# Compressed Histograms with Arbitrary Bucket Layouts for Selectivity Estimation

Dennis Fuchs[1], Zhen He[2], Byung Suk Lee[3]
[1]Tele Atlas, Lebanon, NH 03766, USA*
[2]Department of Computer Science and Computer Engineering, La Trobe University, Bundoora, VIC 3086, Australia†
[3]Department of Computer Science, University of Vermont, Burlington, VT 05405, USA
Emails: defuchs@gmail.com, z.he@latrobe.edu.au, bslee@cems.uvm.edu

**Abstract**

Selectivity estimation is an important step of query optimization in a database management system, and multidimensional histogram techniques have proved promising for selectivity estimation. Recent multidimensional histogram techniques such as GenHist and STHoles use an *arbitrary* bucket layout. This layout has the advantage of requiring a smaller number of buckets to model tuple densities than those required by the traditional grid or recursive layouts. However, the arbitrary bucket layout brings an inherent disadvantage of requiring more memory to store each bucket location information. This diminishes the advantage of requiring fewer buckets and, therefore, has an adverse effect on the resulting selectivity estimation accuracy. To our knowledge, however, no existing histogram-based technique with arbitrary layout addresses this issue. In this paper, we introduce the idea of *bucket location compression* and then demonstrate its effectiveness for improving selectivity estimation accuracy by proposing the STHoles+ technique. STHoles+ extends STHoles by *quantizing* each coordinate of a bucket relative to the coordinate of the smallest enclosing bucket. This quantization increases the number of histogram buckets that can be stored in the histogram. Our quantization scheme allows STHoles+ to trade precision of histogram bucket locations for storing more buckets. Experimental results show that STHoles+ outperforms STHoles on various data distributions, query distributions, and other factors such as available memory size, quantization resolution, and dimensionality of the data space.

**Keywords**: selectivity, histogram, compression, self-tuning

## 1 Introduction

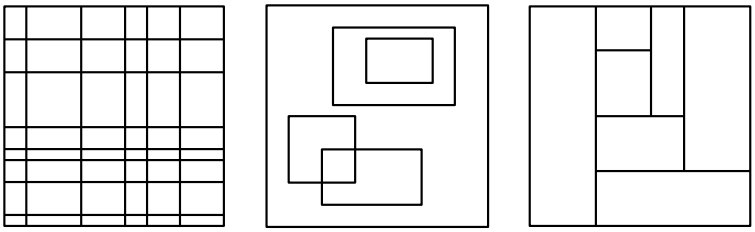
Selectivity estimation plays a key role in the query processing of a database management system. For instance, query optimizers use selectivity estimates when choosing the optimal join order for a query execution plan. It has been shown that selectivity estimation errors can increase exponentially with the number of joins [1] and, thus, adversely affect the join ordering decisions in query optimization. In this regard, the accuracy of selectivity estimation is important to fast query execution.

The selectivity estimation problem has been studied extensively in the existing literature [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]. One class of solutions is based on histograms. Histogram-based techniques [2, 3, 4, 9, 11, 12, 13] have the advantages that they are simple and make estimations quickly. Moreover, they are non-parametric techniques and, therefore, can model data of an arbitrary distribution.

*Work done while at Department of Computer Science, University of Vermont.
†Part of work done while at Department of Computer Science, University of Vermont.

Queries considered for selectivity estimation are range queries on one or more attributes. For these queries, we need *multidimensional* histogram techniques. Single-dimensional histogram techniques may be used under the assumption that the attributes are uncorrelated, but this assumption is rarely satisfied in real-life data [14]. For more accurate modeling of the joint distribution of correlated attributes, mulitidimensional techniques are more appropriate.

Two recent multidimensional histogram techniques, GenHist[9] and STHoles[4] allow buckets to have an arbitrary layout instead of the more traditional grid[2, 12] or recursive[3, 14] layout. Figure 1 shows examples of histograms that are built using the three types of bucket layouts. (These bucket layouts will be described in Section 2.2.2.) The arbitrary layout requires the smallest number of buckets to accurately model tuple density in the queried regions. However, this benefit comes at the cost that more memory is required to store each bucket, particularly its location information (e.g., coordinates). This cost diminishes the benefit and, therefore, has an adverse effect on the resulting selectivity estimation accuracy. Thus, it is desirable to reduce the cost. One natural approach to this is to compress "buckets" (specifically, their location information). As far as we know, however, no existing work has addressed this issue.



(a) Grid. (b) Arbitrary. (c) Recursive.

Figure 1: Histogram bucket layouts.

In this paper, we demonstrate the effectiveness of histogram compression by proposing STHoles+, an arbitrary bucket layout multidimensional histogram technique based on the state-of-the-art STHoles. The reason for choosing STHoles instead of GenHist (the other arbitrary layout technique) is that in a comprehensive performance evaluation[4], STHoles is found to give more accurate selectivity estimation than GenHist in a majority of situations. Additionally, in contrast to GenHist, STHoles does not need to scan the dataset to build and maintain the histogram. This results in a large reduction in the overhead associated with building and maintaining the histogram, especially when the dataset is large. We believe the idea of bucket location compression can be used for GenHist as well. We defer this to future work.

STHoles+ introduces a scheme called *quantized relative coordinates* to compress buckets. In this scheme, each rectangular bucket is represented with the two endpoints of the major diagonal and each endpoint is aligned with a regular grid inside the surrounding (i.e., parent) bucket. This results in quantizing the coordinates of the points. Then, the quantized coordinates are specified relative to the coordinates of the endpoints of the parent bucket, and encoded into a binary integer. A detailed explanation of this process will be given in Section 3.2.1.

Our performance evaluations show that STHoles+ achieves higher selectivity estimation accuracy than STHoles for various data distributions, query distributions, and other factors such as the available memory size, the resolution of coordinate quantization, and the dimensionality of the data space (i.e., Cartesian space formed by the query attributes). The advantage of STHoles+ over STHoles becomes more significant as the memory becomes more limited or the dimensionality becomes higher. (In both cases the selectivity estimation accuracy decreases for both STHoles and STHoles+).

We make the following main contributions through this paper.

- We propose using bucket location compression for building and maintaining more memory-efficient

arbitrary layout histograms.

- We demonstrate the effectiveness of our approach by proposing a new technique, STHoles+, for multidimensional selectivity estimation. STHoles+ is the first histogram technique to use bucket location compression. By taking advantage of the state-of-the-art STHoles technique, it continues to show the strength of STHoles while improving the selectivity estimation accuracy further by storing more buckets within the same amount of memory.
- We present a concrete design of the STHoles+ technique and demonstrate its performance (i.e., selectivity estimation accuracy) advantage over STHoles through a series of experiments.

The remaining sections are organized as follows. Section 2 discusses the related work with a primary focus on histogram-based selectivity estimation techniques. Section 3 describes the details of STHoles and the new STHoles+ techniques. Section 4 presents the experiments performed to compare the two techniques. Section 5 summarizes the paper and suggests future work.

# 2 Related Work

We discuss the related work in three steps of narrowing the focus. First, we give a brief overview of different selectivity estimation techniques, including histogram-based ones. Second, we provide a taxonomy of the histogram-based techniques, including those with an arbitrary bucket layout. Third, we describe two representative histogram-based techniques with an arbitrary bucket layout.

## 2.1 Selectivity estimation and estimation techniques

For a relational query optimizer, selectivity estimation is important to determine the order of join operations. A selection condition is specified as the range of values of attributes on the same relation. Given such a selection condition, its selectivity is defined as the number of tuples satisfying the condition over the number of all tuples in the relation. An example of a selection condition is $35 \leq Age < 40$ *AND* $80000 \leq Income < 100000$, assuming $Age$ and $Income$ are two attributes in the same relation (e.g., $Person$). Figure 2 illustrates how histogram-based selectivity estimation is used in query optimization. The selectivity estimator is part of the optimizer, and estimates the selectivity of a range query (part of the query) using histograms. The query execution plan (QEP) enumerator generates an optimal execution plan of the query by utilizing the selectivity information.

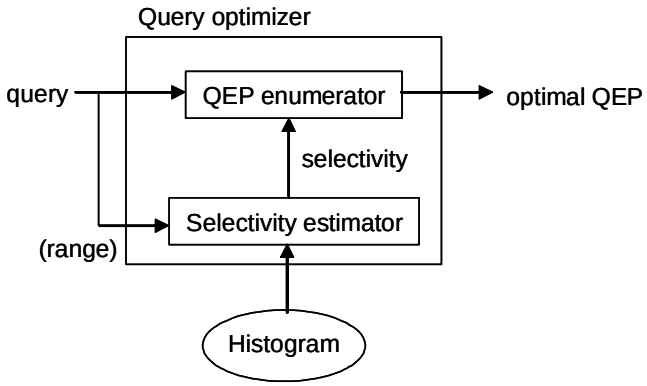



Figure 2: Selectivity estimation for query optimization.

Now we briefly characterize selectivity estimation techniques, identified through a comprehensive survey, in the categories of *sampling*, *parametric*, *probabilistic*, *wavelet-based*, and *histogram-based* techniques. Sampling techniques[16, 17] store a representative subset of the tuples in the data set. Selectivity estimations are then made by counting the sample tuples in the query region and scaling up the count appropriately. Obtaining the right representative sample is an important but often hard problem. Parametric techniques[5] use a parametric function to model the data set. Examples of parametric functions are probability distribution functions and polynomial functions describing curves or surfaces [5]. These techniques work well if the data fits the provided parametric function, but are infeasible otherwise.

Probabilistic techniques[6, 7] create a model that captures the conditional probabilities of the attributes. These techniques work statically and, therefore, cannot accommodate changes of the data set. Wavelet-based techniques[10, 11] work by compressing the values of the cumulative distribution function. These techniques can achieve high accuracy by modeling data at a higher resolution in regions with higher variation in tuple density [9]. However, it is nontrivial to perform dynamic updates on compressed data [11]. Histogram-based techniques[2, 3, 4, 9, 11, 12, 13] divide the data space into buckets, regions associated with a tuple count. The tuple density is assumed to be uniform inside each bucket. Under this assumption, selectivity estimations can be done straightforward by considering only the volumes of the regions intersecting between the query and the buckets.

## 2.2 Taxonomy of Histogram-Based Selectivity Estimation Techniques

The taxonomy is based on the histogram update scheme and the histogram bucket layout.

