Master's thesis

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

Jonatan Sissener Engstad

Selectivity Estimation for JSON Data in MySQL

Master's thesis in Computer Science
Supervisor: Norvald H. Ryeng
July 2023

NTNU
Norwegian University of
Science and Technology

Jonatan Sissener Engstad

Selectivity Estimation for JSON Data in MySQL

Master's thesis in Computer Science
Supervisor: Norvald H. Ryeng
July 2023

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

Sammendrag

Databaser er en av grunnsteinene i vår moderne digitaliserte virkelighet. Deres oppgave er ikke bare å lagre data, men også å servere data og svare på spørring – og dette på mest mulig effektivt vis. De siste to tiårene har et nytt format for datakommunikasjon og lagring, JSON ("JavaScript Object Notation") hatt sitt inntog i databaseverden og mange av de store relasjonelle databaseaktørene har de siste årene lagt til støtte for dette formatet. En av disse databasene er MySQL. For å effektivt kunne svare på store, komplekse spørringer er det vitalt å ha grundig og god databasestatistikk – altså oversikt over verdiene som finnes ligger i kolonnene og tabellene i databasen. Men for JSON, som er mer komplekst og dynamisk enn de datatypene som vanligvis brukes i relasjonelle databaser finnes det ingen allmen enighet om hvordan denne statistikken burde se ut. I MySQL's tilfelle betyr dette at databasen ikke har tilgang til statistikk for JSON i det hele tatt, noe som fører til tap av ytelse i tilfeller der god statistikk ville utgjort en forskjell. Denne oppgaven undersøker eksisterende tilnærminger til innsamling av statistikk for JSON data, for å finne noe som kunne vært implementert i MySQL. Den beskriver så en implementasjon av et slikt system samt presenterer resultater av målinger som er gjort av resulterende endringer i ytelse og nøyaktighet av estimat.

Abstract

Databases are one of the cornerstones of our digital society. Their task is not only to store data but also to serve it by responding to user queries. And they must do so as efficiently as possible. A rise in the use of the JSON ("JavaScript Object Notation") data format over the last decades, has led to many of the major relational database vendors adding support for this format. One of these databases is MySQL. To be able to effectively respond to large, complex queries, it is vital to have accurate and up-to-date database statistics – that is, an overview of the values that exist within the columns and tables of the database. For JSON however, a format that is more complex and dynamic than the data types typically used in relational databases, there is no broad consensus on how these statistics should be gathered.

In the case of MySQL, the database has no access to statistics for JSON data at all, resulting in a theoretical performance loss in cases where good statistics would have made a difference. This master's thesis' aim is to investigate existing approaches to collecting statistics for JSON data to find something that could be implemented in MySQL. It will then attempt an implementation of such a system. Finally, the goal is to perform measurements of the implementation's impact on accuracy and performance, to whether further investigation into the topic could be warranted.

Acknowledgements

Throughout the work on this thesis, I have received assistance without which I would never have gotten this far. First and foremost, I would like to extend my gratitude to my supervisor Norvald H. Ryeng. Without his advice and guidance, none of this work would have been possible.

I would also like to thank Mia Treland for all her love and support, and for helping keep me focused on the task at hand.

Finally, I would like to thank Julie Elisabeth Takacs at NTNU's IE faculty for helping me, at night, during her vacation, to get the deadline extension needed to finish this thesis.

Table of Contents

List of Figures

List of Tables

1 Introduction

Databases are a cornerstone of modern computing and digital infrastructure, tasked with the responsibility of storing much of the worlds information records. A core part of this responsibility is data retrieval. Many things must be done right for a database to have good retrieval speeds. Clever use of algorithms and data structures is crucial, as is proper management of main memory and of processing power. In the context of relational database systems especially, one cannot – excepting the very simplest of queries – achieve good query performance without a good query optimizer.

SQL queries, being written in a declarative language, specify only the expected result of a query, providing no information about how the database is expected to complete its task[1]. The task of figuring out how best to execute the query is instead given to the database. Specifically, the database's query optimizer. Query optimization is a hard problem and the optimizer is faced with both time constraints, and a massive search space. To this end, the query optimizer relies on a slew of clever techniques to prune the search space, save on redundant computations and quickly rank alternate execution plans. Central to the query optimizer's ability to make good predictions on which query plans will perform well are *database statistics*.

Database statistics is a general term used to refer to metadata on the contents of the tables and columns in the database. In its simplest form, database statistics is the row count of a table, but modern database systems usually rely on much more complex statistical information such as histograms that approximate the distribution of values within a column. These advanced forms of statistics are not always available for all data types however, negatively impacting the query optimizer's ability to reason about the behavior and attributes of columns using these data types. One such case is the JSON datatype, a relatively fresh type in the relational database world, but also one with a quickly growing mind- and storage space share.

This thesis is meant as an exploration of the solutions to the challenge of collecting viable statistics for JSON data in databases. Specifically, the thesis aims to survey existing approaches to the problem, and then implement a prototype system for gathering JSON data statistics in the MySQL database based on the findings of the survey. Experiments will then be performed to examine the viability of the chosen approach, measuring gains in accuracy and performance and cost in terms of storage and speed.

1.1 Background and Motivation

Currently, there is no broadly accepted approach to collecting database statistics for JSON data. And due to the relatively recent addition of the datatype to most RDBMS systems, few existing systems support the collection of statistics for JSON columns. One of these databases is MySQL, which not only does not support the creation of statistics for JSON columns, but also does not use heuristics to make approximations for JSON data.

The work of this master's thesis is based on the hypothesis that MySQL's handling of JSON data can be improved. And while there is no commonly accepted way to solve this challenge, research efforts into solving the problem have been made and their approach and results publicized. Additionally, some RDBMS systems have implemented solutions that

[1]This is no longer completely true for modern databases, which things like hints to of influence the decisions of the query optimizer

either help solve or circumvent the issue of JSON column statistics. This previous work is discussed in greater detail in section 3. We expect that through inspection of previous work, a solution suitable for MySQL will be found. The goal is then to implement a prototype of this solution as a proof-of-concept of the method, and to report the impact it has on the accuracy of MySQL's estimates for predicates over JSON data.

1.2 Goals, Research Questions and Scope

The core goal of this thesis is to find a viable and sensible approach to improving selectivity estimates for JSON data in MySQL. To this end a survey of existing methods and approaches to collecting statistics for JSON data in and outside of databases will be made. These approaches will then be judged by their applicability in the context of an implementation for MySQL. If one or more promising candidates are found, one of them, or a synthesis of multiple approaches, will be implemented as prototype in MySQL. If no good candidates are found, the task will instead be to attempt to find a novel approach to the problem which is suitable for implementation in MySQL. Finally, the aim of the project is to implement a prototype of the chosen method in MySQL, and to evaluate its performance. It is hoped that the work done for this thesis along with this report can serve as a resource for those doing further work on implementing a proper solution to the problem.

The goal of this report is to document the process, findings and outcome of this work. As such, the report aims to answer the following *research questions* (RQs):

RQ1 What are the existing, published methods for collecting and using statistics for JSON data in an RDBMS context, which also fit well with MySQL's design and architecture?

RQ2 Can a working system for collecting and using JSON statistics realistically be implemented into MySQL?

RQ3 With such a system implemented, are MySQL's estimates for the selectivity of predicates over JSON expressions improved in a meaningful way?

RQ4 How would a system for JSON statistics, if introduced to MYSQL, affect MySQL's performance when dealing with JSON data?

Finally, it is important to discuss what the goals of the project are not, and which questions will not be explored in this report.

First and foremost, is is important to state that this project is an exploration. Because so much is unknown about the work required for this project, its goal is not to implement the very best and most finetuned and perfected solution. The goal is to find one approach that seems promising and then to implement a simple, working prototype to get some preliminary results that show whether the approach is promising or not.

The project will not, for example, explore which specific histogram types (beyond a common few) are optimal, nor will it attempt to squeeze as much performance out of the system as possible. Some obvious and maybe simple optimizations will likely be left on the table.

1.3 Work and Contributions

To meet the abovementioned goals and to answer the stated research questions, the following work was performed:

Initially, a survey was conducted of earlier published work related to the topic at JSON statistics and selectivity estimation for JSON data. Both research and commercial systems were examined. A summary of the findings of this work can be found in section 3.

Two prototype implementations of the chosen solution were than created. First, a prototype was created in Python, with the goal of generating statistics and measuring the degree of accuracy that they provided. Second, a MySQL prototype was created, making use of the statistics generated by the Python prototype to provide MySQL's query optimizer with improved estimates for filtering and join predicates over JSON data.

Finally, experiments were performed with both prototypes. The experiments performed with the Python prototype focus on how changing various parameters of the statistics collection algorithm affect the accuracy that the statistics provide. The experiments performed with the MySQL prototype examine the impact that the added statistics have on MySQL, both in terms of how much they boost the accuracy of estimates given to the query optimizer, and in terms of the impact that the changed estimates have on MySQL's performance.

1.4 Thesis Structure

The first section after the introduction is the "Background" section. It presents material helpful for the unfamiliar reader to more easily grasp the later contents of the report. The next section, "Related Work", will then explore earlier published work on the topic of gathering and using JSON statistics in a database context. Section 4 ("Implementation") describes the work that went into implementing the two prototypes. The next two sections, "Python Experiments" and "MySQL Experiments" then present the experiments and experimental results for each of the two prototypes. These sections also contain a discussion of the experimental findings. Finally, section 7, "Conclusion" concludes the report.

2 Background

This chapter presents the background knowledge that is foundational to the work in this thesis. All concepts presented here will be referred to in later chapters and assumed to be known. A basic familiarity with general computer science, programming and database concepts is also assumed.

The chapter starts with two sections that explain the motivating forces behind this thesis: section 2.1 covers the motivation behind and basics of query optimization in databases. Section 2.2 discusses the topic of selectivity estimation in query optimization in greater detail. Next, section 2.3 covers the basics of JSON, the data format that this thesis focuses on. Familiarity with key concepts and terminology related to JSON is vital to understanding the rest of this thesis, and unfamiliar readers are recommended to at a minimum be comfortable with the concepts presented in this section. Finally, section 2.4 discusses the MySQL database system, providing a brief overview of its relevant features

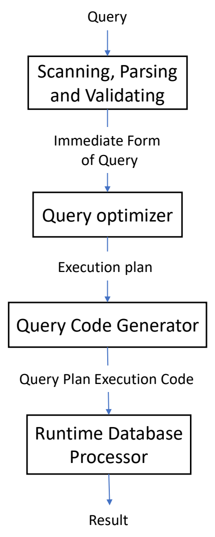

Query
Scanning, Parsing and Validating
Immediate Form of Query
Query optimizer
Execution plan
Query Code Generator
Query Plan Execution Code
Runtime Database Processor
Result

Figure 1: The query processing pipeline as presented in Elmasri and Navathe [10].

and an introduction to the concepts and techniques used extending the MySQL codebase.

2.1 Query Optimization

Databases are highly optimized storage systems for digital data, designed to safeguard everything from medical records and financial data, to recipes for vegan quinoa pancakes, to weather data – typically providing very strong storage guarantees. But databases commonly provide more than just storage functionaly. Support for inserting, updating and, crucially to this project, *retrieving* data is also at the core of the services provided. As it turns out, there are quite a few ways to go about retrieving data in a modern database system, especially as the queries grow more complex. Because retrieval performance is a key comptetitive area for database systems, the envelope is constantly being pushed in terms of latency, bandwidth and scale. And core to achieving good retrieval performance is *query optimizaiton.*

Query optimization is a process that occurs in the context of the *query pipeline* (see figure 1). The query pipeline describes the process that a query goes through, from being accepted by the database systems to being executed and retrieving and return data to the client. The first step in the query processing pipeline is to receive and validate (both syntactically and semantically) the query statement. At this stage, the access rights of the client are also confirmed. The query is then rewritten into a canonical form, so that expectations that further stages have of the query structure hold true. The query is then ready for query optimization. The query optimization stage, which will be explained in much greater detail soon, has as its goal to find the most efficient way to answer the query's request, i.e., to retrieve the data being asked for. The query optimization stage produces as its output a *query plan* – a description of how of the data will be retrieved. The final stage is query plan execution. As the name implies, at this stage the query plan produced by the optimizer is interpreted (or compiled) and executed, resulting in actual data being retrieved (and returned to the client).

To understand the massive impact that the query plan can have on database performance, one must understand which aspects of the retrieval process it controls and the impact that these aspects have. First, the query plan determines how the base data is retrived. There are two main approaches to this: doing a full table scan, or doing an index based lookup. Depending on the query, one of these can be much more expensive than the other. For example, using an index for a point lookup can be several orders of magnitude cheaper than doing a table scan if scanning all the data is expensive enough (slow disk, lots of data, ...). Another example of an impactful decision made by the query optimizer is join ordering. In general, it is good for performance to first perform joins that will result in smaller result sets, while saving the larger joins for last. Doing the opposite can have a severely negative impacts on query performance. Many more aspects of the query plan can have large performance impacts. These include (but are not limited to): join methods, group by methods, sort ordering, and more [10].

For complex queries, the execution time of the worst possible query plan can easily be thousands of times worse than the best possible plan. And while modern query optimizers have gotten pretty good, they still struggle with finding the most optimal plans for many complex queries [19].

The 1979 System R paper by Selinger *et. al* is often referred to as the "bible" of the field of query optimization[32]. In it, Selinger and her co-authors and colleagues at IBM detail many approaches to query optimization which since then have come to constitute the foundation that other database developers build their query optimizer upon. Among the techniques presented in the seminal paper were: using dynamic programming for query optimization, using heuristics values to estimate selectivity of various predicate functions, as well as a cost-based model which was used to evaluate and compare the (assumed) performance of the generated plans.

Compared to todays database systems, the System R query optimizer is quite rudimentary. For example, the System R and other early database system optimizers relied heavily on heuristics to estimate the size of intermediate relations. The only information they had about the actual size of the relations was table and index cardinalities. When filtering predicates and join conditions were applied to these relations, these estimates could quickly become very wrong. The System R cost model was also very simple, assuming the cost of reading a tuple was the same for all relations in all situations [5, 32]. Modern databases on the other hand, while still building on those same fundamentals, have expanded on these ideas to create much more complex and accurate systems. Modern *cost-based* optimizers feature much more complex cost models that take into account how much of a relation is currently in main memory, disk vs SSD I/O costs, computation costs and intermediate storage costs, and more[10]. In addition, they feature much more advanced techniques for estimating the size of relations, both for base tables as well as intermediate tables. The use of such advanced techniques, as well as simpler approaches like the heuristics featured in System R, is collectively referred to as *selectivity estimation.*

2.2 Selectivity estimation

When modern cost-based optimizers calculate the cost of executing a query plan, there are two main inputs. The first is information about where the data that the plan will be accessing is placed, the size of records, and other such information, plus information on buffer sizes and such. The second is an estimate of the cardinality (the number of records) of every relation, both intermediate and otherwise, involved in the plan. All this information is then fed into the query optimizer's *cost model*, which assigns an estimated cost to every operation involved in the plan. The plan with the lowest calculated cost is then chosen for execution. Each of these components play a vital role in the optimizer. But while having a good cost models is vital to the query optimizer, given a working cost model, good cardinality estimates have been found to have a much greater potential impact on performance [19].

There are many approaches and techniques that can be used to estimate cardinalities in databases. In general, we refer to all these sources used for cardinality estimation as *statistics*. Perhaps the simplest statistic, and one that is almost universally available, is the number of tuples in a table. Further, when indexes exist on a column, the index itself can be used to estimate the number of tuples that will satisfy predicates on that column (usually – this is dependant on the type of index and the predicate). However, these two sources of statistics are only enough to give estimates for the simplest (or the most contrived) queries. The database catalog, which stores various metadata on the tables and columns in the database, can often provide further statistics for use in query optimization. For example, some catalogs store the number of distinct values (”NDV”) per column. However, for predicates that filter on (e.g.,) equality with a specific value, more specifific statistics are needed. While an index could provide estimates for this exact example, indexes have their own drawbacks – they can take up a large amount of space and must be updated on every change to the column data.

2.2.1 Histograms

A more lightweight data structure whose explicit goal is to provide more accurate database statistics is the *histogram*. Histograms provide reasonably accurate information on the distribution of values within (usually) a single column at a relatively low cost compared to indexes [10]. A standard assumption within databases is that the distribution of values is completely uniform. And while this is a reasonable assumption to make provided nothing more is known about the data, it is often absolutely wrong. Database histograms, like their commonly seen data visualization aid namesake, divide the complete range of values into sub-ranges (*buckets*) and track the number of values that fall into each. Thus, histograms give the query optimizer much more accurate information about the true distribution of the data. Within each bucket however, the assumption of uniformity is still usually applied.

Size is a limiting factor when it comes to histograms, negatively impacting lookup speeds and taking up space that could be used for other operations. Therefore, histograms must attempt to convey as much information about the value distribution as possible while minimizing the amount of space taken up. What this usually means in practice is that the number of buckets that histograms are allowed to use is bounded. In MySQL for example, histograms may not contain more than 1024 buckets [15].

Many different types and variations on histograms are used in modern databases. Perhaps the most commonly used type (although the exact implementation differs) is the *equi-height* histogram. Equi-height histograms attempt to size the buckets in such a way that each bucket holds approximately the same amount of values (i.e., the same total count). This is considered advantageous because it tends to avoid keeping multiple frequently occurring values in the same bucket. It also tends to put more of the less frequently occurring values in the same bucket. Another common type of histogram is the *singleton* histogram. Singleton histogram buckets, instead of representing a range of values, keep only the count for a single value. This is very useful in cases where there are few distinct values in a column such that a separate bucket can be kept for every single value. A final type of histogram which is relevant to this specific project is a variation of the singleton histogram which stores only the most frequently occurring values in buckets. The Oracle database implements this histogram under the name ”top frequency histogram” [7].

Histograms in databases is a deep topic, and this introduction to the topic only covers a small part of the field. For a more in-depth introduction to the topic, see *Building Query Compilers (Under Construction)* by Moerkotte.

2.2.2 Notes on Database Statistics

When discussing improvements in selectivity estimation and database statistics, one must keep in mind that while selectivity estimates are of a continuous (as far as computers can represent continuous data, that is) nature, the output of the query optimizer is very much discrete – one plan from a finite set of plans. This means that large improvements in selectivity estimation may lead to no change in the choice of plan if the change in the final calculations in the cost model do not change enough for some threshold to be crossed. On the other hand, even a tiny change in estimates may have a massive, outsized impact if the change is just enough to make the cost of a radically different plan to become cheapest.

Additionally, query optimizers are far from perfect, and even a large improvements in estimate accuracy may have zero impact on the actual performance of the database. In fact, improved selectivity estimates can even lead to worse performance in some cases. A database optimizer optimizer may be caused by some ”happy accident” to make better plan choices when provided with worse / less granular data, and then make more rational but worse choices when provided with more accurate data.

Query optimizers are complex pieces of machinery with many ”moving” parts that are, despite large advancements in complexity, still based on heuristics and assumptions about how computers, networks and the data work and fit together. Optimally handling all the complex scenarios and edge cases that can arise in database queries with these assumptions is simply not possible. In addition, query optimizers may have blind spots, bugs, half-implemented features and more that lead it to making poor choices when certain conditions are met. This does not mean that query optimizers cannot be improved however. And their improvement, rather than better cost models, in large part is dependent on making more accurate estimates of the size of query relations – a challenge that database statistics is a very good fit for [19].

2.2.3 The Assumption of Statistical Independence

The assumption of a uniform distribution of values within columns is not the only common assumption made by database systems which oftentimes leads to a poor choice of query plans. Filtering predicates seldom appear alone – and when multiple predicates appear it is not unusual for them to in some way be semantically (in terms of the data’s domain) related. For example in the case of the Airbnb data, the number of beds in an apartment and the number of people a the apartment can accommodate are quantities that are strongly related. Most databases don’t collect information on the correlations between the different columns of the a database however, at least not automatically. This makes sense – just the number of column pairs quickly balloons to an unmanageable size. When faced with no way to gleam any information on the relations between the different predicates

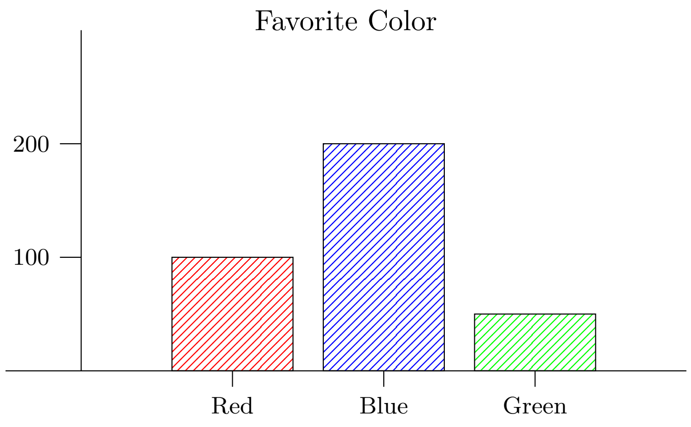


Figure 2: Histogram showing a (hypothetical) group of people's favorite color.

of a query, the default action of most databases is to assume that all query predicates are statistically independent.

This assumption can quickly lead to severe underestimations. In an extreme case, one might have a set of predicates, all of which in practice check the same semantic property and thus have a combined selectivity about equal to the selectivity of just one of the predicates alone. The assumption of independence however will result in a calculated selectivity equal to the product of the selectivity of each of the predicates. Of course, correlations between predicates may also be much more subtle. In the Airbnb data for example, there is a correlation between the number of beds in a listing and the latitude and longitude values of the listing because certain areas of Los Angeles typically have larger houses than others.

There exist several solutions to the problem. PostgreSQL for example, allows users to order the creation of an advanced multivariate statistic referred to as *functional dependencies* which encodes the correlation between two columns. Combining this with PostgreSQL's most-common values statistic allows for the collection of correlation statistics for pairs of values from the two columns in question [36, 26]. Another, cheaper solution is to use a damping factor [19, 2]. With damping, selectivity is adjusted upward as the number of predicates in a query grows to account for the probability of independence growing smaller.

2.3 JSON

JavaScript Object Notation, or JSON as it is commonly referred to, is a plaintext human-readable data serialization, transfer and storage format. As the name implies, the format is heavily inspired by the notation used for object literals in the Javascript language. Despite having the name of a specfic programming language embedded in its name, JSON is intended to function as a universal data-interchange format.

JSON is built on two fundamental data structures. First is the unordered collection of key-value pairs – referred to as an *object*, but known in other languages as records, structs, dictionaries to name a few. Second is the ordered list, or *array*. The values, both in the array and in the objects, may be another structure or one of the *primitive* data types defined by the JSON standard. These primitive data values are: strings, numbers, true and false (collectively referred to as *boolean* values) and null. Listing 1 contains an

```
[
    {
        "number": 1,
        "string": "abc",
        "another_number": 1.7e10,
        "boolean": true,
        "optional_number": null,
        "empty_string": ""
    },
    {
        "list": [0, null, "false", {}, true, []]
    }
]
```

Listing 1: Example JSON document. In this case, the JSON document is an array containing two objects, the first of which contains five keys that all lead to primitive values. The Second object has a single member and whose value is a list containing one value for each of the types that a JSON document can contain. The names of the keys hold no semantic meaning and were chosen for the sake of pedagogy.

example of a JSON document. By combining the two data structures with the primitive data values, JSON documents can contain massive amounts of data and easily translate many complex and hierarchical data structures. The term *key path* is used to indicate a sequence of one or more keys (and array indices) that lead to some value that is nested within a document.

Most programming languages differentiate between integers and floating point numbers because the range and type of values that they can represent is very different. JSON itself is in fact agnostic about how the number values it contains are interpreted – allowing any sequence of digits and characters that can be parsed as number, irrespective of size. The JSON RFC recommends that parsers at a minimum support to interpreting the values as 64-bit floating point numbers (also known as ”doubles”) [4]. Parsers are free however, to interpret JSON numbers as any numeric type, including integers. A common approach seems to be for JSON serializers to write all floating point numbers, including whole numbers, with a dot separator and at least one digit behind the dot (0 if the number is whole). So the floating point value 1 is encoded as `1.0`, while the integer value 1 is encoded as `1`. Parsers usually follow this scheme as well, parsing any number with decimal digit separator as a floating-point number and numbers without one as integers. Therefore, when this text refers to there being four non-null primitive JSON types, they are as follows: string, boolean, float and integer.

JSON started out as a data exchange format for internet applications, and has with time become the accepted standard for this use case, with large public APIs like Twitter's, Reddit's, Facebook's and more all supporting JSON. However, JSON has also come to be widely used for many other applications as well. Examples include logging services, web APIs, RPCs, and even as a storage format, as seen with recent rise in popularity of document (or ”NoSQL”) databases which store JSON. A large benefit of using document databases versus traditional relation databases is exactly this use of JSON. Due to its highly dynamic nature, document databases can allow users to insert any kind of document to a table, without requiring any predefined schemas. This dynamicity not only has the benefit of enabling rapid development, but also eases integration with external data sources. In recent years, many large relational database vendors have added proper support for JSON to their databases – including MySQL.

A JSON *document collection* is a collection of JSON documents. The documents in a collection may all be the exact same, have the same structure but have different values –

either of the same type or different types –, or they may all be completely different. In this text, the terms *homogeneity* and *heterogeneity* will be used to describe how similar (and different, respectively) in structure the documents in a collection are. As an example, the document in listing 1 could be seen as a very heterogeneous collection of two documents. A much more homogeneous document collection can be seen in listing 24. Note that across documents, keys can both be missing or be null.

The lack of structure enforcement for collections of JSON documents presents a challenge in several contexts, for example during data analysis of JSON data which relies heavily on assumptions about the structure of the data which are/cannot be guaranteed. Another area where the lack of schema and schema enforcement is presenting a challenge is databases trying to optimize queries over JSON data. This will be discussed in greater detail in section 3.

2.3.1 Deriving Schemas From Schemaless Data: JSON Schema

See foundations of json schema paper, IETF standard. 2016 paper claims that in grey areas, implementations differ "significantly". json-schema.org shows/links to drafts that show the specification is still under development. MongoDB's JSON schema resource/help page directs its readers to a 2013 IETF standard. 2023 blog post titled "the last breaking change in JSON Schema". Anyways – JSON Schema is kind of orthogonal to our use case. And in the case of a relational database, requiring it would destroy many of the advantages of the JSON data type.

2.3.2 JSON in Databases

Support for proper JSON data storage and retrieval is now common among the major RDBMS systems. The number of RDBMS systems which support statistics structures tailored to JSON data however, seems to be much smaller. This does not mean that users using JSON data have no options for helping the query optimizer make better estimates for queries involving JSON columns. A commonly supported feature for example, is to allow users to defined virtual (also called "functional") columns over JSON access expressions. Indexes and advanced statistics structures like histograms can then be created for these virtual columns. As discussed in section 3, some database vendors have gone much further in their support.

2.4 MySQL

MySQL is a highly popular free and open-source relational database management system (RDBMS) first released in 1995 and still under active development.

Since 2017, MySQL has had support for histogram statistics in the form of singleton and equi-height histograms [15]. Currently, histograms are supported for numeric data types, date and time types, strings, enums and the set type. Histograms must be both created and kept up to date manually. This is achieved using the `ANALYZE TABLE` statement [3], which will create fresh histogram statistics from scratch when run. This is achieved by, for each colum listed, sampling the data of the column (at the page level) until the sampling buffer as full, and then computing a histogram using that data. The size of the sampling buffer can be adjusted by the user. The MySQL query optimizer always prefers using indexes over histograms when performing selectivity estimation, so the creation of histograms should be reserved for columns for which no indexes exist.

2.4.1 Two Optimizers

The MySQL optimizer is known to be somewhat primitive compared to that of other competitor systems. A new optimizer, referred to as the *hypergraph optimizer*, is currently being developed by the MySQL team.

Among many other improvements, the hypergraph optimizer is better able utilize cost estimates, able to create bushy query plans (instead of only left-deep query plans), and may use histogram data to estimate the size of joins [12, 29, 28]. This in turn means that histogram statistics are likely to have a larger impact on the hypergraph optimizer than on MySQL's current query optimizer. By toggling a few compile time flags, the hypergraph optimizer can be made available in release builds of MySQL 8.0.32 and can then be enabled through the `@@optimizer_switch` setting variable.

```sql
SELECT
    listings.data
FROM
    listings,
    calendar
WHERE
    -- Join listings with their calendar
    listings.data->"$.id" = calendar.data->"$.listing_id"

    -- Filter out listings which have poor ratings
    AND listings.data->"$.review_scores_rating" >= 4

    -- Filter out listings which are not available on the given date
    AND calendar.data->"$.available" = "t"
    AND calendar.data->"$.date" = "2023-12-12"
ORDER BY
    listings.data->"$.last_review";
```

Listing 2: An example query for the Airbnb dataset (see sections 5 and 6) retrieving well-rated listings which are available on a certain date.
The column `j` of the `listings` table contains the JSON documents describing the listings in a City, with each document containing all data directly related to one listing (advert for a rental unit). The `calendar` table keeps track of when each listing is available and unavailable for booking.

2.4.2 JSON Support in MySQL

MySQL has extensive support for JSON, with a separate data type, specialized storage format and a long list of JSON-specific functions for accessing, creating, operating on, and converting JSON data [38, 18]. In the context of this project, the most central of these functions is `JSON_EXTRACT`. `JSON_EXTRACT(column, path)` (in its simplest form) returns data from the JSON documents in the given column that is retrieved by the argument path. If the argument path is not found, `NULL` is returned. The more common form of the function is its alias form as the `->` operator, used `column->path`. The `->>` operator is an alias for a call to `JSON_EXTRACT`, wrapped in a `JSON_UNQUOTE`. `JSON_UNQUOTE(json_val)` will "unquote" the argument json value, such that it will be interpreted as a literal value and be converted to whatever type that MySQL deems apropriate for that literal. Listing 3 shows how these two functions behave differently.

```
> CREATE TABLE t (data JSON);
> INSERT INTO t VALUES('{"a": "7"}');
> INSERT INTO t VALUES('{"a": 7}');

> SELECT * FROM t WHERE data->"$.a" = 7;
data
{"a": 7}

> SELECT * FROM t WHERE data->>"$.a" = 7;
data
{"a": "7"}
{"a": 7}
```

Listing 3: The difference between the `->` and `->>` operators.

2.4.3 Selectivity Estimation for MySQL JSON Queries

As of MySQL version 8.0.32 – the current release of MySQL at the time of this report's writing and deadline (July 20 2023) – there is no support for creating or using (histogram) statistics for JSON columns. Attempts to generate histograms or insert custom histograms for columns with the JSON column type result in an error message being returned, explaining that MySQL does not support histograms for the JSON data type.

In cases where indexes and histogram statistics are lacking, MySQL will normally fall back on a set of heuristic multiplers when performing selectivity estimation. However, these heuristic values are not emplyed when accessing JSON data using MySQL's JSON parsing functions such as `JSON_EXTRACT`. Instead, all such functions end up with a selectivity of 1.0, meaning that all rows are estimated to pass the filtering predicate.

These poor cardinality estimations are a detriment to MySQL, preventing its optimizer from making informed choices on query plan performance when dealing JSON queries. As will be shown in section 6, when the hypergraph optimizer is in use, poor choices of query plans are often made as a result with a resuting negative impact on MySQL's performance.

2.4.4 Examining MySQL's Query Plans

Prepending a query with the `EXPLAIN` statement will make stop MySQL from executing the query, and instead make it return its chosen query plan for the query [11]. Each step ("iterator") of the query plan is annotated with the estimated number of rows that will be returned by the step, as well as the estimate cost of performing the step. An example `EXPLAIN` output can be seen in listing 4, which shows the explanation for the example Airbnb query in listing 2.

By adding the `ANALYZE` keyword to get the `EXPLAIN ANALYZE` command, one can make MySQL execute the query and add additional information to each step in the execution. This added information includes the time spent executing each step, the number of loops performed and the real number of rows returned by the iterator. By comparing the estimated and real number of rows, one can calculate the estimation error made by MySQL at each step of the query plan.

```
-> Sort: json_extract(listings.data, '$.last_review')
  -> Stream results  (cost=15291378.98 rows=2633)
    -> Filter: (json_extract(listings.data, '$.id') = json_extract(calendar.data, '$.listing_id'))
    ↪  (cost=15291378.98 rows=2633)
      -> Inner hash join (<hash>(json_extract(listings.data,
      ↪  '$.id'))=<hash>(json_extract(calendar.data, '$.listing_id')))  (cost=15291378.98
      ↪  rows=2633)
        -> Filter: ((json_extract(calendar.data, '$.available') = 't') and
        ↪  (json_extract(calendar.data, '$.date') = '2023-12-12'))  (cost=12375.78 rows=2169)
          -> Table scan on calendar  (cost=12375.78 rows=2169417)
        -> Hash
          -> Filter: (json_extract(listings.data, '$.review_scores_rating') >= 4)  (cost=5558.20
          ↪  rows=1214)
            -> Table scan on listings  (cost=5558.20 rows=3642)

-----------------------------------------------------------------------------------------------

-> Sort: listings.j->'$.last_review'
  -> Stream results (rows=2633)
    -> Filter: listings.j->'$.id' = calendar.j->'$.listing_id' (rows=2633)
      -> Inner hash join (<hash>(listings.j->'$.id')=<hash>(calendar.j->'$.listing_id'))
      ↪  (rows=2633)
        -> Filter: (calendar.j->'$.available' = 't') and (calendar.j->'$.date' = '2023-12-12')
        ↪  (rows=2169)
          -> Table scan on calendar (rows=2169417)
        -> Hash
          -> Filter: listings.j->'$.review_scores_rating' >= 4 (rows=1214)
            -> Table scan on listings (rows=3642)
```

Listing 4: Example `EXPLAIN` output. To reduce the visual clutter, latter EXPLAIN output listings may (among other things) have their cost estimates removed and revert to using the arrow syntax for JSON expressions. The result of these changes is shown in version below the dashed line.

2.4.5 Extending MySQL

When extending MySQL, one of the core concepts one must be aware of is arena allocators. An arena allocator is a heap allocation structure and algorithm that provides very efficient short-term memory allocation by allocating all objects to the same region (”arena”). The entire region can then be reallocated or deallocated at once when the region is no longer needed. MySQL’s arena allocator structure is named `MEM_ROOT`, and is among other things used to store histogram data.

MySQL’s histogram component is relatively modern, being introduced in MySQL version 8.0.3 which released in 2017. All histogram implementations inherit from a common `Histogram` base class which defines the public interface of its implementors and also acts as a public interface for the component. The component is not heavily coupled to the rest of the codebase, with a single function serving as the entry point for retrieving a histogram, and another for estimating the selectivity of a predicate using a given histogram. The logic for when to retrieve actually retrieve a histogram and perform selectivity estimation however, is not centralized. Instead it is up to each `Item` object – the representation of a relational expression in a query whether – to decide. This is explored further in section 4.3.4

3 Related Work

This section will discuss previously published work on the topic of dealing collecting statistics for and querying large collections of JSON documents. The works have been organized into subsections by their approach to the problem. For each topic, the system and its context will first be introduced. Then, The parts of the work which are relevant to the goals of this project will be examined.

