STHoles: A Multidimensional Workload-Aware Histogram [1]

Nicolas Bruno
Columbia University
`nicolas@cs.columbia.edu`

Surajit Chaudhuri
Microsoft Research
`surajitc@microsoft.com`

Luis Gravano
Columbia University
`gravano@cs.columbia.edu`

Technical Report
MSR-TR-2001-36

Attributes of a relation are not typically independent. Multidimensional histograms can be an effective tool for accurate multiattribute query selectivity estimation. In this paper, we introduce *STHoles*, a "workload-aware" histogram that allows bucket nesting to capture data regions with reasonably uniform tuple density. *STHoles* histograms are built without examining the data sets, but rather by just analyzing query results. Buckets are allocated where needed the most as indicated by the workload, which leads to accurate query selectivity estimations. Our extensive experiments demonstrate that *STHoles* histograms consistently produce good selectivity estimates across synthetic and real-world data sets and across query workloads, and, in many cases, outperform the best multidimensional histogram techniques that require access to and processing of the full data sets during histogram construction.

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
`http://www.research.microsoft.com`

[1]A shorter version of this paper appears in the Proceedings of the 2001 ACM International Conference on Management of Data (SIGMOD'01)

1 Introduction

A variety of problems require succinct summary representations of large data sets. Histograms are an important example of such summary representation structures. In the database field, they are mainly used for selectivity estimation during query optimization [11, 4] and also for approximate query processing [13, 25] to give rough and fast responses to expensive queries. Query optimization in relational database systems has traditionally relied on single-attribute histograms to compute the selectivity of queries. For queries that involve multiple attributes, most database systems make the attribute value independence assumption, i.e., assume that $p(A1{=}v1, A2{=}v2) = p(A1{=}v1) \cdot p(A2{=}v2)$, which may of course lead to significant inaccuracy in selectivity estimation (e.g., see [26]).

An alternative to assuming attribute value independence is to use histograms over multiple attributes, which are generally referred to as *multidimensional histograms* [19, 26]. Ideally, multidimensional histograms should consist of *buckets* that enclose regions of the data domain with close-to-uniform tuple density, so they can accurately estimate the selectivity of range queries. At the same time, multidimensional histograms should be sufficiently compact and efficiently computable. Unfortunately, existing multidimensional histogram construction techniques fail to satisfy these requirements robustly across data distributions, as we show in this paper through a thorough experimental evaluation over synthetic and real-life data sets. A fundamental problem with many of these techniques is that they make bucket generation decisions based on *unidimensional* information, as we will discuss.

A key observation that we exploit in this paper is that we can build good quality histograms by exploiting workload information and query feedback. Typically, histogram construction strategies only inspect the data sets that they characterize, without considering how the histograms will be used. In particular, if the histograms are to help in query processing, the implicit assumption is that all queries are equally likely. This assumption is rarely true in practice, and certain data regions might be much more heavily queried than others. Intuitively, we will exploit query workload to zoom in and spend more resources in heavily accessed areas, thus allowing some inaccuracy in the rest. We will also exploit query feedback as truly multidimensional information to identify promising areas to enclose in histogram buckets. As a result, we will obtain a customized histogram that is more accurate for the expected workload than traditional workload-independent ones would be.

In this paper we present *STHoles*, a novel workload-aware histogram technique. This histogram identifies a novel partitioning strategy that is especially well suited to exploit workload information. We present algorithms that show how to exploit results of queries in the workload and gather associated statistics to progressively build and refine an *STHoles* histogram (Figure 1). Thus, our technique uses information about both the workload (range selection queries) and the data distribution itself (through statistics collected from query result streams). An important consequence of this refinement procedure is that our histograms can gracefully adapt to changes in the data distribution they approximate, without the need to periodically rebuild them. Our experiments strongly suggest that our approach results in a customized histogram that is robust across different data sets and workloads and in many cases results in more accurate estimations for the expected workload than those for the best workload-independent histogram construction techniques. Of course, it is inevitable that histograms built using only query feedback are susceptible to errors for queries that target unseen data regions. As we will see, these errors can be reduced by starting even with a coarse initial histogram. We also studied the overhead of the refinement procedure over Microsoft SQL Server 2000, and found that it slows down query execution by less than 10%, which is acceptable and can be regarded as an amortized cost we pay for the online construction of *STHoles* histograms.

The rest of the paper is structured as follows. Section 2 reviews related work. Section 3 describes existing multidimensional histogram techniques and motivates the introduction of the *STHoles* histograms that we then present in Section 4. Section 5 discusses some implementation details for building and refining *STHoles*

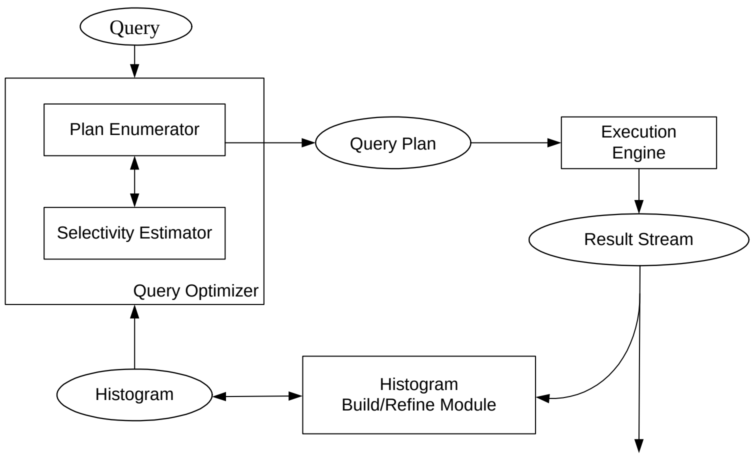


Figure 1: Workload-driven histogram construction.

histograms. Finally, Sections 6 and 7 report an extensive experimental evaluation of the new histograms using both real and synthetic data sets and a variety of query workloads.

2 Related Work

Several techniques exist in the literature to compute selectivity estimators of multidimensional data sets. These techniques include wavelets (e.g., [17]) and discrete cosine transformations (e.g., [15]), sampling (e.g., [22]), and multidimensional histograms. The focus of this paper is on multidimensional histograms, which have been the topic of much theoretical and experimental work in the last few years. A conceptually interesting class of histograms is the *V-optimal(f,f)* family [27], which groups contiguous sets of *frequencies* into buckets and minimizes the variance of the overall frequency approximation. These histograms are optimal for estimating the result size of equality join and selection queries under a definition of optimality that captures the average error over all possible queries and databases [12]. However, these histograms need to record *every* distinct attribute value inside each bucket, which is clearly impractical and makes these techniques be only of theoretical interest. Moreover, the construction algorithm involves an exhaustive and exponential enumeration of all possible histograms. A more practical approach is to restrict the attention to *V-optimal(v,f)* histograms, which group contiguous sets of *values* into buckets, minimizing the variance of the overall frequency approximation. A dynamic programming algorithm is presented in [14] for building *unidimensional V-optimal(v,f)* histograms in $O(N^2b)$ time, where N is the number of tuples in the data set and b is the number of buckets. Unfortunately, it can be shown [20] that even for two-dimensional data sets, building the *V-optimal(v,f)* histogram using arbitrary rectangular buckets is NP-Hard. Therefore, practical static multidimensional histogram techniques use heuristics to partition the data space into buckets, as discussed below.

A multidimensional version of the *EquiDepth* histogram [24] presented in [19] recursively partitions the data domain, *one dimension at a time*, into buckets enclosing the same number of tuples. Reference [26] introduced *MHist* based on underlying *MaxDiff(v,a)* histograms [27]. The main idea is to iteratively partition the data domain using a greedy procedure. At each step, *MaxDiff(v,a)* analyzes *unidimensional* projections of the data set and identifies the bucket in most need of partitioning. Such a bucket will have the largest “area gap” [27] between two consecutive values along one dimension. Using this information, *MHist* iteratively splits buckets until it reaches the desired number of buckets. Recently, reference [9] introduced *GenHist* histograms, which allow unrestricted overlap among buckets. If more than two buckets overlap, the density of tuples in their intersection is approximated as the sum of the data densities of the overlapping buckets. For the technique to work, a tuple that lies in the intersection of many buckets is counted in only one of them (chosen probabilistically). The main idea is to construct progressively coarser grids over the data set, convert the densest cells into buckets of the histogram, and remove a certain percentage of tuples in those cells to make the resulting distribution smoother. In contrast, our histograms allow only a restricted kind of bucket overlap that is less expensive to manage, which will be crucial to exploit workload refinement.

All the histogram techniques above are *static* in the sense that, after histograms are built, their buckets and frequencies remain fixed regardless of any changes in the data distribution. Histograms are typically rebuilt or reorganized if the number of data set updates or a certain inaccuracy threshold is exceeded. References [6] and [8] are examples of reorganization strategies for unidimensional histograms. As we will see in Section 3, partitioning multidimensional spaces is challenging, and there are no obvious generalizations of these techniques for more than one dimension.

The idea of using feedback from the query execution engine is introduced in [5]. Their approach is to represent the data distribution as a linear combination of model functions. The weighting coefficients of this linear combination are adjusted using feedback information and a least squares technique. The main problem with this approach is that it depends on the choice of the "model" functions, and moreover, it assumes that the data follows some smooth and known distribution, which is not the case for arbitrary data sets.

Reference [1] presents the first multidimensional histogram that uses query feedback to refine buckets, and shares some ideas with our work. We refer to this technique as *STGrid* histograms in this paper. ("ST" stands for "self tuning.") An *STGrid* histogram greedily partitions the data domain into disjoint buckets that form a grid, and refines their frequencies using query feedback. After a predetermined number of queries, the histogram is restructured by merging and splitting *rows* of buckets at a time (to preserve the grid structure). Efficiency in histogram tuning is the main goal of this technique, at the expense of accuracy. Since *STGrid* histograms need to maintain the grid structure at all times, and due to the greedy nature of the technique, some *locally* beneficial splits and merges have the side effect of modifying distant and unrelated regions, hence decreasing the overall accuracy. For that reason, the resulting histograms are generally less accurate than their static counterparts and are not robust across different data distributions. In contrast, our new technique introduces a new histogram structure with bucket nesting that results in better accuracy than standard "flat" histograms, at the expense of slightly higher execution overhead.

Multidimensional histogram construction shares some intriguing features with multidimensional access method construction. Multidimensional access methods [7] support efficient search operations in spatial databases. They partition the data domain into buckets, and assign to each bucket some information about the tuples it covers (usually the set of *rid*s). There is a connection between access methods and histogram techniques regarding the different ways in which they partition the data domain. For instance, the partitioning strategy used in *STGrid* histograms is similar to that of the Grid File technique [21]. *MHist* histograms share the hierarchical or recursive partitioning of K-D-B Trees [28]. Our proposed *STHoles* technique uses a similar partitioning scheme to hB-Tree's holey bricks [16]. Finally, *GenHist* histograms use overlapping buckets, just as R-Trees [10] do. A natural question arises then. Why not use the "shell"[1] of the existing access methods directly to model density distributions of data sets? This idea is explored for example in the CONTROL project [2], which uses the shell of R-Trees to provide online visualization of large data sets, by traversing the R-tree breadth-first and approximating the underlying data distribution with the aggregated information at each level. In spite of these connections between histograms and access methods, we believe that there are fundamental differences between the two. The main goal of multidimensional access methods is to allow efficient access to each "bucket" using only a few disk accesses, so the main objective is to distribute tuples

[1]That is, we discard tuple-level information and some internal nodes so we do not exceed the available storage for the "histogram," and replace these new "leaves" with aggregate information.

