Master's thesis

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

Sander Nicolausson
Jørgen Aasvestad

Semi-Automatic Statistics Management: Exploiting Existing Data Streams to Improve Selectivity Estimation in MySQL's Hypergraph Optimizer

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

NTNU
Norwegian University of Science and Technology

Sander Nicolausson
Jørgen Aasvestad

Semi-Automatic Statistics Management: Exploiting Existing Data Streams to Improve Selectivity Estimation in MySQL's Hypergraph Optimizer

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

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

NTNU
Norwegian University of
Science and Technology

Abstract

MySQL's new hypergraph optimizer employs more cost-based decision-making compared to the original optimizer, and should therefore more prominently utilize database statistics. The initial experiment highlights the effect of accurate statistics, where injected correct selectivities for the Join Order Benchmark queries leads to improved performance. This emphasizes the importance of accurate and up-to-date statistics, leading to the idea of exploiting existing data streams in MySQL to create, update and manage such statistics.

This thesis presents a novel implementation of count-min and count-min mean sketch for estimating data distribution in MySQL and compares the performance with histograms on the Join Order Benchmark. Results indicate an overall improvement in performance and selectivity estimation accuracy, albeit with some exceptions where performance is negatively affected. Later implementations of the hypergraph optimizer with a more refined cost model are assumed to reduce the degree of such occurrences. The experimental results show promise regarding the exploitation of existing data streams to generate statistics in MySQL, serving as a foundation for subsequential implementations of semi-automatic statistic management.

Sammendrag

MySQLs nye hypergraf optimizer benytter flere kostnadsbaserte valg sammenlignet med den orginale optimizeren, og bør derfor kunne utnytte database-statistikk i større grad. Innledende eksperimenter synliggjør effekten av nøyaktig statistikk hvor korrekte selektiviter for Join Order Benchmark-spørringer fører til økt ytelse. Dette understreker viktigheten av nøyaktig og oppdatert statistikk i databaser, som leder inn på ideen om å utnytte eksisterende datastrømmer i MySQL for å lage, oppdatere og vedlikeholde slik statistikk.

Denne avhandlingen presenterer en ny implementasjon av count-min og count-min mean skisser for estimering av datadistribusjon i MySQL, og sammenligner ytelsen med histogrammer for Join Order Benchmark. Resultatene indikerer en helhetlig forbedring i ytelse samt bedre treffsikkerhet på selektivitetsestimering, riktignok med enkelte unntak der ytelsen blir påvirket negativt. Senere implementasjoner av hypergraf-optimizeren med en mer robust kostnadsmodell er antatt å redusere graden av slike hendelser. Eksperimentene viser lovende resultater i retning mot å utnytte eksisterende datastrømmer for å generere statistikk i MySQL. Dette legger grunnlaget for påfølgende implementasjoner av semi-automatisk håndtering av statistikk.

Acknowledgement

Thank you to our supervisor Norvald H. Ryeng for feedback and advice on this Master's thesis. We would also like to show gratitude towards Oracle's MySQL team here in Trondheim for allowing us to use their facilities, with a special thanks to Knut Anders Hatlen who guided us through the intricate web of c++ code that is the MySQL source code.

Sander Nicolausson
Jørgen Aasvestad

Trondheim, June 12, 2022

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The internet sees an influx of new people connecting every day. During a five-year span - from 2018 to 2023 - the number of internet users is estimated to increase with 1.4 billion, leading to a total of around 5.3 billion users in 2023 according to Cisco [2]. An increasingly connected population paired with increasing individual consumption of internet services leads to a larger total data volume and more data traffic every year. These demands require fast and reliable systems, and database management systems continue to be the basis of most mission-critical applications [3]. This pressures the database system developers to continuously work on achieving the best possible performance and efficiency. Improvements in hardware components will necessarily play a part in the performance of the systems, but the inner components of a database system will have an impact, such as the query optimizer, query processing, and storage methods.

An important component in the database management system is the query optimizer, which is a large contributing factor to the speed of query execution. The optimizer determines how the query should be executed by selecting an optimal or suitable query plan. A query can be executed in an endless number of ways, based on different join orders, join methods, or access methods. Database statistics may provide valuable insight to the optimizer regarding the data itself, leading to more accurate estimations and a better foundation for the cost metrics, thus aiding the plan selection. A significant portion of the earlier research on the topic has been on the creation of various types of database statistics that are accurate and efficient. Recent years have seen a rise in research on maintenance and automatic adaption of said statistics and has shown promising results [3; 4; 5; 6]. Automatic management of database statistics may provide more up-to-date statistics better suited for changes in the data, significantly increasing usability and accuracy.

This thesis intends to further explore MySQL's new hypergraph optimizer, how it responds to various types of statistics, and different methods of creating these statistics. The rest of this chapter is structured as follows: Section 1.1

covers the background and motivation for the thesis and its place in the area of database management systems. Section 1.2 presents the goal for the thesis and the research questions utilized to accomplish said goal. Section 1.3 describes the research methodology applied and the reasoning behind the methodology. The following section, Section 1.4, briefly summarizes the thesis' main contributions. Lastly, Section 1.5 covers the structure of the thesis for the following chapters.

1.1 Background and Motivation

Oracle's MySQL team is currently in the process of developing and implementing a new query optimizer that represents join relations as edges of a hypergraph. This optimizer makes decisions substantially more based on cost than the current optimizer, and has the aim of improving query plan selection with better cardinality estimates. The use of statistics to estimate cardinalities is already extensive in relational database management systems with the use of histograms. MySQL also relies on histograms to provide statistics for use in selectivity estimates [7], but as of now, these need to be created and maintained manually. There are also other structures used for statistics, such as sketches [8; 9], that may be eligible for use in MySQL to improve selectivity estimates. The need for accurate or elaborate statistics has not been as prevalent in MySQL, as the current optimizer is significantly more heuristic-based, forming a large portion of its decisions based on existing heuristics [10]. As the hypergraph optimizer is still in a phase of development it is too early to determine the exact impact of having precise information available, but experiments show that the response to more accurate statistics is promising [11]. These factors allow for further exploration on the topic of automatically gathering information on the data that are inserted into the database.

In this thesis, the prospect of piggybacking an existing data stream in MySQL to build relevant statistics for use in the optimization stage of query processing is explored. It aims to determine what data structures are suitable to achieve a fast and precise estimation of the data distribution, and how this compares to traditional histogram statistics. The hypergraph optimizer is also given the correct information in order to determine how it behaves in a best-case scenario.

1.2 Goals and Research Questions

The goal of the thesis is as follows:

Goal *Exploring the possibility of automatic statistic creation in MySQL with the intent of improving the performance of the hypergraph optimizer.*

An important aspect of a database management system is performance. A database that is responsive and fast will positively impact the user experience and its applicability. While there are several factors to explore on the topic of performance, the query plan selection done by the optimizer has a large impact on the execution speed. MySQL's new hypergraph optimizer follows a more rigid cost-based approach, with estimates based to a higher degree on the information available to the optimizer. Automatic statistic creation and management may ensure that the stored statistics are more up-to-date and accurate, which in turn will benefit the hypergraph optimizer greatly. A stepping stone towards this direction may be to explore the effect of utilizing the data streams created by existing database operations, to improve the accuracy of the statistics. These results will hopefully set precedence or a starting point for further exploration of automatic statistic management in MySQL.

To accomplish the goal, the following research questions are explored:

Research question 1 *How does MySQL's hypergraph optimizer respond to having access to correct statistics?*

In theory, the hypergraph optimizer should perform better with improved statistics due to the implementation of a stricter cost-based approach. What effect does it have when correct selectivities for predicates in a query are injected into it?

Research question 2 *How can existing data streams in MySQL be exploited to create statistics?*

During the execution of a query, some of the data or records are handled by the database system. How can these existing operations that already read the data be exploited to continuously create or update statistics, in contrast to explicitly conducting a sample?

Research question 3 *What are the alternatives to histograms when it comes to storing statistics from data streams?*

Considering alternatives to the histogram may highlight areas where the histogram underperform, and provide an indication on advantages and disadvantages of both histograms and other alternatives.

Research question 4 *How can a combination of sketches and histograms improve the performance of MySQL's hypergraph optimizer?*

Different statistic types will serve dissimilar purposes and may therefore not have the same strengths and weaknesses. A solution that utilizes a combination of both sketches and histograms could therefore be able to amplify the strengths of the two types while reducing the impact of their weaknesses.

1.3 Research Method

The research strategy applied in this thesis is an experiment and is chosen due to the fact that **Research Question 1** and **Research Question 2** insinuate an experiment to evaluate the feasibility of implementations that can help answer these questions.

As an experiment is used as the research method, two main hypotheses have been created in order to prove or disprove which factors contribute to certain outcomes [12]. These hypotheses are closely tied with the aforementioned research questions:

Hypothesis 1 *MySQL's hypergraph optimizer improves performance when statistics improve.*

The experiment tied to **Hypothesis 1** consists of executing the Join Order Benchmark queries with MySQL's hypergraph optimizer with selectivity estimates of various stages of precision. The stages are no statistics (1), histogram statistics (2), and correct selectivity (3). The implementation of (3) is further elaborated in Section 3.1, while the experimental setup is presented in Section 4.2.1.

Hypothesis 2 *Existing data streams in MySQL can be utilized to get more precise selectivity estimates.*

Hypothesis 2 is explored by implementing two different sketch-datastructures, the count-min, and the count-min mean sketches. The Join Order Benchmark queries are also executed here with the hypergraph optimizer, now with the selectivity being estimated by sketches of varying sizes. Implementation of the sketches can be found in Section 3.2, while the setup of the experiment is presented in Sections 4.2.2 and 4.2.3.

Quantitative data analysis is used to present the results. As the data from the experiment has true zero to the scale of measurement, the results provide ratio data. The central tendency used to describe the data in Section 4.3 is the mean of several executions of each JOB query, while the visual aid that is used to present the data is bar charts.

1.4 Contributions

In this section, the main contributions of the work are summarized in a brief fashion, while a further elaboration on these contributions can be found in Section 5.2.

1. *Determining the impact of injecting correct selectivities to MySQL's hypergraph optimizer for the Join Order Benchmark.*
2. *Conducting an experiment to determine the possible impact of using count-min and count-min mean sketches to improve selectivity estimates for MySQL's hypergraph optimizer.*
3. *Highlighting the effect of database statistics created from data streams in MySQL.*
4. *Addressing directions for future work on exploiting existing data streams for semi-automatic statistic management.*

The injection of correct selectivities illustrates the impact of improved statistics for MySQL's new hypergraph optimizer, laying the foundation for further research on the topic of improved database statistics. The implementation of the count-min and count-min mean sketches allows the exploitation of existing data streams in MySQL and highlights the impact of such statistics on the optimizer. Exploiting existing data streams reduces the need for explicit statistic collection, such as samples, which in turn leads to fewer resources required to achieve equivalent or even better statistics. Experimental results from the aforementioned topics are used to address direction for further work on the topic of semi-automatic statistic management based on existing data streams.

1.5 Thesis Structure

The thesis consists of a total of five chapters with the next one, Chapter 2, covering background theory related to the experiment. We take a look at how query processing is done in relational database management systems in general and MySQL specifically and how the query optimizer uses statistics to make informed choices on how to execute the query. Topics such as data stream processing and hash functions are covered, alongside the theory on count-min sketches which are implemented and used in the experiment conducted.

Chapter 3 provides an explanation of how the structures used in the experiment are implemented in MySQL, in which files, and how this changes the flow of execution for the MySQL hypergraph optimizer.

In Chapter 4 the experimental plan is presented, with an explanation of which questions or sub-questions each part of the experiment aims to answer. Further, the setup of the experiment is covered in such a fashion that it can be reproduced. Finally, a presentation and evaluation of the most relevant results to lay the foundation for the discussion and conclusions to come in Chapter 5.

Chapter 2

Background Theory

This chapter presents the background theory that lays the foundation for the rest of the thesis. Section 2.1 covers how a query is processed by an RDBMS and takes a deeper dive into how the query optimizer operates by looking at optimization techniques and query plan selection. It also looks at the new hypergraph optimizer in MySQL. After having covered how the optimization of a query is done, Section 2.2 investigates the types of statistics the optimizer utilizes to make decisions on what query execution plan should be chosen. Further, a closer look at four different commercial database systems and what kinds of statistics these systems use. Finally, Sections 2.3 and 2.4 explore the topics of data stream processing and hash functions respectively, before existing theory on count-min sketches is investigated.

2.1 Query Processing

Utilizing a SQL query is a way of retrieving meaningful or relevant information from a database. From the user's point of view, this seems like a fairly straightforward task: execute a query and the results are presented almost instantaneously, for many queries at least. However, before the query can yield its desired results, the DBMS must process it internally. Query processing can be defined as the collection of operations involved in extracting the data from a database [13]. Although there are some common similarities, various database systems will have different approaches for how they handle this processing. As presented by Elmasri and Navathe, a general approach might be explained with a few common steps [14]. The first step is **scanning** the query, identifying query language keywords, relation names, and attribute names, also known as tokens or language components. Next up is **parsing**, checking the query syntax to determine if it corresponds to the rules of the query language. Scanning and parsing is essentially performed simultaneously, as scanning continuously produces lexemes sent to parsing. After this, the query is **validated**, making sure that attribute and relations names are semantically meaningful and valid considering the database in use. Lastly, an internal representation of the query is created, known as a **query plan**. This internal representation describes an approach for retrieving the results, usually based on relational algebra. There are however many ways of reaching the same result, and a single query will have multiple possible methods of execution. An important part of query processing is consequently the selection of a suitable execution path, which is the job of the **query optimizer**.

The query optimizer then selects a query plan or execution plan and passes this to the **code generator**. In this stage, the query plan is transformed into executable code that executes the given plan. Lastly, the code is run and the results are produced, with the help of the **runtime database processor**.

As mentioned previously, the ways of handling the internal query processing may differ between database systems. Figure 2.1 highlights the differences between the textbook approach for query processing compared to the described approach for a selection of modern database systems, including MySQL, PostgreSQL, and Microsoft SQL Server. Overall, the query processing approaches are relatively similar, which is not as surprising as they all intend to reach an equal end goal. There are however some differences in the naming schema of components and phases, in addition to dissimilar placements of the operations. As an example, Microsoft SQL Server performs rewriting as an early part of the optimizer step, while PostgreSQL has a defined "Rewrite system" component prior to the planner/optimizer step. MySQL also has differences compared to the textbook approach, defining the first step as parsing [10]. The parsing step in MySQL checks the syntax of the query, parses the languages, and creates an abstract syntax tree, which in

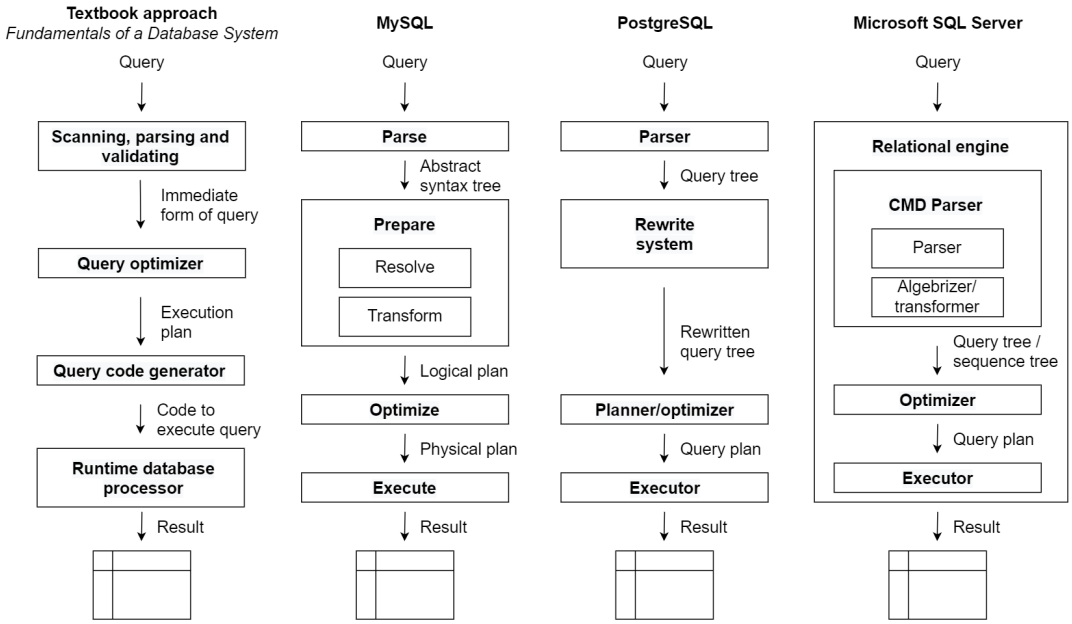


Figure 2.1: Query processing as described in MySQL, PostgreSQL and MS SQL Server compared to textbook approach described in *Fundamentals of a Database System*. Main differences are seen in the form of a merge or split of selected steps and renamed phases or components. Figure adapted and based on information from [14; 10; 15; 16]

practice is a combination of scanning and parsing as described earlier. The next step is preparing, which consists of resolving and transforming. This step performs both name-binding and static query transformation. Since these transformations are static - and not cost-based - this includes transformations such as IN/EXISTS/ANY expressions to SEMI JOIN and view expansion transformations. The prepare-step creates a logical plan that is sent to the next step, optimization. This step intends to find the best, or more accurately the *most optimal* query plan. The executable physical plan is sent to the execute-step, where the query is executed and at last a result is returned. The next section goes further into detail on an important part of the query processing, namely the query optimizer.

2.1.1 Query Optimizer

Due to the nature of SQL being a declarative language, a query will specify **what** it wants to be done, without regarding specifics on **how** it should be done [14]. Exactly how a query should be executed is a concern for the DBMS and the main

purpose of the query optimizer. The query optimizer intends to transform the internal query representation into an executable query plan that is as efficient as possible [17]. Since there are multiple ways of reaching the same result, the optimizer must choose between multiple plans when selecting the most desirable plan of execution. As a side note, the term query optimization can be seen as a bit misleading, as in most cases finding the absolute best - or most optimal - plan can be excessively time-consuming or even impossible [14]. A more suitable term would be that the optimizer aims to find reasonably efficient execution strategy.

Traditional query optimization can be split into two main phases of processing, the logical and the physical optimization. While the logical optimization intends to rewrite the query to a semantically equivalent but more optimal form, the physical optimization looks at access methods, join orders, and methods in order to optimize the query plan [18]. There are generally two methods of optimization, **heuristic optimization**, and **cost-based optimization**. Many modern DBMSes utilize a combination of both heuristic and cost-based optimization. An important point is that heuristic optimization can be applied first to reduce the number of plans the cost-based approach must consider.

Heuristic Optimization

The goal of heuristic optimization is to rewrite the internal representation of the query to a more efficient one. This may sometimes require catalog lookups, but does not require access to the data. Heuristic optimization and rewriting are similar in practice, where they both share techniques such as transformation of the query based on predefined rules without changing the semantics of the query. It is hard to set a defined border between the two as it may vary among different systems, with MS SQL Server being an example of a system where rewriting is performed as an early step within the optimizer [17]. An important difference is that the goal of rewriting is to translate the query to canonical or normalized form, making plan comparisons easier for the optimizer, while the goal of heuristic optimization is to create a fast execution plan. The rewriting to canonical form ensures a known starting point for the optimizer and makes sure that semantically equivalent queries are optimized equally. Some parts of the rewriting are view expansion, logical rewriting of predicates, constant arithmetic evaluation, and subquery flattening. Without changing the semantics of the query, the optimizer can then transform and alter the query to improve efficiency [14].

Based on this, it is therefore important to have a set of rules for equivalence among algebraic expressions which can be exploited when optimizing the query. One of the main heuristic rules is to apply SELECT or PROJECT operations before applying binary operations such as JOIN. JOIN or other binary operations are usually multiplicative functions of the input, therefore SELECT or PROJECT

should be applied first to reduce the number of tuples and attributes respectively. Additionally the SELECT or PROJECT operations that are the most restrictive, leading to the fewest amount of tuples or attribute size, should be applied prior to other similar operations. For the optimizer to be able to perform the aforementioned transformation of the query, it must have a set of rules on relational algebra equivalence to ensure that the query is still semantically equivalent. Some examples of common rules used are [14]:

Cascade of selection A conjunctive selection can be split into a sequence of multiple individual selections.
$\sigma_{AandBandC}(R) \equiv \sigma_a(\sigma_b(\sigma_c(R)))$

Commutativity of selection The select operation is commutative, meaning that the order of selection can be swapped.
$\sigma_a(\sigma_b(R)) \equiv \sigma_b(\sigma_a(R))$

Cascade of projection For a sequence of project operations, all but the last one may be removed.
$\pi_{a,b,c}(\pi_{b,c}(\pi_c(R))) \equiv \pi_c(R)$

Commutativity of inner join and cross product The order of join does not affect the result.
$R \bowtie S \equiv S \bowtie R$
$R \times S \equiv S \times R$

Commutativity of selection and inner join/cross product If all attributes in the selection condition c are only on one of the relations R or S, the operations are commutative, meaning that selection can be done prior to the join.
$\sigma_c(R \bowtie S) \equiv (\sigma_c(R)) \bowtie S$

The rules presented are a small sample of the rules that can be utilized for the heuristic optimization. These rules allow the optimizer to transform the query step by step to a more efficient representation, while still preserving semantic equivalence.

Cost-Based Optimization

For cost-based optimization, the optimizer tries to determine the cost of operations needed to execute the plan [19]. Multiple possible parameters could be taken into account for these estimations, where typical examples can be CPU cost, memory, I/O, and network (for distributed systems). These costs are summarized and then used as an overall metric when comparing the different plans. It is essential to note

that these cost functions are estimates, which means that the plan with the lowest cost is not necessarily guaranteed to be the absolute best plan. The accuracy of these estimates can be increased with catalog information, where both table statistics and column statistics have been proven to be helpful [14]. This may be statistics such as tuple size, number of tuples, number of disk blocks, number of distinct values, selectivity, and value distribution.

An important part of cost-based optimization is the cost model. The cost model defines how the costs of the plans are calculated, for example by defining the attributes and operations to consider and the weighting between them. As a simplified example, one could consider the cost model of single relation access in System R [1]:

$$COST = PAGE\ FETCHES + W \cdot RSI\ CALLS$$

In essence, this cost model utilizes a weighted measure between I/O (page fetches) and CPU (RSI calls). W is the weighting factor, that can be adjusted to emphasize either I/O or CPU, page fetches is the number of pages fetched from disk while RSI calls is the predicted tuples returned from the Research Storage System (RSS), the storage subsystem in System R. In theory, this cost model aims to ensure that the plan with the lowest cost is the plan requiring the least resources to execute. However, this might not always be the case as it is dependent on both the cost model and the information available to the optimizer. As previously stated the RSI calls is the *predicted* number of tuples returned from the RSS, meaning that this is an estimation and not the exact number. Knowledge of the exact number of tuples is impossible to retrieve unless the query is already executed or the data is explicitly read. These estimations, therefore, require the optimizer to make assumption and have information regarding the data, which is where the catalog look-ups play their part. The catalog look-ups in System R retrieves statistics on relations and the access paths available in the query and are defined in Table 2.1.

Table 2.1: Selection of catalog statistics in System R. Table adapted from [1].

Abbreviation	Definition
NCARD(T)	The cardinality of relation T
TCARD(T)	The number of pages that contains tuples of relation T
P(T)	The fraction of pages that contains tuples of relation T. P(T) = TCARD(T) / (number of non-empty pages in segment
ICARD(I)	The number of distinct keys on index I
NINDX(I)	The number of pages on index I

In system R, these statistics are updated periodically to reduce the inaccuracies after modifications of the data. These statistics are used to assign a selectivity factor F to each of the predicates. The selectivity factor will be an estimation of the fraction of tuples that satisfies the given predicate, and if possible are calculated with the aid of the catalog look-up statistics. The following list displays an excerpt of predicate types and their respective selectivity factors F in System R [1]:

column = value F=1/ICARD(Column Index) if there exists index on column.
F=1/10 for all other cases.

column1 = column2 F=1/MAX((ICARD(column1 index), ICARD(column2 index)) if indexes on both columns.
F=1/ICARD(column-x index) if index on column-x.
F=1/10 for all other cases.

column > value F=(high key value - value) / (high key value - low key value) if column is arithmetic and the value is known at access path selection.
F=1/3 for all other cases.

column IN (value list) F=(number of items in list) · (selectivity factor for column = value).
Maximum allowed value is 1/2.

The selectivity factors can be used to calculate the query cardinality, which is the product of the relation cardinalities multiplied by the product of all selectivity factors in the query block's predicate list. The estimated number of RSI calls, the CPU cost can then be calculated as the product of all relation cardinalities times

the selectivity factor of all sargable boolean factors. These selectivities together with statistics on available access paths are used to find the optimal access path for a single relation. In System R, this cost model is expanded and generalized to handle n-way joins and nested queries by combining the cost of scans on each of the relations and cardinalities.

Query Plan Selection

According to Chaudhuri [19], desirable features in a query optimizer are a search space that includes low-cost query plans, a cost estimation technique that is accurate to assign a cost to each of the plans in the search space, and an efficient enumeration algorithm. An important part of query plan selection is limiting or reducing the search space [19], which can be done by using heuristics such as grouping k-table joins and redefining edges of the joins are often applied in order to limit the size of the search space and avoid exhaustive enumeration [20]. These are "rules" that can be used to eliminate certain query plans, and can for instance be to use index scans whenever possible or to prefer merge joins if the input is already sorted [14]. The reduction of search space enhances the importance of an accurate cost estimation algorithm, as it will be the determining factor of how favorable the selected plan is. One aspect that can help on the cost estimation of the query plan is the available statistics. Certain operations can be done in various ways, and their efficiency will be determined by what data is in the relations. Consequently, statistics that provide information on this data will benefit the cost estimation of the algorithms.

Query trees are mainly represented either as a *left-deep*, *right-deep*, or *bushy* tree. In a left-deep tree, the child to the right of a non-leaf node is a base relation, while right-deep trees are the same, but opposite. Bushy trees have no such restrictions and contain both of the above [12]. Selinger et al. [1] limited the search space of the query optimizer to only consider left-deep trees due to (1) the number of left-deep trees is smaller than for example bushy trees, and (2) left deep trees generate only one intermediate result [21]. Figure 2.2 shows a left deep (left), right-deep (center), and bushy (right) query tree.

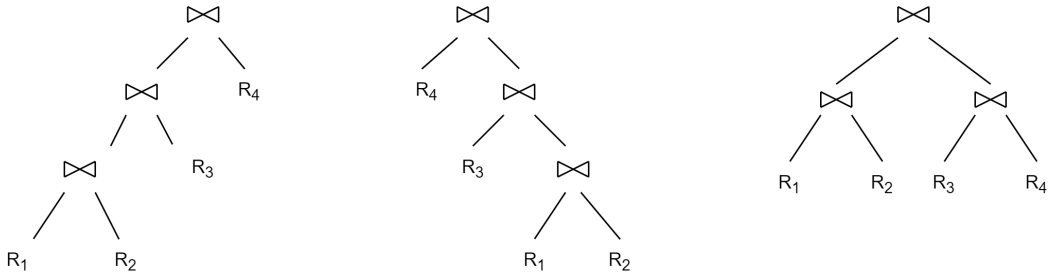


Figure 2.2: Left-deep (left), right-deep (center) and bushy (right) query trees.

When evaluating the plan space, there are two main approaches: top-down and bottom-up. Using the top-down approach, the optimizer starts at the top and works itself down the tree, considering the best option at each step, while the bottom-up approach starts at the bottom and works up the tree. In theory, both strategies can evaluate the entirety of the plan space, but the bottom-up approach is often used in commercial database systems as it is better suited to pipelining [12] due to the fact that each node can be evaluated separately. While bottom-up optimizers use dynamic programming by breaking problems down into simpler sub-problems [22], top-down optimizers use a technique called *memoization*. This is the equivalent of dynamic programming for top-down approaches [23] and is used to produce the best join tree by recursively calling itself for every subset of the set of relation that it is considering [21].

Cardinality Estimation

Cardinality estimates play an important part in determining the query plan selected by the optimizer and are the estimated number of rows returned from an operation. The estimates influence the order in which the query optimizer chooses to execute join-, and access methods, and also the type of methods used. For instance, a low cardinality estimate could lead to the optimizer choosing to perform a nested loop join when joining two tables, while a high cardinality estimate may lead to a hash join being chosen instead. If this cardinality estimate is wrong, the join operation could take longer than initially expected and other query plans could prove to be more efficient. As pointed out in [24] errors multiply through joins. This means that if we join three relations R_1, R_2, R_3, and the cardinality estimates for each of the relations are off by a factor of 5, the total cardinality estimate for $R_1 \bowtie R_2 \bowtie R_3$ will be off by a factor of 125 [21]. Cardinality is estimated in database systems due to the complexity of trying to calculate them exactly. This is usually done with the help of single-column statistics - like histograms - or certain assumptions in regards to the distribution of data, like uniformity, independence, or inclusion [25].

The Query Optimizer in MySQL

In 2007, Hellerstein et al. stated that the MySQL optimizer was entirely heuristic-based, and mostly relied on exploiting key/foreign key constraints and indexes [17]. This statement is not entirely true today, as the MySQL optimizer has been utilizing a combination of cost metrics and heuristics for query plan selection for many years now. However, the optimizer still relies too heavily on heuristics during important decisions and is not optimally rigged for the future, according to Ryeng in his presentation *"Refactoring Query Processing in MySQL"* [10]. An example

is that the optimizer currently performs an exhaustive search on plan selection for joins up to 7 tables, while additional tables in the join leads to greedy search based on heuristics. Some other notes worth mentioning is that the optimizer does not track interesting orders and only supports left-deep trees [26]. In MySQL version 8.0 there was a refactor of the entire query processing pipeline, with an exception of the query optimizer that was left mostly untouched. The optimizer has done a good job this far but struggles with more complex queries [10], which led to the development of the **hypergraph optimizer**.

The hypergraph optimizer is built to rely on a better defined cost-based query plan selection, making it resemble the optimizer from System R [1] more closely. It is, as the name implies, based on expressing join relations as edges of a hypergraph, defined as [27]:

Definition 1. *A hypergraph is a pair H = (V, E) such that*

1. *V is a non-empty set of vertices, and*
2. *E is a set of hyperedges between the vertices, where each hypergraph is a set of vertices* $E \subseteq \{\{u, v, \dots\} \in 2^V\}$.