3.1 Translation Layer Solutions

When JSON was first becoming popular as a data storage and data exchange format, few major RDBMS systems had proper support for it. JSON functions were only added to the ISO SQL standard in 2016. Because of this, the approach taken by some developers and researchers was to put a translation layer in front of the RDBMS which would alter incoming JSON queries into something more 'traditional' that the system could understand and handle more gracefully.

3.1.1 Argo

One of the earliest published works on improving the performance of JSON data in a relational database is Chasseur et al. [6]. In this work, Chasseur et al. present an automatic mapping layer ("Argo") for queries that insert and retrieve JSON data. Instead of JSON being stored as large string blobs, JSON documents are mapped onto relational tuples and inserted into special tables for JSON documents. Queries that access JSON data are then translated to access these relational tables and tuples.

This approach allows the query optimizer to treat JSON queries as any other query accessing normal relations, and thus it can use its existing statistics and selectivity information infrastructure to make better cost estimates than it otherwise would be able to. In fact, the authors find that with Argo on top, MySQL's query optimizer is able to make some very good choices, outperforming MongoDB on many of the queries on the benchmark. However, no mention of histogram statistics is in the work.

Interestingly, Argo does no differentiate between integers and floats, even when storing types in separate tables. The authors do not elaborate on this choice however, so their reasoning remains unclear.

3.1.2 Sinew

A similar approach was taken by the authors of Tahara et al. [37]. However, instead of translating the JSON documents into sets of relational tuples that would be inserted into relational tables, Tahara et al. instead opted for a hybrid column-oriented approach using both physical and virtual columns. Sinew was intended not only as an improvement for JSON data, but any schemaless hierarchical data (such as XML). Where Argo inserts all tuples into the same table(s) and column(s), Sinew instead analyses the data to find common key-value pairs to extract into columns of their own. Uncommon key-valye pairs are inserted into a physical column for "leftover" data, and virtual columns are constructed for each unique key-value pair in this column.

As with Argo, Sinew's approach enables the collection and use of indexes and database statistics for JSON data. Being built on top of PostgreSQL means that Sinew benefits from automatic statistics collection (including histograms) for the physical columns. By not materializing every single key-value pair, but only the most frequent ones (above some frequency threshold), Sinew avoids gathering statistics for the most infrequent key-value pairs. An issue with this approach, as mentioned in Durner et al. [9], is that for very heterogeneous data, Sinew might end up materializing very few key-value combinations and end up using virtual columns for most key-value pairs. This will in turn make moot many of the advantages of the system, degrading its performance.

While Sinew achieves impressive improvements in performance compared to the base system, it is not clear how much of these gains that can be attributed to the availability of more detailed and accurate statistics.

3.2 Database Updates

The most conceptually straightforward solution to the problem of improving JSON performance and collecting JSON data statistics is to simply update the database system to better handle these cases.

3.2.1 Oracle

JSON support has been one of the Oracle RDBMS' many strengths for a long time. As of the time of writing, the database supports multiple types of indexes, a bespoke binary format and auto-generated schemas for JSON data [20, 21, 22]. It seems that the Oracle database does not gather any advanced statistics for its JSON data – no references to histograms or sketches or statistics structures of another kind was found during research. It is not unlikely that through use of one or more of the four different types of indexes supported for JSON data, additional statistics gathering becomes superfluous. Oracle themselves suggest that developers create functional indexes[2] over JSON key paths which are expected to be queried often. A more interesting index structure is the one referred to as the "JSON search index". It is intended to be used when many different key paths are likely to be explored, or when the developer is unsure of which paths are likely to be looked up. The search index consists of two indexes used together: First is an index mapping all key path strings to their respective document's id. Second is an index of leaf node values (usually referred to as primitive values in this report) [31].

The most interesting (with respect to the goals of this project) of all the techniques supported by the Oracle DBMS, is the DataGuide. The DataGuide is a structure that summarizes the structural and type information contained in a JSON column [1]. This information can then be used by the database to perform various optimizations. One of these optimizations is to automatically add and updating virtual columns when new fields or changes to existing fields are detected. The DataGuide also allows for the creation of soft schemas, and is automatically kept up-to-date if used in conjuction with a search index [1]. The DataGuide keeps track of several statistics for each path in the collection. This includes information such as frequency, the number of null values, and the minimum and maximum values [20, 21]. However, this statistics gathering is not automatic, but must be initiated by the user. Updates of these statistics must also be initiated manually [1].

Finally, a recent update of the Oracle DBMS added *JSON-Relation duality views.* This is a feature that is quite similar to traditional views, but which maps relational tables

[2]An index over a virtual column. In this case, the virtual column would be an expression accessing some JSON path.

to and from JSON. The feature allows for hierarchical relationships and nesting of data. Statistics are then collected for the relational tables as normal, and used when incoming queries are translated into accessing the underlying relational tables instead of the JSON structure specified in the query. Thus, the feature is similar to the translation layer solutions – with an interface emulating interacting with JSON directly while the reality is that the underlying tables are relational. However, because the feature is implemented at the database level instead of as a middleware on top of the database, it likely achieves both better performance and robustnes than the previously discussed translation layer solutions. While this solution is quite elegant, its approach is very different from supporting JSON data directly like the goal is with this project. Thus, it cannot really serve as an inspiration for this project. It is possible that, with respect to support for and use of advanced statistics, the approach of mapping JSON to relational tables will (once implemented) yield better results and with less work than the approach chosen for this project. This because existing statistics gathering and estimation techniques likely can be used as-is with the JSON-relational views. Statistics gathered for JSON data directly meanwhile, can possibly also re-use much existing logic, but there will likely need to be accommodations and changes made to existing statistics techniques to account for the particularities of JSON data[3].

3.2.2 JSON Tiles

'JSON Tiles: Fast Analytics on Semi-Structured Data' by Durner et al. is a very recently published work in which Durner et al. present the implementation of support for JSON in their columnar RDBMS Umbra. While the goals and techniques of the authors are broader than that of this project – going so far as to create a brand new storage format for JSON – some of the techniques used for collecting statistics very much also applicable in the context of MySQL.

Several novel techniques for dealing with JSON data are presented. A critique which the authors offer to earlier systems is that they handle heterogeneous data poorly. Their solution to this is the paper's namesake technique of tiling. Instead of analyzing the entire collection as one, the collection is analyzed such that the data can be collected into separate, possibly (inter-) heterogeneous sub-collections ("tiles") which are homogeneous internally. Then, each tile is mined for frequently occurring key-value combinations which are used in a further optimization in which these commonly occurring combinations are materialized into an efficient columnar storage format. Infrequent combinations are stored separately. This allows the system to very effectively deal with heterogenous data quickly narrowing down the search space to only the tiles which actually contain relevant data, and likely storing the data in a very efficient format. This is very much like the approach taken by Sinew, where each common key-path becomes a column. But unlike in Sinew, there's not just a single global relation – each tile becomes its own relation. A header is kept with each tile, describing the data it contains.

The eponymous tiling technique itself is not applicable in the context of this project, as it is intended for column-store database systems. However, for each of the separate tiles, the JSON tiles system collects statistics on the data within the tile before combining the statistics of each tile into a single statistics usable for the whole column. Umbra

[3]As discussed in section 4.3, all histogram estimation logic had to be re-created for the JSON statistics implementation and very little could be re-used. However, this was due to the structure of the JSON histograms being different from MySQL's existing histograms. It is plausible that a proper implementation, could use the same histogram logic for both cases.

does not collect histogram data, instead opting for the use of frequency counters and HyperLogLog sketches for its statistics. It is not clear if this decision was made because the authors believed it would yield the best statistics for the use case, or if the approach was chosen mainly due to HyperLogLog sketches being chosen solution for existing statistics gathering. The system supports limiting the total number of frequency counters and sketches used such that the memory footprint of the statistics is kept bounded and small. Because both the frequency counters and sketches gathered for each tile are combined, the query optimizer only has to do a single lookup of the global statistics when performing estimations. If the key being looked up is not found in the global frequency counter, the smallest frequency value in the statistics is used. The HyperLogLog sketches are used to extract 'domain statistics' for each key-value pair (which I interpret as meaning min and max values and the distinct values count). Finally, Umbra performs some sampling at runtime during query plan generation to improve (and create new if not present) its existing estimates.

MySQL does not currently use sketches to collect any statistics. Previous work on on this topic has found the use of sketches for statistics collection to lead to overall improvements in performance in MySQL[30]. However, implementing such a technique is outside the scope of this project, and so that exact feature is not very relevant to this project. However, the idea of keeping at most N number of counters and statistics objects per collection is very relevant.

3.3 Other Solutions

This section describes work that did not fit within the previous two categories.

3.3.1 Dremel

Another interesting solution from a team at Google (the first being Sinew) is Dremel [24, 23]. Dremel is a query execution engine designed to handle large-scale analysis of schemaless and hierarchical data such as JSON and XML. Dremel does not support type heterogeneity (same key leading to different types), but it does support making some key-value pairs optional. The development team have made multiple references to statistics being collected and used to speed up data collection [23, 17]. However, little has been published on the details of exactly which statistics are collected and how they are used.

3.4 Looking Back at Similar Problems

Another hierarchical human-readable (and potentially schemaless) data-interchange format that has seen widespread use is XML. In fact, in the cases where XML would previously have been used, JSON is now often used in its stead [6]. This replacement includes databases and storage systems, for which much effort has previously been spent on researching efficient ways of storing and retrieving XML data. The similarities between the use cases and structure of the two data formats, one would hope that the findings when it comes to optimizing XML retrieval would be applicable or somehow transferable to the case of optimizing JSON retrieval. But while there is published work showing that it is possible to successfully draw inspiration from the innovations of XML-oriented database systems [41], there are certain fundamental differences between the two data formats which can make the process very challenging.

In ‘Enabling JSON Document Stores in Relational Systems.’, Chasseur et al. [6] note some of the crucial differences between the two data formats.

- In JSON, keys always map to a single value. The equivalent of keys in XML (nodes) however may appear any number of times beneath the same parent.
- In the XML-equivalent of the JSON object, the order of keys (nodes) may have semantic meaning. JSON objects on the other hand are defined to be unordered, and one must instead of use arrays if the order of the children is intended to communicate semantically meaningful information [4].
- JSON supports several primitive data types: strings, booleans, numbers and null. And each of these can be determined from the syntax itself without any ambiguity. Basic XML, on the other hand, support only a single one: strings.
- Some XML document collections may have associated XML schemas (see figure **??**) or document type definitions (DTDs). These are descriptions of the structure, typing and other invariants of the documents in the collection. While there has been an effort to create something similar for JSON (see section ??? about JSON schema), the work is not yet finished and its use is not widespread. Therefore, techniques which exploit DTDs or schemas are not directly applicable to the JSON use case.

In addition to these semantic and syntactic differences, the use cases of JSON and XML are not as similar as they might at first appear. Chasseur et al. [6] make the observation that XML documents typically are large and heavy, and collect all entities belonging to the same conceptual group. For this reason XML documents are often not accessed directly or in their entirety, but rather are accessed via special APIs like SAX that parse the source XML document and emit only the requested information. JSON documents on the other hand are often more lightweight with less nesting. In modern applications, clients usually retrieve, manipulate and emit entire JSON document.

4 Implementation

Before proceeding with the discussion of implementation details, I wish to reiterate the goal of this project. This will hopefully help explain the motivation behind some of the decisions discussed further in this section.

The core goal of this thesis is to explore ways to improve MySQL’s selectivity estimates for JSON data. As discussed in section 3, little published work on this topic exists at the moment, and the work that exists is not directly applicable to MySQL (the actual degree of applicability varies from solution to solution). As such, the purpose of this thesis is to, based on previous research, synthesize a new approach to collecting database statistics for JSON data and explore the viability of that approach. What this thesis does not do, is take an existing and proven solution that we expect to work well and attempt to integrate it into MySQL in the best and most efficient way possible. It is also not the goal of this project to create and implement a structure that is thoroughly optimal and efficient. Rather, it is to design and test a broader approach to the problem, to see if that approach is even viable at all. However, some ideas as to how to improve and optimize the structure are explored.

The actual, technical implementation work was divided into two stages: first, a free-standing, proof of concept prototype was built in Python. The goal of this stage was to enable rapid iterative development and to avoid getting bogged down by limitations imposed by the existing MySQL codebase. The stage allowed me to explore various approaches to the problem and enabled me to quickly test the effect of various parameters and architecture decisions. The second stage of the project was the implementation of the solution in MySQL. Both stages of implementation will be discussed in this section. For each stage, I will be discussing the technical implementation of the system, as well as some of the technical challenges faced and how they were dealt with.

The remainder of this section is divided into four main parts. The first section contains an examination of the design of the proposed data structure itself. Next, section 4.2 describes a standalone prototype implemented in Python. Finally, in section 4.3 I will discuss the implementation of another prototype, this time in implemented MySQL and written in C++.

4.1 Concepts and Design

The focus of this project has chiefly been the actual writing/coding and implementation of a system for estimating selectivity for JSON. But while the survey of related work revealed concepts and ideas to could be used within the context of a MySQL implementation, none of the surveyed projects provided an exact template for something that could feasibly be directly implemented in MySQL. Therefore, the project also required a degree of synthesis and invention to come to a design which could be implemented as a prototype in MySQL. There is nothing novel in the individual components of the data structure – counters, singleton histograms, equi-height histograms, I do believe that the final makeup and use of the structure is fairly novel. As such, much of my early efforts were spent as much on theoretical considerations related to the design of the data structure as they were on its actual implementation. As most of these design decisions are present in both the Python prototype and the C++ MySQL prototype, I will spend this current section presenting that shared design at a non-technical level (as in no code or implementation details).

The goal of the algorithm is to take a collection of JSON documents and produce some data that very succinctly describes the data in the collection in as much (useful) detail as possible. I.e., exactly the same as the goal of other database histogram generation algorithms. The difference lies in the structure of the input data, as well as the nature of the filtering predicates. While predicates for other common data types like integers and floats ask questions about the entire value/piece of data itself, predicates over JSON data (may) ask about the attributes of only certain parts of the contents of the entire field. To allow us to estimate the selectivity of such predicates, the data structure must be able to answer questions about not the whole of the item, but rather those specific parts referenced in the predicate. This requires a much more granular data structure than standard histograms (which, again, only describe the distribution of each value as a whole).

A natural way to structure the JSON statistics is in a format that mirrors the documents of the JSON collection itself. A simple example of the transformation of some collection to a basic structure counts the number of occurrences of each path could then look something like the following:

```
[{"a": 1, "b": 1}, {"b": true}]  →  {"a": 1, "b": 2}
```

Querying this structure is as simple as querying the source document, and the same key path used in the predicate can be used to query the data structure. So for a query like the one seen in listing 5, we can use the exact same key path `"$.a"` to query the data structure. By using the same storage format for the statistics as the original data, we ensure good compatibility and avoid the added complexity of having to support any additional storage formats. Finally, it allows the statistics to have a very dynamic structure, which is both great for matching the dynamicity of JSON data as well as for allowing rapid iteration during development.

The approach of mimicking the structure of the JSON documents themselves also works really well when implementing the scheme n MySQL because MySQL stores its histogram data as JSON. Extending MySQL to accept and use histograms structured as a JSON document will therefore not require any changes to MySQL's core histogram storage and retrieval logic. See section 4.3 for more on this topic.

```
SELECT ... FROM t WHERE t.data->"$.a" = 1
```

Listing 5: A simple SELECT query featuring an equality predicate over the JSON column 'data'. The `->` syntax is shorthand for the MySQL JSON functions `JSON_EXTRACT(column, path)`.

4.1.1 Extending the Design

We can see that even for a simple query such as the one in listing 5 that collecting only the count of each key path is not sufficient to estimate the selectivity of all predicates. However, the proposed structure can easily be extended to hold practically any type of (relevant) information by having the values in the statistics object be objects themselves, rather than just a simple counter. The information we can store for each path is then only limited by what the JSON format can convey, and it can easily accommodate the information that histograms usually contain. Using this scheme, the statistics object from before becomes

```
{"a": {"count": 1}, "b": {"count": 2}}.
```

Exactly what information to store in the value associated with each key path is an open question. The cardinality (or alternatively, the frequency) of the key path is a cheap but very useful statistic. Other information that may be useful is the number of (or, again, the frequency of) null values, the number of distinct values, and the minimum and maximum values. We can even store entire histograms for each path. Storing actual values that appear in the documents presents a challenge, however: across multiple documents, the same key path can lead to values of different types. Determining the minimum of a string, an integer and an array is not something that is commonly understood or well-defined – not to speak of creating a histogram of such disparate values. Thus, the design must either allow the keeping of separate statistics for each type that appears for each path, or only keep the statistics for a single type, similar to how Json tiles solves the problem.

Other considerations that must be made include how to deal with nested paths. The natural solution to this, and the solution which continues the theme of simply mirroring the source structure, is to nest the statistics objects. Listing 6 shows how this might look. While this approach likely is a fine solution, it has a couple of drawbacks: speed is not optimal (it's not "hashmappable", could be very nested but few keys overall, less predictable structure) and wildcard paths become very expensive (`"\$.**.filtering\_effect"` for example). Simply flattening the structure and putting all keys on the same level with no

```
[
    {"a": 1, "b": {"c": 3}},
    {"b": null},
    {"a": {"c": 3}}
]
```

$$\downarrow$$

```
{
    "a": {
        "count": 2,
        "null-count": 0,
        "children": {
            "c": {
                "count": 1
            }
        }
    },
    "b": {
        "count": 2,
        "null-count": 1
        "children": {
            "c": {
                "count": 1
            }
        }
    }
}
```

Listing 6: How a nested JSON statistics structure might look.

further modification of the keys will also not work because keys with different parentage may have the same name. An example of this can be seen in 6, where the key ”c” occurs both as a child of two different parent keys: ”b” and ”a”. The solution chosen for this project was to keep all the key paths at the top level but to combine all nested keys with the key of every single parent into a single string. This allows the structure to easily be converted into a hashmap, providing constant time lookup of keys and making the structure itself very ”predictable” in the sense that there’s no question about how deep something is nested. The statistics structure in listing 7 shows an example of this design.

Listing 6 showcases another challenge: JSON keys can both be missing and lead to a null value. In the listing, this is tracked through the use of a separate counter for nulls.

4.1.2 The Final Design

Listing 7 shows an example of the final design that was decided upon. This exact listing was generated by the Python prototype and so it tracks cardinality rather than frequency. Going from cardinality to frequency can be achieved by dividing the counters by the collection’s size.

```
{
    "user": {
        "count": 10,
        "null_count": 0
    },
    "user_obj": {
        "count": 10,
        "null_count": 0
    },
    "user_obj.id": {
        "count": 10,
        "null_count": 0
    },
    "user_obj.id_num": {
        "count": 10,
        "null_count": 0,
        "min_val": 1,
        "max_val": 10,
        "ndv": 10,
        "histogram": [
            "EQUI_HEIGHT",
            [[4, 4, 4], [8, 4, 4], [10, 2, 2]]
        ]
    },
    "user_obj.screen_name": {
        "count": 8,
        "null_count": 5
    },
    "user_obj.screen_name_str": {
        "count": 3,
        "null_count": 0,
        "min_val": "harry",
        "max_val": "ron",
        "ndv": 3,
        "histogram": [
            "SINGLETON",
            [["harry", 1], ["hermoine", 1], ["ron", 1]]
        ]
    },
}
```

Listing 7: The statistics created for a collection of 10 documents, containing information on some users, as output by the Python prototype. The actual JSON collection be found in listing 24 in appendix A.

Each path is recorded, both with and without a type suffix. The underscore character is used as the separator between the key string and the type suffix. Each key along with its type suffix, is separated from its child keys by the dot character. These will be referred to as the ”type separator” and the ”key separator”. For every path, the count (”cardinality”) is stored, as well as the null count, i.e., the count of how many times the key path was present in a document but led to a null value. If a key path is not present in a document, it is not counted. For key paths typed with a primitive type, i.e., key paths that lead to a primitive value, additional information may be stored. The minimum and maximum values encountered are stored as ”min_val” and ”max_val” respectively. The number of distinct values may also be stored as ”ndv”. Finally, a histogram may also be present. Histograms are either equi-height or singleton and are very simple in their construction. An equi-height bucket contains three values. These are as follows (in order): the bucket’s (inclusive) upper bound, the count of values the bucket accounts for, and the number of distinct values in the bucket. The buckets of the singleton histogram are even simpler. Each bucket contains first the value itself, and second the count for that value. In both cases of histograms, the buckets are always sorted in ascending order – the equi-height buckets by their upper bound and the singleton buckets by their value.

Null values are handled differently from the other primitive data types supported by JSON. Whereas the other types have their own type suffix and have information collected separately for them, the null values are largely ignored. The base/untyped path records the number of null values encountered for that path, but that's all. But this is also sufficient for getting accurate estimates related to null values.

The key paths do not nest as their document counterparts do. Instead, everything is kept at a single level of nesting, no matter how deeply nested the documents in the source collection may be. Instead, non-top-level keys are prefixed with all of their parents' combined (recursively) key strings. Each key is separated by a special character – the "key separator". Before the keys are combined, they are suffixed with information on their type. This allows implementing systems to look up the statistics for a specific type by simply appending the respective type suffix to the key it's interested in. If the system is interested in all types related to a key path, then it can omit the type suffix, and retrieve the "parent" statistics that share the same key path irrespective of the type of the value that it leads to. This can be seen in listing 7, where the untyped key path "user_obj.screen_name" leads to information about all occurrences of the key "screen_name" inside an object lead to by the "user" key, while the typed key path "user_obj.screen_name_str" leads to information about the documents where the "screen_name" key lead to a string value.

While the underscore ('_') and dot ('.') characters are used as type and key separators respectively, the design does not mandate the use of these characters. In fact, they are likely poor choices, as they are prone to appear in existing key strings and thus may confuse implementing systems. Any other characters allowed in JSON key strings could be used in their stead. Which characters to use in their stead is a complex issue and not something I will spend more time discussing here. However, using the underscore and dot characters was not found to cause any trouble during the implementation and experiments of this project, so they were kept because due to their clarity and readability.

Arrays are treated like they are in Javascript: as objects with integer strings ("0", "1", etc.) for keys. This makes it so that almost no changes to the structure are needed to accommodate them. Arrays do get a separate type suffix from objects, however, as being able to differentiate between the two could prove useful.

As discussed in section 2.3, the JSON format itself does not explicitly distinguish between floating-point and integer numbers, but serializers and parsers often do. It is therefore possible for the statistics structure to differentiate between integers and floating point numbers. Tracking the two types separately could allow for increased accuracy, as more assumptions can be made about the distribution of integers than the distribution of real numbers[4], which again can lead to more accuracy estimations. In addition, separating the two types would make the statistics more fine-grained. However, there are a few problems when taking this approach. First, one cannot rely on serializers to always clearly distinguish between integers and floats. For example, Javascript's `JSON.stringify`, will encode both 1 and 1.0 as '1.0'. Thus, the optimization cannot be applied evenly. Additionally, comparisons against numeric values currently lack type information by default. For example, given a predicate like, `j->"\$.n" > 1` the system does not know whether to look up statistics for integers or floats. Assuming the wrong type could lead to very poor estimates. One could look up both, but that would defeat the point of separating the two values. Alternatively, all such comparisons would have to be explicitly typed by the user, which they currently are not required to be in MySQL. For these reasons, it was decided that the simplest choice was to track both integers and floating point numbers together.

[4]For example, given an equi-height histogram

The design does contain some redundant information in a few places. For example, paths that lead to primitive JSON values will always have a null count of 0. Including it in all statistics objects simplifies the design, however. For paths where no null values were found, the count and null values count will be equal for both the typed and untyped paths, leading to duplicate information being stored. Some of this redundancy may be removed by certain optimization techniques. However, the base design keeps the slight redundancy for the sake of simplicity of design and ease of implementation.

4.1.3 Histograms

The design supports the creation of two common types of histograms: equi-height histograms and singleton histograms. Both types can be seen in the statistics structure in listing 7. The design does not mandate the use of any specific histogram types or implementations. However, both prototypes use almost the exact same design: equi-height buckets contain the bucket's upper bound, the count/frequency, and the number of distinct values. Singleton histogram buckets contain the value and then the value's count/frequency. Very simple structures were intentionally selected to ease the task of implementing the histograms. It is not unlikely that employing more advanced histogram structures would lead to a moderate improvement in the accuracy of estimations.

A minor variation on the singleton histogram, where only the most frequent values are included in the histogram, is also permitted. This histogram type is only created when the number of distinct values is higher than the allowed bucket count for the singleton histogram, and when some values have an outsize frequency. For example, in the Twitter data set (see section 6.2 for more details on the data), 37 distinct string values are found for the "source" key. However, the distribution of these values is heavily skewed, with the most frequent value having a frequency of 0.9, and another having a frequency of 0.08. The 35 remaining strings have a combined frequency of 0.02. In this case, the singleton histogram stores those two most frequent strings, as well as the mean frequency of the remaining items: $0.02/35 = 0.0006$.

This is not a novel technique, and other RDBMS systems implement a similar system. The Python implementation refers to these as "singleton plus" histograms, while the MySQL prototype features an optional field ("rest_mean_frequency") in its histogram that indicates that this singleton histogram type was used.

4.1.4 Handling Array Operations

A unique advantage of using JSON documents as a data store is that they allow one to store ordered collections (by storing the items in an array). The JSON standard guarantees that the order of (dissimilar) items in an array is fixed [4]. As Databases implementing JSON support are required (by the SQL ISO) to conform to the JSON RFC, RDBMS systems must make this guarantee as well. Ordered collections are technically possible to achieve in relational databases like MySQL by having a table for the items and within that table storing the index of the item within the parent collection. An example of this can be seen in the tables used in the TPC-H benchmark, where the LINEITEM table stores a LINENUMBER column [40]. I would argue however, that this is not really comparable to having an actual ordered list directly available.

JSON arrays are also used for unordered collections. An example of such a usage can be found in the Airbnb data used for the experiments in sections 5 and 6. Each Airbnb listing includes an unordered list of the amenities provided. Some examples amenities are: "Fire extinguisher", "Bed linens", "Wifi" and "Host greets you". A typical filter for the listings data would be to check whether a certain amenity or a certain set of amenities is provided by a listing. However, with the base statistics structure as presented so far, the only estimation technique that could be used would be to estimate equality for every single amenity for every single existing index in the array. This is both very inefficient and also likely to lead to errors in the estimate if we are unable to make a good estimate for certain combinations of amenities and indexes.

The design handles this case by collecting statistics for the entire array as a single unit, instead of for each array index separately. This collective array information is then stored in a separate path that is formatted somewhat differently from the paths of other statistics, so that it can easily be identified as containing array statistics. The implementation can then look up this path to make more accurate estimates more quickly than otherwise when faced with filters that check a subset/superset condition, like MySQL's JSON_CONTAINS and JSON_MEMBEROF. The statistics kept for the path are exactly the same as the ones tracked for other paths. The difference is – of course – that the statistics are for all values in all places of the array rather than just at a single index. The only histogram type supported for array statistics is the singleton histogram. The singleton histogram tracks the frequency with which each child element appears in the array. Figures 8 and 9 show an example of how of array statistics look, compared to the statistics that are normally collected for arrays.

It should be noted that the statistics collected in this manner are not always applicable. If the arrays contain duplicates, or if the data is not "enum-like" then its accuracy will suffer. No information on the ordering of the array elements is stored. Additionally, this technique is only intended to be used with string or integer arrays.

4.1.5 Limitations

The design in its current form has several limitations. This section lists a few that did not fit elsewhere.

- The design does not really define how to handle primitive types at the top level. The Python prototype will handle this case fine, but a more explicit protocol would be preferable.
- MySQL's histogram system is currently designed to handle data types which average much smaller values (in terms of size in bytes) than what JSON documents can and often tend to be. This is important because when MySQL samples a for the purpose of building a histogram, it samples only as many values (blocks, really) as will fit within the allocated buffer. The size of this buffer is controlled by the variable *histogram_generation_max_mem_size* which has a default size of 20 000 000 and a minimum size of 1 000 000 – 19 and 0.95 megabytes respectively [33]. Though it's not common, a single JSON document can easily reach a megabyte in size, so if these values are any indication of the buffer sizes used in real applications, the sampling rate for a large collection could end up being absolutely miniscule.
- Join estimates, when the HG optimizer is used, do not necessarily (the can by accident if a filter is first applied on the join condition element) take into account

```
{
    ...,
    "host_verifications_arr.0_str": {
        "count": 42440,
        "null_count": 0,
        "min_val": "email",
        "max_val": "phone",
        "ndv": 2,
        "histogram": [
            "SINGLETON", [
                ["email", 38727],
                ["phone", 3713]
            ]]
        },
    "host_verifications_arr.1_str": {
        "count": 38947,
        ...,
        "histogram": [
            "SINGLETON", [
                ["phone", 38682],
                ["work_email", 265]
            ]]
        },
    "host_verifications_arr.2_str": {
        "count": 5373,
        ...,
        "histogram": [
            "SINGLETON", [
                ["work_email", 5373]
            ]]
    },
    ...
}
```

Listing 8: The statistics collected for each index of the 'host_verifications' array. Some details have been left out.

the selectivity of the path being joined on. This makes makes sense for all other data types, but is not necessarily true for JSON data. For example, a key being joined on could only be present in 1% of all documents in the collection. Ideally, an implicit IS NOT NULL estimate would be performed on the path that was being joined on first, to handle this case.

4.2 The Python Prototype

The purpose of the Python prototype was twofold: first, I wanted to quickly build a proof-of-concept codebase that would implement the data structure and be able to build it from raw JSON data. This would allow me to quickly encounter any unforeseen technical or design challenges and to work around them in an environment where making changes was cheap. The second purpose of the Python prototype was to enable me to measure the performance of the data structure and the impact that various variations in the design would have. Again, the Python prototype seemed like it would ease the process of testing variations in design and the measuring the impact of those variations on performance. This was not only due to the Python prototype being a "green field" project (compared to the MySQL-prototype which had to fit into the existing MySQL codebase), but also due to Python's dynamic nature and due to it being an interpreted language (i.e., no waiting for compilation). Additionally, I assumed that the development velocity would be much higher creating a prototype in Python instead of MySQL and C++ due to me being much more familiar with the development process in the former. This assumption turned out

```
{
    ...,
    "host_verifications_arr[str]": {
        "count": 42440,
        "null_count": 0,
        "ndv": 3,
        "histogram": [
            "SINGLETON", [
                ["phone", 42395],
                ["email", 38727],
                ["work_email", 5638]
            ]]
    },
    ...
}
```

Listing 9: Array statistics summarizing the 'host_verifications' key path.

to be correct.

The end result was a program can ingest JSON data, generate a corresponding statistics collection, and then make cardinality estimates using those statistics and compare them to the true cardinality. All of this can be done while settings are changed dynamically during runtime so that the differences between varying configurations can be easily compared.

In this section I will be discussing the implementation of the Python Prototype and detail how it differs from the core design explained in section 4.1.

4.2.1 Overall Structure

The Python prototype consists of three main modules: `compute_stats.py` holds the code for taking a JSON collection and capturing statistics of the data. `use_stats.py` holds all the functions for computing estimates from the statistics. Finally, `analyze.py` uses the two aforementioned modules to create statistics with variations in parameters and then to test the accuracy of the estimations that those statistics allow the system to make. Additionally, several smaller modules provide additional functionality to the system. The various tweakable parameters which affect statistics creation are stored in a settings object initially created in `settings.py`. The statistics creation code reads this settings object before beginning its work to determine which statistics to compute, how to compute those statistics, and finally which optimizations to apply.

4.2.2 Tunable Parameters

The Python prototype features several tuneable parameters/settings that change both which data is captured and how it is captured. This section details those settings and their impact on the program.

By far the most impactful setting is which statistics "type" to create. The possible allowed values for this setting and their effects are as follows:

- **BASIC**: The BASIC option is intended to be the cheapest and most efficient option, both in terms of computation cost and storage. It minimizes cost by recording as little data as possible, storing only singular scalar values and no arrays or hashmaps.

Using this option, the count, null count as well as the minimum and maximum values (where possible to compute) will be recorded and stored.

- **BASIC_NDV**: This option is nearly identical to the BASIC option – the only difference between the two iss that using this option, the system will also attempt to record the number of distinct values for each (typed) key path. Using a naive implementation, this approach is much more expensive than the preceding option, as computing the NDV requires storing all unique values encountered. This option is in part included to be used as a baseline to compare against the next option.

- **NDV_HYPERLOG**: The structure of the statistics output when using the NDV_HYPERLOG is exactly the same as when using the BASIC_NDV setting. The difference between the two settings lies in how the ndv statistic is computed. Instead of using a hashmap, this method relies on the HyperLogLog method[13]. The specific implementation used was the PyPi-hosted 'hyperloglog' library[5]. Using HyperLogLog allows for the estimation of NDV statistics with high accuracy while significantly reducing the memory footprint compared to the previous approach. Controlling the allowed HyperLogLog error can be done through a separate setting. An allowed error of 0.05 (5%) was found to produce good estimates while having a low memory cost and was used during all further work.

- **NDV_WITH_MODE**: Using this setting, the system will collect the same information that the previous two settings would result in, plus a mode value and count for each path where applicable. With this setting, the system uses the same brute force approach to collecting the ndv statistic as the BASIC_NDV option.

- **HISTOGRAM**: When the statistics "type" is set to HISTOGRAM, the system will attempt to compute histograms for all paths ending in non-null primitive values. In addition, all the previously mentioned statistics will be collected as well. Because computing a histogram requires all values to be collected (or all unique values along with their frequency), HyperLogLog is not used to compute the NDV statistic. The number of buckets allowed per histogram is a tunable parameter. The system will create singleton histograms for all collections of values whose cardinality is less than or equal to the number of allowed buckets. Otherwise, equi-height histograms will be created. Equi-height histograms are not supported for string values.

JSON document collections may have a massive number of unique paths can potentially be massive. Because our method collects statistics for each unique (typed) path, this will in turn lead to a massive statistics collection. This may be problematic for the performance of the system, both because traversing a larger structure is more expensive, and because keeping the structure in memory may negatively affect the other operations of the system by reducing the remaining space available for other operations. The simplest approach to reducing the size of the statistics object is to collect less detailed statistics. Specifically, not collecting histogram data. For collections with many paths that lead to primitive values, the storage required to store all the histogram data – especially if the number of allowed buckets is high – may far outstrip the storage required for the remainder of the structure. This is especially true when lots of string values are involved.