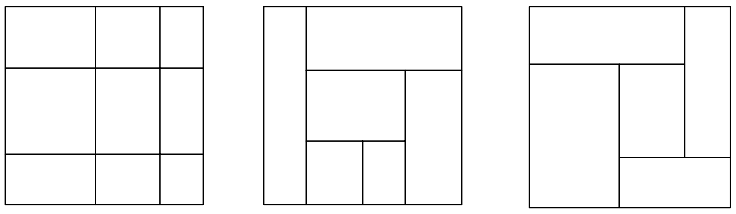

(a) Grid **(b)** Recursive **(c)** Arbitrary

Figure 2: Partitioning schemes for building multidimensional histograms.

evenly among buckets and maintain a high fraction of bucket utilization to prevent long searches. On the other hand, histogram techniques need to form buckets enclosing areas of uniform tuple density whenever possible. Furthermore, since histograms store aggregated information at the bucket level, the number of tuples covered by each bucket need not be uniform across buckets.

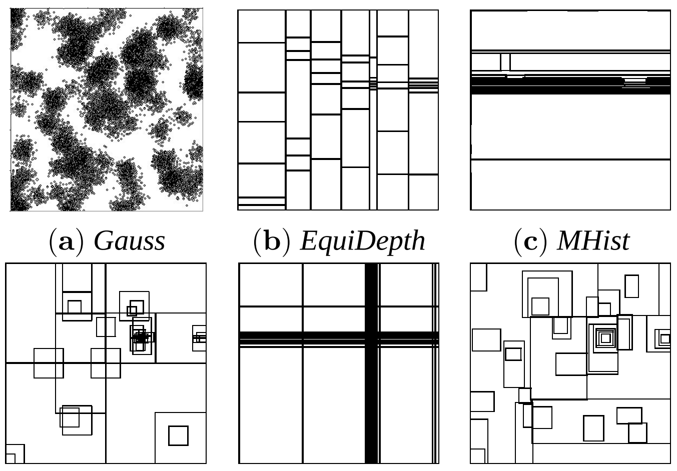
3 Analysis of Existing Multidimensional Histograms

Good histograms partition data sets into "smooth" buckets with close-to-uniform internal tuple density. In other words, the frequency variance of the tuples enclosed by such buckets is minimized, leading to accurate selectivity estimations for range queries. Unfortunately, current multidimensional histogram techniques do not always manage to produce close-to-uniform partitions of the data sets, as we discuss next. Later, Section 7 reports a thorough experimental evaluation of these techniques that complements the discussion in this section.

A partition of a multidimensional data domain results in a set of disjoint rectangular buckets that cover all the points in the domain and assigns to each bucket some aggregated information, usually the number of tuples enclosed. The choice of *rectangular* buckets is justified by two main reasons: First, rectangular buckets make it easy and efficient to intersect each bucket and a given range query to estimate selectivities. Second, rectangular buckets can be represented concisely, which allows a large number of buckets to be stored using the given budget constraints. Reference [20] presents a taxonomy of partitioning schemes for building multidimensional histograms, which we illustrate in Figure 2. In the *grid* partitioning scheme (Figure 2(a)), each dimension d_i is divided into p_i disjoint intervals, which induce a grid of $\prod_i p_i$ buckets. A *recursive* partition (Figure 2(b)) starts with one bucket covering the whole domain, and repeatedly divides some existing bucket in two along some dimension. Finally, the *arbitrary* partition scheme (Figure 2(c)) imposes no restrictions on the arrangement of buckets. In principle, all the schemes are equivalent in the sense that we can simulate any partition that follows one scheme with the others (possibly using more buckets). We say that each partitioning scheme in Figure 2 is *more flexible* than those to its left, since we can simulate any partition following the more flexible scheme with a partition that follows the others using at most the same number of buckets, but not vice-versa.

Consider Figures 3 and 4, which show different histograms built for a multigaussian data distribution and a two-dimensional projection of US Bureau Census data [3], respectively [2]. All histograms use the same amount of memory (250 bytes), which in turn results in different numbers of buckets allocated to each histogram, since the amount of information needed to describe each bucket is different across histograms. As we can see in Figures 3(b) and 4(b), *EquiDepth* histograms correctly identify the core of the densest tuple clusters in the two data sets. However, the partitioning of the rest of the domain is problematic. For example, the histogram in Figure 4(b) contains long buckets that touch the boundaries of the main tuple cluster and stretch all the way to regions having almost no tuples. As a result, the tuple density inside each of these buckets is far from uniform.

[2]See Section 6 for more details on the data distributions.



(**a**) *Gauss* (**b**) *EquiDepth* (**c**) *MHist*

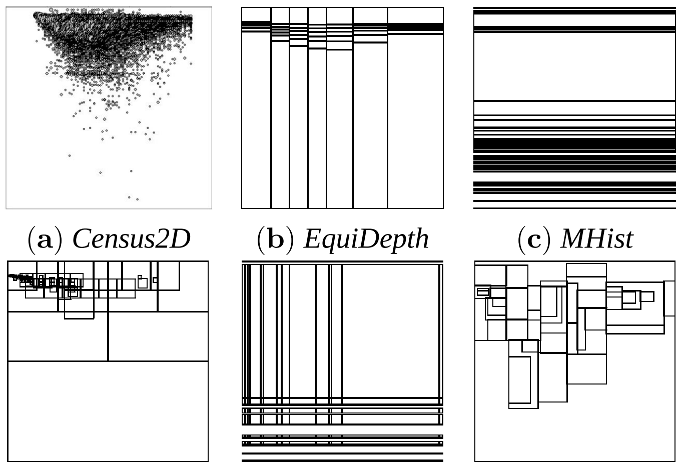
(**d**) *GenHist* (**e**) *STGrid* (**f**) *STHoles*

Figure 3: *Gauss* data set and histograms.

Figure 3(b) shows a similar phenomenon. The reason these poor buckets are generated is that the partition strategy for *EquiDepth* histograms is guided by the wrong principle, that is, instead of capturing buckets with almost uniform tuple density, *EquiDepth* buckets have all the same number of tuples. Consequently, *EquiDepth* buckets mix regions with very different tuple densities and therefore exhibit inadequate buckets around cluster boundaries.

MHist histograms try to solve some of the problems with *EquiDepth* histograms for cases when the data distribution is highly skewed. Figures 3(c) and 4(c) show the *MHist* histograms for our two data sets using *MaxDiff(v,a)* as the underlying unidimensional partitioning strategy [27]. As we can see in Figure 3(c), the *MHist* histogram devotes too many buckets to the densest tuple clusters, and almost none to the rest of the data domain, which degrades the overall histogram accuracy. This problem arises from the way in which *MHist* recursively partitions the data set. At each step, *MHist* assigns scores to each bucket-dimension pair by analyzing *unidimensional* projections of the original data set, and chooses the pair in most need of partitioning as the one with the highest score. Unfortunately, the scores for each bucket-dimension pair are absolute numbers that depend only on the underlying unidimensional histogram *MaxDiff(v,a)*. When deciding where to partition in each step, it is crucial to include some information about the total number of tuples, the total volume of the bucket, or even the shape of the bucket. Otherwise, as the example illustrates, some bad initial partitioning choices are amplified in later steps. In particular, almost all partitions are done along the same dimension, leading to "thin" non-uniform buckets. The *MHist* histogram in Figure 4(c) presents a new anomaly. The underlying *MaxDiff(v,a)* histogram dictates to partition along the position that lies between the pair of values with the largest difference in area (defined as frequency multiplied by spread [27]). This data set has different ranges of values along different dimensions. For instance, the *Age* attribute has values in the $[15\ldots95]$ range, and the *Income* attribute has values in the $[-15,000\ldots330,000]$ range. Therefore, although the lower half of the data distribution is virtually empty, the difference between consecutive values along the *Income* attribute is high enough to cause several bucket boundaries to lie in that region. This problem cannot be easily solved by simply normalizing the values along all dimensions once, because after a few splits the resulting buckets will start to exhibit this behavior (i.e., non "normalized" ranges) again.

GenHist histograms are introduced in [9] to address some of the problems outlined above. They find buckets of variable size and allow unrestricted overlap among buckets. Successively coarser grids are built over the data set and the densest cells are converted to buckets. Figures 3(d) and 4(d) show the *GenHist* histograms for the data set in Figure 3(a). A drawback of this technique is the difficulty of choosing the right values for a


(**a**) *Census2D* (**b**) *EquiDepth* (**c**) *MHist*

(**d**) *GenHist* (**e**) *STGrid* (**f**) *STHoles*

Figure 4: *Census2D* data set and histograms.

crucial number of parameters. Specifically, the initial grid size and the number of buckets created per iteration are given values in [9] that are independent of the data set. As we will see in Section 7, this technique generally results in better accuracy than the techniques discussed above that make bucket generation decisions based only on unidimensional information. However, this uniform parameter setting produces degraded performance in some cases. Another drawback of this technique is that it requires multiple passes (at least 5 to 10) over the whole data set [9].

Another promising direction to capture multidimensional areas with close-to-uniform tuple density and address the problems explained above is to incorporate *workload information* and *feedback from query execution* to progressively refine the histogram buckets. In this way, we could detect buckets that do not have uniform density and "split" them into smaller and more accurate buckets, or realize that some adjacent buckets are too similar and "merge" them, thus recuperating space for more critical regions. *STGrid* histograms use query workloads to refine a grid-based histogram structure. Figures 3(e) and 4(e) show the *STGrid* histograms for our data set when the query workload used for refinement consists of range queries that follow the data distribution. Although using workload information helps make the histogram more accurate, the figures show that workload alone is not powerful enough to get good results, since the grid partitioning strategy is still too rigid. Data distributions generally contain clusters or sub-regions with similar density, which we would like to capture using as few buckets as possible. However, the *STGrid* partitioning scheme results generally in many not-so-useful buckets. In particular, the split (merge) of each bucket entails the splitting (merging) of several other buckets that could be far away from and unrelated to the original one, just to restore the grid partitioning constraint. Besides, as *STGrid* histograms are also based on *MaxDiff(v,a)* unidimensional histograms, the problems discussed for *MHist* histograms also apply in this case.

To avoid the poor bucket layout problems of *STGrid* histograms and still use query workloads to refine histograms, we introduce a new partitioning scheme for building multidimensional histograms that allows buckets to overlap. Specifically, we will allow inclusion relationships, i.e., some buckets can be *completely* included inside others. This way, we implicitly relax the requirement of rectangular regions while keeping rectangular bucket structures. By allowing bucket nesting, the resulting histograms do not suffer from the problems outlined above and can model complex shapes (not restricted to rectangles anymore); by restricting the way in which buckets may overlap, the resulting histograms can be efficiently created and updated incrementally by using workload information. In contrast to multidimensional histogram techniques that use unidimensional projections of the data set for bucket creation, *STHoles* exploits query feedback in a truly multidimensional way to improve the quality of the resulting histograms. Figures 3(f) and 4(f) show *STHoles* histograms in which nested buckets capture naturally regions that exhibit varying tuple density.