Figure 2.3 illustrates the difference between a regular graph and a hypergraph, where the edges of a regular graph connect exactly two vertices, while a hyperedge connects two or more vertices. In the figure, a hyperedge is represented as an ellipse and denoted e_n. The vertices of a hypergraph represent the relations of a query, while the hyperedges represent a join operation. When having a traditional query graph, two edges are used to represent the join predicate $P_{XY} \wedge P_{XZ}$ and reordering is used to have a query that is still equivalent. With hypergraphs, predicates joining more than two relations are never broken up but represented with one single hyperedge [28].

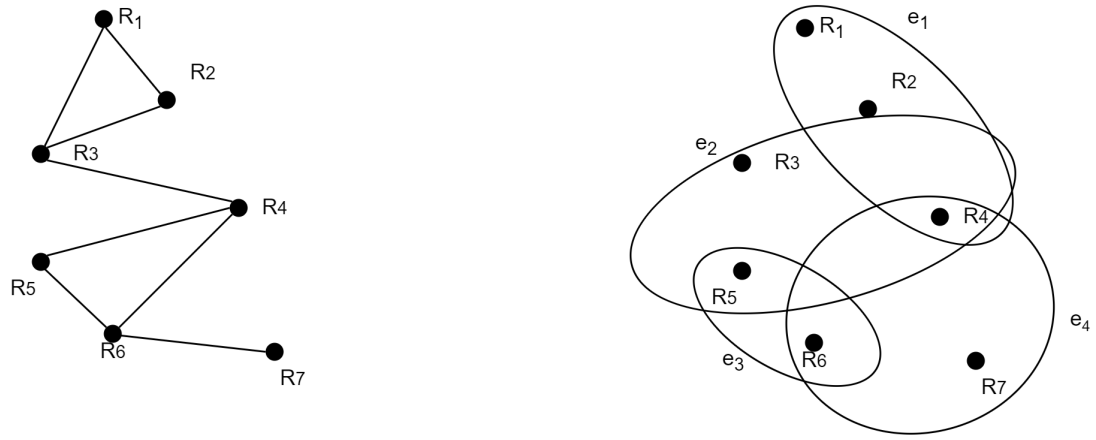


Figure 2.3: The figure shows the difference between a regular graph (left) and a hypergraph (right). The edges (line) of a regular graph connects exactly two vertices, where a hyperedge (ellipse) connects two or more vertices. Figure taken from [29].

While it can be said that System R utilizes multiple dimensions for the model - cost and interesting orders - the hypergraph optimizer takes advantage of more dimensions when estimating the cost, providing a more generalized model [30]. This allows the optimizer to store multiple plans concerning different cost-metric dimensions, and also opens up the possibility to discard query plans that are dominated by others in all dimensions. This essentially creates a skyline of the lowest-cost query plans for each dimension.

The goal for the hypergraph optimizer is to eventually replace the old optimizer. This replacement will ensure that MySQL is better equipped for handling more complex queries and future development. However, according to the source code documentation [31], the hypergraph optimizer is currently still in an experimental stage, with a simplistic cost model and other limitations.

2.2 Statistics

A cost-based optimization model as mentioned in Section 2.1.1 is used in the majority of relational database management systems when choosing a query plan [32]. This model makes decisions based on cost estimates of the various sub-plans of the query plan, which again are reliant on the cardinality estimates. As pointed out in Section 2.1.1, good optimizer statistics and statistics on base tables are needed for precise cardinality estimates.

The most commonly stored attributes for a relation in an RDBMS include the number of tuples (r), the average size of the records (R), how many blocks the relation occupies on disk (b), and finally, the blocking factor (bfr) which is the number of tuples in a block [14]. For the attributes within a relation, the RDBMSs often store the number of distinct values (NDV) and the selectivity (sl). An attributes selectivity is defined as the number of tuples that satisfy a certain equality condition and allows estimation of the selection cardinality ($s = sl * r$). This is an estimate of the number of tuples that satisfy an equality condition on the given attribute [14].

Databases commonly store information on the distribution of data within a column using histograms [32]. Histograms split the attribute over ranges called *buckets* and store the number of tuples that belong in a bucket alongside the NDV of a bucket [14]. Traditionally, the assumption of query optimizers has been a uniform distribution and independence of data [21]. This could lead to substantial errors in estimation if the dataset is skewed. Assume a relation storing information on employees within a company where one attribute is the gender of the employee stored as a `CHAR` (i.e., M/F). The optimizer would assume that the gender column would have a possibility of 256 distinct values, while the real number is 2.

Several different types of histograms can be used in a database system, the most common being *equi-width*, *equi-depth*, and *frequency* histograms [33]. Figure 2.4 illustrates a comparison of the different histogram types. Usage of equi-width histograms in databases was first introduced in [34] by Kooi, although their usage in the field of statistics dates back decades [35]. In equi-width histograms, all buckets are of equal size, and the approximation of the frequency of attributes is done through the height of the bucket.

Equi-depth histograms were in [36] recommended by Piatetsky-Shapiro and Connell and populates its buckets with approximately the same number of values. The buckets are thereby of equal depth and an approximation of a value is found by dividing the population of the bucket by the number of attribute values.

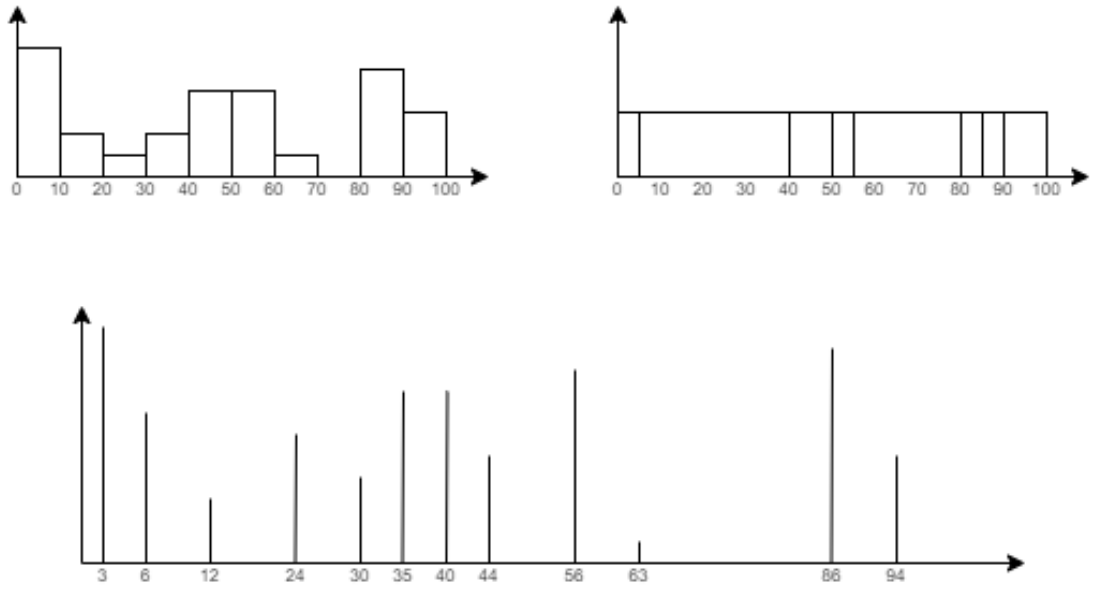


Figure 2.4: Equi-width (top-left), equi-depth (top-right) and frequency (bottom) histograms.

Frequency histograms keep track of the frequency of each distinct value by placing it in its own bucket. This requires the frequency histogram to have an equal amount of buckets, or more, than the number of distinct values [37], and allows for a recreation of the value distribution that is more precise than equi-width and equi-height histograms. A variation on the frequency histogram is called the *top frequency histogram* and allows having a number of buckets that is lower than the number of distinct values. This is done by ignoring values that are nonpopular or insignificant, utilizing the statement from [38] that accurate data on the most common values makes a significant contribution to selectivity estimates.

2.2.1 Statistics in Modern Database Systems

This section presents the state-of-the-art when it comes to statistics in four different database systems: MySQL, PostgreSQL, Microsoft SQL Server and Oracle Database.

MySQL

In MySQL, histograms are created using the `ANALYZE TABLE` statement along with `UPDATE HISTOGRAM` and are stored in the *information_schema.column_statistics* table [39]. The default number of buckets for a histogram is 100 if not otherwise

specified. If the number of distinct values (NDVs) is less than the number of buckets, the histogram is created as a frequency histogram, otherwise equi-depth histograms are chosen [7]. MySQL store histograms in the JSON [1] format, and also keep track of information such as the sampling rate used to create the histogram and the fraction of null values in the column.

PostgreSQL

PostgreSQL has, in addition to regular single-column statistics, what they call extended statistics. While the single-column statistics have no knowledge of any correlations across columns, the extended statistics objects are able to capture such correlations [40]. Creating the extended statistics object does not compute multivariate statistics itself, but it merely tells the PostgreSQL server to gather statistics on interesting columns.

Functional dependencies are the simplest kind of correlation that is tracked by extended statistics, where a column y is dependent on a column x if knowing the value of x is enough to say what the value of y will be [41]. Two columns that have a functional dependency are for instance the columns *zipcode* and *city*, where one always is able to tell what city belongs to a certain zip code. Another statistic that is stored by extended statistics is multivariate N-distinct counts. Here, the statistics object tells the server to keep track of the number of distinct values across N columns [42] Finally, extended statistics can store multivariate most-common value (MCV) lists. MCVs are often stored by regular statistics on a column level, but extended statistics can keep track of such lists across several columns [42]. Table 2.2 shows an example of a most-common value list of city and state and reveals that Washington and DC is the most common combination. It also shows with the base frequency, that if the value had been computed using single-column frequencies, it would be underestimated by two orders of magnitude.

[1] JavaScript Object Notation

Table 2.2: Example multivariate most-common value (MCV) list stored in PostgreSQL extended statistics. The table implies that the MCV of city and state is Washington and District of Columbia with a frequency of 0.003467, while the base frequency which is computed from single-column frequencies, underestimates by two orders of magnitude. Table adapted from [40].

index	values	frequency	base_frequency
0	{Washington, DC}	0.003467	2.7e-05
1	{Apo, AE}	0.003067	1.9e-05
2	{Houston, TX}	0.002167	0.000133
3	{New York, NY}	0.001967	0.000114

Microsoft SQL Server

In Microsoft's SQL server, the statistics that are used for query optimization is stored in binary large objects (BLOBs). The BLOBs are created on one or several columns of a table and contain information about the distribution of data, alongside a histogram. The histogram displays the distribution of the first column of the BLOB. If the BLOB is created on multiple columns, it also includes information on the correlation of values between the columns by calculating the density ($1/NDV$) [43]. Histograms linked to a BLOB are created either through a sampling or a full scan of all rows of the table in question. This is done by sorting the values and then aggregating them into a max of 200 histogram steps.

The Microsoft SQL Server the flags `AUTO_CREATE_STATISTICS` and `AUTO_UPDATE_STATISTICS` to create or update statistics respectively, and both are set to ON by default [44]. Automatic creation of statistics happens for columns for which there are no current histograms in any existing BLOB, and are created strictly single-column. The recompilation threshold used for updating statistics is presented in Table 2.3, and shows the number of modifications (insert, delete, update, merge) needed before an update is triggered.

Table 2.3: Microsoft SQL Server recompilation threshold by table type and table cardinality. Adapted from [43].

Table type	Table cardinality (n)	Recompilation threshold (# modifications)
Temporary	n<6	6
Temporary	6<=n<=500	500
Permanent	n<=500	500
Temporary or permanent	n>500	MIN(500 + (0.20 * n), SQRT(1000 * n))

Oracle Database

The Oracle Database collects statistics on the number of rows, the number of blocks, and the average length of a row for a table, along with the number of distinct values and the number of nulls for a column. For columns, the distribution of data is also stored as a histogram, and Oracle DB chooses between four different types of histograms: Frequency histograms, equi-depth histograms, hybrid histograms, and top n frequency histograms [45]. An illustration of the decision-making process in Oracle Database is shown in Figure 2.5.

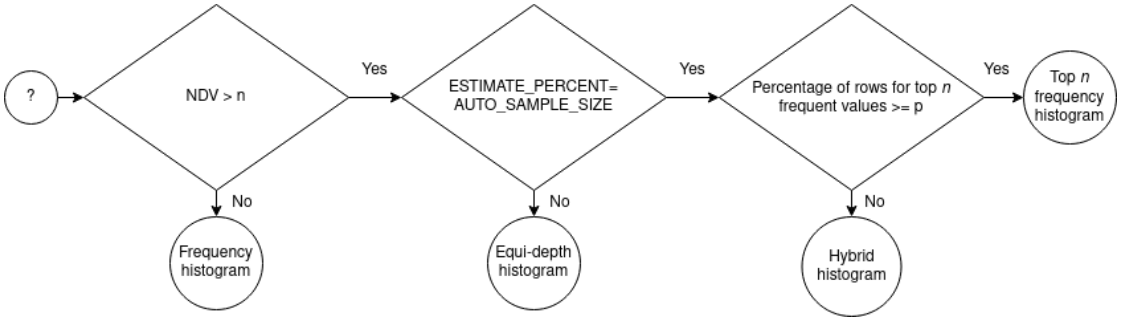


Figure 2.5: Figure illustrating the decision tree used by Oracle DB in order to choose what histogram to create. Adapted from [46].

The initial step in the decision-making tree is to check whether NDV is larger than the number of buckets specified for the histogram (default 256). If the number of distinct values is less than the number of buckets, a frequency histogram can be constructed. The next check is on whether the parameter *estimate_percent* is changed from the default value of equalling `AUTO_SAMPLE_SIZE`, and if it is, an equi-depth (here called height-balanced) histogram is chosen. The final step includes checking whether the percentage of rows for the top n most frequent values is greater than or equal to what is called the *internal percentage threshold*. This threshold, in the Figure named p, is defined as $(1 - (1/n)) * 100$ where n is the number of buckets in the histogram. A top n frequency histogram is created if the check returns true, otherwise, a hybrid histogram is chosen [46].

2.3 Data Stream Processing

A data stream can be defined as a sequence of instances - a continuous flow of data - that can be read only once or a small number of times [47]. Typical examples of systems producing such data streams can be wireless networks, radio frequency identification (RFID), or multimedia data services [48]. Although a traditional RDBMS is not explicitly a streaming system, it will still produce data streams of some sort, for instance during a table scan or when presenting the user with the desired results. This is also the case for MySQL, where records read through an internal iterator can be seen as a stream of data. However, an important difference to note is that these data streams are not infinite/continuous in the same manner as ordinary data streams, as there is a defined end to the data stored in the database. Processing and utilizing the information in this stream is not something previously done in MySQL but might prove beneficial for the management of database statistics. During processing of a data stream, there are multiple challenges to consider. Generally speaking, the system does not have control over the arrival order of the elements in a data stream or across data streams [47]. In addition to this, a data stream may be unbounded in size. The final size of the data stream can be impossible to predict, and in some instances, there is no end to the data stream as new data is continuously produced, typically the case with sensors. Another challenge is that once an element has been processed, it is usually archived or discarded. This makes re-accessing the element quite problematic unless it is stored for instance in memory. As the size of the data stream can be rather large, storing all elements in memory is often not a possible approach.

One of the major differences between traditional processing and data stream processing involves processing time. In traditional processing, the data is stored as data in raw form, then processed at a later stage as shown in Figure 2.6. An example of this would be the creation of histograms in DBMSs. The data is inserted into the database and stored before a request to create the histograms is made. The processing time for creating histograms does not then have any constraints and can in theory be unlimited.

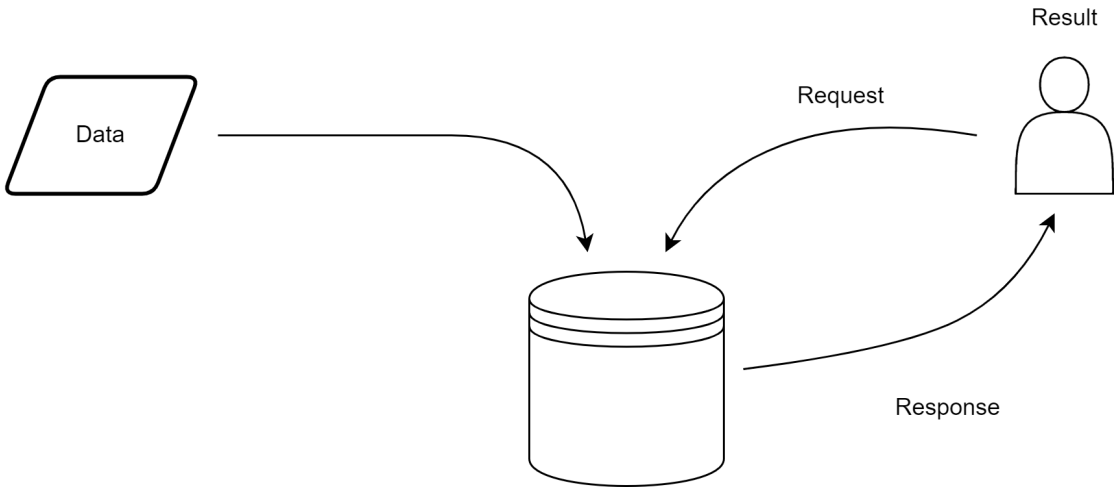


Figure 2.6: Example of traditional processing, where data is stored, and a request needs to be made in order to receive the processed result.

Data stream processing on the other hand needs the data needs to be processed before storage. This makes the processing time-limited, as the stream is a continuous flow of data that needs to be handled when it comes in. A representation of data stream processing can be seen in Figure 2.7.

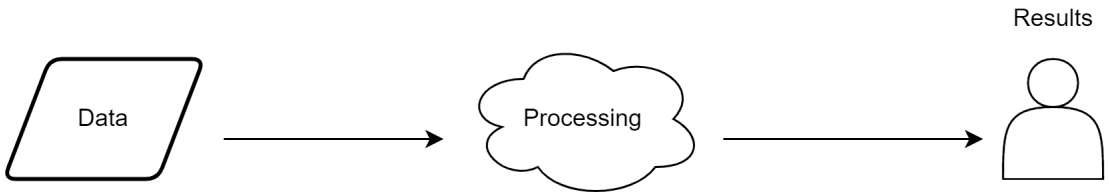


Figure 2.7: Example of stream processing where the data is analyzed before it is stored.

2.3.1 Compact Summary Data Structures

An important aspect of data stream processing is being able to utilize the relevant information retrieved from the data stream. To access the information at a later stage, the data must be stored. It is possible to store all of the data from the stream, but this might quickly become too large to handle and is not necessarily scalable. Compression is a way of solving this but since most compression algorithms serves as an archival space-saving method, generally using or querying the

information at a later point requires decompression. A compact data structure aims to solve this problem, maintaining the data in a form that uses less space, but also allows for querying the data in this form without decompression [49]. These are essentially methods for efficiently representing the data from the data stream while still being able to navigate and operate on the data.

Summary data structures is a term that has seen more usage in recent years, due to the handling of increasingly larger data sets [50]. While a compact data structure, in general, will aim to cover all aspects of the data, a summary data structure will need to consider which parts of the data should be conserved and how accurate this preservation needs to be. However, they both share the trait of representing the data in a smaller manner, while still being able to query and operate on the structure. While traditional lossless compression can be decompressed into the original data, a summary data structure does not come with any guarantees of such reconstruction [50]. Some examples of summary data structures are *samples*, *sketches*, *histograms*, and *wavelets* [8]. The different structures will have various advantages and disadvantages compared to each other, based on dimensions such as accuracy, space efficiency, performance, or maintenance. For instance, samples have the advantage of being flexible and simple, making them applicable in many use-cases. However, samples may in some cases have high error bounds, and will not be optimal for raw count problems, such as estimating the number of distinct values [50]. Sketches on the other hand are also known for being flexible, while allowing for a variety of operations, including distinct counts, dot products, and matrix computations. Some of the drawbacks of the sketches have been that they may become large and slow to update, and supports a limited set of operations on a single type of sketch.

2.4 Hash Functions

Hash functions are used to map elements from a large domain D to a smaller domain m [51]. In practice, this is often done by storing the hash value $h(x)$ of element x at location $h(x)$ in an array m. From good hash functions operations can be expected to perform in $O(1)$ time, and are defined as follows [52].

$$h : x \rightarrow (ax + b \; mod \; p) \; mod \; R \tag{2.1}$$

where p is a prime, $a \neq 0$, and $b \in \mathbf{Z}_p$. It is desirable for the hashed values $h(x)$ to have a "random behaviour" and be distributed uniformly across m, and in order to obtain this, k-independent hashing was introduced by Wegman and Carter in [53]. A family of hash functions is k-wise independent if a randomly selected function can guarantee that the hash-values $h(x)$ from any of the k keys are random and independent variables [2].

Wegman and Carter states that a family of hash functions $\mathbf{H} = \{h : U \rightarrow [m]\}$ is k-wise independent in the case of $\forall a_1, \ldots, a_k \in U^k$ and distinct $x_1, \ldots, x_k \in [m]^k$, if

$$\Pr_{h \in H}[h(a_1) = x_1 \; \wedge \; \ldots \; \wedge h(a_k) = x_k] = m^{-k} \tag{2.2}$$

This definition states that for any distinct keys $a_1, \ldots, a_k \in U$, and randomly drawn hash function h from H the hashed values $h(x)$ are both independent random variables, and uniformly distributed in $[m]$.

[2] Random variables are independent if the realization of one does not affect the probability distribution of the others [54].

2.5 Count-Min Sketch

The count-min sketch data structure was proposed by Cormode and Muthukrishnan in [55] with the intention of being used to approximately summarize a stream of data. It is essentially a two-dimensional array that takes two parameters (ϵ, δ) that determines the width $w = \lceil \frac{e}{\epsilon} \rceil$ and depth $d = \lceil ln \frac{1}{\delta} \rceil$. The array is filled with counters that are initially set to zero. In addition, d number of pairwise independent hash functions are chosen randomly. Items inserted into the sketch are hashed and mapped to a cell in each of the rows in the two-dimensional array, and the counter of that cell is incremented [56]. This procedure is shown in Figure 2.8.

To estimate items using the count-min sketch, one simply needs to hash the item in question to then look at the counter of the given cell. As the matching counters of the d number of rows may differ in value, the lowest is chosen as the estimate to minimize the error due to hash collisions.

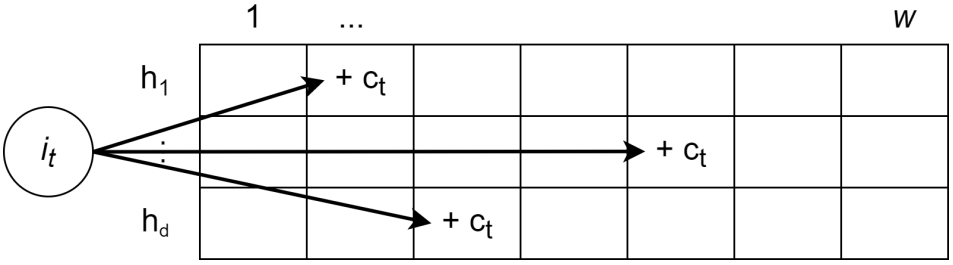


Figure 2.8: Every item i_t is mapped to exactly one cell in each row. When an update is called for the item, the specified cells are incremented by c_t. Figure adapted from [57].

Count-min sketches can be utilized to answer point queries, with an approach similar to the one updating the sketch. A point query on value i can be estimated by finding $min_{1 \leq j \leq d}\ CM[j, h_j(i)]$ [58]. Due to the fact that every i is mapped to exactly one cell in each row, these counters will be an approximation of the result. It is expected that collisions might occur when the sketches are updated as they are not exact representations, therefore the row with the smallest value for the specified cell is selected as the answer. This is an approximation and might therefore not be the actual answer. However, the count-min sketch comes with some guarantees regarding the accuracy of the estimate: with a probability of 1-δ the error is not larger than the actual answer plus $\epsilon \cdot ||a||_1$. In other words, if we define the actual answer as a_i with vector a, the estimated answer est_i will be defined as following, with a probability of 1-δ:

$$est_i \leq a_i + \epsilon ||a||_1$$

Inner product queries can be answered with the count-min sketch, by estimating the join size between relations $a \cdot b$ [57]. This estimation is achieved by using the count-min sketch matrix as a collection of d vectors with a length of w, and calculating the dot-product between corresponding rows for the two sketches. The minimum value of the dot-products for all rows is then selected as the join size estimation. Utilizing the row with the smallest value has the same intention as other estimates, which is to reduce the impact of possible collisions. An important note is that this requires the count-min sketches for the two relations to be of equal depth and width, and also that corresponding rows use the same hash functions. The estimation for inner product also comes with some error guarantees. For relations $\mathbf{a} \cdot \mathbf{b}$ with the actual join size $\mathbf{js_{a \cdot b}}$, the estimation will be as follows with a probability of $1 - \delta$:

$$est_{a \cdot b} \leq js_{a \cdot b} + \epsilon ||a||_1 ||b||_1$$

2.5.1 Count-Min Can Do More

In 2007, Deng and Rafiei built upon the existing count-min sketch structure and proposed new algorithms for estimation of multiplicity-queries (point-queries) and self-join queries, known as the count-min mean *(CMM)* [59]. The original estimation techniques would lead to inaccuracies on less skewed data, a problem aiming to be solved with the new algorithms. The paper found that the new estimation algorithms significantly improved the estimation accuracy compared to the original ones when the data was less skewed.

The original algorithm for estimating point queries finds each cell that item i hashes to for every row in the sketch, and returns the minimum value as the estimate. Since these are sketches, the counter is usually affected by other elements as well due to collisions in the hashing. In other words, elements other than i might also hash to the same cell in the sketch, characterized as *noise* by Deng and Rafiei [59]. The CMM algorithm aims to calculate the noise for each counter, remove the noise and then return the median value of d counters. Finding the exact value of noise is generally not feasible, but the expected value can be estimated. The estimated noise for each counter can be calculated as the average value of all the other counters in the row except the counter itself. If we define the total number of elements (stream size) as N, the noise n can be estimated as follows: $\frac{1}{w-1}(N - CM[j, h_j(i)])$. The noise is found and subtracted for all d counters, and lastly returning the median value of the subtracted counters as the estimate for the point query.

The new algorithm for solving inner product queries follows a similar approach, by utilizing noise estimation. Originally, the estimate was calculated as the sum of all dot-products for each row and then returning the minimum value found. With the CMM approach, the noise for each counter is subtracted before multiplying

with the equivalent counter in the other sketch. For each row, the product of the sum and *(w-1)/w* is calculated, and the median value of d rows is returned as the final estimate. The noise is estimated in a similar manner to the method for point-queries, by finding the average value of all counters in the row except the actual counter itself. If we reuse the previous definition of noise, we can define noise for a given counter on any sketch x as follows: $noise_{x(j,k)} = \frac{1}{w-1}(N - CM_x[j,k])$. Using this definition, the inner product query estimation algorithm is as follows:

$$\frac{w-1}{w}\sum_{k=0}^{w-1}(CM_a[j,k] - noise_{a(j,k)}) \cdot (CM_b[j,k] - noise_{b(j,k)})$$

Chapter 3

Architecture/Model

This chapter presents how the implementation of the experiment is done in MySQL. First, Section 3.1 gives a thorough explanation of how the correct per-predicate selectivities are injected into the hypergraph optimizer. Further, Section 3.2 shows how the count-min and count-min mean datatypes are implemented, how they utilize the existing data streams to approximate the distribution of data, and how they are used in order to estimate the selectivity of a predicate.

3.1 Injection of Correct Selectivity

During startup of the MySQL server, all the predicates from the JOB data set and their selectivities are read from a CSV file and stored in the Table struct. This is done using the file *selectivity_reader.cc*, reading the data from the CSV file, creating a vector of tuples with their respective selectivity and handing this over to the Table struct.

The hypergraph optimizer utilizes the EstimateSelectivity method from *estimate_selectivity.cc* when deciding the selectivities for a given query. Our implementation latches on to this method and forces the pre-calculated correct selectivity to be returned for each predicate or condition expression. To preserve original functionality, an optimizer switch flag is checked, deciding if the method should return the injected selectivities or the original estimates. Finding the correct selectivity is done by matching the predicate expression with the corresponding tuple from the Table struct, and returning the selectivity from this tuple.

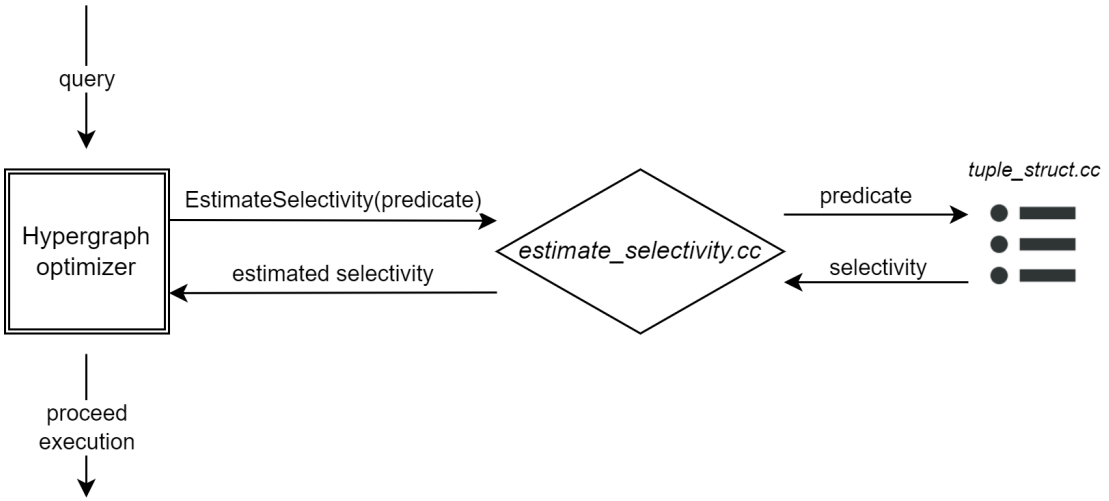


Figure 3.1: Architectural sketch of the selectivity injection. The predicate is sent to *tuple_struct.cc*, returning the respective selectivity.

Figure 3.1 presents a simplified overview of how the selectivity injection fits in the current system. For a given predicate, the hypergraph optimizer makes use of *estimate_selectivity.cc* to find the selectivity. If the optimizer switch flag `OPTIMIZER_SWITCH_JOB_SELECTIVITIES` is enabled, the predicate is sent further to *tuple_struct.cc*, taking advantage of the Table struct to retrieve the actual selectivity.

3.1.1 tuple_struct.cc

The file *tuple_struct.cc* is created to define the Structures *Table* and *Row* used to manage the correct selectivities for the JOB data set. The Table struct keeps track of a vector of Row structs, essentially keeping track of a vector of tuples. Each tuple (Row) contains four elements. The first one is the left side of the predicate. The next one is the operator, such as IN, '=' or LIKE. The third element is the right side of the predicate, and the last element is the correct selectivity for the predicate.

