

Selectivity Estimation by Batch-Query based Histogram and Parametric Method

Jizhou Luo¹ Xiaofang Zhou² Yu Zhang² Heng Tao Shen² Jianzhong Li¹

¹Harbin Institute of Technology, China

²University of Queensland, Australia

{luojizhou,ljzh}@hit.edu.cn {zxf,zhang,shenht}@itee.uq.edu.au

Abstract

Histograms are used extensively for selectivity estimation and approximate query processing. Workload-aware dynamic histograms can self-tune itself based on query feedback without scanning or sampling the underlying datasets in a systematic and comprehensive way. Dynamic histograms allocate more buckets not only for the areas with most skewed data distribution but also according to users' interest. However, it takes long time to 'warm-up' (i.e., a large number of queries need to be processed before the histogram can provide a satisfactory coverage and accuracy). Thus, it is less effective to adapt with workload pattern changes. In this paper, we propose a novel online query scheduling algorithm which can significantly reduce the warm-up time for dynamic histograms. A parametric method is proposed to remedy the problem of inaccurate query selectivity estimation for the areas with poor histogram coverage. Experimental results demonstrate a significant effectiveness and accuracy improvement of our approach.

1 Introduction

Most commercial database systems maintain histograms for the purpose of selectivity estimation and approximate query processing (Lim, Wang & Vitter 2003). Typically, the process of building histograms involves sorting and partitioning the data into buckets, based on scanning or sampling the data. Histograms built in such a way are often called *static* histograms in a sense that, once being built, they will remain unchanged even when the underlying data distribution is changed over time. This type of histograms needs to be rebuilt periodically or when the error of selectivity estimation reaches a pre-specified threshold. In order to reduce the cost of building and maintaining histograms for very large datasets, self-tuning histograms have attracted growing attention recently (Aboulnaga & Chaudhuri 1999, Bruno,

Chaudhuri & Gravano 2001, Srivastava, Haas, Markl, Megiddo, Kutsch & Tran 2006, Loannidis 2003). Self-tuning histograms are built based on query feedback. It provides an inexpensive way to construct histograms for large datasets with a low up-front overhead. While such histograms can adapt well to the changes of the underlying data distribution, they typically assume that user queries follow a stable workload pattern. That is, newly arrived queries are more likely to visit the areas where previous queries just visited, and the areas visited by more queries in the past are likely to be visited more frequently in the future. This type of 'workload-aware' histograms concentrates on building high quality histograms for these 'hot' areas, at the expense of poor quality or even no coverage for other areas. For many database applications where the query workload patterns do change over time, this type of histograms may not work well. It is important to investigate self-tuning histograms that can adapt to the changes of both data distribution and workload patterns.

In this paper, we consider an online query scheduling method to speed up accuracy convergence process such that a histogram can reach a satisfactory average selectivity estimation accuracy more rapidly when workload pattern changes. This is achieved by giving a higher priority to the queries in the areas which are 'hot' (according to recently arrived and executed queries) and skewed (according to recent query feedback). On the other hand, there are always some queries which do not follow the current workload pattern. Estimation accuracy for the queries to these areas can be poor for self-tuning histograms, adversely affecting the performance of the entire system, often measured by the average estimation inaccuracy. This problem is more serious at the beginning of workload pattern changes. Instead of using uniform distribution assumption, a novel parametric selectivity estimation method is proposed in this paper for queries in the areas with poor or no histogram coverage. The information required for applying parametric estimation is simple and online maintainable. Our work in this paper is based on *STHoles* histograms (Bruno et al. 2001), but provides a significant improvement for convergence speed and overall estimation accuracy for queries with changing workload patterns.

The rest of the paper is organized as follows. Section 2 introduces notations, definitions and the basic idea of *STHoles*. An online query scheduling method is introduced in Section 3, and selectivity estimation for range queries using the novel parametric method in Section 4. A performance evaluation is reported in Section 5, with related work reviewed in Section 6.

Supported by ARC under Grant No.DP0663272; the National Natural Science Foundation of China under Grant No.60273082.

