Evaluating Multidimensional Histograms in ProstgreSQL

Dougal Sutherland
Swarthmore College
500 College Ave
Swarthmore, PA
dsuther1@swarthmore.edu

Ryan Carlson
Swarthmore College
500 College Ave
Swarthmore, PA
rcarlso1@swarthmore.edu

Abstract

Query optimization depends heavily on estimating the fraction of rows returned by a query, a process known as selectivity estimation. When queries include multiple predicates, most current database systems assume that the predicates are mutually independent: selectivities for each predicate are evaluated via histograms and multiplied together. When attributes are not actually independent, selectivity estimates are often much smaller than they should be, and non-optimal access paths may be chosen. This can result in significant slowdowns.

Multidimensional histograms (Muralikrishna and DeWitt, 1988) solve this problem by accounting for the correlation present between columns in the database. If a query contains predicates that are stored in the multidimensional histogram, we can use it to estimate selectivity much more accurately.

We have implemented a two-dimensional histogram in PostgreSQL. We evaluate the effectiveness of our histogram on correlated data, real and contrived, and conclude that it is a worthwhile addition to the code base. Moreover, by implementing this simpler two dimensional histogram, we show that it is feasible and advantageous to implement a generalized multidimensional histogram in PostgreSQL.

1 Introduction

A key component of a database management system (DBMS) is the query optimizer, which, given a query, chooses the access path with the minimum expected cost. These costs are computed in large part based on the *selectivity* of its predicates, that is, the fraction of rows that satisfy those predicates.

In most databases, selectivity estimation relies on a precomputed histogram for each column. These histograms are generally *equi-depth*, meaning that each bin of the histogram contains about an equal number of elements of the data; such histograms were shown by Shapiro and Connell (1984) to be more accurate than traditional *equi-width* histograms for database usages.

To estimate selectivities of compound predicates (i.e. those joined with a SQL `AND`), most database systems employ the *attribute-value independence assumption*. This assumption states that the attributes of a given relation are mutually independent. In this case, the joint distribution of the data can be expressed simply as the product of the marginal distribution for each attribute. The selectivity of a query can therefore be computed by multiplying the selectivity of each of its predicates, calculated through single-dimensional histograms.

Unfortunately, however, most real data is not independent. In a typical personnel table, salary might be correlated with age; in a table of vehicle data, the make and the model of a car are highly dependent.

To see why this can be problematic, suppose that we have two queries: P selects cars with `model='Camry'`, while Q selects cars with `model='Camry' AND make='Toyota'`. P and Q will clearly have identical result sets, and thus their selectivities should be equal. When calculating the selectivity of Q under the assumption of independence, however, the selectivities of the `make` and of the `model` are multiplied; the `make` selectivity is "double-counted." Thus, systems that assume attribute independence can greatly underestimate the true selectivity.

Selectivities are, as mentioned, a key component in the query optimizer's choice among access paths. Suppose the database has indexes on `make` and `model`. Broadly speaking, the access path options

are a sequential scan through the entire relation, or a heap scan using the indexes. If the selectivity is high enough, a sequential scan avoids excessive random I/O through the data. Since Q's selectivity has been underestimated, however, the DBMS may incorrectly choose an index scan, which could lead to a significant performance hit as the disk seeks back and forth.

It is therefore desirable for the database to be aware of the data's joint distribution, at least in some cases. The histograms described thus far can in fact be extended to do so in a natural way (Muralikrishna and DeWitt, 1988). A *multidimensional histogram* can be constructed by creating a single-dimensional histogram on the first attribute, then a conditional histogram for the second attribute corresponding to each bin from the first dimension, and so on. (The process is described in more detail in Section 2.1.) We can then walk down the chain of histograms to compute a selectivity estimate in the query optimization process (as described in Section 2.2).

We present an implementation of multidimensional histograms, albeit restricted to the two-dimensional case, in the PostgreSQL database system (Stonebraker et al., 1990). We choose to do so for two reasons. First, multidimensional histograms improve performance on queries involving correlated data. Provided that we have a histogram for the columns in the query, we can calculate selectivities much more accurately and thus choose better access paths. Second, this implementation demonstrates that it is reasonable to expect that multidimensional histograms could be incorporated into an existing DBMS without major changes to the codebase. We thus encourage developers to actively pursue high-quality implementations for their database systems.