### 2.2.1 Histogram Update Scheme

The update scheme determines when a histogram is to be updated. Accordingly, we categorize histogram techniques into *static* techniques and *dynamic* techniques. Dynamic histogram techniques are further categorized into *data-driven* and *query-driven* techniques.

Static histogram techniques[9, 12, 13] build a histogram using all of the tuples in the data set, and then never update the histogram. Naturally, they perform poorly in situations where the data distribution changes rapidly. GenHist [9], to be described in Section 2.3, is the most recent example. In [12], Muralikrishna et al. introduce the equi-depth histogram technique which recursively partitions the multi-dimensional space into buckets containing near-uniform tuple density. In [13], Muthukrishnan et al. present near-linear time algorithms that generate static near-optimal bucket "tilings" for the three types of bucket layouts to be discussed in Section 2.2.2.

Data-driven dynamic histogram techniques[8, 15] update or regenerate a histogram as the data changes, in order to keep the histogram from becoming stale. Gibbons et al. [8] introduce a technique that dynamically updates a single-dimensional histogram as the data changes. Thaper et al. [15] describe a technique that keeps a summary of the changes to the data and generates a histogram on demand. Techniques using the data-driven update scheme do not take advantage of information from query executions that could be used to improve the accuracy of the histogram.

Query-driven dynamic histogram techniques[2, 4] update a histogram based on the tuples returned by queries (called query feedback). Histograms are refined through this feedback process, and, in this regard, are called "self-tuning" histograms as in STGrid [2] and STHoles [4]. (Section 2.3 contains a further discussion of STHoles.) The advantage of this update scheme is that the histogram is more refined in regions queried more frequently.

### 2.2.2 Histogram Bucket Layouts

As mentioned in Section 1, there are three types of bucket layouts: *grid*, *arbitrary*, and *recursive*.

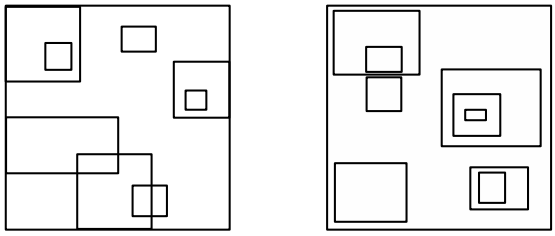
Grid bucket layout histograms[2, 12] arrange their buckets in rows and columns, as shown in Figure 1(a). This layout requires the least amount of memory per bucket among the three types of layouts because storing the grid information, comprising only the partition information in each dimension, is sufficient to specify the locations of all buckets in the histogram. However, due to its rigid layout, it ends up creating many unnecessary buckets, and thereby ends up with the largest number of buckets among the three types of layouts.

Arbitrary bucket layout histograms[4, 9] allow rectangular buckets to be placed at any location, as shown in Figure 1(b). As mentioned in Section 1, this layout requires the smallest number of buckets for accurate modeling of tuple density among the three types of layouts. The reason is that only the necessary buckets need to be created due to the layout's flexible bucket placement independent of other buckets. By the same token, this layout requires the most memory per bucket among the three types because buckets' locations should be specified separately for each bucket. We have found two histogram techniques in this category: GenHist [9] and STHoles [4]. GenHist's bucket layout is completely arbitrary, whereas the STHoles' bucket layout is slightly restricted—buckets are not allowed to overlap. (Section 2.3 covers these two techniques further.)

Recursive bucket layout histograms[3, 14] create buckets by recursively partitioning the data space in only dimension each time, as shown in Figure 1(c). Its bucket geometry is a mixture of those of the other two types. Hence, naturally the number of buckets and the amount of memory required for this layout are between those of the other two layouts.

### 2.3 Histogram Techniques Using the Arbitrary Bucket Layout

GenHist [9] and STHoles [4] are the two histogram techniques we have found to use the arbitrary bucket layout. Both can benefit from the bucket location compression.



(a) GenHist. (b) STHoles.

Figure 3: Histogram techniques employing an arbitrary bucket layout.

GenHist is a static histogram technique that creates a histogram consisting of rectangular buckets. Figure 3(a) depicts a GenHist histogram. The histogram is constructed using the following heuristic approach: while increasing the regular grid spacing starting with a given smallest size, select a given number of densest grid cells and make each of them into a bucket. The resulting histogram may contain overlapping buckets. When estimating selectivities for regions where two or more buckets overlap, the sum of the densities of the buckets is used. GenHist does not make use of query feedback and is difficult to tune due to multiple parameters such as the initial grid spacing, the number of buckets created in each iteration, and the rate at which the grid spacing increases for each iteration [4].

STHoles is a dynamic histogram technique that has an arbitrary layout with the restriction that buckets may not overlap, as depicted in Figure 3(b). It allows buckets to be contained within other buckets, thus creating a parent-child hierarchy of buckets. STHoles uses query feedback to capture the tuple count and the processed range query, and creates one or more new buckets as a result of each feedback. When the number

of buckets becomes too large for all the buckets to fit in the available memory, some buckets are merged. (Section 3 provides the details of the bucket insertion and merging.)

# 3 The STHoles+ Histogram Technique

In this section we first describe the existing STHoles technique, and then present the design of the new STHoles+ technique.

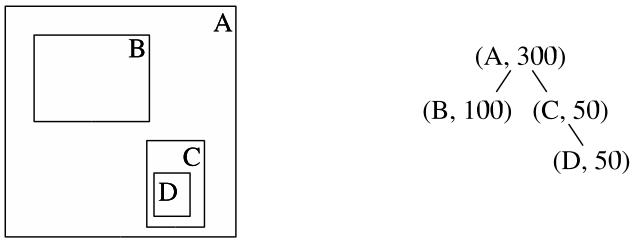
## 3.1 STHoles

We summarize only the key structures and operations of STHoles here. More details can be found in the paper by Bruno et al. [4].

### 3.1.1 Histogram Structure

As already mentioned in Section 2.3, an STHoles histogram consists of rectangular buckets that do not overlap. Buckets, however, may be fully enclosed within another bucket. In this case, the enclosed bucket is called the "child" of the enclosing bucket, which is called the "parent". STHoles buckets thus form a tree structure based on the parent-child relationship. Each bucket has an associated tuple count, i.e., the number of tuples in the region covered by the bucket.

The histogram is interpreted in the following way. The volume inside a child bucket does not belong to the parent bucket. (Thus, it appears as if the child bucket cuts a "hole" out of the parent bucket – hence the name STHoles.) This means that the tuple count of a parent bucket does not reflect the tuples inside its children buckets. Likewise, the volume of a parent bucket is the volume of the parent bucket's rectangle minus the volumes of all children buckets' rectangles.




(a) Buckets. (b) Tree structure of buckets.

Figure 4: STHoles example histogram.

**Example 1** Figure 4(a) shows the buckets of an example STHoles histogram, and Figure 4(b) shows the corresponding tree structure of nodes, where each node is represented as the pair (bucket, tuple count). $A$, $B$, $C$, and $D$ are buckets with tuple counts of 300, 100, 50, and 50, respectively. Bucket $A$ is the parent of bucket $B$ and bucket $C$, and bucket $C$ is in turn the parent of bucket $D$. The total number of tuples reflected in the histogram is 500 ($= 300 + 100 + 50 + 50$). Note that tuples are not double-counted; that is, if a tuple is counted as one in a child bucket, then it is not counted as one in the parent bucket. Likewise, the volume occupied by a parent bucket does not include the volume occupied by its children buckets. □

### 3.1.2 Selectivity Estimation

Given a range query and an STHoles histogram, computing the selectivity estimation is straightforward because both the buckets and queries are rectangular in shape. For each bucket in the histogram, we first compute the volume of the intersecting region (if there is any) between the bucket and the query. Second, we divide the volume of the intersecting region by the volume of the bucket. Third, we multiply the resulting fraction by the tuple count of the bucket; this gives an approximate tuple count of the intersecting region. Once the tuple count is computed for the intersecting region of each bucket, then we obtain the estimated selectivity of the query by summing the approximate tuple counts of all intersecting regions and dividing the sum by the total number of tuples in the data set.

Thus, the selectivity estimate for range query $q$ using histogram $H$, $est(H, q)$, covering a data space $D$ is given by

$$\mathbf{est(H, q)} = \frac{\sum_{\mathbf{b \in H}} \frac{\mathbf{v(q \cap b)}}{\mathbf{v(b)}} \cdot \mathbf{f(b)}}{\mathbf{f(D)}} \tag{1}$$

where $f(b)$ and $f(D)$ denote the tuple counts of the regions $b$ and $D$, respectively, and $v(b)$ and $v(q_i \cap b)$ denote the volume of the bucket $b$ and the volume of the intersecting region between $b$ and $q_i$, respectively. Note, as mentioned above, $v(b)$ does not include the volumes of $b$'s children buckets, and $f(b)$ does not include the tuple counts of $b$'s children buckets.

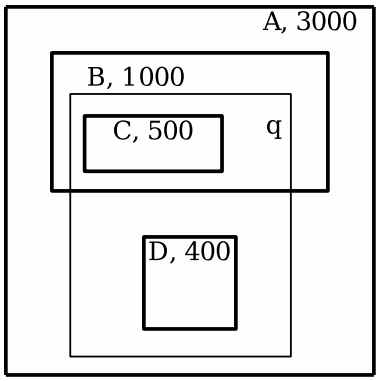



Figure 5: STHoles selectivity estimation.

**Example 2** Figure 5 illustrates the process of estimating the selectivity of a range query. The volume of query $q$ consists of 30% of the volume of $A$, 50% of the volume of $B$, and 100% of the volumes of $C$ and $D$. Thus, the selectivity estimation is computed as

$$\frac{(0.3 \times 3000) + (0.5 \times 1000) + 500 + 400}{4900} = \frac{2300}{4900} = 0.47$$

□

### 3.1.3 Histogram Refinement

As already mentioned in Section 2.2, STHoles is updated as a result of query feedback after the execution of each query. Figure 6 shows such a feedback loop added to Figure 2. The query feedback information is comprised of the query's range information and the actual tuples. The sequence of histogram updates resulting from a sequence of queries is called the *histogram refinement* in [4].