Copyright ©2007, Australian Computer Society, Inc. This paper appeared at the Eighteenth Australasian Database Conference (ADC2007), Ballarat, Victoria, Australia. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 63. James Bailey and Alan Fekete, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

We conclude this paper in Section 7.

2 Preliminaries

As we consider only range queries in this paper, we can assume that there is only one relation of n attributes. We also assume that an attribute contains only countable value coded as integers. Let $R = (X_1, \dots, X_n)$ be the schema of a relation, whose domain is $\mathcal{D} = (\mathcal{D}_1 \times \dots \times \mathcal{D}_n)$. Let r be an instance of R , and $\mathcal{V}_i \subseteq \mathcal{D}_i$ be the set of X_i values in r .

A hyperrectangle B in \mathcal{D} is defined as $B = \{(x_1, \dots, x_n) \in \mathcal{D} | v_l^i < x_i < v_u^i, i = 1..n\}$. B can be represented using two n -dimensional points $v_l = (v_l^1, \dots, v_l^n)$ and $v_u = (v_u^1, \dots, v_u^n)$, where $v_l^i = \min_{v \in B} v^i$ and $v_u^i = \max_{v \in B} v^i$. Thus, B can be denoted as $B = \langle v_l, v_u \rangle$. The volume of $B(v_l, v_u)$ is $vol(B(v_l, v_u)) = \prod_{i=1..n} (v_u^i - v_l^i)$. The topological relationships we are interested between two hyperrectangles B and B' are disjoint ($B \cap B' = \emptyset$), overlapping ($B \cap B' \neq \emptyset$) and containment (B is nested in B' , denoted as $B \subseteq B'$). A containment tree can be constructed for a set of hyperrectangles $B_i, 1 = 1..w$, such that 1) the root node is the entire space of interest, 2) a child node is nested inside a parent node, and 3) no sibling nodes are nested among each other.

A histogram \mathcal{H} for relation r is a collection of hyperrectangle and frequency pairs. That is, $\mathcal{H} = \{(B_i, f_i) | i = 1..m\}$, where f_i is the expected number of r tuples inside B_i (the points on a hyperplane of a hyperrectangle is assigned to only one hyperrectangle following the open-close hyperplane convention). The estimated frequency of r tuples in B_i is also denoted as $\mathcal{H}(B_i)$, and B_i is referred to as a bucket. For different types of histograms, all buckets in a histogram of r may form a partition, a cover, or a partial cover of \mathcal{D} or \mathcal{V} .

For a hyperrectangle $B_q = \langle v, v' \rangle$, a range query $q(B_q)$ against r retrieves all $\{x \in r | x \in B(v, v')\}$. B_q is called the query region of q . The collection of query result is the query feedback of q , and the number of tuples in the result collection is denoted as $act(r, q)$. We use $est(\mathcal{H}, q)$ to denote the estimated number of tuples returned from executing q against r according to histogram \mathcal{H} . If data distribution inside any bucket is assumed to be uniform, this estimation can be done in a straightforward way:

$$est(\mathcal{H}, q) = \sum_{B_i \in \mathcal{H}} \frac{vol(B_i \cap B_q)}{vol(B_i)} \mathcal{H}(B_i)$$

Next we introduce *STHoles* histograms (Bruno et al. 2001), which is a representative method to build and maintain histograms using query feedback. To simplify our discussion and for better visualization, we set $n = 2$ hereafter. However, our discussions are applicable to higher dimensions.

There are many different ways to construct the initial histogram. Assuming that initially the histogram is empty. Once a query is executed, the query feedback becomes available and can be used to identify any skewed data distribution by checking the precise number of tuples returned for the query in the areas overlapping with other existing buckets. Once a significant variation of data distribution is found for any part of a bucket, a ‘hole’ is created to give a more accurate expected frequency for that part using a new bucket, and the frequency of the existing

bucket affected will be adjusted with the new information. Some sophisticated algorithms are proposed in (Bruno et al. 2001) to determine bucket shapes and perform bucket mergers. This approach does not require a systematic/comprehensive scan of the underlying dataset, but allocates buckets according to user query feedback. This approach is called ‘workload-aware’, as the refinement of a histogram is directed by user’s collective interest (as per queries). It can also adapt to the changes of data distribution, as such changes are eventually reflected in query feedback that leads to changes of the histogram.