We first describe the details of the multidimensional histogram implementation within PostgreSQL (Section 2) and present an experimental validation (Section 3). We then discuss related research (Section 4) and future directions for the work (Section 5).

2 Histogram Implementation

We modify two components of the PostgreSQL system, as discussed below. The standard statistics collection process[1] must be modified to create and store the multidimensional histograms. The query optimizer must also detect when a query contains predicates on both of the columns of a multidimensional histogram, and use that histogram for selectivity estimation.

Our implementation of a two dimensional histogram is a proof of concept for the wider PostgreSQL community. Broadly speaking, the method we use is generalizable to arbitrary dimension. Some aspects of the system, however, limit its generalizability, as discussed in Section 2.3.

2.1 Creating the Histogram

The standard PostgreSQL system constructs a single dimensional histogram as follows:

(*1*) take a sample of the values in the column;
(*2*) sort the sample;
(*3*) create a list of most common values and track their frequencies separately (so they are not entered into the histogram);
(*4*) calculate the number of histogram bins, which will be user-specified except that the system does not create more bins than there are distinct values in the data;
(*5*) find histogram boundaries by reading the values at evenly-spaced positions in the sorted list.

The DBMS stores histograms in a *statistics catalog*, which can contain up to a specified number of entries of arbitrary statistical information about a column's attributes. This catalog is visible throughout PostgreSQL; the query optimizer examines the histograms stored here when it makes selectivity estimates.

Our modified PostgreSQL employs the multidimensional histogram construction algorithm of Muralikrishna and DeWitt (1988). When defined, the attributes of a multidimensional histogram must be ordered. We first collect and sort a sample of the data according to the histogram's first column, just as in the standard model.[2] Since we are primarily interested in the effect of the multidimensional histogram, we simplify the implementation by avoiding

[1] Users initialize this process with the `ANALYZE` command.

[2] Postgres collects a sample of size 300 times the desired number of histogram bins, following Chaudhuri et al. (1998). We currently use an arbitrary sample size of 300,000 for multidimensional data; see Section 3.2.

most-common-values lists. Thus, the next step is to determine the number of histogram bins for the first attribute and find the histogram boundaries. As before, this is stored in one of the slots in the statistics catalog.

To calculate the second level of the histogram (which we call a set of conditional histograms), we must consider the first level histogram's bucket bounds. For each bucket, we now sort the values in that bucket according to the second attribute. Once they are sorted, we compute the conditional histogram bounds and store them in another slot in the catalog. This now allows us to identify how many tuples match the first predicate, and then how many of those match the second predicate.

Muralikrishna and DeWitt (1988) find that the cost of constructing a D-dimensional histogram with b buckets per level on a relation of size Z is approximately a constant times

$$ZD\left[\log Z - \frac{D-1}{2}\log b\right]. \qquad (1)$$

The cost of constructing a two-dimensional histogram on a relation, then, is somewhat less than twice the cost of a one-dimensional histogram on that same relation. We empirically examine the time required for histogram creation in Section 3.2.

2.2 Using the Histogram

Our system currently supports only queries of a very specific form. If we have a multidimensional histogram on attributes a and b in table `t`, the system can employ the histogram on queries of the form

```
SELECT * FROM t
       WHERE a > m AND b < n;
```

where m and n are constant values. Note that any inequality operator will work in either clause. Queries with more than two clauses are not currently handled by our implementation. Equality operations are also unsupported.

For queries of the form processed by our system, the first step in selectivity estimation is to find the multidimensional histogram on this pair of variables. When the histogram is generated, it is assigned a unique identifier, determined based on the attributes involved. This is stored along with the histogram. Now that we are trying to retrieve the histogram, we recompute the unique identifier and load the multidimensional histogram from the catalog.

To identify which values satisfy the $a > m$ clause, the system finds (via binary search) the position of m in the first-level histogram, which contains N bins. Let P_i be the bin selectivity of $a > m$, so that $P_i = \frac{1}{N}$ if the entire bin satisfies $a > m$. For each bin with nonzero P_i, the system then runs binary search through the conditional histogram for b to identify those bins that also satisfy $b < n$. Let Q_{ij} be the selectivity within that conditional bin, defined analogously to P_i. The total selectivity is thus a sum over all of the bins:

$$\text{selectivity} = \sum_i \left[P_i \cdot \sum_j Q_{ij}\right]. \qquad (2)$$

2.3 Challenges of Generalization

Though we attempted to make our multidimensional histogram implementation generalizable to n dimensions, we made some choices that assume two-dimensionality. The storage of the conditional histograms is the biggest concern.