4 STHoles Histograms

We now describe the general structure of *STHoles* histograms (Section 4.1). Then, we introduce the various algorithms needed for constructing the new histograms (Section 4.2).

4.1 Histogram Definition

As explained in the previous section, the inclusion relationship among buckets provides an extra degree of flexibility compared to partitioning schemes that use disjoint buckets. Each bucket b in an *STHoles* histogram is composed of a rectangular bounding box, denoted $box(b)$, and a real valued frequency, denoted $f(b)$, which indicates the number of tuples enclosed by bucket b. In a traditional histogram (see Section 2), a bucket b would be "solid," with no "holes," and hence the region that b covers would be regarded as having uniform tuple density. In contrast, an *STHoles* histogram identifies sub-regions of b with different tuple density and "pulls" them out from b. Hence a bucket b can have *holes*, which are themselves first-class histogram buckets. These holes are bucket b's *children*, and their bounding boxes are disjoint and completely enclosed in b's bounding box [3]. Therefore, an *STHoles* histogram can be conceptually seen as a tree structure, where each node represents a bucket.

The volume of bucket b is defined as $v(b) = \textit{vBox}(b) - \sum_{b' \in \, children(b)} \textit{vBox}(b')$, where $\textit{vBox}(b)$ is the volume of $box(b)$. Given a histogram H over a data set D, and a range query q, the estimated number of D tuples that lie inside q, $est(H, q)$, is:

$$est(H,q) = \sum_{b \in H} f(b) \frac{v(q \cap b)}{v(b)}$$

where $v(q \cap b)$ denotes the volume of the intersection of q and b (not $box(b)$). In the next sections we introduce the algorithms used to build and refine *STHoles* histograms.

Example 1: *Figure 5 shows a histogram with four buckets. The root of the tree is bucket b_1, with frequency 100. It has two children, namely buckets b_2 and b_3, with frequencies 500 and 1,000, respectively. Finally, bucket b_3 has one child, b_4, with frequency 200. The region associated with a particular bucket excludes that of its descendants, which can be thought of as "holes" in the parent space. Note that the region modeled by bucket b_1 (shaded in Figure 5) is not rectangular, even though we only use rectangular buckets for partitioning the space. A query that covers the lower half of bucket b_3 will be estimated to return nearly 1,000 tuples, even when it covers half of b_3's bounding box, because the other half is not considered as part of b_3. More precisely, there is another bucket (b_4) that covers that region.* ∎

4.2 Histogram Construction

A key idea for building *STHoles* histograms is to intercept the result of queries in the workload and efficiently gather some simple statistics over them to progressively refine the layout and frequency of the existing buckets. This way, the regions that are more heavily queried will benefit from having more buckets with finer

[3]Note that an alternative design could add the frequency of a bucket's descendants to the frequency of the bucket proper. It is easy to see that this alternative design conveys exactly the same information as our *STHoles* histograms do.

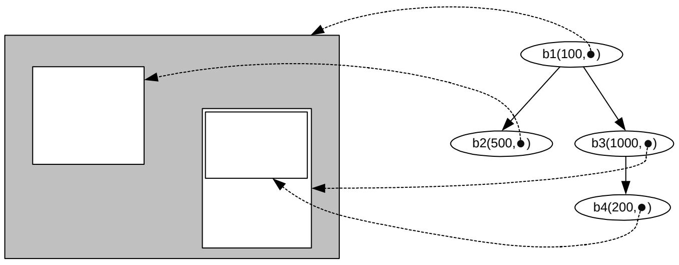


Figure 5: A four-bucket *STHoles* histogram.

granularity. To build an *STHoles* histogram, we start with an empty histogram that contains no buckets. Alternatively, if we have more information about the data distribution, e.g., the total number of tuples in the data set and the maximum and minimum value for each attribute [4], we can start with a single-bucket histogram. More generally, we can use an existing histogram and start with a more accurate model of the data set (see Section 7 for more details). For our experiments, we assume we know nothing about the data set and therefore we start with an empty histogram. (However, for completeness we ran all experiments in Section 7 using the simple variations above and obtained similar results.)

After we set up the initial histogram, for each query q in the workload we intercept the result stream and count how many tuples lie inside each bucket of the current histogram. If the current query q extends beyond the boundaries of the root bucket (or when considering the first query) we expand the bounding box of the root bucket so that it covers q. Then we determine which regions in the data domain can benefit from using this new information (Section 4.2.1), and refine the histogram by "drilling holes," or zooming into the buckets that cover the query region (Section 4.2.2). Finally, we consolidate the resulting histogram by merging similar buckets so that we do not exceed our fixed storage budget (Section 4.2.3). These high level steps are summarized below:

```
BuildAndRefine (H: STHoles, D: Data Set, W: Workload)
Initialize H with no buckets (empty histogram).
// Or use an existing histogram if available.
for each query $q \in$ W do
   Gather statistics from $q \cap b_i$ $\forall$ buckets $b_i$ in H.
   Identify candidate holes in H (Section 4.2.1).
   Drill candidate holes as new buckets in H (Section 4.2.2).
   Merge superfluous buckets in H (Section 4.2.3).
```

4.2.1 Identifying Candidate Holes

In this section we show how we can use the results of a query q to identify holes in the buckets of an *STHoles* histogram. Such holes correspond to bucket's sub-regions with distinctive tuple frequency, which we exploit to refine and make the *STHoles* histogram more accurate.

In general, a query q intersects some buckets only partially. For each such bucket b_i, we know the exact number of tuples in $q \cap b_i$ by inspecting the results for q. Intuitively, if $q \cap b_i$ has a *disproportionately* large or small fraction of the tuples in b_i, then $q \cap b_i$ is a candidate to become a hole of bucket b_i. Hence, each partial intersection of q and a histogram bucket could in principle be used to improve the quality of our histogram, as illustrated in the example below.

[4]Although the approximate total number of tuples in the data set can be efficiently retrieved from system catalogs, the maximum and minimum values for each attribute could be expensive to maintain in the absence of indexes.

Example 2: *Figure 6 shows a bucket b with frequency $f(b) = 100$. Suppose that from the result stream for a query q we count that $T_b = 90$ tuples lie in the part of bucket b that is touched by query q, $q \cap b$. Using this information, we can deduce that bucket b is significantly skewed, since 90% of its tuples are located in a small fraction of its volume. We can improve the accuracy of the histogram if we create a new bucket b_n by "drilling" a hole in b that corresponds to the region $q \cap b$ and adjust b and b_n's frequencies accordingly as illustrated in Figure 6.* ▮

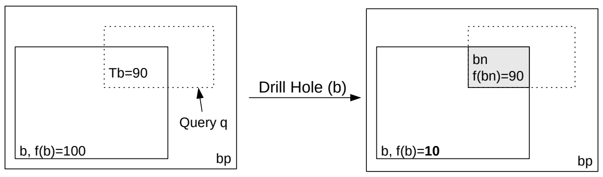


Figure 6: Drilling a hole in bucket b to improve the histogram quality.

If the intersection of a query q and a bucket b is rectangular, as in Figure 6, we can always consider $q \cap b$ as a candidate hole and proceed as in the previous example. However, in general it is not always possible to create a hole in a bucket b to form a new bucket $q \cap b$. The problem is that some children of b might be taking some of b's space, and therefore the bounding box of $q \cap b$ might not be rectangular anymore, thus violating the partitioning constraint we impose on the histogram. For instance, in Figure 6 the intersection between q and b's parent b_p has an L shape, due precisely to bucket b. We could simply ignore those intersections in our analysis, but that would result in low quality histograms, since a significant fraction of the intersections are not rectangular. We have chosen a middle ground to approximate the shape of $q \cap b$ when it is not rectangular. Essentially we *shrink* $q \cap b$ to a large rectangular sub-region that does not *partially* intersect with the bounding box of any other bucket. We then estimate the number of tuples in this sub-region assuming uniformity. That is, if T_b is the number of tuples in $q \cap b$ and c is the result of shrinking $q \cap b$, we estimate T_c, the number of tuples in c, as $T_c = T_b \frac{v(c)}{v(q \cap b)}$.

Example 3: *Figure 7 shows a four-bucket histogram and the progressive shrinking of the initial candidate hole $c = q \cap b$. At the beginning, the buckets that partially intersect with c, called participants in the algorithm below, are b_1 and b_2 (b_3 is completely included in c). We first shrink c along the "vertical" dimension so that the resulting candidate hole c' does not intersect with b_1 anymore. Then, we shrink c' along the "horizontal" dimension so that the resulting candidate hole c'' does not intersect with b_2. At this point there is no bucket that partially intersects with c''. The resulting candidate hole c'' is rectangular and covers a significant portion of the original $q \cap b$ region.* ▮

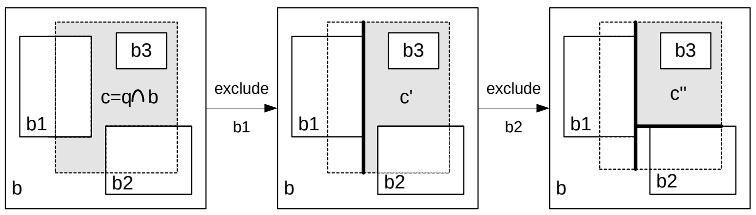


Figure 7: Shrinking a candidate hole $c = q \cap b$.

More generally, the procedure for shrinking the intersection of a bucket b and a query q is:

```
Shrink (b:bucket, q:query, Tb: # of tuples in b)
c = $q \cap b$
participants = $\{b_i \in$ children(b): c$\cap b_i \neq \emptyset \wedge b_i \not\subseteq$c$\}$
while (participants $\neq \emptyset$)
   Select bucket $b_i \in$ participants and dimension $j$
     such that shrinking c along $j$ by excluding $b_i$
     results in the smallest reduction of $c$
   Shrink c along $j$
   Update participants
end while
Tc = Tb * v(c) / v($q \cap b$) // adjust frequency
Return candidate hole c with frequency Tc
```

In summary, for each query q of our workload we identify the candidate new holes to refine a given histogram. Specifically, these new candidate buckets are the result of invoking shrink(b_i, q, T_{b_i}) for all buckets b_i such that $q \cap b_i \neq \emptyset$, where T_{b_i} is the number of tuples in the result of q that lie inside bucket b_i.

4.2.2 Drilling Candidate Holes as New Histogram Buckets

In the previous section we saw how we identify candidate new holes to refine an *STHoles* histogram. Each candidate hole c with frequency T_c that results from shrinking from $q \cap b_i$ is completely included in b_i and does not intersect partially with any child of b_i. (As illustrated in Figure 7, some of b_i's children could be *fully* enclosed in c.) We now show how to effectively "drill" such candidates as new histogram buckets. For this, we identify three possible scenarios:

1. Bucket b_i and candidate hole c reference exactly the same region in the data domain, i.e., $box(c) = box(b_i)$. In this case, the candidate hole c carries updated information about the number of tuples in b_i, T_c, but we do not drill c in b_i, since they represent essentially the same region. We handle this situation by simply replacing b_i's frequency with T_c.