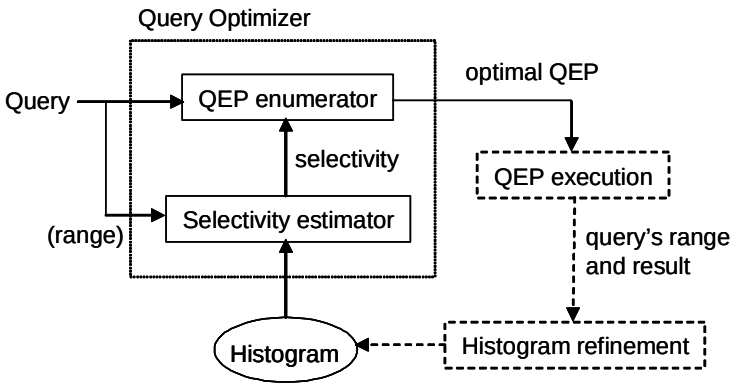



Figure 6: Query feedback loop in STHoles.

Histogram refinement is a two step process: bucket *insertion* and bucket *merging*. After processing each query, new buckets are built based on the query feedback information and are inserted into the histogram. The histogram may become too large to fit in the allocated memory if all the new buckets are inserted. In this case, some buckets are merged until the histogram fits within the memory. We now explain each of these two steps in more detail.

**Bucket insertion**

Bucket insertion is done in two steps: identifying candidate buckets (or holes) and adding some or all of the candidate buckets to the histogram. ("Adding a bucket" is called "drilling a hole" in [4].) The first step takes place only if the query rectangle intersects at least one existing buckets. In this case, if an intersecting region is a *rectangle*, then the rectangle makes a candidate. Otherwise, the intersecting region is shrunken until it becomes a rectangle, and the resulting rectangle makes a candidate. If there are more than one way to shrink, then the rectangle with the smallest shrunken volume is chosen. The candidate holes identified this way do not *partially* intersect with any existing bucket.

Each of the new candidate buckets thus identified is then added to the histogram if its tuple count is sufficiently different from what would be estimated from the histogram without the bucket. The tuple count of each new bucket is determined by counting the number of tuples that fall inside the new bucket among those returned by the query. If a new bucket is created inside an existing bucket, the tuple count of the new bucket is subtracted from that of the enclosing bucket (i.e., the parent). Specifically, there are three cases considered depending on how a candidate hole ($c$) is placed with respect to a particular existing bucket ($b$):

- Candidate hole $c$ occupies exactly the same region as the bucket $b$. That is, $box(c) = box(b)$, where $box(\cdot)$ denotes the rectangle of a bucket. In this case, we do not drill the hole $c$ but simply replace $b$'s tuple count with $c$'s tuple count ($T_c$).

- Candidate hole $c$ fills up the bucket $b$, that is, covers all the region remaining in $b$ after excluding the regions covered by all of $b$'s children. This is a rare case. (See Figure 7 for an illustration.) In this case, we merge $b$ with its parent ($b_p$) and then drill the hole $c$ as a child of $b_p$. The bucket $b$ is removed (through the merging) because there is no region left over for $b$ after inserting $c$.

- Candidate hole $c$ does not fill up the bucket $b$, that is, covers only part of the region remaining in $b$ after excluding the regions covered by all of $b$'s children. This is the most common case. In this case, we drill the hole $c$ inside $b$ and update the tuple counts of the enclosing buckets.

The algorithm *DrillHole* in the following figure summarizes these three cases.

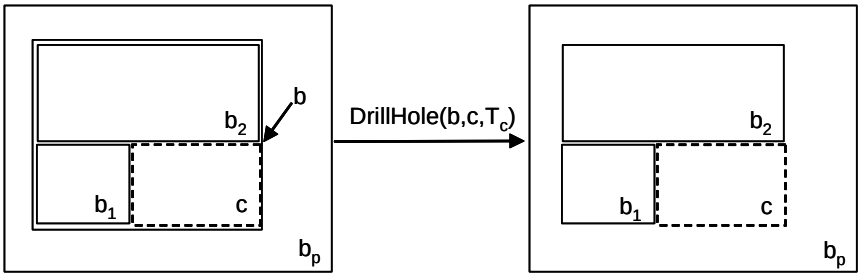



Figure 7: Case of a candidate hole filling up a bucket (adopted from [4]).

DrillHole($b$: bucket, $c$: candidate hole, $T_c$: tuple count of $c$)

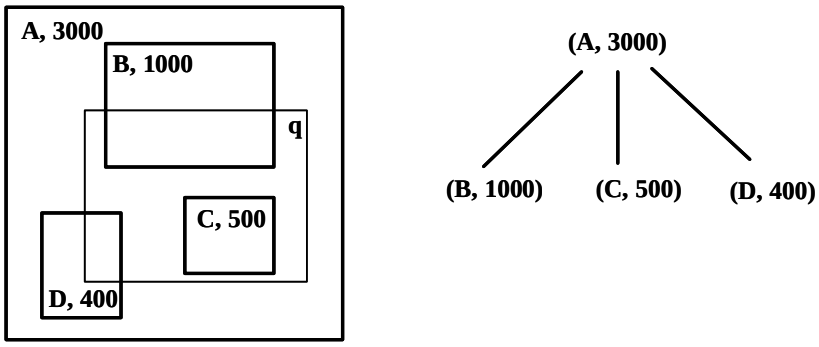
1. if box($b$) == box($c$) then // Case 1
2. f($b$) = $T_c$ // f($b$): tuple count (i.e., frequency) of bucket $b$
3. else
4. if $c$ covers all the region remaining in $b$ then // Case 2
5. merge $b$ with its parent bucket $b_p$
6. DrillHole($b_p$, $c$, $T_c$)
7. else // Case 3
8. add $c$ as a new child bucket $b_c$ of $b$
9. f($b_c$) = $T_c$
10. migrate all children of $b$ that are enclosed by $c$ so that they become children of $b_c$
11. f($b$) = max(0, f($b$) - $T_c$)

Figure 8: DrillHole algorithm for STHoles (adapted from [4]).

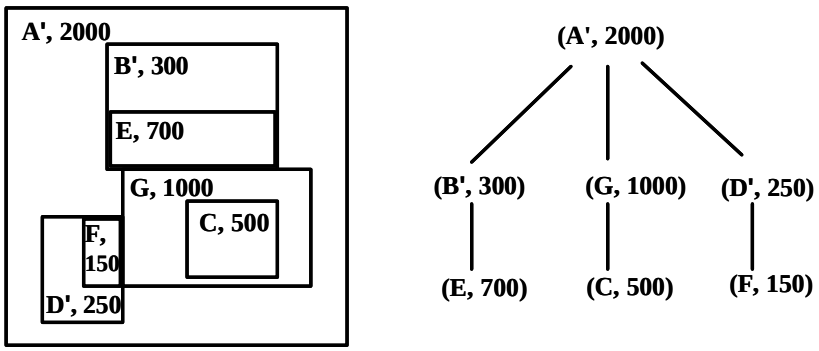
**Example 3** Figure 9 illustrates the process of inserting new buckets into an STHoles histogram after executing a query $q$. The query rectangle of $q$ intersects with $B$ and $D$. ($C$ is enclosed in $q$, not intersects with $q$.) Since the two intersecting regions are rectangles, the buckets $E$ and $F$ are identified as candidate holes. The query rectangle also intersects with $A$, but the intersecting region (in a shade in the figure) is not a rectangle. This region is thus shrunken to a rectangular region and the bucket $G$ is identified as a candidate hole. Let us assume the number of tuples in $E$ is 700, in $F$ is 150, and in $G$ is 1000 (excluding the tuples in $C$). Also assume that the tuple counts of all three buckets are different enough from what can be estimated without the buckets. Since $E$ is enclosed in $B$, $E$ becomes a child of $B$, and its tuple count 700 is subtracted from the tuple count of $B$. Likewise, $F$ becomes a child of $D$, and its tuple count 150 is subtracted from the tuple count of $D$. In addition, since $G$ is enclosed in $A$, $G$ becomes a child of $A$, and its tuple count 1000 is subtracted from the tuple count of $A$. Note that $G$ is the parent of $C$, as $C$ is enclosed in $G$. □

**Bucket merging**

When the histogram becomes too large to fit in the allocated memory, some pairs of buckets are merged. To determine which buckets to merge, all pairs of buckets with a parent-child relationship (that is, one bucket contains the other) or a sibling-sibling relationship (that is, both buckets share a common parent bucket) are examined. In a parent-child merge, if the merged child bucket has any children of its own, the children are "adopted" by the parent. This is straightforward. It is a bit more complicated in a sibling-sibling merge. When two siblings are merged, the merged bucket should cover as small a region as possible; this is necessary to minimize the loss of selectivity estimation accuracy. However, the minimum bounding region may intersect with other siblings. In this case, the merged bucket should be enlarged (to the minimum extent possible) until there is no intersecting bucket.




(a) Histogram before bucket insertion.




(b) Histogram after bucket insertion.

Figure 9: STHoles bucket insertion.

For each of those merged pairs, the *merge penalty* is calculated. The metric of the penalty is the expected loss of selectivity estimation accuracy resulting from the merge. Then, pairs of buckets are merged starting with the pair with the smallest merge penalty until all remaining buckets can fit in the allocated memory. In [4], the following formulas are used to compute merge penalty. (Readers are referred to [4] for details of deriving these formulas.)

- *Parent-child merge penalty*: given a histogram $H$, the penalty of merging a parent bucket $b_p$ and a child bucket $b_c$ into a new bucket $b_n$ is computed as:

$$|f(b_p) - f(b_n)\frac{v(b_p)}{v(b_n)}| + |f(b_c) - f(b_n)\frac{v(b_c)}{v(b_n)}| \tag{2}$$

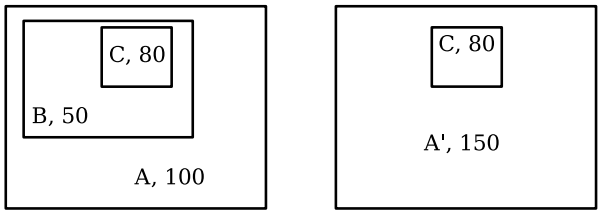
  where $f(b)$ denote the tuple count of the bucket $b$.

- *Sibling-sibling merge penalty*: given a histogram $H$, the penalty of merging two buckets $b_1$ and $b_2$ that have a common parent $b_p$ into a new bucket $b_n$ is computed as:

$$|f(b_n)\frac{v_{old}}{v(b_n)} - f(b_p)\frac{v_{old}}{v(b_p)}| + |f(b_1) - f(b_n)\frac{v(b_1)}{v(b_n)}| + |f(b_2) - f(b_n)\frac{v(b_2)}{v(b_n)}| \tag{3}$$

  where $v_{old}$ is the volume of the region taken from $b_p$ into $b_n$ as the outcome of finding $b_n$; $f(b)$ denote the tuple count of the bucket $b$.

