })
    df = df.astype({'rack_id':int,'rack_cent_id':int})

    return df

def create_center(multiplier):
    cent_id = np.arange(no_of_centers*multiplier)+1
    cent_city_id = np.random.randint(1,no_of_city+1,no_of_centers)
    df = pa.DataFrame({
        'cent_id':cent_id,
        'cent_city_id':cent_city_id
    })
    df = df.astype({'cent_id':int,'cent_city_id':int})

    return df

def create_city(multiplier):
    city_id = np.arange(no_of_city*multiplier)+1
    city_name = np.random.randint(1,4,no_of_city*multiplier)
    df = pa.DataFrame({
        'city_id':city_id,
        'city_name':city_name
    })
    df = df.astype({'city_id':int,'city_name':int})
    return df

def create_component(multiplier):
    component_id = np.arange(no_of_components*multiplier)+1
    component_name = np.flip(component_id)
    df = pa.DataFrame({
        'component_id':component_id,
        'component_name':component_name
    })
    df = df.astype({'component_id':int,'component_name':int})
    return df

#Wrapping function for creation of the dataset
def create_dataset(multi=1):
    msm_df = create_measurement(multi)
    server_df = create_server(multi)
    rack_df = create_rack(multi)
    center_df = create_center(multi)
    city_df = create_city(multi)
    component_df = create_component(multi)
    return msm_df,server_df,rack_df,center_df,city_df,component_df




# In[5]:


##Creating the data used for testing

### Clean dataset directory first
dataset_paths = 'datasets/*'
files = glob.glob(dataset_paths)
for file in files:
    os.remove(file)

#Create dataset files
msm_df,server_df,rack_df,center_df,city_df,component_df = create_dataset
    (1)

# Creating the inserting sql for measurements dataset.
msm_df['insert_string']='insert into measurement (msm_datetime,msm_value,
    msm_component_id,msm_serv_id) values(from_unixtime('+msm_df['
    msm_datetime'].map(str)+'), '+msm_df['msm_value'].map(str)+','+msm_df[
    'msm_component_id'].map(str)+','+msm_df['msm_serv_id'].map(str)+');'
msm_df['insert_delete_string']= 'insert into measurement (msm_datetime,
    msm_value,msm_component_id,msm_serv_id) values(from_unixtime('+msm_df[
    'msm_datetime'].map(str)+'), '+msm_df['msm_value'].map(str)+','+msm_df
    ['msm_component_id'].map(str)+','+msm_df['msm_serv_id'].map(str)+');
    DELETE FROM measurement LIMIT 1;'

#Dividing the measurement datasets into several files.
partition_length = len(msm_df.index)//no_of_partitions
msm_df['insert_string'][:partition_length].to_csv('datasets/
    unformatted_measurement_0.sql',index=False,header=False)
for i in range(1,no_of_partitions+1):
    file_name = 'unformatted_measurement_{}.sql'.format(i)
    msm_df['insert_delete_string'][(i)*partition_length:partition_length*(
    i+1)].to_csv('datasets/'+file_name,index=False,header=False)


#Formatting the measurement.sql files to ready them for import.
unformatted_measurement_files = sorted(glob.glob(dataset_paths))
for file in unformatted_measurement_files:
    output_file_name=re.sub('^datasets/unformatted_','datasets/',file)
    with open(file, 'r') as f, open(output_file_name, 'w+') as fo:
#       #  fo.write("set autocommit = 0;")
        for line in f:
            fo.write(line.replace('"', '').replace("'", ""))
#        # fo.write("COMMIT;")
    os.remove(file)

#Adding in the rest of the dataset as csv files.
server_df.to_csv('datasets/server.csv',index=False)
rack_df.to_csv('datasets/rack.csv',index=False)
center_df.to_csv('datasets/center.csv',index=False)
city_df.to_csv('datasets/city.csv',index=False)
component_df.to_csv('datasets/component.csv',index=False)

#Creating a copy of the data in the required file location to load into
    mysql.
#We dont do this anymore, we use a symlink to the dataset location now !
    cp datasets/* /home/svestrhe/Documents/Forprosjekt/code/mysql-server/
    mysql-test/std_data/
#!scp -rp datasets svestrhe@loki12:/export/home/tmp/bygging/mysql-test/var
    /std_data