Alternatively, a number of other techniques may be employed to reduce the storage requirements of the structure. We refer to the use of these techniques as *pruning* the JSON tree structure. The Python prototype supports several pruning approaches, all of which

[5] https://pypi.org/project/hyperloglog/

can be combined with the others if so desired. Which pruning approaches are used, as well as the parameters of these algorithms, is controlled by the "prune strats" (pruning strategies) and "prune params" (pruning parameters) settings. The pruning strategies are as follows:

- **MIN_FREQ**: This setting, along with its threshold parameter will drop the statistics of all paths whose frequency is determined to be below the given threshold. This allows the system to remove all paths whose frequency is so low that the system is unlikely to benefit much from the estimates that they enabled. As an extreme example of such a case, MySQL prevents the histograms module from returning estimates lower than 0.001 [34]. Keeping statistics for paths whose frequency is at or below that threshold then offers no benefits.

- **MAX_NO_PATHS**: While the previous setting may reduce the number of paths stored, this is by no means guaranteed. The MAX_NO_PATHS setting guarantees that no more than a specific number of paths, as set by the algorithm's parameter, will be included in the final structure. Paths whose frequency is lower are removed first. Ties are currently broken through lexicographical ordering, which is not ideal. A better approach would be to somehow prioritize paths by somehow judging their likely benefit to selectivity estimation. For example, accessing array elements may be less likely than accessing object values. If the goal is to reduce size, then larger objects should be removed first.

- **MAX_PREFIX_LENGTH**: The previous two approaches were based on reducing the number of paths in the final output, this approach instead sets the maximum length that paths in the final system may have. The length here refers to the number of keys included, rather than the string length itself. For example, with a parameter of max length = 2, the path user_object.friends_array.2_object.id_number would instead become 2_object.id_number. While this technique does lead to a reduction in the size of the statistics object, it may also lead to the statistics of two or more separate paths being combined into one, degrading the estimates for each path involved. A more advanced approach than simply cutting off the first part of the string could likely allow us to avoid most collisions. However, this algorithm would have to be implemented by the system querying the statistics as well.

  An alternate, and likely better approach to this kind of pruning could be to replace the paths by their hash value. I did not have time to explore this avenue.

- **UNIQUE_SUFFIX**: This algorithm is similar to the MAX_PREFIX_LENGTH algorithm, in that it attempts to reduce the storage requirements of the statistics by pruning paths. However, as opposed to the aforementioned algorithm, UNIQUE_SUFFIX does not lead to collisions between the pruned paths. This is because, rather than having a set maximum allowed length, the algorithm attempts to shorten each path as much as possible, but only performs the pruning of the path string if the pruned path does not collide with any existing paths.

It should be noted that, except for the MAX_NO_PREFIX pruning strategy, all of these techniques are applied after the complete statistics have been collected. Therefore, they do not reduce the running time or storage size requirements of the system that creates the statistics – these alterations only benefit the system that stores and queries the statistics. None of the pruning settings are aware of, or take into account the sampling rate.

The `singleton_plus_enabled` and `enum_statistics_enabled` settings respectively toggle the "singleton plus" and array histogram features. The `sampling_rate` setting determines the sampling rate used when analyzing the data. Finally, some parameters were not included in the settings object, but are still easily tweakable. Examples of these include the type and key separator values, as well as various settings adjusting the behavior of the analysis code.

4.2.3 Collecting statistics

When generating the statistics, the prototype examines one document at a time, traversing all paths recursively and (depending on the settings used) storing the data it encounters for later aggregate analysis. After all the documents have been iterated over, aggregate statistics (ndv, histograms) are then computed for these collections. The null values count and minimum and maximum statistics are calculated throughout.

After all the statistics have been computed, the pruning strategies are applied to get the final statistics object.

Along the way, several meta-statistics are tracked and kept in a separate object. These are then stored together with the statistics themselves and provide context about the collected statistics. The most important of these extra statistics are the sample rate, the size of the collection, and the highest cardinality of the paths not included in the final statistics.

4.2.4 Querying the Statistics

The Python prototype not only supports creating the JSON collection statistics themselves. It also allows for measuring the performance (read: accuracy) of the gathered statistics. To this end, a module for querying the statistics was implemented. The statistics querying module supports the following operations by implementing special logic for each: EXISTS, IS NULL, IS NOT NULL, GREATER THAN, LESS THAN, EQUAL TO, JSON_MEMBEROF, JSON_CONTAINS, JSON_OVERLAPS. An example of the implementation of one of these functions can be found in listing 10.

Each operation is different logic for each type of statistics gathered, at least where applicable. Each function follows much of the same logic. All functions first check to see if they can find a match for the argument query path in the collected statistics. If not, then the highest cardinality seen and which is not included in the statistics is used as a substitute. The cardinality is multiplied by a heuristic value which is dependent on the operation. For example, the greater than operation will use 0.3 as its heuristic constant. This approach of using heuristic constants when no other information is present goes back to System R [32]. If the statistics structure created for the given path is found, the function starts by comparing the lookup value and the minimum and maximum values encountered. If the lookup value appears to be outside the range of encountered values, the function will return an estimate of 0. The next steps depend on the type/depth of the gathered statistics. If histograms are present, the function will loop over the histogram until it finds the bucket that matches the compare value. It will then use the information in the bucket (and possibly the previous buckets as well) to make its estimate.

The functions that estimate JSON array membership/subset/overlap have slightly different logic from the other predicate estimation functions. Each of these functions relies on an internal function that estimates the cardinality of the JSON_MEMBEROF func-

```python
def estimate_eq_cardinality(query_path: str, compare_value: Json_primitive) -> int:
  if query_path not in statistics:
    max_skipped_count = meta_statistics["highest_count_skipped"]
    return max_skipped_count * FALLBACK_EQUALITY_MULTIPLIER

  path_stats = statistics[query_path]

  if compare_value > path_stats.max_val or compare_value < path_stats.min_val:
    return 0

  match settings.statistics.stat_type:
    case StatType.BASIC:
      return path_stats.count * FALLBACK_EQUALITY_MULTIPLIER
    case StatType.BASIC_NDV | StatType.NDV_HYPERLOG:
      return path_stats.count / path_stats.ndv
    case StatType.NDV_WITH_MODE:
      mode, mode_count = path_stats.mode_info
      if compare_value == mode:
        return mode_count
      if path_stats.ndv == 1:
        # If mode was the only value encountered during statistics creation
        # then compare_value was seen 0 times
        return 0
      # Divide by ndv-1 as we can exclude the mode
      return (path_stats.count - mode_count) / (path_stats.ndv - 1)
    case StatType.HISTOGRAM:
      match path_stats.histogram.type:
        case HistogramType.SINGLETON:
          return next((b.count for b in path_stats.histogram.buckets
                 if b.value == compare_value), 0)

        case HistogramType.EQUI_HEIGHT:
          for bucket in path_stats.histogram.buckets:
            if bucket.upper_bound >= compare_value:
              return bucket.count / bucket.ndv
          return 0

        case HistogramType.SINGLETON_PLUS:
          normal_buckets = path_stats.histogram.buckets[:-1]
          plus_bucket = path_stats.histogram.buckets[-1]
          fallback = plus_bucket.count / plus_bucket.ndv
          return next((b.count for b in normal_buckets
                 if b.value == compare_value), fallback)
```

Listing 10: The Python function for estimating the number of documents that satisfy the condition "= compare_value". Somewhat simplified compared to the original implementation. Rounding, adjusting for the sampling rate, and the handling of prefix/suffix pruning is done in a wrapper function that is not shown. The value of FALLBACK_EQUALITY_MULTIPLIER is 0.1.

tion for a single value. The ”memberof” function can use this value directly. For the JSON_OVERLAPS function, the ”memberof” cardinality is calculated for each member of the argument array of values to look up, and the maximum of those is returned (or the maximum times the square root of the size of the lookup array). For JSON_CONTAINS, the minimum of the ”memberof” cardinality of the array members is returned.

```python
def _get_memberof_cardinality_estimate(base_stat_path, lookup_val):
  arr_stat_path = make_enum_array_stat_path(base_stat_path, type(lookup_val))
  if arr_stat_path not in statistics:
    # If we can't find array statistics, look up the stats of the common parent path instead.
    # If the stat type is histogram, we could optionally look through the histograms
    #   of each child element of the array, though that would be very expensive.
    return statistics[base_stat_path].count * JSON_MEMBEROF_MULTIPLIER

  arr_stats = statistics[arr_stat_path]

  match settings.stats.stats_type:
    case StatType.BASIC:
      return arr_stats.count * JSON_MEMBEROF_MULTIPLIER
    case StatType.BASIC_NDV | StatType.NDV_HYPERLOG:
      return arr_stats.count / arr_stats.ndv
    case StatType.NDV_WITH_MODE:
      if arr_stats.mode_info:
        if arr_stats.mode_info.value == lookup_val:
          return arr_stats.mode_info.count
        return (arr_stats.count - arr_stats.mode_info.count) / (arr_stats.ndv - 1)
      return arr_stats.count / arr_stats.ndv
    case StatType.HISTOGRAM:
      histogram = arr_stats.histogram

      for bucket in hist_buckets:
        if type(bucket) == EquiHeightBucket:
          # Reaching an equi-height bucket means we reached the final bucket of
          #   the "singleton-plus" histogram
          return bucket.count / bucket.ndv  # average of the remaining cardinalities
        elif bucket.value == lookup_val:
          return bucket.count
      return 0
```

Listing 11: The Python function for estimating the number of documents that satisfy the condition ”JSON_MEMBEROF([...])”. Some details have been left out.

4.2.5 More Approaches to Reducing Size

Several more approaches to reducing the size of the histogram are possible. For example, the type suffix string for internal nodes of the JSON representation (”obj” and ”arr”) are not strictly necessary for the system to function, as the type information of the system is currently only used for the primitive leaf nodes. This would not only eliminate the type suffix itself from those path strings, but also remove duplicate information because the system will no longer store both the path ’...key_obj’ and ’...key’, but only the ’...key’ path. This was in fact implemented in the Python prototype and found to reduce sizes by a couple of percentage points.

A very simple technique that was not implemented is to shorten the type string suffix that is appended to each key in a path. As each type supported by the system starts with a different letter, this would be very simple: ’arr’ would become ’a’, ’bool’ would become ’b’, and so on. The type suffix separator (set to ’_’ during our work on the system) could also be completely dropped.

A final approach to seriously reducing the size of the statistics object would be to disallow storing string values (both min/max, mode, and in histograms), or to place a very strict maximum length restriction on the strings stored in the statistics object. MySQL does this when building its histograms, considering at most 42 characters per string value [35].

4.2.6 Preparing for Data Analysis

The Python prototype also includes code for testing and comparing the different configurations against each other on a series of metrics including storage size required, time to create, and accuracy of estimates. This is achieved by first generating a set of test data for every single key in the source collection. Then a set of configurations is iterated over, and for each configuration, statistics are built from the ground up. For each configuration, the entire set of test data is then iterated over, and estimations are made for each value and for each applicable operator. The ground truth is also computed for each, and compared to the estimate to generate an error value. These error values are then collected and make up the basis for which the configuration is rated against the competing configurations, at least in terms of accuracy.

4.3 MySQL Implementation

While the Python prototype is interesting on its own, I had as a goal from the very start of the project to implement much of the same logic in MySQL. The reason for this is simple: the structure is not intended to be used as a standalone project – it was invented and designed with the express purpose of integrating it into a database system. And while the Python prototype does provide some data, both in terms of the feasibility of an actual implementation and in terms of performance, the database factor is still at its core. Implementing the design in MySQL allows for a much more realistic view of the work and challenges that would face a proper integration of the design into a database system. Additionally, while the measurements obtained with the Python prototype are valuable, they are not a substitute for measuring the estimates made by MySQL itself when using these data structures, as well as how these estimates impact MySQL's performance. And as become clear late in this section, there are several differences between the Python and MySQL/C++ prototypes. And while many of these differences arose due to time/resource constraints, others stem from the design of MySQL and the C++ language.

I will being this section with a discussion of the work that was done to create a working (though minimal) implementation of the proposed JSON statistics structure in MySQL. Starting with loading the JSON statistics into MySQL, I will go on to discuss the design of the structures used, before looking into how selectivity estimations are made.

Please note that the final MySQL prototype does not generate its own histogram data. However, once such data has been inserted, the prototype is able to store and retrieve it and use it to make accurate estimates both for use in estimating the selectivity filtering predicates and join conditions (with some caveats – see section 4.3.4).

4.3.1 Inserting Custom Histogram Data in MySQL

As discussed in section 2.4, MySQL has an existing statistics system that is responsible both for the creation and interpretation of histograms. The SQL command `ANALYZE TABLE t UPDATE HISTOGRAM ON col_a;` will lead to MySQL gathering statistics for the column col_a. The computed histogram is stored in an internal table as a JSON document. When retrieved (for use in selectivity estimation), the JSON representation is converted into an in-memory representation of a histogram and is then queried for the selectivity of the given predicates. With the release of MySQL version 8.0.31, released on October 11th, 2022, the `USING DATA` optional parameter was added to the `ANALYZE TABLE` statement, allowing users to insert their own histogram data as a JSON string [3]. The inserted data must conform to the same structure and standards as the histograms that MySQL itself generates. For example, the JSON data type is not supported for histograms in MySQL 8.0.31, so inserting custom data for a column with a JSON data type is not allowed. These hurdles were overcome relatively quickly however, and the new `ANALYZE TABLE` statement greatly simplified the process of loading custom histograms into MySQL.

The structure of the data that is inserted to create a histogram is exactly like the one that is returned when querying for the histogram data in the `column_statistics` table, and can be seen in listing 12.

4.3.2 Setting Up the Basic Structures

The shared functionality of MySQL `histogram` module is defined in the file `sql/histograms/histogram.h`. The file defines a base `Histogram` class, error reporting logic and structures, and some public functions for finding the histogram associated with a table and column. Common properties defined in the base class include the sample rate, the histogram type, the type of the data itself, and an arena allocator onto which the histogram contents are loaded. The two existing histogram implementations, `Singleton` and `Equi_height`, then define their own histogram bucket structure. The base Histogram class is quite flexible, and as long as the implementing class conforms to the interface of the base class, it can basically put any kind of data in its actual buckets. This is in many parts due to MySQL's histogram system storing its data in a JSON column.

```
> CREATE TABLE test(i INT);

> INSERT INTO test VALUES(1);
> INSERT INTO test VALUES(2); -- repeated once
> INSERT INTO test VALUES(3); -- repeated twice
> INSERT INTO test VALUES(5); -- repeated four times
> INSERT INTO test VALUES(100);

> SELECT HISTOGRAM FROM information_schema.column_statistics WHERE TABLE_NAME = 'test' AND
↪  COLUMN_NAME = 'i';
HISTOGRAM
{
    "buckets": [[1, 0.0833], [2, 0.25], [3, 0.5], [5, 0.9166], [100, 1.0]],
    "data-type": "int",
    "null-values": 0.0,
    "collation-id": 8,
    "last-updated": "2023-06-19 08:53:37.166280",
    "sampling-rate": 1.0,
    "histogram-type": "singleton",
    "number-of-buckets-specified": 100
}
```

Listing 12: The resulting histogram (singleton) data after running the command `ANALYZE TABLE test UPDATE HISTOGRAM ON i;` on a small integer column. The JSON representation of the equi-height histogram is very similar – the "histogram-type" value becomes "equi-height", and the structure of the buckets changes slightly.

The JSON representation of the histograms aligns quite closely with how MySQL represents them internally. An example of the JSON representation of a MySQL singleton histogram can be seen in listing 12.

Because the Singleton class was the simplest of the two, it was chosen as the starting point for the implementation of the JSON histogram structure. Specifically, the `Singleton<String>`[6] specialization was used, as the bucket value could then represent the key path that the statistics were collected for. The `cumulative_frequency` struct member was kept, though its name was changed to `frequency` to reflect the fact that in the Json_flex structure, each top-level bucket is basically an independent and complete histogram similar (but not identical!) in structure and function to MySQL's existing singleton and equi-height histograms. Because the JSON histogram only deals with a single columnar data type (JSON), all templating and other type information was dropped from the Json_flex class. A few minor modifications had to be made to the common histogram logic to allow histograms with `"data-type": "json"` and `"histogram-type": "json-flex"` to be inserted. After these changes were made, inserting custom histograms for JSON data was technically possible, though the allowed structure was very limited. The remainder of the structuring work was spent primarily on developing the structure. The base Singleton bucket structure that was used as a base had only two members, while the design had 8 (or 9): path, frequency, null frequency, minimum value, maximum value, distinct values count, and the histogram (and mode information).

Adding new members to a struct is simple as long as their type is fixed. However, while the table column type is always JSON, the type that a path in the JSON statistics structure leads to can vary between any of the primitive JSON values (except null). While most of the members in the struct are of a fixed type (path is a String, frequency a double, ndv an int), the minimum and maximum value members (as well as the values in the histogram buckets, as will be discussed later) are not. Additionally, some of the members are dynamic length arrays (vectors) and thus require special attention because they will be separate from the struct in memory.

To address the first issue (differing types), a tagged union type was created to hold all possible primitive JSON values. It can be seen in listing 13 with the name `json_primitive`. The enum `BucketValuesType` is used to indicate which member is correct. The second issue requires slightly more work to get around.

If the type that the path leads to does not require any additional allocations – as is the case for double, integer, and boolean values – the value can simply be added to the bucket struct with no extra work. The issue is that in addition to the C++ fundamental types, JSON primitive values may also be strings, which due to their dynamic size must be allocated for at runtime. Because the string value, as a part of JSON string representation of the histogram, only lives for the duration of the query, the string value must first be copied into an area of memory that has a lifetime that is as long as or longer than that of the histogram object. This was done by allocating string values onto the same MEM_ROOT that the histogram itself is allocated on. Care also had to be taken when histograms were being copied (as they are each time one is inserted) to make sure that string values were copied onto the MEM_ROOT of the newly created copy.

A similar challenge arose when implementing the histograms belonging to each path. Because the histograms themselves contain a dynamically sized array of buckets, each histogram had to be specially allocated on the MEM_ROOT just as with the string values. One must also be careful to handle the string values inside those histograms in the same

[6] `String` is a string implementation internal to MySQL.

manner. A small difference between the histograms and the JSON primitive union is that the histogram values are implemented using generics rather than a union. The enum used for the primitive union doubles as an indicator of the specification of the histogram (if a histogram is present).

Because none of the objects described here are particularly long-lived, further memory management like reducing the memory footprint and avoiding leaks could be approached with a lax attitude. When the owner of the arena allocator dies, all the memory allocated onto MEM_ROOT is freed.

An alternate approach would have been to use fixed-size arrays for both strings and histograms, but this approach would have resulted in a massive use of space for any realistically allowed number of elements. This is because each json_primitive member would take up as much space as the fixed string buffer, and each histogram would contain the maximum allowed number of buckets. These two combined would result in both a massive use and a massive waste of space.

The fit between Json_flex and the Histogram interface isn't the best. This is because Json_flex requires more information to make its estimates than the other Histogram implementers, and as such they do not have the exact same interface. Whenever we want an estimate for JSON data, we have to detect this and specifically call Json_flex methods. Another place where this became an issue was when implementing NDV estimations for joins. Here, the caller must specifically cast the histogram to Json_flex and call a Json_flex-specific method for estimating ndv. This could be rectified by changing the Histogram interface so that the information needed by Json_flex is passed to a common ndv estimation call.

4.3.3 The Final Structure

The final set of data structures used in the MySQL prototype consists of a top-level class inheriting from the Histogram class, as well as two (main) structs that contain the actual statistics data. In addition, three additional structs and two enums were needed to make things come together. I will now present these structures in detail, explaining how they come together to implement the solution and how each of them fits together. A compressed version of the actual code can be seen in listing 13, while figure 3 shows a simplified class diagrams showing the relations between the different structures.

The class `Json_flex` (defined on line 61 of listing 13) inherits the common Histogram class and as such acts as the entry point and interface for building and querying the JSON statistics structure. The Json_flex class itself, while implementing the bulk of the prototype's logic, does not contain much interesting data in and of itself. The first member of note is the (global) minimum frequency, which tracks the smallest frequency value encountered while building the histogram. The minimum frequency is used during selectivity estimation when no data is found for a given path. The second member is a dynamic array (allocated on the MEM_ROOT arena allocator) containing all the remaining statistical data. Each array element is a `KeyStat` (defined on line 42 of listing 13) struct. Each KeyStat struct stores the statistics for a single key path in the JSON statistics document. The struct has three required members which will always be present: the frequency of the path, the frequency of null values (as a ratio 0-1) of the path, and the type of the path. The type of the field is indicated using the `BucketValuesType` enum which defines five values: one for each of the four non-null primitive JSON types, plus the value `NOTHING`. The NOTHING value indicates that the KeyStat struct holds no values from the original

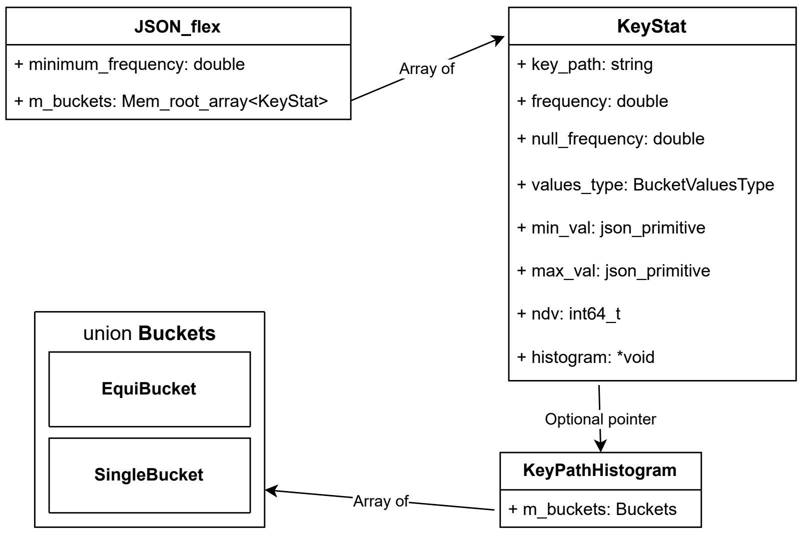


Figure 3: The classes and structs that make up the JSON statistics implementation in MySQL.

JSON path. Additionally, the KeyStat struct has four optional members: the minimum and maximum values encountered for the path, the number of distinct values counted for the path, and a histogram. As mentioned in section 4.3.2, the minimum and maximum values can be any of the four non-null primitive JSON types, and this is facilitated by the use of a union type called `json_primitive`. The minimum, maximum, and ndv members are all wrapped in an `optional`[7] object to make explicit the fact they may not be defined. Due to its potential size, the histogram is not a contiguous part of the KeyStat struct. The histogram is allocated separately (though in the same arena as all the other structures), and the KeyStat struct only holds a pointer (which may be null) to it. The key path histogram itself is defined in the `KeyPathHistogram` (defined on line 17 of listing 13) struct. This struct is both generic to any of the allowed JSON primitive value types and may be either a singleton or an equi-height histogram. The KeyPathHistogram struct also does not contain histogram bucket data directly. Instead, it holds a pointer to an array allocated on MEM_ROOT and of which each element is a histogram bucket. The histogram buckets themselves have the exact same structure as in the Python implementation, although one member has been slightly changed: Instead of holding a count of the number of values that fit in the bucket, the buckets hold the frequency of the value, as a fraction of the total frequency of the key path. The bucket array, like json_primitive, is a tagged union and the caller must check which type of histogram they are accessing before reading the buckets array either as an array of singleton buckets or as an array of equi-height buckets. Finally, the KeyPathHistogram struct has one field that is not present in the Python prototype. The `rest_mean_frequency` member is used in conjunction with singleton buckets when there are more elements than the singleton can hold, but an equi-height cannot be used (basically, for string values). The singleton histogram then holds the most frequent values, and the mean frequency of the remaining values is stored in the `rest_mean_frequency` variable. In other words, it fulfills the same function as the ”singleton plus” structure in the Python prototype.

Due to time constraints, the JSON statistics structure implemented in MySQL does not support the *mode* information that that the Python prototype collects when running with the NDV_WITH_MODE statistics setting. However, I believe that what is implemented

[7] https://en.cppreference.com/w/cpp/utility/optional

is enough to cover all interesting bases when it comes to measuring the performance and accuracy of the solution.

A final thing to note is that while the existing histogram buckets (singleton and equi-height) store cumulative frequency, the KeyPathHistogram buckets store point frequency (i.e., they store the frequency of that key only, independent of the frequency of the other keys). This has to be the case because each top-level bucket is basically a separate complete histogram like the histogram of a column over any other type is in MySQL. The histogram (”KeyPathHistogram”) buckets themselves could have used cumulative frequency, but they do not. The reason for this is that it simply matched the Python implementation better to store point frequency, and thus made translating both the code and statistics themselves simpler.

4.3.4 Selectivity Estimation with Json_flex

Internally, MySQL represents all (relational) expressions in its queries as *Items*[27]. Item classes are defined for constant literals, identifiers, fields, variables, and more. For example, the Item subclass `Item_func_eq` represents the equality comparison operator. Because the equality operator is a binary operator, the equality item has two child items, one for each side of the equality expression. Each item class implements a method `get_filtering_effect`, whose responsibility it is to figure out the selectivity of the expression that the item represents. And this function may come to use MySQL’s histogram statistics. However, before this function is even called, the system checks whether it is even possible for the item in question to contribute to the filtering effect. In the current release of MySQL (8.0.32), function calls that act on JSON data do not pass this test of contribution, which means that selectivity estimation is never attempted when these function calls appear inside an expression. In fact, any filtering expression involving MySQL’s JSON functions was found during early testing to return a filter effect of 1 (meaning no effect)[8]. Accessing JSON data without the use of JSON functions is very limited and basically invalidates the point of implementing a complete JSON data type.

To enable selectivity estimations to be made using the JSON statistics structures, this had to change. The changes that enable selectivity estimations to be made for expressions involving JSON functions, however, mean that the prototype alters the behavior of MySQL even when no statistics are present for the JSON data being accessed. The system attempts to make estimates no matter whether statistics are actually present or not, whereas it never attempted to do so before. The MySQL prototype, just like the Python prototype (see section 4.2.4), implements fallback heuristic constants which are used when no data is found. Thus, the estimates for certain predicates may change drastically from the base 8.0.32 version of MySQL, even if no JSON statistics data has been inserted.

This change was not the goal of the project/thesis and is not especially interesting with respect to the stated goals of the project. However, it is important to be aware of this fact as it will affect the change in performance of the system when compared to the base 8.0.32 version of MySQL. This will be discussed in more detail in section 6.

The entry point for selectivity estimation for histograms is the Histogram method `bool get_selectivity(Item **items, size_t item_count, enum_operator operator, double **selectivity_buffer)`. This function previously operated only on either the

[8]Why this is, I cannot say for certain. Investigating exactly why the heuristic filtering effects are not applied when JSON functions are involved was outside the scope of this project.

```cpp
// Simplified version of the MySQL String class with no ownership of the data it points to.
// This makes it much simpler to use in a union with the other primitive data types.
struct BucketString {...};

union json_primitive {
  double _float;
  int64_t _int;
  bool _bool;
  BucketString _string;
};

// An actual histogram. Contained by Json_flex's KeyStat
// Allowed type specifications are: double, int64_t, bool and BucketString
template<typename T>
struct KeyPathHistogram {
  enum class JFlexHistType {
    SINGLETON, EQUI_HEIGHT,
  };
  struct SingleBucket {
    T value;
    double frequency; // As a fraction of the total frequency of the key_path
  };
  struct EquiBucket {
    T upper_bound;
    double frequency;
    int64_t ndv;
  };
  union Buckets {
    Mem_root_array<SingleBucket> single_bucks;
    Mem_root_array<EquiBucket> equi_bucks;
  };

  JFlexHistType buckets_type; // Indicates whether this is a singleton or equi-height histogram
  Buckets m_buckets;
  std::optional<double> rest_mean_frequency = std::nullopt;
};

// Keeps the information related to a single statistics path.
// Not a traditional histogram bucket, despite the (admittedly bad) name.
struct KeyStat {
  enum class BucketValuesType {
    NOTHING, INT, FLOAT, STRING, BOOL,
  };

  // Required members
  const String key_path;
  const double frequency;
  const double null_frequency;
  const BucketValuesType values_type; // Indicates the type of the min/max_val and histogram members

  // std::Optional members
  const std::optional<json_primitive> min_val;
  const std::optional<json_primitive> max_val;
  const std::optional<int64_t> ndv; // number of distinct values
  void *histogram; // optional pointer to a histogram (KeyPathHistogram)
};

// Top-level class that inherits Histogram and is called
class Json_flex : public Histogram {
  Mem_root_array<KeyStat> m_buckets; // Array of KeyStat structs
  double minimum_frequency = 1.0;    // Set to the smallest frequency encountered during creation.
  ...
}
```

Listing 13: All data structures used for the internal representation of the JSON statistics. Some minor details have been left. The complete code can be found in the file `sql/histograms/json_flex.h` in the supplied code or on repository's Github page:https://github.com/jonaengs/mysql-server.

combination of a field item and a constant item or only a field item in the case of unary operators like `IS NULL`. A field item represents some value/item in a record (or all of a specific item/value in all records of some table) – usually a column (Norvald: is this correct?). However, when called to estimate selectivity for JSON data, the field is always wrapped inside one or more function call items (because, again, we don't just read the JSON column, we access specific parts of it using MySQL's JSON_* functions). This is incompatible with what the other Histogram implementations expect. Therefore, when the `get_selectivity` function detects that one of the passed items is a function item, it immediately delegates the remaining work to the Json_flex class which handles everything internally. This stands in contrast to how much selectivity estimation logic is shared between the other two histogram classes Singleton and Equi-height.

The Json_flex selectivity estimation function is relatively simple. It has three main tasks: first, it must determine whether the JSON function results in a quoted or an unquoted value, i.e., whether the expression returns the value as encoded in the JSON document, or whether MySQL will attempt to convert any strings to the literal that the string value represents. This is important because when collecting the statistics, the Python prototype does not do any conversion of string values, meaning it leaves string values as quoted. This is because we assume that data is generally stored as the type that it is supposed to represent[9], and that values that are represented as strings are meant to be interpreted as strings. Changing the system to assume and work with unquoted values does not require any large changes, however. Next, the function must construct the path that will be used to query the statistics. If the value is unquoted, a type suffix will not be added to the final key, so the statistics for all possible types will be used (the downside is of course that the statistics for this path are less detailed because only the frequency and null frequency is stored for these paths). Finally, the function examines the comparand and operator arguments to determine which selectivity estimation function to dispatch to for estimation.

The selectivity estimation process itself is not very complex. First, an attempt to find statistics matching the generated path is made[10]. If nothing is found, then the recorded minimum frequency is used as a base and then multiplied with one of the heuristic constants like in the Python prototype (section 4.2.4). If statistics are found, then the function proceeds in a manner quite similar to the estimation functions used in the Python prototype. First, the code checks to see whether the value is within the recorded range. If not, a selectivity of 0 is returned. If histogram statistics are available, it uses the histogram data to make an estimate. Otherwise, the *ndv* statistic is used where applicable, before finally falling back on the frequency and null frequency information. See listing 25 in the appendix section B for an example. For untyped paths, only the frequency and fromnull frequency information can be used to make the estimate.

Due to time constraints, statistics for arrays were not implemented in the MySQL prototype. Support for the functions JSON_CONTAINS, JSON_MEMBEROF, and JSON_OVERLAPS is also missing. However, the MySQL prototype supports the "IN LIST" operation (`... WHERE col IN (1, 2, 3);`), which is somewhat similar.

[9]The exception to this is data types that are not supported by the JSON standard, such as date strings. However, none of the two prototypes currently support any such values. If support for dates were to be added, the system may have to change to using unquoted JSON.

[10]This is done by doing a linear scan of the KeyStat array, not by way of a dictionary lookup like the Python prototype does. The reason this was done was that constructing the array was simpler than using a hashmap, and time was limited. In a proper implementation, I would recommend using a hashmap for lookups, or at least doing a bisection search of the array.

In total, there are six "fundamental" selectivity estimation functions implemented: equal to, less than, greater than, exists, equals null, and not equals null. All other operations, like 'greater than or equal to' and 'between' are implemented using these functions.

Another feature supported by the MySQL prototype but which the Python implementation does not support is making join estimates, though this is only the case when the hypergraph optimizer is enabled (which required some small changes, as the hypergraph optimizer is hard-coded to be disabled for release versions of MySQL). This was relatively simple to implement, as the histogram module only needs to supply the estimated number of distinct values, and the hypergraph optimizer will take care of the rest of the calculations. The existing Histogram class method for getting the *ndv* stat does not take any arguments, however, as the method is little more than a histogram bucket count. When estimating the *ndv* for a join condition on a JSON function, however, it is impossible to make a good estimate without knowing the path argument of the function. Therefore, a separate method had to be added to the Json_flex class to support passing information about the field and the function accessing it.

One final difference between the Python and MySQL/C++ versions is how null estimation is handled. In the Python prototype, the "is null" estimation function estimates the number of times that the given key path leads to a null value. The "exists" function estimates the number of documents in which the key path occurs at all. In MySQL however, a third case has to be handled: the document itself may not be present for the row – i.e., it may be NULL. The Python prototype does not handle this case, as it makes little sense for a row itself to be null when examining only a collection of JSON documents. It is important to note here that the JSON "null" and the MySQL "NULL" are distinct entities and are not interchangeable. At least, this is how MySQL views it. Additionally, when accessing the value of a key that is not present in a JSON document, MySQL will return a NULL for that row. Therefore, MySQL's IS NULL acts more like the "exists" query of the Python prototype, letting those rows pass where either the key path is not present in the document or the document is not defined. If the key is present in the document and leads to "null", this does not count towards MySQL's IS NULL. However, using the function JSON_VALUE, JSON "null" values can be converted into MySQL NULL values. However, if NULL would already be returned, this does not change. Therefore JSON_VALUE($path) IS NOT NULL acts like the Python "is not null" function, but the JSON_VALUE($path) IS NULL does not act like the Python "is null" function, because it includes missing key paths and missing documents. [11]

4.3.5 The Complete Process

When a histogram is inserted into MySQL as a JSON histogram literal, the first that happens is that the JSON string is parsed and wrapped in an internal class for handling JSON data. This wrapper class is then passed on to the histogram module which begins the construction of a histogram object matching the JSON data. First, the base Histogram class constructs the common top-level properties, before handing the creation of the bucket data over to the implementing subclass. This construction step involves a lot of verification of properties and values and features an error-handling system that returns detailed and specific errors if encountered. Finally, the histogram is converted back into JSON and

[11] The simplest way to get the Python "is null" behavior I think is to do equality with a JSON null literal, if that is possible – I couldn't figure out how when I looked into it. The other approach is to do equality on the *type* of the value and the string "null", which I think is the suggested approach to this problem. However, detecting this exact case in the selectivity estimation code is quite inelegant.

