A Comparison of Algorithms for Self-Tuning Histograms

Julián Hernández, Rodrigo Ipince, José Muñiz

December 13, 2008

Abstract

In this paper, we explore the feasibility of implementing self-tuning histograms for OLTP databases. We describe different algorithm strategies for building and maintaining self-tuning histograms and explore their performance in an environment of frequent updates over databases with large amounts of data where the current model of statistic collection is too costly.

We conclude by showing that self-tuning histograms provide comparable performance to statistic collection with almost no overhead for selectivity estimations over one field.

1 Introduction

Traditional database management systems based on the System-R design often include an optimizer. As explained in [2], this optimizer receives a query plan as input and is responsible for evaluating a family of equivalent query plans and choosing a low-cost equivalent for the input plan. In order to estimate the costs of all the feasible plans, the optimizer requires to predict the sizes of intermediate results within the execution of the query before its actual execution. For example, if a join is required between two tables R and S, the expected sizes of each relation would dictate the join algorithm to be used. In particular, an optimizer requires knowledge of the distribution of values over different fields. When a query, say, of the form `SELECT` $x_1, x_2, ..., x_n$ `FROM T WHERE` $P(x_1, x_2, ..., x_n)$, is entered to the database, the optimizer must approximate the number of tuples in T that satisfy $P(x_1, x_2, ..., x_n)$. We define this expected number of tuples to be the selectivity of P, in a way analogous to [9].

A particular type of selectivity function of interest is the selectivity of predicates of the form $P(x_i) \equiv x_i \in [a, b]$. These are predicates defined over a single field, and express the fraction of tuples in that field that lie within some range of its domain $[a, b]$. These predicates are common in practice and are also used as building blocks for estimating more complicated multivariable predicates. For example, if $P(x, y)$ is $(x \in [a, b]) \wedge (y \in [c, d])$ and we assume fields x and y are not correlated, then we can express $P(x, y)$ as $P(x) \times Q(y)$, where $P(n) \equiv n \in [a, b]$ and $Q(n) \equiv n \in [c, d]$.

A common approach for estimating these types of selectivities is to use a histogram associated with each field on each relation. Estimating the selectivity from a given query is easy and requires almost no overhead when consulting an already existing histogram. However, the process of building a histogram is usually expensive. Histograms get periodically rebuilt form scratch by an independent component of the database system called the statistics gatherer. This component uses statistical sampling and scans several pages of the table to estimate an approximate distribution of values. This process incurs significant I/O costs, which makes the operation expensive. Current commercial database systems like Oracle, Microsoft SQL Server, and Sybase, as well as open source alternatives like PostgreSQL and mySQL, all use this approach to single field selectivity estimation, as discussed in [6]. When facing large-sized tables with constant updates, statistics analysis can only be performed at low frequencies, which implies the estimates degrade in quality over extended periods of time. Situations of large-sized tables with constant updates may be present in certain kinds of OLTP transactions, which may experience degradation in performance due to the imprecision effects of the histogram.

Several authors ([1], [3], [4]) have explored the notion of replacing the independent statistics gatherer with a feedback system by which the results from previous or current queries are fed back to the optimizer for future decision-making. In this paper, we explore some of these works, and propose several modifications for improving the accuracy of these algorithms

under our assumptions of large data sets with high frequency updates. We then show that self-tuning histograms represent a viable option for cost estimation under these conditions, and comment on the differences between the different versions of the histogram building algorithms.

This paper is organized into seven parts. Section 2 describes how histograms are used in database systems. Section 3 shows some previous work on using results from queries to improve future cost estimates. Section 4 then explains our modifications to the previous algorithms. Finally, Sections 5, 6, and 7 describe our framework for benchmarking these algorithms in a transactional environment.

2 Histogram querying and traditional histogram building

2.1 Basic histogram definitions

In this paper, we use a definition of a histogram similar to [7] and [8]. Let V_i be the set of possible values of a field i, and $f(v), v \in V_i$ be a function that specifies the frequency (number of tuples where the ith field equals v) of a given value within a field. Partition the data distribution into disjoint ranges which we call *buckets*. A histogram is then a mapping from each bucket to some frequency, which is computed as some well known function of the frequencies of all values that fall within that bucket.

Two special type of histograms are commonly used in database management systems, and are described by [6].

Equi-height: The frequency of any two buckets is equal. In order to represent a non-uniform distribution of values, this type of histogram carefully chooses the bucket sizes to preserve the equal frequency discipline.

Compact: In addition to storing the distribution of the complete range of values, an additional histogram stores the most common values and their associated frequencies.