The method *GetSelectivityForCondition* is important for the usage of the Table struct, and takes the arguments Item and string. The class Item is an existing MySQL base class that can represent any kind of expression in a relational query [60], and the string is essentially used for handling optional output from the optimizer trace. The Item argument contains the condition or predicate and is therefore utilized to find the correct selectivity. The condition is compared with the stored Rows in the Table structure, and given a match the respective selectivity from the Rows last element is returned. If no match is found, a selectivity of -1.0 is returned. In theory, this should not occur for any of the JOB-queries, as all of the predicates along with their selectivity should be loaded on startup.

3.1.2 selectivity_reader.cc

Reading the selectivities from the CSV file is performed with the help of *selectivity_reader.cc* and its method *GetSelectivitiesFromFile*. The method takes the file path as an argument in the form of a string, and parses the file line by line, removing unwanted characters such as the column separators. For each line, the elements are placed in their correct order in the tuple and added to a vector of tuples. Lastly this vector of tuples is returned.

The current implementation utilizes the read-method within the existing *init.cc*-file that is run during boot-up. The selectivities are first read using the selectivity_reader, and then given to the Table struct which populates the Rows. In theory, these operations could be performed in other parts of the code structure as long as it happens once before the JOB queries are run. Init.cc was chosen for simplicity as it was guaranteed to happen only once at boot-up of the server.

3.2 Count-Min Sketch Implementation

The count-min sketch implementation is based on the theoretical description presented in section 2.5. The sketch is intended to be used as a replacement for histogram statistics, providing the optimizer with selectivity estimations for predicates in the query. The current implementation is able to estimate selectivity for multiplicity queries, join size relations, and IN-predicates.

There are several reasons for the choice of implementing a count-min sketch as a representation of the statistics. For one, as described in section 2.3.1, sketches are known for being flexible and allow for a wide variety of operations such as distinct counts, dot-products, and matrix computations, making them very suitable for selectivity estimation. Additionally, the count-min sketch addresses many of the drawbacks of a regular sketch such as becoming too expensive in space and update processing, making them more fast and suitable for database statistics. The count-min sketch can also be implemented in MySQL in a relatively simple manner, providing selectivities directly to the optimizer without altering too many other parts.

3.2.1 count_min_sketch.cc

The file *count_min_sketch.cc* defines the class CountMinSketch and takes the two parameters epsilon ϵ and delta δ. From these values, the width w and depth d of the sketch are calculated as per the definitions from Section 2.5. It also includes a counter which keeps a count of the total number of inserts into the sketch. Finally, a two-dimensional array with the size of $w \times d$ is initialized alongside d pairwise independent hashes.

The *Update* method is used to insert elements into the sketch, accepting either an int or a string as the argument. When inserting a string into the sketch, a hash value (int) is generated using the djb2 hash function [61]. This value is then hashed to a cell C[h(x)] for each of the d rows using Formula 2.1, and the counter for those specific cells is incremented.

The *Estimate* method works similarly to the update and takes either a string or an int as an argument. The initial process is equal to an update, hashing the string and hashing the int value to a cell for each of the rows. The difference is in the latter part, where instead of incrementing the specified counters, the lowest counter is returned as the estimate for the value.

3.2.2 Using the Count-Min Sketch

To get an estimation of the contents of each column of a table, a count-min sketch object is created for each of the columns and put in a map. The key of the map

is the table name and the column name such that the individual sketches can be easily retrieved. Inserts into the sketches happen in iterators where a reading of records already happens such as the `TableScanIterator`.

When using the count-min sketch implementation to estimate the selectivity of a predicate the method first checks whether the optimizer flag `OPTIMIZER_SWITCH_AUTO_STATISTICS` is set. It then uses the table- and column name to get the correct sketch(es) for the predicate. The value(s) of the predicate are retrieved from the sketches and used to calculate the estimated selectivity. If the optimizer flag is not set, estimation proceeds as normal.

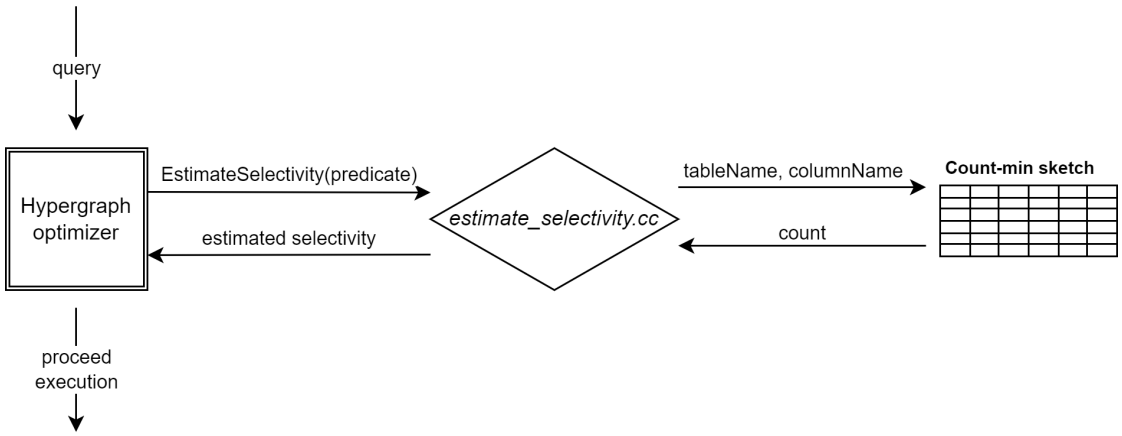


Figure 3.2: The figure visualizes the selectivity estimation process when using the count-min sketch. After the hypergraph optimizer receives a query it asks for the estimated selectivity of each predicate. The method in *esitmate_selectivity.cc* that is responsible for selectivity estimations finds the correct count-min sketch which returns a count of the value that is to be estimated. Furthermore, the selectivity estimation method calculates the selectivity based on the estimated count of the value, and returns this to the hypergraph optimizer.

An approximate visualization of the estimation process using sketches is shown in Figure 3.2, where the hypergraph optimizer receives a query. For each predicate in the query it asks for an estimated selectivity from a method in *estimate_selectivity.cc*. This method finds the correct count-min sketch which returns the estimated count of the value in question. The estimated selectivity is calculated and returned to the hypergraph optimizer, which is able to proceed in its execution.

3.2.3 Using the Count-Min Mean Sketch

The count-min mean sketch is implemented in the same manner as the count-min sketch, storing the sketches for each column in a map with table name and column name as the key. Inserts also happen correspondingly as the CM-sketches, taking advantage of already present record reading. The important difference between the two types of sketches is how the value sent from the sketch is handled. The algorithm for estimating point-queries and join size relations are different for count-min mean sketches. This implementation uses the original CM-algorithm for point-queries, but the new method for estimates on join size relations. As explained in subsection 2.5.1, this is done by removing the estimated noise in each counter before multiplying with the equivalent counter for the other sketch. This is performed in *estimate_selectivity.cc*, where the value is returned and the estimated selectivity is calculated according to the specified algorithm.

Chapter 4

Experiments and Results

This chapter presents the experiment, with Section 4.1 showcasing what question each step of the experiment aims to answer. In Section 4.2, the setup of the experiment is explained in order to make it reproducible by other researchers. This includes the hardware used and the settings of MySQL. Finally, Section 4.3 presents the most interesting and relevant results, alongside a discussion on what these results imply.

4.1 Experimental Plan

The plan for the experiments is divided and aims to answer several questions. One of the goals is to pinpoint how well the hypergraph optimizer performs with optimal statistics injected. This serves as a stepping stone for following tests, exploring the usage of count-min sketches for selectivity estimation. These experiments aim to cover the feasibility of automatic creation of such statistics, where the sketches are created by listening to the existing data stream from a table scan. Additionally, the sketches serve as an experiment on the efficiency and accuracy of an alternate method for storing statistics other than histograms.

4.1.1 Injecting Correct Selectivity

The first part of the experiment includes injecting the correct values where the optimizer asks for estimates of selectivity on predicates from the *Join Order Benchmark*[1] queries. This experiment is done with the intent of answering **Hypothesis 1**, determining whether the performance of the hypergraph optimizer improves when it has the "optimal" information available. To have a reasonable comparison, the same tests are also done on the standard hypergraph optimizer without statistics available, and the hypergraph optimizer with histograms on all relevant columns.

One could argue that injection of correct cardinalities should be performed instead, but there are some reasons for the choice of using selectivities. First and foremost, the injection of selectivities is a simpler approach that fits more easily into the existing MySQL code base. Additionally, this injection can be performed without altering other aspects of the optimizer, which essentially means providing better statistics to the optimizer without directly changing any other parts of decision-making or behaviour. The optimizer can then utilize its cost model and functions to determine query cardinalities.

4.1.2 Count-Min Sketch for Selectivity Estimation

The next part of the experiment is an implementation of a data stream summary algorithm, the count-min sketch, and utilization of these sketches to estimate the selectivity for predicates in the JOB-queries. The sketches are implemented in a manner so that they can answer point queries (also collections of them, such as IN-queries) and inner product queries. The sketches "highjack" the data stream of a table scan for all of the tables in the JOB-dataset, making this part of the experiment descend from **Hypothesis 2**.

[1]Join Order Benchmark dataset is found at *https://github.com/gregrahn/join-order-benchmark*.

Several executions of the JOB-queries are run with different sizes and accuracy requirements for the CM sketches. This is done with the intention of being able to pinpoint the optimal accuracy required for the optimizer, by possibly discovering breakpoints in the query plan selection, and be able to find a suitable trade-off between performance, accuracy, and space requirement for the sketches.

4.1.3 Count-Min Mean for Selectivity Estimation

The count-min mean (CMM) experiment is similar to the count-min, being a direct result of **Hypothesis 2**, whereas the difference is the CMMs algorithm for estimating join size relations. The CMM sketches are implemented in the same manner as count-min, meaning they can answer identical types of queries. The experiment will test the effectiveness of CMM sketches compared to CM sketches, and discover changes in accuracy while still requiring an equal amount of memory as previously.

4.1.4 Selectivity Estimation Error

As mentioned in Section 2.1.1, MySQL's hypergraph optimizer uses a more cost-based optimization technique than the current MySQL optimizer which partly depends on heuristics. This, along with the fact that the hypergraph optimizer is still under development, reveals the need to evaluate both the count-min and count-min mean sketches in more dimensions than just execution time. The most prominent evaluation of whether the sketch implementations can improve the performance of the hypergraph optimizer, and thus give a definitive answer to **Hypothesis 2**, is to examine the error of the selectivity estimates.

4.1.5 Memory Usage

In order to determine whether the implemented sketches are usable in practice, their performance needs to be evaluated in context with the space they require in memory. No matter how accurate it is, for a statistics structure to be feasible in a real-life database system, the size requirement of the structure can not be of such a size that it may impact the performance of the rest of the system.

4.2 Experimental Setup

All experiments are done in MySQL version 8.0.28 built from source with the hypergraph optimizer enabled. To enable the hypergraph optimizer in a production build, the CMake option `-DWITH_HYPERGRAPH_OPTIMIZER=1` has to be provided. In each session, the `SET optimizer_switch="hypergraph_optimizer=on"` -option has to be set, as the hypergraph optimizer is not the default optimizer. Table 4.1 shows the hardware setup used for all tests.

Table 4.1: Specifications for the computer used in the experiment

Hardware	Specification
OS	Ubuntu 21.10
CPU	Intel i7-8700
Memory	48 GB DDR4 2400 MHz
Disk	512 GB Samsung PM981 Polaris M2 NVME SSD

The experiments in this project utilize the *Join Order Benchmark* introduced in [62]. These queries are based on the Internet Movie Data Base (IMDB) dataset and have been proven to be more suitable than other benchmarks such as TCP-H for evaluating cardinality estimates. This is because a real-world dataset such as the one from IMDB is non-uniform and has correlations in the data.

For all experiments, the JOB-queries are run six times in succession. The first execution is to "warm-up" the database and ensure that parts of the dataset are in memory. The final five executions are timed and the average is calculated and reported as query execution times.

4.2.1 Injecting Correct Selectivity

The structure presented in Section 3.1 keeps track of the selectivities of all predicates in the JOB dataset to be able to feed the optimizer with the correct selectivities when asked for. For the selectivities to be read on startup and injected, the optimizer flag `OPTIMIZER_SWITCH_JOB_SELECTIVITIES` must be set. As the population of the structures defined in *tuple_struct.cc* is done on server startup, the overhead when injecting the selectivities is approximately the same as the method MySQL uses and thereby negligible in terms of results.

4.2.2 Count-Min Sketch for Selectivity Estimation

The tests for the count-min sketch implementation are performed in the same manner as the tests for selectivity injection, but with the optimizer flag

`OPTIMIZER_SWITCH_AUTO_STATISTICS` set. To simulate that the MySQL instance has been running and sketches have been populated, a table scan of each relation in the JOB dataset is performed prior to the tests being started.

The tests are executed in three separate runs with a variety of parameters epsilon (ϵ) and delta (δ) for the count-min sketch. The parameters alongside a calculation of the width w and depth d are shown in Table 4.2. To change the parameters of the sketches, the values need to be changed where the count-min sketch object for each column is created in *basic_row_iterators.cc* and MySQL need to be rebuilt.

Table 4.2: Table showing experimental parameters epsilon (ϵ) and delta (δ), alongside the width w and depth d for the count-min sketch.

ϵ	δ	$w = \lceil \frac{e}{\epsilon} \rceil$	$d = \lceil ln\frac{1}{\delta} \rceil$
0.000001	0.001	2718282	7
0.0001	0.01	27183	5
0.001	0.01	2719	5

4.2.3 Count-Min Mean for Selectivity Estimation

Preliminary testing highlighted that the original algorithm for estimating point queries had sufficient accuracy, with near-perfect selectivity estimation in many cases. The CMM-estimation was therefore only implemented on inner-product queries, causing our CMM-implementation to be a hybrid of both. The setup is similar to the CM experiment, whereas the major key difference is the estimation algorithm used for selectivity estimation of inner-product queries. The experiment is run twice, with different values for ϵ but with a static value of 0.01 for δ. Table 4.3 displays the parameters used for the CMM sketch implementations.

Table 4.3: Experimental parameters ϵ and δ alongside width and depth for the count-min mean sketch.

ϵ	δ	$w = \lceil \frac{e}{\epsilon} \rceil$	$d = \lceil ln\frac{1}{\delta} \rceil$
0.0001	0.01	27183	5
0.001	0.01	2719	5

4.2.4 Selectivity Estimation Error

In order to get per-predicate selectivity estimates for JOB queries, MySQL needs to be built with sketches of the wanted parameters. In this experiment, the same parameters as presented in Figures 4.2 and 4.3 were used for count-min and count-min mean sketches respectively. Selectivity estimates can be read from

the optimizer trace of the hypergraph optimizer. Before running a JOB query, the statement `SET OPTIMIZER_TRACE="enabled=on"` needs to be executed in order to enable optimizer tracing. The selected query can then be run, before running `SELECT * FROM information_schema.OPTIMIZER_TRACE` to extract the trace from the query.

4.2.5 Memory Usage

By running the Linux `htop`-command before and after a table scan, the memory used by the MySQL instance in an idle state with no sketches, and after sketches are created can be found. Thus, one can calculate the memory used by the sketch structures.

Furthermore, the theoretical memory usage is calculated in order to determine the best-case space requirement. For simplicity, theoretical memory usage is calculated by looking solely at the size of the counter-matrices, where each cell requires 4 bytes of space for storing the counter. Final size calculations are summarized for sketches on all 103 columns. Defining N as the total number of columns, the formula for calculating theoretical memory usage is therefore:

$$\lceil e/\epsilon \rceil \cdot \lceil ln(1/\delta) \rceil \cdot 4 \cdot N$$

4.3 Experimental Results

This section displays the most notable results from the experiments conducted. The full set of results can be found in appendices A - C, additionally all raw data produced from the experiments are found in appendices D - G. In Section 4.3.1, we look at how the hypergraph optimizer behaves when it has different types of information present, both histograms and correct selectivity, in order to determine the best possible return for having statistics available. Section 4.3.2 investigates how count-min sketches of various sizes compare to histograms. In Section 4.3.3 histograms and the most optimal count-min sketch are compared to count-min mean sketches of two different sizes to determine whether there are major differences in terms of performance. These three sections concerns executions time exclusively. As mentioned previously, the hypergraph optimizer is in a state of development, which suggests that it does not necessarily make the optimal choice even though it is presented with more precise estimates. We will therefore present the estimation errors provided by the various structures in Section 4.3.4. Finally, in Section 4.3.5, the measured memory use of the data structures presented in this thesis is compared to their theoretical memory use.

4.3.1 Injecting Correct Selectivity

The figures presented in this section are an excerpt of interesting results from the experiment, highlighting queries where correct injected selectivity improves performance but also queries displaying the opposite effect. It also shows occurrences where having histograms on base columns deteriorates in performance compared to not having statistics available at all. Complete results are found in Appendix A.

Figure 4.1 shows a graph of the execution time of five queries from the Join Order Benchmark run without statistics, with histograms on relevant columns, and with correct selectivity injected. The graph shows that query 3a is slightly slower with histograms than it is without, but sees a significant boost in performance when the correct selectivities are injected. Query 2b has approximately the same execution time without statistics and with correct selectivities but performs substantially worse with histograms. The rest of the queries (19c, 17c, and 9d) see a slight improvement with histograms and correct selectivity over no statistics, although there seems to not be much gain from having the correct selectivities over the ones calculated from histograms in these queries.

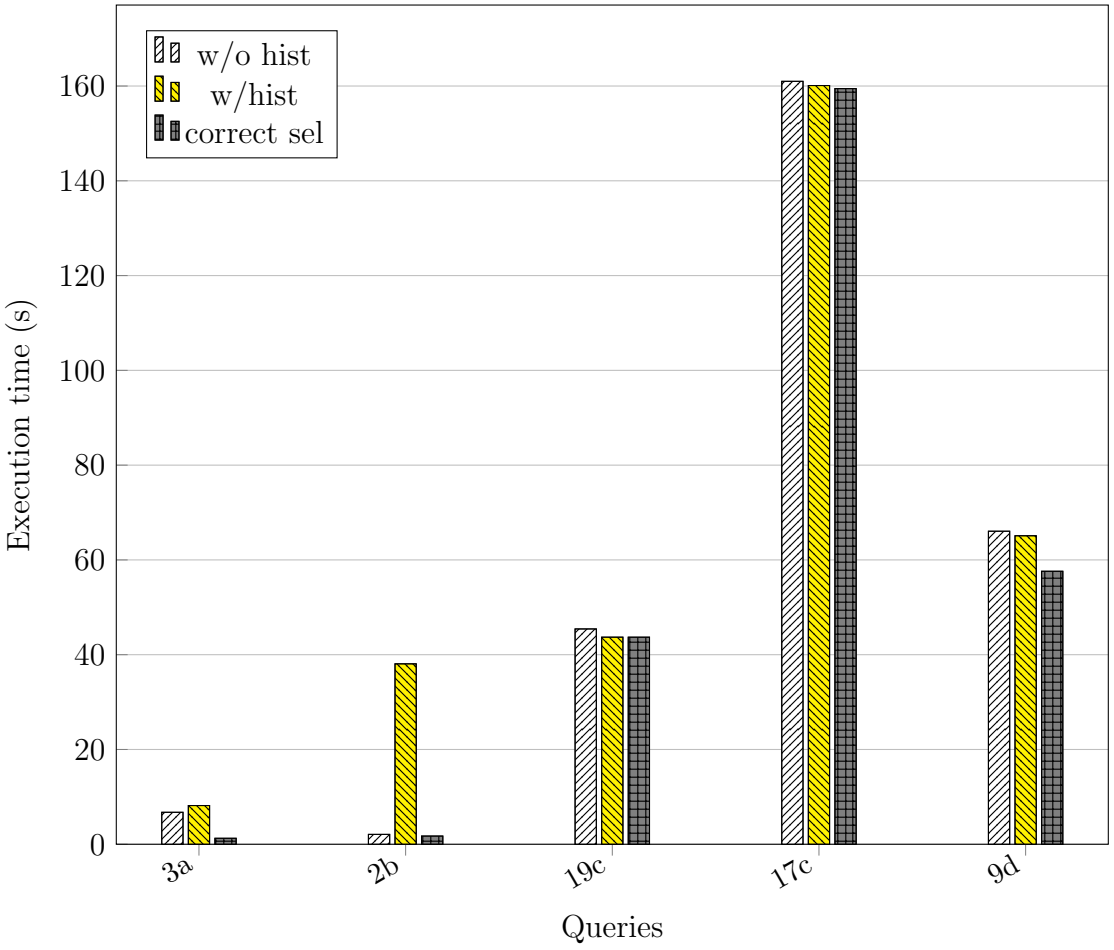


Figure 4.1: Execution times for a selection of queries displaying the difference between no statistics, histograms, and correct selectivity. Lower is better.

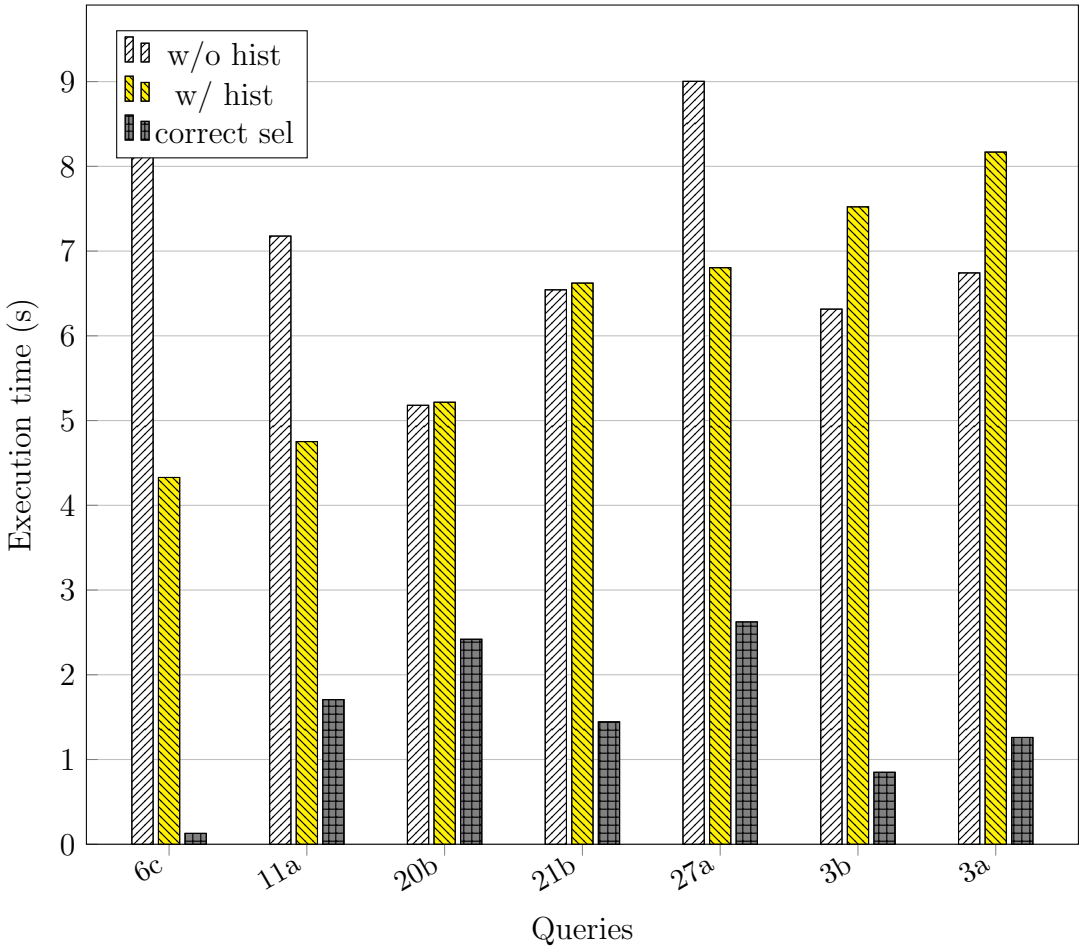


Figure 4.2: Execution times for a selection of queries displaying difference between no statistics, histograms, and correct selectivity. Lower is better.

Figure 4.2 shows queries for which injection of correct selectivity improves the execution time significantly. Queries 6c, 11a, and 27a show enhanced results with histograms but even more so with correct selectivity, while queries 20b, 21b, 3b, and 3a perform slightly worse with histograms but still improve with correct selectivities. Results from the experiment show that execution times when injecting the correct selectivity into the hypergraph optimizer are lower than both histograms, and also without statistics. There are certain exceptions from this case where the execution time drastically increases, which can be seen in Figure 4.3.

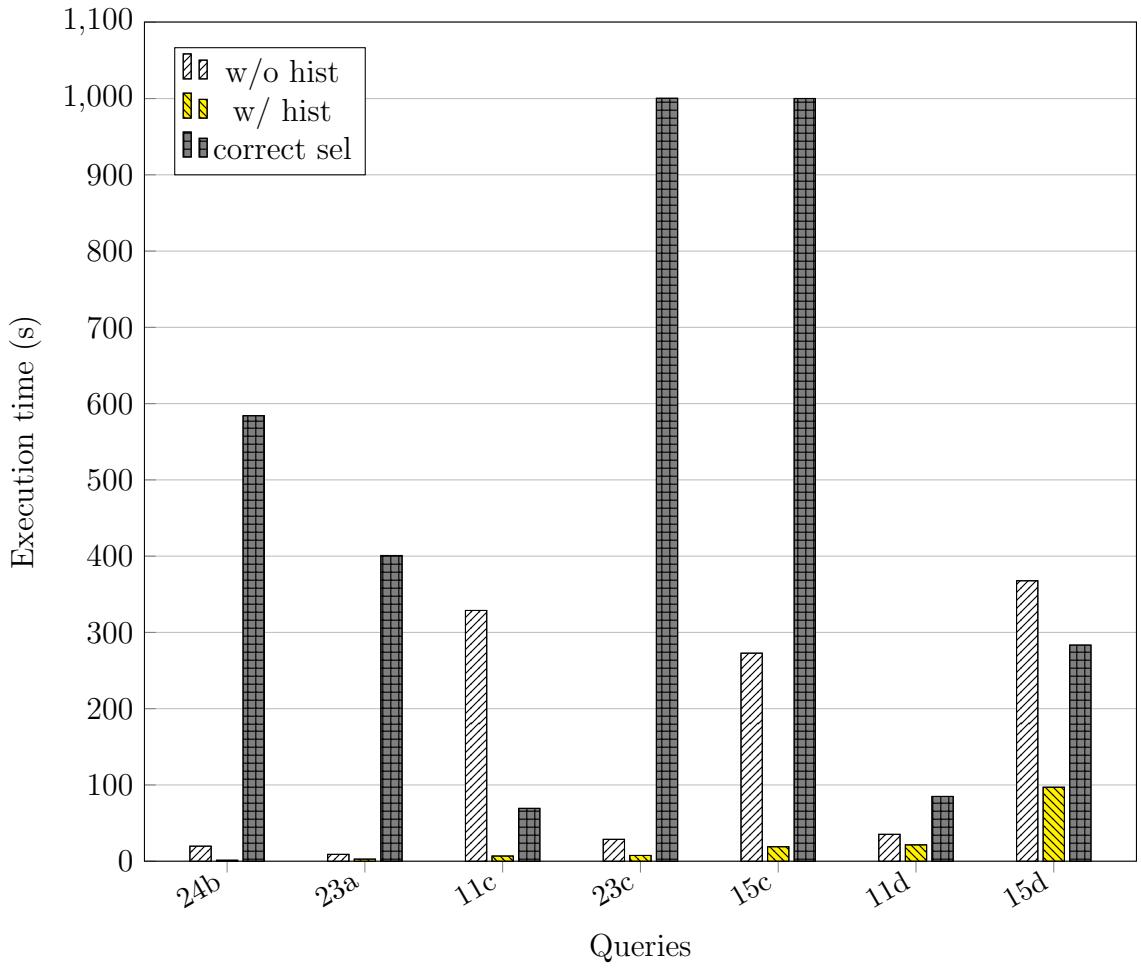


Figure 4.3: Queries where the execution times become significantly worse when correct selectivity is injected.

It is stated in Section 1.1 that the hypergraph optimizer chooses query plans considerably more cost-based than the current MySQL optimizer. The selected results from injecting the correct selectivities for the Join Order Benchmark queries show that in its current state, this needs not always be the case. There are also cases where the hypergraph optimizer chooses the wrong plan when using histogram statistics, and not when having the correct selectivities. The results show that if histogram statistics perform better than no statistics for a given query, then correct selectivities will most likely outperform histograms. This indicates that the hypergraph optimizer more often than not chooses better plans when having more accurate information available, confirming **Hypothesis 1**.

4.3.2 Count-Min Sketch for Selectivity Estimation

This section presents interesting results of the count-min sketch implementation, comparing the execution time with histogram statistics. Figure 4.4 presents the results of running the same queries as in Figure 4.1 with an implementation of the count-min sketch and is an excerpt of the full results which can be found in Appendix B. The columns represent different sizes of count-min sketches, and the results from the count-min sketch implementation are shown in comparison to the standard histograms. Queries 3a and 19c do not see any improvements in execution time for sketches of any size, and query 19c becomes severely slower. Query 2b improves with the introduction of sketches over histograms, but Figure 4.1 shows that histograms made the query perform worse than without any statistics. Improvements in execution time when using count-min sketches are shown for queries 17c and 9d, and the largest gain comes from the count-min sketch with $\epsilon = 0.0001$ and $\delta = 0.01$.