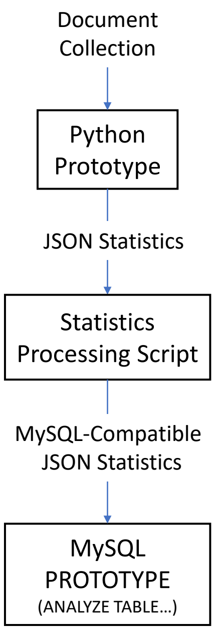


Figure 4: The process of converting a document collection into statistics for the Python prototype, and then converting those statistics into a format that is acceptable to MYSQL, before inserting that data in a query.

stored in the statistics table, and a success message is returned to the client.

Before selectivity estimation can be performed, the stored histogram JSON document is retrieved and converted into a histogram again. A common method, `get_selectivity` is then called, and performs some verification of the validity of the arguments before passing them on to the implementing subclasses where the actual selectivity estimation is performed.

Inside Json_flex, a query path (the key to use for the statistics object) is constructed before the argument items, their types, and values, are inspected to determine how and which selectivity estimation dispatching function to call. The selectivity dispatch functions then inspect the filtering predicate's operator to determine which of the ”primal” selectivity estimation functions to use to make the estimate. One or more of these functions are then called and their results are combined (if several calls were made) to get a final estimate that is returned to the caller.

4.3.6 Putting it All Together

The MySQL prototype does not have the capability of generating JSON statistics from scratch on its own and instead relies on this data being given to it through the aforementioned histogram insertion query statement. For the experiments in section 6, the statistics used are therefore generated by the Python prototype before being inserted into MySQL. Because the exact structure and contents that the MySQL prototype expects differ somewhat from what the Python prototype generates, a script to convert between the two structures was created. The script processes the Python prototype statistics in the following way:

1. Converts the cardinality count to a frequency value, where the frequency indicates how many percent of the complete document collection the key path is present in. The null count is also converted to a number in the range $[0, 1]$ indicating the frequency with which the key path, when present, leads to a null.
2. This conversion from count to frequency is also applied to all buckets.
3. The `'rest_frequency'` property is calculated for the histograms that were of the type "singleton-plus".
4. All strings inside the buckets, both keys and values (and as parts of values) are encoded as base64 and prefixed with the string `"base64:type254:"`, as expected by MySQL's JSON string values.
5. The sampling rate is taken from the meta statistics object and set as the value of the `"sampling-rate"` key in the top-level histogram string.

5 Experiments With Python Prototype

The real test of the efficacy of the proposed design is whether it improves MySQL's estimates and choice of query plans. However, experiments with the Python prototype can still result in valuable information. The experiments on the Python prototype are also much simpler and cheaper to run, so the number of configurations that can feasibly be tested is much greater.

The goal of the Python prototype experiments is to learn more about the following:

- Can the proposed structure actually be used to make estimations for predicates over JSON data?
- What is the cost, in terms of storage, of the different statistics types?
- How are estimations affected by sampling?
- How are estimations affected by pruning?

The remainder of this section is organized as follows: First, the experimental setup will be examined. I will be presenting the testing strategy, the data used, and various drawbacks and caveats that apply to the approach taken. Next, the experimental results will be presented, examining various aspects of the efficacy and cost of the statistics and how these are affected by the tunable parameters. Finally, a broad discussion of the results will follow.

5.1 Experimental Setup

The experiments were run an a computer with the following specs.

- Ubuntu 20.04.6 LTS

- 16GB RAM
- Intel® Core™ i7-8550U CPU @ 1.80GHz × 8
- Python 3.11.3

The quantities measured for the Python prototype experiments do not depend on the underlying hardware in any significant manner, however.

5.1.1 The Data

Two datasets were used for the Python experiments. The first, referred to as the Twitter dataset, is a collection of tweets from the 2014 Recys challenge[8, 39] in JSON format. Of the three datasets published for the challenge, the largest ("training") was used. The collection consists of 170285 documents, totalling 416 MB of data. The data is structurally heterogenous, with many paths occurring with a very low frequency. In total, the Twitter data has 809 unique key paths.

The second dataset, referred to as the Airbnb dataset, is a collection of information about Airbnb listings published by the Inside Airbnb organization [16]. Specifically, the collection used for the Python experiments was the listings data for Los Angeles published in on the 7th of March 2023 (accessed on June 13th, 2023). The collection has a total size of 189 MB and contains 42451 documents with 448 unique paths.

The Twitter dataset has no arrays eligible for array statistics collection. The Airbnb dataset on the other hand contained two such key paths: "host_verifications" (see figure 9) and "amenities".

The structure of the documents in these two collections is explored in further detail in section 6.2.

While the Twitter dataset is structurally heterogeneous to an extent, finding eligible JSON document collection datasets that were heterogeneous with respect to typing (same path leading to different types) proved a major challenge. This precluded the possibility of testing the impact of keeping only the most frequent type per path versus keeping all types per path. Whether this is a large issue or not is hard to say. A lack of of real-world queries against these types of datasets made it hard to tell whether type heterogeneity is common and how it is handled by developers – i.e., whether queries specify which data type they are looking for or not.

The experiments included testing the impact of sampling on the accuracy and size of the statistics. Table 1 contains an overview of the number of documents expected to be sampled for the different sampling rates. Please note that the method of sampling used by the Python prototype is not the exactly same as the one used by MySQL. MySQL samples on a per-block basis, while the Python prototype implementation does it on a per-row basis. Because there is a chance of inter-block correlation, MySQL's method of sampling may result in slightly skewed results. The Python prototype does not have this problem. Thus, it is possible that the impact of sampling measured for these experiments may be slightly more positive than it would be in a proper implementation where MySQL is responsible for creating the statistics.

Sample Rate	Twitter	Airbnb
1.0	170285	42451
0.3	51086	12736
0.1	17029	4245
0.02	3405	849
0.005	851	212

Table 1: Expected number of documents sampled per dataset and sampling rate.

5.1.2 The Testing Scheme

The experiments for the Python prototype were run as follows: First a dataset was selected for testing, before a list of all combinations of settings (parameter values) intended to be tested was generated. Then, for each such setting, statistics were collected.

1. Select a dataset for the experiments to be executed for.
2. Collect test values for the dataset.
3. Generate a list of all combinations of settings (parameter values) intended to be measured.
4. For each settings combiation, do the following:
    (a) Apply the settings.
    (b) Collect new statistics for the dataset using the given settings.
    (c) Iterate over the collected test values and measure how well the statistics can be used to estimate the selectivity of each of the supported operators.

The test values are gathered from the source dataset used for the experiment. For each key path, the *set* of values corresponding to that key path is sampled without replacement. No other values are used. This means that no values that are not present in the dataset are used as test values. This may have both a positive and negative on the measured accuracy, as unseen test values that are outside the encountered range of values for a path will always be correctly estimated to 0, while unseen test values inside the range are likely to have less accurate estimates than the test values generated from the dataset. The choice of sampling from the set of values rather than the list of values for each key path is likely to lead to a lowered measured accuracy as the statistics keep better track of popular values than unpopular values (NDV with mode and histogram do, at least).

The operators supported by the Python prototype can be divided into three categories. First are the unary operators, whose only arugment is a key path. These are: 'exists', 'is null', and 'is not null'. Then there are three scalar binary operators: equality, greater than, and less than. These take a path and a constant scalar value. Finally there are the three array operators, which compare an argument value (scalar for 'memberof', arrays for the two remaining) against arrays in the data set. These are 'memberof', 'contains' and 'interesects'. Due to the nature of the data and the operators themselves, the distribution of test values, and thus measurements, is not even between the different operators. For the two data sets tested, there were more operands available for the scalar binary operators than the other two, and more operands available for the unary operators than for the

```
('Central heating', 'Smoke alarm', 'Washer', 'Shampoo', 'Kitchen',
↪  'Hair dryer', 'Dedicated workspace', 'Dryer', 'Free parking on
↪  premises849', 'Wifi', 'Indoor fireplace', 'Carbon monoxide
↪  alarm', 'Bathtub', 'Iron', 'TV', 'Refrigerator', 'Essentials',
↪  'Cleaning products', 'Coffee maker', 'Keypad', 'Dishes and
↪  silverware', 'Hot water', 'Self check-in', 'Body soap', 'Cooking
↪  basics')
```

Listing 14: Example test value gathered from the Airbnb dataset and used to test estimations for the 'contains' and 'includes' operators.

array operators. Additionally, there will be more operands available for testing with the equality operator than for the inequality operators. This is because the equality operator accepts all four scalar value ('primitive') types, while the inequality operators only accept numeric values. For the unary operators, there will be as many operands as there are key paths. For example for the Airbnb data set the following number of tests were performed for each operator:

- Equality: 3967
- Greater/less than: 875
- Unary operators: 448
- Array operators: 80

Whether this is a realistic distribution of operations as compared to real-world situations is of course impossible to say as such things differ wildly. In the visualizations which aggregate accuracy across operators, the error is combined through a non-weighted mean unless otherwise stated.

The configurations tested were the cartesian product of the sets of options seen in figure 15. Invalid or redundant configurations were dropped. In total, 468 different configurations were tested. MAX_NO_PATHS and MIN_FREQ were not tested together because it was found that one of the two would almost always take "precedence" and make the inclusion of the other moot.

5.1.3 Applicability

The results from these experiments are unlikely to match the real-world experience if one were to implement this system in MySQL. It is unlikely that all paths in a document collection are accessed with the same frequency. It is also unlikely that the constants used for the binary operations will be distributed in randomly in the universe of values. Finally, the frequency of each operation used in these experiments is also unlikely to match that of a real-world scenario. The results of the experiments in this section therefore, while accurately measuring the quality of the predictions that the statistics provide, may not necessarily match well with the gains in estimation accuracy experience when using the same statistics in real-world scenarios. However, I believe that the results are still useful for comparing the different configurations and measuring their performance against each other. – The absolute values of the numbers is meaningless, but the relative difference between them is still valid.

```
{
    "stats_type": [st for st in StatType],
    "prune_strats": [
        [],
        [PruneStrat.MAX_NO_PATHS],
        [PruneStrat.MIN_FREQ],
        [PruneStrat.MAX_PREFIX_LENGTH],
        [PruneStrat.NO_TYPED_INNER_NODES, PruneStrat.UNIQUE_SUFFIX],
    ],
    "num_histogram_buckets": [6, 16, 32, 64, 128, 512],
    "sampling_rate": [0.0, 0.7, 0.9, 0.98, 0.995],
    "prune_params": [
        {
            "min_freq_threshold": 0.01,
            "max_no_paths_threshold": 200,
            "max_prefix_length_threshold": 4,
        },
        {
            "min_freq_threshold": 0.003,
            "max_no_paths_threshold": 400,
            "max_prefix_length_threshold": 4,
        },
        {
            "min_freq_threshold": 0.001,
            "max_no_paths_threshold": 500,
            "max_prefix_length_threshold": 5,
        },
    ],
}
```

Listing 15: The object used to generate the all the different configurations used during testing.

5.1.4 Experimental Data Collected

For each combination of settings, the following data was collected:

- For each operator and test value: the true and estimated cardinality.
- The settings used when collecting and processing the statisticss.
- The space required to store the collected statistics.

Some information on time use and memory consumption during statistics creation was also collected.

5.1.5 Exclusion

From the experimental results, it is clear that there is little benefit in using the BASIC statistics type – it has the worst accuracy in general, and it is not significantly cheaper to compute than the other non-histogram statistics (see figures 5 and 6). We can also see that there seems to be little advantage in using BASIC_NDV over HYPERLOG_NDV (see also section 5.2.4). Because the latter is cheaper with respect to memory costs, it almost always will be preferable to the former. Because they are outperformed by the other statistics approaches, BASIC and BASIC_NDV will therefore be left out of most figures.

Additionally, the operations EXISTS, IS NULL and IS NOT NULL all use the same statistics that are available at the same level of granularity for all statistics types. For this reason, these operators have been left out of several of the figures in this section.

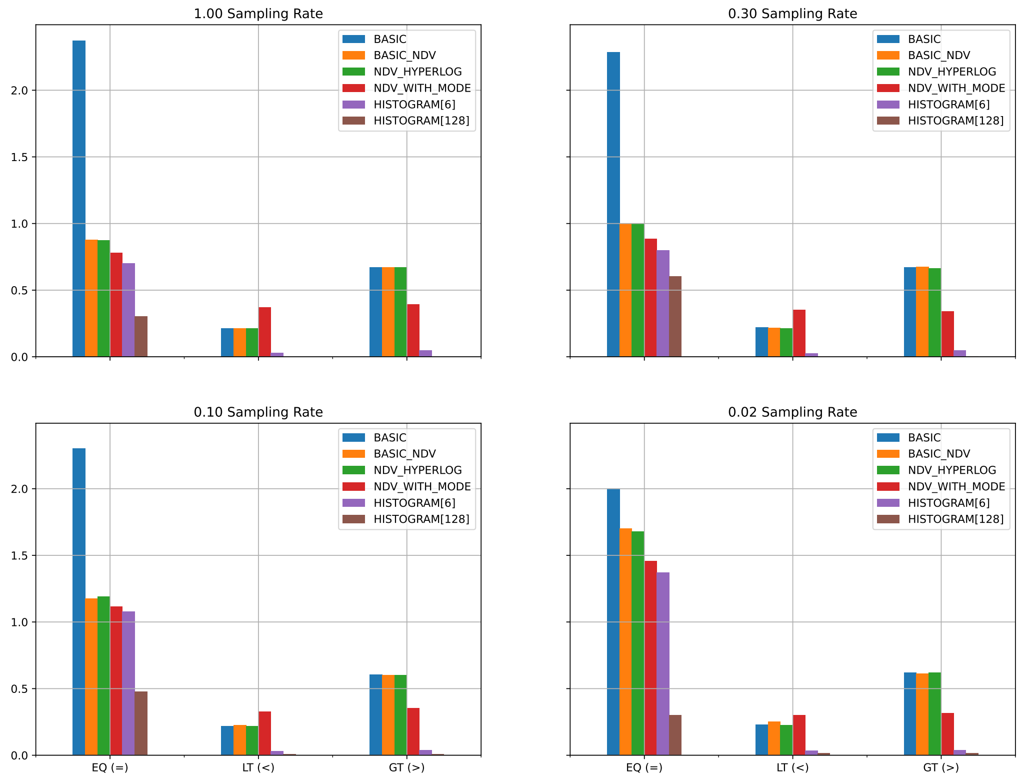


Figure 5: Airbnb. Median estimation error per statistics type for the different sampling rates. The difference in error between the 'less than' and 'greater than' operators for NDV_WITH_MODE statistics type stems from the randomly selected test values favoring one operator over the other.

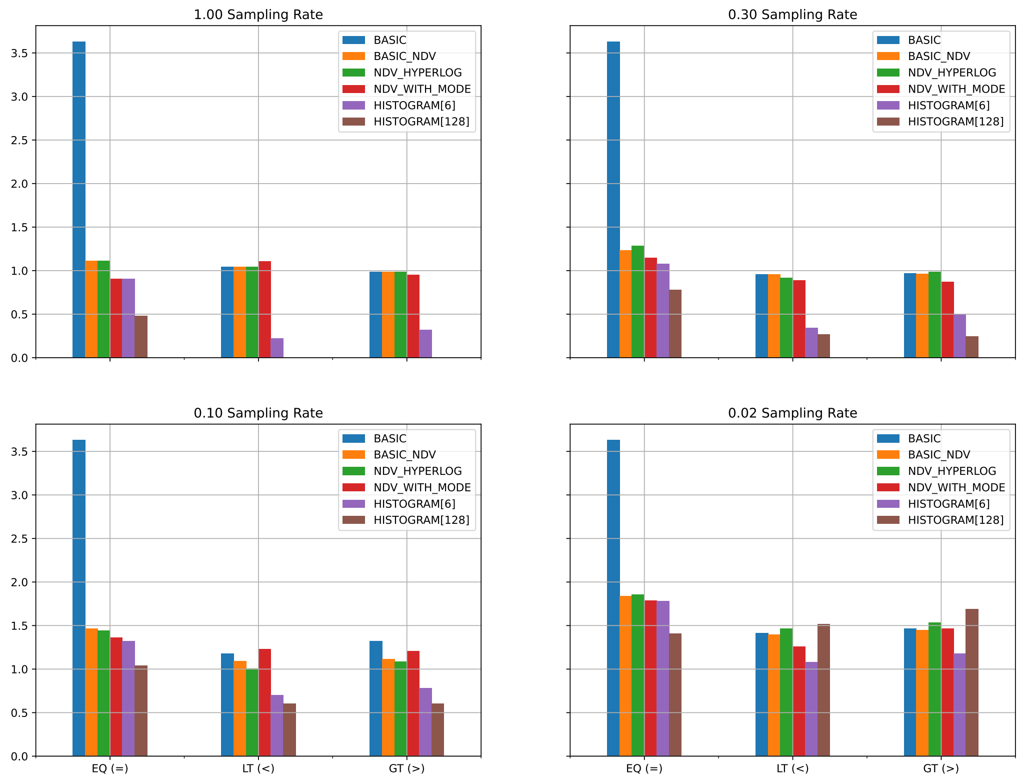


Figure 6: Twitter. 90th quantile estimation error per statistics type for the different sampling rates. NDV and mode statistics are of little use when estimating selectivity for inequality operators. Histogram statistics do well for these types of operators however.

5.2 Overall Results

Figures 7 and 8 summarize the results gathered from the Python prototype well. Unsurprisingly, histogram statistics are the most accurate, and the more histogram buckets used, the more accurate the estimates using histogram statistics become. Diminishing returns quickly kick in however, so doubling the maximum histogram size will not lead to the estimation error being halved. Of course, histograms have an additional cost when it comes to storage compared to the basic and NDV variants. Interestingly, though not unexpected, the size/accuracy tradeoff differs between the two test datasets.

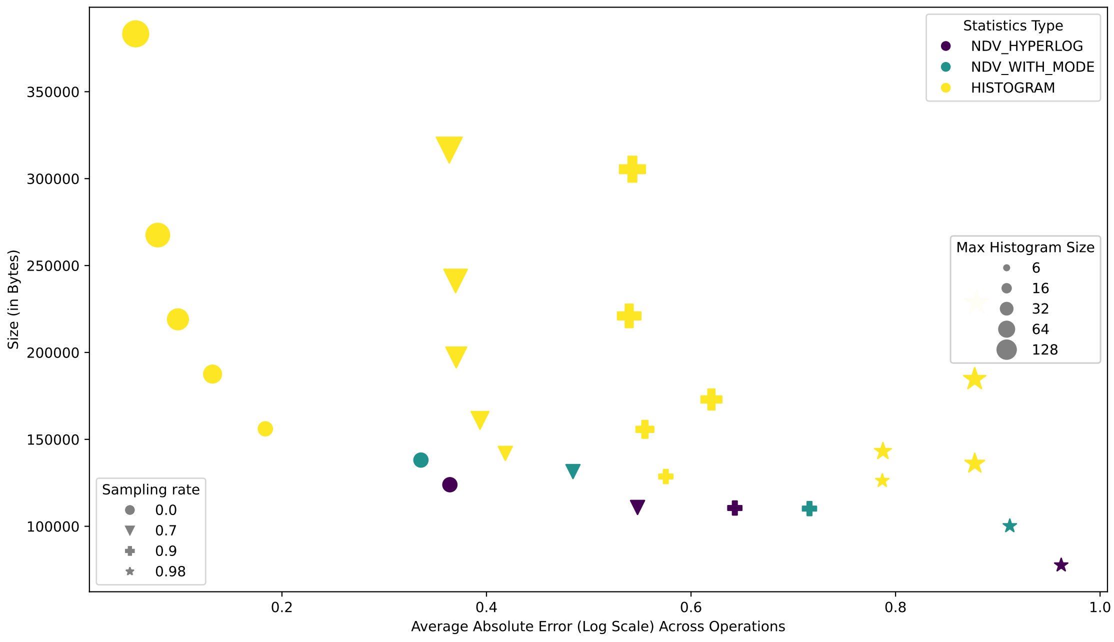


Figure 7: Comparison of the storage size required for the different statistics types using the Twitter dataset. No Pruning techniques used.

In some cases it can be observed that as the sampling rate decreases, the histogram statistics object increases in size. This is due to more singleton histograms being created. This happens because singleton histograms are only created if the number of distinct values seen for a key path is at or below the maximum allowed histogram bucket count. When sampling, enough of these unique values were then not encountered so that the distinct values count fell below the threshold. The algorithm then decides to create a singleton histogram for that key path. It should be noted that this only happens to key paths leading to string values, as equi-height histograms will be created for numeric values if there are too many to fit in a singleton histogram. In general, reducing the sampling size will reduce the size of the statistics.

Figures 7 and 8 also show that the sampling rate does affect the storage use, with a reduction in sampling leading to a general reduction in the size of the statistics. As one would expect, reduced sampling is not the most effective technique for achieving a reduction in size while keeping accuracy high, however. Instead, the better option is to use the NDV (with or without mode) statistics type, which is less demanding on space while scoring better on accuracy than histogram statistics with high sampling. The same effect can be seen with pruning techniques as well (see figure 14), where more severe pruning is effective at reducing the size of the statistics, but with NDV statistics scoring better on accuracy versus size compared to pruned histogram statistics.

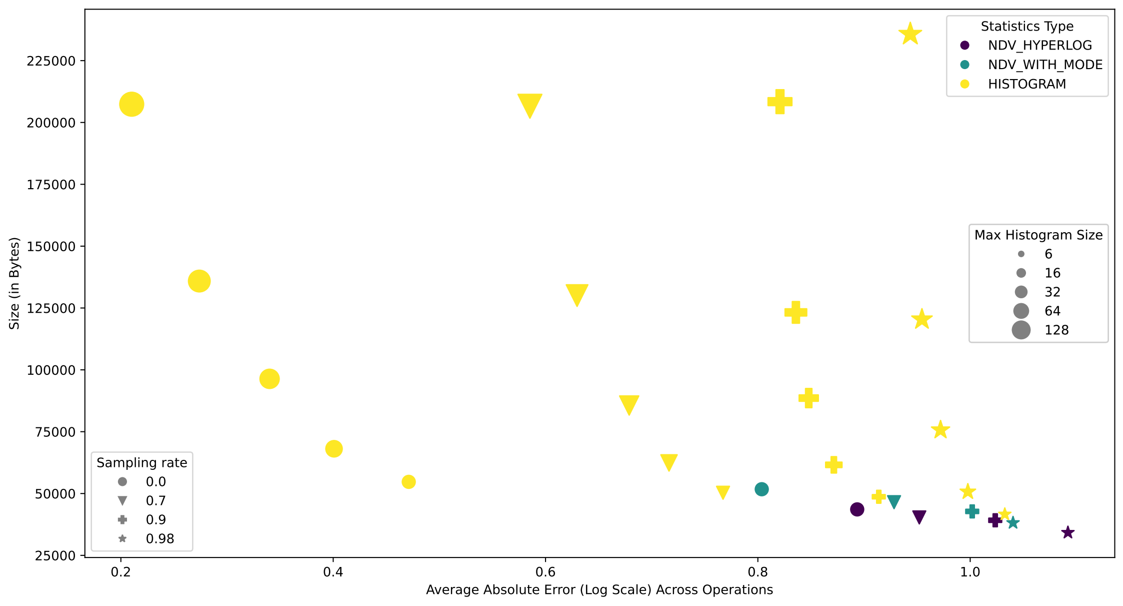
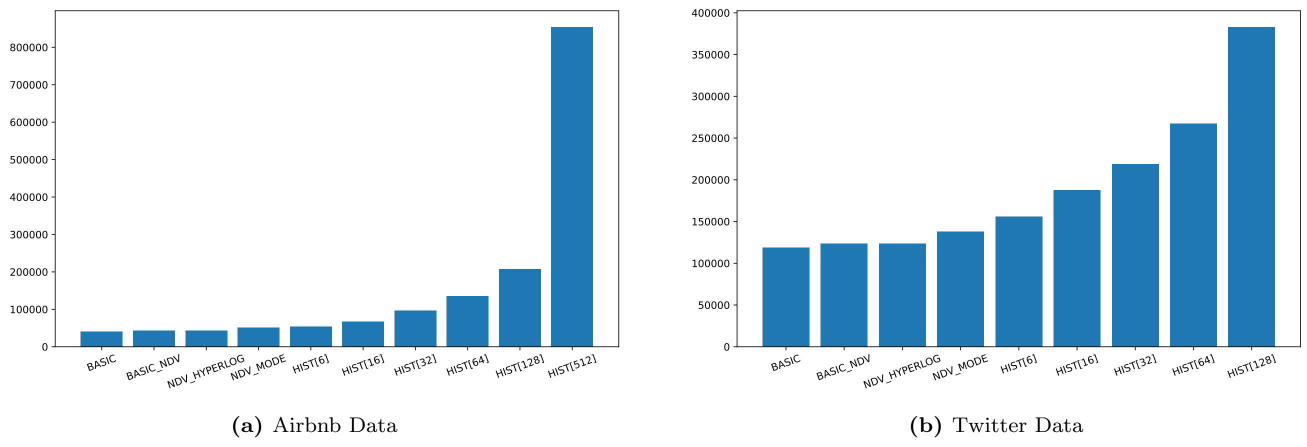


Figure 8: Comparison of the storage size required for the different statistics types using the Airbnb dataset. Histogram with 512 buckets omitted. No Pruning techniques used.



(a) Airbnb Data **(b)** Twitter Data

Figure 9: Comparison of the storage size required for the different statistics types for both datasets. Sizes are in bytes. Note that the y-axes do not use the same scale.

5.2.1 Storage Requirements

As expected, the experimental results show that with respect to estimation accuracy, the NDV statistics outperform the base statistics and the histogram statistics outperform the NDV statistics. This gain in accuracy naturally comes at the cost of increased storage requirements. Figure 9 shows the storage requirements for the two Airbnb and Twitter datasets with no sampling employed.

The reason for the massive size of the histogram statistics with a maximum of 512 buckets computed on Airbnb data is due to there being a lot of key paths whose value sets fit wholely within a singleton histogram with 512 buckets or less. The bulk of this data is made up of values for the "amenities" key, which is an array. Each array counts as a unique key path and is therefore collected statistics for. As there are many different amenities and many listings list a lot of amenities, the histogram statistics up with majority of its contents being singleton histograms listing the various amenities in the different positions they appear in (the amenities are not sorted across documents). Amenities are string values and can be quite long, so this adds even more cost to the storage of these histograms.

This shows a weakness in the current approach to collecting statistics: a lack of conservativeness when it comes to deciding which key paths to include. With arrays containing more elements than can be represented by a single decimal digit, it is probably unlikely that queries will query for specific indexes of those arrays. Therefore, a fix that would likely be simple and effective and cause little degradation in performance would be to not collect statistics for array indexes after the first few, or to simply drop all array index statistics where the array is detected to be over a certain length. In addition, when summary array statistics are collected, the need for data on individual array indexes is also likely to decrease even further.

It should be noted that the structure of the statistics is different for the MySQL and the Python versions. The Python prototype stores the statistics for each as an object (see listings 7, 8, and 9). The MySQL version instead uses arrays, meaning that the key strings for each object can be dropped. This leads to significant savings, as storing all the object keys ends up requiring quite a bit of space. However, the relative difference between measurements are the same for both versions – only the absolute values are lower.

5.2.2 The Impact of Sampling

It is clear from figures 7 and 8 that sampling has a large impact on the accuracy of estimations. The sampling rate does not affect the different statistics types equally. We can see from the same figures that as the sampling rate becomes lower, the gap between the histogram and NDV statistics lessens. This is likely due to the fact that the NDV statistic is less susceptible to resulting in large misestimations compared to the histogram statistics. This effect becomes even more clear when looking at figure 10. Here, we can see that the varying sampling rate has almost no effect on the BASIC statistics type. On the other hand, when the sampling rate is very low, the assumption that the histogram data is correct proves to be so wrong that the larger histogram is outperformed by the smaller histogram statistic. This effect is not present in the Airbnb data, however.

It should be noted that this latter effect is most pronounced for the inequality operators. For the unary operators, all statistics types perform the same.

5.2.3 Error Distributions

Figures 11 and 12 show the distribution of estimation errors for each operator for the different statistics types. While both the NDV and histogram statistics lead to overestimations, the NDV statistic – and especially NDV and mode value statistic – lead to underestimations at a far higher frequency than the histogram statistics do. Interestingly, the accuracy for the equality operator is almost the same for the NDV and histogram statistics up until 64 or 128 histogram buckets are allowed. This is likely due to both relying on the assumption of uniform distribution, which either proves to be relatively true, or equally false within the equi-height buckets as across the whole value range.

Another observation is that the histograms with few buckets perform very poorly on the array operations. This is because the key path that most heavily contributes test values to these operations, the "amenities" key, has a ton of different possible values. As seen with the example test value for this key in listing 14, the test values can be quite large.

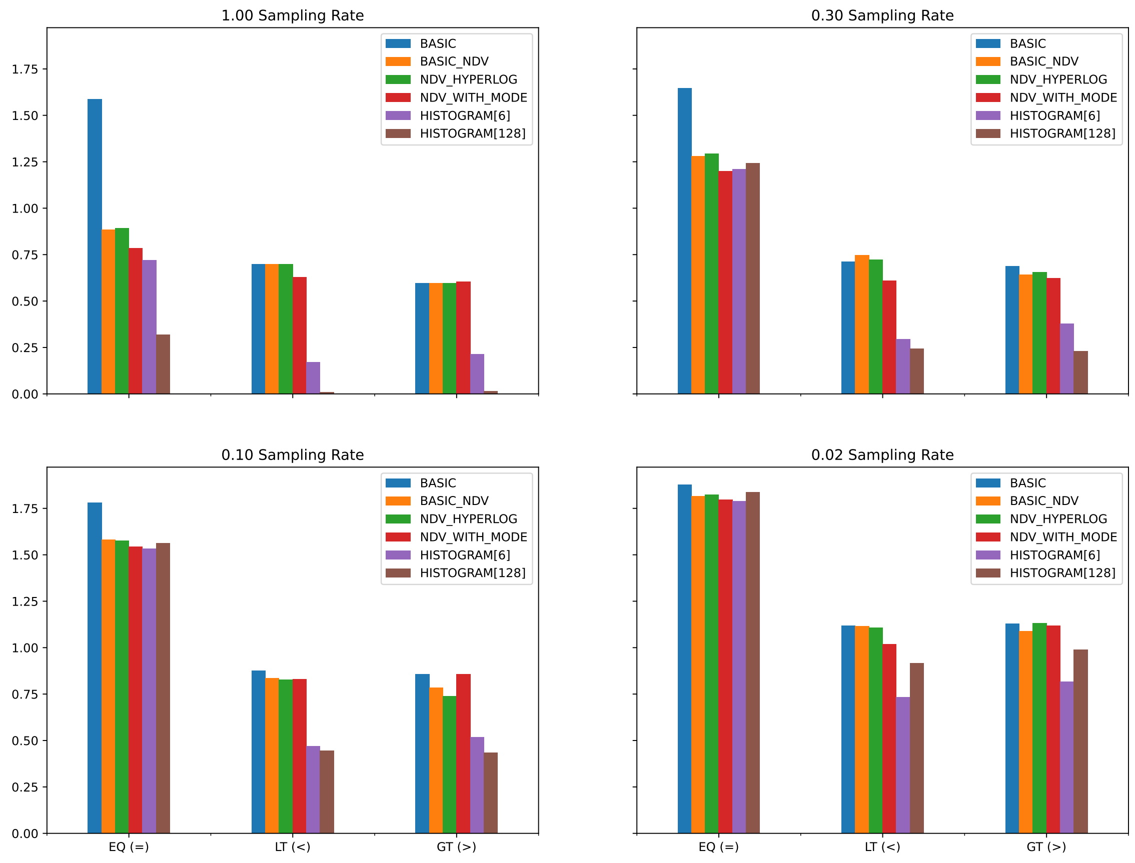


Figure 10: Comparison of the impact of different sampling rates on the estimation errors of the different statistics types for the three binary operators. Twitter dataset.

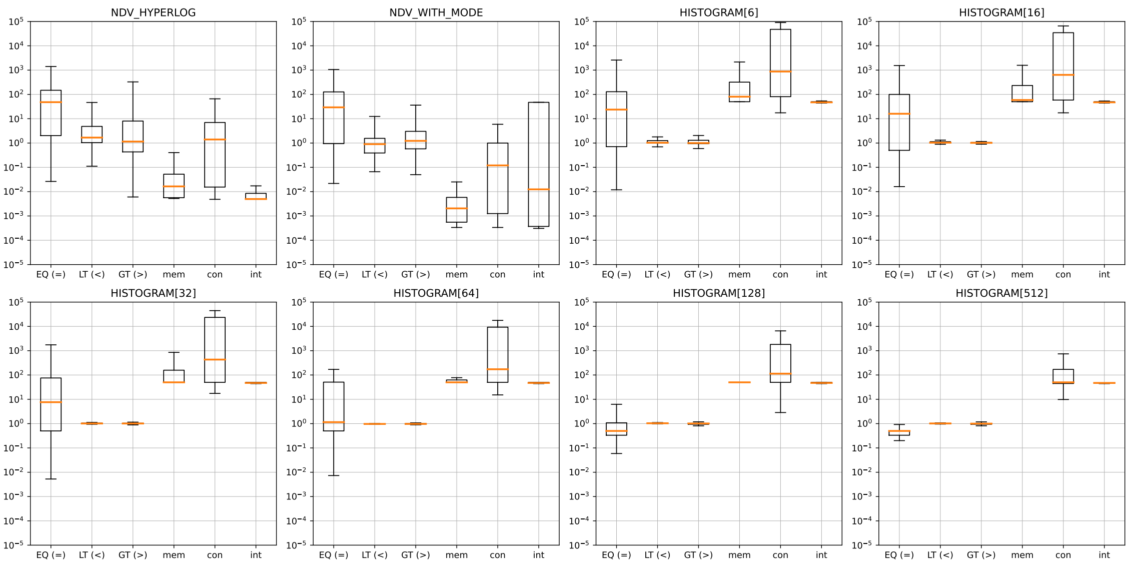


Figure 11: Airbnb. Comparison of the distribution of estimation errors for each operator and statistics type. Sample rate of 0.02.