Postgres uses a *compact* representation for the most common values of a field, and an *equi-height* histogram to store the results of the remaining values.

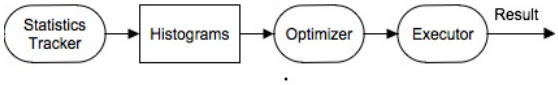

Figure 1: Traditional model for query optimization and execution

2.2 Histograms in the traditional cost model

In the traditional model, the process of query execution contains three phases:

Optimization The optimizer uses the histogram generated by the statistics gatherer in order to estimate the costs of different scans and then predict the costs of other higher order operations.

Execution Given an optimized query plan, this component is responsible for performing the actual I/O operations and producing the resulting relations.

Statistics gatherer An independent component that runs in parallel and is in charge of updating the histogram.

This flow is depicted in Figure 1. We notice that there is no connection between the *traditional-executon* and *statistics collection* subsystems.

The optimizer uses the histogram to compute an estimate for the selectivity of a predicate $P(x_i)$ of the form $x_i \in [a, b]$ as follows:

- If range is of type $[a, a]$, then we search in the most common values table.
  - If it is present, then we can compute its selectivity $\sigma = f(a)$, where f is the frequency function of the most common values histogram.
  - If it is not present, then we can estimate its selectivity as

$$\frac{\sigma}{\sum_k f(k)} = \frac{1 - \sum_{k<a} f(k)}{|b - a| - \sum_{k<a} 1}$$

Here, b and a are the end and start range of the bucket where a is located, and $\sum_{k<a} 1$ means the number of most common values less than a. In other words, whenever a value is not in the most common values list, we first use the list of

most common values to obtain the frequencies of the values less than a. We then estimate the frequency of the remaining unknown values up to a by looking at the *equi-height* histogram and obtaining its frequency. We divide by $\sum_k f(k)$ to obtain an absolute number, and not a proportion.

- If range is of type $[a, b]$, we follow the same approach as the previous example. In other words, we view the total selectivity as the sum of the frequencies of the values in the range in the most common values list and the frequencies from the *equi-depth* table of the remaining range.

As a simple example, consider a sample `Students` table. This table contains an `age` field. Figure 2 shows the histograms for the most common values and then for the remaining values. Now suppose we wish to perform the query `SELECT * FROM WHERE AGE` ≤ 18. The shaded areas represent those regions of the histogram which satisfy the predicate. There are 7 people with a smaller or equal age in the list of most common values. Therefore, we estimate for the two most common ages a total of 7. For the remaining 16 ages, we use the *equi-height* histogram and compute an estimated 14 people. Therefore, the selectivity σ_h according to the *equi-height* histogram is 14 and the selectivity σ_{MCV} according to the most common values histogram is 7. The amount α of people in the range not in the list of common values is $\frac{16}{18} = \frac{8}{9}$

Therefore, the total number of tuples can be estimated as

$$\sigma = \alpha \times \sigma_h + \sigma_{MCV} = \frac{8}{9}(14) + 7 = 19.4$$

It is important to notice that this use of histograms make several key assumptions regarding the data, such as a uniform distribution of values inside of a bucket. In addition, as described in detail in [4], there is an assumption of *query uniformity*, which implies an expectation that queries will refer to all values in a field with equal frequency.

2.3 Statistical generation of histograms

As described in [6], a statistical approach to building histograms consists of obtaining a set of sample tuples from the relations, performing queries and collecting statistics on this sample set. Although very accurate, the process of statistical sampling requires reading a large amount of pages from disk in order to guarantee statistically valid estimations. In Postgres, statistics collections is triggered after some threshold number of insertions, currently 50. There are two important trade-offs to consider regarding statistics collections:

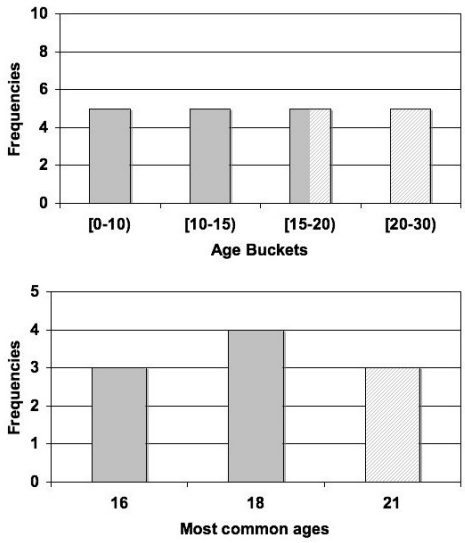

Figure 2: Compact and equi-height histograms for field `Students.age`

Sample size ↔ performance The larger the sample size, the more costly it is to generate a histogram. However, this histogram will be more accurate.