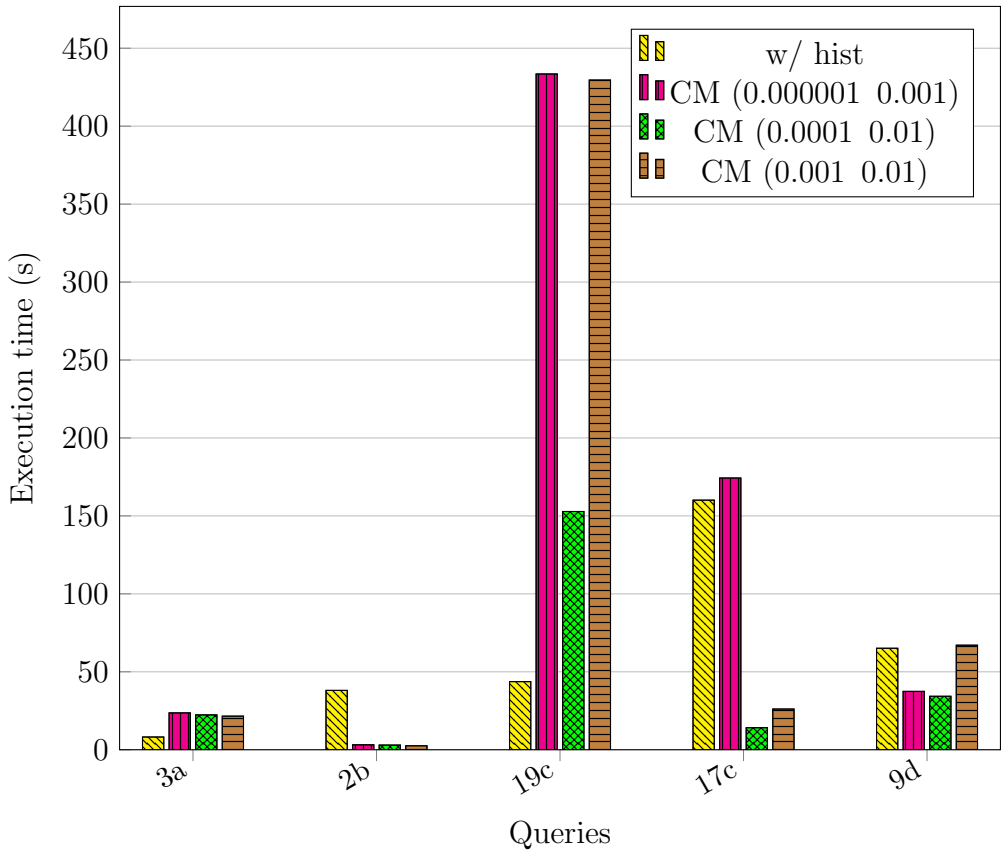


Figure 4.4: JOB query execution times for hypergraph optimizer with histograms on base tables and hypergraph optimizer with count-min sketches of various sizes. Lower is better.

The experiment shows a variation of results between the queries when using count-min sketches over histograms. The sketch size that gives the overall best results is the same as for queries 17c and 9d as mentioned previously with $\epsilon = 0.0001$ and $\delta = 0.01$. On certain queries such as 19c, 5c, 9a, and 8d one or more of the count-min sketches suddenly make the hypergraph optimizer choose a much slower plan even though it is presented with more accurate estimates. This might be due to the current simplistic cost model of the hypergraph optimizer. Even so, the cause of the abnormal plan selection should be evaluated in context with the selectivity estimates, and thus is further discussed in Section 4.3.4.

The results indicate that the theoretically more precise sketch in fact does not yield the best results. Lower values for ϵ and δ create a bigger sketch, making hash collisions for the hashed values less likely and giving more accurate estimates. This increased theoretical accuracy does not necessarily improve performance thereafter. On the contrary, the trend of the results is that the count-min sketch with $\epsilon = 0.0001$ and $\delta = 0.01$ is the one that has the overall lowest execution time for JOB queries.

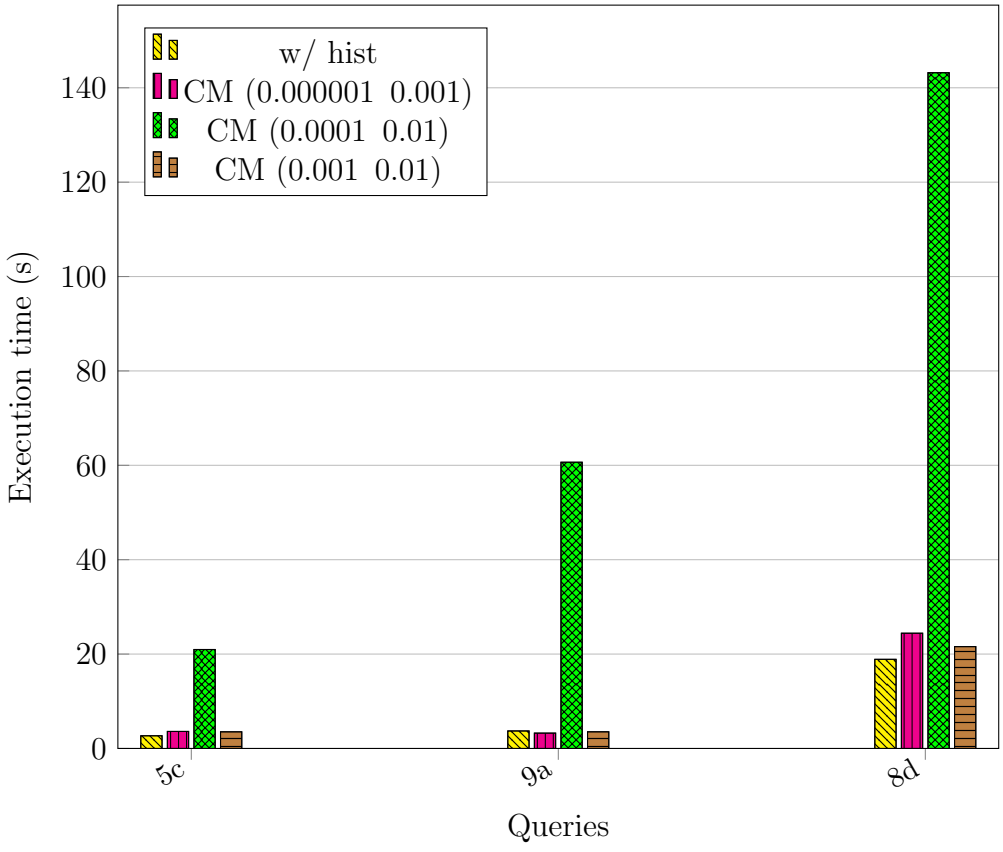


Figure 4.5: Queries where the count-min sketch with parameters $\epsilon = 0.0001$ and $\delta = 0.01$ suddenly spikes in execution time.

Similar to the injection of correct selectivity, the count-min sketches also show certain oddities in terms of execution times. Examples can be seen from the queries in Figure 4.5 where the execution time for these queries suddenly spike when using estimates from the sketch that shows the overall best results. These spikes in execution time need to be evaluated in the context of whether the selectivity estimates become better with larger sketches or not in order to confirm or disprove **Hypothesis 2**, as the hypergraph optimizer in theory should improve with better estimates.

4.3.3 Count-Min Mean for Selectivity Estimation

This section presents interesting results of the count-min mean sketch implementation, comparing the execution time with histogram statistics and one of the previous count-min sketches. The full results from the experiment can be found in Appendix C. Figure 4.6 displays the results for two types of count-min mean sketches on the same queries as Figures 4.1 and 4.4. In terms of execution time, there is a slight improvement with the count-min mean for queries 3a and 2b. For the rest of the selected queries, the result from using the count-min mean sketches shows no improvement over regular count-min sketches.

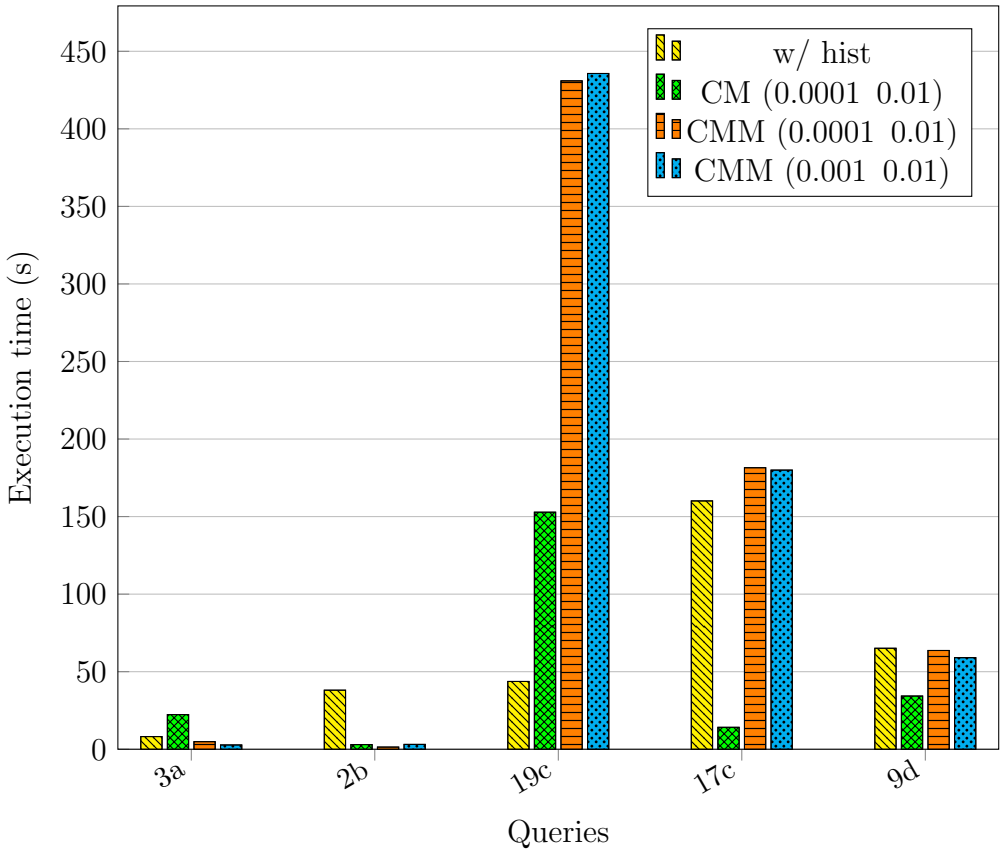


Figure 4.6: JOB query execution times comparison between histograms, count-min sketch and count-min mean sketch. Lower is better.

The results show that the smallest count-min mean sketch with $\epsilon = 0.001$ and $\delta = 0.01$ is the one that is overall closest to the most optimal count-min sketches in execution time. Initially, this seems somewhat counter-intuitive as one would

expect the larger sketch to have more accurate statistics, which in turn lead to improved performance. For most queries it is very similar or even slightly better, but in certain cases, the count-min mean sketches become severely slower than both regular count-min sketches and histograms. This topic is explored further in Section 4.3.4.

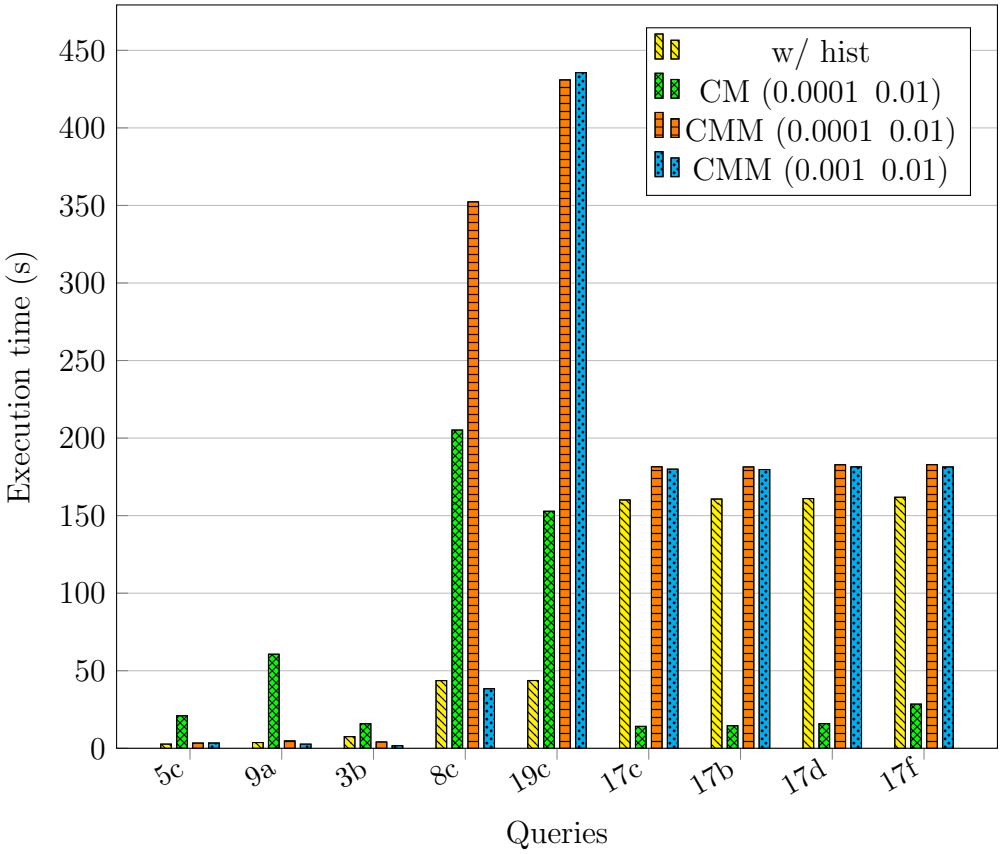


Figure 4.7: Count-min mean sketches outperforming regular count-min sketches for queries 5c, 9a and 3b, while being significantly worse on queries 8c, 19c, 17b-d, and 17f.

The smallest CMM-sketch, with ϵ value of 0.001 and δ value of 0.01, performs somewhat similarly to the two smallest CM-sketches, but there are still outliers in execution time both positively and negatively. Figure 4.7 shows that the CMM-sketches significantly outperforms the selected CM-sketch for query 5c, 9a, 3b, and 3a, while the opposite is shown for query 8c, 19c, 17b-d, and 17f. This initial testing shows promising results for the effectiveness of the CMM-sketch but opens up for a further investigation into the causes of the outliers and oddities present.

4.3.4 Selectivity Estimation Error

Figure 4.8 shows the cumulative relative error in selectivity estimation for the selected queries. The errors are calculated by summarizing the relative error for each predicate in a single query.

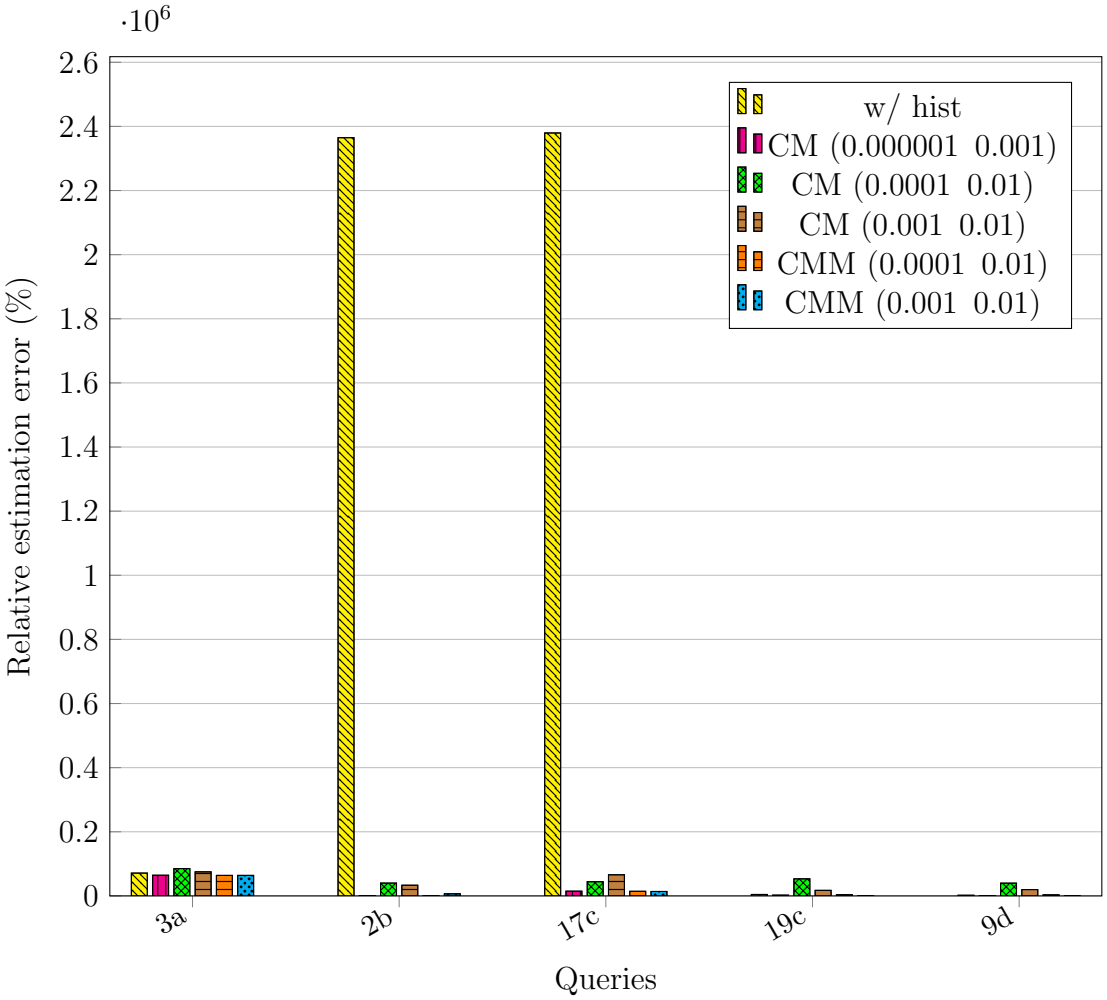


Figure 4.8: Per-query cumulative relative selectivity estimation error for histograms, count-min sketch and count-min mean sketch. Lower is better.

The histogram estimation error protrudes for queries 2b and 17c, with a summarized error several orders of magnitude larger than the sketches. The predicate *k.keyword = 'character-name-in-title'* is present in both of these queries and contributes to a huge portion of the error. For query 2b, the histogram estimates a

selectivity over 20 000 times larger than the actual selectivity for this predicate. A common denominator for several queries is that point predicates on text columns with a large number of distinct values will drastically affect the accuracy of the selectivity estimation of the histograms. Due to the size of the histogram errors, removing the histogram bar from the chart gives a better visual representation of the summarized error for the sketches:

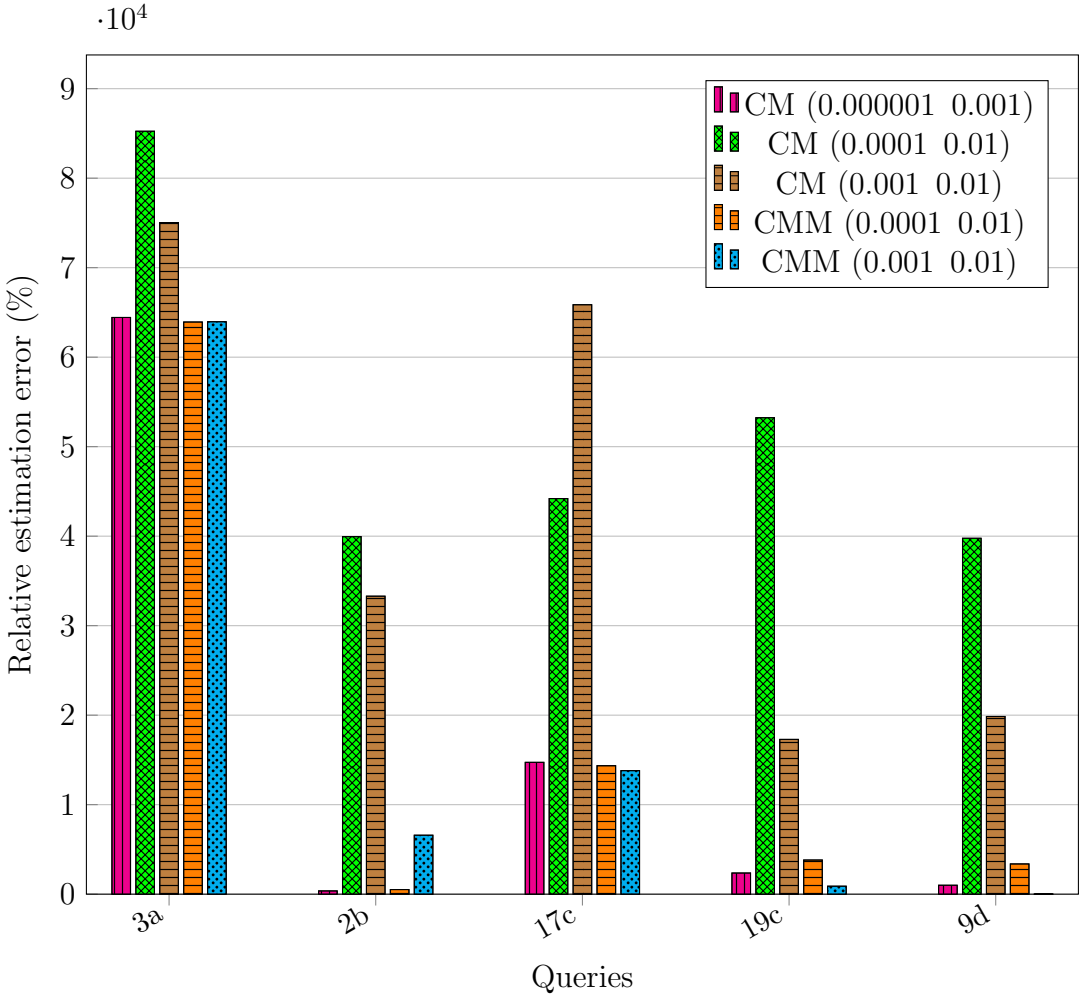


Figure 4.9: Per-query cumulative relative selectivity estimation error for count-min sketch and count-min mean sketch. Lower is better.

Figure 4.9 shows the cumulative selectivity estimation error for all of the sketches. As the graph displays, a larger sketch leads to more accurate estimations. Additionally, the CMM-sketches of equal size to the original count-min

sketches see a drastic improvement in accuracy. This is also the case for queries 19c, 17c, and 9d, despite performing slower than the selected count-min sketch as shown in Figure 4.6. This per-query cumulative estimation error serves as an initial overview, but is a simplification and will not highlight the extremes or other interesting finds. Figure 4.10, therefore displays the average estimation error for a selection of predicate type categories:

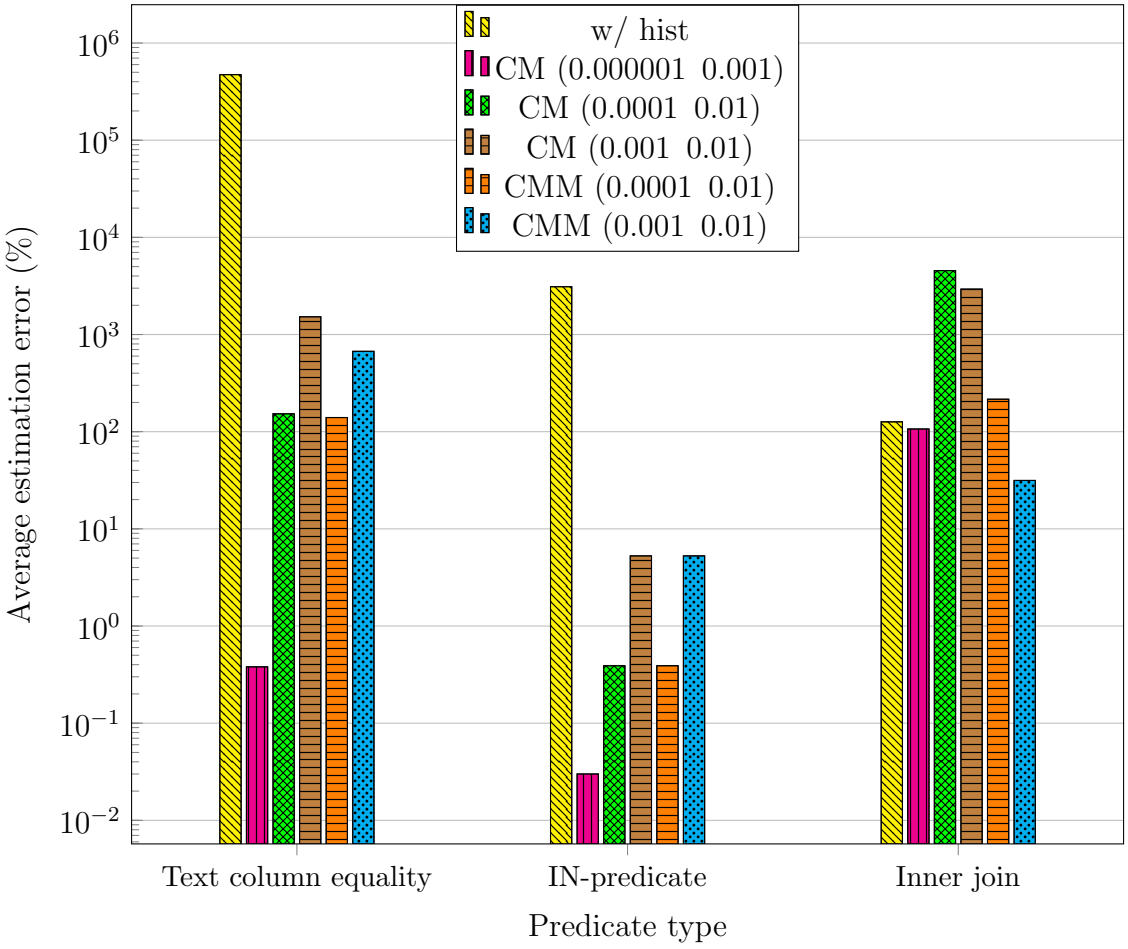


Figure 4.10: Average estimation error grouped by selected predicate categories. Histograms see a significantly larger average error than both of the sketch types for text column equality and IN-predicates.

Average estimation error by predicate type/category gives a more precise visual representation of the results. On inner join predicates, the histogram estimation error somewhat resembles the estimation errors of the sketches. Additionally, the

CMM sketches outperform their CM-sketch counterparts of equal size on inner join, with an average error estimation several orders of magnitude less. For IN-predicates and equality on text columns, all of the sketches have lower estimation errors than those of the histogram. It is also worth noting that for both of these categories, the CM- and CMM-sketches of the same size see a roughly equal estimation error.

The results show that for a selection of the queries, the per-query cumulative estimation error is significantly larger for histogram statistics than for the sketches. The CMM-sketches also have lower estimation error than the CM-sketches of equal size. Interestingly enough this does not guarantee better performance, an example that can be seen with for instance queries 19c and 17c, where the CMM sketches perform considerably worse than the CM sketch based on execution speed while still having better estimation accuracies. The hypergraph optimizer is still in a relatively early phase and under development, which may have an impact on the achieved results. Prior testing has also shown similar results happening [11]. Additionally, as highlighted in section 2.1.1 the cost model and functions for utilizing selectivity factors will play an important role. As an example, if the cost model is poorly adjusted for a certain type of query, improved statistics might not lead to a more accurate cost metric. Since the optimizer is still under development, there is reason to believe that the cost model and functions might still have some of these challenges.

Figure 4.10, displaying the average estimation error per-predicate type, shows the largest differences for text column equality and IN-predicate. For both of these categories, the histogram statistic leads to an estimation error several orders of magnitude larger than the error present with the sketches. The inner join predicates on the other hand, show a higher inaccuracy for the two smallest CM-sketches compared to both the other sketches and the histogram. Identifying categories where different types of statistics perform contrasting may be a promising start for the development of a hybrid model. The results may indicate that a combination of a CMM-sketch on columns usually queried by text column equality or as a part of IN-predicates and histogram statistics on columns often used for joins/inner joins such as ids can lead to accurate statistics while maintaining a better space-efficiency.

4.3.5 Memory Usage

Altering the parameters ϵ or δ will affect the accuracy and maximum error for estimations, but will also impact the disk space occupied by the sketches. This is because ϵ and δ are deciding factors for the width and depth of the sketch respectively. Approaches for calculating theoretical and measured memory usage are defined in Section 4.2.5. Table 4.4 highlights both the measured memory usage and theoretical memory usage for the earlier parameters used, for sketches on all the columns.

Table 4.4: Theoretical and measured memory usage for different parameters of ϵ and δ on JOB-dataset

ϵ	δ	**Theoretical memory usage (MB)**	**Measured memory usage (MB)**
0.000001	0.001	7839	7934.40
0.0001	0.01	56.00	141.70
0.001	0.01	5.601	93.80

Table 4.4 shows that the topmost row with the lowest δ and ϵ values most likely is not feasible in practice, as both theoretical and measured memory usage is almost 8 GB in size. Incorporating database statistics of these size proportions is probably unheard of, especially considering the size of the data set itself. Additionally, the increased estimation accuracy does not have a great impact on performance as seen by prior results, meaning that the trade-off between size and performance is not optimal in this case. The other two rows in the table highlight a more promising disk usage, with a static value of 0.01 for δ and three or four decimal place accuracy for ϵ having a measured memory usage of 93,80 and 141,70 MB, respectively. For both cases, the measured memory usage might be slightly large, but it is reasonable to believe that these numbers can be greatly reduced. First and foremost, the current implementation is a prototype with an experimental approach, without a focus on implementing the sketches in an optimal or space-efficient way. Improvements in the implementation should considerably reduce the gap between theoretical memory usage and measured memory usage. Later implementations should additionally include the opportunity of compression when stored outside of memory. Sketches were also created on all of the columns in the data set, which is generally not the case for real-world usage. In practice, such thorough column statistics are usually generated for a selection of interesting columns. Further testing may experiment with sketches on a limited amount of columns and the impact on accuracy compared to memory usage.

Chapter 5

Conclusion and Future Work

In this chapter, the experimental results are evaluated along with the experiment itself. Section 5.1 discuss what the results mean and the impact these have on the research, including limitations and other eventual drawbacks of the experiments. Conclusions are drawn in Section 5.2 based on the research questions presented in Chapter 1. Finally, in Section 5.3, we present work that can be done by future researchers extending this topic based on limitations of the experiment and/or experimental results.

5.1 Discussion

As seen from the results in Section 4.3 both the count-min and the count-min mean sketches show promising results when it comes to improving the information available to the hypergraph optimizer when making an informed and correct query plan selection. Results from the tests where the optimizer was injected with the correct selectivities for the Join Order Benchmark show that the hypergraph optimizer on a general basis seems to perform better with more precise estimates although some odd results occur. One could argue that multiple different benchmarks could have been utilized, but the Join Order Benchmark has been acknowledged as an improvement from previous industry-standard benchmarks, such as TPC-H, and deemed a good fit for evaluating query optimizers [62]. The sketches improve execution times for a majority of queries over histograms, but the biggest improvement is seen when looking at the selectivity estimates. As mentioned previously, the hypergraph optimizer may not always make better choices although it has more accurate statistics. This might occur due to several reasons, where one of them is since the optimizer still is in a state of development, impacting the cost model or other functions that determine behaviour and plan selection. This makes the results from the selectivity estimates of the sketches even more promising, as there is still room for improvements in execution time.