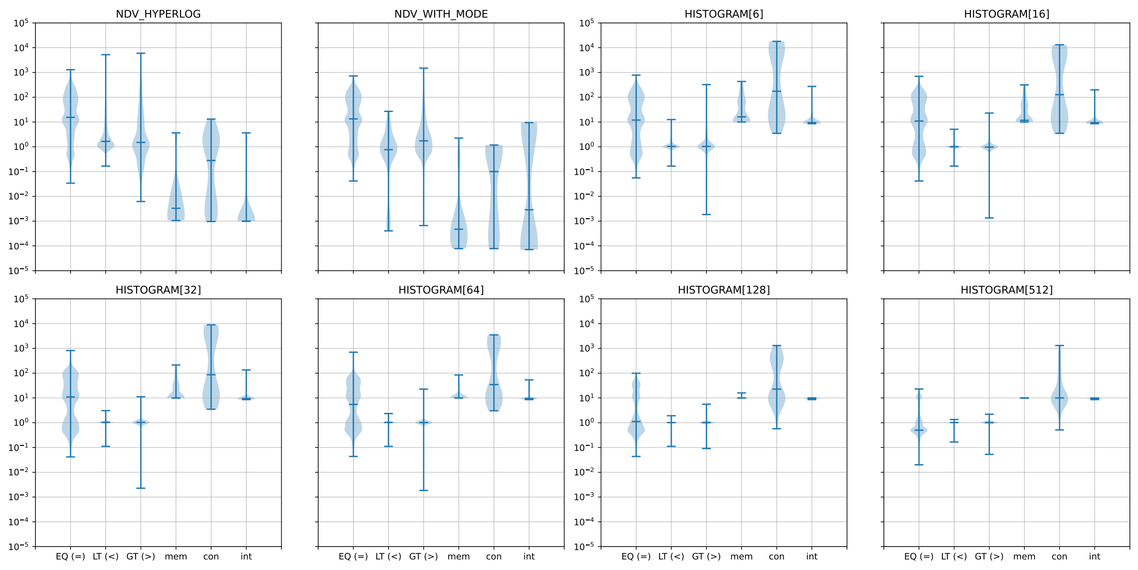


Figure 12: Airbnb. Comparison of the distribution of estimation errors for each operator and statistics type. Sample rate of 0.1.

A good estimate for such a test list value therefore requires accurate estimates for most if not all of the individual values within that test list. This again, requires histograms to allow many buckets.

5.2.4 HyperLogLog Accuracy

The impact of using HyperLogLog on the accuracy of estimations is negligible, as can be seen in figure 13. In fact, when sampling is involved, the small inaccuracies that come with the use of HyperLogLog seem to make the statistics more robust.

5.2.5 Pruning Techniques

Figures 15 and 16 show the impact on accuracy for the two pruning strategies which remove paths: MIN_FREQUENCY and MAX_NO_PATHS. For the Twitter dataset, even the very lenient MIN_FREQUENCY threshold of 0.1% results in a noticeable degradation of accuracy. With a threshold of 1%, this effect becomes even more pronounced, with estimates being off with several orders of magnitude compared to when no pruning was employed.

The structure of the documents of the dataset very much affects the outcome of the pruning step. For certain document structures, the pruning may result in large changes. However, the opposite is also true. If there are few paths, then the MAX_NO_PATHS strategy will have no effect. If there is little heterogeneity, and all/most paths are present in all documents, then the MIN_FREQ pruning strategy will have no effect. Both of these effects were seen in the Airbnb data, which features few paths overall, and most paths are always present.

For the Twitter data however, there are many more paths, and the documents are much more heterogeneous (with respect to paths missing or not). This effect can be seen in figures 16 and 15. Another effect that can be seen in these plots is that as the pruning

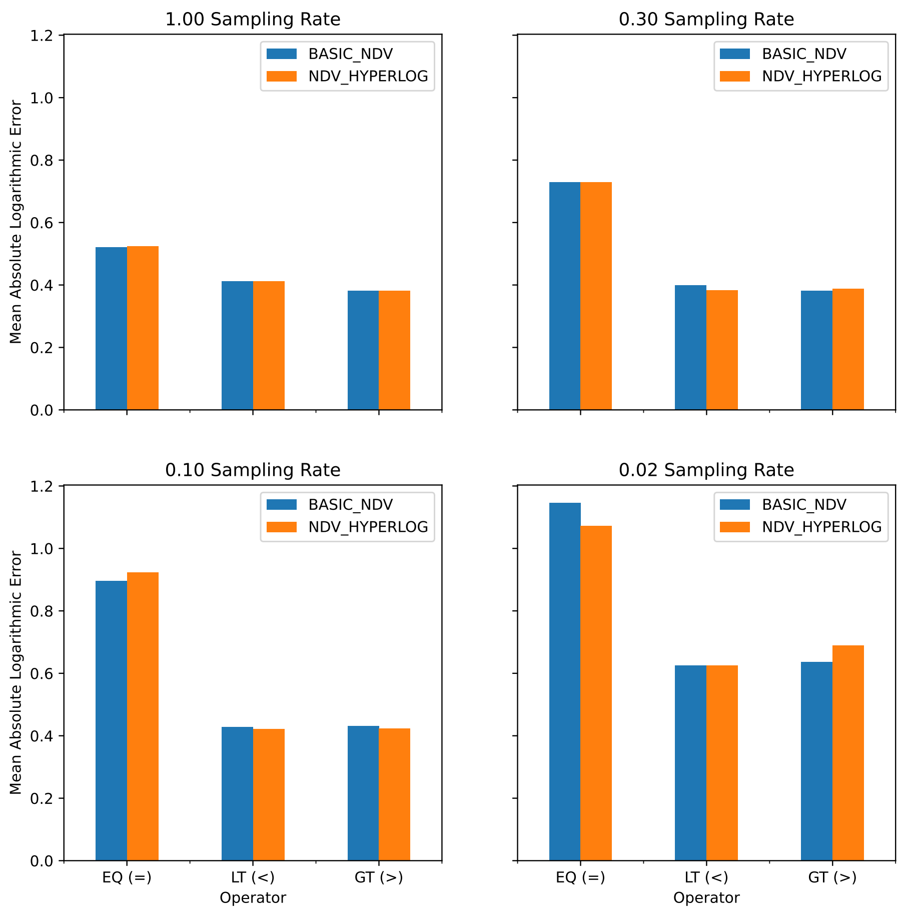


Figure 13: Comparison of the estimation error using the BASIC_NDV and HYPERLOG_NDV statistics gathering settings.

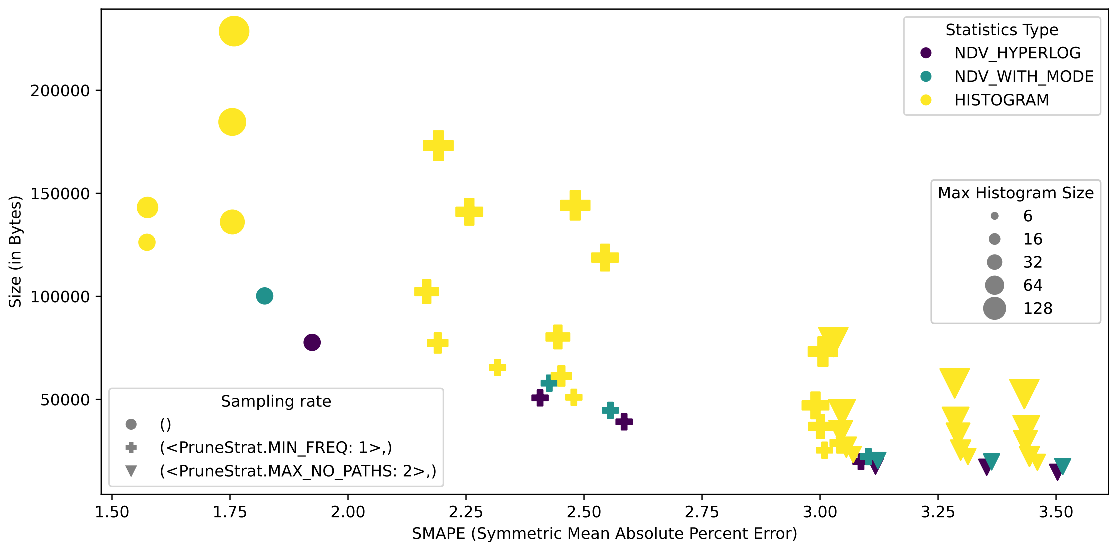


Figure 14: Twitter. The impact of pruning on accuracy and size. The marker type indicates the pruning strategy used, with circle indicating none. Sample rate is 0.02.

becomes more severe, the advantages of more detailed statistics seem to lessen somewhat. The effect can perhaps most clearly be observed by comparing figures 17 and 18

5.2.6 Estimating Array Operations

With the unary operators, every statistics type is equally effective getting accurate estimates. With the scalar binary operators, except for the BASIC type perform very poorly, the statistics gradually and evenly improve in accuracy as more data is added to them. Large histogram statistics perform slightly better than small histogram statistics which perform slightly better than statistics with NDV and mode which again perform slightly better than statistics with only NDV.

With the array operations however, the only statistics type found to perform well were statistics with histograms. As seen in figure 19, the NDV and NDV_WITH_MODE types perform very poorly on these operations, severely underestimating the true cardinality of the test value. This is likely due to the distribution of array values being very uneven, leading to the assumption of uniformity being very wrong. The statistics which include a mode value will subtract the count of the mode value from the total count before dividing by NDV when the test value does not equal the mode value, leading to an even lower estimate. The histogram statistics on the other hand seem to perform quite well with the array operations, and the accuracy also improves quite significantly when more buckets are added.

Figure 20 compares the error distributions for the two datasets with and without array operations. From these error distribution plots, it seems that the array operations depend on histogram statistics to a greater degree than the other operations tested. It also seems that they benefit from an increase in the number of histogram buckets than the other operations.

It should be noted that the Aribnb dataset contains quite a limited number of arrays that could be tested against the array operations. Therefore these results might change significantly when tested on larger and better suited datasets. Sadly this was not possible to achieve for this project due to time constraints.

5.2.7 Overall Costs

When collecting and processing statistics, the difference in memory use and time taken between the different statistics types was found to be small. This is because the size of the document collection itself dominates the size of the statistics, and because the most time consuming part of the statistics creation process is traversing all the documents. Compared to these, the cost of collecting and processing individual collections of values is miniscule. There are therefore little savings to be made in terms of statistics creation costs by choosing simpler statistics types – at least for datasets of the size used in these experiments. It is possible that for more massive datasets, the distribution of costs would be different.

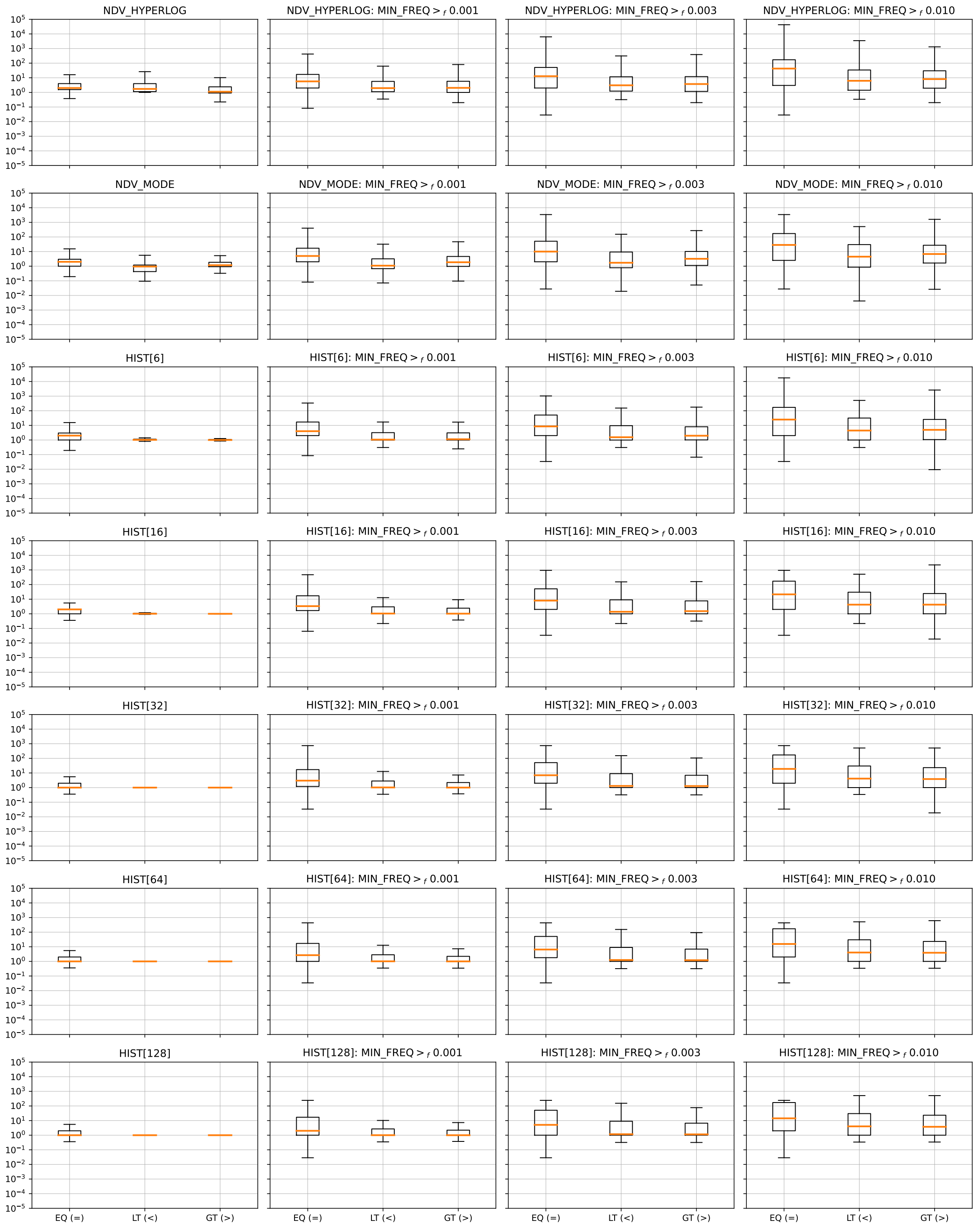


Figure 15: The impact of the MIN_FREQ pruning strategy on Twitter dataset with no sampling. Each row contains a different statistics strategy. The columns are ordered from least severe (no pruning) to most severe (strictest pruning setting). Y axis shows the log error, with smaller (closer to the middle) being better.

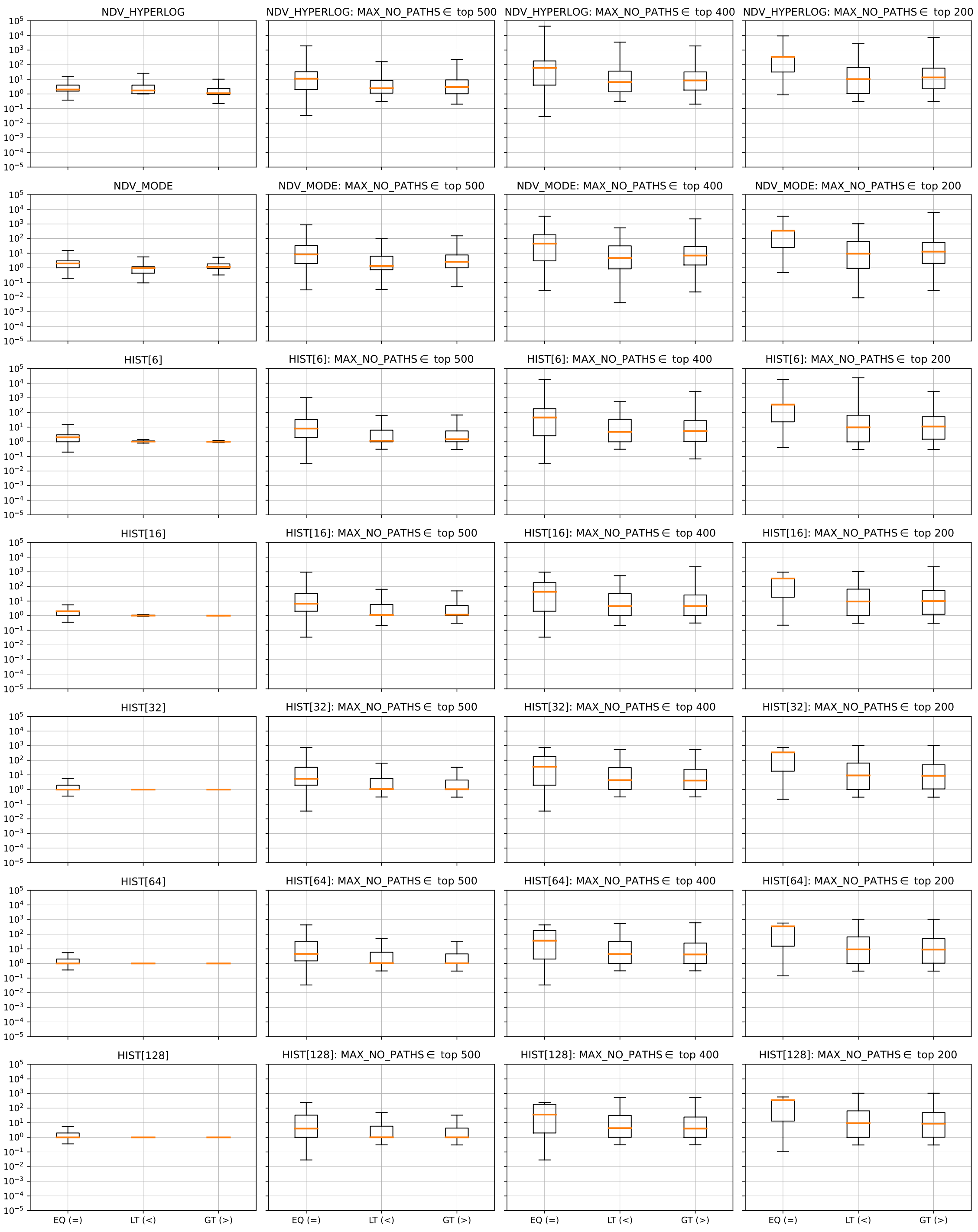


Figure 16: The impact of the MAX_NO_PATHS pruning strategy on Twitter dataset with no sampling. Each row contains a different statistics strategy. The columns are ordered from least severe (no pruning) to most severe (strictest pruning setting). Y axis shows the log error, with smaller (closer to the middle) being better

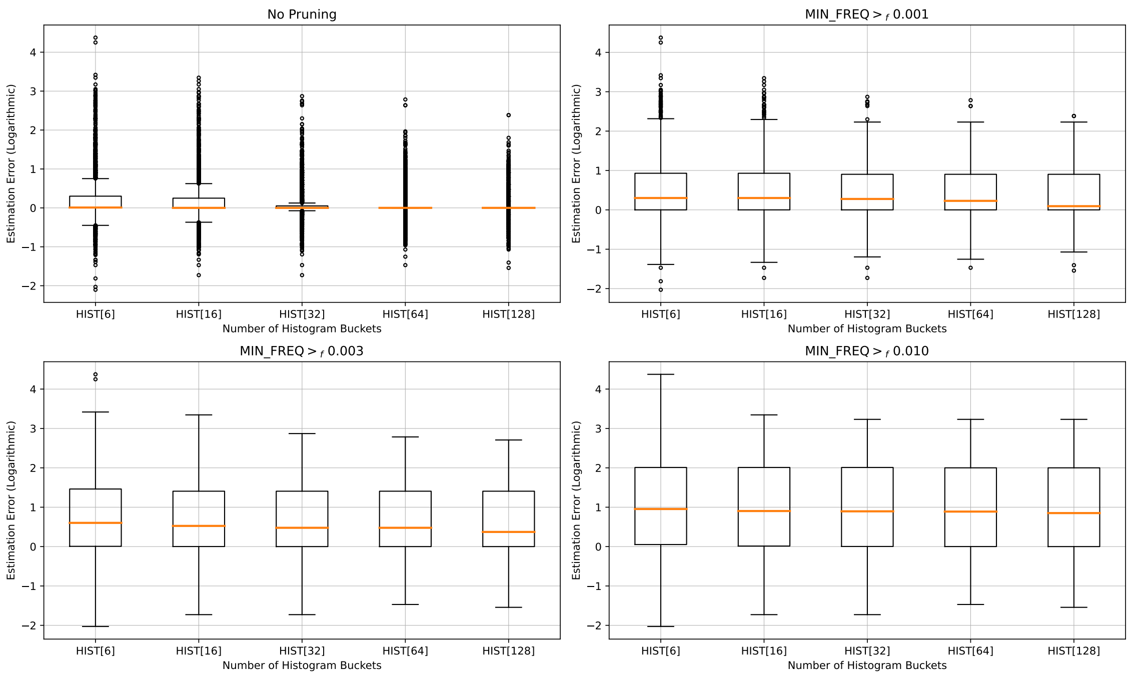


Figure 17: Twitter. The effect that pruning has on the improvement in estimation accuracy yielded by increasing histogram bucket counts. It appears that as the pruning becomes more strict, increasing the bucket count yields smaller gains in accuracy overall.

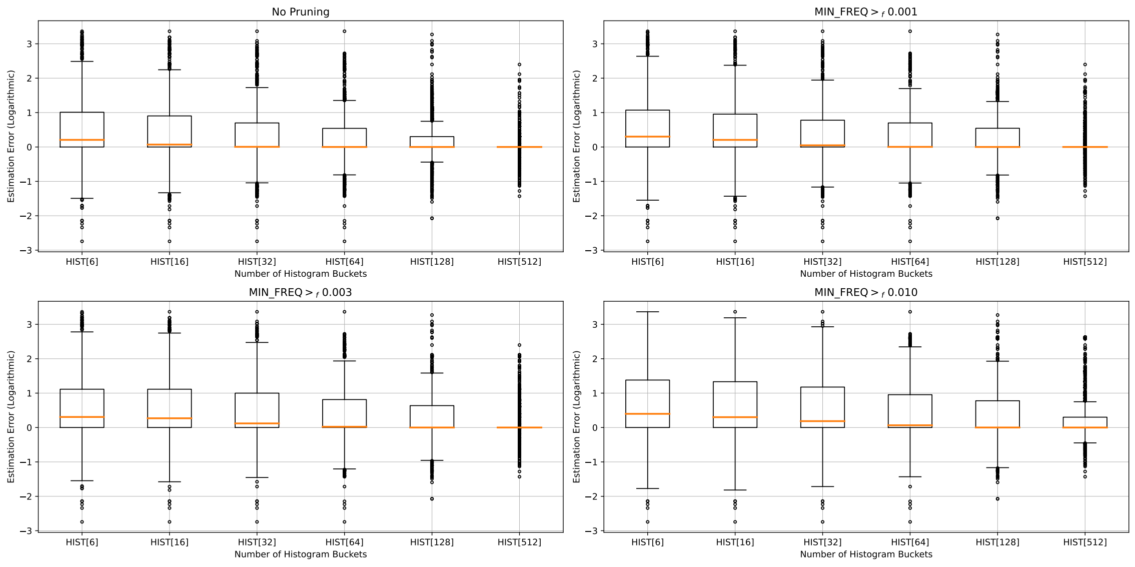


Figure 18: Airbnb. The effect that pruning has on the improvement in estimation accuracy yielded by increasing histogram bucket counts.

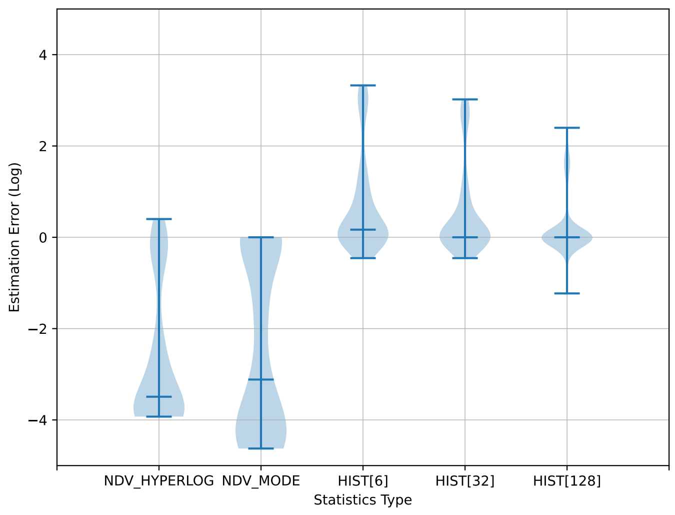
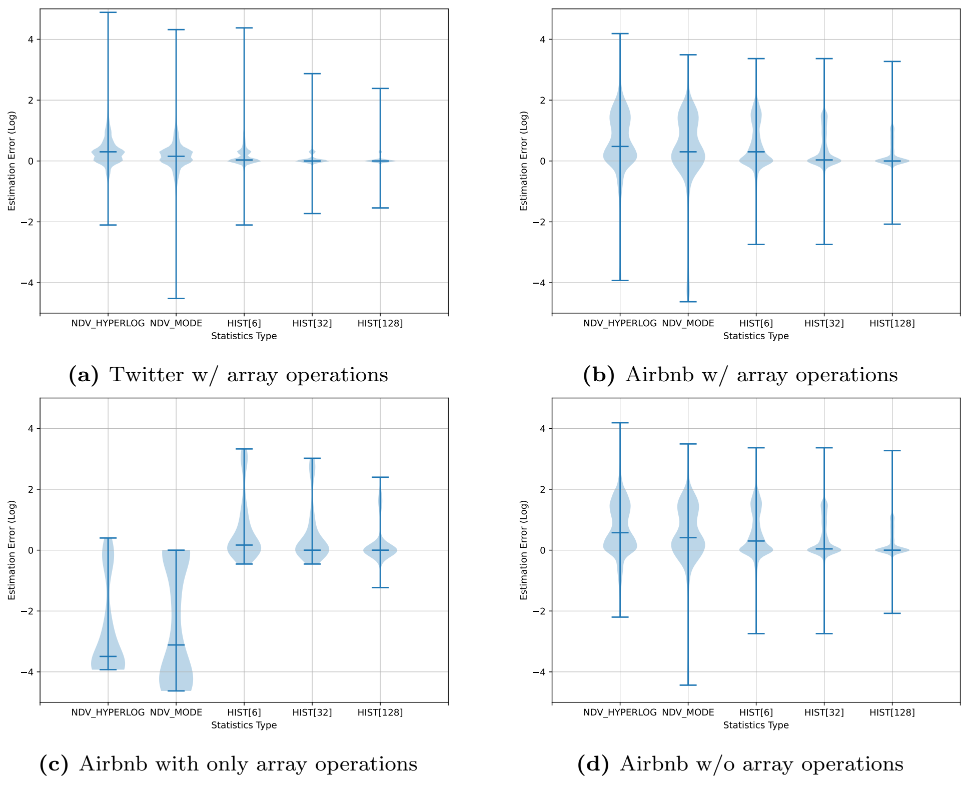


Figure 19: Airbnb. Overall accuracy distribution for five different statistics configurations. Full sample size and no pruning.



(a) Twitter w/ array operations

(b) Airbnb w/ array operations

(c) Airbnb with only array operations

(d) Airbnb w/o array operations

Figure 20: Comparison of the impact of Array operators on error distributions. Sampling rate 0.

5.3 Discussion

The general trend seen in the experiments of the Python prototype is that the more data is included in the statistics, the better the estimates become – although the returns are diminishing. When sample sizes are very low, the utility of the more advanced and detailed statistics types is reduced as they become more susceptible to becoming biased and blindsided (see for example figure 10 where the larger histogram performs worse than the smaller one at a sample rate of 0.02). The accuracy of the less detailed statistics types suffers less when the sample sizes are low, closing the gap between the more and less detailed statistics types.

This effect of diminishing returns on more detailed statistics can also be seen when pruning techniques are used. See for example figure 17, where there is practically no gain in accuracy when increasing the maximum number of buckets allowed for the histogram statistics when the pruning is severe.

The severity of this effect of diminishing returns is heavily dependent on the dataset however! For the Twitter data set, which is highly heterogeneous and contains many key paths, many of which appear infrequently, the impact of missing data can be very high. In the case of pruning, this happens because the paths simply disappear from the statistics due to being being below the set threshold limit. When sampling, paths that already appear infrequently become encountered even more rarely and may even never be encountered at all when the statistics are built. Both cases clearly lead to statistics that do not contain information that is sufficient to

Comparing figures 17 and 18, and 5 and 6, it is clear that the Airbnb dataset is much less severely impacted both by sampling and by pruning. The smaller impact from pruning makes sense – the dataset contains fewer distinct key paths so the MAX_NO_PATHS removes less data. The dataset is also more homogenous than the Twitter dataset and as such is less impacted by the MIN_PATH_FREQUENCY pruning strategy as well. The smaller impact of sampling can be explained by the homogeneity of the airbnb dataset. The chances of missing a key path are much lower, and the distribution of values within for each key path can be approximated in many fewer samples because each key path is almost always present in each document sampled. This can also be seen in figures 7 and 8, where histogram with sampling scores better for than NDV statistics with no sampling in the case of the Airbnb data. This is not the case for the Twitter data, where the NDV statistics are both smaller in size and provide better accuracy than histogram statistics with sampling[12].

Therefore, the choice of whether histogram or NDV (with or without mode) statistics depends also on the dataset, with NDV statistics being better at accommodating more diverse and heterogeneous datasets, and histogram statistics being a better fit for more homogeneous and less diverse datasets – provided that the statistics size should be limited.

6 Experiments With MySQL Implementation

This section presents the experiments ran on the MySQL prototype. The goal of the performing these experiments on the MySQL prototype was to discover the impact that JSON

[12]The Airbnb histogram data is also positively influenced by the fact that the array operators respond more positively to histogram data than the other operators. However, the number of array operator tests is so small that this data has very little impact on the figure in questions (see figure 20).

statistics have on the estimates made by MySQL, and the impacts that these changed estimates have on the MySQL's performance when executing queries. To achieve this goal, a number of data sets were loaded into MySQL and then queries associated with those data sets were executed and the estimates and query plans produced by MySQL for these queries were measured and analyzed.

While section 5 discussed in great detail the different configurations and parameters possible for the statistics, the measurements of their impact were not made in a realistics environment. The Python prototype experiments only reveal differences between different configurations in that testing environment. While this data is undoubtedly interesting and useful, it can only tell us about the idealized accuracy enabled by the statistics structures. It tells us little about their real-world performance.

The section is structured as follows: first, the hardware and software setup used for the experiments will be examined. Then, a description of the data sets and queries used for the experiments will follow. Finally, a discussion of the experimental results will follow. The presentation of the experiment results are divided into two parts: the first part examines the impact of statistics on the accuracy of estimations. The second part examines whether these changes in estimations have any impact on the performance of the queries.

6.1 Experimental setup

For each data set, a total of eigth different configurations of MySQL was tested. Four "levels" of differing statistics configurations were tested:

1. The base MySQL 8.0.32 release.
2. MySQL 8.0.32 with the modifications described in section 4.3, but with no statistics inserted.
3. Same as level 2, but with statistics gathered with using the BASIC_NDV statistics type.
4. Same as level 2, but with statistics gathered with using the HISTOGRAM statistics type.

Each statistics type was tested both with the hypergraph optimizer enabled and disabled (meaning the current MySQL optimizer was used). The reason for this is that both the impact on the current version on MySQL and future versions is of interest. The hypergraph optimizer will be taken into use in the future, and so ignoring it will yield results that will soon no longer be applicable to current releases of MySQL. In addition, because the hypergraph optimizer is able to consider a larger set of query plans, and will also use the JSON statistics for join size estimations, the statistics are likely (and do in fact, as will be shown later in the section) to have a larger impact when this optimizer is used. When the hypergraph optimizer is said to be "off" or disabled, this means that that MySQL's current optimizer is being used. The reason for testing both the base release version of MySQL as well as the modified version without any statistics is to isolate the effect that simply allowing heuristics to be used makes on estimation accuracy and query performance. Again, the current release of MySQL (8.0.32) simply assigns a selectivty factor of 1.0, i.e., no reduction in cardinality, to all JSON functions.

The statistics used in the experiments were gathered withoutout any sampling, and no pruning techniques were used. The histogram statistics allowed at most 64 buckets. Due to time constraints, running the MySQL experiments with any more configurations was not feasible. However, I believe that the chosen configurations are be sufficient to show the impact that the statistics have. For a detailed discussion of how the sampling and pruning affect the accuracy of estimations, see section 5. No indexes were created for the data.

For the timing experiments, timing data was gathered using MySQL's built-in performance schema. As no modifications were made to the MySQL client for this project, the base MySQL 8.0.32 client was used for all experiments. To prevent the amount of table data currently in memory from affecting the judgements of the query optimizer, the cost model variables `io_block_read_cost` and `memory_block_read_cost` were updated to have the same value.

The machine used for the MySQL experiments is not the same as the one used for the Python experiments. The specs of the machine used for the MySQL prototype experiments were as follows:

- Ubuntu 22.04.2 LTS
- 32 GiB RAM
- 11th Gen Intel® Core™ i7-1185G7 @ 3.00GHz × 8
- 512 GB SSD[13]

The queries measuring query execution timing were executed multiple times and their results averaged to ensure that even and representative measurements were used.

6.2 Data and Queries

Few benchmarks or real-world datasets with complex queries against RDBMS JSON data were found when researching suitable experiments to perform for the MySQL prototype. This is problematic because relational data is structured and queried differently from document data like JSON. It is especially problematic for operations that are unique to JSON such as the JSON array operations because these are not found in existing benchmarking suites for relational data – meaning that improvements in estimates for these operations will not be measured using existing suites without heavy modifications.

Instead of existing JSON data benchmarks, a combination of existing relational benchmarks and custom data sets and queries created specifically for use in these experiments was used. The goal of the inclusion of the relational benchmark is to test the statistics on established data and queries, while the custom data sets and queries were used to test the statistics on (likely) more realistic data and queries. The next few sections will detail the data sets and queries used in the experiments.

Due to time and technical constraints, not all queries could be executed for all configurations. Most simply took longer than the set timeout limit, while some caused the machine running the experiment to run out of disk space by storing massive intermediary data sets. The queries for which this happened are marked in the figures. The timeout limit used varied between the datasets, and will be noted for each. Unless otherwise stated, the timeout limit used was 10 minutes.

[13]with about 256 GB of available space when running the experiments

The queries created for these experiments are all relatively complex. This is because the main things that statistcs impact, and which is visible when measuring performance, is join ordering. Observing differences in performance between different configurations for simple queries where no joins are performed is highly unlikely, and so such simple queries are uninteresting results-wise. It is important to keep in mind however, when seeing that a majority of queries are impacted somehow by statistics, that the queries chosen for the experiments do not necessarily represent a realistic sampling of real-world queries.