# In[ ]:


#Show graph of the distribution of values we have created.
fig = plt.figure(figsize=(9,6))
ax=sns.lineplot(x="msm_id", y="msm_value",data=msm_df)
xlabels = ['{:,.0f}'.format(x) + 'K' for x in ax.get_xticks()/1000]
ax.set_xticklabels(xlabels)
plt.title("")
ax.set_xlabel("Row number")
ax.set_ylabel("Measurement value")
#plt.axvline(x=0)
#plt.axvline(x=-2,ymax=0.25)
#plt.axvline(x=2,ymax=0.25)
msm_dist_plt = plt.gcf()
msm_dist_plt.savefig("/export/home/tmp/Dropbox/Apper/ShareLaTeX/Master/
    eval_plots/msm_distribution_plt.png",dpi=360,bbox_inches='tight')
plt.show()
```

Results.py

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:
```

```


#This file ingests, interprets and then presents the results generated by
    our tests.

get_ipython().run_line_magic('matplotlib', 'inline')
import numpy as np
import pandas as pa
import seaborn as sns
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
import glob, os
import matplotlib.style as style
#style.available
plt.style.use('seaborn')
#style.use('seaborn-poster') #sets the size of the charts
import pprint
from IPython.display import HTML

import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)
pa.options.display.max_rows = 50


# In[2]:


#Clean up result directory first
results_path = 'timing_results/*'
r = glob.glob(results_path)
for i in r:
    os.remove(i)

#Then clean the plan directory
plans_path = 'query_plans/*'
r = glob.glob(plans_path)
for i in r:
    os.remove(i)

#Then clean the timing_results directory
timing_histogram_path = 'hist_timing_results/*'
r = glob.glob(timing_histogram_path)
for i in r:
    os.remove(i)

#Setup variables
result_list = []
histogram_timing_results_list = []
query_plan_list = []
i = -1


# # Format the .result files from our mtr run so that we can import them
    using pandas csv importer

# In[3]:

```

```
#with open("/home/svestrhe/Documents/Forprosjekt/code/mysql-server/mysql-
    test/suite/histogram_plugin/r/execute_timing_loki12.result") as
    openfileobject:
with open("/home/svestrhe/Documents/Forprosjekt/code/mysql-server/mysql-
    test/suite/histogram_plugin/r/execute_timing_local.result") as
    openfileobject:
    for line in openfileobject:
        if not "#" in line[:5]:
            if "Query" in line:
                i+=1
                result_list.append([line])
                query_plan_list.append([])
            if "Analyse statement" in line[:17]:
                histogram_timing_results_list.append([line])
            if "select" in line[:8]:
                result_list[i].append(line)
            if "EXPLAIN" in line[:7] or "->" in line or "id" in line[:4]
    or "1" in line[:2]:
                query_plan_list[i].append(line)
            if "explain" in line[:7]:
                query_plan_list[i].append("\n")
                query_plan_list[i].append(line)
            if "analyze table measurement" in line.lower():
                histogram_timing_results_list[i].append(line)
    i = -1    #Cleanup


# In[4]:


#Export the timing results for queries
array_result = np.array(result_list)
result_list=[]      #Cleanup so that we can run the cells again and again
    without having to restart our kernel.
counter = 0
for result in array_result:
    string="timing_results/{}.csv".format(counter)
    f = open(string,"w+")
    for line in result:
        f.write(line)
    f.close()
    counter+=1


#Export the timing results for histograms
array_hist_result = np.array(histogram_timing_results_list)
histogram_timing_results_list = []
counter = 0
for hist_result in array_hist_result:
    string="hist_timing_results/{}.csv".format(counter)
    f = open(string,"w+")
    for line in hist_result:
        f.write(line)
    f.close()
    counter+=1