As each query is executed six times, with the first being a warm-up run, the execution times reported in the results are averages of the five runs preceding the warm-up. An average of several executions allow for better accuracy and lowers the impact of a "malfunctioning" run. One could argue that five runs are too low and that the impact such a malfunctioning run could have on the specific result still would be too high. Considering the time and resources available for the experiment, alongside the fact that the initial testing with more runs done in [11] showed little to no benefits to accuracy, five runs was a good trade-off between efficiency and preciseness.

More thorough checks on why the hypergraph optimizer chooses certain query plans over others when specific information is available to it could have been done in the experiment, to better determine why the aforementioned oddities in execution time occur. This was not strictly prioritized due to limitations in resources and the scope of the thesis. One could also argue for the possible redundancy of such investigation, as later versions of the hypergraph optimizer might not include such occurrences - at least not to the same degree.

The experiment as presented in Section 4.2 is simplified in comparison to what a full implementation of count-min or count-min mean sketches would look like and does not take into account how updates to the data would be handled. It is also designed to simulate a MySQL instance that has been running for a while to fully populate the sketches with all data in all tables, through the table scan

that is performed previous to the experiment itself. In a real-world scenario, the population of sketches would happen in the MySQL iterator classes - during table scans, join iterations, etc. - meaning that sketches may be populated with only parts of relations. Considering this, the experiment could have been performed with various fractions of the total amount of data in the sketches to compare usability. Additionally, since the sketches were only populated using a table scan, further exploration on exploiting other existing data streams would most likely require taking into account several other questions. If the data stream does not include all records, how would one handle requests on missing records? Is the record missing or has it just not yet been a part of the data stream? What if a duplicate stream comes along and the sketch is not marked as "full", will this lead to non-existing skew?

Research question 3 (1.2) looks at the alternatives to histograms with regard to storing statistics from data streams specifically. Several possible structures that could be used for data stream processing are discussed in Section 2.3, but only count-min and count-min mean sketches are implemented and tested against histograms in the experiment. This limits the foundation for answering **Research Question 3**, as several of the other structures remain untested in this use-case. Due to limitations in resources available, a selection of able and interesting structures had to be made. The reasoning for the selection of sketches was explained in detail in Section 2.5. Another option that was not explicitly explored in this thesis, was techniques allowing the histogram to be used to store data from a data stream, which showed promising results in [63].

5.2 Conclusions

The thesis aims explore the possibility of automatic statistic creation in MySQL with the intent of improving the performance of the hypergraph optimizer. The research questions defined in Section 1.2 are revisited and answered in order to accomplish the goal of the thesis.

Research question 1 *How does MySQL's hypergraph optimizer respond to having access to correct statistics?*

The effect of correct predicate statistics on MySQL's hypergraph optimizer is observable in the results from Section 4.3.1, where the correct selectivities for the JOB data set are injected. These results are overall promising in terms of performance, with some caveats to consider. For most of the queries, the execution time is lower with the injected correct statistics, but some queries do display the opposite, with an increasing execution time in the presence of the correct statistics.

The complete results highlight that the negative effect is limited to a selection of the queries, with the execution time of the remaining queries either being similar or improved compared to histograms or no statistics present. Additionally, as mentioned previously the hypergraph optimizer is still in an early phase of development [31], with a simplistic cost model and certain limitations that will affect the behaviour of the optimizer. It is therefore reasonable to expect later implementations to be even more consistent on correct statistics present.

To summarize, MySQL's hypergraph optimizer generally responds well to the presence of correct statistics - with some exceptions - but with an overall trend of improvements regarding execution times.

Research question 2 *How can existing data streams in MySQL be exploited to create statistics?*

Experimental results from Section 4.3.2 and 4.3.3 shows how statistics based on existing data streams can be used for selectivity estimation in queries. These experiments use a compact summary data structure in the form of a count-min or count-min mean sketch that is populated using the basic row iterator present in a table scan. For simplicity, these sketches were created by forcing an initial table scan, but these results also open up the possibility of later exploiting other existing data streams, such as those of a join iterator or a filter.

It is essential that the statistic can represent and/or utilize the data from the data stream in a sensible way that is helpful to the optimizer. Storing the entire

data stream gives a good representation of the data, but is inherently pointless as it requires double the storage, but storing too little information will on the other hand reduce the benefit of the statistic. These initial results highlight the usefulness of the CM and CMM sketches and provide promising results in regard to performance and space efficiency. Section 5.1 highlights some of the drawbacks of the simplistic implementations, and questions needing to be addressed when considering other data streams than the table scan. The sketches show promising results, but exploiting other types of data streams will require adjustments in the existing implementation.

Research question 3 *What are the alternatives to histograms when it comes to storing statistics from data streams?*

Some of the alternatives to storing statistics from data streams are discussed in Section 2.3 and include sketches, wavelets and samples. The CM and CMM sketches were selected for the experiments, based on their advantages and areas of use compared to the other alternatives, as mentioned in Section 3.2. These sketches provide space efficiency, fast performance, reasonable guarantees for error bounds and can be used to answer multiplicity queries (point queries) and estimate join size relations among other things. The experiments showed improvements in execution time for many of the queries in the JOB dataset, highlighting the possibility of the sketches as reasonable alternatives. As mentioned in Section 5.1, there has also been research on techniques allowing histograms to be used as a tool for storing data from a data stream. Wavelets might also be an interesting alternative to histograms in MySQL, where previous research in other systems has featured promising results [64; 65].

Research question 4 *How can a combination of sketches and histograms improve the performance of MySQL's hypergraph optimizer?*

The experimental results presented in Section 4.3 show that the count-min and count-min mean sketches are superior to histograms on point-predicate columns. This is displayed well when looking at the selectivity estimation errors in Section 4.3.4. Table 4.8 shows that for point predicates, histograms hold up just as well, if not better than sketches of various sizes. This, along with the fact that the sketches are somewhat large and occupy a big portion of memory when they are created for all columns leads us to conclude that a combination of sketches and histograms would be a prominent solution. Sketches could then be created for point-predicate

columns, and histograms would be used elsewhere. Not only would this be an improvement in terms of selectivity estimates, but would also give better space-to-performance efficiency than the use of sketches exclusively.

Testing a combination of sketches and histograms was not a part of the experiment conducted in this thesis due to time constraints, and will be further discussed in Section 5.3.

This thesis has presented a novel implementation of the count-min and count-min mean sketch data structures for estimating data distribution in MySQL. Comparisons with histograms on the Join Order Benchmark have shown promising results in performance, and a vast improvement in selectivity estimation accuracy. This increased accuracy is notably prominent for equality predicates on text columns, with an average estimation error several orders of magnitude lower than histograms.

Based on the experiments, utilizing the existing data streams in MySQL emerges as a viable solution in terms of automatic statistics management. As the hypergraph optimizer is more reliant on up-to-date statistics due to its more cost-based approach, it is theoretically better suited to make use of such solutions.

5.3 Future Work

In this section, we present directions that could be taken when conducting further research on the topic of *exploring the possibility of automatic statistic creation in MySQL with the intent of improving performance of the hypergraph optimizer*. As the focus of this thesis has been to utilize the existing data streams in MySQL to generate statistics automatically, we will both suggest work that takes a similar approach, but also varying directions that have the same ultimate goal.

The obvious suggestion based on the setup of the experiment is to do a full implementation of count-min or count-min mean sketches in MySQL that are populated in the iterators. Each time a record passes through the iterator, the sketch data structure is updated to keep the statistics as up-to-date as possible. This also requires handling of updates and deletes of records. As per the discussion in Section 5.1, the experiment conducted in this thesis is simplified and designed to simulate a MySQL instance that has been running for a while with a sketch implementation that has been fully populated.

Another way to go in terms of extending the work that is done in this thesis is to examine other data structures used for data stream processing and compare results with the results from Section 4.3 and Appendices B & C. Some data structures that are fit for data stream processing are mentioned in Section 2.3, but there are also techniques such as the ones presented in [63] that allow histograms to be used to store data from a data stream.

As mentioned in Section 5.2 and in line with **Research Question 4**, a combination of sketches and histograms should be implemented and tested to determine whether this has any effect on selectivity estimates. Such a solution would result in the sketches handling point-query estimations while histograms taking the rest. Further exploration on what types predicates the sketches have a major advantage over histograms, if any, could also lead to improving this implementation.

Based on the results and the hypergraph optimizer's current state one should try to pinpoint why the optimizer fails to improve when having accurate selectivities on certain queries. We can see that it has a considerable improvement on JOB query 11a (Table 4.2) with injected selectivity but becomes significantly worse on 11c (Table 4.3). Groups of Join Order Benchmark queries (i.e. 11a - 11d) are similar in structure and only certain `WHERE`-clauses are replaced from one to another. The results show several occurrences where queries within a group respond differently to correct selectivity, opening up the possibility to further determine *when* this happens and *why*.

Another direction for future work is to examine whether the optimizer can learn from queries that are executed in order to incrementally adjust its selectivity estimates. Currently, if there are no statistics or indices available for a predicate, the fallback selectivity used in MySQL's `estimate_selectivity.cc` is calculated

based on the number of rows in the table, which often leads to an overestimation. It would be imaginable that adjusting this number for certain predicates could be possible in order for the MySQL hypergraph optimizer instance to become more and more accurate for each query that is executed. A similar approach is taken when developing the Neo optimizer [66], even though the traditional optimizer is replaced with a deep neural network. We believe that a "learning" effect can be achieved with the hypergraph optimizer too if an adjustment of the estimated selectivity happened after the selectivity is returned.

Bibliography

[1] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access path selection in a relational database management system," in *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '79. New York, NY, USA: Association for Computing Machinery, 1979, p. 23–34. [Online]. Available: https://doi.org/10.1145/582095.582099

[2] Cisco Systems Inc. (2020) Cisco annual internet report (2018–2023) white paper. [Online]. Available: https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html

[3] S. Chakkappen, S. Budalakoti, R. Krishnamachari, S. R. Valluri, A. Wood, and M. Zait, "Adaptive statistics in oracle 12c," *Proc. VLDB Endow.*, vol. 10, no. 12, p. 1813–1824, aug 2017. [Online]. Available: https://doi.org/10.14778/3137765.3137785

[4] S. Chaudhuri and V. Narasayya, "Automating statistics management for query optimizers," in *Proceedings of 16th International Conference on Data Engineering (Cat. No.00CB37073)*, 2000, pp. 339–348.

[5] Q. Zhu, B. Dunkel, W. Lau, S. Chen, and B. Schiefer, "Piggyback statistics collection for query optimization: Towards a self-maintaining database management system," *The Computer Journal*, vol. 47, no. 2, pp. 221–244, 2004.

[6] A. Aboulnaga and S. Chaudhuri, "Self-tuning histograms: Building histograms without looking at data," *SIGMOD Rec.*, vol. 28, no. 2, p. 181–192, jun 1999. [Online]. Available: https://doi.org/10.1145/304181.304198

[7] Oracle. (2021) Mysql :: Mysql 8.0 reference manual :: 8.9.6 optimizer statistics. [Online]. Available: https://dev.mysql.com/doc/refman/8.0/en/optimizer-statistics.html

[8] G. Cormode, M. Garofalakis, P. J. Haas, and C. Jermaine, "Synopses for massive data: Samples, histograms, wavelets, sketches," *Found. Trends Databases*, vol. 4, no. 1–3, p. 1–294, jan 2012. [Online]. Available: https://doi.org/10.1561/1900000004

[9] G. Cormode and M. Muthukrishnan, "Approximating data with the count-min sketch," *IEEE software*, vol. 29, no. 1, pp. 64–69, 2011.

[10] N. H. Ryeng. (2020) Refactoring query processing in mysql. Oracle. [Online]. Available: https://www.youtube.com/watch?v=u7JOinvbMxc&ab_channel=CMUDatabaseGroup

[11] S. Nicolausson and J. Aasvestad, "Evaluating the performance of mysql's hypergraph optimizer," 2021.

[12] B. J. Oates, *Researching information systems and computing.* London, UK: SAGE Publications Ltd, 2006.

[13] A. Silberschatz, H. Korth, and S. Sudarshan, *Database Systems Concepts*, 5th ed. USA: McGraw-Hill, Inc., 2005.

[14] R. Elmasri and S. Navathe, *Fundamentals of Database Systems*, 7th ed. USA: Addison-Wesley Publishing Company, 2016.

[15] T. P. G. D. Group. (2022) Postgresql 14.3 documentation. [Online]. Available: https://www.postgresql.org/files/documentation/pdf/14/postgresql-14-A4.pdf

[16] M. Corporation. (2022) Query processing architecture guide. [Online]. Available: https://docs.microsoft.com/en-us/sql/relational-databases/query-processing-architecture-guide?view=sql-server-ver16

[17] J. M. Hellerstein, M. Stonebraker, and J. Hamilton, "Architecture of a database system," *Foundations and Trends in Databases*, vol. 1, no. 2, p. 141–259, Feb. 2007. [Online]. Available: https://doi.org/10.1561/1900000002

[18] R. Ahmed, A. Lee, A. Witkowski, D. Das, H. Su, M. Zait, and T. Cruanes, "Cost-based query transformation in oracle," in *Proceedings of the 32nd International Conference on Very Large Data Bases*, ser. VLDB '06. VLDB Endowment, 2006, p. 1026–1036.

[19] S. Chaudhuri, "An overview of query optimization in relational systems," in *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, ser. PODS '98. New York, NY, USA:

Association for Computing Machinery, 1998, p. 34–43. [Online]. Available: https://doi.org/10.1145/275487.275492

[20] A. Nica, "A call for order in search space generation process of query optimization," in *2011 IEEE 27th International Conference on Data Engineering Workshops*, 2011, pp. 4–9.

[21] G. Moerkotte, *Building Query Compilers (Under Construction)*, 2020.

[22] R. Bellman, "Dynamic programming," *Science*, vol. 153, no. 3731, pp. 34–37, 1966. [Online]. Available: https://www.science.org/doi/abs/10.1126/science.153.3731.34

[23] D. DeHaan and F. W. Tompa, "Optimal top-down join enumeration," in *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 785–796. [Online]. Available: https://doi.org/10.1145/1247480.1247567

[24] Y. E. Ioannidis and S. Christodoulakis, "On the propagation of errors in the size of join results," in *Proceedings of the 1991 ACM SIGMOD International Conference on Management of data*, 1991, pp. 268–277.

[25] V. Leis, B. Radke, A. Gubichev, A. Kemper, and T. Neumann, "Cardinality estimation done right: Index-based join sampling," in *CIDR*, 2017.

[26] A. P. Marathe, S. Lin, W. Yu, K. E. Gebaly, P. Åke Larson, and C. Sun, "Integrating the orca optimizer into mysql," in *Advances in Database Technology - Volume 25*. OpenProceedings.org, 2022, pp. 511–523.

[27] P. E. Black. hypergraph, in Dictionary of Algorithms and Data Structures. [Online]. Available: https://www.nist.gov/dads/HTML/hypergraph.html

[28] G. Bhargava, P. Goel, and B. Iyer, "Hypergraph based reorderings of outer join queries with complex predicates," in *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '95. New York, NY, USA: Association for Computing Machinery, 1995, p. 304–315. [Online]. Available: https://doi.org/10.1145/223784.223847

[29] M. Mohammadi, Dr.Shabankhah, and A. Kamandi, "Improving hypergraph attention and hypergraph convolutional networks," 01 2021.

[30] N. H. Ryeng, personal communication, 2022.

[31] Oracle. (2021) Mysql source code documentation 8.0.27 : join_optimizer.h file reference. [Online]. Available: https://dev.mysql.com/doc/dev/mysql-server/latest/join__optimizer_8h.html

[32] N. Bruno and S. Chaudhuri, "Exploiting statistics on query expressions for optimization," in *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '02. New York, NY, USA: Association for Computing Machinery, 2002, p. 263–274. [Online]. Available: https://doi.org/10.1145/564691.564722

[33] Y. E. Ioannidis and V. Poosala, "Histogram-based solutions to diverse database estimation problems," *IEEE Data Eng. Bull.*, vol. 18, pp. 10–18, 1995.

[34] R. P. Kooi, "The optimization of queries in relational databases," Ph.D. dissertation, USA, 1980, aAI8109596.

[35] B. J. Oommen and L. G. Rueda, "The Efficiency of Histogram-like Techniques for Database Query Optimization," *The Computer Journal*, vol. 45, no. 5, pp. 494–510, 01 2002. [Online]. Available: https://doi.org/10.1093/comjnl/45.5.494

[36] G. Piatetsky-Shapiro and C. Connell, "Accurate estimation of the number of tuples satisfying a condition," *SIGMOD Rec.*, vol. 14, no. 2, p. 256–276, Jun. 1984. [Online]. Available: https://doi.org/10.1145/971697.602294

[37] Oracle. (2019) Histograms. [Online]. Available: https://docs.oracle.com/database/121/TGSQL/tgsql_histo.htm#TGSQL366

[38] Y. Ioannidis, "The history of histograms (abridged)," in *Proceedings 2003 VLDB Conference*, J.-C. Freytag, P. Lockemann, S. Abiteboul, M. Carey, P. Selinger, and A. Heuer, Eds. San Francisco: Morgan Kaufmann, 2003, pp. 19–30. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780127224428500112

[39] Oracle. (2021) Mysql :: Mysql 8.0 reference manual :: 13.7.3.1 analyze table statement. [Online]. Available: https://dev.mysql.com/doc/refman/8.0/en/analyze-table.html

[40] The PostgreSQL Global Development Group. (2021) Postgresql: Documentation: 14: 14.2. statistics used by the planner. [Online]. Available: https://www.postgresql.org/docs/14/planner-stats.html

[41] T. Halpin and T. Morgan, *Information modeling and relational databases*. Morgan Kaufmann, 2010.

[42] The PostgreSQL Global Development Group. (2021) Postgresql: Documentation: 14: 72.2. multivariate statistics examples. [Online]. Available: https://www.postgresql.org/docs/14/multivariate-statistics-examples.html

[43] Microsoft. Statistics - sql server — microsoft docs. [Online]. Available: https://docs.microsoft.com/en-us/sql/relational-databases/statistics/statistics?view=sql-server-ver15

[44] ——. Alter database set options (transact-sql) - sql server — microsoft docs. [Online]. Available: https://docs.microsoft.com/en-us/sql/t-sql/statements/alter-database-transact-sql-set-options?view=sql-server-ver15

[45] Oracle. Managing optimizer statistics. [Online]. Available: https://docs.oracle.com/cd/E18283_01/server.112/e16638/stats.htm

[46] ——. Histograms. [Online]. Available: https://docs.oracle.com/database/121/TGSQL/tgsql_histo.htm#TGSQL367

[47] J. Gama and P. Rodrigues, *Data Stream Processing*, 01 2007, pp. 25–39.

[48] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, ""models and issues in data stream systems."," 06 2002, pp. 1–16.

[49] G. Navarro, *Compact Data Structures: A Practical Approach*. Cambridge University Press, 2016.

[50] G. Cormode, "Summary data structures for massive data," in *The Nature of Computation. Logic, Algorithms, Applications*, P. Bonizzoni, V. Brattka, and B. Löwe, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 78–86.

[51] U. of California San Diego, "Cse 190, great ideas in algorithms: Pairwise independent hash functions." [Online]. Available: https://cseweb.ucsd.edu/~slovett/teaching/SP15-CSE190/pairwise_hash_functions.pdf

[52] S. Arora and S. Sachdeva, "Lecture 2: Hashing," 2008. [Online]. Available: https://www.cs.princeton.edu/courses/archive/fall08/cos521/hash.pdf

[53] M. N. Wegman and J. L. Carter, "New hash functions and their use in authentication and set equality," *Journal of computer and system sciences*, vol. 22, no. 3, pp. 265–279, 1981.

[54] C. Mckay, *Probability and Statistics*. Scientific e-Resources, 2019.

[55] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0196677403001913

[56] G. Cormode, "Sketch techniques for approximate query processing," *Foundations and Trends in Databases. NOW publishers*, 2011.

[57] ——, "Count-min sketch," in *Encyclopedia of Database Systems*, 2009.

[58] G. Cormode and S. Muthukrishnan, "Summarizing and mining skewed data streams," 04 2005.

[59] F. Deng and D. Rafiei, "New estimation algorithms for streaming data : Count-min can do more," 2007.

[60] Oracle. (2022) Mysql source code documentation 8.0.29 : Item class reference. [Online]. Available: https://dev.mysql.com/doc/dev/mysql-server/latest/classItem.html

[61] O. Yigit. String hash functions. [Online]. Available: http://www.cse.yorku.ca/~oz/hash.html

[62] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann, "How good are query optimizers, really?" *Proc. VLDB Endow.*, vol. 9, no. 3, p. 204–215, Nov. 2015. [Online]. Available: https://doi.org/10.14778/2850583.2850594

[63] A. C. Gilbert, S. Guha, P. Indyk, Y. Kotidis, S. Muthukrishnan, and M. J. Strauss, "Fast, small-space algorithms for approximate histogram maintenance," in *Proceedings of the Thiry-Fourth Annual ACM Symposium on Theory of Computing*, ser. STOC '02. New York, NY, USA: Association for Computing Machinery, 2002, p. 389–398. [Online]. Available: https://doi.org/10.1145/509907.509966

[64] K. Chakrabarti, M. Garofalakis, R. Rastogi, and K. Shim, "Approximate query processing using wavelets," *The VLDB Journal*, vol. 10, no. 2–3, p. 199–223, sep 2001.

[65] Y. Matias, J. Vitter, and M. Wang, "Dynamic maintenance of wavelet-based histograms," 02 2000.

[66] R. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul, "Neo," *Proceedings of the VLDB Endowment*, vol. 12, no. 11, pp. 1705–1718, jul 2019. [Online]. Available: https://doi.org/10.14778%2F3342263.3342644

Appendices

A Injecting Correct Selectivity - Full Results

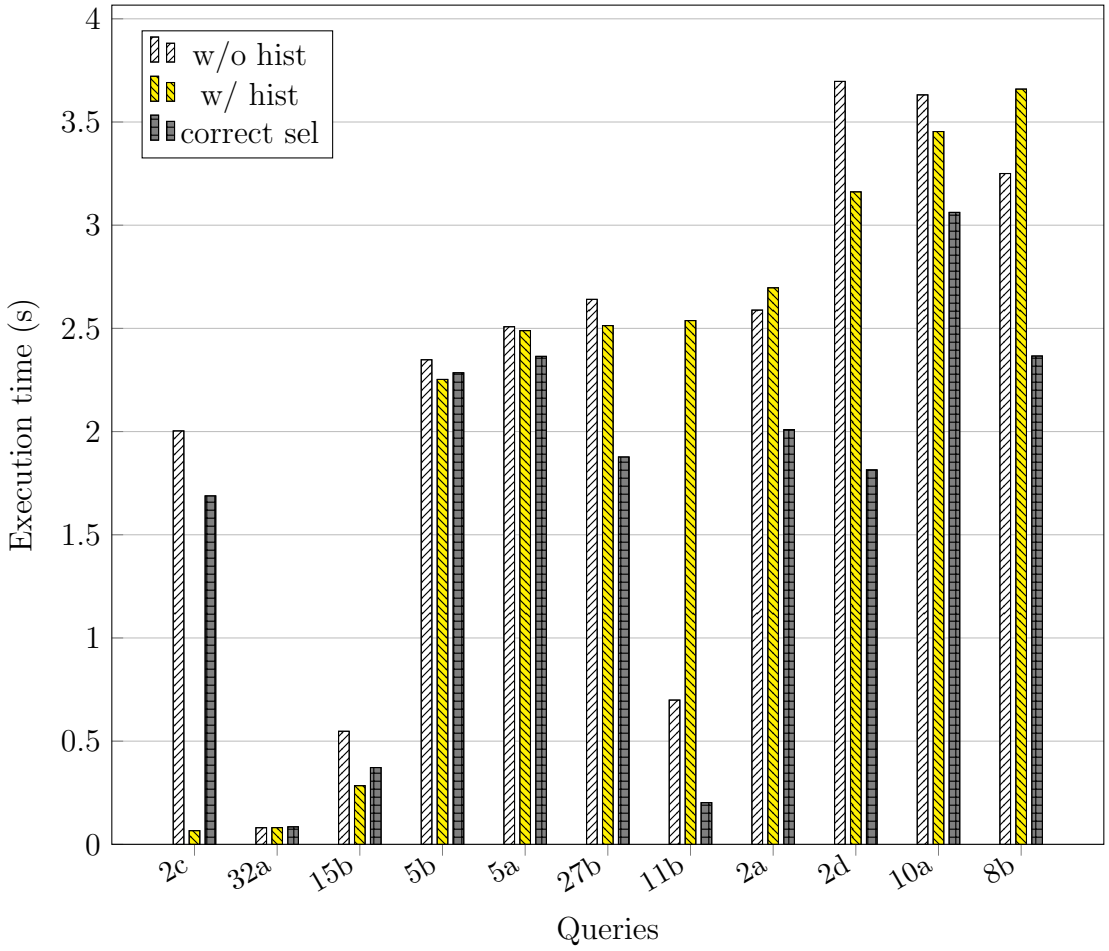


Figure A.1: JOB query execution time results between plain hypergraph optimizer, hypergraph optimizer with histograms on base tables, and hypergraph optimizer with correct selectivity injected (1/6).

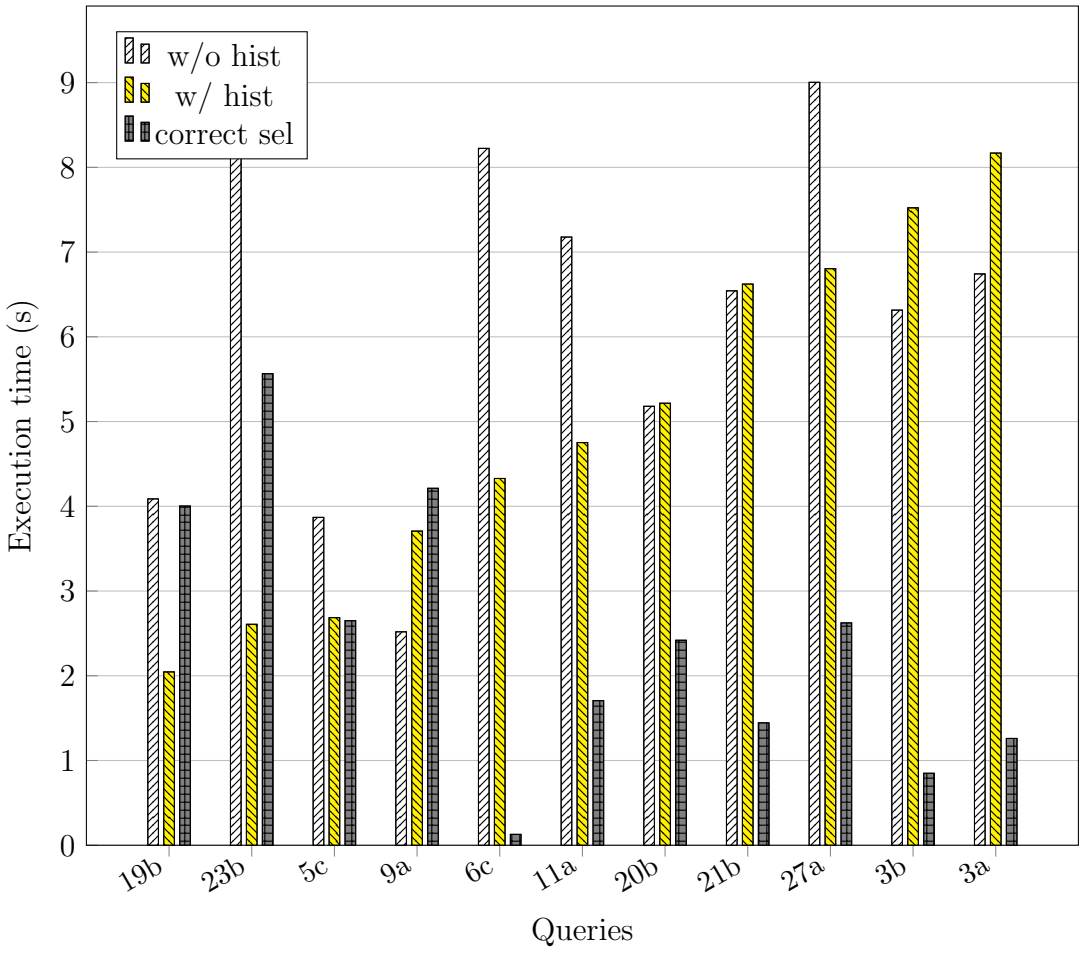


Figure A.2: JOB query execution time results between plain hypergraph optimizer, hypergraph optimizer with histograms on base tables, and hypergraph optimizer with correct selectivity injected (2/6).

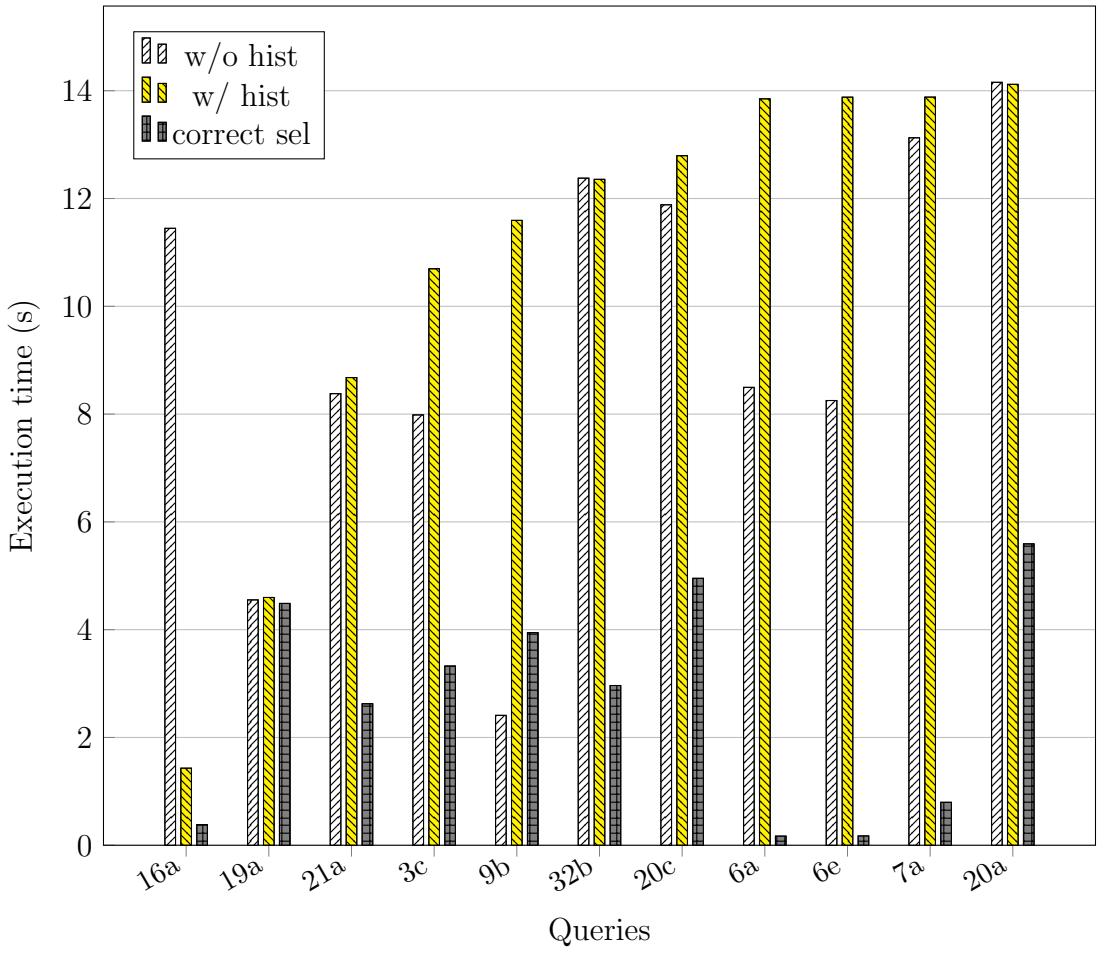


Figure A.3: JOB query execution time results between plain hypergraph optimizer, hypergraph optimizer with histograms on base tables, and hypergraph optimizer with correct selectivity injected (3/6).