6.2.1 TPC-H

The TPC-H data is relational when generated, and so it had to be converted to JSON for use with the experiments. The JSON version of the data was a direct translation of the TPC-H data to JSON format, with the tables retained, but with all the data for each table being placed into a single column containing JSON data. Each row was translated into an object with the table's column names used as keys and the row values being used as values. Several columns in the TPC-H data are of the decimal data type, which is not directly supported by JSON. The values in these columns were converted to integers and multiplied by 100 (as they had two values after the fractional point). The decimal columns were defined to support up to 15 digits in total, giving the numeric values of these columns a range which can comfortably be supported by 64-bit integers (log[2]$10^1$5 $<$ 50). It is possible that the very extremes of this range cannot be represented by doubles, which can accurately represent integers up to $2^5$3 [4] ((log[2]$10^1$6 $=$ 53.15), accounting for signed numbers), the numbers were serialized as whole numbers which means that MySQL will interpret them as 64-bit integers. Several of the columns also encode date values, which JSON does not support directly. Instead of turning the date values into date strings, they were converted to UNIX timestamps and stored as integers. This was done because the current prototype does not support detailed statistics for date strings (see section 4.3.4 for discussion), but adding support for them is seen as only a minor technical challenge. Because time constraints did not allow for the implementation of date string support, the use of UNIX timestamps with the TPC-H data was used as a simpler, temporary workaround. In theory, date strings and UNIX time stamps would be treated exactly the same by the estimation system.

The TPC-H queries, being written (generated) to make use of the standard relational tables and column types also had to be rewritten. This was done programmatically, with a minor errors in the translation corrected by hand afterwards. For reasons related to various technical challenges and time constraints, the TPC-H queries X, Y, Z were excluded from the experiments.

The data generated by TPC-H is not ideal for testing the impact of statistics. TPC-H famously generates statistically independent data [14], so the TPC-H queries will benefit much from the addition of accurate selectivity estimates without experiencing one of the major downsides when the assumption of independence turns out to be incorrect. In addition, the distribution of the data itself within each column is often practically uniform. This leads to lessening in the impact of histogram statistics compared to NDV statistics, because the assumption of a value's frequency being 1/NDV will always hold true, something which is usually not the case in real data sets. Finally, the directly translated TPC-H data makes for perhaps unrealistically structured JSON data. There are no null values, no nesting, no missing keys, no "type confusion", and all key paths have a frequency of 1. However, the TPC-H dataset is widely known and an accepted part of database performance benchmarking, and the TPC-H queries explore many different approaches to data retrieval and query structures. And importantly, the TPC-H queries are independent of this work and created by an outside party. I therefore believe that they will offer good and valid insight into the effect that the JSON statistics have. And while limitations of the generated data must be kept in mind, the other datasets used for these experiments do not suffer from the same issues.

The data was generated using the following arguments to the data generation application: `-f -s 0.05`.

Note: Some tpch queries use the BETWEEN operator, whose implementation was slightly wrong for the experiments. It assumed that the base frequency of the key path was always 1, which of course is unlikely to always be the case and in some cases may be completely wrong. However, as just discussed, this assumption is true for the TPC-H data. And as the TPC-H datasat was the only dataset for which the BETWEEN operator was used, the bug had no real impact on the experiments.

For a detailed description of the structure and semantics of the TPC-H data, see the TPC-H manual.

6.2.2 Airbnb

The Airbnb dataset used in the testing of the MySQL prototype is a superset of the Airbnb dataset used when testing the Python prototype. In addition to the listings data, three additional document collections were used: calendar, reviews, neighbourhoods. All four document collections stem from the Los Angeles dataset published March 7th. 2023. A table representation of the datasets is presented in table 2, showing the most important key paths for each of the collections. All keys shown in the tables have a frequency of 1.

The calendar data keeps the availabilities and prices for each listing for every day of the coming year. The Neighbourhoods data groups neighbourhoods into greater areas. For example, both neighbourhoods of Arleta and Bel-Air are in City of Los Angeles. Val Verde on the other hand, is located in Unincorporated Areas. The reviews data contains the user reviews for each listing. The actual ratings of the listings however, are stored in the listings data set.

The sizes of the different tables can be seen in table 3.

The queries of the Airbnb dataset feature multiple joins, and filters. Typically, the calendar and listing tables are joined to find listings available for a certain date and filtered on the properties of the listings like rating, amenities and location. The Airbnb queries can be seen in their entirity in appendix C.

6.2.3 Twitter

The Twitter dataset consists of the exact same data as the Twittier data used in the Python experiments. It is all inserted as a single table with a single column, with each row of that column containing the data for one tweet. The Twitter data is quite heterogeneous structurally, with more than 80 of its 809 paths occurring less than 1000 times, meaning that they have a frequency of less than 0.6%.

The dataset contains tweets only. However, the various metadata available with the tweets is quite extensive and includes geographic information, language information, some user

Table 2: The Airbnb data fields used in the experiment queries.

(a) Calendar

Key Name	Usual Type	Example Value
listing_id	string	”10000255”
date	string	”2023-03-07”
available	string	”t”
minimum_nights	integer	1

(b) Neighbourhoods

Key Name	Usual Type	Example Value
neighbourhood_group	string	”City of Los Angeles”
neighbourhood	string	”Acton”

(c) Reviews

Key Name	Usual Type	Example Value
listing_id	string	”10000255”
date	string	”2023-03-07”

(d) Listings

Key Name	Usual Type	Example Value
id	string	”10000255”
review_scores_rating	float	5.0
bedrooms	integer	2
host_is_superhost	string	”f”

Table	Row Count	Size
listings	170285	190 MB
reviews	1398664	538 MB
neighbourhoods	270	23 KB
calendar	15492121	2.7 GB

Table 3: File sizes and row counts for the different document collections in the Airbnb dataset.

profile information, data on the hashtags used, and more.

The queries for the Twitter data in general feature fewer joins than the Airbnb and TPC-H data, and uses some operators that were not used for the Airbnb data such as the not equals operator `<>` and the `IS NULL` operator with the `JSON_VALUE` function. Operators such as IS NULL depend on very simple statistics, which are gathered for all the statistics types supported by the Python prototype. As such, it is also likely here that the differences in the estimates using the NDV and histogram statistics may be small. As with the Airbnb queries, all the queries of the Twitter data can be found in appendix C.

6.3 Estimation Experiments

Accuracy of estimates can be hard to compare across different configurations because the query plans may differ significantly in their structure. While the filtering predicates are the same for all configurations, their place and context within the query plan may differ. This is especially true when using the hypergraph optimizer, which was found to produce a much more diverse set of plans than the current query optimizer. Because errors propagate and (usually) grow larger the further up the execution tree you go, the input to the predicates may differ greatly between configurations. Therefore, naively comparing the estimates for each filter or join is not always possible. Additionally, MySQL's estimate of the row count of a table can be off by quite a few percent (the author has observed misestimations of up to 30%). This error will of course propagate, affecting all subsequent estimations.

To minimize the influence of some of these factors on the estimation accuracy calculations and comparisons, the scheme used to calculate estimations does not simply compare the estimated row count and the actual row for each filter or join in the query plan. Instead, the selectivity estimate is calculated by comparing the estimated row count by the estimated row count of the child iterator feeding the current step. The real selectivity is calculated by comparing the actual row counts of the two iterators. Then, two selectivity estimates are compared to get the estimation error used in the visualizations in this chapter.

Another challenge is that in cases where queries time out or run out of memory or disk space, we can only see the estimated row counts because the real ones could not be computed. This is important to keep in mind when comparing error distributions because when query plans time out, a lack of an actual row count to go with the estimated row counts makes calculating the estimation error very challenging. These errors cannot then be added to the distribution – and so the worst mis-estimates don't show up. Due to these challenges, this section relies somewhat on comparing the query plans and associated esimates directly instead of by using visualizations and graphs.

6.3.1 TPC-H

For the TPC-H data and queries, as with the other data sets and associated queries (see the next few subsections), massive improvements in the quality of selectivity estimations were observed when MySQL prototype compared to when using the current release version (the "base" version) of MySQL. A summary of these improvements can be seen in figure 21.

Because base MySQL assigns selectivity factor of 1.0 to all predicates, it is incapable of underestimations. This results in the massive overestimations seen in figure 21. The ver-

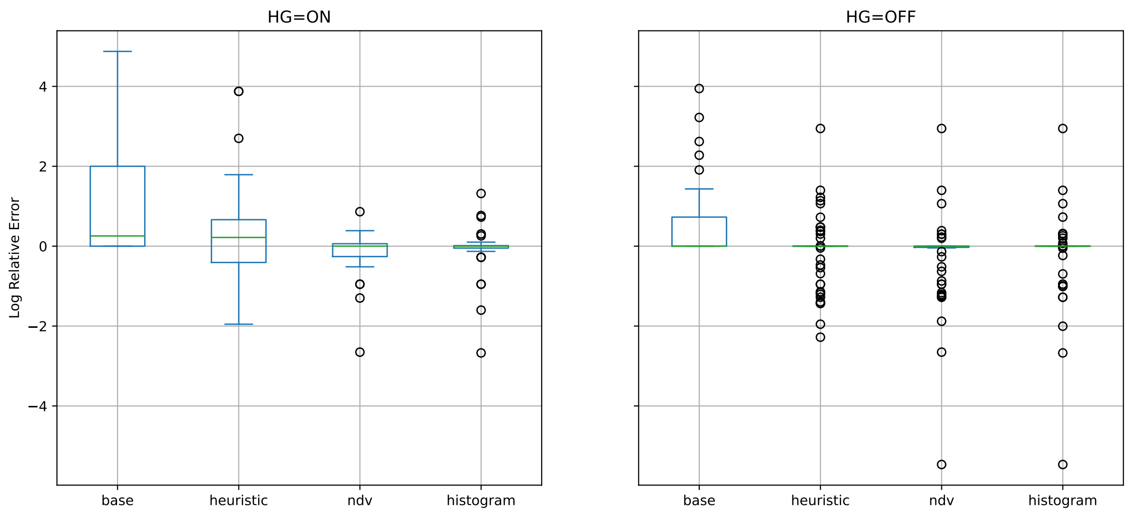


Figure 21: TPC-H. Comparison of the logarithmic relative estimation errors measured at all steps in the queries.

sion using heuristics and no statistics will sometimes make underestimations, but because the strictest selectivity factor it can produce is 0.1, its estimates also tend towards large overestimations. With large queries with many joins, these overestimations propagate and multiply causing a ballooning of estimates. In the case of the base version of MySQL, the most extreme case resulted in an overestimation of 13 orders of magnitude.

The figure also shows that estimation errors in general are larger when MySQL's current optimizer is used. This happens because MySQL's current optimizer is unable to make use MySQL's histogram statistics to make selectivity estimates for join conditions, so the result sizes of joins are often to estimated to be much larger than they turn out to really be. This ends up causing much larger estimation errors for complex queries with many joins than what is usually seen with the hypergraph optimizer. A filter may be applied after the join to alleviate this issue, but this is not a perfect solution, and can cause estimates to become too small. While this effect is not as pronounced in the TPC-H data, it is very clear in the Airbnb and Twitter results (see figures 22 and 23).

As expected, the even distribution of values within columns caused the measured differences in error for NDV statistics and histogram statistics to be miniscule.

6.3.2 Airbnb

For the Airbnb queries, one can clearly see an improvement an estimation accuracy when moving from the least amount of tools and informations (base MySQL) to the most (prototype with histogram data). This effect can be observed both using the current and with the hypergraph optimizer. Figure 22 shows this progression well. The estimates are clearly not perfect, with both the NDV and histogram statistics causing relative overestimations in the order one million in the worst case. However, they are a massive improvement over the estimations offered by the base version and the version using only heuristics.

The impact of the improved estimates can be clearly seen in the choice of query plans (and thus the execution times, as will be discussed in the performance section) when the hypergraph optimizer is enabled, with the majority of queries getting different and improved plans when NDV and histogram statistics are introduced. Listings 17, 18 and

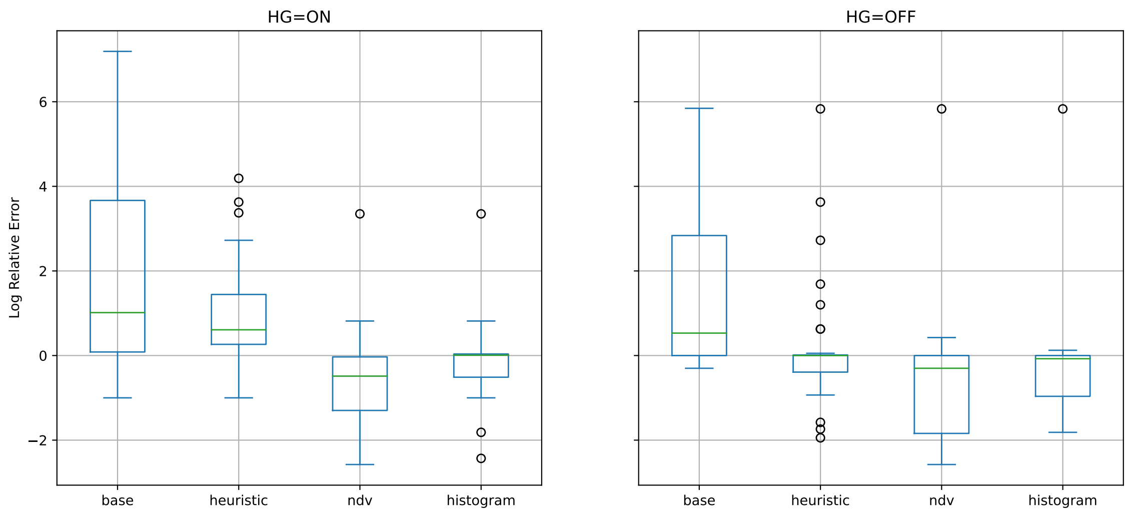


Figure 22: Airbnb. Comparison of the logarithmic relative estimation errors measured at all steps in the queries.

19 make up an example of how the query plans change with the introduction of statistcs, and how how accurate the estimates used to construct these query plans are.

```sql
SELECT listings.j
FROM listings, calendar
WHERE
    -- Join conditions
    listings.j->"$.host_id" = "144214204" -- Host with the most listings in LA, ca. 1000 in total
    AND listings.j->"$.id" = calendar.j->"$.listing_id"
    -- Filters on availability
    AND calendar.j->"$.available" = "t"
    AND calendar.j->"$.date" = "2023-12-12"
    AND calendar.j->"$.minimum_nights" = 1
ORDER BY
    listings.j->"$.last_review"
;
```

Listing 16: The `host_availabilities_d` query. One of the eight queries for the Airbnb data sets. It retrieves the listings of a certain host which are available for booking one night only at a certain date.

As expected, the estimates made by the Base version of MySQL (listing 17) are way off. The filters are not applied and a cartesian join is assumed, resulting in the estimation of a humongous number of rows (463 308 494 745) being returned. The misestimation of the sizes of the two join relations causes the optimizer to believe that a sort followed by a sort-preserving nested loop join is the cheapest alternative. This is likely due to the optimizer perceiving the post-join number of rows to so large that performing the sorting after the join would be incredibly expensive. In truth, the nested loop join is very slow compared to the hash join due to the real sizes of the relations being joined, and the join result itself, being very small (see figure 18).

The query plans chosen when using NDV and histogram statistics are the same. However, the actual estimates made using the two statistics approaches are quite different. For the listings filter, the prototype has to rely on the assumption of uniformity when using NDV statistics. The average host has two listings, so that is what forms the basis of the estimate. However, the host id used in this specific query is that of the host with the most listings (1003) in the data set. Thus, the NDV statistics lead to massive underestimate in

```
-> Nested loop inner join  (rows=463308494745)
  -> Sort: json_extract(listings.j,'$.last_review')  (rows=31893)
    -> Filter: (json_extract(listings.j,'$.host_id') = '144214204')  (rows=31893)
      -> Table scan on listings  (rows=31893)
  -> Filter: (((json_extract(calendar.j,'$.available') = 't') and
  ↪  (json_extract(calendar.j,'$.date') = '2023-12-12') and
  ↪  (json_extract(calendar.j,'$.minimum_nights') = 1)) and (json_extract(listings.j,'$.id') =
  ↪  json_extract(calendar.j,'$.listing_id')))  (rows=14526965)
    -> Table scan on calendar  (rows=14526965)
```

Listing 17: The EXPLAIN output for the `host_availabilities_d` query for the base MySQL configuration with the hypergraph optimizer enabled. Because the query timed out, only the estimates are included with the query plan. The true row count for some of the steps can be seen in listing 18 which includes EXPLAIN ANALYZE output.

```
-> Sort: json_extract(listings.j,'$.last_review')  (rows=0.006) (actual rows=0)
  -> Inner hash join
  ↪  (<hash>(json_extract(listings.j,'$.id'))=<hash>(json_extract(calendar.j,'$.listing_id')))
  ↪  (rows=0.006) (actual rows=0)
    -> Filter: ((json_extract(calendar.j,'$.date') = '2023-12-12') and
    ↪  (json_extract(calendar.j,'$.minimum_nights') = 1) and
    ↪  (json_extract(calendar.j,'$.available') = 't'))  (rows=178) (actual rows=3813)
      -> Table scan on calendar  (rows=14360514) (actual rows=15492121)
    -> Hash
      -> Filter: (json_extract(listings.j,'$.host_id') = '144214204')  (rows=2) (actual rows=1003)
        -> Table scan on listings  (rows=31893) (actual rows=42451)
```

Listing 18: The EXPLAIN ANALYZE output for the `host_availabilities_d` query for the MySQL using NDV statistics and with the hypergraph optimizer enabled.

this case. The NDV statistics also lead to the size of the filtered calendar relation being underestimated. While the calendar's date values conform exactly to the expectation of uniform distribution, and the availability values are quite close to uniformly distributed, the minimum nights predicate has a true selectivity of 17%. This is quite far from the selectivity of $\frac{1}{110} \simeq 0.9\%$ that is given to the query optimizer. Because none of the listing ids found through the calendar predicates match the listings owned by the host however, the join of the two relations ends up having zero actual rows, which is almost exactly what is predicted as the outcome of the join when using NDV statistics. This is coincidental however, and not the result of clever use of statistics.

```
-> Sort: json_extract(listings.j,'$.last_review')  (rows=68) (actual rows=0)
  -> Inner hash join
  ↪  (<hash>(json_extract(listings.j,'$.id'))=<hash>(json_extract(calendar.j,'$.listing_id')))
  ↪  (rows=68) (actual rows=0)
    -> Filter: ((json_extract(calendar.j,'$.date') = '2023-12-12') and
    ↪  (json_extract(calendar.j,'$.minimum_nights') = 1) and
    ↪  (json_extract(calendar.j,'$.available') = 't'))  (rows=3841) (actual rows=3813)
      -> Table scan on calendar  (rows=14360514) (actual rows=15492121)
    -> Hash
      -> Filter: (json_extract(listings.j,'$.host_id') = '144214204')  (rows=754) (actual
      ↪  rows=1003)
        -> Table scan on listings  (rows=31893) (actual rows=42451)
```

Listing 19: The EXPLAIN ANALYZE output for the `host_availabilities_d` query for the MySQL with histogram statistics and the hypergraph optimizer enabled.

The estimates produced using the histogram statistics are very accurate, with most of the error in the estimations coming from the fact that MySQL misjudges the number of rows in the base tables. The reason that the estimate for the host id predicate does so well is that a ”singleton plus”/top frequency histogram was created for the key due to its uneven distribution of values. The host id used in this query being by far the most frequent of the host values means that it is present in this histogram and can be accurately estimated for. These accurate estimates do not lead to any changes in the choice of query plan, however.

Using the current optimizer, while estimates are improved, no major changes in the choice of query plan was found to have been made as a result.

6.3.3 Twitter

Perhaps somewhat unexpectedly, analysis of the estimation errors for the Twitter queries did not show any significant difference in the estimations made using NDV and histogram statistics. With the hypergraph optimizer enabled, the mean estimation error using histogram statistics can be seen to be every so slightly closer to 0 (which is good), and the worst recorded overestimation is one order of magnitude lower than for the NDV statistics.

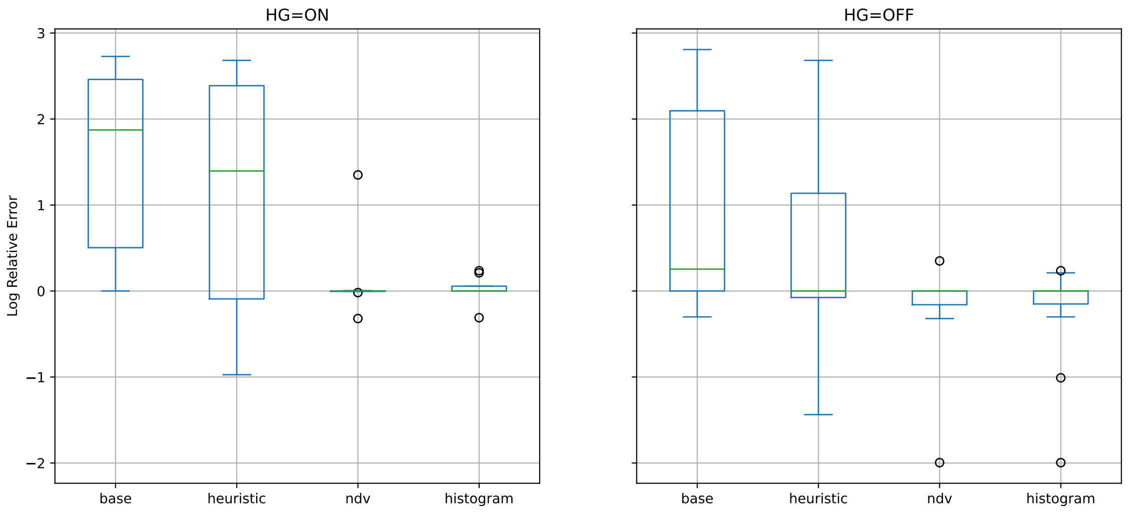


Figure 23: Twitter. Comparison of the logarithmic relative estimation errors measured at all steps in the queries.

The one query for the Twitter dataset which exhibited major changes in the performance of with the hypergraph optimizer enabled was for the `count_replies_and_retweets` query, seen in listing 20.

Listings 21 and 22 and 23 show the query plans generated for this query for three of four statistics configurations with the hypergraph optimizer enabled. The base configuration estimates that a total of 18 665 024 400 (18 billion) rows will be input into final sorting step. With heuristics, this number is reduced by three orders of magnitude to 82 228 688 rows. Using NDV statistics, the final estimate instead comes out to be 2 278. Because none of predicates in the query are such that histogram statistics can be used, the configurations using NDV and histogram statistics arrive at the exact same estimates and thus the same query plan. The true number of rows input to the final sorting step is 15. The reason for these massive misestimations is that replies and retweets are a rare occurrence in the dataset, with the `"retweeted_status"` key appearing in about 1% of documents, and the `"in_reply_to_status_id"` value being null in 99.84% of documents. Thus, with no heuristics or statistics to draw from, the base MySQL version massively overestimates

```
SELECT
    tweeters.j->"$.user.screen_name",
    tweeters.j->"$.id",
    COUNT(*) AS count
FROM (
    SELECT * FROM twitter
    WHERE
    (JSON_VALUE(j, "$.in_reply_to_status_id") IS NULL OR j->"$.in_reply_to_status_id" IS NULL)
    AND (JSON_VALUE(j, "$.retweeted_status.id") IS NULL OR j->"$.retweeted_status.id" IS NULL)
) AS tweeters, (
    SELECT * FROM twitter
    WHERE JSON_VALUE(j, "$.in_reply_to_status_id") IS NOT NULL
) AS replies, (
    SELECT * FROM twitter
    WHERE JSON_VALUE(j, "$.retweeted_status.id") IS NOT NULL
) AS retweets
WHERE
    tweeters.j->"$.id" = replies.j->"$.in_reply_to_status_id"
    AND
    tweeters.j->"$.id" = retweets.j->"$.retweeted_status.id"
GROUP BY
    tweeters.j->"$.user.screen_name",
    tweeters.j->"$.id"
ORDER BY
    count DESC
LIMIT 20
;
```

Listing 20: One of the four queries for the Twitter data sets. It counts the number of replies and retweets for each of each tweet in the data set where that count is greater than 0. It then retrieves the 20 users with the most replies and retweets in total.

the size of the `replies` and `retweets` relations. On the other hand, the configuration relying on heuristics makes the opposite mistake, calculating a much too small size for the `tweeters` table, whose predicates in actuality have a miniscule selectivity. The result of the misestimations of the base and heuristic configurations is that the resulting query plans are based on absolutely wrong assumptions about the sizes of the tables. These overestimations result in the use of nested loop joins where hash joins should be used, and the queries end up timing out before they are able to be completed.

6.4 Timing Experiments

As stated in section 2.2, improved database statistics and selectivity estimates need not necessarily lead to improved database performance, and may even lead to performance regressions in some cases. In addition, if the assumptions which the selectivity estimation calculations rely on turn out to be wrong, the more accurate statistics may in fact lead to cardinality estimations that are worse than when using heuristics or no estimates at all. These worse estimates may also lead to performance regressions.

Therefore, there is no guarantee that the increases in estimation accuracy seen so far will lead to an increase in performance.

6.4.1 TPC-H

Figures 25 and 24 summarize the timing findings for the experiments using the TPC-H benchmark data and queries with hypergraph optimizer off and on, respectively.

```
-> Sort: count DESC, limit input to 20 row(s) per chunk  (rows=20)
  -> Table scan on <temporary>  (rows=18665024400)
    -> Temporary table  (rows=18665024400)
      -> Group aggregate: count(0)  (rows=18665024400)
        -> Nested loop inner join  (rows=2550015633528000)
          -> Nested loop inner join  (rows=18665024400)
            -> Sort: json_extract(twitter.j, '$.user.screen_name'), json_extract(twitter.j,
            ↪  '$.id')  (rows=136620)
              -> Filter: (((json_value(twitter.j, '$.in_reply_to_status_id' returning char(512))
              ↪  is null) or (json_extract(twitter.j, '$.in_reply_to_status_id') is null)) and
              ↪  ((json_value(twitter.j, '$.retweeted_status.id' returning char(512)) is null) or
              ↪  (json_extract(twitter.j, '$.retweeted_status.id') is null)))  (rows=136620)
                -> Table scan on twitter  (rows=136620)
            -> Filter: ((json_value(twitter.j, '$.in_reply_to_status_id' returning char(512)) is
            ↪  not null) and (json_extract(twitter.j, '$.id') = json_extract(twitter.j,
            ↪  '$.in_reply_to_status_id')))  (rows=136620)
              -> Table scan on twitter  (rows=136620)
          -> Filter: ((json_value(twitter.j, '$.retweeted_status.id' returning char(512)) is not
          ↪  null) and (json_extract(twitter.j, '$.id') = json_extract(twitter.j,
          ↪  '$.retweeted_status.id')))  (rows=136620)
            -> Table scan on twitter  (rows=136620)
```

Listing 21: The EXPLAIN output for the `count_replies_and_retweets` query for the base MySQL configuration with the hypergraph optimizer enabled. Because the query timed out, only the estimates are included with the query plan. The true row count for some of the steps can be seen in listing 23 which includes the EXPLAIN ANALYZE output.

```
-> Sort: count DESC, limit input to 20 row(s) per chunk  (rows=20)
  -> Table scan on <temporary>  (rows=82228688)
    -> Temporary table  (rows=82228688)
      -> Group aggregate: count(0)  (rows=82228688)
        -> Sort: json_extract(twitter.j, '$.user.screen_name'), json_extract(twitter.j, '$.id')
        ↪  (rows=745650035403)
          -> Inner hash join (<hash>(json_extract(twitter.j,
          ↪  '$.id'))=<hash>(json_extract(twitter.j, '$.retweeted_status.id')))
          ↪  (rows=745650035403)
            -> Nested loop inner join  (rows=60642662)
              -> Filter: (((json_value(twitter.j, '$.in_reply_to_status_id' returning char(512))
              ↪  is null) or (json_extract(twitter.j, '$.in_reply_to_status_id') is null)) and
              ↪  ((json_value(twitter.j, '$.retweeted_status.id' returning char(512)) is null) or
              ↪  (json_extract(twitter.j, '$.retweeted_status.id') is null)))  (rows=4932)
                -> Table scan on twitter  (rows=136620)
              -> Filter: ((json_value(twitter.j, '$.in_reply_to_status_id' returning char(512)) is
              ↪  not null) and (json_extract(twitter.j, '$.id') = json_extract(twitter.j,
              ↪  '$.in_reply_to_status_id')))  (rows=12296)
                -> Table scan on twitter  (rows=136620)
            -> Hash
              -> Filter: (json_value(twitter.j, '$.retweeted_status.id' returning char(512)) is
              ↪  not null)  (rows=122958)
                -> Table scan on twitter  (rows=136620)
```

Listing 22: The EXPLAIN output for the `count_replies_and_retweets` query for the MySQL using heuristics only and with the hypergraph optimizer enabled.

```
-> Sort: count DESC, limit input to 20 row(s) per chunk  (rows=20) (actual rows=15)
  -> Table scan on <temporary>  (rows=2278) (actual rows=15)
    -> Temporary table  (rows=2278) (actual rows=15)
      -> Group aggregate: count(0)  (rows=2278) (actual rows=15)
        -> Sort: json_extract(twitter.j, '$.user.screen_name'), json_extract(twitter.j, '$.id')
        ↪  (rows=108699) (actual rows=150)
          -> Inner hash join (<hash>(json_extract(twitter.j,
          ↪  '$.id'))=<hash>(json_extract(twitter.j, '$.retweeted_status.id')))  (rows=108699)
          ↪  (actual rows=150)
            -> Inner hash join (<hash>(json_extract(twitter.j,
            ↪  '$.id'))=<hash>(json_extract(twitter.j, '$.in_reply_to_status_id')))
            ↪  (rows=110681) (actual rows=231)
              -> Filter: (((json_value(twitter.j, '$.in_reply_to_status_id' returning char(512))
              ↪  is null) or (json_extract(twitter.j, '$.in_reply_to_status_id') is null)) and
              ↪  ((json_value(twitter.j, '$.retweeted_status.id' returning char(512)) is null) or
              ↪  (json_extract(twitter.j, '$.retweeted_status.id') is null)))  (rows=136392)
              ↪  (actual rows=168212)
                -> Table scan on twitter  (rows=136620) (actual rows=170285)
              -> Hash
                -> Filter: (json_value(twitter.j, '$.in_reply_to_status_id' returning char(512))
                ↪  is not null)  (rows=213) (actual rows=265)
                  -> Table scan on twitter  (rows=136620) (actual rows=170285)
            -> Hash
              -> Filter: (json_value(twitter.j, '$.retweeted_status.id' returning char(512)) is
              ↪  not null)  (rows=1451) (actual rows=1808)
                -> Table scan on twitter  (rows=136620) (actual rows=170285)
```

Listing 23: The EXPLAIN ANALYZE output for the `count_replies_and_retweets` query for the MySQL with histogram statistics and the hypergraph optimizer enabled.

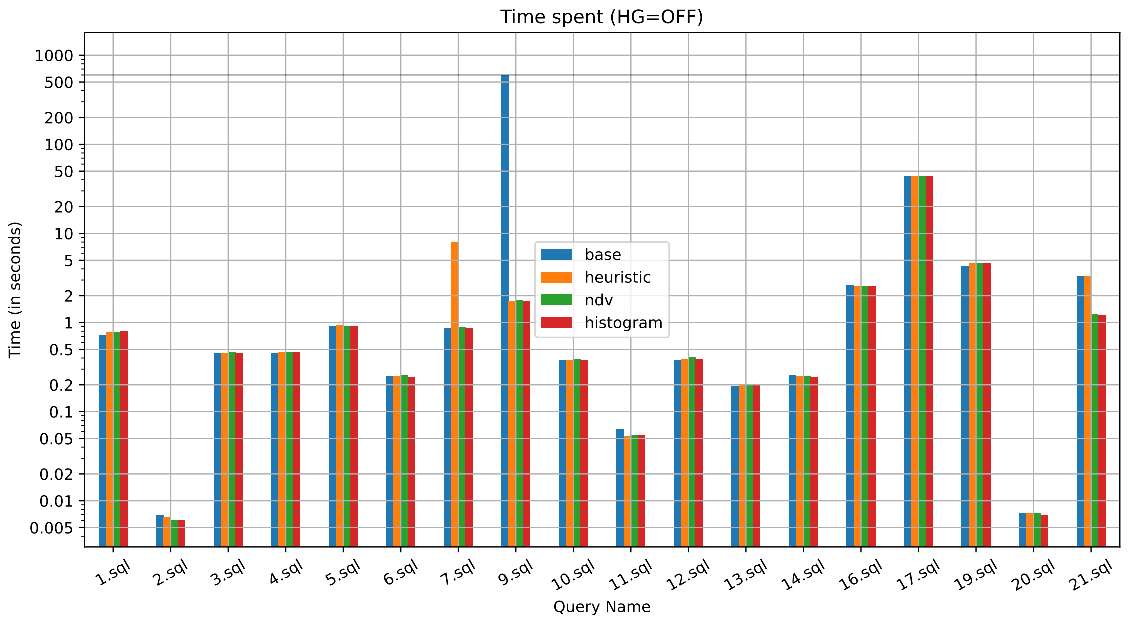


Figure 24: TPC-H. Time spent per query for each of the four statistics configurations using the current MySQL optimizer. The statistics configurations are grouped together for each query in the order of base MySQL, heuristics only, NDV statistics, and histogram statistics. Note the logarithmic y-axis. Higher is worse.

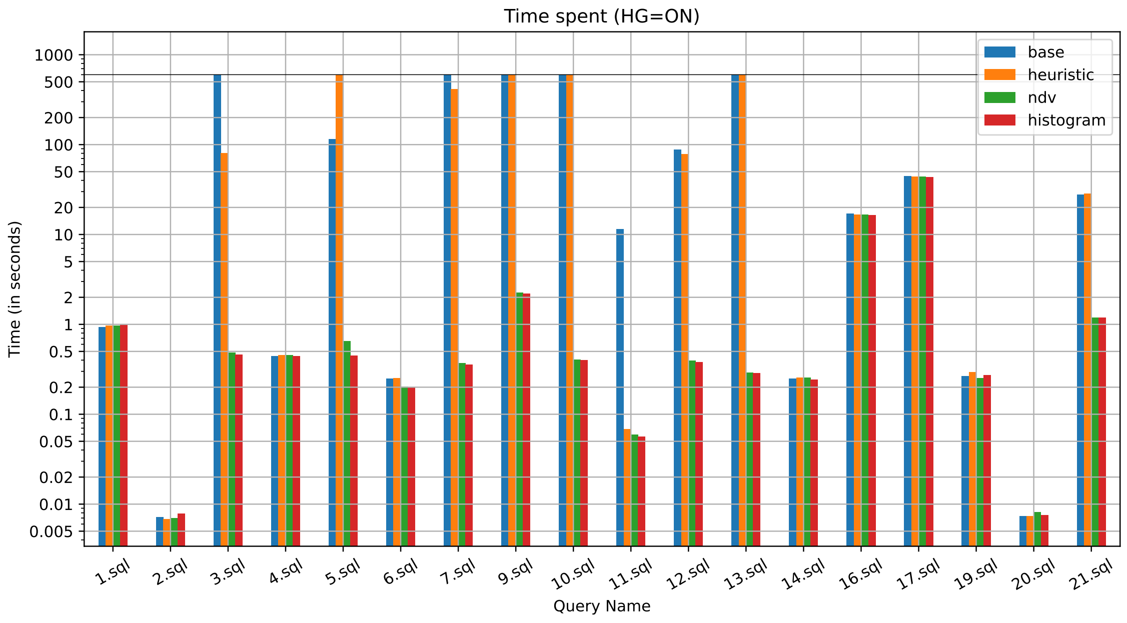


Figure 25: TPC-H. Time spent per query for each of the four statistics configurations using the hypergraph optimizer. Higher is worse.