```

```

# # Now let's import the csv files into a dataframe.

# In[5]:


result_files = sorted(glob.glob(results_path),key=lambda x: int(x.split('.')[0][15:]))
frames = []
helper = []
counter = 0
for file in result_files:
    temp_df = pa.read_csv(str(file),'\t')
    helper.append([temp_df.iloc[0]['Test type'],counter])
    frames.append(temp_df)
    counter+=1

result_df = pa.concat(frames,sort=False)



hist_result_files = sorted(glob.glob(timing_histogram_path),key=lambda x: int(x.split('.')[0][20:]))
frames = []
counter = 0
for file in hist_result_files:
    temp_df = pa.read_csv(str(file),'\t')
 #   helper.append([temp_df.iloc[0]['Test type'],counter])
    frames.append(temp_df)
    counter+=1

hist_result_df = pa.concat(frames,sort=False)
hist_result_df=hist_result_df.astype({'No_of_executes': 'int32'})


#CREATE new SENS ds
sens_with_err = result_df.reset_index().drop(columns=['index'])
for index, row in sens_with_err.iterrows():
    sens_with_err.loc[index,'Internal run no'] = index%9
sens_with_err = sens_with_err.astype({'Internal run no': 'int'})

temp_sens = sens_with_err.groupby(['Test type','Parameter run no','Internal run no','Supplied parameters']).sum().reset_index().rename(columns={"Duration": "Query execution duration"})
temp_hist_sens = hist_result_df.groupby(['Test type','Parameter run no','Supplied parameters']).sum().reset_index().rename(columns={"Duration": "Histogram execution duration"})

another_temp_sens = pa.merge(temp_sens,temp_hist_sens,how='outer',on=['Test type','Parameter run no']).fillna(0)
another_temp_sens['Query and histogram execution duration'] = another_temp_sens['Query execution duration'] + another_temp_sens['Histogram execution duration']
another_temp_sens = another_temp_sens.drop(columns = ['Histogram execution duration','Number of Inserts','Returned rows','No_of_executes','Supplied parameters_y']).set_index(['Test type','Parameter run no','Internal run no','Supplied parameters_x'])
```

```

almost_done_df = pa.DataFrame(another_temp_sens,index = another_temp_sens.
    index).stack().to_frame().reset_index()
almost_done_df.columns = ['Test type','Parameter run no','Internal run no'
    ,'Supplied parameters','Duration type','Duration']
almost_done_df.head(500)

temp_3 = almost_done_df
temp_3 = temp_3.loc[temp_3['Test type'].isin(['No histogram ','Perfect
    histogram ','Stale histogram ']) & temp_3['Duration type'].isin(['
    Query execution duration'])]
temp_4 = pa.merge(almost_done_df,temp_3, indicator=True, how='outer').
    query('_merge=="left_only"').drop('_merge', axis=1)
condensed_sens_with_err = temp_4
condensed_sens_with_err.drop_duplicates(keep = 'first', inplace = True)


# In[6]:



#Export the query_plans
array_query_plan = np.array(query_plan_list)
query_plan_list=[]
counter = 0
for plan_set in array_query_plan:
    string="query_plans/plan_{}_{}.txt".format(counter,helper[counter][0].
    strip())
    f = open(string,"w+")
    for plan in plan_set:
        f.write(plan)
    f.close()
    counter+=1


# In[7]:


#Create the sensitivity DF
sensitivity_df = result_df.groupby(['Test type','Supplied parameters','
    Parameter run no']).sum().reset_index()
sensitivity_df = sensitivity_df[['Test type','Supplied parameters','
    Parameter run no','Duration']]
sensitivity_df = sensitivity_df.rename(columns={"Duration": "Query
    execution duration"})
sensitivity_df_hist_data = hist_result_df.groupby(['Test type','Supplied
    parameters','Parameter run no']).sum().reset_index().rename(columns={"
    Duration": "Histogram execution duration"})
temp = pa.merge(sensitivity_df,sensitivity_df_hist_data,how='outer',on=['
    Test type','Supplied parameters','Parameter run no'])
temp = temp.fillna(0)
temp['Query and histogram execution duration'] = temp['Query execution
    duration'] + temp['Histogram execution duration']
df_5 = temp
temp = temp.drop(columns = ['Histogram execution duration'])
temp = temp.set_index(['Test type','Supplied parameters','Parameter run no
    '])
```

```
sensitivity_df = pa.DataFrame(temp,index = temp.index).stack()
#new_df = new_df.columns(['Test type'])
sensitivity_df = sensitivity_df.to_frame()
sensitivity_df =sensitivity_df.reset_index()
sensitivity_df.columns = ['Test type','Supplied parameters','Parameter run
     no','Duration type','Duration']

#Splitting the params for the base classes test types
temp_data_df = sensitivity_df.loc[sensitivity_df['Test type'].isin(['No