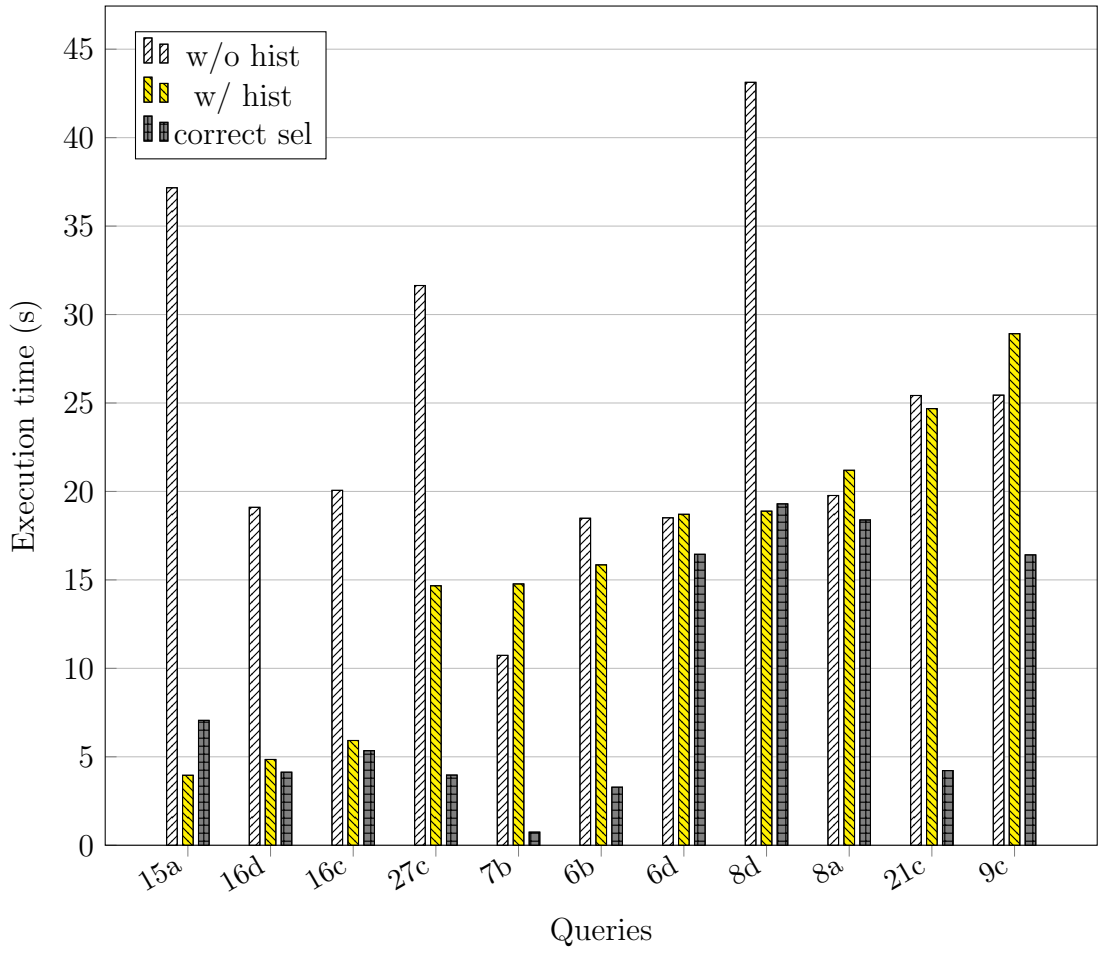


Figure A.4: JOB query execution time results between plain hypergraph optimizer, hypergraph optimizer with histograms on base tables, and hypergraph optimizer with correct selectivity injected (4/6).

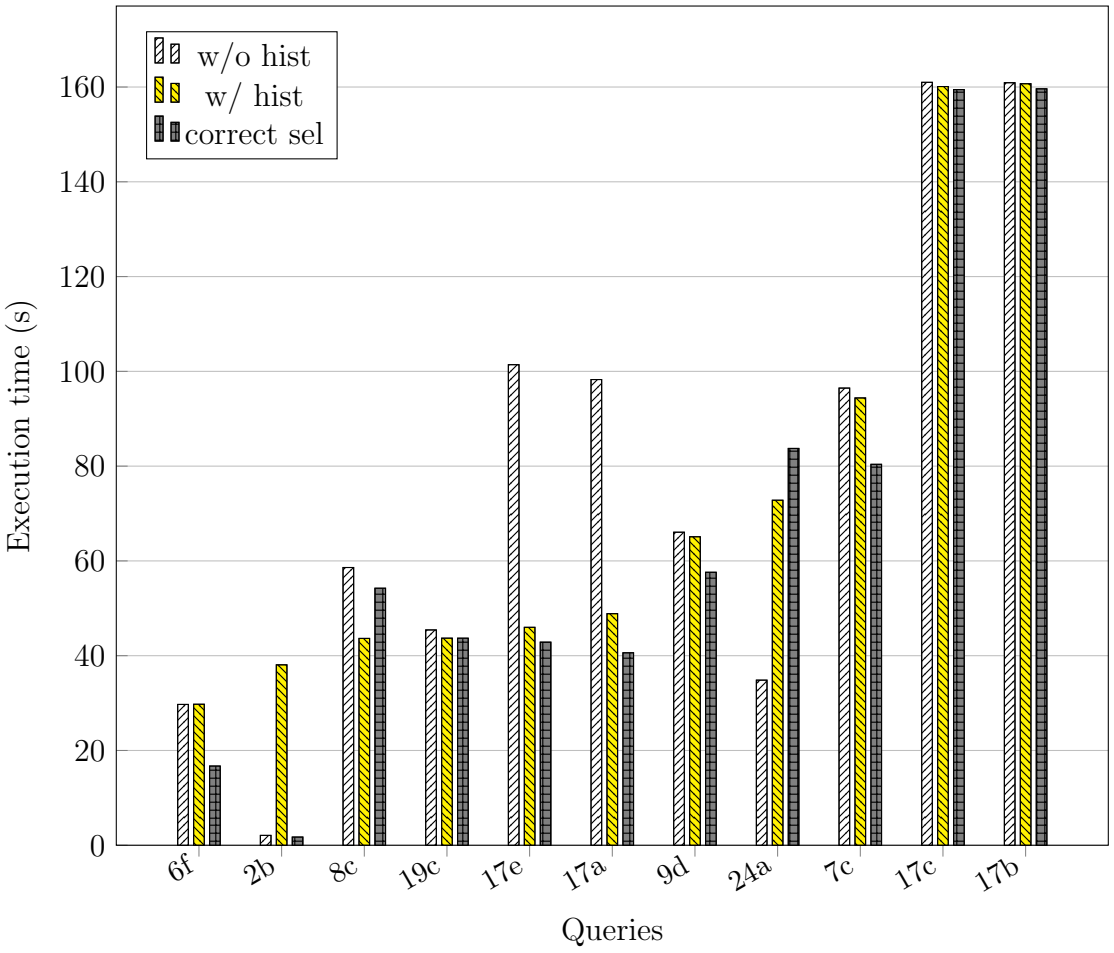


Figure A.5: JOB query execution time results between plain hypergraph optimizer, hypergraph optimizer with histograms on base tables, and hypergraph optimizer with correct selectivity injected (5/6).

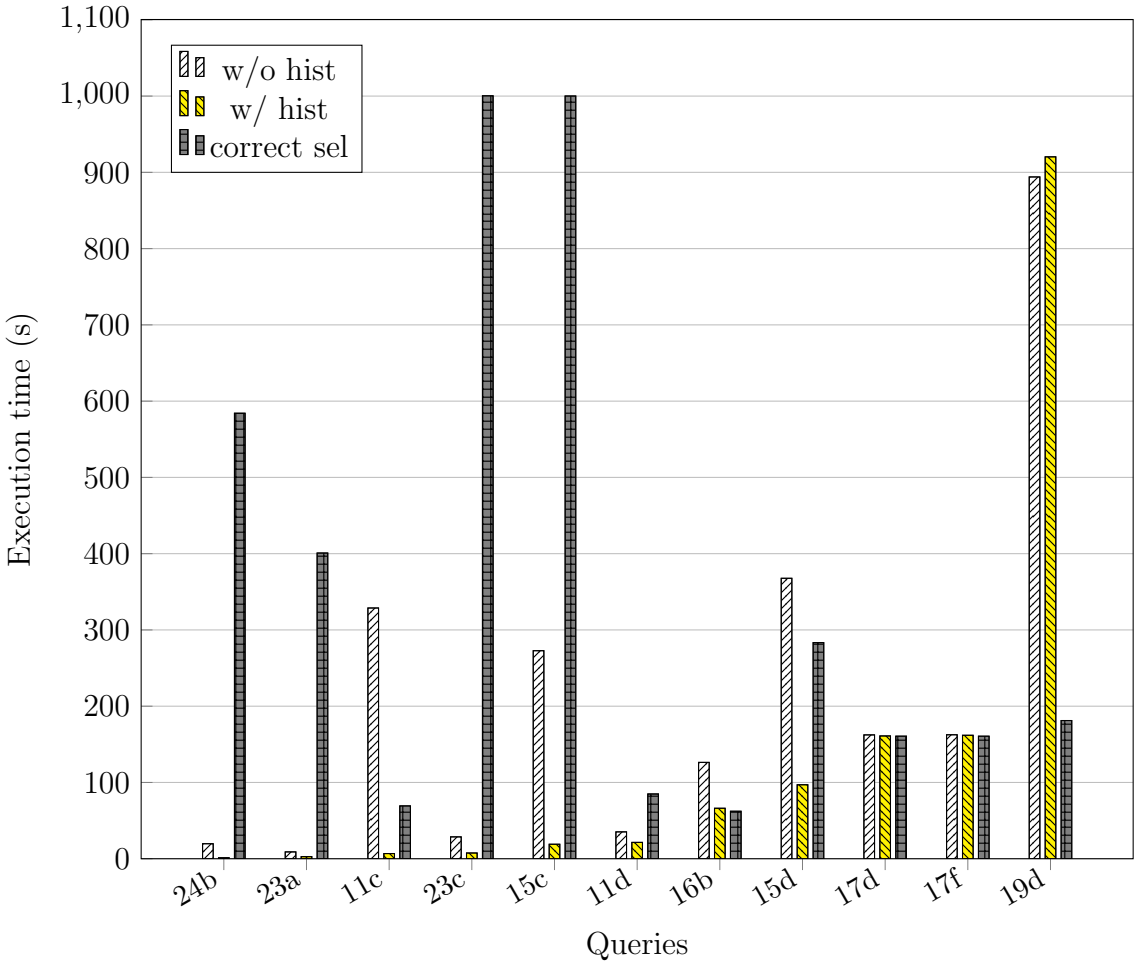


Figure A.6: JOB query execution time results between plain hypergraph optimizer, hypergraph optimizer with histograms on base tables, and hypergraph optimizer with correct selectivity injected (6/6).

B Count-Min Sketch - Full Results

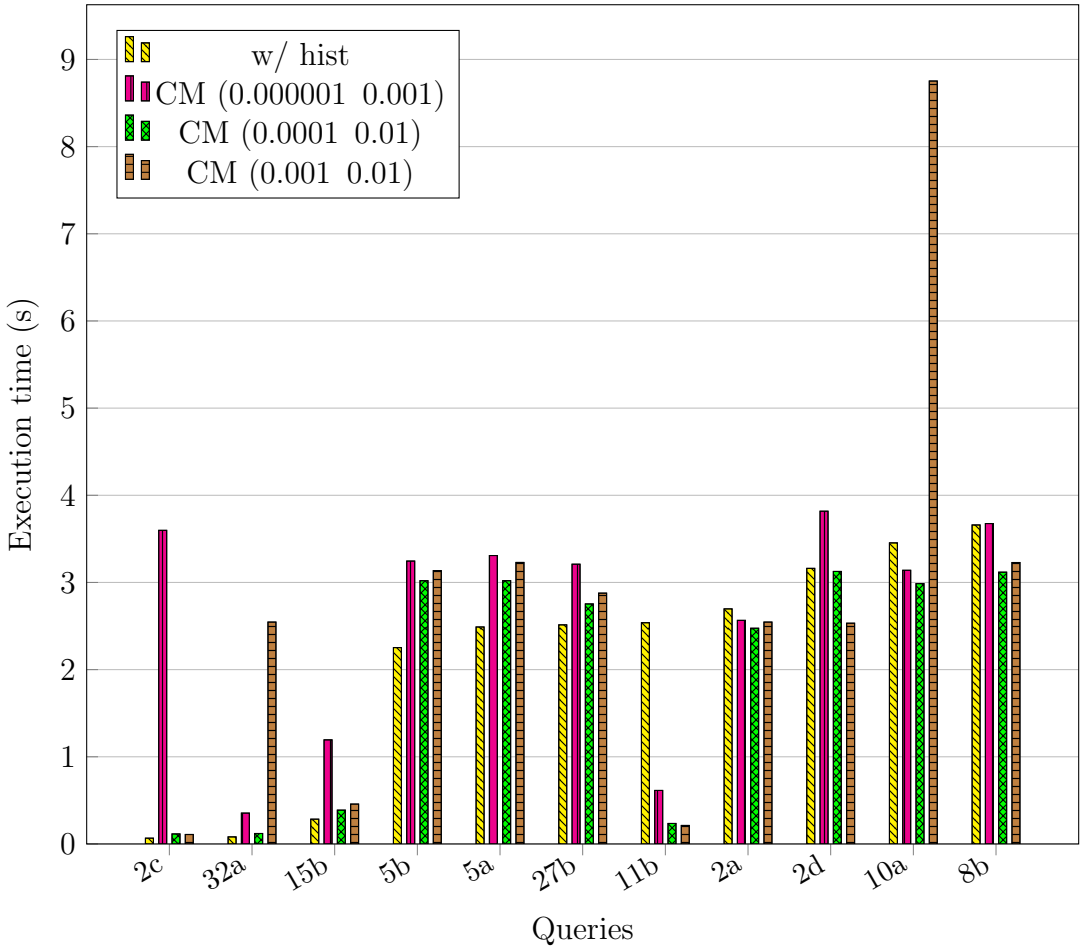


Figure B.1: JOB query execution times for hypergraph optimizer with histograms on base tables and hypergraph optimizer with count-min sketches of various sizes (1/6).

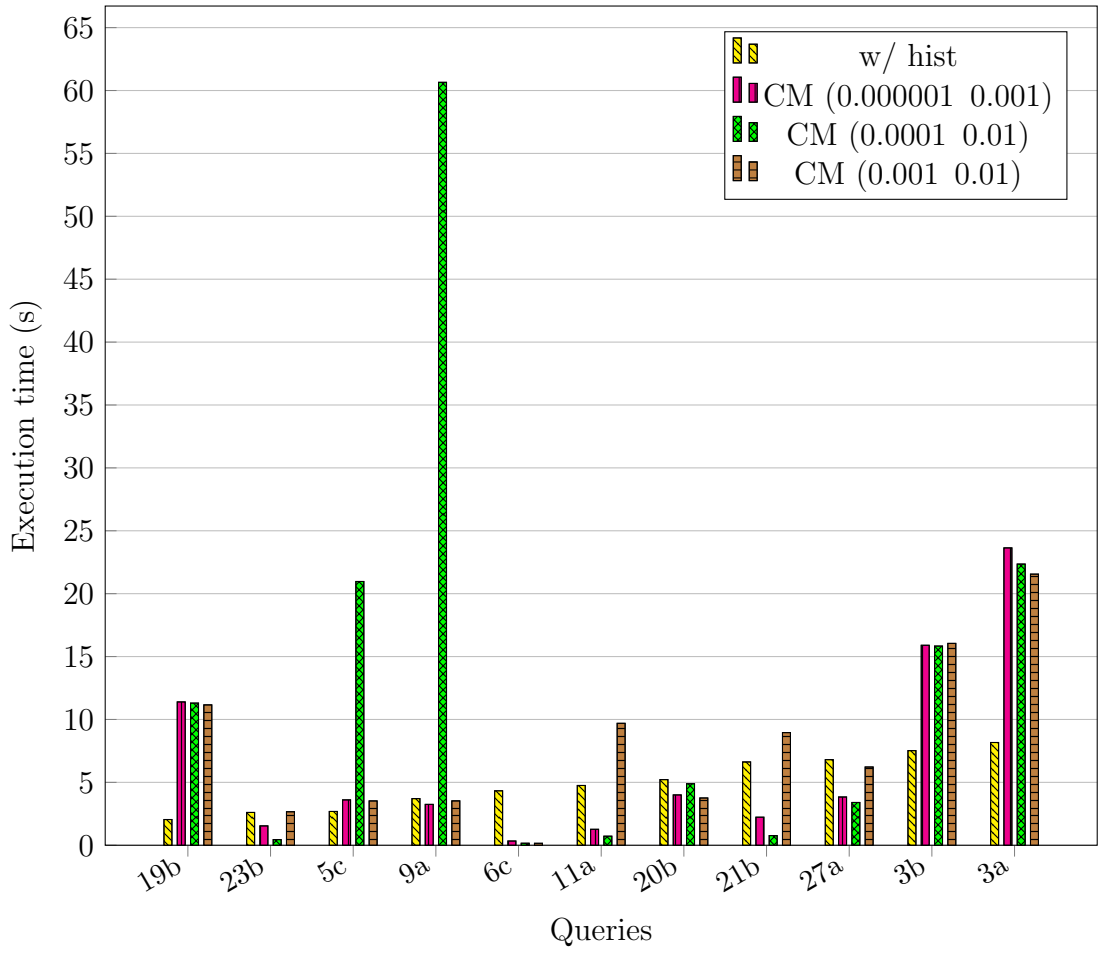


Figure B.2: JOB query execution times for hypergraph optimizer with histograms on base tables and hypergraph optimizer with count-min sketches of various sizes (2/6).

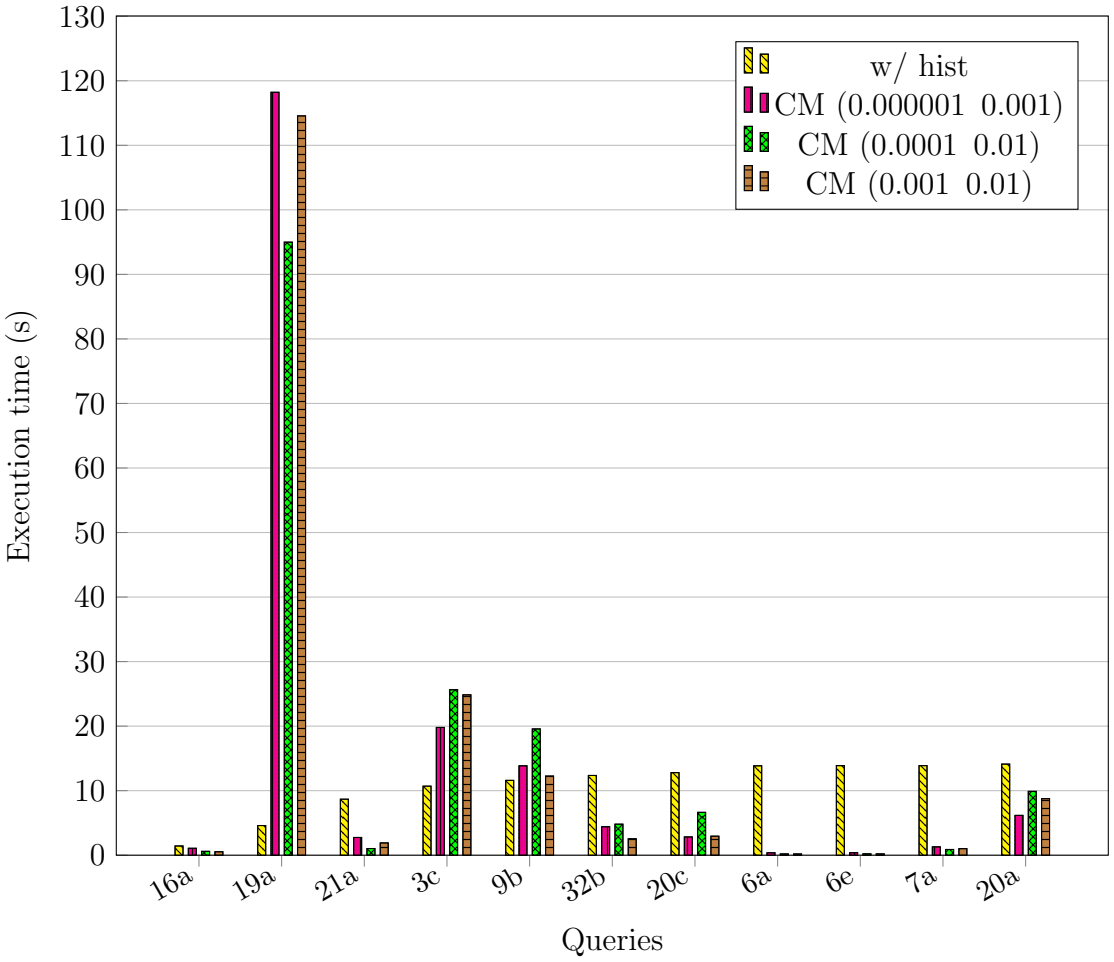


Figure B.3: JOB query execution times for hypergraph optimizer with histograms on base tables and hypergraph optimizer with count-min sketches of various sizes (3/6).

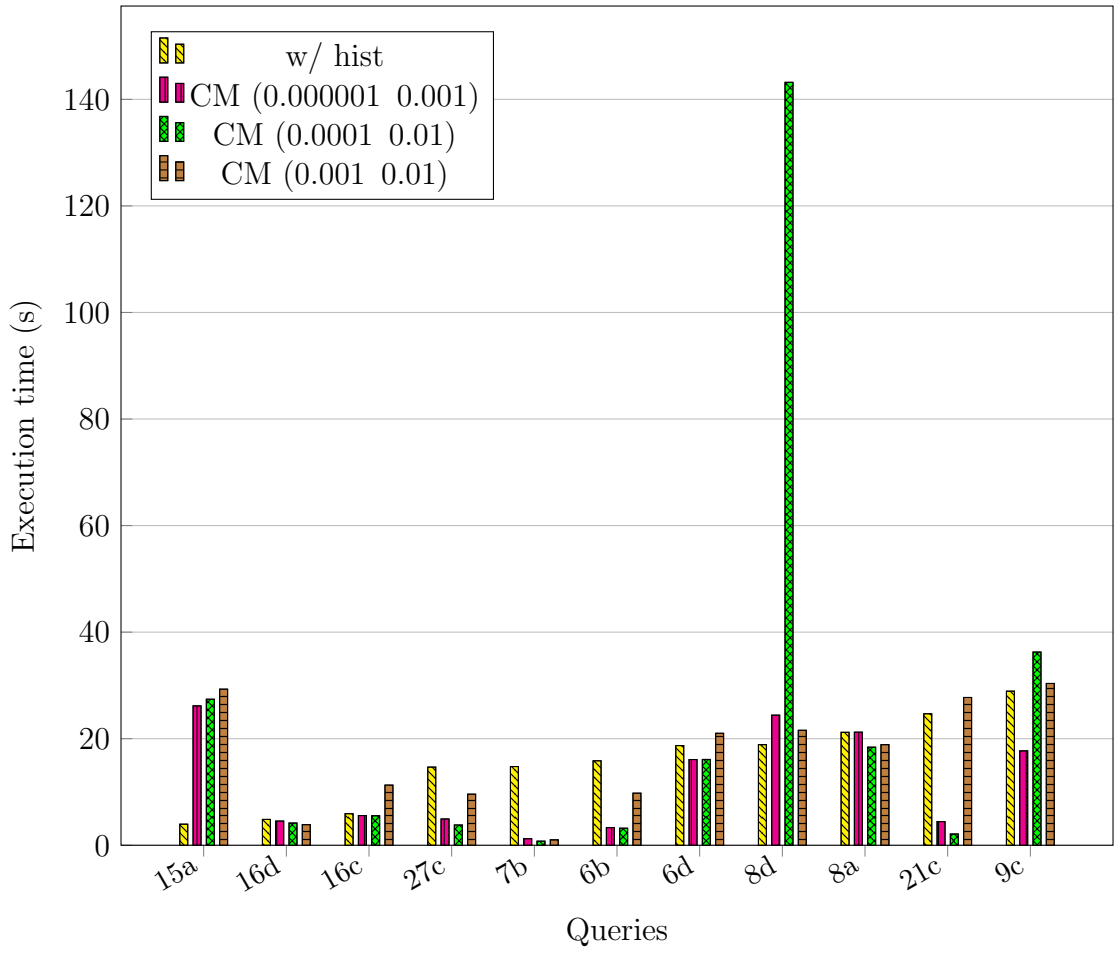


Figure B.4: JOB query execution times for hypergraph optimizer with histograms on base tables and hypergraph optimizer with count-min sketches of various sizes (4/6).

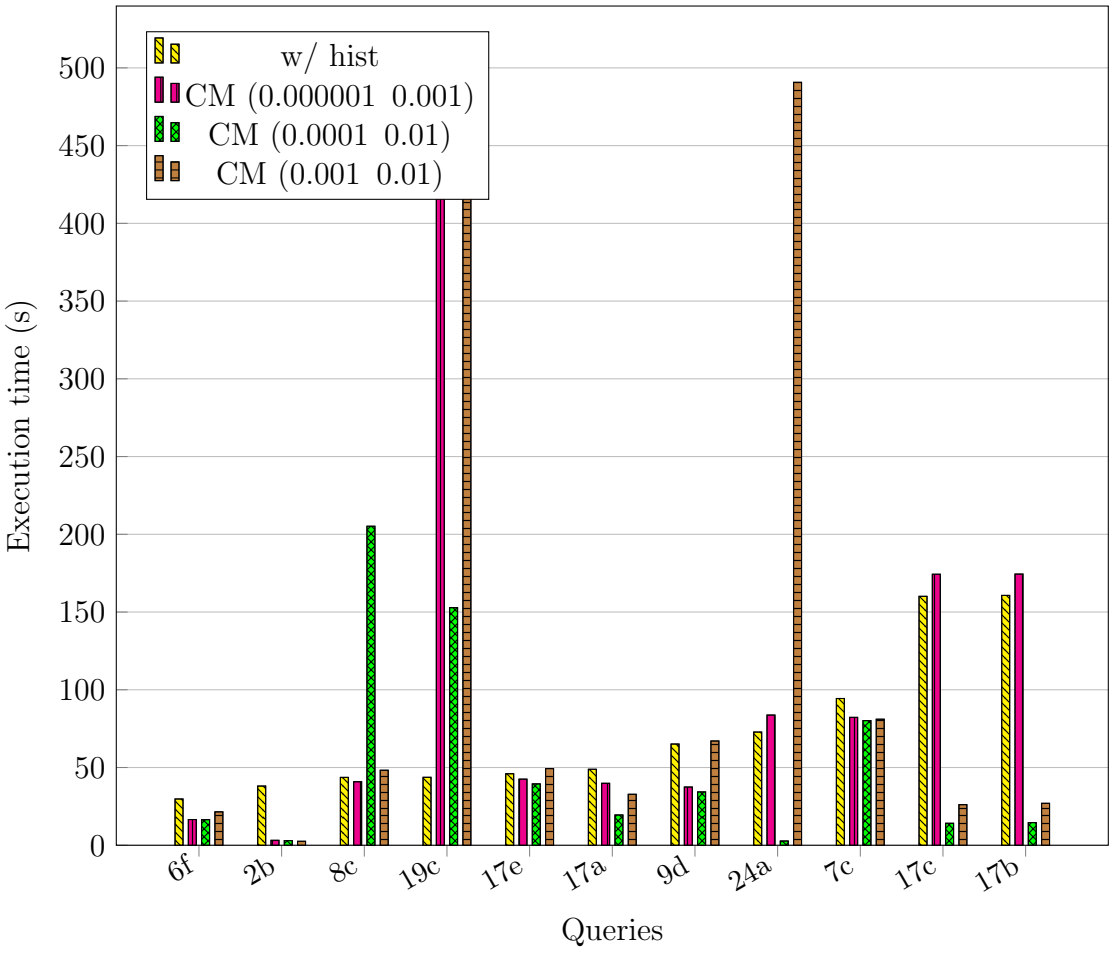


Figure B.5: JOB query execution times for hypergraph optimizer with histograms on base tables and hypergraph optimizer with count-min sketches of various sizes (5/6).