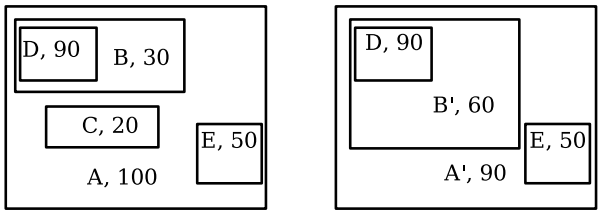
**Example 4** Figure 10(a) illustrates merging two buckets in a parent-child relationship in an STHoles histogram. There are two pairs of buckets that can be merged—the pair of $A$ and $B$ and the pair of $B$ and $C$.




(a) Before. (b) After.

Figure 10: STHoles parent-child merge.

Assume the calculated merge penalty is smaller for the pair of $A$ and $B$. Then, bucket $B$ is merged into its parent, bucket $A$. During this merge, the tuple count of the child bucket B is added to that of the parent bucket $A$. Besides, the children buckets of $B$ (in this case, bucket $C$) are made the children buckets of $A$. Finally, $B$ is removed. The resulting buckets are shown in Figure 10(b). □




(a) Before. (b) After.

Figure 11: STHoles sibling-sibling merge.

**Example 5** Figure 11(a) illustrates merging buckets in the sibling-sibling relationship in an STHoles histogram. There are three pairs of buckets that can be merged—the pairs of $B$ and $C$, $C$ and $E$, and $E$ and $B$. Assume that, among these three, the calculated merge penalty is the smallest for the pair of $B$ and $C$. Then, $B$ and $C$ are merged into a new bucket $B'$, whose rectangle is the minimum bounding box containing $B$ and $C$. The tuple count of bucket $B'$ is calculated by adding the tuple counts of $B$ and $C$ and the portion of $A$'s tuple count corresponding to the volume taken from $A$ into $B'$. The children buckets of $B$ and $C$ (in this case, bucket $D$) are made children of $B'$. Finally, $B$ and $C$ are removed. The resulting buckets are shown in Figure 11(b). □

## 3.2 STHoles+

We describe first the quantized relative coordinate scheme used by STHoles+, and then the STHoles+ technique with a focus on the changes required according to the scheme. The main differences appear in the histogram structure and refinement.

### 3.2.1 Quantized Relative Coordinates

The idea of quantized relative coordinate scheme is similar to the one used by the A-tree[18] for indexing high dimensional spaces. The scheme compresses the coordinates of a rectangle by (1) measuring the coordinates of the rectangle relative to the coordinates of the smallest enclosing rectangle, (2) quantizing the resulting relative coordinates, and (3) encoding the resulting value using the smallest number of bits.

Figure 12 illustrates this scheme. A rectangle $A$ in a $d$-dimensional space is represented by the two extreme corners of the major diagonal, $\mathbf{a_s} = \langle a_{s_1}, a_{s_2}, \ldots, a_{s_d} \rangle$ and $\mathbf{a_e} = \langle a_{e_1}, a_{e_2}, \ldots, a_{e_d} \rangle$ such that

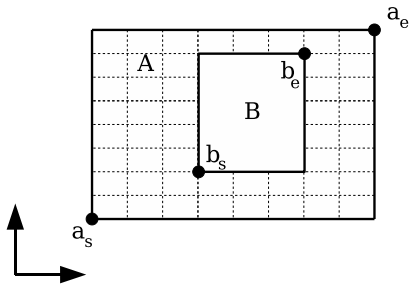



Figure 12: Rectangle $A$ with a grid and rectangle $B$ contained in $A$.

$a_{s_i} \leq a_{e_i}$ for $i = 1, 2, \ldots, d$. Then, given a rectangle $A \equiv (\mathbf{a_s}, \mathbf{a_e})$ that contains a rectangle $B \equiv (\mathbf{b_s}, \mathbf{b_e})$, for each $i^{th}$ dimension, $i = 1, 2, \ldots, d$, it quantizes the start value $b_{s_i}$ and the end value $b_{e_i}$ of the interval $(b_{s_i}, b_{e_i})$ relative to the interval $(a_{s_i}, a_{e_i})$ .

Specifically, a regular grid is first created within the rectangle $A$, where the number of partitions in each dimension is provided as the *quantization resolution* $k$. Then, each side of the rectangle $B$ is aligned with the grid. For this alignment, the start value $b_{s_i}$ ($i = 1, 2, \ldots, d$) is quantized using the following *quantization function* $Q_S$:

$$Q_S(b_{s_i}) = \left\lceil \left( \frac{b_{s_i} - a_{s_i}}{a_{e_i} - a_{s_i}} \right) k \right\rceil \tag{4}$$

where $\lceil \cdot \rceil$ is a rounding function and $a_{s_i} \leq b_{s_i} < a_{e_i}$. Likewise, the end value $b_{e_i}$ ($i = 1, 2, \ldots, d$) is quantized using the following quantization function $Q_E$:

$$Q_E(b_{e_i}) = \left\lceil \left( \frac{b_{e_i} - a_{s_i}}{a_{e_i} - a_{s_i}} \right) k \right\rceil - 1 \tag{5}$$

where $a_{s_i} < b_{e_i} \leq a_{e_i}$. $Q_E$ is essentially the same as $Q_S$, with the only exception that 1 is subtracted to keep the resulting value between 0 and $k - 1$ (inclusive). This way, the quantized value can be stored using $\lceil log_2 k \rceil$ bits.

**Example 6** In Figure 12, let us assume the quantization resolution $k$ equals 8. (This value is too small to be practical, but used here for the clarity of presentation.) The rectangles A and B are defined by the coordinates $\mathbf{a_s} \equiv \langle a_{s_1}, a_{s_2} \rangle = \langle 45, 25 \rangle$, $\mathbf{a_e} \equiv \langle a_{e_1}, a_{e_2} \rangle = \langle 200, 100 \rangle$, $\mathbf{b_s} \equiv \langle b_{s_1}, b_{s_2} \rangle = \langle 100, 40 \rangle$, and $\mathbf{b_e} \equiv \langle b_{e_1}, b_{e_2} \rangle = \langle 160, 90 \rangle$. The quantized coordinates $Q_S(b_{s_1})$, $Q_S(b_{s_2})$, $Q_E(b_{e_1})$, and $Q_E(b_{e_2})$ are computed as

$$Q_S(b_{s_1}) = \left\lceil \left( \frac{b_{s_1} - a_{s_1}}{a_{e_1} - a_{s_1}} \right) k \right\rceil = \left\lceil \left( \frac{100 - 45}{200 - 45} \right) 8 \right\rceil = 3$$

$$Q_S(b_{s_2}) = \left\lceil \left( \frac{b_{s_2} - a_{s_2}}{a_{e_2} - a_{s_2}} \right) k \right\rceil = \left\lceil \left( \frac{40 - 25}{100 - 25} \right) 8 \right\rceil = 2$$

$$Q_E(b_{e_1}) = \left\lceil \left( \frac{b_{e_1} - a_{s_1}}{a_{e_1} - a_{s_1}} \right) k \right\rceil - 1 = \left\lceil \left( \frac{160 - 45}{200 - 45} \right) 8 \right\rceil - 1 = 5$$

$$Q_E(b_{e_2}) = \left\lceil \left( \frac{b_{e_2} - a_{s_2}}{a_{e_2} - a_{s_2}} \right) k \right\rceil - 1 = \left\lceil \left( \frac{90 - 25}{100 - 25} \right) 8 \right\rceil - 1 = 6$$

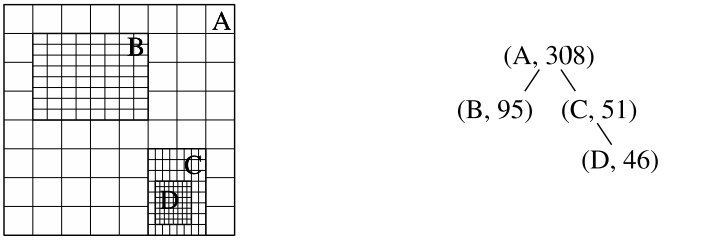
Thus, the quantized coordinates of $\mathbf{b_s}$ (i.e., $\langle Q_S(b_{s_1}), Q_S(b_{s_2}) \rangle$) are $\langle 3, 2 \rangle$ and the quantized coordinates of $\mathbf{b_e}$ (i.e., $\langle Q_E(b_{e_1}), Q_E(b_{e_2}) \rangle$) are $\langle 5, 6 \rangle$. Each coordinate value is then encoded into a binary integer of length $\lceil log_2 k \rceil = \lceil log_2 8 \rceil = 3$ bits, that is, as $\langle 011, 010 \rangle$ for $\mathbf{b_s}$ and $\langle 101, 110 \rangle$ for $\mathbf{b_e}$. □

| Memory Size | STHoles (Single-Precision) | STHoles (Double-Precision) | STHoles+ |
|---|---|---|---|
| 512 bytes | 21 | 12 | 42 |
| 1024 bytes | 42 | 25 | 85 |

Table 1: Maximum number of buckets that can be stored (dimensionality = 3).

### 3.2.2 Histogram Structure

Since the bucket coordinates are quantized, STHoles+ places an additional restriction on the location of histogram buckets. That is, the quantized coordinate of a bucket in each dimension must be an integer between 0 and $k$ (inclusive) within its parent bucket. Here, $k$ is the quantization resolution.




(a) Buckets (k=8). (b) Tree structure of buckets.

Figure 13: STHoles+ example histogram.

**Example 7** Figure 13(a) shows the buckets of an STHoles+ histogram that corresponds to the one shown for STHoles in Example 1. The grid for each bucket is also shown, with the quantization resolution of the grids set to 8. The corresponding tree structure of buckets is shown in Figure 13(b). $A$, $B$, $C$, and $D$ are buckets with tuple counts of 308, 95, 51, and 46, respectively. These tuple counts are slightly different from those in Figure 4. This is due to the need for adjusting the sizes of buckets to align them with the grid lines. The tuple counts are then obtained by counting the actual tuples in the adjusted buckets among the tuples in the query feedback. □

As described in Section 3.2.1, quantized relative coordinates need $\lceil log_2 k \rceil$ bits to store each coordinate component. Since a bucket coordinate contains $d$ components (where $d$ is the dimensionality of the data space) and each bucket is specified by a pair of coordinates (i.e., the start point and end point of the major diagonal), the total amount of memory required to specify the location of a bucket is $2d\lceil log_2 k \rceil$ bits.