2. Candidate hole c covers all b_i's remaining space. This is a relatively rare special case, but we need to handle it properly to avoid wasting space. Consider the histogram in Figure 8, with four buckets b_1, b_2, b, and b_p, and suppose that we want to drill c in bucket b. Although $c \neq box(b)$, c covers all of b's remaining space (the rest is covered by buckets b_1 and b_2). If we simply added a new child b_n to bucket b with $box(b_n) = c$, then bucket b proper would carry no information, because b would be completely covered by its children b_1, b_2, and b_n. Hence adding b_n as a new child of b would result in wasted space. To avoid this situation, we eliminate bucket b from the histogram and transfer b's children to b's parent b_p. More specifically, we first merge b with its parent b_p, and then we drill c again but this time in b_p instead of in b, thus saving one bucket's worth of space [5].

3. The default situation. We can directly apply the ideas from the beginning of Section 4.2. That is, we create a new child of b_i, denoted b_n, with $box(b_n) = c$ and $f(b_n) = T_c$. We then migrate all of b_i's children whose bounding boxes are completely included in c so they become children of the new bucket b_n. Finally, we adjust the frequency of b_i to restore, whenever possible, the previous frequency counts. That is, if we had enough tuples in b_i, i.e., $f(b_i) \geq T_c$, we subtract T_c from $f(b_i)$. Otherwise, we simply set $f(b_i)$ to zero.

[5]As an alternative, we could avoid merging b and b_p and then drilling b_n by simply changing the frequency of b. However, our preferred choice results in less overlap among buckets, which is in general desirable.

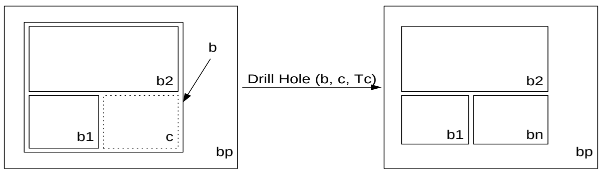


Figure 8: Drilling b_n in bucket b would make b carry no useful information.

The complete procedure is described below:

```
DrillHole (b: bucket, c: candidate hole, Tc: c’s frequency)
// c is included in b and does not partially intersect
// with any child of b.
if box(b)=box(c) // (Scenario 1)
   f(b)=Tc
else if v(b) = volume(c∩b) then // (Scenario 2)
   merge b with its parent $b_p$
   DrillHole($b_p$, c, Tc)
else // default case (Scenario 3)
   add a new child of b, $b_n$, to the histogram
   box($b_n$)=c ; f($b_n$)=Tc
   migrate all children of b that are enclosed by c
      so they become children of $b_n$
   f(b) = MAX{0, f(b) - Tc}
```

4.2.3 Merging Buckets

The previous section showed how we can refine an *STHoles* histogram by adding buckets as holes to existing buckets. In doing so, we might temporarily exceed our target number of histogram buckets. Hence, after adding buckets, we need to reduce the number of histogram buckets by merging *similar* ones, more specifically those buckets with the closest tuple density.

Example 4: *Consider the three-bucket histogram H in Figure 9, and suppose that we have a two-bucket budget. Two choices we have to eliminate one bucket are: merging buckets b_1 and b_2, which results in histogram H_1, and merging buckets b_1 and b_3, which results in histogram H_2. Although buckets b_1 and b_3 have the same frequency in H (100 tuples each), histogram H_1 is more similar to the original, three-bucket histogram H than H_2 is. In fact, both H and H_1 result in the same selectivity estimation for arbitrary range queries, since b_1 and b_2’s* densities *in H are the same. In contrast, histogram H_2 returns lower selectivity estimations than H for range queries that only cover the lower half of the new bucket b_n, since the tuple density of bucket b_3 is lower than the tuple density of bucket b_1 in histogram H.* ▮

More generally, to decide which buckets to merge, we use a *penalty* function that returns the cost in histogram accuracy of merging a pair of buckets.

Calculating Penalties

Suppose we want to merge two buckets b_1 and b_2 in a given histogram H. Let H' be the resulting histogram after the merge. We define the *penalty* of merging buckets b_1 and b_2 in H as follows:

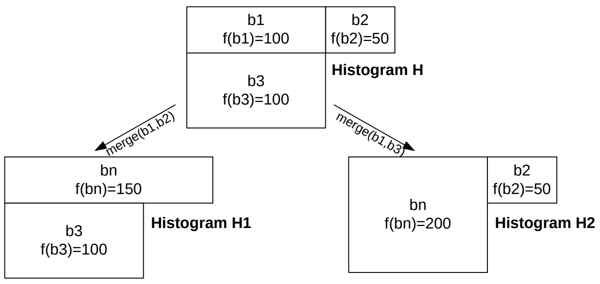


Figure 9: Merging similar buckets.

$$penalty(H, b_1, b_2) = \int_{p \in dom(D)} |est(H, p) - est(H', p)| \, dp$$

where $dom(D)$ is the domain of the data set D. In other words, the penalty for merging two buckets measures the difference in approximation accuracy between the old, more expressive histogram where both buckets are separate, and the new, smaller histogram where the two (and perhaps additional buckets) have been collapsed. A merge with a small penalty will result in little difference in approximation for range queries and therefore will be preferred over another merge with higher penalty. Since the estimated density of tuples inside a bucket is constant by definition, we can calculate penalty functions efficiently. We can identify all regions r_i in the data domain with uniform density of tuples *both* before and after the merge, and just add a finite number of terms of the form $|est(H, r_i) - est(H', r_i)|$ [6]. This procedure will become clearer in the rest of this section when we instantiate it to concrete situations.

We identified two main families of merges for *STHoles* histograms, which correspond to merging "adjacent" buckets in the tree representation of an *STHoles* histogram: *parent-child* merges, where a bucket is merged with its parent, and *sibling-sibling* merges, where two buckets with the same parent are merged possibly taking some of the parent space (since we need to enclose both siblings in a rectangular bounding box). The motivation behind these two classes of merges is as follows: Parent-child merges are useful to eliminate buckets that become too similar to their parents, e.g., when their own children cover all interesting regions and therefore carry all useful information. On the other hand, sibling-sibling merges are useful to extrapolate frequency distributions to yet unseen regions in the data domain, and also to consolidate buckets with similar density that cover close regions. Below we define these two merge variants in detail.

Parent-Child Merges

Suppose we want to merge buckets b_c and b_p, where b_p is b_c's parent. After the merge (Figure 10) a new bucket b_n replaces b_p, and bucket b_c disappears. The new bucket b_n has $box(b_n) = box(b_p)$ and $f(b_n) = f(b_c) + f(b_p)$. The children of both buckets b_c and b_p become children of the new bucket b_n. Therefore, we have that $v(b_n) = v(b_c) + v(b_p)$. The only regions in the original histogram that change the estimated number of tuples after the merge are b_p and b_c. In conclusion, we have that:

$$penalty(H, b_p, b_c) = \underbrace{\left| f(b_p) - f(b_n)\frac{v(b_p)}{v(b_n)} \right|}_{|est(H, b_p) - est(H', b_p)|} + \underbrace{\left| f(b_c) - f(b_n)\frac{v(b_c)}{v(b_n)} \right|}_{|est(H, b_c) - est(H', b_c)|}$$

[6]We can think of this procedure as taking all points $p \in dom(D)$, "`group` them `by` $est(H, p), est(H', p)$," and processing each group individually.

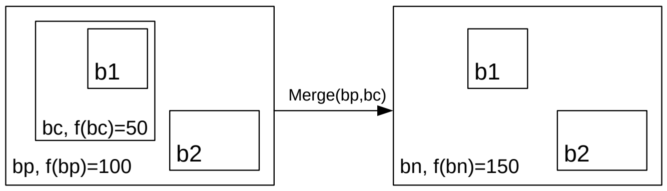


Figure 10: *Parent-Child* merge.

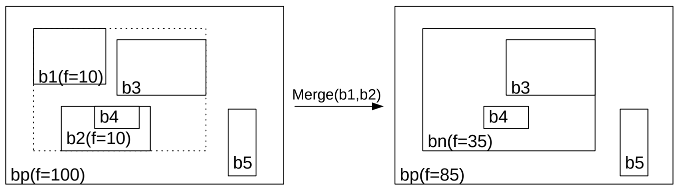


Figure 11: *Sibling-Sibling* merge.

where H' is the histogram that results from merging b_p and b_c in H. The remaining points p in the histogram domain are such that $est(H,p) = est(H',p)$, so they do not contribute to the merge penalty.

Sibling-Sibling Merges

Consider the merge of buckets b_1 and b_2, with common parent b_p (Figure 11). We first determine the bounding box of the resulting bucket b_n. We define $box(b_n)$ as the smallest box that encloses both b_1 and b_2 and does not intersect partially with any other child of b_p (that is, we start with a bounding box that tightly encloses b_1 and b_2 and progressively expand it until it does not intersect partially with any other child of b_p). In the extreme situation that $box(b_n)$ is equal to b_p, we transform the sibling-sibling merge of b_1 and b_2 into two parent-child merges, namely b_1 and b_p, and b_2 and b_p. Otherwise, we define the set I of "participant" buckets as the set of b_p's children (excluding b_1 and b_2) that are included in $box(b_n)$. After the merge, the new bucket b_n replaces buckets b_1 and b_2. In general, b_n will also contain a part of the old b_p. The volume of that part is $v_{old} = \mathit{vBox}(b_n) - (\mathit{vBox}(b_1) + \mathit{vBox}(b_2) + \sum_{b_i \in I} \mathit{vBox}(b_i))$. Therefore, the frequency of the new bucket is $f(b_n) = f(b_1) + f(b_2) + f(b_p)\frac{v_{old}}{v(b_p)}$. Also, the modified frequency of b_p in the new histogram becomes $f(b_p)(1 - \frac{v_{old}}{v(b_p)})$. To complete the merge, the buckets in I and the children of the old b_1 and b_2 become children of the new b_n. Therefore, we have that $v(b_p) = v_{old} + v(b_1) + v(b_2)$. The only regions in the original histogram that change the estimated number of tuples after the merge are the ones corresponding to b_1, b_2, and the portion of b_p enclosed by $box(b_n)$. Hence:

$$penalty(H, b_1, b_2) = \underbrace{\left| f(b_n)\frac{v_{old}}{v(b_n)} - f(b_p)\frac{v_{old}}{v(b_p)} \right|}_{|est(H,r_{old}) - est(H',r_{old})|} + \underbrace{\left| f(b_1) - f(b_n)\frac{v(b_1)}{v(b_n)} \right|}_{|est(H,b_1) - est(H',b_1)|} + \underbrace{\left| f(b_2) - f(b_n)\frac{v(b_2)}{v(b_n)} \right|}_{|est(H,b_2) - est(H',b_2)|}$$

where H' is the histogram that results from merging b_1 and b_2 in H, and r_{old} is the portion of the old bucket b_p covered by the new bucket b_n. The remaining points p in the histogram domain are such that $est(H,p) = est(H',p)$, so they do not contribute to the merge penalty.