As expected, the change in performance due to the introduction of statistics (and heuristics) is much more noticeable when the hypergraph optimizer is used. When using the old/current optimizer, most query plans stay exactly the same (see figure 26 for more details on the exact differences). The three queries with a significant change in performance queries 7, 9 and 21, and for the first of these, the only change is that when using only heuristics, a 9x slowdown is observed. This slowdown is not seen however when statistics are introduced. In fact, with statistics inserted, no real regressions in performance are observed. On the other hand, for query 9, the base version of MySQL times out, resulting in the configurations using statistics seeing a calculated speed-up of (at the very least) 500x. For query 21, a speedup of around 2.5x was measured.

When using the hypergraph optimizer, more queries experienced speedups than those that experienced no change in performance and those that experienced performance regressions combined (see figures 27, 28). And in fact, when the hypergraph optimizer was not used, the only case of a serious performance regression was seen when using heuristics and disappeared once collected statistics were inserted. With the hypergraph optimizer disabled, the largest query speedup was observed for all three cases of heuristics, NDV statistics and histogram statistics. If it is the case that the largest increases in performance can be achieved by simply enabling heuristics for JSON data, this weakens the argument for introducing JSON statistics. However, with the hypergraph optimizer enabled, the majority of the speedups observed appear only when proper statistics were used, or the speedup achieved using heuristic was much smaller. This can be seen in further detail in figure 29, which shows the cases in which the NDV and histogram statistics outperformed the heuristics-based statistics configuration.

As expected, no significant performance differences were measured when using NDV statistics and histogram statistics.

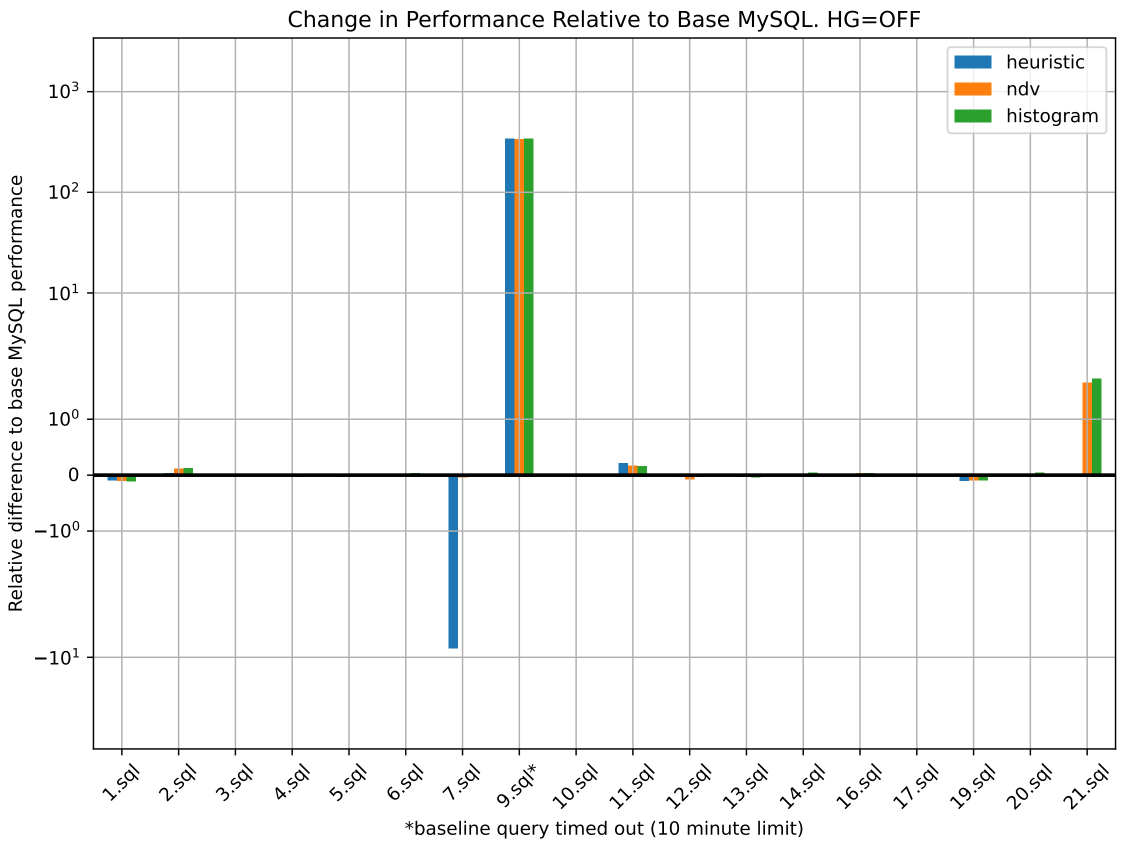


Figure 26: TPC-H. The measured relative difference between the performance of the non-base statistics configurations compared to the performance of base MySQL. Higher is better.

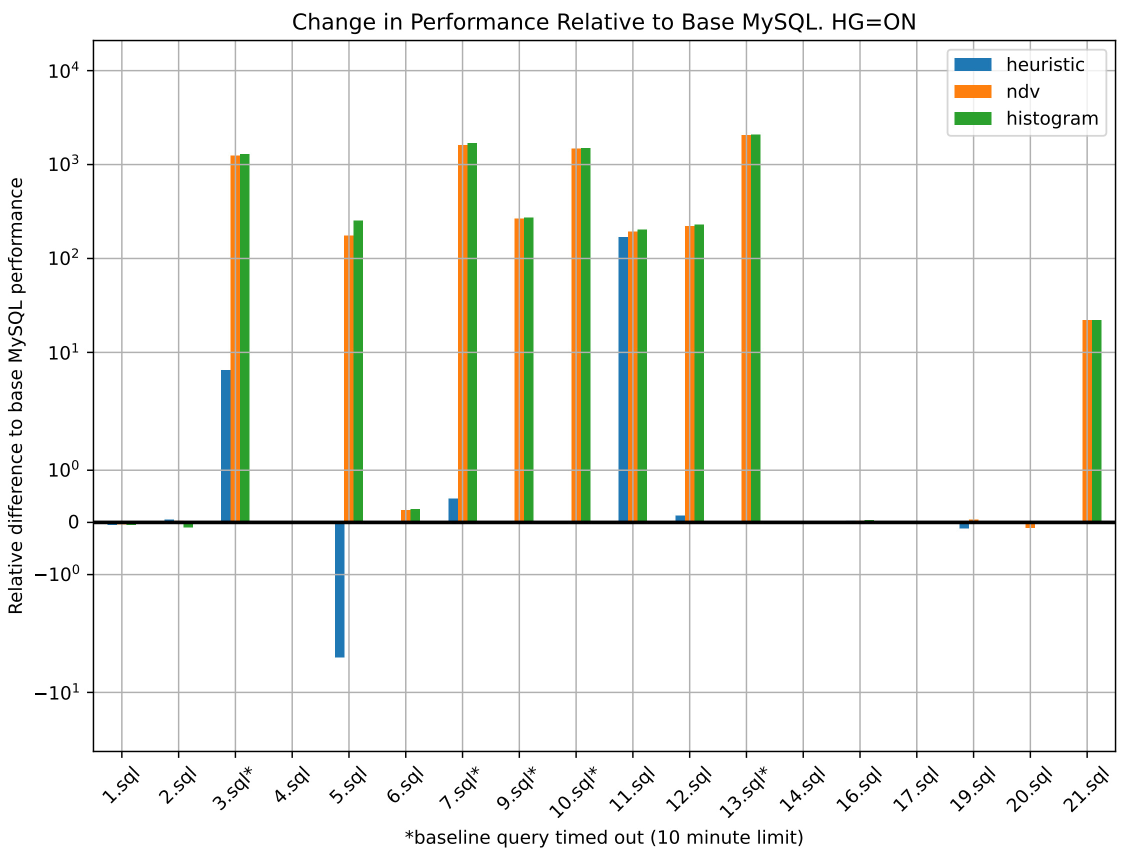


Figure 27: TPC-H. Changes in performance relative to the baseline query. Positive and negative scales are symmetric, so -10^1 means that a query was 10x slower than the baseline query. Higher is better.

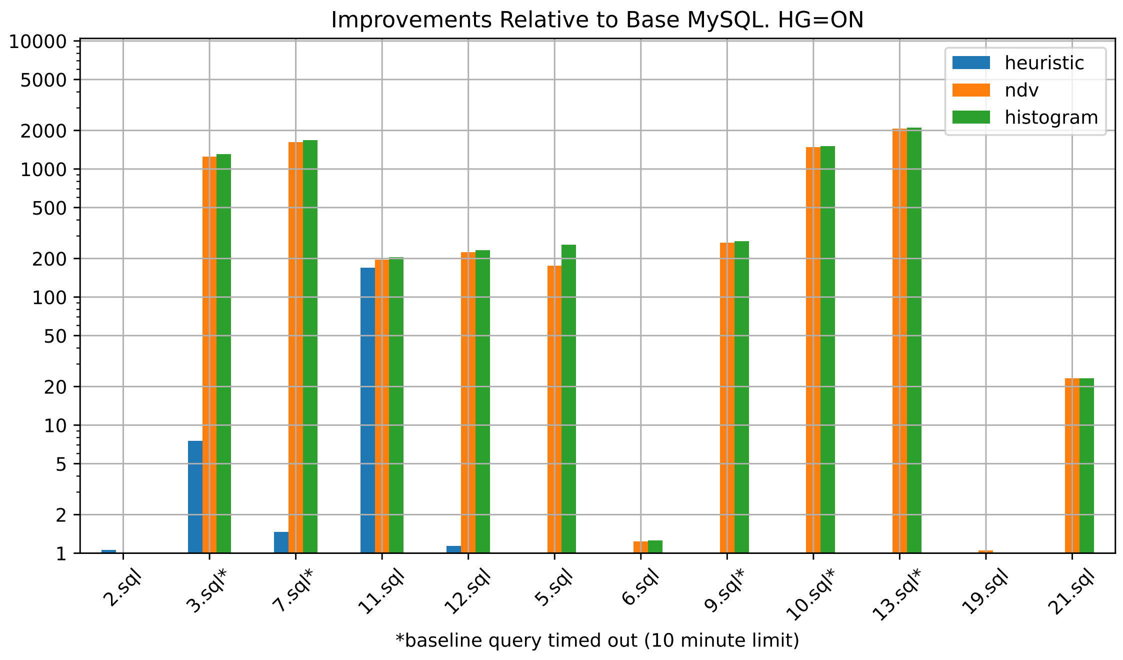


Figure 28: TPC-H. Positive changes in performance relative to the baseline query. Higher is better.

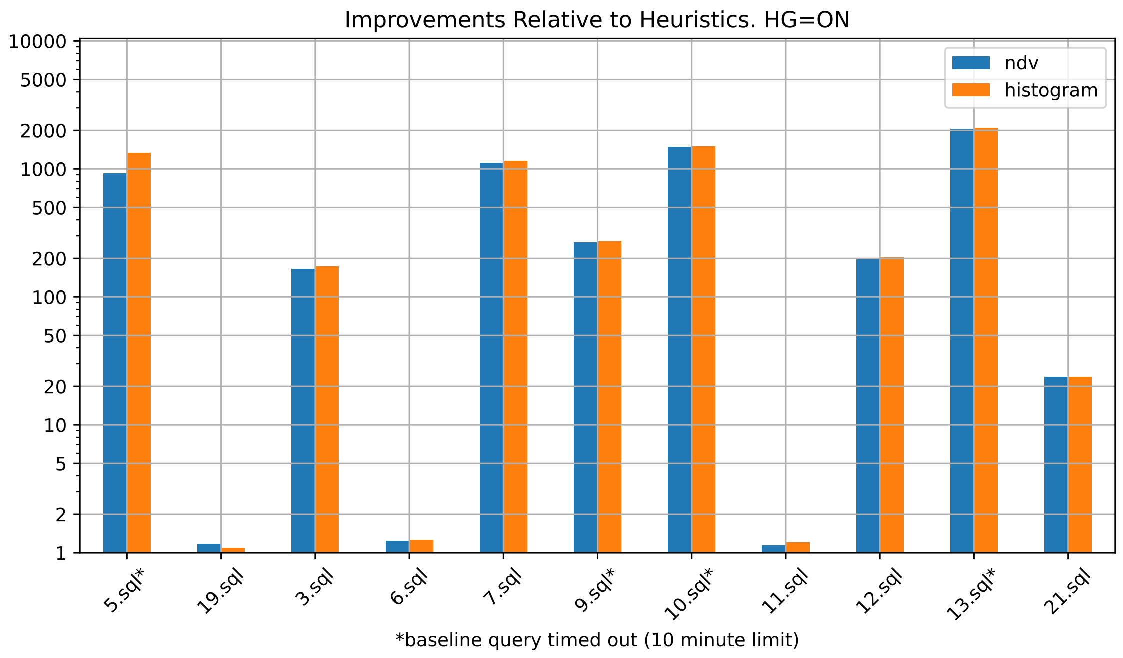


Figure 29: TPC-H. Positive changes in performance relative to the performance of the heuristics-only configuration. Higher is better.

6.4.2 Airbnb

With the current optimizer, no significant change in query execution times between the different MySQL configurations was detected (see figure 31). With the hypergraph optimizer enabled however, a majority (five out of eight) of queries saw major speedups when statistics were introduced. Of these five queries, three were not helped when heuristics were enabled and required NDV or histogram statistics to achieve a gain in performance.

No regression in performance was detected for the Airbnb data and queriess, both with using the current optimizer and using the hypergraph optimizer.

6.4.3 Twitter

Of the four queries for the Twitter data, only the `count_replies_and_retweets` query exhibited major changes in performance when statistics were added to the system. However, whereas the discussion on estimation accuracy and the resulting query plans focused on the case where the hypergraph optmizer is enabled, a change (albeit a smaller one) was also observed when the current optimizer was used.

This regression is due to a mis-estimation that only occurs using the old hypergraph optimizer, where the size of the `tweeters` relation is massively mis-estimated, causing the wrong join method to be used. Why this misestimation happens I cannot explain. The size of the `tweeters` relation is correctly estimated when the hypergraph query optimizer is in use, but is incorrectly estimated when the current optimizer is in use – despite the fact that the relation is only a table scan plus a filter. Sadly, time constraints preclude further investigation into the matter. It is possible that this regression is caused by a bug and is not a true limitation of the current optimizer. Becausee hypothesis of the cause being a minor and fixable bug cannot be established, I will regard the regression as a true

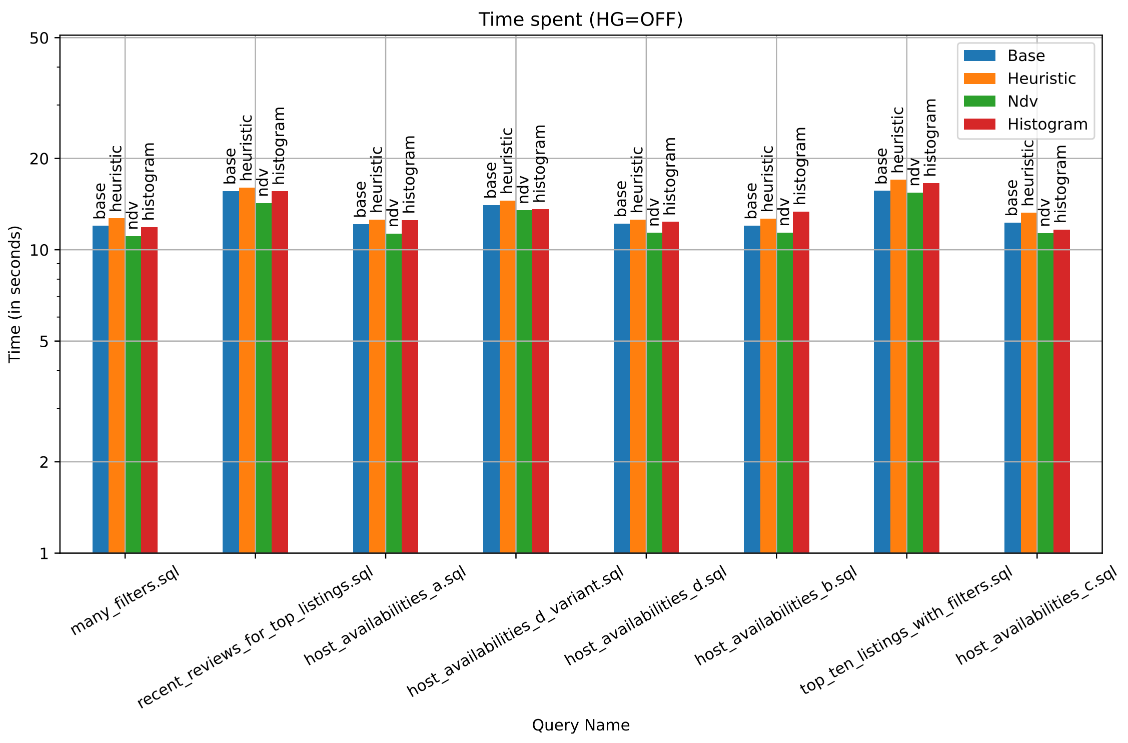


Figure 30: Airbnb. Time spent per query for each the four statistics configurations with using the current MySQL optimizer. The minor variations are not significant.

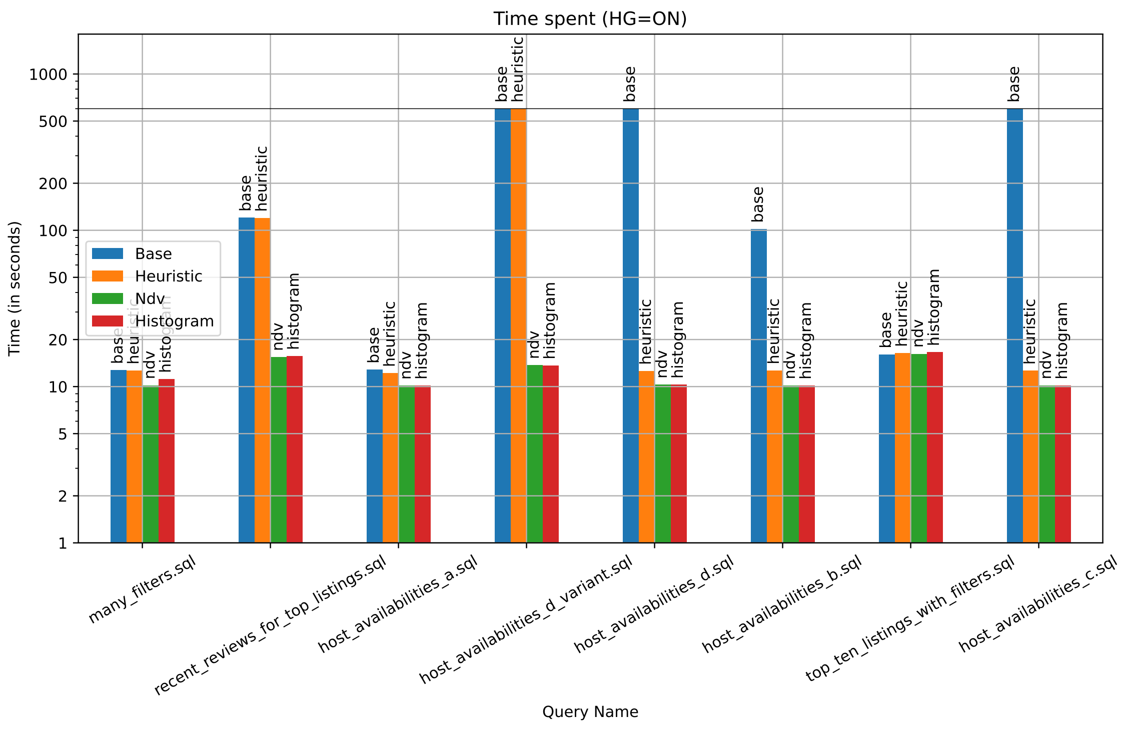


Figure 31: Airbnb. Time spent per query for each the four statistics configurations with using the hypergraph optimizer. The thin line above the 500 mark shows the timeout limit. Higher is worse.

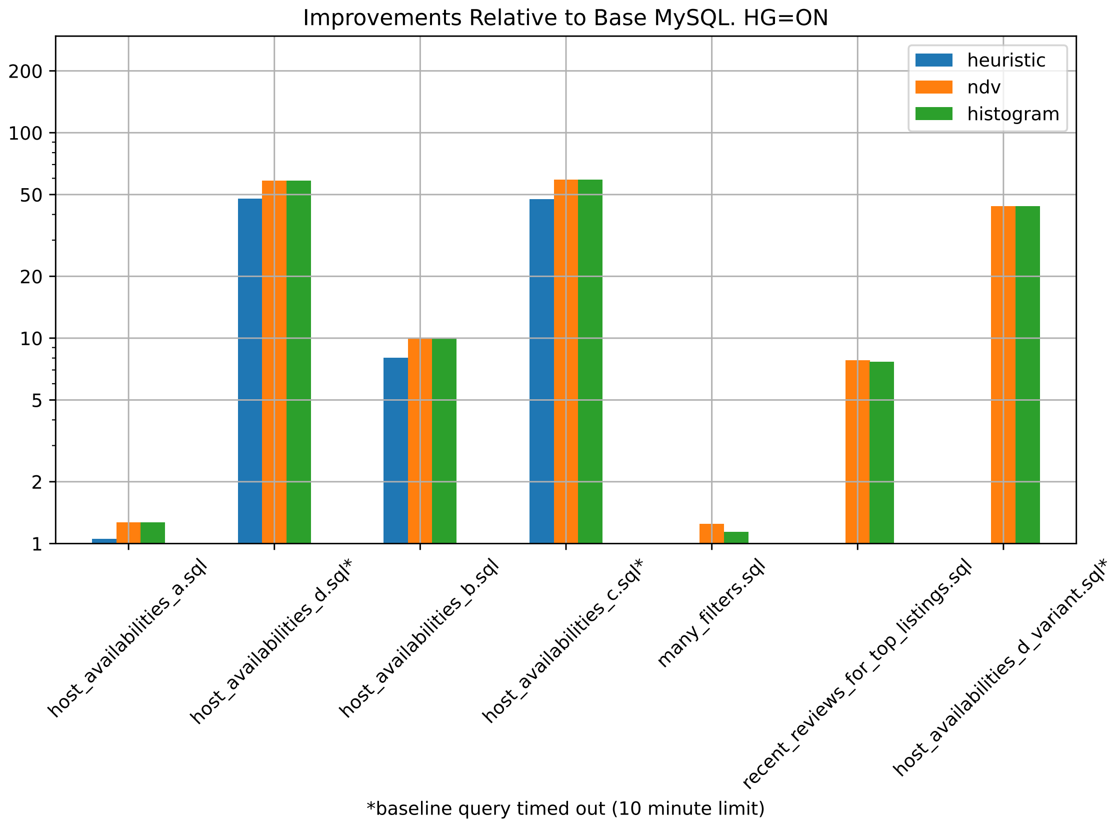


Figure 32: Airbnb. Positive changes in performance relative to the baseline query. Higher is better.

regression for this report.

However, the major difference in performance is the massive speedup achieved for `count_replies_and_retweets` when using statistics and with the hypergraph optimizer enabled. Here, an increase in execution speed of far more than 100x was measured.

A keen-eyed reader may notice that there seem to be minor regressions for the three other Twitter data queries, especially with the hypergraph optimizer enabled. However, inspection of the query plans generated for these queries reveals that they are identical. These variations are therefore caused by some unknown external factor. The execution times of these other queries are relatively short, ranging from half a second to two seconds, which makes their exact timing more susceptible minor changes in the computing environment.

As discussed in the section on estimates for Twitter data, the heuristics-based version's query plan was better than that of the base version. It is therefore likely that, with a higher timeout limit, the performance of the heuristics-based configuration would be measured to be better than the base version. Thus, in the case of the Twitter queries, using heuristics would only yield gains in performance for both optimizers and no regressions. Those gains would clearly be significantly smaller than the gains achieved if proper statistics were to be used along with the hypergraph optimizer however.

6.5 Discussion

Overall, the introduction of JSON statistics lead to an improvement in the quality of estimations produced by MySQL – not only in comparison to the base version of MySQL,

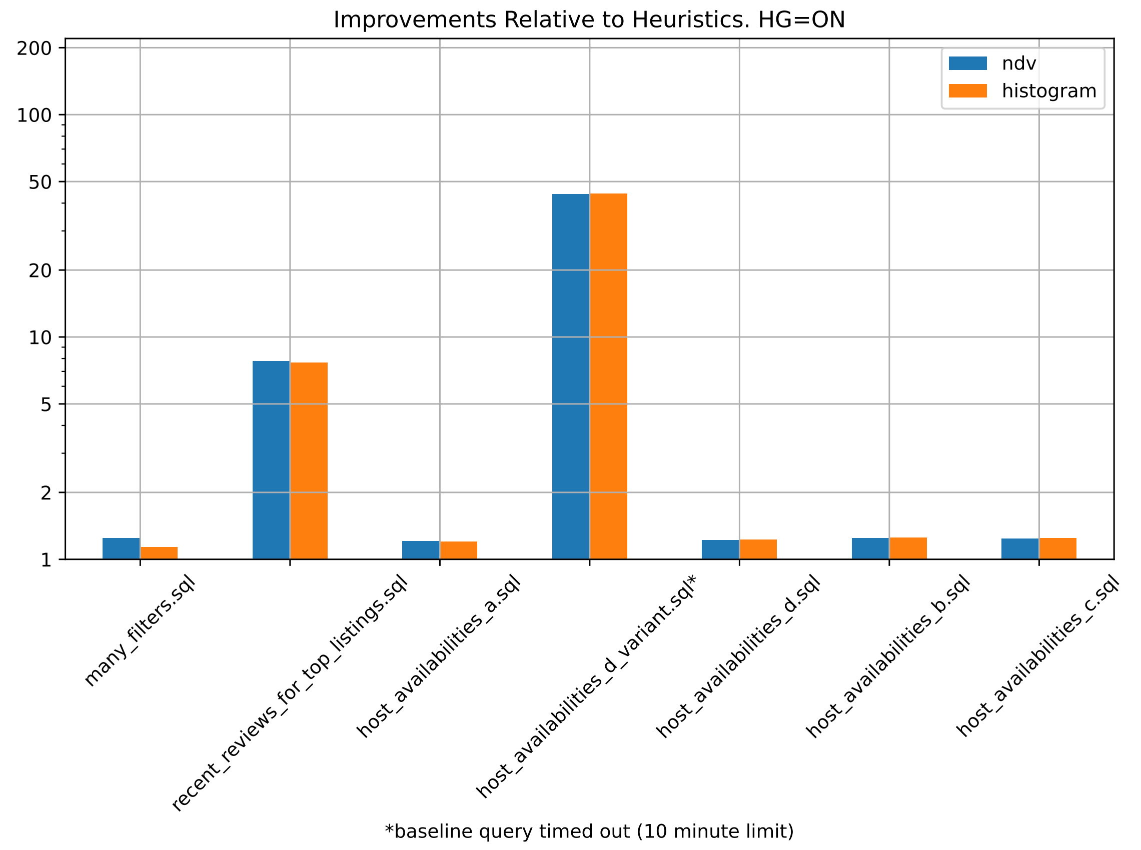


Figure 33: Airbnb. Positive changes in performance relative to the performance of the heuristics-only configuration. Higher is better.

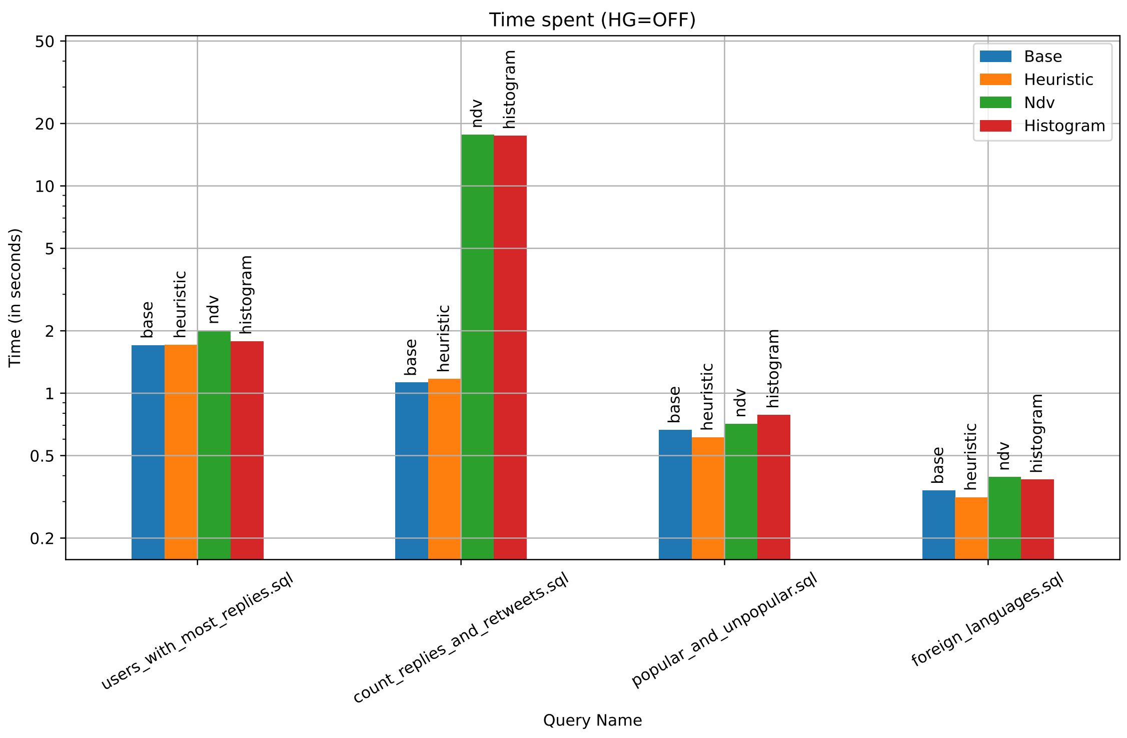


Figure 34: Twitter. Time spent per query for each the four statistics configurations with using the current MySQL optimizer. Higher is worse.

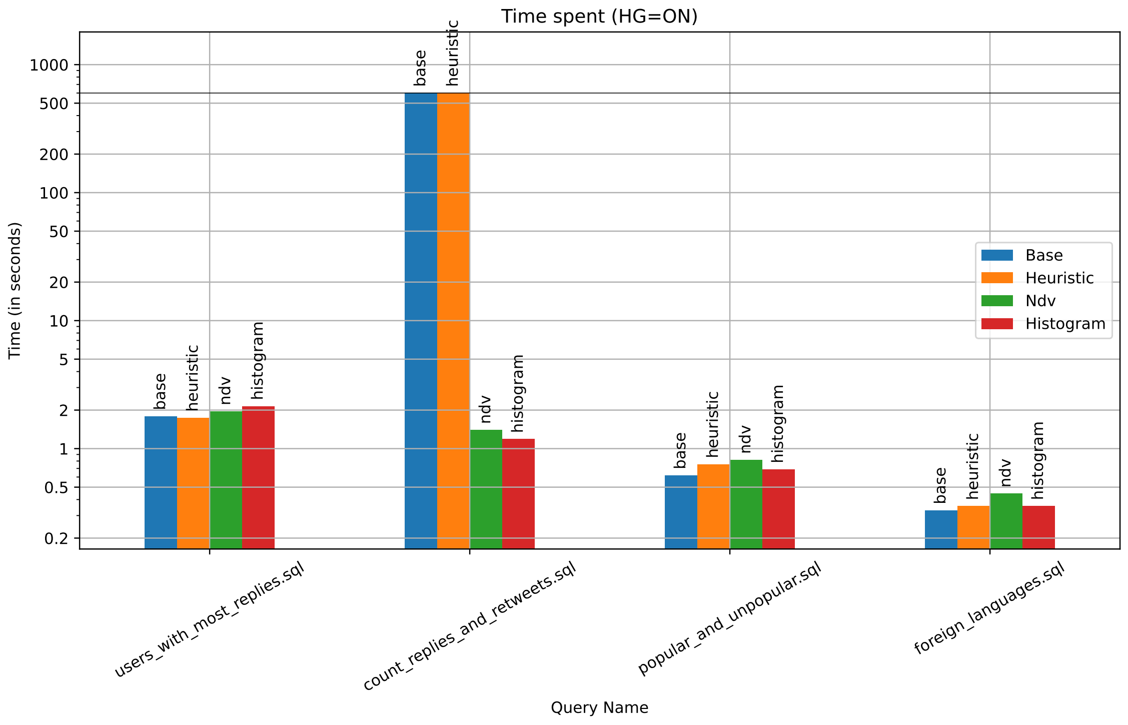


Figure 35: Twitter. Time spent per query for each the four statistics configurations with using the hypergraph optimizer. Higher is worse.

but also in comparison to the modified version employing heuristics. The change was positive both when the current and hypergraph optimizers were used.

The impact of the changes to MySQL can also be observed to have had a mostly positive impact on the performance of MySQL (with respect to queries of JSON data). With MySQL's current optimizer in use, there were cases where a degradation in performance was observed. However, most were due to misestimations caused by only heuristics being available and only case of degraded performance persisted when statistics were introduced (specifically, the query `count_replies_and_retweets` for the Twitter dataset).

With hypergraph optimizer enabled, the heuristics-only statistics configuration can also not be said to be a perfect improvement over MySQL's current behavior. For example, TPC-H query 5 saw at least a 5-fold increase in its execution time when heuristics were used. However, in the majority of cases, heuristics can be seen to lead to an improvement in performance. The introduction of statistics to the hypergraph optimizer cannot be observed to be anything other than a major success in terms of impact on performance: not a single one of the experiment queries experienced performance regressions when JSON statistics were used. And half were seen to enjoy major gains in performance.