Table 1 compares STHoles and STHoles+ in terms of the number of buckets that can be stored within the same amount of memory, assuming a three-dimensional data space. The calculations are based on a quantization resolution of 256 (the default value used in our experiments), a single-precision floating-point number size of 32 bits, and a double-precision floating-point number size of 64 bits. It shows that STHoles+ stores twice as many buckets as STHoles if STHoles uses single-precision coordinates and more than three times as many buckets as STHoles if it uses double-precision coordinates.

In STHoles+, we introduce the notion of *adapter holes*. An adapter hole is a bucket used solely for increasing the precision of a child bucket location. For instance, if a new bucket to be added is smaller than

the grid spacing, then an adapter hole is created to refine the grid spacing so that the new bucket can be aligned with the grid. Adapter holes are further explained in Section 3.2.4.

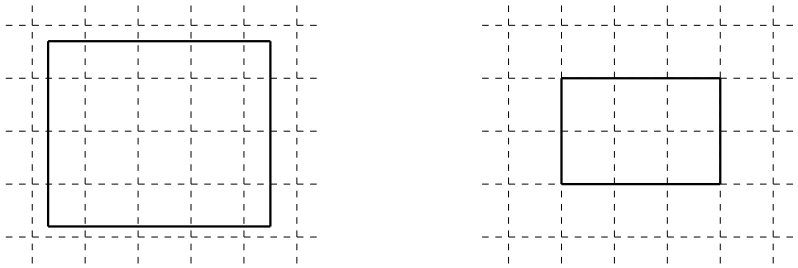
### 3.2.3 Selectivity Estimation

Selectivity estimation is done in the same manner as in STHoles. The only difference is that, when finding the intersections between a query rectangle and the existing buckets, the bucket coordinates need to be converted from quantized relative coordinates to absolute coordinates to be compared with the query rectangle coordinates. The overhead of this number conversion is negligible.

Note that adapter holes are not used in calculating selectivity estimates because they do not come from queries and, therefore, have no associated tuple counts. Thus, if a query rectangle intersects an adapter hole, the overlapping region is assumed to have the density of the adapter hole's parent bucket.

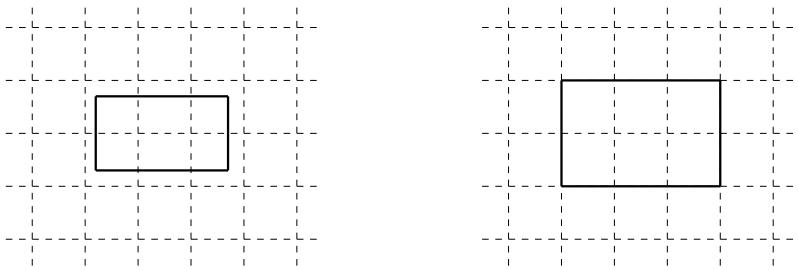
### 3.2.4 Histogram Refinement

In the bucket insertion and merging process, the quantized relative coordinate scheme necessitates two additional operators specific to STHoles+: *snap-in* and *snap-out*. The snap-in operator quantizes the coordinates of a new bucket so that it *shrinks* to fit the grid lines of the parent bucket. In contrast, the snap-out operator quantizes them so that a new bucket *expands* to fit the grid lines of the parent bucket. Figures 14 and 15 illustrate these two operators. Their algorithms are straightforward, and are shown in Figures 16 and 17, respectively.



(a) Before. (b) After.

Figure 14: Bucket before and after snap-in.



(a) Before. (b) After.

Figure 15: Bucket before and after snap-out.

Snap_in($b$: bucket to be inserted, $p$: parent bucket of $b$, $k$: quantization resolution,
$d$: dimensionality of the data space)

1. for each $i^{th}$ ($i = 1, 2, \cdots, d$) dimension in the data space
2. $p_{s_i}$ = minimum value of bucket $p$'s coordinate in the $i^{th}$ dimension
3. $p_{e_i}$ = maximum value of bucket $p$'s coordinate in the $i^{th}$ dimension
4. $b_{s_i}$ = minimum value of bucket $b$'s coordinate in the $i^{th}$ dimension
5. $b_{e_i}$ = maximum value of bucket $b$'s coordinate in the $i^{th}$ dimension
6. $q_{p_i} = (p_{e_i} - p_{s_i})/k$ // $q_{p_i}$: quantum size of bucket $p$'s coordinate in the $i^{th}$ dimension
// Let $\langle s_{s_i}, s_{e_i} \rangle$ be the pair of the minimum and maximum values of the coordinate
// of the snapped-in bucket, $s$, in the $i^{th}$ dimension. Then:
7. $s_{s_i} = p_{s_i} + q_{p_i} \lceil (\frac{b_{s_i} - p_{s_i}}{p_{e_i} - p_{s_i}})\, k \rceil$
8. $s_{e_i} = p_{s_i} + q_{p_i} \lfloor (\frac{b_{e_i} - p_{s_i}}{p_{e_i} - p_{s_i}})\, k \rfloor$
9. end for
10. return $s$

Figure 16: Snap-in algorithm of STHoles+.

Snap_out($b$: bucket to be inserted, $p$: parent bucket of $b$, $k$: quantization resolution,
$d$: dimensionality of the data space)

1. for each $i^{th}$ ($i = 1, 2, \cdots, d$) dimension in the data space
2. $p_{s_i}$ = minimum value of bucket $p$'s coordinate in the $i^{th}$ dimension
3. $p_{e_i}$ = maximum value of bucket $p$'s coordinate in the $i^{th}$ dimension
4. $b_{s_i}$ = minimum value of bucket $b$'s coordinate in the $i^{th}$ dimension
5. $b_{e_i}$ = maximum value of bucket $b$'s coordinate in the $i^{th}$ dimension
6. $q_{p_i} = (p_{e_i} - p_{s_i})/k$ // $q_{p_i}$: quantum size of bucket $p$'s coordinate in the $i^{th}$ dimension
// Let $\langle s_{s_i}, s_{e_i} \rangle$ be the pair of the minimum and maximum values of the coordinate
// of the snapped-out bucket, $s$, in the $i^{th}$ dimension. Then:
7. $s_{s_i} = p_{s_i} + q_{p_i} \lfloor (\frac{b_{s_i} - p_{s_i}}{p_{e_i} - p_{s_i}})\, k \rfloor$
8. $s_{e_i} = p_{s_i} + q_{p_i} \lceil (\frac{b_{e_i} - p_{s_i}}{p_{e_i} - p_{s_i}})\, k \rceil$
9. end for
10. return $s$

Figure 17: Snap-out algorithm of STHoles+.

Snap-in is needed to align the sides of a new bucket with the parent bucket's grid lines. The reason for snapping *in* for the alignment is as follows. A new bucket is originated from a query rectangle, and data inside the query rectangle is assumed to be uniformly distributed (for the purpose of selectivity estimation) but data outside the query rectangle is not. Thus, we do not allow a query rectangle to expand along any dimensional axis. In other words, only snap-in is allowed.

The snap-out operator is needed for the following reason. If a new bucket is too small to enclose any grid cell of the parent bucket, then a snap-in causes the new bucket to disappear (i.e., to zero volume). This problem can be avoided by making the grid spacing small enough to house the new bucket. An *adapter hole* is needed for this purpose. If a hole disappears as a result of snap-in, an adapter hole is created by snapping out the new bucket. Given the same quantization resolution, the resulting adapter hole has a grid spacing smaller than that of the parent bucket, thus allowing the new bucket to be inserted. In case the new bucket is extremely small compared with the parent bucket's grid spacing, then adapter holes may need to be nested, created recursively until the grid spacing becomes small enough to house the new bucket.

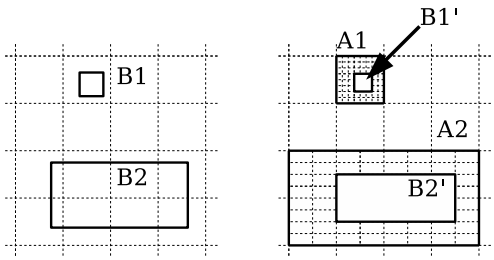



Figure 18: Adapter hole creation.

**Example 8** Figure 18 shows two new buckets $B1$ and $B2$, neither of which can be snapped in. Therefore, the adapter holes $A1$ and $A2$ are created by snapping out $B1$ and $B2$, respectively, and then $B1$ and $B2$ are snapped in to $B1'$ and $B2'$ within the grids of $A1$ and $A2$, respectively. □

**Bucket insertion**

The bucket insertion algorithm of STHoles+ differs from that of STHoles only in the second step, that is, "drilling a hole." Figure 19 shows the *DrillHole* algorithm of STHoles+. Compared with the algorithm in STHoles, the main difference comes from the use of adaptor holes. Another difference is that we do not consider Case 2, the case in which a candidate hole $c$ occupies a bucket $b$'s remaining space. In this case, $b$ is merged with its parent in STHoles. This, however, cannot be done in STHoles+ because STHoles+ allows only leaf nodes to be merged (see the description of bucket merging below). Omitting this case has hardly any adverse effect on the performance of STHoles+, since it is a rare case.

Now, let us describe the details of the algorithm. First, if the candidate hole $c$ occupies exactly the same region covered by the bucket $b$, the algorithm simply replaces $b$'s tuple count by $c$'s tuple count without drilling a hole; this is the same as Case 1 in STHoles (Lines 1–2). Otherwise (i.e., if $c$ occupies only part of $b$, i.e., Case 3 of STHoles), $c$ is snapped in to the grid of $b$; if the snap-in causes $c$ to disappear (i.e., size becomes zero), then adapter holes are created recursively until the last (smallest) adapter hole is small enough to let $c$ snap in without disappearing (Lines 4–7). Then, the snapped-in $c$ is drilled as a child of $b$ in the same manner as in STHoles (Lines 9–11). Adapter holes do not have tuple counts associated with them and, therefore, the tuple count of $b$ is updated only if $b$ is not an adapter hole; if $b$ *is* an adaptor hole, then we update the tuple count of the closest ancestor of $b$ that is not an adapter hole (Lines 12–16).