Putting all pieces together, we are now ready to refine the *STHoles* construction algorithm from the beginning of this section as follows:

```
**BuildAndRefine** (H: STHoles, D:Data Set, W: Workload)
Initialize H with no buckets (empty histogram)
// Or use an existing histogram if available
for each query $q \in$ W do
   Expand H's root bucket (if needed) so that it contains $q$.
   Count, for all buckets $b_i$, the number of tuples in $q \cap b_i$, $T_{b_i}$
   for each bucket $b_i$ such that $q \cap b_i \neq \emptyset$ do
      // approximate shape if necessary
      $(c_i, T_{c_i})$ = Shrink ( $b_i, q, T_{b_i}$ )
      if (est(H, $c_i$ )$\neq T_{c_i}$) then
         DrillHole($b_i, c_i, T_{c_i}$)
   end for
   while H has too many buckets,
      merge the pair of buckets in H with the lowest penalty
end for
```

5 Implementation Issues

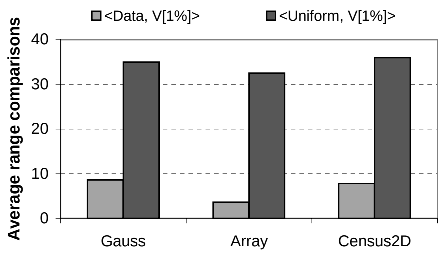
While we are tuning an *STHoles* histogram for a specific query workload, we need to incur certain overhead for each query that is used for histogram refinement. In this section, we discuss the overhead involved in two important aspects of the histogram construction technique of Section 4.2. More specifically, Section 5.1 explains how to compute the merge penalty function of Section 4.2.3 efficiently. Then, Section 5.2 analyzes the impact of gathering statistics "on the fly" to refine *STHoles* buckets in a real commercial DBMS.

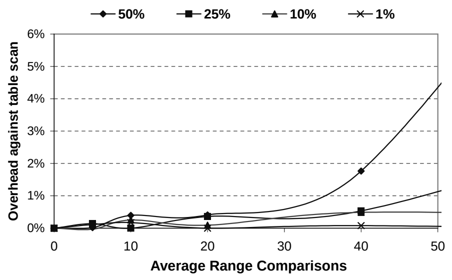
5.1 Approximating Penalties

In Section 4.2.3 we discussed how to merge pairs of histogram buckets with low associated *penalty*. To implement our *penalty* function efficiently, we could maintain a two-dimensional array P in memory, where $P[i,j]$ is the penalty of merging buckets b_i and b_j. This array would need to be updated as we refine the histogram. Unfortunately, the size of this array is quadratic in the number of buckets. Although the array is needed only during the tuning of the *STHoles* histogram, we can use instead an approximation of this array that requires only linear space in the number of buckets and that results in virtually no significant degradation in histogram accuracy. Specifically, we propose to *weaken* the definition of "best penalty" by allowing some times to merge a pair of buckets with a relatively low, but not lowest, penalty. Intuitively, we use the fact that after merging two buckets or drilling a hole to an existing bucket, most of the penalties remain unchanged, or change only slightly. Therefore, in some cases we do not recalculate the penalty function for some combination of buckets, which results in an approximate, but more efficient technique. In Section 7.2 we show experimentally that this approximation, which requires less space, is more efficient and results in only slightly worse accuracy than the original, more expensive, technique. The implementation details of such approximation are explained in Appendix A.

5.2 Performance of the Counting Procedure

To refine an *STHoles* histogram, we need to intercept queries that participate in the tuning and analyze their results at run time. It becomes crucial, then, to quantify the overhead that this analysis is adding to regular





$(\mathbf{a})$ Average number of range comparisons for four different data sets and workloads.

$(\mathbf{b})$ Overhead of the counting procedure for different query selectivities.

Figure 12: Performance evaluation on *Microsoft SQL Server 2000.*

query execution on a real DBMS. In this section we study this overhead and report experimental results over a commercial DBMS, namely *Microsoft SQL Server 2000*.

As explained in Section 4.2, given a workload query q we need to calculate the number of tuples in the answer of q that lie inside each bucket of the current histogram. We can efficiently identify the buckets $b_1, \dots, b_k$ that intersect with q before its execution. Let $(L_i^1, \dots, L_i^d)$ and $(U_i^1, \dots, U_i^d)$ be the lowest and highest d-dimensional points corresponding to the boundary of $box(b_i)$, $i = 1, \dots, k$. We can then interleave a new operator right after the filter operator in q's query execution plan that maintains one counter per bucket and updates them accordingly after analyzing each tuple that is pipelined to the next operator in the execution plan. Using the fact that if b_i is a child of b_j then $box(b_i) \subset box(b_j)$, and assuming that buckets are kept in order,[7] the counting operator can be written as follows:

```
if      ($L_1^1 \leq t^1 < H_1^1$ and ... and $L_1^d \leq t^d < H_1^d$) then counter[1]++
else if ($L_2^1 \leq t^1 < H_2^1$ and ... and $L_2^d \leq t^d < H_2^d$) then counter[2]++
...
else if ($L_{k-1}^1 \leq t^1 < H_{k-1}^1$ and ... and $L_{k-1}^d \leq t^d < H_{k-1}^d$) then counter[k-1]++
else counter[k]++
```

where $t = (t^1, \dots, t^d)$ is the current streamed tuple coming from the filter operator.

We conducted some experiments to determine the average number of range comparisons that are needed for different data sets and workloads. Figure 12(a) shows the results for two-dimensional data sets when we allocate 200 buckets for the histograms. The average number of range comparisons per query in the workload required by *STHoles* is less than 10 for $\langle$*Data*$, V[1\%]\rangle$ workloads, and around 35 for $\langle$*Uniform*$, V[1\%]\rangle$ workloads (see Section 6.3 for a discussion of workload notation).

We studied the overhead of this new operator in the code of *Microsoft SQL Server 2000*, and tested it for different numbers of range comparisons and query selectivities. Figure 12(b) shows the overhead of the counting procedure for range queries with different selectivities. When the number of range comparisons is zero, we are back to the case when no counting is done at all, and a traditional table scan is executed (we did not use indexes in our experiments). We can see that the overhead imposed by considering about 35 range comparisons is about 2%. In fact, the overhead by considering 60 range comparisons (many more than the numbers reported in Figure 12(a)), is still below 10%. This overhead is acceptable and can be regarded as an amortized cost we pay for the online construction of *STHoles* histograms. Moreover, this overhead can be drastically reduced if we sample the workload and refine the histogram using only a subset of the queries.

[7]That is, if b_i is a descendant of b_j in the tree, then b_i appears before b_j. This order can be achieved by traversing the tree in postorder, and keeping only the buckets that intersect with q.

6 Experimental Setting

This section defines the data sets, histograms, and workloads used for the experiments of Section 7.

6.1 Data Sets

We use both *synthetic* and *real* data sets for the experiments. The real data sets we consider [3] are: *Census2D* and *Census3D* (two- and three-dimensional projections of a fragment of US Census Bureau data) consisting of 210,138 tuples, and *Cover4D* (four-dimensional projection of the CovType database, used for predicting forest cover types from cartographic variables), consisting of 545,424 tuples. We also generated synthetic data sets for our experiments following different data distributions, as described below.

Gauss: The *Gauss* synthetic distributions [29] consist of a predetermined number of overlapping multidimensional gaussian bells. The parameters for these data sets are: the number of gaussian bells p, the standard deviation of each peak σ, and a zipfian parameter z that regulates the total number of tuples contained in each gaussian bell.

Array: These data sets were used in [1]. Each dimension has v distinct values, and the value sets of each dimension are generated independently. Frequencies are generated according to a zipfian distribution and assigned to randomly chosen cells in the joint frequency distribution matrix. The parameters for this data set are the number of distinct attributes by dimension v, and the zipfian value for the frequencies z. When all the data points are equidistant, this data set can be seen as an instance of the *Gauss* data set with $\sigma = 0$ and $p = v^d$.

The default values for the synthetic data set parameters are summarized in Table 1.

Data Set	Attribute	Value
All	d: Dimensionality	2
	N: Cardinality	500,000
	R: Data domain	$[0 \ldots 1000)^d$
	z: Skew	1
Gauss	p: Number of peaks	100
	σ: Peaks' standard deviation	25
Array	v: Distinct attribute values	100

Table 1: Default values for the synthetic data sets.

6.2 Histograms

We compare our *STHoles* histograms against the following multidimensional histograms: *MHist* based on *MaxDiff(v,a)* [26], *EquiDepth* [18], *STGrid* [1] and *GenHist* [9], using the values of parameters that the respective authors considered the best. (See Section 2 for a summary of these techniques.) All experiments allocate the same amount of memory for all histograms techniques, which however translates to different numbers of buckets for each. Consider the space requirements for B d-dimensional buckets. Both *EquiDepth* and *MHist* histograms require $2 \cdot d \cdot B$ values for the bucket boundaries plus B frequency values. *STGrid* histograms need B values for frequencies plus around $\sqrt[d]{B}$ values for the unidimensional rulers [1]. *GenHist* histograms require $2 \cdot B$ values for bucket positions plus B frequency values. Finally, *STHoles* histograms use $2 \cdot d \cdot B$ values for bucket boundaries, B values for frequencies, and $2 \cdot B$ pointers for maintaining the tree structure, since each bucket needs to point to its "first" child plus a sibling [8]. By default, the available memory for a histogram is fixed to 1,000 bytes.

6.3 Workloads

We use a slightly modified version of the framework given in [23] to generate probabilistic models for range queries. Given a data set, a range query model is defined as a pair $\langle C, R[v] \rangle$, where C is the distribution of the query centers, R is a function that constrains the query boundaries, and v is a constant value for R. To obtain a workload given a query model, we first generate the query centers using C and then expand their boundaries so they follow $R[v]$.

For our experiments, we consider the following center distributions, which are considered representative of user behavior [23]:

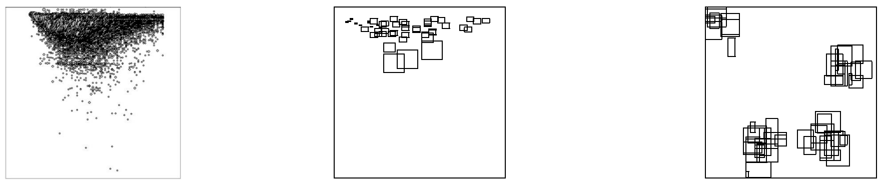
- $\mathbf{Data}$: The query centers follow the data distribution.
- $\mathbf{Uniform}$: The query centers are uniformly distributed in the data domain.
- $\mathbf{Gauss}$: The query centers follow a *Gauss* distribution independent of the data distribution.

The range constraints we used for our experiments are:

- $\mathbf{V[c_v]}$: The range queries are hyper-rectangles included in a hypercube of *volume* c_v, and model the cases in which the user specifies the query values in terms of a window area.
- $\mathbf{T[c_t]}$: The range queries are hyper-rectangles that cover a region with c_t *tuples*, and model the situations in which the user has knowledge about the data distribution and issues queries with the intention of retrieving a given number of tuples.

Parameters c_v and c_t are specified as a percentage of the total volume and number of tuples of the data distribution, respectively.

By combining these parameters we obtain six different probabilistic models for query workloads. By default, we use 1% for both c_v and c_t. As an example, the query model $\langle \textit{Data}, T[1\%] \rangle$ results in queries with centers that follow the data distribution and contain 1% of the tuples in the data set. Similarly, the query model $\langle \textit{Gauss}, V[1\%] \rangle$ corresponds to queries with centers that follow a multi-gaussian distribution and have an average volume of around 1% of the data domain. Figure 13 shows two sample workloads of 50 queries each for the *Census2D* data set.



$(\mathbf{a})$ *Census2D* data set $(\mathbf{b})$ $\langle \textit{Data}, T[1\%] workload \rangle$ $(\mathbf{c})$ $\langle \textit{Gauss}, V[1\%] workload \rangle$

Figure 13: Two workloads for the *Census2D* data set.