As currently implemented, two-dimensional histograms use two of the statistics arrays in the catalog. The entire multidimensional histogram cannot be stored in a single array, because PostgreSQL requires that each element of a statistics array be of the same type, while our histogram must support attributes of differing types. The number of available statistics arrays is a constant (defaulting to 4), specified in the underlying code. Thus it can be adjusted, but once the code has been compiled and the statistics arrays constructed, new histograms of higher dimensionality could not be created. We do not know of a workable solution at this time outside of fundamental alterations to the form of the statistics catalog.

The histogram creation code would also need to be somewhat more complex in the n-dimensional case. In particular, we are not aware of a clean way to deal with arrays of arbitrary dimensionality in C.

3 Experiments

We conducted experiments on two datasets, one contrived and one from the real world. We have three

id	value	value2	firstname	lastname
1	0.043	0.086	Maria	Johnson
2	9.427	18.854	James	Davis
3	8.909	17.818	James	Jones
4	8.695	17.390	William	Miller
5	9.563	19.126	William	Williams
...	...	...	...	...

Table 1: A small sample of `Double`. Note that `value2 = 2 * value`.

basic tests for each dataset. First, we want to know how much longer statistics generation takes when it must also construct multidimensional histograms. Second, for each query, we compare the selectivity estimations generated from the multi- and single-dimensional histograms, which helps determine the chosen access paths. Finally, we look at total query runtimes when using the standard histogram system and when using our modified histograms.

All experiments were conducted on a machine with a 2.8 GHz Quad-Core AMD Opteron processor and 16 GB of memory. PostgreSQL configuration options were left at their default values except when otherwise noted.

3.1 Datasets

We first discuss experiments run on synthetic data. This data, which we call `Double`, is a single relation which consists of a sequential `id`, a `value` derived from a uniform distribution between 0 and 10, a `value2` field that is double the first field, and `firstname` and `lastname` fields generated independently from a list of 64 total names. A small sample of `Double` is shown in Table 1. Note that there is a perfect correlation between `value` and `value2`. This allows us to run precise tests that combine the attributes but still return identical results. In our tests, the relation contained ten million rows.

We also experimented with U.S. census data from the 2009 Public-Use Microdata Sample (U.S. Census Bureau, 2009), containing data on 1.27 million individuals. The raw data contains over a hundred attributes, ranging from age and ethnicity to how long an individual has lived in their current home. We select only two attributes for our relation, which we call `Census`: income and schooling. Income is measured in dollars. Schooling is coded in values from 1 to 24: 1 represents no school, 2 through 15 represent various levels of grade school, and 16 through 24 represent different levels of post-secondary education. A graphical summary of the portion of the data with incomes less than $200,000 is shown in Figure 1; note the correlation between the attributes.

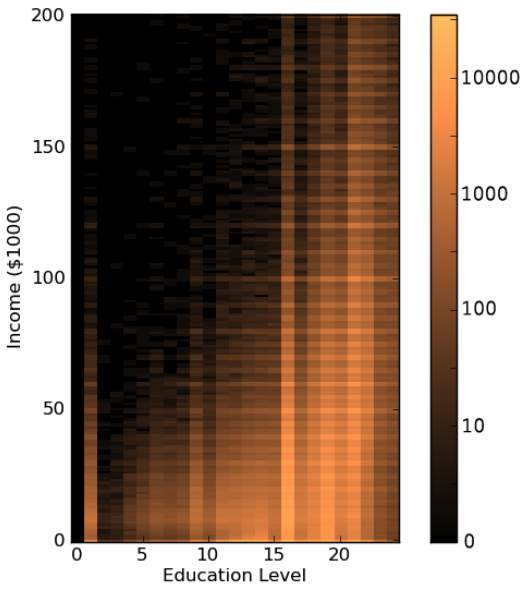


Figure 1: The data distribution of our `Census` dataset; the color of each point represents the number of tuples with education level given by the horizontal position and income given by the vertical position. Less than two percent of the tuples have incomes above $200,000.