**Bucket merging**

```
DrillHole(b: bucket, c: candidate hole, T_c: tuple count of c, k: quantization resolution,
          d: dimensionality of data space)

1. if box(b) == box(c) then     // c occupies the same region as b
2.     f(b) = T_c     // f(b): tuple count (i.e., frequency) of bucket b
3. else
4.     b_in = snap-in(c, b, k, d)    // snap in c using the grid of b
5.     if size of b_in == 0 then
6.         b_a = snap-out(c, b, k, d)    // create a new adapter hole b_a (of b) by snapping out c
                                         // using the grid of b
7.         DrillHole(b_a, c, Tc)
8.     else
9.         add b_in as a new child bucket of b
10.        f(b_in) = T_c
11.        migrate all children of b that are enclosed by c so that they become children of b_in
12.        if b is an adapter hole then
13.            b_na = closest non-adapter hole ancestor of b.
14.        else
15.            b_na = b
16.        f(b_na) = max(0, f(b_na) - T_c)
```

Figure 19: DrillHole algorithm for STHoles+.

In addition to the modifications for snap-in's and possible adapter holes, we make another modification to STHoles' bucket merging process: bucket merging is considered only at *leaf* nodes of the parent-child bucket tree. In other words, in the case of a parent-child merge, the child bucket must not contain any child of its own and, in the case of a sibling-sibling merge, neither bucket may contain any child. The reason is that, if non-leaf nodes were to be merged, then all the buckets contained in the merged bucket would have to be realigned (i.e., snapped in) with the grid of the merged bucket. Then, the tuple counts of the realigned buckets must be *calculated* assuming uniform tuple density – not counted from the actual tuples – because the tuples in the previous query feedback are not available anymore. (Besides, the realignments change the bucket locations to some extent.) This results in a decrease of the histogram accuracy. As mentioned above in the discussion of bucket insertion, the only problem caused by limiting the merge to non-leaf nodes is the omission of STHole's Case 2 in the bucket insertion algorithm. Since this case is very rare, the penalty for this modification is negligible.

Furthermore, in a parent-child merging, we should check if the parent is an adapter hole and, if it is, delete it and check its parent. As a result, all adapter holes between a child and its closest non-adapter hole ancestor are deleted during the merge. A sibling-sibling merge works the same way as in STHoles except that only leaf nodes are considered (for the reason explained above). No special care is needed to handle adapter holes for sibling-sibling merges, since only leaf nodes are merged and an adapter hole cannot be a leaf node.

### 3.2.5 Build and Refine

Figure 20 describes a high level algorithm of STHoles+, from building the initial histogram to refining it with the addition of new queries. The algorithm shown here is the same as that shown in the STHoles paper [4] except for the following three differences. First, DrillHole is the new algorithm shown in Figure 19. Second, merging is done the way described above in the bucket merging part of Section 3.2.4. Third, unlike STHoles the root bucket is never expanded, since expanding the root bucket would change the resolution of its grid lines, which then would require all its children to be re-aligned to the new grid lines. This would

have a cascading effect on the children of its children, etc. To prevent the need to expand the root bucket, we normalize both queries and data space to the range of [0, 1] in each dimension of the multidimensional space. For instance, in a 3-dimensional space, the closest (to the origin) corner and the farthest (from the origin) corner of the root bucket are located at the coordinates $\langle 0, 0, 0 \rangle$ and $\langle 1, 1, 1 \rangle$, respectively.

```
BuildAndRefine(H: STHoles+, D: Data Set, W: Workload)
1.   Initialize H with an empty histogram or an existing histogram
2.   for each query q ∈ W do
3.     for all buckets b_i ∈ H
4.       compute the number of tuples in q ∩ b_i and store the result in T_{b_i}
5.     end for
6.     for each bucket b_i such that q ∩ b_i ≠ 0 do
7.       // Shrink the candidate hole c using the same algorithm used in STHoles
8.       (c_i, T_{c_i}) = Shrink(b_i, q, T_{b_i})
9.       if the estimated selectivity of c_i ≠ T_{c_i} then
10.            DrillHole(b_i, c_i, Tc_i) // using the algorithm shown in Figure 19
11.      end for
12.      while H has too many buckets, merge the leaf buckets with the lowest penalty in
H.
13.  end for
```

Figure 20: BuildAndRefine algorithm of STHoles+.

# 4 Experiments

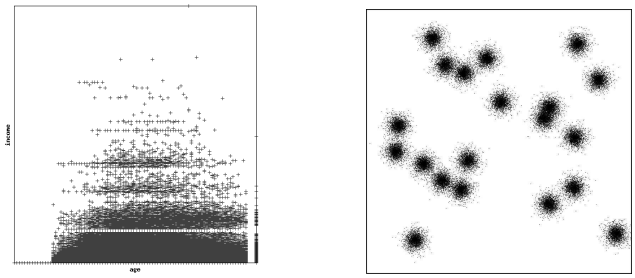
We have conducted experiments to compare the selectivity estimation accuracy of STHoles+ against that of STHoles with a focus on the following factors: query distribution, memory limit, quantization resolution, and data space dimensionality.

## 4.1 Experiment Setup

### 4.1.1 Data Sets

We use both real data sets and synthetic data sets. The real data sets are generated from 500,000 tuples sampled from the Year 2000 U.S. Census Bureau data. Using these sample tuples, we have generated two real data sets. One is for a two-dimensional data space defined by the attributes *Age* and *Income*; the other is for a three-dimensional data space defined by the attributes *Age*, *Income*, and *Weight*. We refer to the former as *Census2D* and the latter as *Census3D*. Figure 21(a) depicts the Census2D data with *Age* on the horizontal axis and *Income* on the vertical axis. These real data sets are very similar to those used in [4], but there are still significant differences. These differences cause the STHoles performances obtained in our experiments to be quite different from those reported in [4]. (See Appendix A for details.)

The synthetic data sets are populated with a *multi-Gaussian* distribution. This distribution simulates non-uniform real data and consists of a number of overlapping Gaussian bells, each centered on its own peak and containing the same number of tuples. Table 2 summarizes the parameters of the distribution: dimensionality of the data space ($d$), number of tuples in the data set ($N$), domain of the attribute defining the data space ($R$), number of Gaussian bells ($p$), and standard deviation of each Gaussian distribution ($\sigma$). Using these parameter values, we have generated data sets with dimensionality from two to eight. Among these, only the two-dimensional and three-dimensional data sets are used in all experiments except the one



(a) Census2D. (b) Gauss2D.

Figure 21: Data sets.

| Parameter Description | Value |
|---|---|
| $d$: Data space dimensionality | 2 - 8 |
| $N$: Data set cardinality | 500,000 |
| $R$: Data attribute domain | $[0, 1)^d$ |
| $p$: Number of Gaussian bells | 20 |
| $\sigma$: Gaussian standard deviation | 0.025 |

Table 2: Data set parameters for the multi-Gaussian distribution.

for varying the dimensionality of data space. We refer to the two-dimensional data set as *Gauss2D* and the three-dimensional one as *Gauss3D*. Figure 21(b) shows the Gauss2D data set.

#### 4.1.2 Query Workloads

We use two types of query workloads. One type, called *data-centered*, is created by choosing the query centers from the same distribution as the data set. This results in queries that "follow the data pattern," which simulates users querying regions where tuples are expected to exist—for example, *Age* between 22 and 55 and *Income* between \$20,000 and \$200,000. The other type, called *uniform*, is created by choosing the query centers from the uniform distribution. In both types, the volume of the query rectangle is set to 1% of the volume of the entire data space. This is the same size as that used in [4] for STHoles.

#### 4.1.3 Histogram Settings

Unless stated otherwise, histograms are given 1024 bytes of memory. STHoles+ uses the quantization resolution of 256 by default. For each experiment, the first 1000 queries are used as a training set to initialize the histogram and, subsequently, a workload consisting of another 1000 queries is used to measure the accuracy of the selectivity estimation.

#### 4.1.4 Performance Metrics

Given a query workload $W$, we use the *normalized absolute error* as the measure of selectivity estimation accuracy, calculated as

$$\frac{\sum_{q \in W} |est(H, q) - act(D, q)|}{\sum_{q \in W} |est_{unif}(q) - act(D, q)|} \tag{6}$$

where $est(H, q)$ denotes the selectivity estimated using the histogram $H$ for a range query $q$ in $W$, and $act(D, q)$ denotes the actual selectivity observed for the range query $q$ on the data set $D$, and $est_{unif}(q)$ is the selectivity estimated for the range query $q$ assuming that the entire data space has a uniform tuple density. That is,

$$est_{unif}(q) = \frac{v(q)}{v(D)} \tag{7}$$

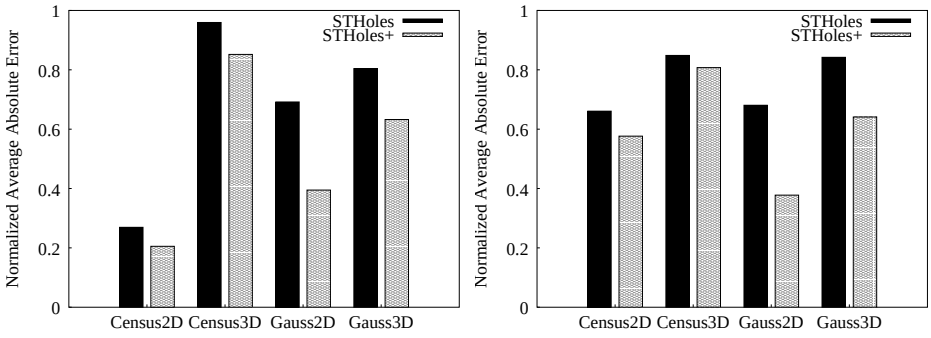
where $v(q)$ and $v(D)$ are the volumes of $q$ and $D$, respectively. Note that relative errors are not suitable because they are easily biased by data with small values; absolute errors are not suitable, either, because they vary greatly for different data sets while we compare the errors for different data sets in our experiments.

## 4.2 Experiment Results

In the first experiment, the query distribution is varied. In the second experiment, the memory limit is varied. In the third experiment, the quantization resolution is varied for STHoles+. In the fourth experiment, the dimensionality of data space is varied.

### 4.2.1 Experiment 1: Varying Query Distribution

Figure 22 shows the normalized absolute errors of STHoles and STHoles+ for data-centered and uniform query workloads. In the figure, STHoles+ outperforms STHoles for both types of workload across all real and synthetic data sets used. This stems from the ability of STHoles+ to store more buckets than STHoles in the same amount of memory and, thus, model the data distribution more accurately .