Assume bucket B in a *STHoles* histogram contain a number of holes (i.e., a set of buckets B_1, \dots, B_k , where these child buckets do not overlap by definition). The net volume and net frequency of B can be defined as

$$vol_{net}(B) = vol(B) - \sum_{i=1}^k vol(B_i)$$

and

$$f_{net}(B) = f(B) - \sum_{i=1}^k f(B_i)$$

The density of a bucket B is defined as

$$d(B) = \frac{f_{net}(B)}{vol_{net}(B)}$$

Define a workload W as a sequence of $|W|$ queries. A common metric to measure the performance of a histogram is the average absolute error over W :

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

In *STHoles*, a histogram may not fully cover the entire data space, thus there may not exist any buckets for certain areas. To make robust comparison across different datasets, a normalized absolute error metric is used in (Bruno et al. 2001) by introducing $est_{uni}(r, q)$ which is the estimated result size by assuming uniform data distribution for the areas where no histograms are available. That is,

$$E'(r, \mathcal{H}, W) = \frac{E(r, \mathcal{H}, W)}{E_{uni}(r, W)}$$

where

$$E_{uni}(r, W) = \frac{1}{|W|} \sum_{q \in W} |est_{uni}(r, q) - act(r, q)|$$

The *STHoles* approach makes an explicit assumption that workload pattern is stable. Therefore, histogram performance is evaluated by using a training workload to create a histogram, and then to use a validation histogram to compute the error metric, where the validation workload has the same distribution pattern as to that of the training workload (in terms of both coverage and query foci areas). For many database applications, queries do follow some workload patterns within a period, but the patterns may change over time. Not being able to divide queries into training and validation queries, the order

Notation	Description
id	query ID
B	query area
hot, gap	hotness scale and gap value
est, rst	the estimated and actual number of tuples
$children$	pointers to children nodes
$parent$	pointer to the parent node
$next$	pointer to the next execution candidate node

Table 1: A table of attributes.

of executing queries becomes important as the feedback from a query can impact on other queries. In other words, this problem is an online query scheduling problem with a continuous stream of range queries targeting variable query regions. We approach this problem from two aspects: 1) a fast-convergence online scheduling algorithm to give higher priority for the queries in the areas where more queries need to be observed to improve the overall quality of the histogram; and 2) a novel estimation method using concise and online maintainable information to estimate the selectivity for queries in an area with no or poor histogram coverage.

3 Online Query Scheduling

In this section, we propose a scheduling algorithm based on both the recent past query history and the queries in the buffer (i.e., arrived but yet to be processed). For each query q , we define two measures first: hotness scale and the gap value.

The *hotness scale* of q in relation to a set Q of queries is defined as $hot(q, Q) = \sum_{i=1}^k S_i \times n_i$, where $\{S_i | i = 1..k\}$ is a set of disjoint rectangles inside $q.B$ generated by overlaying all $q'.B$ together, $q' \in Q$, and n_i is the number of queries in Q whose query regions cover S_i (because each S_i is obtained by a complete overlay of all query regions in Q , the case for a query in Q to partially cover any S_i does not exist). Intuitively, a query is hotter if it is in a larger area that many other queries also are interested in. Obviously, the hotness scale changes when new queries arrive.