If the changes introduced in this prototype were to be introduced to MySQL, the lack of automatic statistics-gathering would likely lead to a majority of estimates being made with the use of heuristics rather than statistics. Thus, despite the good results achieved when introducing JSON statistics, it is likely that users would experience regressions in performance for many of their JSON data queries. Even if statistics were gathered, the queries used in these experiments are by no means exhaustive and there is practically guaranteed to exist queries for which, even when detailed JSON statistics are collected and used, the query optimizer will make worse a choice in query plans than it currently does. The exact impact on the proposed changes on real-world JSON queries is difficult to judge, and deciding when the gains from increases in performance are worth the losses from the eventual regressions even more so. However, I believe that, given the results found in this section, that this might be a matter worth investigating.

The performance gain from changing from no JSON predicate estimation support at all, to heuristics, to proper statistics is visible at each step. And when examining estimates, it is clear that histograms do offer much more accuracy than NDV statistics. The Python prototype shows the same increase in accuracy when comparing the latter to the former. However, this increase in estimation accuracy could not be seen during these experiments to translate into any tangible improvements in performance. I theorize that the reason for this is that for the queries used in this experiment, the NDV statistics are good enough for the query optimizer to choose the best possible plan it can come up with for the query; not even perfectly accurate estimates could make it come up with a better plan[14]. The reason for this, I belive, is twofold: First, of the predicates used in the queries, only a few truly benefitted from the availability of histogram statistics[15]. Second, there is a diminishing return effect when going from NDV to histogram statistics compared to when going from nothing to heuristics and from heuristics to NDV statistics (see for example figures 21 and 23). The queries used in these experiments were not large and complex enough for these much smaller relative differences in estimates to have an impact. However, without further tests to back up this hypothesis, the best option seems to be to use NDV statistics rather than histogram statistics as the results found here do not justify the increased storage costs incurred by the use of histogram statistics. As discussed in section 5.3, NDV statistics can in some cases offer more accurate estimates than histogram statistics if the size of the statistics is limited or the sample size is low.

7 Conclusion

In this section, I will summarize the work done for this thesis by answering the research questions defined in section 1.2 and discussing the overall outcome of the project. Then, a brief discussion of potentially interesting future research areas related to the project.

7.1 Answering the Research Questions

The goal of this thesis has been to find an effective and viable method for improved selectivity estimation for JSON data in MySQL. The goal of this report has been to document the process and its outcome. To this end, section 1.2 introduced four research questions that the report should clearly answer to make clear whether the goals the project were achieved or not. These research questions will now be answered.

> **Research Question 1** *What are the existing, published methods for collecting and using statistics for JSON data in an RDBMS context, which also fit well with MySQL's design and architecture?*

[14]This could be tested by injecting the true cardinalities into the optimizer, but sadly a lack of time did not allow for this

[15]This might have been different if array operations had been implemented into the MySQL prototype and used in the queries. The Python prototype experiments showed that for these predicates, histogram statistics outperformed NDV statistics in terms of accuracy to a much greater degree than for other operators.

Section 3 presents the findings related to this research question in detail. In brief, no existing methods could be found which were directly transferable to the context of MySQL. However, much of the existing work does contain concepts and ideas which are transferable to the context of a MySQL implementation of a JSON statistics system.

> **Research Question 2** *Can a working system for collecting and using JSON statistics realistically be implemented into MySQL?*

Section 4 shows that it is possible to implement a prototype system for using JSON statistics to make estimates for JSON predicates in MySQL. This is not a fully fledged system, and it is brittle and does suffer from instability issues. However, the core functionaly was put in place and no major problems or incompatibilities were encountered during this implementation. It is therefore likely that a properly engineered version of the system could be implemented in MySQL.

The MySQL prototype does not generate its own statistics. The Python prototype shows that collecting and generating the statistics of a design as presented in section 4.1 is very much feasible.

However, a question arises over how the restrictions that MySQL places on sampling – specifically the restrictions on the size of the sampling buffer – would impact the statistics gathered by such a system. From the Python experiments (section 5), it is clear that for certain datasets, accuracy quickly degrades when samples become too small. The average size of a document in the Twitter dataset is 2.5 KB. If one imagines the Twitter dataset to have the modest size of 2 GB (which makes it contain 838 860 docuemnts total), the default sampling buffer size of 20 MB would give a sample rate of 0.1%. For the Airbnb documents, which are larger in size on average, the sampling rate would be even lower. It is possible that such low sampling rates for even such modestly sized data sets would lead to statistics of poor quality. In addition, MySQL's sampling strategy is coarser than that of the Python prototype and can more easily lead to biases in the data. This will require further investigation before a conclusion can be drawn.

Additionally, while the proposed system has been implemented and been shown to work, there are some naive choices in the design which hamper it. Among these are a susceptibility to ballooning in size when collecting histograms for large arrays and duplication of information. However, these are problems that can be fixed (or at least alleviated) with relative ease.

> **Research Question 3** *With such a system implemented, are MySQL's estimates for the selectivity of predicates over JSON expressions improved in a meaningful way?*

The answer to this question is yes. As shown in section 6, the introduction of JSON statistics to MySQl lead to a marked improvement in estimates. How dramatic the improvement in estimation accuracy is depends on the dataset and the queries tested. For example, the introduction of statistics led to a huge improvement in the estimates related to the Twitter dataset and queries (see figure 23). Improvements were also clearly visible for the TPC-H and Airbnb datasets and queries, but the improvements were less marked and differed more between the two optimizers.

As discussed in section 2, improved estimate accuracy does not necessarily lead to improvements in performance. Query optimizers are complex and faulty, and may not always respond positively to better quality information. Therefore, the improvements in selectivity estimation accuracy can be seen as big 'win' in and of itself.

However, for the method to be practical and viable, it should be possible to integrate it into database system and expect and overall gain in performance. Making changes (large ones at least) to a database's query optimizer so that it will better utilize the added information is usually neither practical nor advantageous. Therefore, for the proposed system to truly be practical and worth an implementation, it should demonstrate that not only does it increase accuracy, but that noticeable gains in performance are achieved without modifications that change the expected behavior of the rest of the system. This leads into the fourth and final research question.

Research Question 4 *How would a system for JSON statistics, if introduced to MYSQL, affect MySQL's performance when dealing with JSON data?*

As seen in section 6 and discussed in section 6.5, the answer to this question seems to depend very much on which optimizer is used. The current MySQL optimizer is rather conservative with the plans it chooses. This is in part due to a design decision of examining only left-deep trees, and in part because it does not consider all information that would technically be available to it. This leads to fewer opportunities for the large increase in JSON predicate selectivity estimation accuracy to shine. Of the 30 of queries tested in total, only four showed a significant change in performance – either positive or negative – with the introduction of statistics and/or heuristics for JSON data to MySQL. And for one of these queries the only time a difference in performance was recorded was when only heuristics were used.

In addition, the changes in performance were both negative and positive (though the mean speedup was larger than the mean slowdown). As such, the conclusion becomes that when using MySQL's current optimizer, the statistics had little overall impact on the system.

The same cannot be said of MySQL's behavior with the hypergraph optimizer enabled, however. Because of its ability to consider a broader range of plans and because it can draw on more statistical information when doing so, the hypergraph optimizer was found to produce a much broader set of query plans for the queries tested[16]. This resulted in a much more diverse set of measurements, and one in which the advantage of JSON statistics became very clear. With the queries tested, half were observed to have increased in performance when executed with statistics enabled and inserted, compared to the base version of MySQL. And over a quarter were observed to have better performance than the prototype version of MySQL using heuristics only. And when statistics were available, none of the queries tested were seen to exhibit worse performance than either the base version of MySQL, or the version utilizing heuristics.

[16]Interestingly, it seems that the hypergraph optimizer, when fed with JSON statistics, came up with many of the same plans that the current MySQL optimizer chose. And when statistics were missing, worse plans were chosen than what would have been the case had the current optimizer been used. It is therefore possible, in the case of MySQL specifically, that a cheaper solution than the introduction of JSON statistics would be to simply always use the current/old optimizer when detecting JSON queries. However, it is also possible that the current optimizer would benefit more from JSON statistics than could be shown in the experiments performed for this project when presented with larger datasets and more complex queries.

7.2 Future Work

Several topics that would have been interesting to explore were left out of the project due to feasability and time constraints. Many of these could be interesting to explore in future research efforts on the topic. Some of these include:

- Exploring better algorithms for choosing which key paths to keep, and which values key paths to collect which information on. The current design system is very naive in certain cases. For example like if each document contains a massive array of floats (say for example, the vertices of some large multi-polygon), statistics will be recorded for every single index of that array, despite the very low likelihood of that data ever being used in a predicate. It is also possible that certain key paths will benefit more from histogram statistics than others. Exploring whether to, instead of choosing either NDV or histogram statistcs, one could choose a hybrid method which collects histogram statistics from only the most prominent key paths or key paths whose properties indicate that they will benefit more than average from histogram data. How to determine this is an open question.

- I would really have liked to expand on the array statistics aspects of this work, which is the technique I feel is most unique and interesting of the ones presented in the design. The Python experiments with the Airbnb data show that the technique has promise. However, a closer look at the Twitter data set and the statistics generated for it reveals its limitations (though also sparking imagination – seeing its potential). Again, arrays are not used as ordered collections, but unordered collections, so the use for summary statistics is likely there. This could be especially useful with wildcard paths[17], which I also could not explore. One could image a query like `j->"$.hashtags.*.string" = "MySQL"` to get all tweets where any of the hashtags (whose information is stored in an array) equal "MySQL".

- An alternative to better algorithms would be to explore allowing the user to choose which key paths either to include in statistics gathering, or exclude from statistics gathering.

- Support for automatically determining that certain predicates preclude other predicates from impacting selectivity estimates. For example, if the predicate `j->"$.object" IS NULL` is applied, then that would mean that the predicate `j->"$.object.number" > 0` could not apply, because that path cannot be reached in documents for which the first predicate is true. This goes the other way as well: a predicate checking that a property is not null should not impact selectivity estimates if another predicate in the same iterator filters on the values of that property or children of that property. A simple example would be the selectivity factor A=`j->"$.path" IS NOT NULL` and factor B=`j->"$.path" = "some string"`. Right now, if a query checks both of these and the path is uncommon, the system will end up with a severe underestimation of the selectivity due to squaring (or worse) the real selectivity of the predicates (which is just B). So if B=A=0.01, the selectivity will be reported is 1/100th of its real value.

- It can be imagined that an attacker could construct a lookup path such that the system is tricked into assuming either very small result size for an actually large

[17]Adding wildcard support to the current MySQL implementatation of the design would not be difficult. A design consideration to be had though is that it would be better for performance reasons to store paths either as a list or using a nested structure, instead of using a hashmap (as I suggested for the design).

result (and thus choosing a very inefficient plan), or assuming a very large result size for a tiny result and also choosing a worse plan. This could potentially lead to a severe degradation in performance, and may be worth investigating.

- Compare to the cost and gains in performance when using virtual columns with histograms vs JSON histograms. It would also be interesting to see the performance differences between virtual columns with indexes plus histograms vs virtual columns with indexes plus JSON histogram.

Bibliography

[1] *22.1 Overview of JSON Data Guide.* URL: https://docs.oracle.com/en/database/oracle/oracle-database/19/adjsn/json-dataguide.html#GUID-8A431168-E23B-493D-8190-8A26A8D0BCF1 (visited on 12th July 2023).

[2] Ashraf Aboulnaga and Surajit Chaudhuri. 'Self-Tuning Histograms: Building Histograms without Looking at Data'. In: *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data.* SIGMOD '99. Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 1999, pp. 181–192. ISBN: 1581130848. DOI: 10.1145/304182.304198. URL: https://doi.org/10.1145/304182.304198.

[3] *ANALYZE TABLE statement.* URL: https://dev.mysql.com/doc/refman/8.0/en/analyze-table.html (visited on 18th June 2023).

[4] Tim Bray and Douglas Crockford. *The JavaScript Object Notation (JSON) Data Interchange Format.* RFC 8259. RFC Editor, 2017, pp. 1–16. URL: https://www.rfc-editor.org/rfc/rfc8259.

[5] Donald D Chamberlin et al. 'A history and evaluation of System R'. In: *Communications of the ACM* 24.10 (1981), pp. 632–646.

[6] Craig Chasseur, Yinan Li and Jignesh M Patel. 'Enabling JSON Document Stores in Relational Systems.' In: *WebDB.* Vol. 13. 2013, pp. 14–15.

[7] *Database SQL Tuning Guide: Histograms.* URL: https://docs.oracle.com/database/121/TGSQL/tgsql_histo.htm#TGSQL-GUID-DA1B97DA-DFE5-47CA-B8A0-57AB248B10EF (visited on 21st June 2023).

[8] Simon Dooms, Toon De Pessemier and Luc Martens. 'MovieTweetings: a Movie Rating Dataset Collected From Twitter'. In: *Workshop on Crowdsourcing and Human Computation for Recommender Systems, CrowdRec at RecSys 2013.* 2013.

[9] Dominik Durner, Viktor Leis and Thomas Neumann. 'JSON Tiles: Fast Analytics on Semi-Structured Data'. In: SIGMOD '21. Virtual Event, China: Association for Computing Machinery, 2021, pp. 445–458. DOI: 10.1145/3448016.3452809. URL: https://doi.org/10.1145/3448016.3452809.

[10] R. Elmasri and S. Navathe. *Fundamentals of Database Systems.* Seventh. Pearson, 2015.

[11] *EXPLAIN Statement.* URL: https://dev.mysql.com/doc/refman/8.0/en/explain.html (visited on 18th July 2023).

[12] *Fast query performance with MySQL Hypergraph Optimizer for HeatWave.* 17th Jan. 2023. URL: https://blogs.oracle.com/mysql/post/fast-query-performance-with-mysql-hypergraph-optimizer-for-heatwave (visited on 20th June 2023).

[13] Philippe Flajolet et al. 'Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm'. In: *Discrete Mathematics and Theoretical Computer Science.* Discrete Mathematics and Theoretical Computer Science. 2007, pp. 137–156.

[14] Joseph M Hellerstein, Michael Stonebraker, James Hamilton et al. 'Architecture of a database system'. In: *Foundations and Trends® in Databases* 1.2 (2007), pp. 141–259.

[15] *Histogram Statistics in MySQL.* 2nd Oct. 2017. URL: https://dev.mysql.com/blog-archive/histogram-statistics-in-mysql/ (visited on 23rd June 2023).

[16] *Inside Airbnb: Get the Data.* URL: http://insideairbnb.com/get-the-data (visited on 12th June 2023).

[17] *Inside Capacitor, BigQuery's next-generation columnar storage format.* URL: https://cloud.google.com/blog/products/bigquery/inside-capacitor-bigquerys-next-generation-columnar-storage-format (visited on 12th June 2023).

[18] *JSON Function Reference.* URL: https://dev.mysql.com/doc/refman/8.0/en/json-function-reference.html (visited on 24th June 2023).

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

[20] Zhen Hua Liu, Beda Hammerschmidt and Doug McMahon. 'JSON Data Management: Supporting Schema-Less Development in RDBMS'. In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data.* SIGMOD '14. Snowbird, Utah, USA: Association for Computing Machinery, 2014, pp. 1247–1258. ISBN: 9781450323765. DOI: 10.1145/2588555.2595628. URL: https://doi.org/10.1145/2588555.2595628.

[21] Zhen Hua Liu et al. 'Closing the Functional and Performance Gap between SQL and NoSQL'. In: *Proceedings of the 2016 International Conference on Management of Data.* SIGMOD '16. San Francisco, California, USA: Association for Computing Machinery, 2016, pp. 227–238. ISBN: 9781450335317. DOI: 10.1145/2882903.2903731. URL: https://doi.org/10.1145/2882903.2903731.

[22] Zhen Hua Liu et al. 'Native JSON Datatype Support: Maturing SQL and NoSQL Convergence in Oracle Database'. In: *Proc. VLDB Endow.* 13.12 (Sept. 2020), pp. 3059–3071. ISSN: 2150-8097. DOI: 10.14778/3415478.3415534. URL: https://doi.org/10.14778/3415478.3415534.

[23] Sergey Melnik et al. 'Dremel: A decade of interactive SQL analysis at web scale'. In: *Proceedings of the VLDB Endowment* 13.12 (2020), pp. 3461–3472.

[24] Sergey Melnik et al. 'Dremel: Interactive Analysis of Web-Scale Datasets'. In: 3.1–2 (Sept. 2010), pp. 330–339. ISSN: 2150-8097. DOI: 10.14778/1920841.1920886. URL: https://doi.org/10.14778/1920841.1920886.

[25] Guido Moerkotte. *Building Query Compilers (Under Construction).* Feb. 2023.

[26] *Multivariate Statistics Examples.* URL: https://www.postgresql.org/docs/current/multivariate-statistics-examples.html (visited on 18th July 2023).

[27] *MySQL Doxygen: class Item.* URL: https://dev.mysql.com/doc/dev/mysql-server/latest/classItem.html#details (visited on 20th June 2023).

[28] *MySQL Doxygen: EstimateFieldSelectivity.* URL: https://dev.mysql.com/doc/dev/mysql-server/latest/estimate__selectivity_8cc.html (visited on 24th June 2023).

[29] *MySQL Doxygen: join_optimizer.h.* URL: https://dev.mysql.com/doc/dev/mysql-server/latest/join__optimizer_8h.html (visited on 20th June 2023).

[30] Sander Nicolausson and Jørgen Aasvestad. 'Semi-Automatic Statistics Management: Exploiting Existing Data Streams to Improve Selectivity Estimation in MySQL's Hypergraph Optimizer'. MA thesis. Norwegian University of Science and Technology, 2022.

[31] *Search indexes for JSON.* 23rd Nov. 2021. URL: https://blogs.oracle.com/database/post/search-indexes-for-json (visited on 12th July 2023).

[32] P. Griffiths Selinger et al. 'Access Path Selection in a Relational Database Management System'. In: SIGMOD '79. Boston, Massachusetts: Association for Computing Machinery, 1979, pp. 23–34. DOI: 10.1145/582095.582099. URL: https://doi.org/10.1145/582095.582099.

[33] *Server System Variables*. URL: https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html (visited on 19th June 2023).

[34] *sql/histograms/histogram.cc:2026*. Version commit 711dfe7397d205ae17fc1f77d82f7ec9a23c3dad. URL: https://github.com/mysql/mysql-server/blob/8.0/sql/histograms/histogram.cc#L2026.

[35] *sql/histograms/value_map.h:62*. Version commit 711dfe7397d205ae17fc1f77d82f7ec9a23c3dad. URL: https://github.com/mysql/mysql-server/blob/8.0/sql/histograms/value_map.h#LL62.

[36] *Statistics Used by the Planner*. URL: https://www.postgresql.org/docs/current/planner-stats.html (visited on 18th July 2023).

[37] Daniel Tahara, Thaddeus Diamond and Daniel J Abadi. 'Sinew: a SQL system for multi-structured data'. In: *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 2014, pp. 815–826.

[38] *The JSON Data Type*. URL: https://dev.mysql.com/doc/refman/8.0/en/json.html (visited on 24th June 2023).

[39] *The RecSys Challenge 2014 Dataset*. URL: https://github.com/sidooms/MovieTweetings/tree/master/recsyschallenge2014 (visited on 15th May 2023).

[40] *TPC BENCHMARK H*. TJA1043. Rev. 3.0.1. TPC. Apr. 2022.

[41] Chao Zhang and Jiaheng Lu. 'Selectivity estimation for relation-tree joins'. In: *32nd International Conference on Scientific and Statistical Database Management*. 2020, pp. 1–12.

Appendix

A JSON Documents

```
[
    {"user": {"id": 1, "screen_name": null}},
    {"user": {"id": 2, "screen_name": "harry"}},
    {"user": {"id": 3}},
    {"user": {"id": 4}},
    {"user": {"id": 5, "screen_name": null}},
    {"user": {"id": 6, "screen_name": "ron"}},
    {"user": {"id": 7, "screen_name": "hermoine"}},
    {"user": {"id": 8, "screen_name": null}},
    {"user": {"id": 9, "screen_name": null}},
    {"user": {"id": 10, "screen_name": null}},
]
```

Listing 24: A collection of ten JSON documents, formatted as a JSON array. Each object contains some information about a user. The user ID is always included. The "screen_name" key is optional, and when it is present in the document it may either lead to a string value or it may be null.

B More Code Samples

C Custom Experiment Queries

C.1 Airbnb Queries

C.2 Twitter Queries

D Reproducing Experimental Results

This section contains a detailed account of the workflow for setting up and running the experiments described in sections 5 and 6 so that others may attempt reproduce the results if they so wish.

D.1 Reproducing Python Experiments

The Python version used was `3.11.3`. The Pip modules used were as follows:

```
contourpy==1.0.7
cycler==0.11.0
fonttools==4.39.3
hyperloglog==0.0.14
kiwisolver==1.4.4
```

```cpp
lookup_result estimate_selectivity(const String &path, const longlong cmp_val) const {
  if (auto candidate = find_bucket(path)) {
    const KeyStat *path_stats = *candidate;
    double base_freq = path_stats->frequency * (1.0 - path_stats->null_values);

    // If float and int have the same type identifier (key path type suffix), then it
    // is possible that an integer comparand was used for a key path which actually holds float values
    // In this case, we convert the comparand to float and lookup that value instead:
    if (path_stats->values_type == BucketValuesType::FLOAT) {
      return estimate_selectivity(path, (double) cmp_val);
    }

    // Check if cmp_val out of range of path_stats values
    if (path_stats->min_val && path_stats->max_val) {
      if ((*path_stats->min_val)._int > cmp_val) {
        return lookup_result{0, 0, base_freq};
      }
      if ((*path_stats->max_val)._int < cmp_val) {
        return lookup_result{0, base_freq, 0};
      }
    }

    if (path_stats->histogram) {
      auto histogram = static_cast<KeyPathHistogram<longlong> *>(path_stats->histogram);

      double cumulative = 0;
      if (histogram->buckets_type == JHistogramType::SINGLETON) {
        for (const auto &jg_buck : histogram->m_buckets.single_bucks) {
          if (jg_buck.value == cmp_val) {
            return lookup_result{
              base_freq * jg_buck.frequency,
              cumulative * base_freq,
              (1 - (cumulative + jg_buck.frequency)) * base_freq
            };
          } else if (jg_buck.value > cmp_val) {
            return lookup_result{
              0,
              (cumulative) * base_freq,
              (1 - cumulative) * base_freq
            };
          }
          cumulative += jg_buck.frequency;
        }
      } else {
        for (const auto &jg_buck : histogram->m_buckets.equi_bucks) {
          if (jg_buck.upper_bound >= cmp_val) {
            return lookup_result{
              (base_freq * jg_buck.frequency) / jg_buck.ndv,
              (cumulative) * base_freq,
              (1 - cumulative) * base_freq
            };
          }
          cumulative += jg_buck.frequency;
        }
      }
    }

    if (path_stats->ndv) {
      return lookup_result{base_freq / (*path_stats->ndv), base_freq * 0.3, base_freq * 0.3};
    }

    return lookup_result{base_freq * 0.1, base_freq * 0.3, base_freq * 0.3};
  }

  // If stats can't be found for the path, return global minimum frequency
  return {min_frequency * 0.1, min_frequency * 0.3, min_frequency * 0.3};
}
```

Listing 25: The selectivity estimation function for integers. Note that the functions returns estimate for both equality, less than and greater than to reduce the amount of code required (and duplicated) to support all the supported operands for each allowed type.

```
SELECT listings.j
FROM listings, calendar
WHERE
    -- Join conditions
    listings.j->"$.host_id" = "62180436" -- ? ('144214204' 1003), ('263524662', 87), ('66796928',
    ↪  8), ('62180436', 1)
    AND listings.j->"$.id" = calendar.j->"$.listing_id"
    -- Filters on availability
    AND calendar.j->"$.available" = "t"
    AND calendar.j->"$.date" = "2023-12-12"
    AND calendar.j->"$.minimum_nights" = 1
ORDER BY
    listings.j->"$.last_review"
;
```

Listing 26: Airbnb query `host_availabilities_a`

```
SELECT listings.j
FROM listings, calendar
WHERE
    -- Join conditions
    listings.j->"$.host_id" = "66796928" -- ('144214204' 1003), ('263524662', 87), ('66796928',
    ↪  8), ('62180436', 1)
    AND listings.j->"$.id" = calendar.j->"$.listing_id"
    -- Filters on availability
    AND calendar.j->"$.available" = "t"
    AND calendar.j->"$.date" = "2023-12-12"
    AND calendar.j->"$.minimum_nights" = 1
ORDER BY
    listings.j->"$.last_review"
;
```

Listing 27: Airbnb query `host_availabilities_b`

```
SELECT listings.j
FROM listings, calendar
WHERE
    -- Join conditions
    listings.j->"$.host_id" = "263524662" -- ? ('144214204' 1003), ('263524662', 87), ('66796928',
    ↪  8), ('62180436', 1)
    AND listings.j->"$.id" = calendar.j->"$.listing_id"
    -- Filters on availability
    AND calendar.j->"$.available" = "t"
    AND calendar.j->"$.date" = "2023-12-12"
    AND calendar.j->"$.minimum_nights" = 1
ORDER BY
    listings.j->"$.last_review"
;
```

Listing 28: Airbnb query `host_availabilities_c`

```sql
SELECT listings.j
FROM listings, calendar
WHERE
    -- Join conditions
    listings.j->"$.host_id" = "144214204" -- Host with the most listings in LA, ca. 1000 in total
    AND listings.j->"$.id" = calendar.j->"$.listing_id"
    -- Filters on availability
    AND calendar.j->"$.available" = "t"
    AND calendar.j->"$.date" = "2023-12-12"
    AND calendar.j->"$.minimum_nights" = 1
ORDER BY
    listings.j->"$.last_review"
;
```

Listing 29: Airbnb query `host_availabilities_d`

```sql
SELECT listings.j, reviews.j
FROM listings, calendar, reviews
WHERE
    -- Join conditions
    listings.j->"$.host_id" = "144214204" -- ? ('144214204' 1003), ('263524662', 87), ('66796928',
    ↪  8), ('62180436', 1)
    AND listings.j->"$.id" = calendar.j->"$.listing_id"
    AND listings.j->"$.id" = reviews.j->"$.listing_id"
    -- Filters on availability
    AND calendar.j->"$.available" = "t"
    -- Filter on recent reviews
    AND reviews.j->"$.date" >= "2023-01-01"
;
```

Listing 30: Airbnb query `host_availabilities_d_variant`

```sql
SELECT listings.j
FROM listings, calendar, neighbourhoods
WHERE
    -- Join conditions
    listings.j->"$.id" = calendar.j->"$.listing_id"
    AND
    listings.j->"$.neighbourhood_cleansed" = neighbourhoods.j->"$.neighbourhood"

    -- Listing predicates
    AND listings.j->"$.review_scores_rating" >= 4
    AND listings.j->"$.bedrooms" >= 3
    -- AND listings.j->"$.accommodates" >= 6
    AND listings.j->"$.host_is_superhost" = "t"

    -- Calendar predicates
    AND calendar.j->"$.available" = "t"
    AND calendar.j->"$.date" = "2023-12-12"
    AND calendar.j->"$.minimum_nights" = 1

    -- Neighbourhood predicates
    AND (
        neighbourhoods.j->"$.neighbourhood_group" = "City of Los Angeles"
        OR
        neighbourhoods.j->"$.neighbourhood_group" = "Unincorporated Areas"
    )
;
```

Listing 31: Airbnb query `many_filters`

```
-- Will retrieve the three most recent reviews for the top 10 rated
-- listings available on a certain date

SELECT top_rated_listings.*, recent_reviews.*
FROM (
    SELECT ls.j->"$.id" AS id
    FROM listings ls, calendar c
    WHERE ls.j->"$.id" = c.j->"$.listing_id"
    AND c.j->"$.date" = "2023-11-17" AND c.j->"$.available" = "t"
    ORDER BY ls.j->"$.review_scores_rating" DESC
    LIMIT 10
) AS top_rated_listings
INNER JOIN
(
    SELECT rs.j->"$.listing_id" AS listing_id, row_number() OVER w AS row_n
    FROM reviews rs
    WINDOW w AS (PARTITION BY rs.j->"$.listing_id" ORDER BY rs.j->"$.date" DESC)
) AS recent_reviews
ON recent_reviews.listing_id = top_rated_listings.id
WHERE recent_reviews.row_n <= 3
;
```

Listing 32: Airbnb query `recent_reviews_for_top_listings`

```
matplotlib==3.6.3
munch==2.5.0
numpy==1.24.3
packaging==23.1
pandas==1.5.3
Pillow==9.5.0
psutil==5.9.5
pyfakefs==5.1.0
pyparsing==3.0.9
python-dateutil==2.8.2
pytz==2023.3
six==1.16.0
tqdm==4.65.0
```

Instructions for installing and running the Python experiments can be found in the `mysql-statistics-dev` repository's `README.md` file. The seed used for all experiments was 6617389845874142767.

D.2 Reproducing MySQL Experiments

The MySQL prototype is somewhat unstable and prone to crashing on repeat inserts of histograms for the same column. The experiments work around this issue by avoiding inserting histograms more than once as much as possible. This may help explain some of the design decisions made in the experiment workflow which make little sense otherwise.

Instructions on setting up and running the MySQL experiments can be found in the `tpch` repository which will be handed in together with this report.

```sql
SELECT
    acceptable_listings.listing_info
FROM (
        SELECT
            DISTINCT j->"$.listing_id" AS listing_id
        FROM
            calendar
        WHERE
            j->"$.available" = 't'
            AND
            j->"$.date" IN ("2023-12-12", "2023-12-13", "2023-12-14", "2023-12-15")
            AND
            j->"$.minimum_nights" <= 4
        ORDER BY listing_id
    )
    AS available_listings,
    (
        SELECT
            listings.j AS listing_info,
            listings.j->"$.id" AS listing_id,
            listings.j->"$.review_scores_rating" AS rating
        FROM
            listings,
            neighbourhoods
        WHERE
        -- Join condition
        listings.j->"$.neighbourhood_cleansed" = neighbourhoods.j->"$.neighbourhood"

        -- Listing predicates
        AND listings.j->"$.review_scores_rating" >= 4.5
        AND listings.j->"$.bedrooms" >= 3
        AND listings.j->"$.host_is_superhost" = "t"

        -- Neighbourhood predicates
        AND neighbourhoods.j->"$.neighbourhood_group" = "City of Los Angeles"
    )
    AS acceptable_listings,
    (
        SELECT
            DISTINCT j->"$.listing_id" as listing_id
        FROM
            reviews
        WHERE
            -- Less than ish a year old
            j->"$.date" >= "2022-01-01"
    )
    AS recently_reviewed
WHERE
    acceptable_listings.listing_id = available_listings.listing_id
    AND
    acceptable_listings.listing_id = recently_reviewed.listing_id
ORDER BY
    acceptable_listings.rating
LIMIT 10;
```

Listing 33: Airbnb query `top_ten_listings_with_filters`

```
SELECT
    tweeters.j->"$.user.screen_name",
    tweeters.j->"$.id",
    COUNT(*) AS count
FROM (
    SELECT * FROM twitter
    WHERE
    (JSON_VALUE(j, "$.in_reply_to_status_id") IS NULL OR j->"$.in_reply_to_status_id" IS NULL)
    AND (JSON_VALUE(j, "$.retweeted_status.id") IS NULL OR j->"$.retweeted_status.id" IS NULL)
) AS tweeters, (
    SELECT * FROM twitter
    WHERE JSON_VALUE(j, "$.in_reply_to_status_id") IS NOT NULL
) AS replies, (
    SELECT * FROM twitter
    WHERE JSON_VALUE(j, "$.retweeted_status.id") IS NOT NULL
) AS retweets
WHERE
    tweeters.j->"$.id" = replies.j->"$.in_reply_to_status_id"
    AND
    tweeters.j->"$.id" = retweets.j->"$.retweeted_status.id"
GROUP BY
    tweeters.j->"$.user.screen_name",
    tweeters.j->"$.id"
ORDER BY
    count DESC
LIMIT 20
;
```

Listing 34: Twitter query `count_replies_and_retweets`

```
SELECT
    j->"$.place.country_code",
    j->"$.user.lang",
    COUNT(*) AS count
FROM twitter
WHERE
(
    (j->"$.user.lang" <> "en" AND j->"$.user.lang" <> "en-gb")
    AND j->"$.place.country_code" IN ("GB", "US", "CA", "AU", "NZ", "IE")
) OR
(
    (j->"$.user.lang" = "en" OR j->"$.user.lang" = "en-gb")
    AND j->"$.place.country_code" NOT IN ("GB", "US", "CA", "AU", "NZ", "IE")
)
GROUP BY j->"$.place.country_code", j->"$.user.lang"
ORDER BY j->"$.place.country_code", count DESC;
```

Listing 35: Twitter query `foreign_languages`

```
SELECT unpopular.tweeter_id, unpopular.tweet_id
FROM (
    SELECT * FROM twitter
    WHERE j->"$.user.followers_count" > 5000
) AS popular, (
    SELECT j->"$.id" AS tweet_id, j->"$.user.id" AS tweeter_id
    FROM twitter
    WHERE
        j->"$.user.followers_count" < 100
        AND j->"$.user.verified" = FALSE
) AS unpopular
WHERE popular.j->"$.in_reply_to_status_id" = unpopular.j->"$.id";
```

Listing 36: Twitter query `popular_and_unpopular`

```
-- Most commonly replied to users: (121377261, 7), (23511065, 7), (99624309, 6), (283335575, 4),
↪  (246082550, 4), (230283763, 3), (71667445, 3), (84389890, 3), (454631670, 3), (212299385, 3)

SELECT
    replied_users.user_id AS user_id,
    tweeting_users.username AS username,
    replied_users.reply_count AS reply_count,
    tweeting_users.tweet_count AS tweet_count
FROM
    (
        SELECT
            j->"$.in_reply_to_user_id" AS user_id,
            COUNT(*) AS reply_count
        FROM
            twitter
        WHERE
            JSON_VALUE(j, "$.in_reply_to_user_id") IS NOT NULL
        GROUP BY
            j->"$.in_reply_to_user_id"
    ) AS replied_users
INNER JOIN
    (
        SELECT
            j->"$.user.id" AS user_id,
            j->"$.user.screen_name" AS username,
            COUNT(DISTINCT j->"$.id") AS tweet_count
        FROM
            twitter
        WHERE
            JSON_VALUE(j, "$.user.id") IS NOT NULL
            AND
            JSON_VALUE(j, "$.id") IS NOT NULL
        GROUP BY
            user_id, username
    ) AS tweeting_users
ON
    replied_users.user_id = tweeting_users.user_id
ORDER BY
    reply_count DESC,
    tweet_count DESC
LIMIT 20
;
```

Listing 37: Twitter query `users_with_most_replies`