(a) Data-centered query distribution. (b) Uniform query distribution.

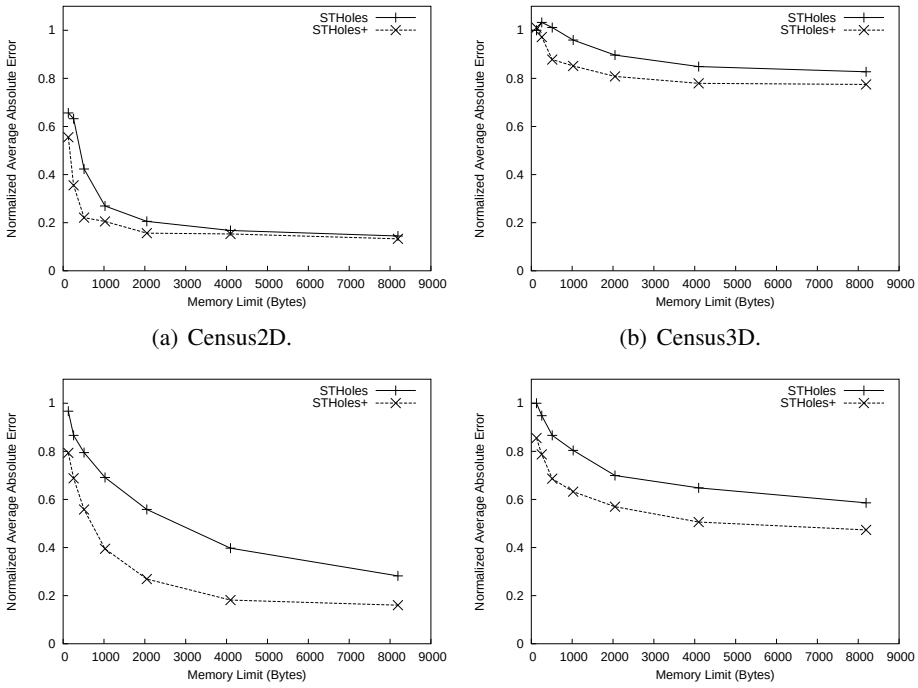
Figure 22: Normalized absolute error for varying query distribution.

The accuracy is higher for the data-centered query workload than for the uniform. The reason for this is as follows. In the case of a data-centered query workload, more queries occur in regions with tuples. This leads to more optimal refinement of the histogram and, thus, a more accurate modeling in these regions. Moreover, those regions have higher variance of tuple density (in the Gaussian bell shape) than the rest of the data space. More accurately modeling regions with higher variance of tuple density leads to higher selectivity estimation accuracy.

### 4.2.2 Experiment 2: Varying Memory Limit

Figure 23 shows the normalized absolute errors of STHoles and STHoles+ for the maximum memory limit varying from 128 bytes to 8192 bytes by a factor of two. The query workloads used are data-centered. In the figure, the errors decrease as the memory limit increases for both STHoles and STHoles+. This comes naturally from the increasing number of buckets that can be stored.

Moreover, the errors approach the minimum asymptotically and then stop decreasing as the memory limit increases beyond a certain amount. This indicates that the benefit of storing additional buckets decreases as more buckets are added. The reason for this is that, since queries are localized around data within certain spatial regions, their selectivity estimation can be done using only a small subset of buckets in the histogram. Adding more memory beyond the amount needed for those buckets does not contribute to the selectivity estimation accuracy so much.




(a) Census2D. (b) Census3D.

(c) Gauss2D. (d) Gauss3D.

Figure 23: Normalized absolute error for varying memory limit.

Between STHoles and STHoles+, STHoles+ outperforms STHoles by a larger margin as memory becomes more limited. This is because when the memory is more limited, the buckets STHoles+ can store in addition to those of STHoles are more likely to be the ones used for selectivity estimation. (Interestingly, STHoles+ does not outperform STHoles as much when the memory is "extremely" limited. This is because in that case there are too few buckets available for either algorithm to properly model the distribution of 50,000 point data sets.) By the same token, the performances of STHoles and STHoles+ become the same as sufficient memory becomes available for both.

### 4.2.3 Experiment 3: Varying Quantization Resolution

Figure 24 shows the normalized absolute error of STHoles+ and the average number of adaptor holes created by STHoles+ for the quantization resolution varying widely from 16 (i.e., $2^4$) to $2^{30}$. We show the normalized absolute error of STHoles using a flat line since it does not quantize bucket coordinates. The query workloads used are data-centered.

There are three objectives for conducting this experiment: (1) to observe the sensitivity of STHoles+'s performance to the changes in quantization resolution, (2) to observe the difference between the performance STHoles+ and STHoles with varying quantization resolution, and (3) to see the effect of adapter holes on selectivity estimation accuracy of STHoles+. Regarding the first objective, we observe that the performance

curve of STHoles+ stays nearly constant at the bottom for a large range of quantization resolution. This means that tuning the STHoles+ performance by varying quantization resolution is very easy. Regarding the second objective, we observe that STHoles+ outperforms STHoles for a wide range of quantization resolution and (not surprisingly) the performance advantage decreases as the quantization resolution increases. Regarding the third objective, by comparing the left column and the right column of the figure, we observe that selectivity estimation accuracy increases as the average number of adapter holes decreases.

STHoles+ performs worse than STHoles if the quantization resolution is "too low" (e.g., 16). This can be explained as follows. If the quantization resolution is too low, the grid spacing is too large and, therefore, too many adapter holes are created. Since adapter holes are not used for selectivity estimation but only consumes memory space, their existence degrades the performance by displacing buckets that could otherwise be used to estimate the selectivity more accurately. On the other hand, as the quantization resolution becomes large enough, the performance of STHoles+ eventually approaches that of STHoles. This is because the bucket size approaches that of STHoles and the number of adapter holes decreases.

#### 4.2.4 Experiment 4: Varying Dimensionality of Data Space

Figure 25 shows the normalized absolute errors of STHoles and STHoles+ for the dimensionality of data space varying from two to eight. Query workloads used for Figure 25(a) are data-centered, and those for Figure 25(b) are uniform. In the figure, the errors increase with the dimensionality for both STHoles and STHoles+. This is because of the increasing size of buckets due to an increasing number of coordinate components, which decreases the number of buckets that can be stored within the same amount of memory.

Between STHoles and STHoles+, STHoles+ always performs better than or the same as STHoles. This comes naturally from the smaller bucket size of STHoles+, which leads to storing more buckets than STHoles.

## 5 Conclusion

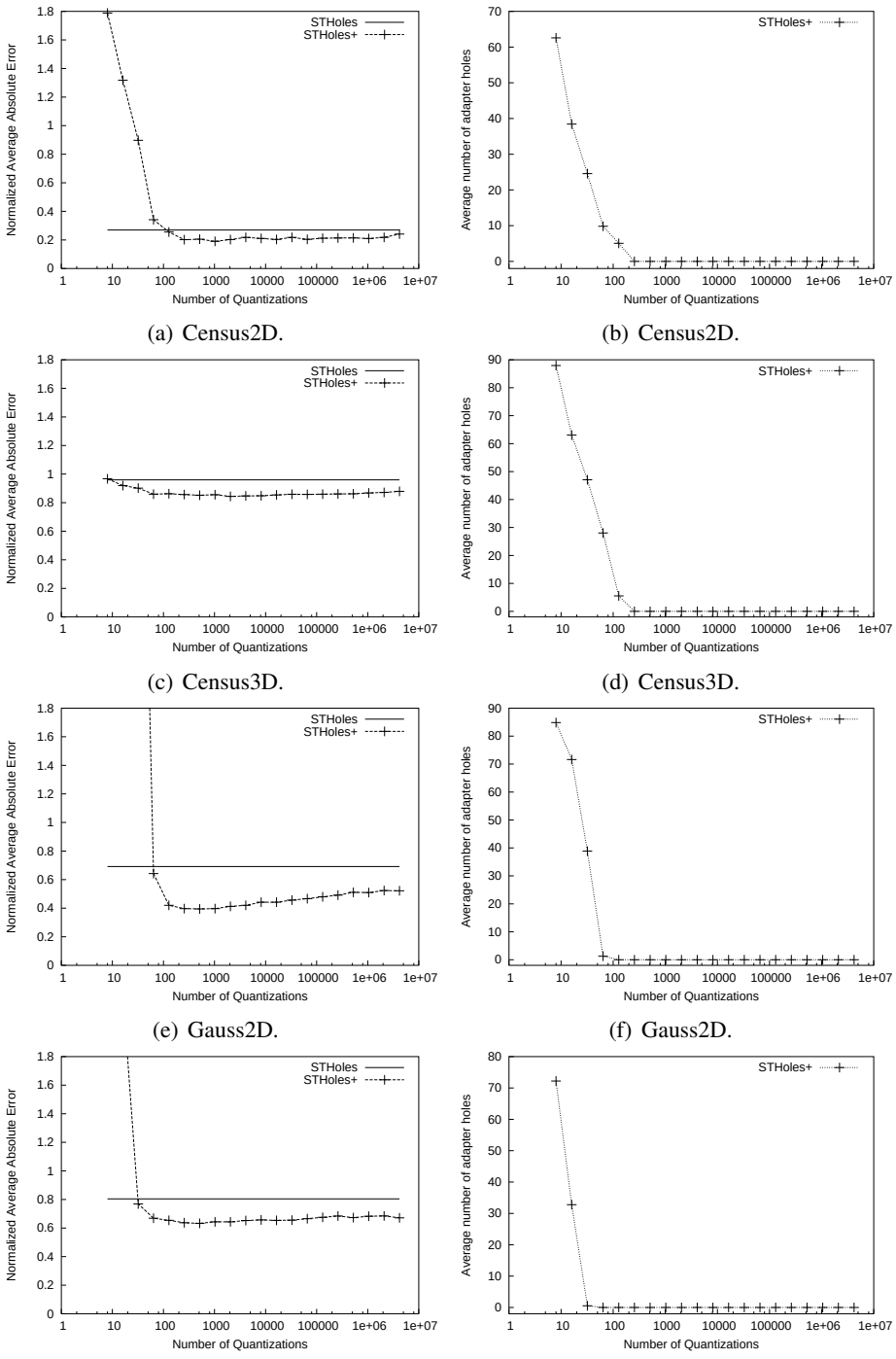
### 5.1 Summary

In this paper, we have proposed bucket location compression as an approach for building and maintaining more memory-efficient arbitrary layout histograms. To demonstrate the effectiveness of this approach, we have presented STHoles+, the first multidimensional histogram that uses bucket location compression to improve the memory-efficiency of histogram buckets. STHoles+ extends the state-of-the-art STHoles by using quantized relative coordinates to compress bucket coordinates. As a result, STHoles+ stores more buckets than STHoles within the same amount of memory. Thus, more information about the data distribution can be retained, and consequently the selectivity estimation accuracy is increased.