For two regions B_1 and B_2 , $B_1 \subseteq B_2$, their density difference can be measured using a *gap* value, which is defined as

$$gap(B_1, B_2) = \frac{d(B_1)}{d(B_2)} = \frac{f_{net}(B_2)}{vol_{net}(B_2)} \times \frac{vol_{net}(B_1)}{f_{net}(B_1)}$$

Clearly, a *gap* value falls into range $(0, +\infty)$, where $gap = 1$ means that there is no difference between the density of B_1 and B_2 ; $0 < gap < 1$ means that the inner bucket B_1 has a lower density (and a smaller value of *gap* indicates a larger density difference); and $gap > 1$ means that the inner bucket B_1 has a higher density (and a larger value of *gap* indicates a larger density difference).

3.1 Query Tree Construction

Assume that continuously arriving query stream is stored in a buffer. The buffer size is limited but large enough so we do not need to consider the problem of query buffer overflow. Further, we assume that a query has a deadline for it to be executed, but in general queries can be deferred for execution within the specified time limit in favor of a better overall system performance. This is a common assumption for

batch query processing. Once a new query arrives, the scheduling algorithm must compute its hotness and gap values, and update the changes of these values in relation to other yet-to-execute queries in the buffer. The queries in the buffer need to be organized in such a way that updates when new queries arrive and the algorithm to select queries for execution can be efficiently supported. We propose to organize the current queries according to their natural containment relationship. Table 1 lists the attributes recorded for each node in the tree.

Algorithm 1: InsertQuery

```

input:  $q$ : arriving query,  $QT$ : root of a query tree
output:  $QT$ : query tree updated with  $q$ 
01.  $t \leftarrow QT$ ;  $q.hot \leftarrow vol(q)$ ;
02. while  $t \neq null$  {
03.   if  $q.B \subseteq t.B$  {
04.      $t.hot \leftarrow t.hot + q.hot$ ;
05.      $t \leftarrow$  the first unvisited child of  $t$ ;
06.   } else { //  $q$  will be inserted here
07.     for each child node  $q' \in t.children$  {
08.       if  $q'.B \subseteq q.B$  // replace a child
09.          $q.hot \leftarrow q.hot + q'.hot$ ;
10.         let  $q'$  be a child of  $q$ , and  $q$  a child of  $t$ ;
11.       else { // exchange hotness with a sibling
12.          $q.hot \leftarrow q.hot + vol(q.B \cap q'.B)$ ;
13.          $q'.hot \leftarrow q'.hot + vol(q.B \cap q'.B)$ ;
14.       }
15.     }
16.      $t \leftarrow$  the next node to visit according to DFS;
17.   }
18. }
19. return  $QT$ ;

```

The root node of the tree is initialized to cover the entire space. The algorithm to insert a new query q into the tree is simple: a depth-first search (DFS) is performed on the current query tree to find all queries whose query regions fully contain $q.B$, and the hotness value for all the nodes traversed during the search will be updated. Algorithm 1 is the sketch of this algorithm. If $q.B$ is nested inside $t.B$, t is a node in the tree, $t.hot$ will be increased by $q.hot$ and the search will continue for all of children nodes of t (lines 3-5). If $q.B$ is not nested inside a node t , then q will be inserted as a child of t , and the search of this branch of the tree is completed and moved to another part of the query tree according to DFS until no more nodes to search (lines 7-13). When q is to be inserted as a child of t , there are two different cases: 1) a child node q' of t are nested inside $q.B$. In this case, q' will become a child of q and pass its hotness value to q before q is inserted as a child of t (lines 8-10); 2) a child node q' of t is not nested in $q.B$, in this case, the volume of overlapping areas of q and $q'.B$ will be added to the hotness value of both q and q' (lines 11-13). Note that a query may be nested inside more than one sibling nodes; in this case, the search will be followed along all these subtrees. That is, a query can be inserted into multiple places in the tree. It is not difficult to see that the above algorithm maintain the hotness scale values for all nodes in the tree after a new query is inserted. The reason for a query q to be inserted into multiple places in the tree is that all nodes that contain $q.B$ must be proessed to get its share of increased hotness scale resulting from q .