3.2 Statistics Generation

We measured the running time of the statistics generation process on relations of various sizes with and without multidimensional histogram creation. Figure 2 shows the time taken to build statistics on a table with and without multidimensional histograms.

Note that the time needed for both types of histogram creation more or less levels off after a certain point: 30,000 for the single-dimensional case, and 300,000 for the two-dimensional case. This is because PostgreSQL creates histograms based on a sample of the data, as described by Chaudhuri

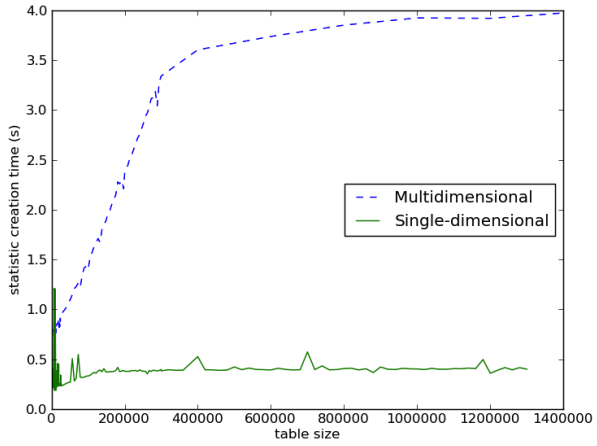


Figure 2: Time taken to create histograms on subsets of the `Double` dataset of various sizes. Times stay essentially the same for larger sizes than shown, because the histograms are generated based on samples of the data.

et al. (1998). For a histogram with the default 100 bins, the sample is of size 30,000. For two-dimensional histograms, we arbitrarily use a sample of size 300,000. Once the relation is larger than the sample size, then, the only difference in histogram creation is that I/O must be done across a larger region of the disk and so will be slightly slower.

For values less than 30,000 – that is, those values for which the histograms are actually being created on relations of the same size – a simple analysis finds that Equation 1 underestimates the rate at which construction times increase. Further analysis would be needed to determine why this is so, or the actual rate at which it does increase.

The total time needed for histogram construction can be effectively limited by the sample size for which they are constructed. Further research into the trade-off between accuracy of representation and histogram construction time would be helpful for a real-world implementation. Histograms need be reconstructed only when the underlying data distribution changes, however, and are made without downtime or significant performance penalty to the system. The cost therefore seems minimal.

3.3 Selectivity Estimates

One measure of success comes directly from the selectivities generated using multidimensional and single-dimensional histograms. We can compare the error rates of each to see how much our system improves over the stock selectivity estimator. We ran range queries on the `Double` dataset of the form

```
SELECT * FROM double WHERE
    value > x AND value2 > y;
```

where x and y range over all values of `value` and `value2`.

Figure 3a shows the selectivity error over each combination of x and y for the single dimensional histogram. Note that most values for Figure 3a are in the five to ten percent range, with some as high as 25 percent.[3]

Figure 3b presents the equivalent measure using our multidimensional estimator. Here all values are between zero and three percent. Our estimates are therefore much more accurate when using multidimensional histograms.

Selectivity errors for the `Census` dataset are shown in Figures 4a and 4b. Although there is some error visible for the multidimensional system, it is far smaller and far more localized than for the independence-assuming system.