Experimental results confirm that STHoles+ achieves higher accuracy (up to 30%) than STHoles for varying query distribution, memory limit, quantization resolution, and data space dimensionality. From these experimental results we conclude that STHoles+ is preferable to STHoles for accurate selectivity estimation.

### 5.2 Future Work

We plan to investigate techniques for bucket location compression for GenHist. Another direction of future work is to further improve the selectivity estimation accuracy of STHoles+ by taking changing query patterns into account when refining the histogram. We plan to address this issue by introducing a *bucket aging* scheme. In this scheme, a flag is initially set on every bucket created. The flag is then cleared when the bucket overlaps a query and, thus, is used to estimate the selectivity of the query. At the time of calculating



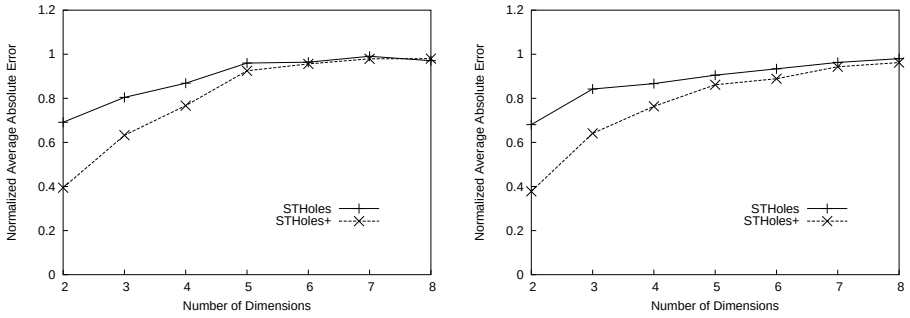

(a) Census2D. (b) Census2D.

(c) Census3D. (d) Census3D.

(e) Gauss2D. (f) Gauss2D.

(g) Gauss3D. (h) Gauss3D.

Figure 24: Normalized absolute error (left column) and the number of adapter holes created by STHoles+ (right column) for varying quantization resolution.




(a) Data-centered query distribution. (b) Uniform query distribution.

Figure 25: Normalized absolute error for varying dimensionality of data space.

the merge penalty, the merge penalty is scaled down for buckets on which the flag is set (because the buckets have not been used recently). This way, the histogram stores more distribution information of data in more recently queried regions of the data space, thereby using the limited memory more efficiently.

## Acknowledgments

We thank X. Sean Wang for his feedback on the experimental results. We owe our special thanks to the anonymous reviewers whose comments were critical to enhancing the quality of the paper. This research has been supported by the US National Science Foundation through Grant No. IIS-0415023 and the US Department of Energy through Grant No. DE-FG02-ER45962.

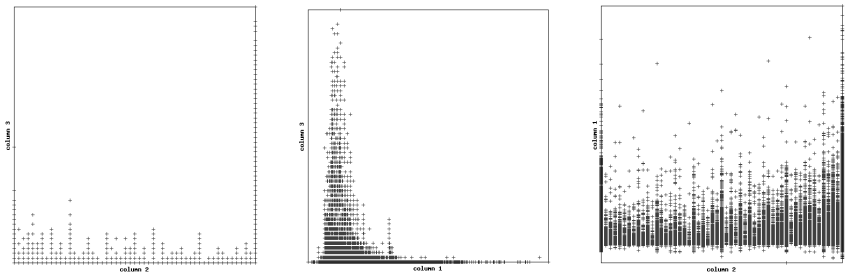
## Appendix

## A Reason for the STHoles Performance Discrepancies

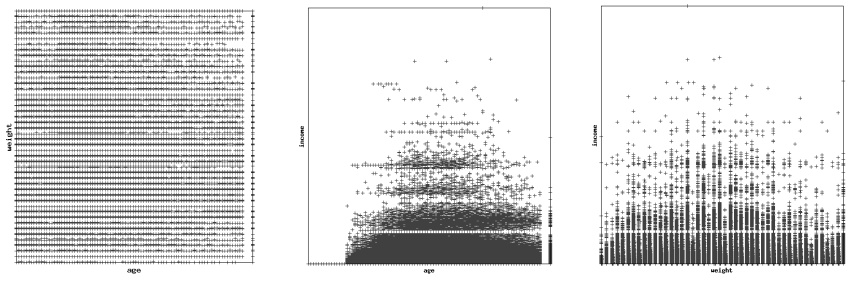
We explain in this section why the performances of STHoles in our experiments are worse than those reported in [4]. For our investigation, we have obtained the Census data sets (Census2D and Census3D) used in [4]. (The metadata, including the attributes of data points, have not been provided.) Then, we have found that our data sets have two main differences from the data sets used in [4]: data distribution and data set size. Conclusion of our investigation is that the difference in data distribution, particularly the correlation among attributes, is the main cause. We now explain each of the two main differences further.

*Data distribution*: Although the exact attributes of data sets are unknown to us, we are able to match the attributes to those of our data sets. The plots of data points in Figure 26 show that there are higher correlations among attributes in the data sets used in [4]. (Note that our data sets cover a larger region of the data space.) If attributes are more correlated, a smaller number of buckets placed at the right places can reduce the error significantly. Conversely, if the attributes are less correlated, then more buckets are needed to achieve the same performance. Therefore, naturally the performances reported in [4] are better than those reported in this paper.

*Data set size*: The sizes of data sets used in [4] are 145,917 data points for Census2D and 165,428 data points for Census3D, in contrast to 500,000 in our experiments. We have done the same experiments using 50,000 data points, which is one-tenth of 500,000, and observed that the performance discrepancies are similar regardless of the data set size. Thus, data set size is not likely to be a cause.




(a) Each pair of the three attributes of Census3D used in [4].




age × weight age × income weight × income

(b) Each pair of the three attributes of Census3D used in this paper.

Figure 26: Distributions of real data sets used in [4] and in this paper.

# References

[1] Y. Ioannidis, S. Christodoulakis, On the propagation of errors in the size of join results, in: Proc. of ACM SIGMOD, ACM Press, Denver, CO, 1991, pp. 268–277.

[2] A. Aboulnaga, S. Chaudhuri, Self-tuning histograms: Building histograms without looking at data, in: Proc. of ACM SIGMOD, ACM Press, Philadelphia, PA, 1999, pp. 181–192.

[3] S. Acharya, V. Poosala, S. Ramaswamy, Selectivity estimation in spatial databases, in: Proc. of ACM SIGMOD, ACM Press, Philadelphia, PA, 1999, pp. 13–24.

[4] N. Bruno, S. Chaudhuri, L. Gravano, STHoles: A multidimensional workload-aware histogram, in: Proc. of ACM SIGMOD, ACM Press, Santa Barbara, CA, 2001, pp. 211–222.

[5] C. Chen, N. Roussopoulos, Adaptive selectivity estimation using query feedback, in: Proc. of ACM SIGMOD, ACM Press, Minneapolis, MN, 1994, pp. 161–172.

[6] A. Deshpande, M. Garofalakis, R. Rastogi, Independence is good: dependency-based histogram synopses for high-dimensional data, in: Proc. of ACM SIGMOD, ACM Press, Santa Barbara, CA, 2001, pp. 199–210.

[7] L. Getoor, B. Taskar, D. Koller, Selectivity estimation using probabilistic models, in: Proc. of ACM SIGMOD, ACM Press, Santa Barbara, CA, 2001, pp. 461–472.

[8] P. Gibbons, Y. Matias, V. Poosala, Fast incremental maintenance of approximate histograms, ACM Transactions on Database Systems 27 (3) (2002) 261–298.

[9] D. Gunopulos, G. Kollios, V. Tsotras, C. Domeniconi, Approximating multi-dimensional aggregate range queries over real attributes, in: Proc. of ACM SIGMOD, ACM Press, Dallas, TX, 2000, pp. 463–474.

[10] Y. Matias, J. Vitter, M. Wang, Wavelet-based histograms for selectivity estimation, in: Proc. of ACM SIGMOD 1998, ACM Press, Seattle, WA, 1998, pp. 448–459.

[11] Y. Matias, J. Vitter, M. Wang, Dynamic maintenance of wavelet-based histograms, in: Proc. of VLDB, Cairo, Egypt, Morgan Kaufmann, Cairo, Egypt, 2000, pp. 101–110.

[12] M. Muralikrishna, D. J. DeWitt, Equi-depth multidimensional histograms, in: Proc. of ACM SIGMOD, ACM Press, Chicago, IL, 1988, pp. 28–36.

[13] S. Muthukrishnan, V. Poosala, T. Suel, On rectangular partitionings in two dimensions: Algorithms, complexity, and applications, in: Proc. of ICDT, Springer-Verlag, London, UK, 1999, pp. 236–256.

[14] V. Poosala, Y. Ioannidis, Selectivity estimation without the attribute value independence assumption, in: Proc. of VLDB, Morgan Kaufmann, Athens, Greece, 1997, pp. 486–495.

[15] N. Thaper, S. Guha, P. Indyk, N. Koudas, Dynamic multidimensional histograms, in: Proc. of ACM SIGMOD, ACM Press, Madison, WI, 2002, pp. 428–439.

[16] Y. Wu, D. Agrawal, A. El Abbadi, Applying the golden rule of sampling for query estimation, in: Proc. of ACM SIGMOD, ACM Press, Santa Barbara, CA, 2001, pp. 449–460.

[17] R. Lipton, J. Naughton, Query size estimation by adaptive sampling, in: Proc. of ACM SIGACT-SIGMOD-SIGART Symp. PODS, ACM Press, Nashville, TN, 1990, pp. 40–46.

[18] Y. Sakurai, M. Yoshikawa, S. Uemura, H. Kojima, The A-tree: An index structure for high-dimensional spaces using relative approximation, in: Proc. of VLDB, Morgan Kaufmann, 2000, pp. 516–526.