[8]Note that this analysis does not account for the temporary space needed for merge-penalty bookkeeping (see Appendix A), which is only kept during histogram refinement.

6.4 Metrics

To compare our new technique against existing ones, we first construct a *training* workload that consists of 1,000 queries and use it to tune the *STHoles* and *STGrid* histograms. Then, we generate a *validation* workload from the same distribution as the training workload that also consists of 1,000 queries, and calculate the average absolute error for all the histograms. Given a data set D, a histogram H, and a validation workload W, the *average absolute error* $E(D, H, W)$ is calculated as follows:

$$E(D,H,W) = \frac{1}{|W|} \sum_{q \in W} |est(H,q) - act(D,q)|$$

where $est(H,q)$ is the estimate of the number of tuples in the result of q, using histogram H for the estimation, and $act(D,q)$ is the actual number of D tuples in the result of q.

We choose average absolute errors as the accuracy metric, since relative errors tend to be less robust when the actual number of tuples for some queries is zero or near zero. In general, however, absolute errors greatly vary across data sets, making it difficult to report results for different data sets. Therefore, for each experiment, we *normalize* the average absolute error by dividing it by $E_{unif}(D,W) = \frac{1}{|W|} \sum_{q \in W} |est_{unif}(D,q) - act(D,q)|$, where $est_{unif}(D,q)$ is the result size estimate obtained by assuming uniformity, i.e., in the case where no histograms are available. We refer to the resulting metric as *Normalized Absolute Error*.

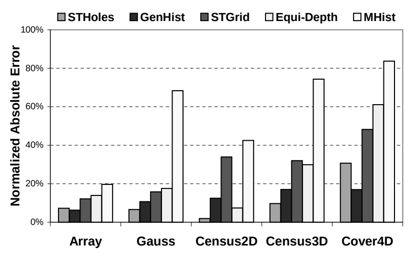
7 Experimental Evaluation

In Section 7.1 we evaluate the performance of *STHoles* histograms against that of existing techniques. Section 7.2 shows some additional experiments that explore specific aspects of *STHoles* histograms.

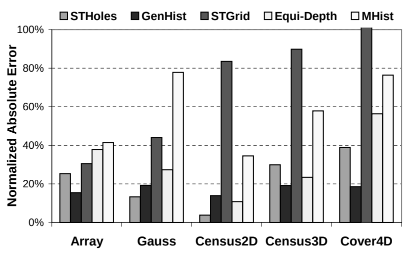
7.1 Comparison of STHoles and Other Histogram Techniques

Accuracy of Histograms: Figure 14 shows normalized absolute errors for different histograms, data sets and workloads. We can see from the figures that the techniques that are based on truly multidimensional analysis of the data, i.e., *STHoles* and *GenHist*, result in better accuracy than the others. In particular, *STHoles* histograms give better results than *EquiDepth*, *MHist* and *STGrid* in virtually all cases. On the other hand, *STHoles* and *GenHist* are comparable in accuracy, and although *STHoles* histograms do not directly inspect the data distributions, in many cases they outperform *GenHist* histograms. The only dataset in which *GenHist* results in significantly better accuracy than *STHoles* is *Cover4D* (see Figure 14). For this high-dimensional data set, the ability to capture interesting data patterns based only on workload information is diminished. However, it is important to note that, even for high dimensions, *STHoles* histograms still produce better results than do *MHist*, *EquiDepth*, and *STGrid* histograms. *GenHist* has a high error rate of 75% for the *Array* data set with the $\langle$*Data*$, T[1\%]\rangle$ workload. This may be due to the choice of histogram construction parameter values in [9], which is independent of the underlying data set. In general, note that *STHoles*, *GenHist* and, to a limited extent, *EquiDepth* histograms are "robust" across different data sets and workloads, in the sense that they consistently produce reasonable results. In contrast, *STGrid* and *MHist* become too inaccurate for some combinations of data sets and workloads.

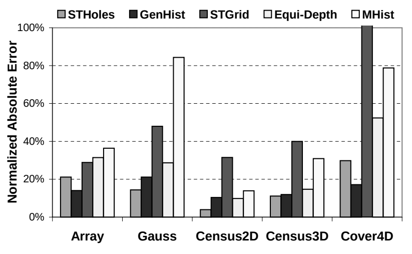
To validate the robustness of our new approach, we varied some parameters in the synthetic data set generation as well as some parameters in the query models.



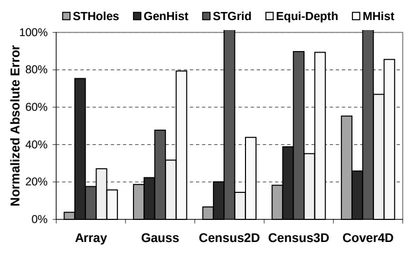
($\mathbf{a}$) $\langle$*Data*$, V[1\%]\rangle$.



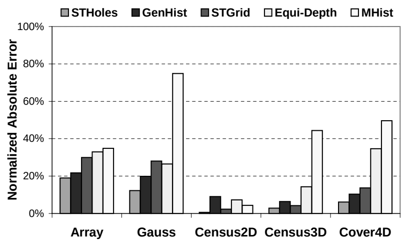
($\mathbf{b}$) $\langle$*Uniform*$, V[1\%]\rangle$.



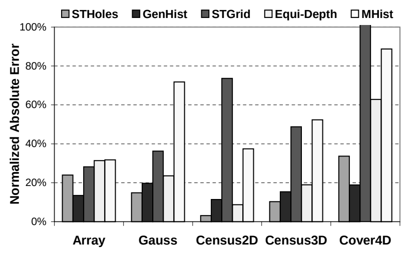
($\mathbf{c}$) $\langle$*Gauss*$, V[1\%]\rangle$.



($\mathbf{d}$) $\langle$*Data*$, T[1\%]\rangle$.



($\mathbf{e}$) $\langle$*Uniform*$, T[1\%]\rangle$.

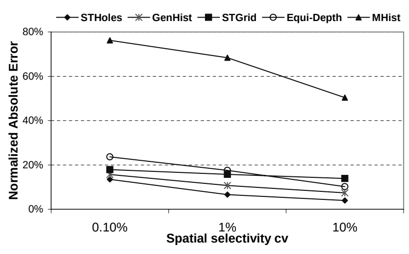


($\mathbf{f}$) $\langle$*Gauss*$, T[1\%]\rangle$.

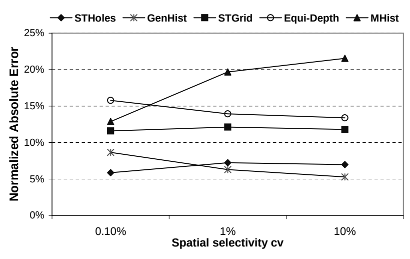
Figure 14: Normalized absolute error for different histograms, data sets and validation workloads.

Robustness across Workloads: Figure 15(a-f) shows the normalized absolute error for different data sets and for varying selectivity s for workloads $\langle$*Data*$, V[s]\rangle$ and $\langle$*Data*$, T[s]\rangle$, respectively. We can see that in almost all cases *STHoles* histograms outperform traditional techniques. Even in the few cases that *STHoles* histograms are not the most accurate, they are a close second, with only one exception. Our technique is not too accurate in Figure 15(d) for tuple selectivity $c_t = 0.1\%$ (and neither are *MHist* and *STGrid*). This is mainly because in the *Gauss* data set the $\langle$*Data*$, T[0.1\%]\rangle$ workload consists of many small and disjoint queries. This workload is particularly bad for any histogram refinement technique like *STHoles* that bases all decisions on query feedback, without examining the actual data sets at any time. To deal with such workloads, we can slightly modify the construction algorithm for *STHoles* histograms (Section 4.2) to start with a more informed representation of the data set. In particular, we can use an existing histogram (e.g., *EquiDepth*) as the starting point for our technique in the algorithm of Section 4.2. We implemented and tested the accuracy of the histograms that result from starting with an *EquiDepth* histogram and turning it into an *STHoles* histogram through workload refinement. The results are *highly* accurate for a variety of data sets and workloads. In particular, for the *Gauss* data set and $\langle$*Data*$, T[0.1\%]\rangle$ workload in Figure 15(d), this alternative version of *STHoles* results in 42% of normalized absolute error, i.e., comparable with *GenHist*, the most accurate histogram for that particular configuration.

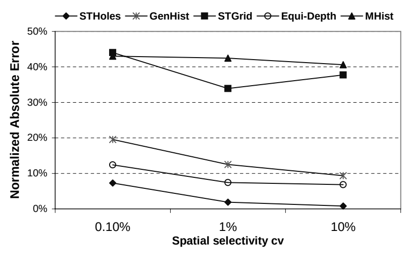
Robustness for Varying Data Set Skew: In Figure 16 we show the results when the skew z used to generate data sets changes from 0.5 to 2. (The experiments we reported so far used $z = 1$.) *STHoles* has the lowest error rates for all skews but for the *Array* data set and $z \in \{0.5, 1\}$ where is a close second after *GenHist*. *MHist* behaves poorly for the *Gauss* data set, only slightly better than assuming uniformity and independence. However, it becomes more accurate for highly skewed *Array* distributions, but only marginally better than the other techniques. In particular, when $z = 2$, *STHoles* and *MHist* result in the same highest accuracy. It is worth noting that *Array* with $z = 2$ is a highly skewed data set with just $10,000$ distinct values. The most popular tuple is repeated 295,054 times, and hence accounts for 59% of the data set. The five most frequent tuples account for 87% of the data set. On the other hand, 95% of the distinct values have frequency one. This data set is then almost a uniform data set with a few prominent peaks. Incidentally, an extreme data set like



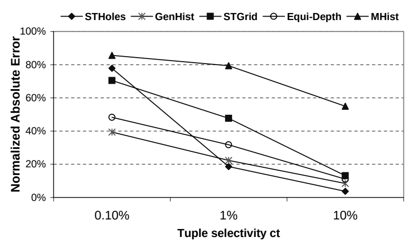
$(\mathbf{a})$ *Gauss* data set.



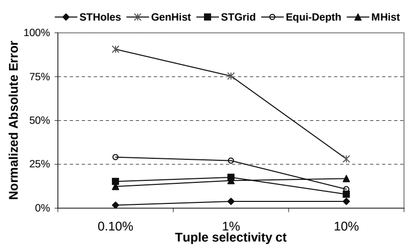
$(\mathbf{b})$ *Array* data set.



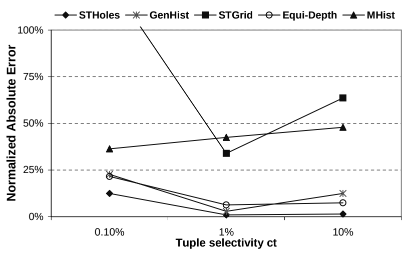
$(\mathbf{c})$ *Census2D* data set.



$(\mathbf{d})$ *Gauss* data set.

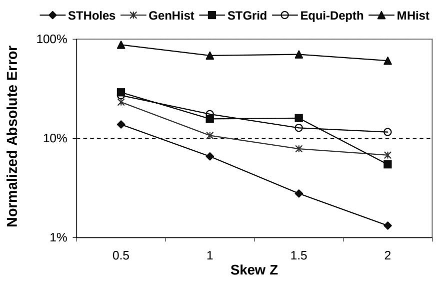


$(\mathbf{e})$ *Array* data set.