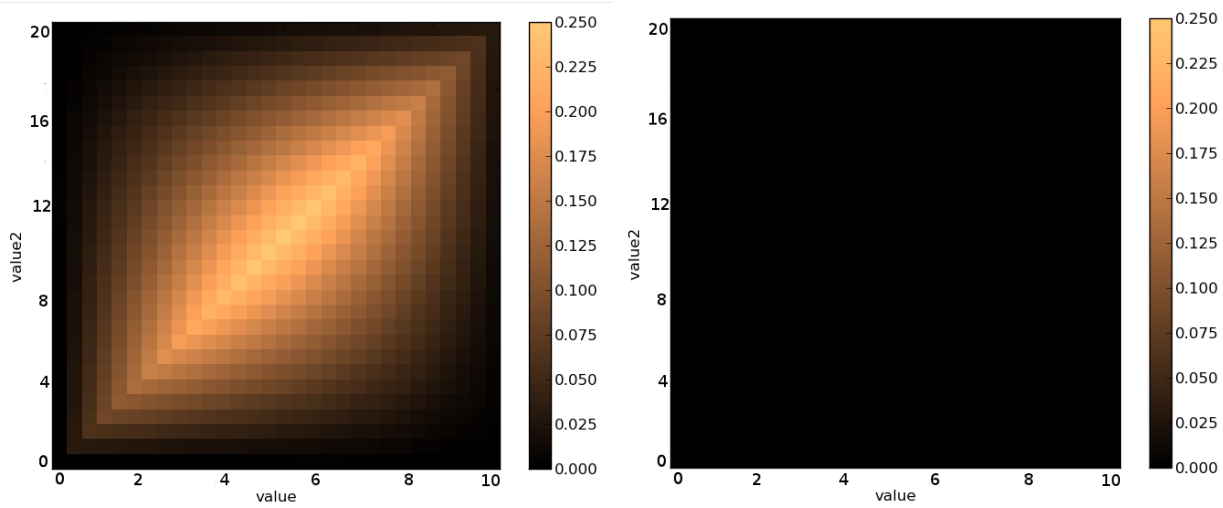
3.4 Access Paths

As discussed in Section 1, selectivity estimates are used to choose between access paths. We ran the same set of queries on `Double` as before on both the single- and multidimensional histograms. These results are shown in Figure 5. The multidimensional system chooses a sequential scan for more values of x and y than its single-dimensional counterpart since it is estimating selectivity more accurately.

In the gray region of Figure 5 where the two methods disagree on the proper access path, the sequential scan favored by the multidimensional system is indeed superior. The query `value > 7 AND value2 > 14`, for example, runs in an average of 2.3 seconds with a sequential scan and 4.3 seconds with an index scan.

Interestingly, despite the more accurate selectivity estimation achieved by the multidimensional histogram, initial tests showed both systems using the

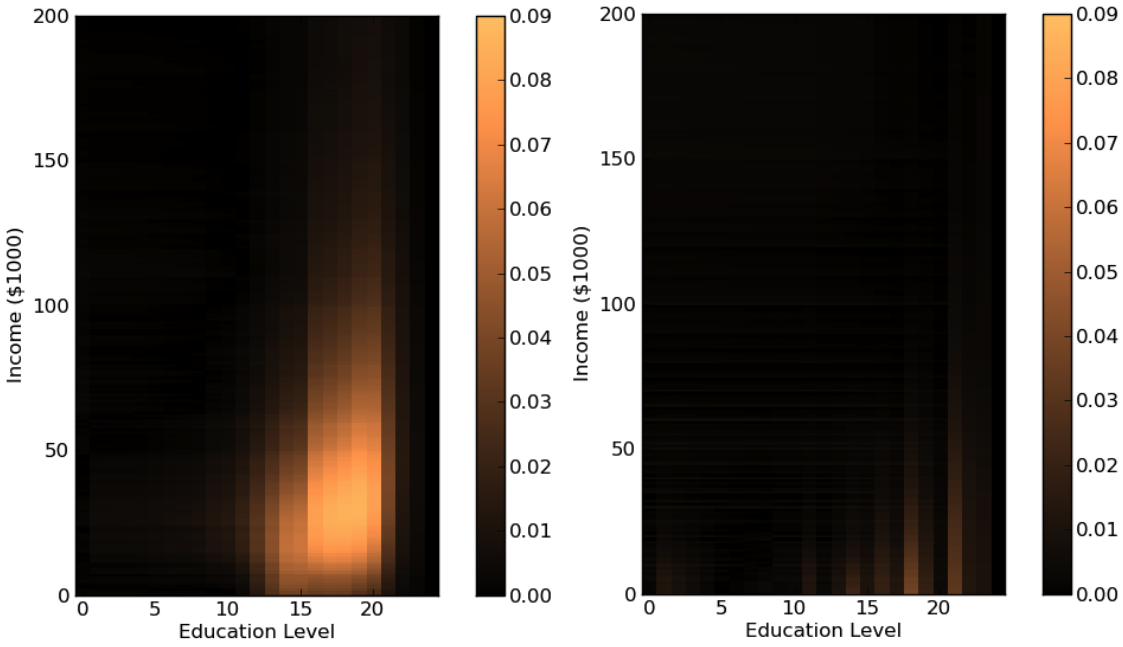
[3]The error pattern shown agrees nearly exactly with the predicted theoretical error, $[\min(x, y) - xy]$ for x and y between 0 and 1.