    histogram ','Perfect histogram ','Stale histogram '])]
temp_data_df = temp_data_df[~temp_data_df['Duration type'].isin(['
    No_of_executes'])]
base_classes_data = temp_data_df[temp_data_df['Duration type'].isin(['
    Query and histogram execution duration'])].groupby(['Test type']).mean
    ().reset_index()
base_classes_data['Duration type'] = 'Query and histogram execution
    duration'
base_classes_data['Supplied parameters']= 'Not applicable'
base_classes_data['Parameter run no']= 'Not applicable'
temp_data_df = temp_data_df.set_index(['Test type','Duration type','
    Duration','Parameter run no'])
temp_data_df = pa.DataFrame(temp_data_df['Supplied parameters'].str.split(
    ' ').tolist(),index = temp_data_df.index).stack()
temp_data_df = temp_data_df.to_frame()
temp_data_df = temp_data_df.reset_index()
temp_data_df = temp_data_df.drop(columns='level_4')
temp_data_df.columns=['Test type','Duration type','Duration','Parameter
    run no','Supplied parameters']
sensitivity_df = sensitivity_df.loc[~sensitivity_df['Test type'].isin(['No
     histogram ','Perfect histogram ','Stale histogram '])]
sensitivity_df = sensitivity_df.append(temp_data_df, ignore_index=True,
    sort=False)
sensitivity_df = sensitivity_df.append(base_classes_data, ignore_index=
    True,sort=False)

#Before we remove some rows, let's save this to another df
full_sensitivity_df = sensitivity_df

#Removing the query and histogram execution duration for the base classes
    since we don't want that to show up in our plots.
temp_3 = sensitivity_df
temp_3 = temp_3.loc[temp_3['Test type'].isin(['No histogram ','Perfect
    histogram ','Stale histogram ']) & temp_3['Duration type'].isin(['
    Query execution duration'])]
temp_4 = pa.merge(sensitivity_df,temp_3, indicator=True, how='outer').
    query('_merge=="left_only"').drop('_merge', axis=1)
sensitivity_df = temp_4
sensitivity_df.drop_duplicates(keep = 'first', inplace = True)


# # Format the queries to be q1 q2 etc. add a result_ratio column and a
    formatted query column

# In[8]:


i=1
```

```
replace_dictionary = {}
unique_queries = result_df.Query.unique().tolist()
for query in unique_queries:
    name = 'q{}'.format(i)
    replace_dictionary[query]=name
    i+=1

result_df['Short_Query']=result_df['Query'].replace(replace_dictionary)

partition_size = min(result_df["Number of Inserts"].unique().tolist())
result_df['Result_ratio']=result_df['Returned rows']/partition_size

queries = result_df['Query'].tolist()
short_query_format=[]
for query in queries:
    no_of_joins = query.count('join')
    where_predicate_value = query.split("test.measurement.msm_value",1)[1]
    short_query_format.append('Query with '+str(no_of_joins)+' joins and
    WHERE MSM_VALUE' +where_predicate_value)
result_df['Short_query_format']=short_query_format



# In[9]:


unformatted_plan_files = glob.glob(plans_path)
for file in unformatted_plan_files:
    output_file_name=file+".tmp"
    with open(file, 'r') as f, open(output_file_name, 'w+') as fo:
        for line in f:
            formatted_line = line.replace('explain format = tree ', '').
    replace(";","")
            # print(formatted_line)
            for query,key in replace_dictionary.items():
                if query==formatted_line:
                    fo.write(key)
                    break
            fo.write(line)
    os.remove(file)


# ## Standard results

# In[ ]:


base_rules = ['No histogram ','Perfect histogram ','Stale histogram ']
for rule in result_df['Test type'].unique().tolist():
    if rule not in base_rules:
        base_rules.append(rule)
        temp = result_df.set_index('Test type')
        temp = temp.loc[base_rules]
        temp = temp.reset_index()
        for params in temp['Parameter run no'].unique().tolist():
            for query in temp.Short_Query.unique().tolist():
                fig = plt.figure(figsize=(13,9))
                gs = gridspec.GridSpec(nrows=4,
                                       ncols=1,
                                       figure=fig,
                                       height_ratios=[1, 1, 1, 1],
                                       wspace=0.3,
                                       hspace=0.3)

                ax1 = fig.add_subplot(gs[1:3,0])
                sns.lineplot(x="Number of Inserts", y="Duration",
                             hue="Test type", style="Test type",markers = True,dashes = True,