Sampling frequency ↔ adaptability The higher the sampling frequency, the faster the histogram will reflect changes in the distribution of values as they are inserted.

Unfortunately, the cost of running statistic sampling is prohibitive for large databases. Statistics are gathered during low-load periods. For example, histogram refresh could take place during the end of the day. For the remaining part of the day, these statistics are not updated, resulting in a gradual degradation of cost estimates.

3 Incremental histogram building

Instead of analyzing the entire table every time the histogram needs to be rebuilt, we consider algorithms that modify the histogram based on recent insertions on updates.

3.1 Curve fitting

A mechanism for self-tuning column selectivity estimates is presented in [3]. This paper presents an algorithm for sequentially approximating attribute value distribution by a polynomial that minimizes the sum of the squares of the estimation errors. The described procedure does not use online sampling or offline database scans, making it adaptable and showing high accuracy rates.

Although the results from [3] seem promising, they represent a radical shift away from the mechanism described in Section 2. Several pieces of software, including subcomponents of the database system and graphical user interfaces for server management rely on the current architecture. For example, in Postgres, applications may interface with the histogram through a regular system table, `pg_statistic`, which relies on the currently present architecture. Therefore, a radical shift in the design of any major database system seems unlikely when such a change would achieve performance gains for just some family of data and update frequency conditions.

3.2 Self-Tuning Histograms

An extension to the traditional model of optimization depicted in Figure 1 is using the result set sizes computed by the query executor to correct for estimation mistakes in the histogram. The histogram is then enriched with the free information generated by all operators during the execution phase of the query. In turn, this feedback loop ensures that subsequent queries that contain overlapping predicates have access to better estimates in the future. Figure 3 shows the modified execution system. The execution component communicates with a histogram builder in order to receive information about actual cardinalities between ranges of fields. The histogram can now provide range estimates based on the information received over the query execution lifetime.

The histogram and histogram builder expose the following interface:

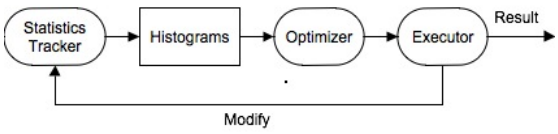

Figure 3: Self-tuning model for query optimization and execution

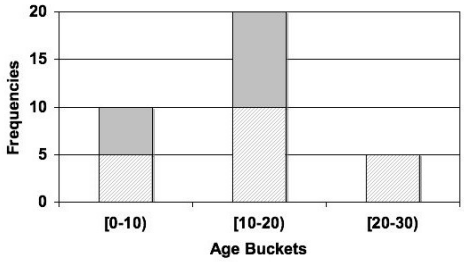

Figure 4: Histogram modification resulting from a call to `receive([1,19], 30)`. The shaded region represents a modification to the original histogram

- `receive([a...b], val)` This function is called by the query executor to inform the histogram builder that there are `val` values between `a` and `b`.
- `int get([a...b]` This function asks the histogram to estimate the number of tuples between `a` and `b`.

In this paper, we disregard the possibility of modifying the query plan while it is being executed, since we expect the number of queries to be so large that it should be possible to see clear performance gains by optimizing just future performance. Further, we are interested not on the performance of a single query but rather the overall throughput of the system.

Aboulnaga and Chaudhuri describe in [1] an algorithm for implementing this interface. In doing so, they consider two factors:

- *Bucket refinement* This means updating frequencies for buckets that are found to contain inaccurate information. Figure 4 shows a sample histogram refinement for a histogram over an age field in a table of people. Originally, this histogram estimates 15 people of age less than 19. However, the `receive` call shows that this frequency should actually be 30. In the example,

the first two buckets completely cover the range defined in the `receive` call, so they're both updated: the first bucket's frequency is increased by 5 and the second by 10. This reflects the fact that error is distributed in such a way that more error is corrected in buckets with higher frequencies. In other words, *blame* is proportional to frequency.

- *Histogram Restructuring* (Merging and splitting buckets) No matter how accurate frequency estimations we obtain for a given range, the assumption of uniform distribution of frequencies along a bucket may create large inaccuracies if a bucket contains wide frequency variations. For example, if our table of people were divided in uniform buckets of size 10 between 0 and 100, and the table were to reflect the people working for a firm, most buckets would be empty, except for maybe those in the range of 20 to 60. We could obtain a more accurate histogram by using a non-uniform bucket distribution in this case.