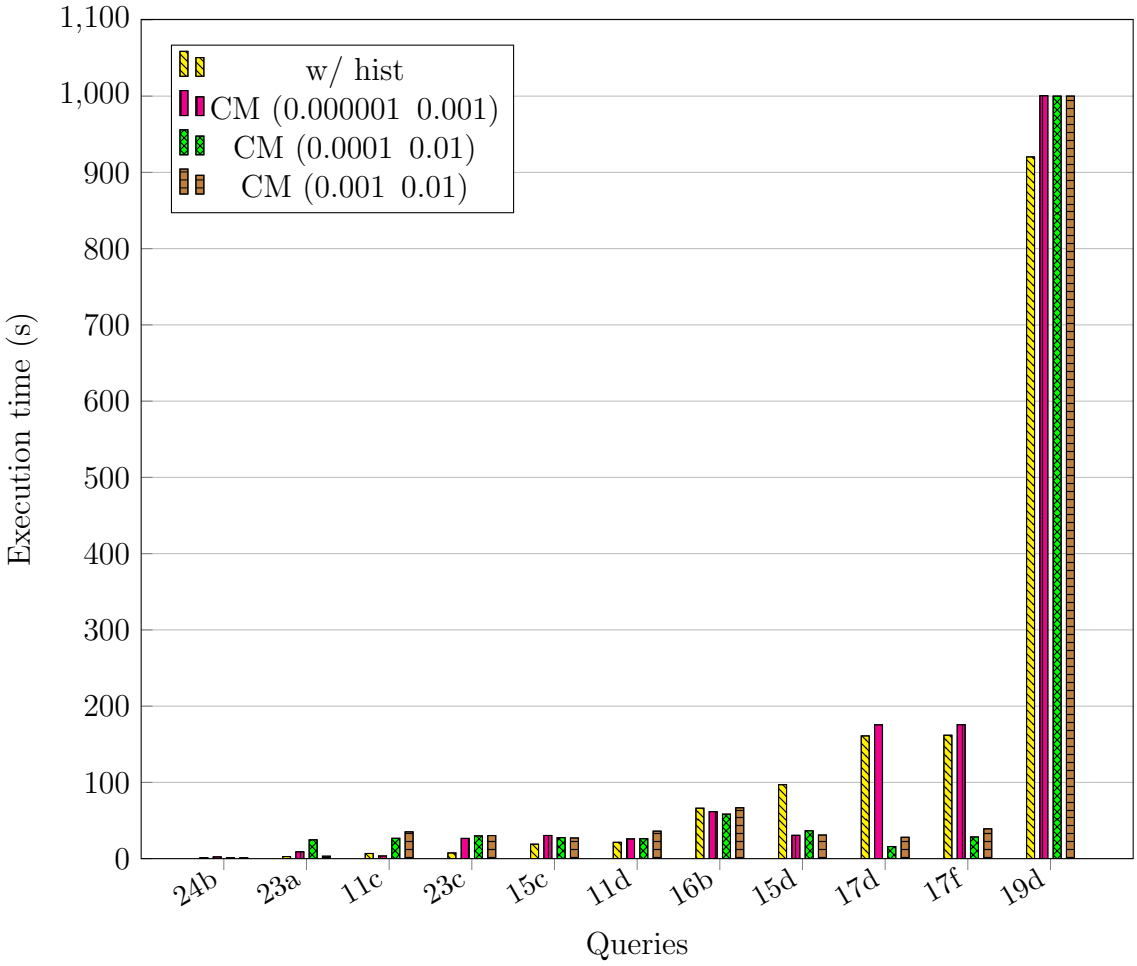


Figure B.6: JOB query execution times for hypergraph optimizer with histograms on base tables and hypergraph optimizer with count-min sketches of various sizes (6/6).

C Count-Min Mean Sketch - Full Results

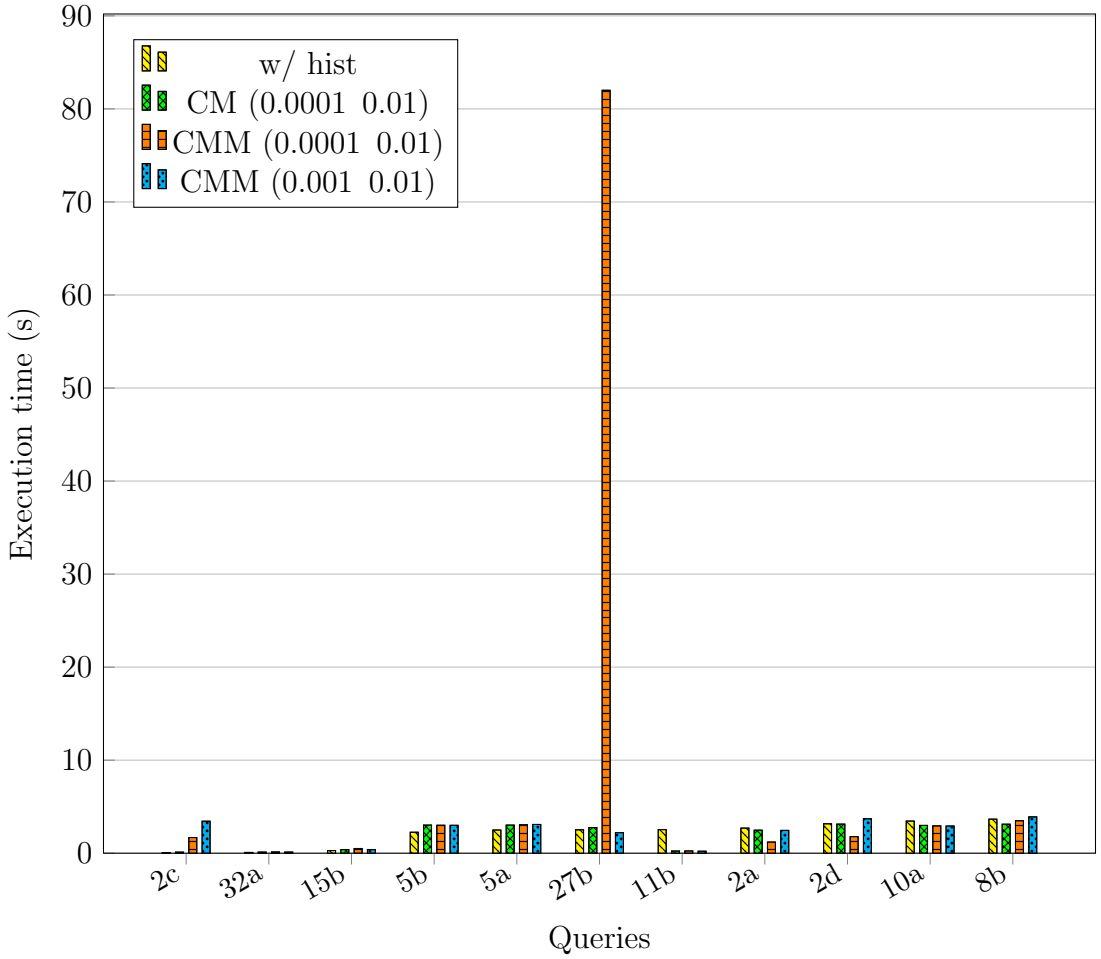


Figure C.1: JOB query execution times comparison between histograms, count-min sketch and count-min mean sketch (1/6).

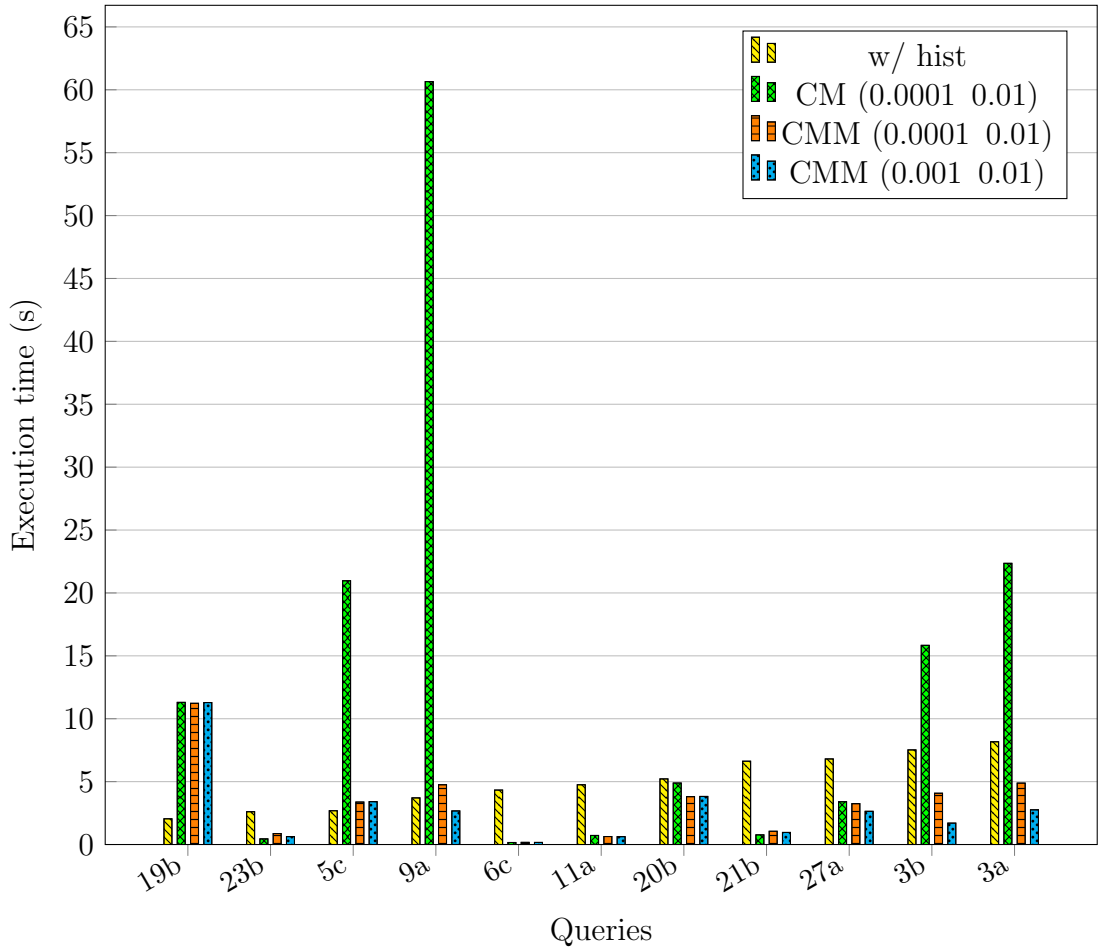


Figure C.2: JOB query execution times comparison between histograms, count-min sketch and count-min mean sketch (2/6).

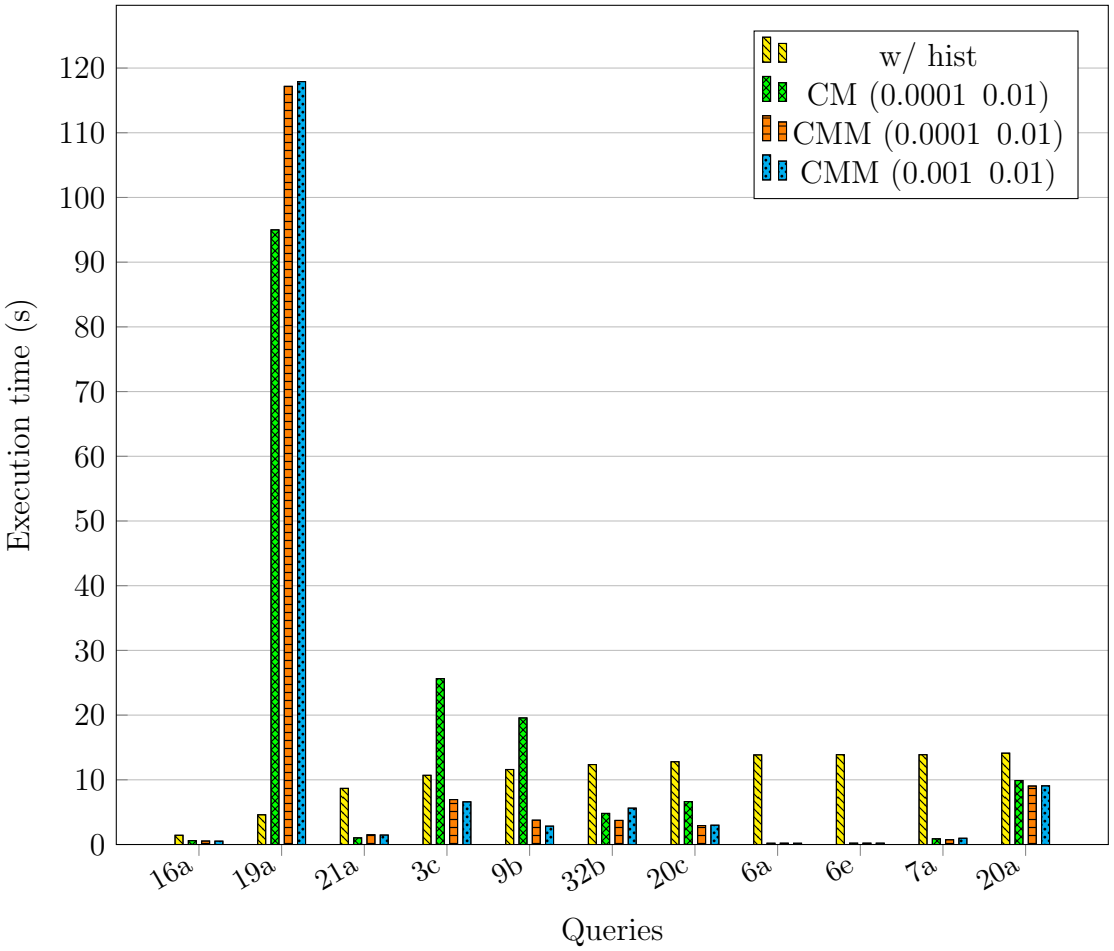


Figure C.3: JOB query execution times comparison between histograms, count-min sketch and count-min mean sketch (3/6).

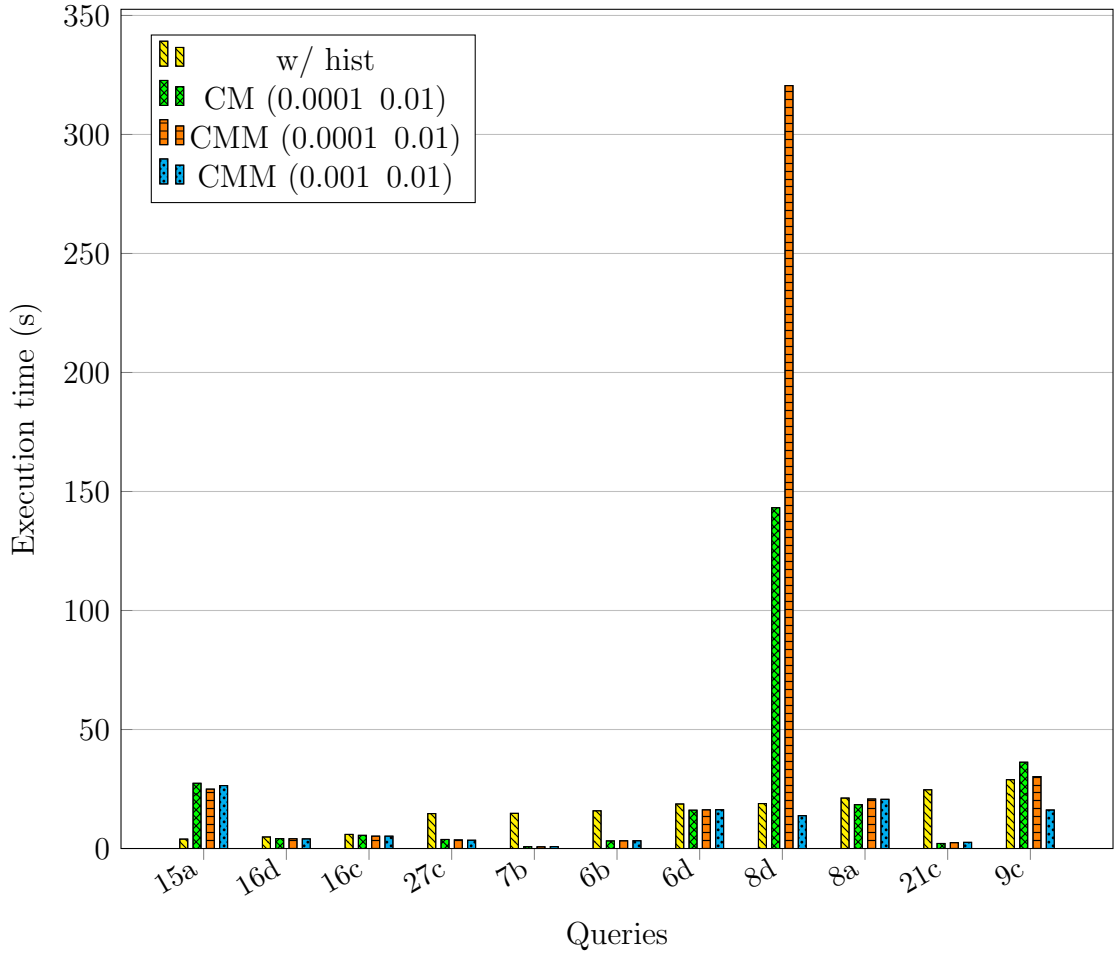


Figure C.4: JOB query execution times comparison between histograms, count-min sketch and count-min mean sketch (4/6).

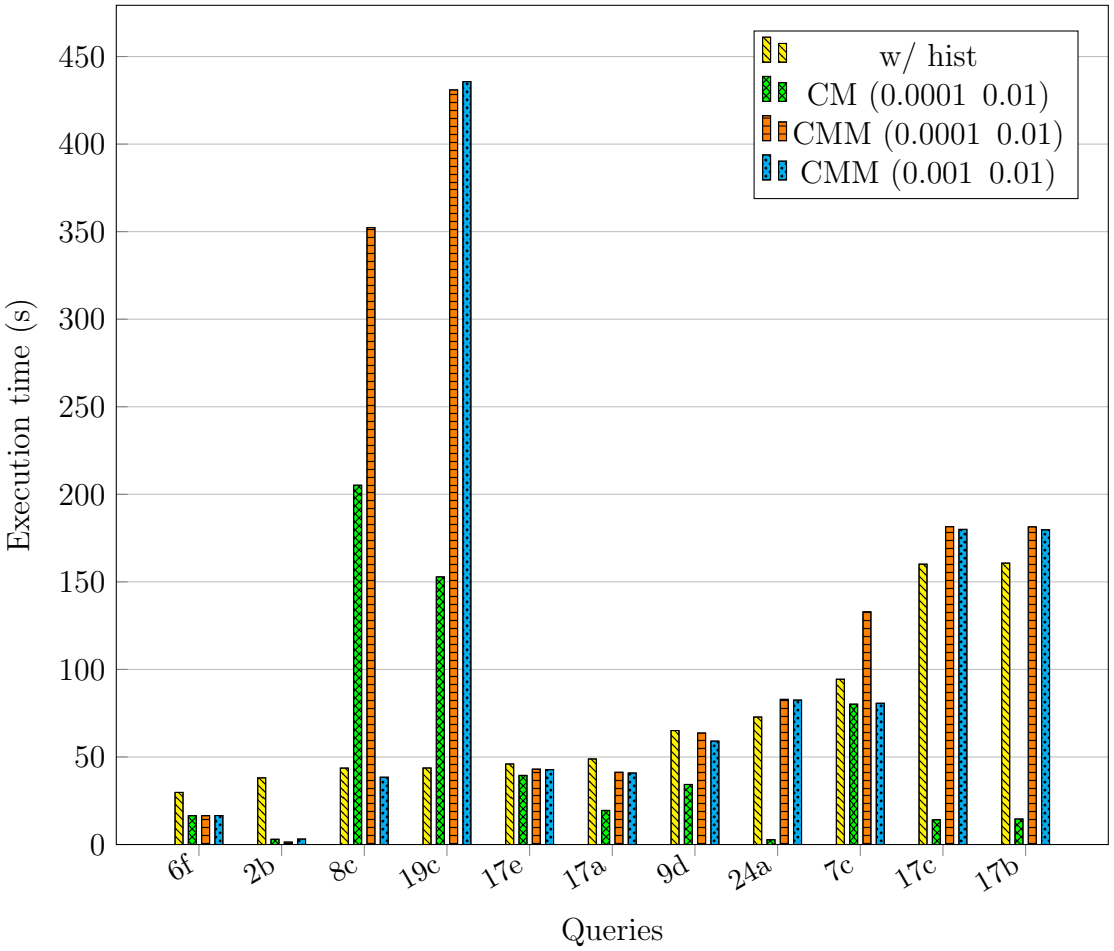


Figure C.5: JOB query execution times comparison between histograms, count-min sketch and count-min mean sketch (5/6).

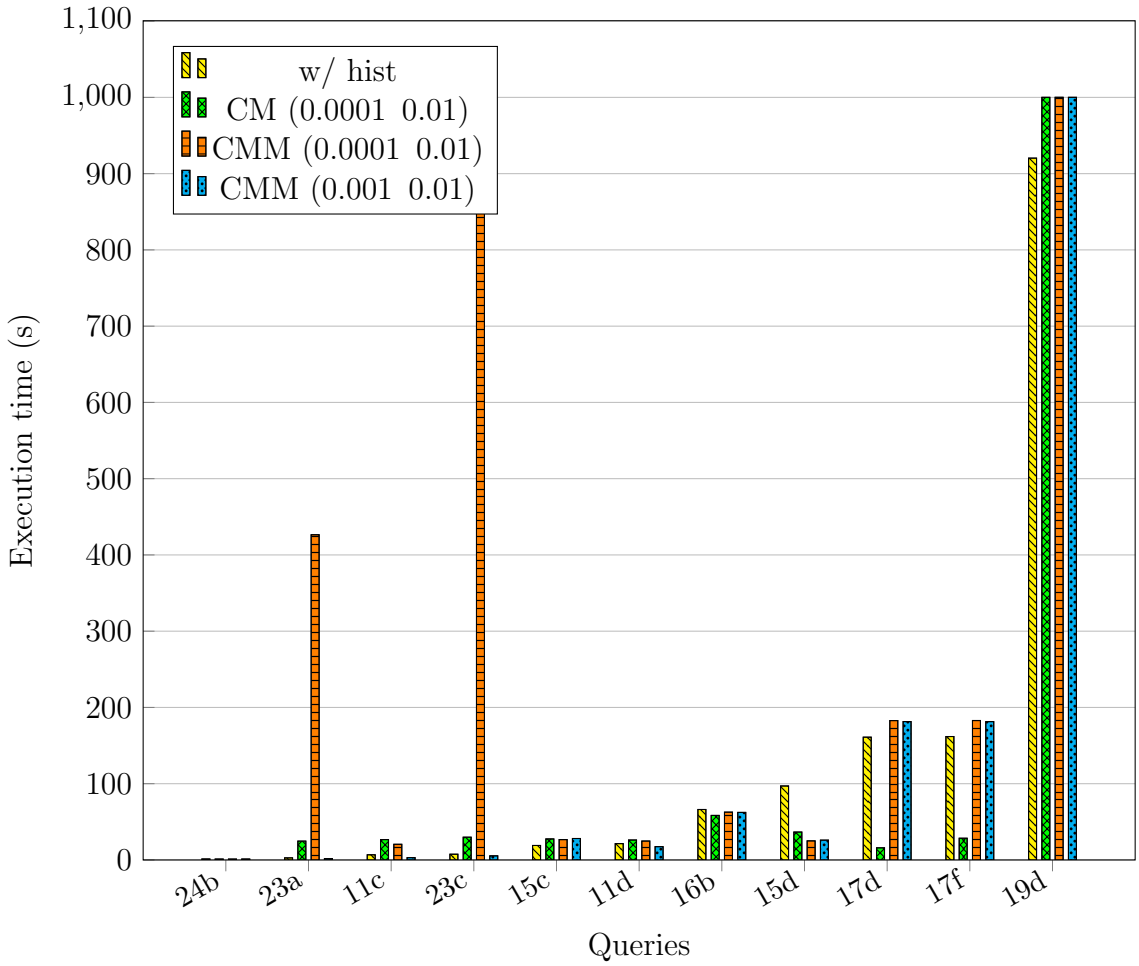


Figure C.6: JOB query execution times comparison between histograms, count-min sketch and count-min mean sketch (6/6).

D Injecting Correct Selectivity - Raw Data

Table D.1: Full results from the experiment where correct selectivity is injected into the hypergraph optimizer. Execution time given in seconds.

Query	w/o hist	w/ hist	correct sel
2a	2.58853675	2.69676675	2.00926375
2b	2.08080775	38.07224025	1.73027175
2c	2.0032295	0.0662555	1.68844175
2d	3.6966445	3.161599	1.814434
3a	6.742449	8.1684315	1.26032225
3b	6.3150285	7.522327	0.84944075
3c	7.984523	10.696397	3.32574975
5a	2.508009	2.489173	2.36445625
5b	2.34805275	2.252633	2.28487775
5c	3.869021	2.68629525	2.649735
6a	8.4957745	13.8494665	0.16962925
6b	18.4843655	15.85076875	3.28243675
6c	8.223749	4.328338	0.12841725
6d	18.51093675	18.708708	16.4478675
6e	8.25196575	13.8802745	0.172997
6f	29.71459925	29.75479475	16.716101
7a	13.126766	13.88148575	0.79686075
7b	10.73327725	14.77227075	0.74129275
7c	96.470184	94.39113625	80.383847
8a	19.77113625	21.1962555	18.3900055
8b	3.2503945	3.659481	2.366226
8c	58.609154	43.645827	54.2379
8d	43.12644175	18.885254	19.2984495
9a	2.518727	3.707231	4.21251625
9b	2.409987	11.594051	3.94440475
9c	25.4429075	28.91763325	16.41382
9d	66.05356375	65.09944	57.604711
10a	3.63146275	3.453816	3.06198
11a	7.17719275	4.7518925	1.70656725
11b	0.699008	2.5378685	0.20213575
11c	328.7634565	6.7738725	69.26705175

Table D.1: Full results from the experiment where correct selectivity is injected into the hypergraph optimizer. Execution time given in seconds.

Query	w/o hist	w/ hist	correct sel
11d	35.21942875	21.41897925	84.91566725
15a	37.167377	3.95552175	7.059823
15b	0.54780875	0.2842715	0.37202325
15c	272.7584675	18.98529025	1000.087345
15d	367.711999	96.99746575	283.3681665
16a	11.44680275	1.43005575	0.3835555
16b	126.3152055	66.16808075	62.257468
16c	20.06148825	5.92175675	5.34589175
16d	19.1024705	4.8488275	4.129613
17a	98.27308975	48.86463975	40.6133175
17b	160.9077675	160.695851	159.6204175
17c	160.9894367	160.1039987	159.4582862
17d	162.3715385	160.9996547	160.6970997
17e	101.4084663	45.98694075	42.8577035
17f	162.5433955	161.8619682	160.6669907
19a	4.55264025	4.5972645	4.4869465
19b	4.087252	2.0456875	4.0031995
19c	45.43445625	43.698162	43.69360025
19d	893.972948	920.3295277	181.213744
20a	14.15565475	14.11826675	5.5925925
20b	5.18019725	5.216318	2.41922
20c	11.88342675	12.79470025	4.9529625
21a	8.3793975	8.67775125	2.625996
21b	6.542759	6.62233575	1.444413
21c	25.42423525	24.68209925	4.22025225
23a	8.95349925	2.7220665	400.8502742
23b	8.946744	2.60756525	5.564684
23c	28.66044375	7.51796225	1000.359032
24a	34.8406315	72.805419	N/A
24b	19.67539325	1.2776215	584.1497695
27a	9.00323225	6.80409	2.6240095
27b	2.64061525	2.513611	1.8769665
27c	31.64145075	14.66880375	3.96732875

Table D.1: Full results from the experiment where correct selectivity is injected into the hypergraph optimizer. Execution time given in seconds.

Query	**w/o hist**	**w/ hist**	**correct sel**
32a	0.08055775	0.0810145	0.085014
32b	12.3776165	12.3556265	2.96388275

E Count-Min Sketch - Raw Data

Table E.1: Full results from the experiment where count-min sketches are used to estimate selectivity. Execution times given in seconds.

Query	CM (0.000001, 0.001)	CM (0.0001, 0.01)	CM (0.001, 0.01)
2a	2.5650785	2.474683	2.545426
2b	3.17262525	2.99944575	2.5458755
2c	3.5978725	0.11484775	0.1088055
2d	3.8174205	3.12629875	2.53245875
3a	23.63970475	22.35376775	21.553254
3b	15.8976475	15.83384425	16.04752025
3c	19.79431475	25.64060225	24.850627
5a	3.3088655	3.02016025	3.22954575
5b	3.24532525	3.02016025	3.13582
5c	3.606082	20.96851525	3.52443075
6a	0.37446575	0.1996845	0.1934495
6b	3.310659	3.201223	9.7753455
6c	0.33623375	0.16017075	0.155253
6d	16.08609275	16.10441275	21.01561175
6e	0.386614	0.21113075	0.20316175
6f	16.540675	16.45962175	21.48759825
7a	1.2938105	0.8861305	1.014807
7b	1.23441475	0.768178	1.01553875
7c	82.25874	80.151132	81.05602725
8a	21.2210805	18.4090715	18.8877435
8b	3.6756245	3.11780675	3.228798
8c	40.8477855	205.1818402	48.2426155
8d	24.41747225	143.194065	21.5784965
9a	3.24869525	60.65085025	3.5278255
9b	13.849919	19.56620575	12.2632595
9c	17.71699175	36.26010075	30.36576875
9d	37.44019925	34.29839025	67.06318775
10a	3.139556	2.9864655	8.7524834
11a	1.26802325	0.720041	9.693694
11b	0.61392175	0.23512675	0.2123025
11c	3.53482525	26.69191375	35.13661625

Table E.1: Full results from the experiment where count-min sketches are used to estimate selectivity. Execution times given in seconds.

Query	CM (0.000001, 0.001)	CM (0.0001, 0.01)	CM (0.001, 0.01)
11d	25.9075645	26.14517725	36.0898175
15a	26.15991925	27.40795674	29.2971195
15b	1.19421725	0.387972	0.45740324
15c	30.384671	27.5669455	27.126717
15d	30.603499	36.5825885	30.9759984
16a	1.07300725	0.59754774	0.53627824
16b	61.65049275	58.403262	66.7617454
16c	5.574065	5.53633625	11.2886695
16d	4.537524	4.17145875	3.865373
17a	39.850939	19.47687224	32.77629824
17b	174.4110575	14.5762125	26.9708005
17c	174.3011943	14.133	26.1122194
17d	175.5115027	15.89654675	28.0601825
17e	42.513318	39.47130525	49.339469
17f	175.7178545	28.47918724	39.10696025
19a	118.2164578	94.99467974	114.5594114
19b	11.39509225	11.3001855	11.1591355
19c	433.4752872	152.8284164	429.5830762
19d	1000.544122	1000.062502	1000.050788
20a	6.16947025	9.89145625	8.76196325
20b	4.00093275	4.888448	3.75852
20c	2.8316284	6.639118	2.94817374
21a	2.748	1.0350324	1.90374075
21b	2.23	0.7678365	8.9547005
21c	4.4457055	2.115167	27.73550774
23a	9.028374	24.70354475	3.22518974
23b	1.53870125	0.4487005	2.6631954
23c	26.5619855	29.8349695	30.3041984
24a	83.73618725	2.732418	490.7009347
24b	2.33209975	1.1832505	1.1305504
27a	3.8362765	3.40057275	6.2210725
27b	3.2093695	2.753577	2.87742724
27c	4.93580425	3.8000165	9.58638625

Table E.1: Full results from the experiment where count-min sketches are used to estimate selectivity. Execution times given in seconds.