                             data=temp[temp.Short_Query.isin([query])&temp['Parameter run no'].isin([params])],ax=ax1)
                xlabels = ['{:,.0f}'.format(x) + 'K' for x in ax1.get_xticks()/1000]
                ax1.set_xticklabels(xlabels)
                ax1.set_xlabel("Number of updates")
                ax1.set_ylabel("Duration (s)")
               # ax1.set_title(str(result_df['Short_query_format'][result_df.Short_Query.isin([query])].iloc[0]))

                ax2 = fig.add_subplot(gs[0, 0])

                all_hist = result_df[result_df["Test type"].str.contains('Perfect histogram')]
                index_position = all_hist[all_hist.Short_Query.isin([query])].index[0]
                final = all_hist[all_hist.Short_Query.isin([query])].loc[index_position]
                sns.barplot(x="Number of Inserts", y="Result_ratio",data=final ,ax=ax2, color=sns.xkcd_rgb["denim blue"])


                ax2.spines['right'].set_visible(False)
                ax2.spines['top'].set_visible(False)
                #ax2.xaxis.set_major_locator(ax1.xaxis.get_major_locator())
                #ax2.set_xticklabels(ax1.get_xticklabels())
                ax2.set_xticklabels("")
                ax2.margins(x=ax1.margins()[0]-0.01)
                ax2.set_xlabel("")
                ax2.set_ylabel("Result size ratio")
              #  ax2.set_title(str(result_df['Short_query_format'][result_df.Short_Query.isin([query])].iloc[0]))

                #fig.suptitle(str(result_df['Short_query_format'][result_df.Short_Query.isin([query])].iloc[0]),y=0.9)

                line_plot = plt.gcf()
                path = "/export/home/tmp/Dropbox/Apper/ShareLaTeX/Master/eval_plots/line_plots/all_q_plots/"
                name = str(temp['Short_query_format'][temp.Short_Query.isin([query])].iloc[0])+str(base_rules)+str(params)+".png"
                line_plot.savefig(path+name,dpi=360,bbox_inches='tight')
                plt.close()
        base_rules.pop()
    if rule == 'No histogram ':
        temp = result_df.set_index('Test type')
        temp = temp.loc[base_rules]
        temp = temp.reset_index()
        for params in temp['Parameter run no'].unique().tolist():
            for query in temp.Short_Query.unique().tolist():
                fig = plt.figure(figsize=(13,9))
                gs = gridspec.GridSpec(nrows=4,
                                       ncols=1,
                                       figure=fig,
                                       height_ratios=[1, 1, 1, 1],
                                       wspace=0.3,
                                       hspace=0.3)

                ax1 = fig.add_subplot(gs[1:3,0])
                sns.lineplot(x="Number of Inserts", y="Duration",
                             hue="Test type", style="Test type",markers = True,dashes = True,
                             data=temp[temp.Short_Query.isin([query])&temp['Parameter run no'].isin([params])],ax=ax1)
                xlabels = ['{:,.0f}'.format(x) + 'K' for x in ax1.get_xticks()/1000]
                ax1.set_xticklabels(xlabels)
                ax1.set_xlabel("Number of updates")
                ax1.set_ylabel("Duration (s)")
               # ax1.set_title(str(result_df['Short_query_format'][result_df.Short_Query.isin([query])].iloc[0]))

                ax2 = fig.add_subplot(gs[0, 0])

                all_hist = result_df[result_df["Test type"].str.contains('Perfect histogram')]
                index_position = all_hist[all_hist.Short_Query.isin([query])].index[0]
                final = all_hist[all_hist.Short_Query.isin([query])].loc[index_position]
                sns.barplot(x="Number of Inserts", y="Result_ratio",data=final ,ax=ax2, color=sns.xkcd_rgb["denim blue"])


                ax2.spines['right'].set_visible(False)
                ax2.spines['top'].set_visible(False)
                #ax2.xaxis.set_major_locator(ax1.xaxis.get_major_locator())
                #ax2.set_xticklabels(ax1.get_xticklabels())
                ax2.set_xticklabels("")
                ax2.margins(x=ax1.margins()[0]-0.01)
                ax2.set_xlabel("")
                ax2.set_ylabel("Result size ratio")
              #  ax2.set_title(str(result_df['Short_query_format'][result_df.Short_Query.isin([query])].iloc[0]))

                #fig.suptitle(str(result_df['Short_query_format'][result_df.Short_Query.isin([query])].iloc[0]),y=0.9)