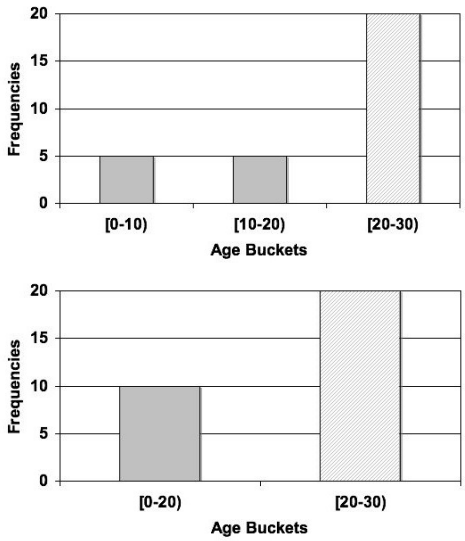

Figure 5: Histogram merging of two buckets with similar frequency into a single one

3.2.1 Bucket refinement

The exact algorithm for bucket refinement assigns blame to both range and frequency. Every time a `receive([a...b], val)` message is passed to the histogram builder, it performs the following operations:

1. Calculate the relative error e (percentage of error with respect to estimated result)
2. For each bucket b_i
   (a) Obtain the fraction p_i of the range `[a...b]` represented by b_i
   (b) Set $f(b_i) = f(b_i)(1 + \alpha \cdot e \cdot p_i)$

Here α is a damping factor, which is meant to characterize the aggressiveness of our modification; a lower value prevents overfitting. The example presented in Figure 4 shows the special case where $\alpha = 1$.

3.2.2 Histogram restructuring

The proposed restructuring method relies on the assumption that splitting high frequency buckets will yield a better histogram. Since the algorithm keeps the number of buckets constant, a separate subroutine is responsible for finding buckets to merge, in order to give the opportunity for buckets to split. The histogram restructuring mechanism is triggered after some threshold T number of histogram updates.

The policy for merging a bucket is to find pairs of contiguous buckets that have frequency differences smaller than a threshold M between each other. We then merge these two buckets into a single one and repeat this operation until no more buckets can be merged. This algorithm will merge runs of buckets with similar frequencies and will result in the merging of a number of buckets, say k.

We decide how we will split each bucket b_i by proportionally assigning splits based on frequency. In other words, the number of buckets after the split for a given bucket b_i will be $\lfloor k \cdot \frac{f(b_i)}{\sum_k f(b_k)} \rfloor$. Figure 5 shows an example of two buckets with similar frequency being merged into a single one.

4 Modifications to proposed algorithms

We implemented a modified version of the algorithm proposed in [1]. In particular, we explore different strategies with respect to three criteria:

Bucket refinement method We explore the following possibilities:

- Assignment of blame with respect to frequency alone.
- Assignment of blame with respect to range alone.
- Hybrid distribution of blame (frequency times range).

Restructuring trigger We decide to restructure based on two different triggers:

- Number of queries.
- Cumulative error (When the amount of error that has been assigned to a given bucket exceeds some threshold).

Split selection method

- Split highest frequency buckets (as in [1]).
- Split buckets based on their cumulative errors. Buckets store the cumulative error that has been blamed on them, and then we split the ones that have accumulated the most error, since they have failed to meet estimates more often.
- Split buckets based on their usage. More frequently used buckets will be split first. This ensures that the buckets the user has been querying the most often are given higher priority and the range estimates are stored in more detail.

5 Benchmark setup

In order to test the accuracy of the previously discussed algorithms for self-tuning histograms, we built a system as shown in Figure 6 on top of Postgres. Postgres was chosen because of three reasons:

1. It is a widely used, open source database system that implements histogram based cost estimation with statistics collection.
2. `EXPLAIN ANALYZE` provides an easy interface for obtaining query plans for a given query.
3. The choice of database is not the main focus of the paper, since the results show the differences between different histogram building algorithms in the presence of large amounts of data and high insertion rates, regardless of platform

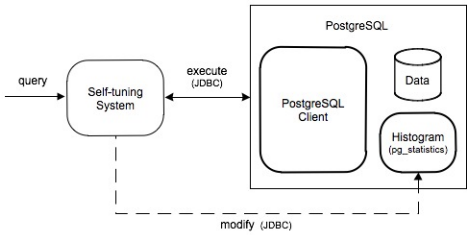

Figure 6: High level overview of the benchmarking system

EmpID	Country
1	3
2	2
3	2
4	1

Table 1: Table for **Employees**

choice. Further, although we do use the statistics analyzer from Postgres, we assume the most popular database systems have highly accurate statistics gatherers.

This system represents an intermediate layer between the end user and Postgres, which intercepts the results from the database to the user and uses the cardinality to build a histogram. The different histograms can then be compared by their accuracy.