3.2 Query Scheduling

In order to minimize the average error for a sequence of queries, a query is selected to execute such that its feedback can benefit the process of tuning the histograms most for the remaining and future queries. Before discussing the scheduling algorithm, we explain two observations.

observation 1. *In order to make a histogram more adaptive to workload pattern changes, one should test the queries on hot spots first. If two spots' hotness degrees are identical, one should test the queries which are as close as possible to those areas where data is more disproportionately distributed. This strategy can speed up accuracy convergence of the histogram.*

observation 2. *If the density of bucket B_1 is much smaller than that of bucket B_2 , then the data distribution in bucket B_2 is more interesting than that in bucket B_1 . So, it is rationale to choose a child query of B_2 to execute in the next step in order to speed up the convergence of the histogram. Otherwise, the data distribution in bucket B_1 is more interesting; thus it is better to choose a sibling query of B_2 to execute in the next step in order to speed up the convergence.*

Based on these two observations and using the query tree data structure, a simple scheduling algorithm can be used to select the top k queries from the query tree according to their hotness values. These k queries will be further selected to choose the one with the smallest *gap* value. Figure 1 illustrate the rationale behind this heuristics of choosing the smallest *gap* value. This can be understood by considering the following three cases:

Case 1: if $g_1 < g_2 < 1$, this means that $q_1.parent$ is more dense than $q_1.parent.parent$ comparing with its counterpart of $q_2.parent$. Based on the above heuristics rule, q_1 is selected to execute.

Case 2: if $g_2 > g_1 > 1$, this means that $q_2.parent$ is less dense than $q_2.parent.parent$ comparing with its counterpart of $q_2.parent$. So q_1 has a higher priority.

Case 3: if $g_2 > 1 > g_1$, it is straightforward to tell $q_1.parent$ is much more dense than $q_1.parent.parent$ comparing with its counterpart of $q_2.parent$. q_1 will be executed first.

The above three cases all suggest to select a query with the smallest *gap* value. Note that a *gap* value is only known after some queries are executed. When a new *query* is inserted, its *gap* value is inherited from its parent, and the *gap* value for the root is defined as 1. Once a query q is executed, $q.rst$ is set to the number of tuples in the query result, and for each children nodes q' of q , $q'.gap = d(q.parent.B)/d(q.B)$. In other words, the *gap* value of a node is the ratio of the densities of its parent and grand-parent nodes. This is important because the *gap* value of a node is used to prioritize query execution (i.e., its own density is unknown), so its *gap* value indicates if its parent node is in an area with skewed data distribution.

Once q is selected, it is not removed from the query tree immediately; rather, $q.est$ is set to the current estimated selectivity of q (when q is inserted into the query tree, $q.est = q.rst = 0$). The reason of doing so is that executed queries are still useful for identifying the areas with large hotness and *gap* values. These executed queries, as explained next, will only be deleted when the query buffer is about to run out for new arrivals. $q.rst$ is set to the actual number of tuples in the result of q after its execution. It is

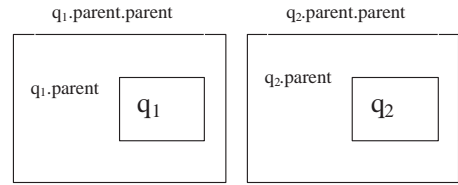


Figure 1: Scheduling order with *gap* value

mentioned before that a query may be inserted into more than one place in the tree. Multiple instances of the same query are treated as different queries in the scheduling algorithm, except that they will be executed only once and are deleted together.

3.3 Discussions

Building Histograms The scheduling algorithm proposed in this paper is an orthogonal improvement to the *STHoles* idea. With the optimized order of query execution, the basic ideas and algorithms in *STHoles* are still applicable here. A noticeable difference between our method and *STHoles* is how the entire data space is approximated in the histogram. In *STHoles*, the histogram is initialized to be empty; in our method, the initial histogram is a single bucket that covers that entire space of interest with uniform distribution. A uniform distribution about a very large area can lead to a considerably worse overall estimation accuracy error; thus, *STHoles* avoids this problem by making an assumption that a consistent workload pattern means that, after a training period, no new queries will retrieve data from previously unvisited region. For the query stream environment considered in this paper, this assumption can no longer be made. Further, consider the case where a query to a large region is followed by a number of queries targeting some small areas. This is equivalent to the problem of how to deal with the entire space. We address this problem by a novel parametric selectivity estimation method we propose in this paper. Another difference comparing to *STHoles* is that a different merger algorithm is used in our approach, as will be discussed later.