(a) Single-dimensional selectivity error

(b) Multidimensional selectivity error

Figure 3: Heat map representation of selectivity errors for a set of queries on the `Double` dataset. The horizontal position within the image corresponds to the value of x in the query run, and the vertical position the value of y; the color represents the selectivity error. As shown in the color key at right, a black data point represents zero error in the selectivity, whereas a white value represents a 25 percent error, so that the database's prediction of the number of rows returned differed from the true value by one-fourth the size of the database.



(a) Single-dimensional selectivity error

(b) Multidimensional selectivity error

Figure 4: Heat map representation of selectivity errors for different queries on the `Census` dataset.

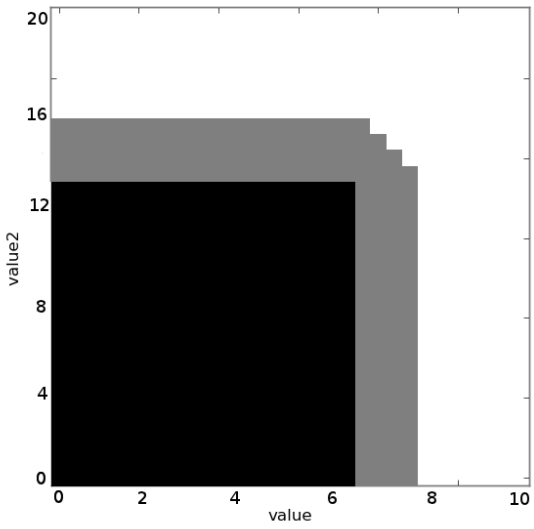


Figure 5: Heat map describing how access paths change as we change queries. Black: both methods use sequential scan. Gray: the multidimensional system uses sequential scan, the single-dimensional system uses an index. White: both methods use heap scan on index.

same access paths for a given query. Since our more complex query involves an additional comparison per returned result, the cost of comparison overpowers the difference in selectivity. It was necessary to tweak a single parameter in the DBMS frontend to see an impact in access paths. This parameter represents the time taken by the CPU to process a comparison relative to the time taken for I/O. The parameter was last set ten years ago, and since CPU speeds have increased much faster than disk access speeds, we find it reasonable to lower the value. We arbitrarily set the value to zero, effectively discounting the comparison time entirely.

4 Related Work

Muralikrishna and DeWitt (1988) describe the use of equi-width multidimensional histograms for selectivity estimation in relational databases. Our implementation is roughly based on this description, and we use the same construction algorithm.

Poosala and Ioannidis (1997) argue, however, that both the simple equi-depth model and the construction model presented here are not ideal. They present experimental results demonstrating that histograms with bins chosen so as to minimize the variance between bins and those with bins chosen to maximize the difference in values between bins can yield more accurate estimates. They also develop a construction algorithm, called *MHIST*, which takes the "most important" split in any dimension at each step.

One other issue with multidimensional histograms is that although it is generally reasonable to keep histograms on all single attributes of a table, it is generally not reasonable to keep joint histograms on all combinations of attributes, as there are an exponential number of such histograms. Deshpande et al. (2001) address this problem through the framework of statistical dependency models, developing a system that can decide which attributes are related and which are not. Markl et al. (2003) take an alternative approach from a learning perspective.

Bruno and Chaudhuri (2002) extend the multidimensional histogram paradigm to cross-table statistics. Joins are among the most expensive operations regularly conducted by a standard relational DBMS, and statistics on join attributes may be able to dramatically improve the ability of the optimizer to choose appropriate join paths.

Another problem occurs when complex queries address more than one multidimensional histogram. Consider a query on attributes A, B, C, D, and E, where the database has information on the joint distribution of A B and C as well as on C and D. How can it combine that into a single selectivity estimate – moreover, one that will be consistent with the results for the next query? What if it also had joint information for B and E? Markl et al. (2007) solve this problem through the principle of maximum entropy, which incorporates all of the available information and makes as few further assumptions as possible.

5 Future Work

This work can be extended in a number of interesting directions, some of them of a larger magnitude than others. We discussed work in Section 2.3 towards generalizing our system to higher dimensions and across tables. In this section we focus on additions to the existing two-dimensional histogram.