Query	**CM (0.000001, 0.001)**	**CM (0.0001, 0.01)**	**CM (0.001, 0.01)**
32a	0.3539035	0.12001575	2.545426
32b	4.41247625	4.807608	2.5458754

F Count-Min Mean Sketch - Raw Data

Table F.1: Full results from the experiment where count-min mean sketches are used for selectivity estimation. Execution time given in seconds.

Query	CMM (0.0001, 0.01)	CMM (0.001, 0.01)
2a	1.197671	2.45072925
2b	1.388152	3.1181505
2c	1.6854225	3.43896225
2d	1.7820235	3.69890325
3a	4.88215825	2.7593645
3b	4.07637725	1.7011865
3c	6.94601175	6.598852
5a	3.05010875	3.08916475
5b	3.00358075	2.99686425
5c	3.37660375	3.40726325
6a	0.20835075	0.19673175
6b	3.25473225	3.270626
6c	0.1706815	0.16594675
6d	16.2538145	16.30323175
6e	0.21205	0.20861325
6f	16.47420375	16.54112575
7a	0.7371715	0.96912175
7b	0.717843	0.78762825
7c	132.8966747	80.66908625
8a	20.80986875	20.67298475
8b	3.4904135	3.90589375
8c	352.3121462	38.42666075
8d	320.533047	13.80165
9a	4.75234025	2.670492
9b	3.76353775	2.84001925
9c	30.24026375	16.18769225
9d	63.6688925	59.04935525
10a	2.9378275	2.921202
11a	0.63497875	0.62576775
11b	0.24231	0.21359875
11c	20.53614775	3.02464374

Table F.1: Full results from the experiment where count-min mean sketches are used for selectivity estimation. Execution time given in seconds.

Query	CMM (0.0001, 0.01)	CMM (0.001, 0.01)
11d	24.79246525	17.445123
15a	24.951208	26.44100024
15b	0.50455925	0.38762174
15c	26.731698	28.12791424
15d	25.022485	25.9797844
16a	0.5408345	0.52151924
16b	62.6908205	62.34864075
16c	5.19604225	5.185538
16d	4.10532575	4.053022
17a	41.2510825	40.83252975
17b	181.410215	179.7245527
17c	181.503286	179.9711144
17d	182.8048305	181.4477162
17e	43.04156875	42.76634175
17f	182.8655517	181.401201
19a	117.1735312	117.9057314
19b	11.2254525	11.2860775
19c	430.946747	435.687544
19d	1000.10052	1000.068411
20a	9.041787	9.07129475
20b	3.80392575	3.82192574
20c	2.93016325	2.98481524
21a	1.50685275	1.46353025
21b	1.053241	0.9601875
21c	2.46161525	2.59177774
23a	426.4736507	1.7320435
23b	0.872163	0.62852
23c	1000.463883	5.38797025
24a	82.899823	82.48641
24b	1.14394625	1.21350575
27a	3.249746	2.639471
27b	82.00266275	2.20942874
27c	3.61531175	3.48180024

Table F.1: Full results from the experiment where count-min mean sketches are used for selectivity estimation. Execution time given in seconds.

Query	**CMM (0.0001, 0.01)**	**CMM (0.001, 0.01)**
32a	0.132668	0.12157
32b	3.7260165	5.62407775

G Selectivity Estimates - Raw Data

Table G.1: Results from checking per-predicate selectivity on selected queries using count-min sketches compared to histogram and actual selectivity. CM(lg) $\rightarrow$ ($\epsilon = 0.000001$, $\delta = 0.001$), CM(md) $\rightarrow$ ($\epsilon = 0.0001$, $\delta = 0.01$), and CM(sm) $\rightarrow$ ($\epsilon = 0.001$, $\delta = 0.01$).

Query	**Predicate**	**Histogram**	**CM(lg)**	**CM(md)**	**CM(sm)**	**Actual**
19c	ci.note IN ('(voice)', '(voice: Japanese version)', '(voice) (uncredited)', '(voice: English version)')	4.000E-01	2.510E-02	2.511E-02	2.521E-02	2.510E-02
	cn.country_code ='[us]'	3.837E-01	3.837E-01	3.837E-01	3.837E-01	3.840E-01
	it.info = 'release dates'	1.000E-01	8.850E-03	8.850E-03	8.850E-03	8.850E-03
	(mi.info LIKE 'Japan:%200%' OR mi.info LIKE 'USA:%200%')	2.099E-01	2.099E-01	2.099E-01	2.099E-01	1.820E-02
	n.gender ='f'	2.311E-01	2.343E-01	2.343E-01	2.343E-01	2.340E-01
	n.name LIKE '%An%'	1.111E-01	1.111E-01	1.111E-01	1.111E-01	2.830E-01
	rt.role ='actress'	8.333E-02	8.333E-02	8.333E-02	8.333E-02	8.330E-02
	t.production_year >2000	6.579E-01	3.333E-01	3.333E-01	3.333E-01	6.360E-01
	t.id = mi.movie_id	2.412E-07	5.601E-07	6.375E-06	2.336E-06	2.110E-07
	t.id = mc.movie_id	4.688E-07	5.564E-07	3.683E-05	2.053E-06	2.110E-07
	t.id = ci.movie_id	3.101E-07	5.548E-07	3.101E-07	3.101E-07	2.110E-07
	mc.movie_id = ci.movie_id	4.688E-07	8.085E-07	4.688E-07	4.688E-07	4.620E-07
	mc.movie_id = mi.movie_id	4.688E-07	1.255E-06	8.404E-06	4.688E-07	9.140E-07
	mi.movie_id = ci.movie_id	3.101E-07	3.101E-07	7.296E-07	3.101E-07	4.790E-07
	cn.id = mc.company_id	3.129E-06	2.914E-06	3.832E-05	3.628E-04	2.800E-06

Table G.1: Results from checking per-predicate selectivity on selected queries using count-min sketches compared to histogram and actual selectivity. CM(lg) $\rightarrow$ ($\epsilon = 0.000001$, $\delta = 0.001$), CM(md) $\rightarrow$ ($\epsilon = 0.0001$, $\delta = 0.01$), and CM(sm) $\rightarrow$ ($\epsilon = 0.001$, $\delta = 0.01$).

Query	**Predicate**	**Histogram**	**CM(lg)**	**CM(md)**	**CM(sm)**	**Actual**
	it.id = mi.info_type_id	8.850E-03	8.850E-03	8.850E-03	8.850E-03	8.850E-03
	n.id = ci.person_id	1.733E-07	5.031E-07	5.016E-06	1.733E-07	1.570E-07
	rt.id = ci.role_id	8.333E-02	8.333E-02	8.333E-02	8.333E-02	8.330E-02
	n.id = an.person_id	1.191E-06	4.978E-07	3.681E-05	1.191E-06	1.570E-07
	ci.person_id = an.person_id	1.191E-06	1.144E-06	1.191E-06	7.543E-06	7.980E-07
19d	chn.id = ci.person_role_id	2.469E-07	4.970E-07	3.675E-06	2.469E-07	1.070E-07
	ci.note IN ('(voice)', '(voice: Japanese version)', '(voice) (uncredited)', '(voice: English version)')	4.000E-01	2.510E-02	2.511E-02	2.521E-02	2.510E-02
	cn.country_code ='[us]'	3.837E-01	3.837E-01	3.837E-01	3.837E-01	3.840E-01
	it.info = 'release dates'	1.000E-01	8.850E-03	8.850E-03	8.850E-03	8.850E-03
	n.gender ='f'	2.311E-01	2.343E-01	2.343E-01	2.343E-01	2.340E-01
	rt.role ='actress'	8.333E-02	8.333E-02	8.333E-02	8.333E-02	8.330E-02
	t.production_year >2000	6.579E-01	3.333E-01	3.333E-01	3.333E-01	6.360E-01
	t.id = mi.movie_id	2.412E-07	5.601E-07	6.375E-06	2.336E-06	2.110E-07
	t.id = mc.movie_id	4.688E-07	5.564E-07	3.683E-05	2.053E-06	2.110E-07
	t.id = ci.movie_id	3.101E-07	5.548E-07	3.101E-07	3.101E-07	2.110E-07
	mc.movie_id = ci.movie_id	4.688E-07	8.085E-07	4.688E-07	4.688E-07	4.620E-07
	mc.movie_id = mi.movie_id	1.733E-07	3.101E-07	8.404E-06	4.688E-07	9.140E-07
	mi.movie_id = ci.movie_id	3.101E-07	5.548E-07	7.296E-07	3.101E-07	4.790E-07

Table G.1: Results from checking per-predicate selectivity on selected queries using count-min sketches compared to histogram and actual selectivity. CM(lg) $\rightarrow$ ($\epsilon = 0.000001$, $\delta = 0.001$), CM(md) $\rightarrow$ ($\epsilon = 0.0001$, $\delta = 0.01$), and CM(sm) $\rightarrow$ ($\epsilon = 0.001$, $\delta = 0.01$).

Query	**Predicate**	**Histogram**	**CM(lg)**	**CM(md)**	**CM(sm)**	**Actual**
	cn.id = mc.company_id	3.129E-06	2.914E-06	3.832E-05	3.628E-04	2.800E-06
	it.id = mi.info_type_id	8.850E-03	8.850E-03	8.850E-03	8.850E-03	8.850E-03
	n.id = ci.person_id	1.733E-07	5.031E-07	5.016E-06	1.733E-07	1.570E-07
	rt.id = ci.role_id	8.333E-02	8.333E-02	8.333E-02	8.333E-02	8.330E-02
	n.id = an.person_id	1.191E-06	4.978E-07	3.681E-05	1.191E-06	1.570E-07
	ci.person_id = an.person_id	1.191E-06	1.144E-06	1.191E-06	7.543E-06	7.980E-07
17c	chn.id = ci.person_role_id;	2.469E-07	4.970E-07	3.675E-06	2.469E-07	1.070E-07
	k.keyword = 'character-name-in-title'	1.000E-01	4.226E-06	3.381E-05	2.874E-04	4.230E-06
	n.name LIKE 'X%'	1.111E-01	1.111E-01	1.111E-01	1.111E-01	7.950E-04
	n.id = ci.person_id	1.733E-07	5.031E-07	5.016E-06	1.733E-07	1.570E-07
	ci.movie_id = t.id	1.505E-06	5.548E-07	1.179E-06	3.101E-07	2.110E-07
	t.id = mk.movie_id	1.505E-06	5.646E-07	3.683E-05	3.063E-06	2.100E-07
	mk.keyword_id = k.id	7.499E-06	4.347E-06	3.922E-05	3.650E-04	1.700E-06
	t.id = mc.movie_id	4.688E-07	5.564E-07	3.683E-05	2.053E-06	1.700E-06
	mc.company_id = cn.id	3.129E-06	2.914E-06	3.832E-05	3.628E-04	1.700E-06
	ci.movie_id = mc.movie_id	4.688E-07	8.085E-07	4.688E-07	4.688E-07	4.620E-07
	ci.movie_id = mk.movie_id	1.505E-06	1.112E-06	1.179E-06	1.505E-06	7.580E-07
9d	mc.movie_id = mk.movie_id;	1.505E-06	2.049E-06	3.826E-05	2.115E-05	1.700E-06

Table G.1: Results from checking per-predicate selectivity on selected queries using count-min sketches compared to histogram and actual selectivity. CM(lg) $\rightarrow$ ($\epsilon = 0.000001$, $\delta = 0.001$), CM(md) $\rightarrow$ ($\epsilon = 0.0001$, $\delta = 0.01$), and CM(sm) $\rightarrow$ ($\epsilon = 0.001$, $\delta = 0.01$).

Query	**Predicate**	**Histogram**	**CM(lg)**	**CM(md)**	**CM(sm)**	**Actual**
	ci.note IN ('(voice)', '(voice: Japanese version)', '(voice) (uncredited)', '(voice: English version)')	4.000E-01	2.510E-02	2.511E-02	2.521E-02	2.510E-02
	cn.country_code ='[us]'	3.837E-01	3.837E-01	3.837E-01	3.837E-01	3.840E-01
	n.gender ='f'	2.311E-01	2.343E-01	2.343E-01	2.343E-01	2.340E-01
	rt.role ='actress'	8.333E-02	8.333E-02	8.333E-02	8.333E-02	8.330E-02
	ci.movie_id = t.id	4.688E-07	5.548E-07	5.016E-06	3.101E-07	2.110E-07
	t.id = mc.movie_id	4.688E-07	5.564E-07	3.683E-05	2.053E-06	2.110E-07
	ci.movie_id = mc.movie_id	4.688E-07	8.085E-07	3.683E-05	2.053E-06	4.620E-07
	mc.company_id = cn.id	3.129E-06	2.914E-06	3.832E-05	3.628E-04	2.800E-06
	ci.role_id = rt.id	8.333E-02	8.333E-02	8.333E-02	8.333E-02	8.330E-02
	n.id = ci.person_id	1.733E-07	5.031E-07	5.016E-06	7.543E-06	1.570E-07
	chn.id = ci.person_role_id	2.469E-07	4.970E-07	3.675E-06	2.469E-07	1.070E-07
	an.person_id = n.id	1.191E-06	4.978E-07	3.681E-05	1.191E-06	7.980E-07
3a	an.person_id = ci.person_id;	1.191E-06	1.144E-06	1.191E-06	7.543E-06	7.980E-07
	k.keyword LIKE '%sequel%'	1.111E-01	1.111E-01	1.111E-01	1.111E-01	1.730E-04

Table G.1: Results from checking per-predicate selectivity on selected queries using count-min sketches compared to histogram and actual selectivity. CM(lg) $\rightarrow$ ($\epsilon = 0.000001$, $\delta = 0.001$), CM(md) $\rightarrow$ ($\epsilon = 0.0001$, $\delta = 0.01$), and CM(sm) $\rightarrow$ ($\epsilon = 0.001$, $\delta = 0.01$).

Query	**Predicate**	**Histogram**	**CM(lg)**	**CM(md)**	**CM(sm)**	**Actual**
	mi.info IN ('Sweden', 'Norway', 'Germany', 'Denmark', 'Swedish', 'Denish', 'Norwegian', 'German')	5.000E-01	7.775E-03	7.866E-03	8.939E-03	7.780E-03
	t.production_year >2005	5.320E-01	3.333E-01	3.333E-01	3.333E-01	5.360E-01
	t.id = mi.movie_id	2.412E-07	5.601E-07	6.375E-06	2.336E-06	2.110E-07
	t.id = mk.movie_id	1.505E-06	5.646E-07	3.683E-05	3.063E-06	2.100E-07
	mk.movie_id = mi.movie_id	1.505E-06	2.365E-06	1.505E-06	3.081E-06	2.020E-06
2b	k.id = mk.keyword_id;	7.499E-06	4.347E-06	3.922E-05	3.650E-04	4.230E-06
	cn.country_code ='[nl]'	1.074E-02	1.074E-02	1.074E-02	1.074E-02	1.070E-02
	k.keyword = 'character-name-in-title'	1.000E-01	4.347E-06	3.922E-05	3.650E-04	4.230E-06
	cn.id = mc.company_id	3.129E-06	2.914E-06	3.832E-05	3.628E-04	2.800E-06
	mc.movie_id = t.id	4.688E-07	5.564E-07	3.683E-05	2.053E-06	2.100E-07
	t.id = mk.movie_id	1.505E-06	5.646E-07	3.683E-05	3.063E-06	2.100E-07
	mk.keyword_id = k.id	7.499E-06	4.347E-06	3.922E-05	3.650E-04	4.230E-06
	mc.movie_id = mk.movie_id;	1.505E-06	2.049E-06	3.826E-05	2.115E-05	1.700E-06

Table G.2: Results from checking per-predicate selectivity on selected queries using count-min mean sketches compared to histogram and actual selectivity. CMM(lg) $\rightarrow$ ($\epsilon = 0.0001$, $\delta = 0.01$), and CMM(sm) $\rightarrow$ ($\epsilon = 0.001$, $\delta = 0.01$).

Query	**Predicate**	**Histogram**	**CMM(lg)**	**CMM (sm)**	**Actual**
19c	ci.note IN ('(voice)', '(voice: Japanese version)', '(voice) (uncredited)', '(voice: English version)')	4.000E-01	2.511E-02	2.521E-02	2.510E-02
	cn.country_code ='[us]'	3.837E-01	3.837E-01	3.837E-01	3.840E-01
	it.info = 'release dates'	1.000E-01	8.850E-03	8.850E-03	8.850E-03
	(mi.info LIKE 'Japan:%200%' OR mi.info LIKE 'USA:%200%')	2.099E-01	2.099E-01	2.099E-01	1.820E-02
	n.gender ='f'	2.311E-01	2.343E-01	2.343E-01	2.340E-01
	n.name LIKE '%An%'	1.111E-01	1.111E-01	1.111E-01	2.830E-01
	rt.role ='actress'	8.333E-02	8.333E-02	8.333E-02	8.330E-02
	t.production_year >2000	6.579E-01	3.333E-01	3.333E-01	6.360E-01
	t.id = mi.movie_id	2.412E-07	6.370E-08	4.700E-08	2.110E-07
	t.id = mc.movie_id	4.688E-07	6.610E-08	2.077E-07	2.110E-07
	t.id = ci.movie_id	3.101E-07	6.540E-08	2.571E-07	2.110E-07
	mc.movie_id = ci.movie_id	4.688E-07	3.113E-07	4.350E-07	4.620E-07
	mc.movie_id = mi.movie_id	4.688E-07	7.343E-07	3.413E-07	9.140E-07
	mi.movie_id = ci.movie_id	3.101E-07	3.248E-07	3.413E-07	4.790E-07
	cn.id = mc.company_id	3.129E-06	1.947E-06	1.458E-06	2.800E-06

Table G.2: Results from checking per-predicate selectivity on selected queries using count-min mean sketches compared to histogram and actual selectivity. CMM(lg) $\rightarrow$ ($\epsilon = 0.0001$, $\delta = 0.01$), and CMM(sm) $\rightarrow$ ($\epsilon = 0.001$, $\delta = 0.01$).

Query	**Predicate**	**Histogram**	**CMM(lg)**	**CMM (sm)**	**Actual**
	it.id = mi.info_type_id	8.850E-03	8.809E-03	8.485E-03	8.850E-03
	n.id = ci.person_id	1.733E-07	5.130E-08	1.330E-07	1.570E-07
	rt.id = ci.role_id	8.333E-02	8.330E-02	8.300E-02	8.330E-02
	n.id = an.person_id	1.191E-06	5.890E-08	2.083E-07	1.570E-07
	ci.person_id = an.person_id	1.191E-06	7.036E-07	8.264E-07	7.980E-07
19d	chn.id = ci.person_role_id	2.469E-07	3.675E-06	2.469E-07	1.070E-07
	ci.note IN ('(voice)', '(voice: Japanese version)', '(voice) (uncredited)', '(voice: English version)')	4.000E-01	2.511E-02	2.521E-02	2.510E-02
	cn.country_code ='[us]'	3.837E-01	3.837E-01	3.837E-01	3.840E-01
	it.info = 'release dates'	1.000E-01	8.850E-03	8.850E-03	8.850E-03
	n.gender ='f'	2.311E-01	2.343E-01	2.343E-01	2.340E-01
	rt.role ='actress'	8.333E-02	8.333E-02	8.333E-02	8.330E-02
	t.production_year >2000	6.579E-01	3.333E-01	3.333E-01	6.360E-01
	t.id = mi.movie_id	2.412E-07	6.370E-08	4.700E-08	2.110E-07
	t.id = mc.movie_id	4.688E-07	6.610E-08	2.077E-07	2.110E-07
	t.id = ci.movie_id	3.101E-07	6.540E-08	2.571E-07	2.110E-07
	mc.movie_id = ci.movie_id	4.688E-07	3.113E-07	4.350E-07	4.620E-07
	mc.movie_id = mi.movie_id	1.733E-07	7.343E-07	6.985E-07	9.140E-07
	mi.movie_id = ci.movie_id	3.101E-07	3.248E-07	3.413E-07	4.790E-07

Table G.2: Results from checking per-predicate selectivity on selected queries using count-min mean sketches compared to histogram and actual selectivity. CMM(lg) $\rightarrow$ ($\epsilon = 0.0001$, $\delta = 0.01$), and CMM(sm) $\rightarrow$ ($\epsilon = 0.001$, $\delta = 0.01$).

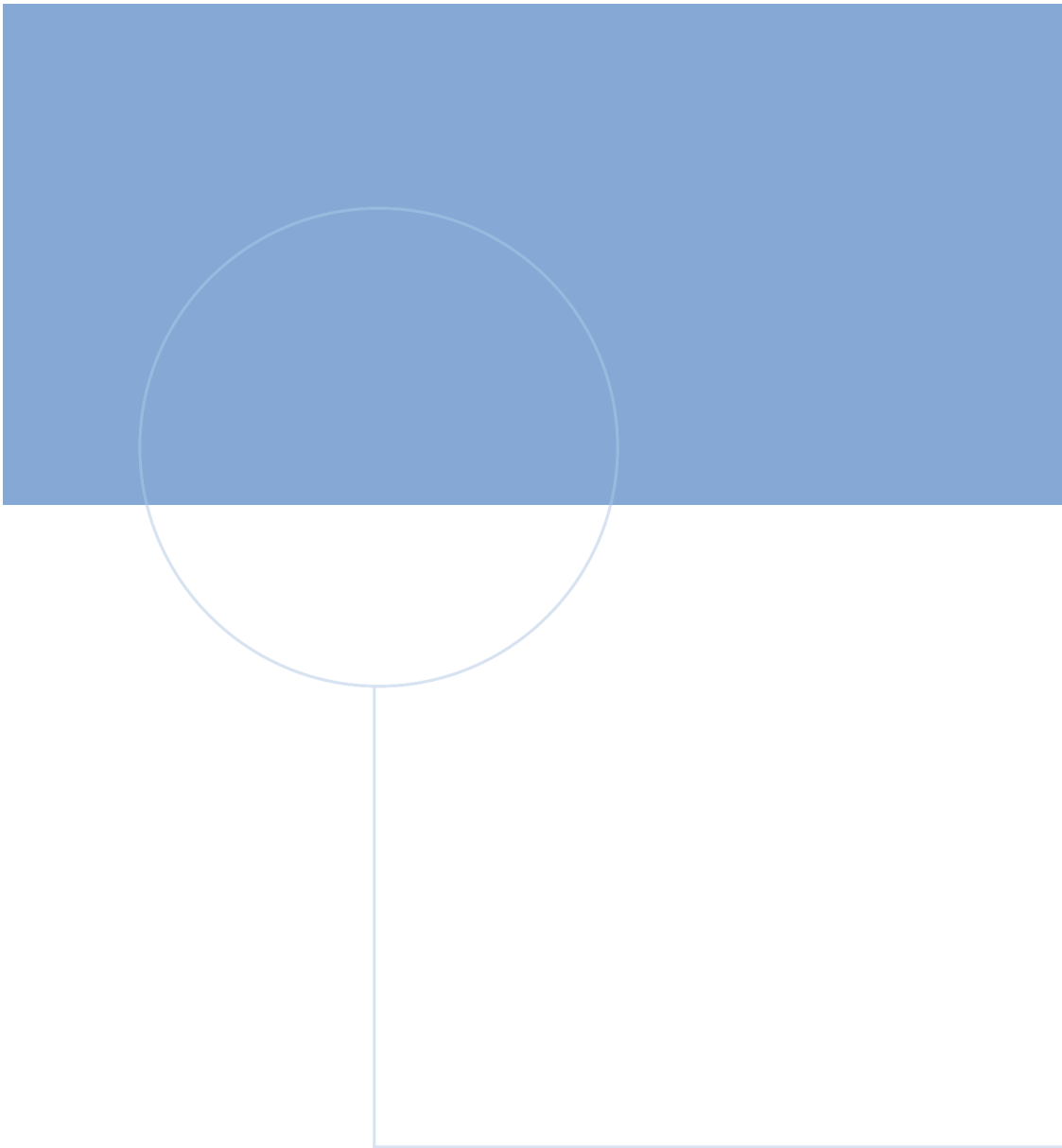
Query	**Predicate**	**Histogram**	**CMM(lg)**	**CMM (sm)**	**Actual**
	cn.id = mc.company_id	3.129E-06	1.947E-06	1.458E-06	2.800E-06
	it.id = mi.info_type_id	8.850E-03	8.809E-03	8.485E-03	8.850E-03
	n.id = ci.person_id	1.733E-07	5.130E-08	1.191E-06	1.570E-07
	rt.id = ci.role_id	8.333E-02	8.330E-02	8.300E-02	8.330E-02
	n.id = an.person_id	1.191E-06	5.890E-08	2.083E-07	1.570E-07
	ci.person_id = an.person_id	1.191E-06	7.036E-07	8.264E-07	7.980E-07
17c	chn.id = ci.person_role_id;	2.469E-07	3.675E-06	2.469E-07	1.070E-07
	k.keyword = 'character-name-in-title'	1.000E-01	3.381E-05	3.293E-06	4.230E-06
	n.name LIKE 'X%'	1.111E-01	1.111E-01	1.111E-01	7.950E-04
	n.id = ci.person_id	1.733E-07	5.130E-08	1.330E-07	1.570E-07
	ci.movie_id = t.id	1.505E-06	6.540E-08	2.571E-07	2.110E-07
	t.id = mk.movie_id	1.505E-06	6.900E-08	1.584E-07	2.100E-07
	mk.keyword_id = k.id	7.499E-06	3.911E-06	3.293E-06	1.700E-06
	t.id = mc.movie_id	4.688E-07	6.610E-08	2.077E-07	1.700E-06
	mc.company_id = cn.id	3.129E-06	1.947E-06	1.458E-06	1.700E-06
	ci.movie_id = mc.movie_id	4.688E-07	3.113E-07	4.350E-07	4.620E-07
	ci.movie_id = mk.movie_id	1.505E-06	5.961E-07	7.045E-07	7.580E-07
9d	mc.movie_id = mk.movie_id;	1.505E-06	1.506E-06	1.503E-06	1.700E-06

Table G.2: Results from checking per-predicate selectivity on selected queries using count-min mean sketches compared to histogram and actual selectivity. CMM(lg) $\rightarrow$ ($\epsilon = 0.0001$, $\delta = 0.01$), and CMM(sm) $\rightarrow$ ($\epsilon = 0.001$, $\delta = 0.01$).

Query	**Predicate**	**Histogram**	**CMM(lg)**	**CMM (sm)**	**Actual**
	ci.note IN ('(voice)', '(voice: Japanese version)', '(voice) (uncredited)', '(voice: English version)')	4.000E-01	2.511E-02	2.521E-02	2.510E-02
	cn.country_code ='[us]'	3.837E-01	3.837E-01	3.837E-01	3.840E-01
	n.gender ='f'	2.311E-01	2.343E-01	2.343E-01	2.340E-01
	rt.role ='actress'	8.333E-02	8.333E-02	8.333E-02	8.330E-02
	ci.movie_id = t.id	4.688E-07	6.540E-08	2.571E-07	2.110E-07
	t.id = mc.movie_id	4.688E-07	6.610E-08	2.077E-07	2.110E-07
	ci.movie_id = mc.movie_id	4.688E-07	3.113E-07	4.350E-07	4.620E-07
	mc.company_id = cn.id	3.129E-06	1.947E-06	1.458E-06	2.800E-06
	ci.role_id = rt.id	8.333E-02	8.330E-02	8.300E-02	8.330E-02
	n.id = ci.person_id	1.733E-07	7.036E-07	1.330E-07	1.570E-07
	chn.id = ci.person_role_id	2.469E-07	3.675E-06	2.469E-07	1.070E-07
	an.person_id = n.id	1.191E-06	5.890E-08	2.083E-07	7.980E-07
3a	an.person_id = ci.person_id;	1.191E-06	7.036E-07	8.264E-07	7.980E-07
	k.keyword LIKE '%sequel%'	1.111E-01	1.111E-01	1.111E-01	1.730E-04

Table G.2: Results from checking per-predicate selectivity on selected queries using count-min mean sketches compared to histogram and actual selectivity. CMM(lg) $\rightarrow$ ($\epsilon = 0.0001$, $\delta = 0.01$), and CMM(sm) $\rightarrow$ ($\epsilon = 0.001$, $\delta = 0.01$).

Query	**Predicate**	**Histogram**	**CMM(lg)**	**CMM (sm)**	**Actual**
	mi.info IN ('Sweden', 'Norway', 'Germany', 'Denmark', 'Swedish', 'Denish', 'Norwegian', 'German')	5.000E-01	7.866E-03	8.939E-03	7.780E-03
	t.production_year >2005	5.320E-01	3.333E-01	3.333E-01	5.360E-01
	t.id = mi.movie_id	2.412E-07	6.370E-08	4.700E-08	2.110E-07
	t.id = mk.movie_id	1.505E-06	6.900E-08	1.584E-07	2.100E-07
	mk.movie_id = mi.movie_id	1.505E-06	1.813E-06	1.777E-06	2.020E-06
2b	k.id = mk.keyword_id;	7.499E-06	3.911E-06	3.293E-06	4.230E-06
	cn.country_code ='[nl]'	1.074E-02	1.074E-02	1.074E-02	1.070E-02
	k.keyword = 'character-name-in-title'	1.000E-01	3.381E-05	2.874E-04	4.230E-06
	cn.id = mc.company_id	3.129E-06	1.947E-06	1.458E-06	2.800E-06
	mc.movie_id = t.id	4.688E-07	6.610E-08	2.077E-07	2.100E-07
	t.id = mk.movie_id	1.505E-06	6.900E-08	1.584E-07	2.100E-07
	mk.keyword_id = k.id	7.499E-06	3.911E-06	3.293E-06	4.230E-06
	mc.movie_id = mk.movie_id;	1.505E-06	1.506E-06	1.503E-06	1.700E-06



NTNU

Norwegian University of
Science and Technology