Our system is responsible for receiving user queries. The information is then fetched back to Postgres, which responds to the middle system with the query plan to be executed. The benchmark tool then simulates the query plan, by reducing it into a family of nested loop joins and sequential scans with a set of conditions. For each query, a collection of `receive` calls is sent to the histogram builder. For each scan, the executor makes one `receive` call. In the case of a join, for each element in the outer loop, the executor can make a `receive` call as it matches the join field in the outer table with the tuples in the inner table. For example, if Tables 1 and 2 are joined by a natural join, where **Employees** is the outer table, then we get the following three distinct `receive` calls:

```
receive(‘‘Dept.Country’’, [3...3], 2)
receive(‘‘Dept.Country’’, [2...2], 0)
receive(‘‘Dept.Country’’, [1...1], 1)
```

DeptID	Country
1	3
2	3
3	1

Table 2: Table for **Departments**

The system assumes independence between selectivities of multiple conditions, either over a single table or over several tables. In other words, if a query of the form `SELECT ... WHERE` p_1 `and` p_2, we assume that $\sigma_{p_1} \times \sigma_{p_2} = \sigma_{p_1 \wedge p_2}$. For this reason, the join algorithms can be modified to send `receive` messages even when facing multiple conditions, especially when one is a join condition and the other is a filter on the data. Any computation of `receive` messages occurs with almost no overhead, since the data it uses needs to be derived by the query executor anyways.

The middle layer strategy for the benchmark system was chosen over direct modification of Postgres because of the following reasons:

- From the viewpoint of the histogram builder, the messages received from the executor of different join strategies will be equivalent. Therefore, when calculating the change of error rates as a result of these messages, only one type of join needs to be considered.
- When computing error rates of row selectivity estimates, we need not consider the overhead between inter-application connections.
- Since Postgres uses an *equi-depth* histogram, storing the results in Postgres would imply modifying the system catalog slightly to store the additional frequency table. We should note, however, that because of the heuristic with which we merge and split buckets, the algorithm converges to an *equi-depth* histogram. Because of the fast speed of convergence demonstrated in Section 7, this means that storing the histogram without associated frequencies will not significantly impact the accuracy of its estimations. For this reason, a compression scheme may be possible for tables with non-changing distributions in order to eliminate the need of a frequency table. This option was not presently explored.

Finally, it should be noted that we choose to measure our results in terms of row estimate errors, instead of query execution times. This decision was taken for several reasons:

- The `pg_statistic` table in Postgres, which is responsible for holding the histogram information, is not modifiable through an external interface. Although this histogram is stored in a regular table, the schema uses `anyarray` as the type for the histrogram buckets column, which makes it impossible to change except from the Postgres C libraries.
- Although small differences in estimation between the self-tuning system and the conventional Postgres system may be irrelevant, order of magnitude differences are certain to be significant when scanning over data that contains several types of indices and therefore multiple scanning strategies for the optimizer to pick. Similarly, these differences will dictate the strategies for joining and aggregating data.
- Estimate errors, in contrast with query execution times, represent a database independent metric of performance.

6 Choice of Queries and Insertions

We use a framework of insertions in the spirit of the TPC-W benchmark (see [5]), modeling an online transaction business. However, unlike TPC-W, we assume the number of products changes very frequently. This situation could arise in an online system similar to online auction sites like eBay. Our benchmark includes the following tables:

1. `Auctions(auc_id, owner_id, category_id, time_start, time_end, winning_bid)`
2. `Users(uid, name)`
3. `Category(category_id, description)`

The `Auctions` relation stores all the items on sale posted by any user of the system. It contains information about its owner, start and end times for the auction, as well as the current winning bid. The `Users` relation contains the list of all users of the systems, acting as buyers, sellers or perhaps both. Finally, the `Category` relation stores the types of products being auctioned. A possible category could be a particular brand of laptop, which could appear several times in

the `Auctions` if it is being sold by several users at the same time.

We use similar transactions as defined in the TPC-W specification, disregarding those that involve lookups on primary keys. The lookups on primary keys can readily be estimated to have a result set size of at most one, and therefore would not yield any interesting results. For this reason, we only consider the following four transactions:

Transaction 1 **:** *Insert auction.* Inserts a new auction. The product and customer of this inserted auction is uniformly drawn from the table of customers and the table of products. The start date is the insertion time, while the end date is a normally distributed random variable with mean of one day after the insertion time and standard deviation of a third of a day. This transaction simulates a user creating a new online auction, and is therefore classified as a frequent write transaction.

Transaction 2 **:** *Expired auctions query.* Selects the auctions that have expired at the point the transaction runs. This transaction simulates an ETL process that moves old information from this table into a data warehouse.