During query processing, we currently only recognize very simple queries for which the multidimensional histogram can be used. Consider the case where we have a query on three variables A, B, C where the two-dimensional histogram applies to the pair A, B. We could certainly benefit from using the histogram on A, B and assuming independence with C, but at the moment we cannot process that query. More generally, if we have multidimensional histograms on A, B and A, C, we want to pick the histogram whose pair (or k-tuple) shows the most correlation. We could also use the maximum entropy work by Markl et al. (2007) described in Section 4 to solve this problem.

Because we sample along two (or more generally n) dimensions when we generate our histograms, an interesting question is how many elements must be sampled. Currently, the sample size is approximately $300k$, where k is the number of bins in the histogram, as described in (Chaudhuri et al., 1998). This works well for a single dimension, but how many elements do we need to sample when looking on more dimensions? It may be that larger sample sizes lead to better selectivity estimates.

As mentioned in Section 2.1, our system does not generate a list of most common values (MCVs) during histogram creation. On distributions where a few values make up a large portion of the data, such as the Zipfian distribution which often occurs in linguistic data, MCVs can greatly increase the accuracy of estimates derived from the histogram. In the two-dimensional case, an MCV is a unique tuple that appears often in the data.

Many of the approaches described in Section 4 would be excellent extensions to our system. Especially interesting is building histograms on attributes from different relations, as described by Bruno and Chaudhuri (2002). We could get significant speedups for certain kinds of joins by creating a histogram on correlated attributes used in those joins.

Currently we need to manually specify which multidimensional histograms to create. It would be desirable to instead have a algorithm that decides which (if any) attributes would be best served by a multidimensional histogram. This might involve running a battery of queries using the single dimensional histograms and evaluating the error of the selectivity estimates. From there one might generate histograms on pairs or k-tuples of attributes with the highest error rates. Some work in this area has been done by Markl et al. (2003), among others, but it remains an open question for further research.

6 Conclusions

When estimating selectivity, the current version of PostgreSQL and many other database systems assume query clauses are independent. When attributes are in fact correlated, this assumption can cause significant slowdown due to incorrect access paths. We have implemented a two-dimensional histogram to account for this correlation and produce more accurate selectivity estimations. With a little tweaking of parameters, these improved selectivity estimates lead to improved access paths, which we expect to result in faster query execution times.

The cost of creating these histograms is also relatively small. Statistic creation times do increase, but because statistics are gathered only infrequently, the benefits of faster query execution should outweigh these costs.

References

Nicolas Bruno and Surajit Chaudhuri. 2002. Exploiting statistics on query expressions for optimization. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on management of data*, pages 263–274, New York, NY, USA. ACM.

Surajit Chaudhuri, Rajeev Motwani, and Vivek Narasayya. 1998. Random sampling for histogram construction: how much is enough? *SIGMOD Rec.*, 27:436–447, June.

Amol Deshpande, Minos Garofalakis, and Rajeev Rastogi. 2001. Independence is good: dependency-based histogram synopses for high-dimensional data. In *SIGMOD '01: Proceedings of the 2001 ACM SIGMOD international conference on management of data*, SIGMOD '01, pages 199–210, New York, NY, USA. ACM.

V. Markl, G. M. Lohman, and V. Raman. 2003. LEO: An autonomic query optimizer for DB2. *IBM Systems Journal*, 42(1):98–106.

V. Markl, P. Haas, M. Kutsch, N. Megiddo, U. Srivastava, and T. Tran. 2007. Consistent selectivity estimation via maximum entropy. *The VLDB Journal*, 16(1):55–76, January.

M. Muralikrishna and David J. DeWitt. 1988. Equi-depth multidimensional histograms. In *SIGMOD '84: Proceedings of the 1984 ACM SIGMOD international conference on management of data*, pages 28–36, New York, NY, USA, June. ACM.

Viswanath Poosala and Yannis E. Ioannidis. 1997. Selectivity Estimation Without the Attribute Value Independence Assumption.

G. P. Shapiro and C. Connell. 1984. Accurate estimation of the number of tuples satisfying a condition. In *SIGMOD '84: Proceedings of the 1984 ACM SIGMOD international conference on management of data*, pages 256–276, June.

Michael Stonebraker, Lawrence A. Rowe, and Michael Hirohama. 1990. The implementation of POSTGRES. *IEEE Transactions and Knowledge and Data Engineering*, 2(1), March.

U.S. Census Bureau. 2009. Public-use microdata samples. `http://www.census.gov/main/www/pums.html`.