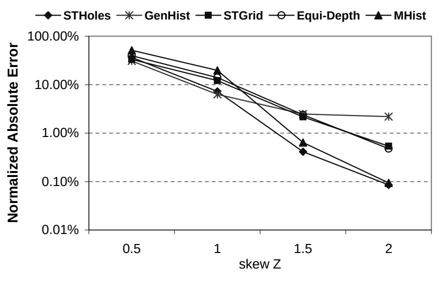


$(\mathbf{f})$ *Census2D* data set.

Figure 15: Normalized absolute error using $\langle \textit{Data}, T[c_t] \rangle$ for varying spatial (c_v) and tuple (c_t) selectivity.



$(\mathbf{a})$ *Gauss* data set.



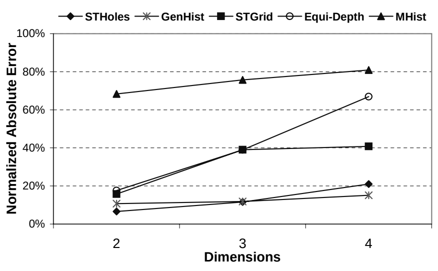
$(\mathbf{b})$ *Array* data set.

Figure 16: Normalized absolute error for varying data skew.

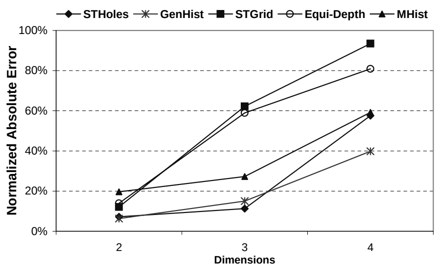
this one would probably be best modelled using an end-biased histogram [27].

Robustness for Varying Data Set Dimensionality: Finally, Figure 17 shows the error for varying the dimensionality d of the synthetic data sets. (The experiments we reported so far used $d = 2$.) *STHoles* and *GenHist* achieve the highest accuracy for all data set dimensionalities, with *GenHist* being more accurate for higher number of dimensions, as discussed above. Again, *MHist* behaves the worst for the *Gauss* data set, and performs better for the *Array* data set. For $d = 4$, the corresponding *Array* data set is especially well suited for the *MHist* technique, since it has only 20 different values per dimension (which adds up to 160,000 different values), and the difference in frequency greatly varies among them. Therefore, *MHist* is able to capture these high frequency values accurately.

In conclusion, although for some particular configurations *STHoles* histograms are slightly outperformed by others (notably in one data point of Figure 15(d)), in general *STHoles* is a stable technique across different workloads and data sets, and typically results in significantly lower estimation errors than multidimensional histograms that inspect the data sets.

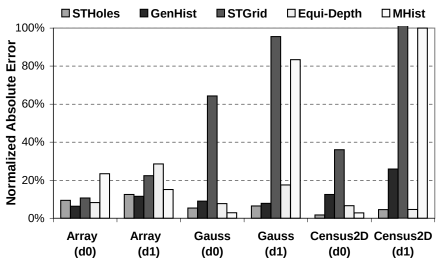


(**a**) *Gauss* data set.

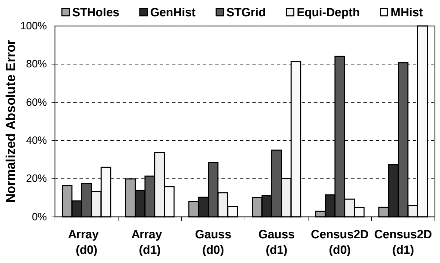


(**b**) *Array* data set.

Figure 17: Normalized absolute error for varying data dimensionality.



(**a**) $\langle$*Data*$, V[1\%]\rangle$ workload.



(**b**) $\langle$*Uniform*$, V[1\%]\rangle$ workload.

Figure 18: Normalized absolute error for workload projections.

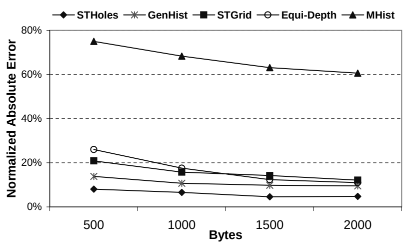
Estimating Selectivities for Queries With Fewer Attributes: In a real system, the attributes mentioned in some range queries might not match exactly the set of attributes covered by the existing histograms. If the set of attributes in a query q is a subset of the attributes used in a histogram H, we can use H directly to answer q by projecting H over the relevant attributes. This section explores the accuracy of d-dimensional *STHoles* histograms over projections of workloads onto $d-1$ dimensions.

Figure 18 shows the normalized absolute error for different data sets, workloads, and projected dimensions. We can see that for $\langle$*Data*$, V[1\%]\rangle$ and $\langle$*Uniform*$, V[1\%]\rangle$ workloads (Figures 18(a)-(b)), the results are consistent with those of Figures 14(a) and 14(b) for *STHoles*, *GenHist*, and *EquiDepth* histograms. On the other hand, *MHist* and *STGrid* histograms present significant differences in accuracy depending on the particular projection. For instance, *MHist* is the best histogram for the *Gauss* data set when we focus on dimension $d = 0$. However, *MHist* is too inaccurate for the same data set when we focus on dimension $d = 1$: For the *Gauss* data set, *MHist* splits buckets almost exclusively along dimension $d = 0$ (see also Figure 3(e)). Therefore, the workload queries projected over dimension $d = 0$ represent a best case scenario for this histogram. However, when we project the queries over dimension $d = 1$, the results are significantly worse.

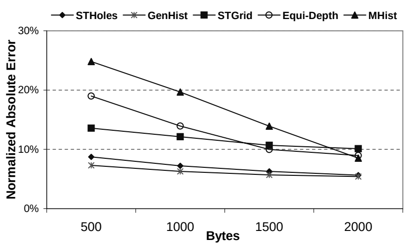
Effect of Varying the Available Storage: Figure 19 shows the normalized absolute error for the *Census2D*, *Gauss*, and *Array* databases for varying histogram size. The errors are presented for histograms using from 500 to 2,000 bytes of memory. *STHoles* histograms scale comparably to traditional histograms for the whole range of available memory.

7.2 Experiments Specific to Histogram Refinement

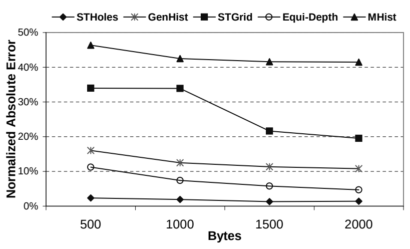
Effect of Using an Approximate Penalty Function: Section 5.1 and Appendix A described how to approximate the computation of the penalty function by maintaining a vector of merge candidates that are close to the



($\mathbf{a}$) *Gauss* data set.

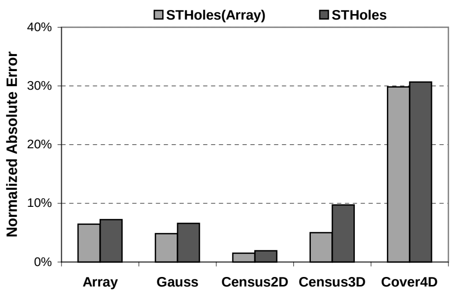


($\mathbf{b}$) *Array* data set.

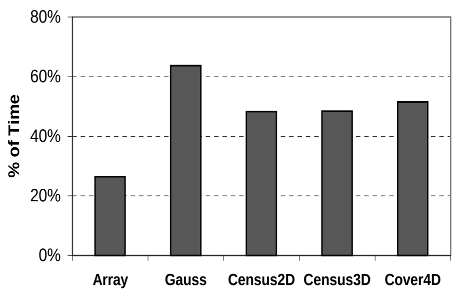


($\mathbf{c}$) *Census2D* data set.

Figure 19: Normalized absolute error for varying histogram sizes.



($\mathbf{a}$) Accuracy.



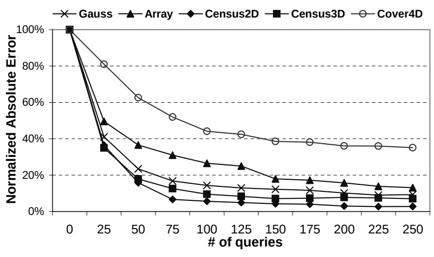
($\mathbf{b}$) Fraction of time of *STHoles* relative to *STHoles*(array).

Figure 20: Comparison of *STHoles* and *STHoles*(array) techniques for 1,000 $\langle Data, V[1\%]\rangle$ queries.

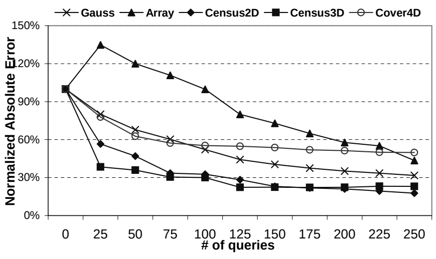
optimal ones. All the experiments that we reported so far use this (inexpensive) approximation. We now study whether using the (more expensive) version with the full array of penalties (denoted *STHoles*(Array)) results in significant improvements in performance. Figure 20(a) shows the normalized absolute error of *STHoles* and *STHoles*(array) techniques for different data sets and $\langle Data, V[1\%]\rangle$ workloads. The results are slightly better (as expected) for *STHoles*(array). Figure 20(b) reports the percentage of time that *STHoles* takes to process 1,000 queries relative to that of *STHoles*(array). We can see that both the space requirements (Section 5.1) and execution time needed to process 1,000 queries (Figure 20(b)) make *STHoles*(array) unattractive given the meager improvement in accuracy over the more efficient approximation of Section 5.1.

Convergence: Our techniques for building *STHoles* histograms keep adjusting the histograms as queries are evaluated. We now study how the quality of the *STHoles* histograms varies with the number of observed queries. To do so, we train the *STHoles* histogram 50 queries at a time, and after each step we calculate the normalized absolute error using the complete validation workload. Figure 21 shows the results for different data sets and workloads. We can see that *STHoles* histograms converge fairly quickly, and generally need only around 150-200 queries to get stable results.

Effect of Updates: Data sets are rarely static, and the data distribution might change over time. We now evaluate how well our new techniques adapts to changing data distributions. For this, we start with the *Array* and *Gauss* data sets, and progressively "morph" one into the other using random tuple swaps. Each column of four points in Figure 22 represents a different experiment where we vary the percentage of tuples that are swapped between the two data sets. For instance, in Figure 22(a) we start with the *Array* data set. We build the static *GenHist*, *MHist* and *EquiDepth* histograms using this data set, and train the *STHoles* and *STGrid* histograms using the *first half* of a $\langle Data, V[1\%]\rangle$ workload. Then, we randomly select a percentage

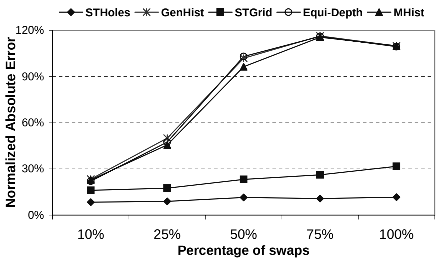


(**a**) $\langle$*Data*$, V[1\%]\rangle$ workload.