Transaction 3 **:** *Current auctions query.* Selects the auctions that are set to expire within one day of the time that transaction runs. This transaction simulates a likely query executed by a user who wishes to know information about current auctions that are close to expiring.

Transaction 4 **:** *Product query.* Counts the number of auctions that correspond to a product that is drawn at random from the list of all products. This transaction simulates a query executed for a particular product in the system.

We simulate the execution of this database over a day. The tables are pre-populated with 10,000 products, 5,000 users, and 100,000 records in the auctions table. The transactions are executed as follows:

- Every second, *Insert auction* is executed.
- Every thirty seconds, we additionally execute *Product query.*
- Every minute, all remaining transactions are executed.

We disable the statistics analyzer from Postgres during the execution of the benchmark, but run a full `VACUUM` and `ANALYZE` just prior to it. This condition simulates a daily statistics collection process under conditions where the statistics collection process is too costly to run frequently. In order to benchmark the results, we make the following comparisons:

- We execute queries and insertions on the benchmarking system described in Section 5. The benchmarking system is run once for each of the different algorithm types to be tested. As described in Section 5, this subsystem will automatically generate the `receive` messages to update the histograms. We measure the error rate of a given query as the average error on selectivity estimation defined over a single field.
- We perform the same queries and insertions (in the same order) directly into Postgres. Using `EXPLAIN ANALYZE`, we define the average difference between actual and real row count over scans. These two definitions of error are equivalent since the query plans on both systems take the same form, and therefore would send the same `receive` and `get` messages to a histogram builder.

We ran the benchmark using a different version of bucket refinement, restructuring trigger, and split selection. We assume that the performance of each of the modifications is independent, so it is not necessary to run all possible combinations of refinements.

7 Results

For brevity, we present the most significant results, related to the split selection mechanisms. The results from the bucket refinement confirm the hypothesis from [1] that there's close to a twofold accuracy gain by using both frequency and range distributed blame instead of range or frequency alone. The restructuring trigger modification yielded no significant performance differences. We now present the results from benchmarking the self-tuning algorithm with three different split selection algorithms against the static histogram built by the statistics gatherer in Postgres. These split selection algorithms were defined in Section 4.

The figures labeled *Number of rows* represent the actual average cardinality of the tables versus the time of the day when the query was executed. These compare Postgres's own row count estimation against the three benchmarked algorithms for self-tuning histograms and the real row count. The *Absolute error* graphs show the difference between the estimated and actual row count as a function of the time of the day

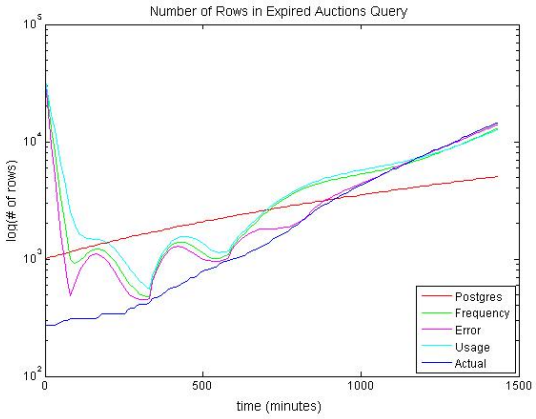

Figure 7: Number of rows for *Expired auctions* query

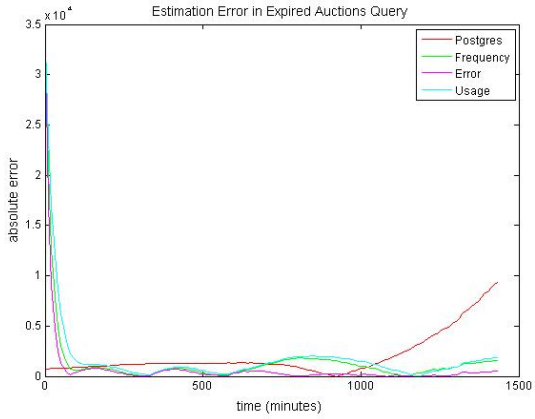

Figure 8: Absolute error for *Expired auctions* query

at which they were executed. The same lines as in the row count are present.

7.1 Expired auctions query

Figure 7 shows the row count for the expired auctions. The cardinality of the result set of this query should be increasing with time, since existing auctions will expire as time goes on. We notice that the self-tuning histogram estimates follow the actual values closely. The estimates are as close as 50 tuples apart in less than 400 queries. However, the estimates oscillate before they finally stabilize. We attribute the oscillation to under-damping in the histogram bucket frequency refinement phase.