                line_plot = plt.gcf()
                path = "/export/home/tmp/Dropbox/Apper/ShareLaTeX/Master/eval_plots/line_plots/all_q_plots/"
```

```
                name = str(temp['Short_query_format'][temp.Short_Query.
    isin([query])].iloc[0])+str(base_rules)+str(params)+".png"
                line_plot.savefig(path+name,dpi=360,bbox_inches='tight')
                plt.close()

fig = plt.figure(figsize=(9,6))
ax3=sns.barplot(x="Test type", y="Duration",hue="Test type", data=
    hist_result_df, estimator=sum,ci=None,dodge = False,palette = sns.
    color_palette("Paired", 9))
ax3.set_xticklabels("")
ax3.set_xlabel("")
ax3.set_ylabel("Duration (s)")
hist_timing_plt = plt.gcf()
hist_timing_plt.savefig("/export/home/tmp/Dropbox/Apper/ShareLaTeX/Master/
    eval_plots/histogram_timing.png",dpi=360,bbox_inches='tight')


# # One off plots#

# In[ ]:


rules = ['No histogram ','Perfect histogram ','Stale histogram ']
rule_to_visit = 'Plugin rule 9'
rules.append(rule_to_visit)
params = '6'
temp = result_df.set_index('Test type')
temp = temp.loc[rules]
temp = temp.reset_index()
blah = temp
for query in temp.Short_Query.unique().tolist():
    fig = plt.figure(figsize=(13,9))
    gs = gridspec.GridSpec(nrows=4,
                           ncols=1,
                           figure=fig,
                           height_ratios=[1, 1, 1, 1],
                           wspace=0.3,
                           hspace=0.3)

    ax1 = fig.add_subplot(gs[1:3,0])
    sns.lineplot(x="Number of Inserts", y="Duration",
                 hue="Test type", style="Test type",markers = True,dashes
    = True,
                 data=temp[temp.Short_Query.isin([query])&temp['Parameter
    run no'].isin([params])],ax=ax1)
    xlabels = ['{:,.0f}'.format(x) + 'K' for x in ax1.get_xticks()/1000]
    ax1.set_xticklabels(xlabels)
    ax1.set_xlabel("Number of updates")
    ax1.set_ylabel("Duration (s)")
    # ax1.set_title(str(result_df['Short_query_format'][result_df.
    Short_Query.isin([query])].iloc[0]))

    ax2 = fig.add_subplot(gs[0, 0])

    all_hist = result_df[result_df["Test type"].str.contains('Perfect
    histogram')]
    index_position = all_hist[all_hist.Short_Query.isin([query])].index[0]
    final = all_hist[all_hist.Short_Query.isin([query])].loc[index_position]
    sns.barplot(x="Number of Inserts", y="Result_ratio",data=final ,ax=ax2, color=sns.xkcd_rgb["denim blue"])


    ax2.spines['right'].set_visible(False)
    ax2.spines['top'].set_visible(False)
    #ax2.xaxis.set_major_locator(ax1.xaxis.get_major_locator())
    #ax2.set_xticklabels(ax1.get_xticklabels())
    ax2.set_xticklabels("")
    ax2.margins(x=ax1.margins()[0]-0.01)
    ax2.set_xlabel("")
    ax2.set_ylabel("Result size ratio")
    #  ax2.set_title(str(result_df['Short_query_format'][result_df.Short_Query.isin([query])].iloc[0]))

    #fig.suptitle(str(result_df['Short_query_format'][result_df.Short_Query.isin([query])].iloc[0]),y=0.9)


    line_plot = plt.gcf()

    path = "/export/home/tmp/Dropbox/Apper/ShareLaTeX/Master/eval_plots/line_plots/"+str(rule_to_visit)+"/"
    name = str(temp['Short_query_format'][temp.Short_Query.isin([query])].iloc[0])+str(rules)+str(params)+".png"#+str(temp['Supplied parameters'][temp.Short_Query.isin([query]) & temp['Test type'].isin([rule_to_visit])& temp['Parameter run no'].isin([params])].iloc[0])+".png"
    line_plot.savefig(path+name,dpi=360,bbox_inches='tight')
    plt.close()


# # Generate Q-ratio data#

# In[13]:


q_ratio_df=df_5.loc[df_5['Test type'].isin(['No histogram ','Perfect histogram ','Stale histogram ','Plugin rule 0'])].groupby(['Test type']).mean().reset_index()
q_ratio_df['Supplied parameters'] = 'Not applicable'
q_ratio_df = q_ratio_df.append(df_5.loc[~df_5['Test type'].isin(['No histogram ','Perfect histogram ','Stale histogram ','Plugin rule 0'])],ignore_index=True,sort=False)
q_ratio_df = q_ratio_df[['Test type','Query and histogram execution duration','Query execution duration','Histogram execution duration','No_of_executes','Supplied parameters']]
q_ratio_df=q_ratio_df.rename(columns={"Query and histogram execution duration": "Total duration", "No_of_executes": "Number of updates","Histogram execution duration":"Histogram duration","Query execution duration":"Query duration"})
q_ratio_df['Q-ratio']=q_ratio_df['Total duration'].apply(lambda x: x/q_ratio_df['Total duration'].loc[q_ratio_df['Test type']=='Perfect histogram '], 0)