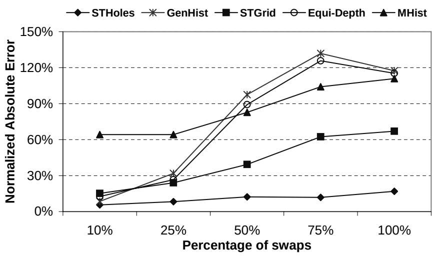


(**b**) $\langle$*Uniform*$, V[1\%]\rangle$ workload.

Figure 21: Normalized absolute error at different points of the online training.



(**a**) *Array*→ *Gauss*.



(**b**) *Gauss*→ *Array*.

Figure 22: Normalized absolute error after updates.

of tuples from the original *Array* data set and interchange them with randomly selected tuples from the *Gauss* data set. After that, we finish the training of the *STHoles* and *STGrid* histograms using the *remaining half* of the workload. Finally, we test all histograms using a validation $\langle$*Data*$, V[1\%]\rangle$ workload. Analogously, Figure 22(b) shows the results when starting with a *Gauss* data set and changing it to an *Array* data set.

Not surprisingly, we can see that the static histograms become really inaccurate when the underlying data distribution changes. In some cases the results are even worse than when assuming uniformity and independence, which highlights that periodically rebuilding such multidimensional histograms is essential (we include in the plots these static histograms just to quantify this behavior). In contrast, both *STGrid* and *STHoles* adapt gracefully to changes in the data distribution. For *STHoles* histograms we observe almost no degradation even when changing the data set completely. That is not the case for *STGrid* histograms. For instance, in Figure 22(b) we can see that *STHoles* keeps the error rate below 17% at all times, while *STGrid* results in over 67% of normalized absolute error for 100% tuple interchanges.

8 Conclusions and Future Work

In this paper, we presented a new histogram construction technique, *STHoles*, that exploits query workload and does not require examining the data sets. *STHoles* histograms allow buckets to be nested, and are tuned to the specific query workload received by the database system. Hence, buckets are allocated where needed the most as indicated by the workload, which leads to accurate query selectivity estimations. We established the robustness of the new histograms through extensive experimentation using a variety of synthetic and real-world data sets, as well as a variety of query workloads. We also experimentally compared *STHoles* histograms against existing multidimensional histogram techniques. We showed that, in many cases, *STHoles* results in more accurate selectivity estimations for the expected workload than those for *GenHist* histograms, a technique that requires at least 5 to 10 scans over the whole data set during histogram construction and that generally dominates the other existing multidimensional histograms in accuracy. Finally, we established that the overhead of our technique is acceptable through an implementation over Microsoft SQL Server 2000. As future work, we plan to extend the estimation techniques to complex queries involving joins in addition to selection conditions. For such queries, these estimations might involve several *STHoles* histograms. This extension will enable seamless integration of *STHoles* histograms into commercial database management systems.

References

[1] A. Aboulnaga and S. Chaudhuri. Self-tuning histograms: Building histograms without looking at data. In *Proceedings of the 1999 ACM International Conference on Management of Data (SIGMOD'99)*, 1999.

[2] R. Avnur, J. M. Hellerstein, B. Lo, C. Olston, B. Raman, V. Raman, T. Roth, and K. Wylie. Control: Continuous output and navigation technology with refinement on-line. In *Proceedings of the 1998 ACM International Conference on Management of Data (SIGMOD'98)*, 1998.

[3] C. Blake and C. Merz. UCI repository of machine learning databases, 1998.

[4] S. Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 1-3, 1998, Seattle, Washington*, 1998.

[5] C.-M. Chen and N. Roussopoulos. Adaptive selectivity estimation using query feedback. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, 1994.

[6] D. Donjerkovic, Y. Ioannidis, and R. Ramakrishnan. Dynamic histograms: Capturing evolving data sets. In *Proceedings of the 16th International Conference on Data Engineering*, 2000.

[7] V. Gaede and O. Günther. Multidimensional access methods. *Computing Surveys*, 30(2), 1998.

[8] P. B. Gibbons, Y. Matias, and V. Poosala. Fast incremental maintenance of approximate histograms. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases*, 1997.

[9] D. Gunopulos, G. Kollios, V. J. Tsotras, and C. Domeniconi. Approximating multi-dimensional aggregate range queries over real attributes. In *Proceedings of the 2000 ACM International Conference on Management of Data (SIGMOD'00)*, 2000.

[10] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM International Conference on Management of Data (SIGMOD'84)*, 1984.

[11] Y. Ioannidis. Query optimization. In *Handbook for Computer Science*. CRC Press, 1997.

[12] Y. E. Ioannidis and V. Poosala. Balancing histogram optimality and practicality for query result size estimation. In *Proceedings of the 1995 ACM International Conference on Management of Data (SIGMOD'95)*, 1995.

[13] Y. E. Ioannidis and V. Poosala. Histogram-based approximation of set-valued query-answers. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, 1999.

[14] H. V. Jagadish, N. Koudas, S. Muthukrishnan, V. Poosala, K. C. Sevcik, and T. Suel. Optimal histograms with quality guarantees. In *Proceedings of the Twenty-fourth International Conference on Very Large Databases (VLDB'98)*, 1998.

[15] J.-H. Lee, D.-H. Kim, and C.-W. Chung. Multi-dimensional selectivity estimation using compressed histogram information. In *Proceedings of the 1999 ACM International Conference on Management of Data (SIGMOD'99)*, 1999.

[16] D. B. Lomet and B. Salzberg. The hB-Tree: A multiattribute indexing method with good guaranteed performance. *ACM Transactions on Database Systems (TODS)*, 15(4), 1990.

[17] Y. Matias, J. S. Vitter, and M. Wang. Wavelet-based histograms for selectivity estimation. In *Proceedings of the 1998 ACM International Conference on Management of Data (SIGMOD'98)*, 1998.

[18] M. Muralikrishna and D. J. DeWitt. Equi-depth histograms for estimating selectivity factors for multi-dimensional queries. In *Proceedings of the 1988 ACM International Conference on Management of Data (SIGMOD'88)*, 1988.

[19] M. Muralikrishna and D. J. DeWitt. Equi-depth histograms for estimating selectivity factors for multidimensional queries. In *Proceedings of the 1988 ACM International Conference on Management of Data (SIGMOD'88)*, 1988.

[20] S. Muthukrishnan, V. Poosala, and T. Suel. On rectangular partitionings in two dimensions: Algorithms, complexity, and applications. In *Database Theory - ICDT '99, 7th International Conference*, 1999.

[21] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems*, 9(1), 1984.

[22] F. Olken and D. Rotem. Random sampling from database files: A survey. In *Statistical and Scientific Database Management, 5th International Conference SSDBM*, 1990.

[23] B.-U. Pagel, H.-W. Six, H. Toben, and P. Widmayer. Towards an analysis of range query performance in spatial data structures. In *Proceedings of the Twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1993.

[24] G. Piatetsky-Shapiro and C. Connell. Accurate estimation of the number of tuples satisfying a condition. In *Proceedings of the 1984 ACM International Conference on Management of Data (SIGMOD'84)*, 1984.

[25] V. Poosala and V. Ganti. Fast approximate answers to aggregate queries on a data cube. In *11th International Conference on Scientific and Statistical Database Management, Proceedings, Cleveland, Ohio, USA, 28-30 July, 1999*, 1999.

[26] V. Poosala and Y. E. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *Proceedings of the Twenty-third International Conference on Very Large Databases (VLDB'97)*, 1997.

[27] V. Poosala, Y. E. Ioannidis, P. J. Haas, and E. J. Shekita. Improved histograms for selectivity estimation of range predicates. In *Proceedings of the 1996 ACM International Conference on Management of Data (SIGMOD'96)*, June 1996.

[28] J. T. Robinson. The K-D-B-Tree: A search structure for large multidimensional dynamic indexes. In *Proceedings of the 1981 ACM International Conference on Management of Data (SIGMOD'81)*, 1981.

[29] S. A. William, H. Press, B. P. Flannery, and W. T. Vetterling. *Numerical recipes in C: The art of scientific computing*. Cambridge University Press, 1993.

A Approximating Penalties

To implement the *penalty* function of Section 4.2.3 efficiently, we can in principle maintain a two-dimensional array P in memory, where $P[i,j]$ is the penalty of merging buckets b_i and b_j. In this appendix we show how we can use an approximation of this array that requires only linear space in the number of buckets. In principle, instead of using the two-dimensional array P for "caching" penalties, we could maintain a pair of unidimensional vectors: $bestB$ and $bestP$. The value of $bestB[b_i]$ is the bucket b_j, among b_i's siblings and parent, for which $penalty(b_i, b_j)$ is minimal. The vector $bestP$ stores such penalty: $bestP[b_i] = penalty(b_i, bestB[b_i])$. We propose to "weaken" the definition of $bestB$, allowing sometimes to return a bucket with a relatively low, but not lowest, penalty. To do so, instead of *resetting* [9] at each step all the buckets that are involved in drills and merges, we slightly modify these algorithms in the following way:

[9]"Resetting" a bucket b means invalidating the value of $bestB[b]$ and the value of $bestB[b_j]$ for all buckets b_j such that $bestB[b_j] = b$, so those values will have to be calculated again before the next merge.

Parent-Child Merge Consider the merge of buckets b_c and b_p, where b_p is b_c's parent, and let b_n be the resulting bucket. We *only* reset bucket b_c, and we set $bestB[b_n]$ equal to the old $bestB[b_p]$. Also, all buckets b_i such that $bestB[b_i] = b_p$ get their $bestB$ value updated so that $bestB[b_i] = b_n$. The motivation for this approximation is that generally b_p and b_n are similar buckets, so the penalty function should also be similar when evaluating it with b_n instead of b_p.

Sibling-Sibling Merge Consider the merge of buckets b_1 and b_2, with common parent b_p. Suppose the resulting bucket is b_n. We reset buckets b_1 and b_2. Also, we reset only b_p's children whose $bestB$ value is some child of b_n (those buckets were siblings before the merge and became separated). Similarly, we reset only b_n's children whose $bestB$ value is some child of b_p (or b_p itself). In other words, we do not reset a bucket b_i if $bestB[b_i]$ is still a sibling (or the parent) of b_i. Consider Figure 23, where an arrow from bucket b_i to bucket b_j means that $bestB[b_i] = b_j$. After merging b_1 and b_2 to obtain b_n, we only reset b_3, since after the merge b_3 and b_5 are not siblings anymore. On the other hand, we do not reset b_5 or b_2.

Drill Hole Consider a bucket b that is drilled using bucket b_n. After drilling b_n, we only invalidate b's children whose $bestB$ value is some child of b_n (those buckets were siblings before the drill). Similarly, we invalidate only b_n's children whose $bestB$ value is some child of b (or b itself).

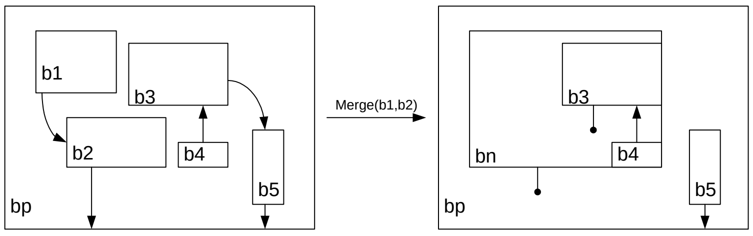


Figure 23: Resetting buckets after a Sibling-Sibling merge.