The initial estimates for all three self-tuning histograms are inaccurate. This happens because the histograms have not yet received any feedback from the queries, so the histogram is initialized with all buckets of the same size and equal frequencies.

Figure 8 plots the absolute estimation error obtained for the same query. Notice that the error of Postgres is relatively low for the first two thirds of the day, at which point it starts increasing rapidly. This behavior is expected since Postgres has information about the tuples that were already in the `Auctions` table at $t = 0$. The tuples that are added after $t = 0$ do not affect the query until they start expiring, and this only starts happening towards the end of the day because of the way we constructed the expiration dates for the auctions (normally distributed at 1 day with standard deviation of $\frac{1}{3}$ of a day).

Notice that this query constantly consults a very large number of buckets at a given time. For this reason, the *usage split* algorithm may be outperformed by the other self-tuning algorithm variations, since most buckets are being queried and thus are equally likely to split. We conjecture that the *error split* algorithm outperforms the *frequency split* algorithm whenever there are enough queries to provide significant feedback to the histogram about the error distributions in the histogram. This is because the *frequency split* heuristic is only based on the idea that higher frequencies are related to higher error, whereas the *error split* heuristic is based on actual error data. It should be noted, however, that all three algorithms converge quickly to the desired result.

7.2 Current auctions query

Figures 9 and 10 show the graphs outlining the results of the *Current auctions* query. We can notice a similar pattern of quick convergence to a low error and a gradual decay of Postgres's estimation accuracy. In this case, however, the insertion of new elements has an immediate adverse effect on the accuracy of Postgres's selectivity estimation. This is a result of the newly introduced auctions, many of which satisfy the predicate for this query.

Notice that this time the queries focus on a much smaller range, and then do not go back to older ranges again. For this reason, the *usage split* algorithm can probably exploit this fact to expand the buckets that are being currently used in order to provide better estimates. The *frequency split* algorithm still outperforms the *error split* algorithm in this situation.