q_ratio_df = q_ratio_df.sort_values('Q-ratio').set_index('Test type').
```

```
    astype({'Number of updates': 'int'}).reset_index()
q_ratio_df = q_ratio_df[['Test type','Total duration','Q-ratio','Query
    duration','Histogram duration','Number of updates','Supplied
    parameters']]
#Formatting
q_ratio_df['Test type'] = q_ratio_df['Test type'].replace("Plugin rule 0",
     "Rule 0")
q_ratio_df['Test type'] = q_ratio_df['Test type'].replace("Plugin rule 2",
     "Rule 2")
q_ratio_df['Test type'] = q_ratio_df['Test type'].replace("Plugin rule 3",
     "Rule 3")
q_ratio_df['Test type'] = q_ratio_df['Test type'].replace("Plugin rule 6",
     "Rule 6")
q_ratio_df['Test type'] = q_ratio_df['Test type'].replace("Plugin rule 7",
     "Rule 7")
q_ratio_df['Test type'] = q_ratio_df['Test type'].replace("Plugin rule 9",
     "Rule 9")



#Generate table file
q_ratio_df = q_ratio_df.reset_index()

temp = q_ratio_df.loc[q_ratio_df.groupby('Test type')['Q-ratio'].idxmin()]
#temp_max = q_ratio_df.loc[q_ratio_df.groupby('Test type')['Q-ratio'].
    idxmax()]
#temp = temp.append(temp_max).drop_duplicates(keep = 'first', inplace =
    False)

summarised_q_ratio = temp.sort_values('Q-ratio')
summarised_q_ratio = summarised_q_ratio[['Test type','Total duration','Q-
    ratio','Query duration','Histogram duration','Number of updates','
    Supplied parameters']].set_index('Test type')

f = open('/export/home/tmp/Dropbox/Apper/ShareLaTeX/Master/eval_plots/q-
    ratio_table.tex',"w+")
f.write("\\noindent\makebox[\\textwidth]{%\n")
f.write("\\begin{tabularx}{\\textwidth}{LLLLLL}\\toprule\n")
f.close()
summarised_q_ratio.to_csv('/export/home/tmp/Dropbox/Apper/ShareLaTeX/
    Master/eval_plots/q-ratio_table.tex',sep='&',line_terminator = "\\\\\n
    ",mode='a',float_format='%.3f')
f = open('/export/home/tmp/Dropbox/Apper/ShareLaTeX/Master/eval_plots/q-
    ratio_table.tex','a')
f.write ("\\bottomrule\n")
f.write ("\\end{tabularx}\n")
f.write("}")
f.close()




#Show data here as well
summarised_q_ratio.head(5000)


# In[11]:
```

```


q_ratio_df.head(100)


# # Generate sensitivity plots for single parameter rules#

# In[12]:


base_rules = ['No histogram ','Perfect histogram ','Stale histogram ']
rules_to_avoid_for_now = []#['Plugin rule 9','Plugin rule 3']
for rule in condensed_sens_with_err['Test type'].unique().tolist():
    if rule not in base_rules and rule not in rules_to_avoid_for_now:
        current_rules=base_rules.copy()
        current_rules.append(rule)
        temp = condensed_sens_with_err.set_index('Test type')
        temp = temp.loc[current_rules]
        temp = temp[~temp['Supplied parameters'].isin(['Not applicable'])]
        run_no_finder = temp.groupby('Test type')
        run_no_finder = run_no_finder.apply(lambda x: x['Parameter run no'
    ].unique())
        unique_run_no_we_want_to_use = run_no_finder.loc[rule].tolist()
        param_list_finder = temp.groupby('Test type')
        param_list_finder = param_list_finder.apply(lambda x: x['Supplied
    parameters'].unique())
        unique_params_we_want_to_use = param_list_finder.loc[rule].tolist
    ()
        temp = temp[temp['Parameter run no'].isin(
    unique_run_no_we_want_to_use)]
        rule_df = temp.loc[rule].reset_index()#.set_index(['Parameter run
    no'])
        base_df = temp.loc[base_rules].reset_index().set_index(['Parameter
     run no'])
        base_df['Supplied parameters'] = np.NaN
        print(rule,unique_run_no_we_want_to_use)
        for param_run_no in unique_run_no_we_want_to_use:
            #if not (rule == 'Plugin rule 9' and param_run_no == 12):
            #    print (unique_params_we_want_to_use[param_run_no-1])
            base_df.loc[param_run_no,'Supplied parameters'] =
    unique_params_we_want_to_use[param_run_no-1]
            data_df = rule_df.append(base_df).astype({'Supplied parameters
    ': 'float'})
        fig = plt.figure(figsize=(13,9))
        gs = gridspec.GridSpec(nrows=4,
                               ncols=1,
                               figure=fig,
                               height_ratios=[1, 1, 1, 1],
                               wspace=0.3,
                               hspace=0.3)
        ax1 = fig.add_subplot(gs[1:3,0])
        g=sns.lineplot(x="Supplied parameters", y="Duration",
                       markers=True, hue = 'Test type', style = 'Duration
    type',ci=99,
                       data=data_df,ax=ax1)
        ax1.set_ylabel("Duration (s)")
        ax1.set_ylim([70,250])
```

```python
    if rule == 'Plugin rule 0':
        plt.xscale("log")
        ax1.set_xticklabels("")
        ax1.set_xlabel("Paramter values - not applicable")
    if rule == 'Plugin rule 2':
        plt.xscale("log")
        plt.yscale("linear")
        ax1.set_xlabel("Parameter values for n, \"Number of rows between updates\"")
        xlabels = ['{:,.0f}'.format(x) + 'K' for x in ax1.get_xticks()/1000]
        ax1.set_xticklabels(xlabels)
    if rule == 'Plugin rule 6':
        plt.xscale("log")
        ax1.set_xlabel("Parameter values for r, \"Ratio of change before updates\"")
    if rule == 'Plugin rule 7':
        plt.xscale("linear")
        ax1.set_xlabel("Parameter values for w, \"Update outside histogram range\"")
    if rule == 'Plugin rule 9':
        plt.yscale("linear")
        plt.xscale("log")
        ax1.set_xlabel("Parameter values for I, \"Inverse sensitivity to change\"")
        xlabels = ['{:,.0f}'.format(x) + 'K' for x in ax1.get_xticks()/1000]
        ax1.set_xticklabels(xlabels)
    g.legend_.remove()
    sensitivity_plot = plt.gcf()
    path = "/export/home/tmp/Dropbox/Apper/ShareLaTeX/Master/eval_plots/sensitivity_plots/"
    name = str(current_rules)+".png"
    sensitivity_plot.savefig(path+name,dpi=360,bbox_inches='tight')
    current_rules.pop
    plt.close()

    if rule == "Plugin rule 0":
        current_rules=base_rules.copy()
        current_rules.append(rule)
        temp = condensed_sens_with_err.set_index('Test type')
        temp = temp.loc[current_rules]
        temp = temp[~temp['Supplied parameters'].isin(['Not applicable'])]
        run_no_finder = temp.groupby('Test type')
        run_no_finder = run_no_finder.apply(lambda x: x['Parameter run no'].unique())
        unique_run_no_we_want_to_use = run_no_finder.loc[rule].tolist()
        param_list_finder = temp.groupby('Test type')
        param_list_finder = param_list_finder.apply(lambda x: x['Supplied parameters'].unique())
        unique_params_we_want_to_use = param_list_finder.loc[rule].tolist()
        temp = temp[temp['Parameter run no'].isin(unique_run_no_we_want_to_use)]
        rule_df = temp.loc[rule].reset_index()#.set_index(['Parameter
run no'])
        base_df = temp.loc[base_rules].reset_index().set_index(['Parameter run no'])
        base_df['Supplied parameters'] = np.NaN
        for param_run_no in unique_run_no_we_want_to_use:
            base_df.loc[param_run_no,'Supplied parameters'] = unique_params_we_want_to_use[param_run_no-1]
            data_df = rule_df.append(base_df).astype({'Supplied parameters': 'float'})
        data_df.head(200)
        fig = plt.figure(figsize=(13,9))
        gs = gridspec.GridSpec(nrows=4,
                           ncols=1,
                           figure=fig,
                           height_ratios=[1, 1, 1, 1],
                           wspace=0.3,
                           hspace=0.3)
        ax1 = fig.add_subplot(gs[1:3,0])
        g=sns.lineplot(x="Supplied parameters", y="Duration",
                     markers=True, hue = 'Test type', style = 'Duration type',ci=99,
                     data=data_df,ax=ax1)
        plt.xscale("log")
        ax1.set_ylabel("Duration (s)")
        ax1.set_ylim([100,350])
        ax1.set_xlabel("Parameter values for a given rule X")
        ax1.set_xticklabels("")
       # handles, labels = ax1.get_legend_handles_labels()
       # ax1._legend.remove()
       # ax1.fig.legend(handles, labels, ncol=2, loc='upper center',
             #   bbox_to_anchor=(0.5, 1.15), frameon=False)
        #ax1.legend(frameon=False, loc='right', ncol=2,framealpha=1)
        handles, _ = g.get_legend_handles_labels()
        g.legend(handles,["Plugin rule","Plugin rule X","No histogram","Perfect histogram","Stale histogram","Duration type","Total duration","Query duration"],frameon=True,bbox_to_anchor=(0.35, 1.3),ncol=2, loc=2, borderaxespad=0.)
        #plt.legend(frameon=True,bbox_to_anchor=(0.25, 1.3),ncol=2, loc=2, borderaxespad=0.)
        sensitivity_plot = plt.gcf()
        path = "/export/home/tmp/Dropbox/Apper/ShareLaTeX/Master/eval_plots/sensitivity_plots/"
        name = "Special_explanation_plot"+".png"
        sensitivity_plot.savefig(path+name,dpi=360,bbox_inches='tight')
        plt.close()
        current_rules.pop()
```

Sevre Vestrheim

Automatic updating of histograms in MySQL