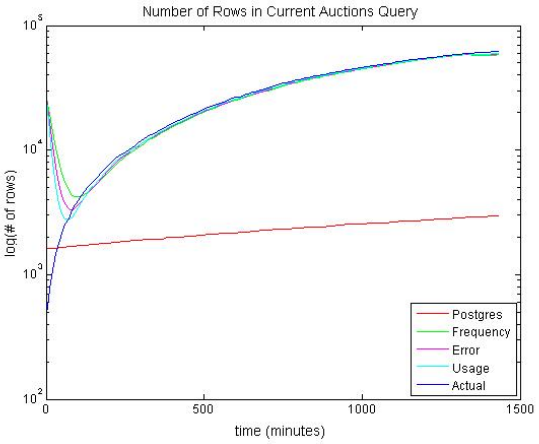

Figure 9: Number of rows for *Current auctions* query

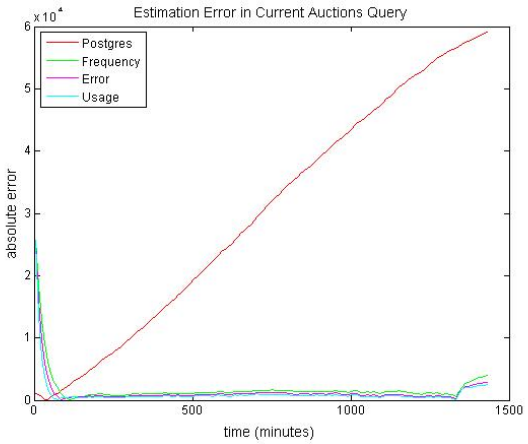

Figure 10: Absolute error for *Current auctions* query

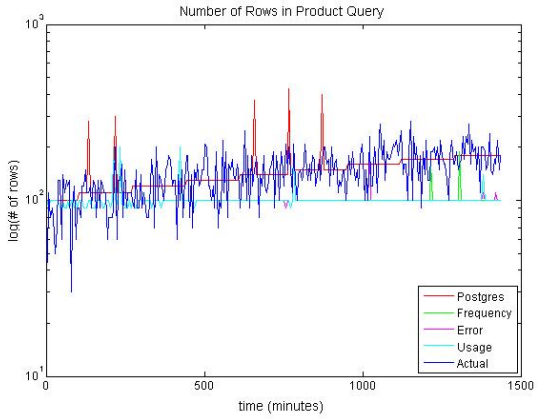

Figure 11: Number of rows for *Product* query

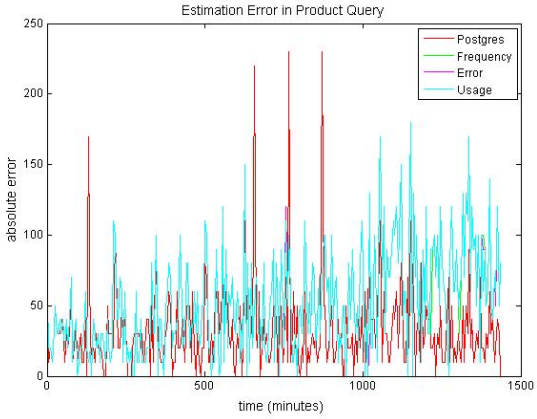

Figure 12: Absolute error for *Product* query

7.3 Product query

The last pair of graphs, shown in Figures 11 and 12, depict the performance on the *Product query*. The three self-tuning histograms estimate the same number of tuples (around 10) consistently. In this case, refinement does not work properly because the errors are so small (around 5) and the per-bucket frequencies are so large (around 1000 or 500, depending on whether we use 10 or 20 buckets, respectively), that bucket refinement does not have a significant impact during estimation. Moreover, since the histogram computed the previous day contains all the products, and the distribution of products over the auctions table does not change over time, the histogram collected by Postgres is expected to continue

being a good estimate.

8 Conclusion

In this paper, we discussed a very efficient family of algorithms for computing cardinality estimates over a single field. We showed that self-tuning histograms may be useful for high frequency update data stores that need to hold large amounts of data and thus make a regular statistics collection system prohibitively expensive. These high frequency updates must also change the distribution of values over time at a speed much larger than the histogram update frequency. Since building histograms based on query feedback modifies a constant-sized histogram with a relatively small amount of buckets, this means that we may update the histogram very frequently without worrying about performance hits.

We also concluded that a histogram building discipline that distributes blame based on both frequency and range is vastly superior to one that only considers range or frequency when deciding how to update frequencies. In a similar manner, there appears to be a slight accuracy improvement by splitting buckets that are more used whenever the number of used buckets is small with respect to the histogram. Further, there is a slight improvement to splitting buckets when the bucket frequency is high by splitting it when the bucket has been corrected very often. This may be a result of using actual feedback data as a means to determine which buckets to split, instead of the simple assumption that higher frequencies may correspond to higher error rates in the histogram.

It should be noted, however, that self-tuning based systems that operate on unchanging range distributions or in read-mostly environments may be outperformed by a regular statistics collection mechanism, since the amount of statistics that require to be collected would be very small with respect to the number of queries. Furthermeore, in most of this report, we have disregarded the effect of errors during estimations of join selectivities. Although self-tuning histograms could provide a good framework for developing multi-dimensional histograms, this option was not explored here. Further, join selectivity errors could well prove to be the driving bottleneck for performance optimization in real systems. A final remark is that the use of self-tuning histogram does not ease the impossibility of using histograms in certain class of one field predicates, such as those using `LIKE`. These queries still require to use alternative estimation mechanisms.

References

[1] Aboulnaga, A., and Chaudhuri, S. Self-tuning histograms: Building histograms without looking at data. pp. 181–192.

[2] Astrahan, M. M., Blasgen, M. W., Chamberlin, D. D., Eswaran, K. P., Gray, J., Griffiths, P. P., III, W. F. K., Lorie, R. A., McJones, P. R., Mehl, J. W., Putzolu, G. R., Traiger, I. L., Wade, B. W., and Watson, V. System R: Relational approach to database management. *ACM Trans. Database Syst. 1*, 2 (1976), 97–137.

[3] Chen, C. M., and Roussopoulos, N. Adaptive selectivity estimation using query feedback. *SIGMOD Rec. 23*, 2 (1994), 161–172.

[4] Christodoulakis, S. Implications of certain assumptions in database performance evauation. *ACM Trans. Database Syst. 9*, 2 (1984), 163–186.

[5] García, D. F., and García, J. TPC-W e-commerce benchmark evaluation. *Computer 36*, 2 (2003), 42–48.

[6] Gibbons, P. B., Matias, Y., and Poosala, V. Fast incremental maintenance of approximate histograms. pp. 466–475.

[7] Ioannidis, Y. The history of histograms (abridged). In *VLDB '2003: Proceedings of the 29th international conference on Very large data bases* (2003), VLDB Endowment, pp. 19–30.

[8] Ioannidis, Y., and Poosala, V. Histogram-based solutions to diverse database estimation problems. *Data Engineering Bulletin 18* (1995), 10–18.

[9] Selinger, P. G., Astrahan, M. M., Chamberlin, D. D., Lorie, R. A., and Price, T. G. Access path selection in a relational database management system. 141–152.