Query Deletion Once a query is executed, it needs to be removed from the query tree. Otherwise the query buffer will reach its limit and new queries can not be stored for optimization. However, if a query is deleted from the query tree immediately after execution (and its children nodes will be ‘promoted’ one level up), then the query tree tends to be a flat structure and becomes ineffective to trace the users’ interests. This may also cause the histogram oversensitive to users’ interest change (such as a small variation of workload pattern), as buckets are moved around too quickly and affected too much by the new arrivals without the balancing factor of recent past queries. To overcome this problem, we store the executed queries in another First-In-First-Out queue of a pre-determined size. After query q is executed, q remains in the query tree but is also pushed into the queue. The oldest query will be removed from the queue when the queue is full, and only at this time, q is deleted from the query tree.

Starvation Using the scheduling algorithm described above, some queries may be forced to wait for a long time or even never get executed (because they are from a ‘cold’ area for example). While such wait-

ing has no negative impact on the error metric we try to minimize, they can cause problems from user’s side. It also has a side effect that the available size of the query buffer is reduced by the queries that cannot be cleared for too long. This problem can be solved by recording, for each query q , a pair of time values $q.t_1$ and $q.t_2$, $q.t_1 \leq q.t_2$, where t_1 is its arrival time and t_2 is the maximum tolerate time which can be a default time set by the system (i.e., 3 minutes after arrival) or a time set by the user. These two values can be used to let the queries that have been waiting for too long or approaching the deadline ‘jump the queue’. Using feedback-based histograms, however, means that these time-activated queries may not in the areas with good histogram coverage, thus may have a negative impact on the error metric measure. This problem is addressed in the next section using the parametric selectivity estimation method.

4 Selectivity Estimation

Traditionally, the data in a bucket is assumed to be uniform. Selectivity estimation, therefore, can be done easily as mentioned before. However, the whole idea of *STHoles* is built on the observation that data inside a bucket may not be uniform (thus, ‘holes’ are opened inside a bucket once a significant variation of data distribution is found in a bucket). An interesting case is how to estimate the selectivity for a region inside a large bucket which consists of a number of hole buckets. We argue that if the query region is close to those hole buckets in a large bucket, a better estimation can be obtained by considering the frequency of those hole buckets. The situation of estimating selectivity for a query in an area which has no histogram is in the same category. We propose a parametric method to estimate the selectivity for areas close to the child buckets.

Algorithm 2: EstInBucket

input: B : a bucket of \mathcal{H} ; q : a query covered by $r(B)$
 $r(B)$ is the rectangle bounded by the boundary of B
output: S : selectivity estimation of q

1. $E \leftarrow 0$;
2. For each child bucket B' of B ;
3. $q_B \leftarrow q \cap r(B)$;
4. if $q_B \neq \emptyset$ then $E \leftarrow E + \text{EstInBucket}(q_B, B')$;
5. $q \leftarrow q \setminus q_B$;
6. if $cr(B) < \text{threshold}$ then $E \leftarrow E + \int_q \varphi(x, y) \, dx dy$;
7. else $E \leftarrow E + \frac{\text{vol}(q)}{\text{vol}_{\text{net}}(B)} * f(B)$;
8. return E ;

Given a user query q , its selectivity is estimated by a recursive algorithm $\text{EstInBucket}(q, B)$, as shown in Algorithm 2. $\text{EstInBucket}(q, B)$ recursively estimates the selectivity of the intersection area between q and each child bucket $B' \in \text{child}(B)$, where $\text{child}(B)$ consists of all the child buckets of B (lines 2-5). If the cover ratio C_r is less than a pre-specified threshold, a density function $\varphi(x, y)$ is proposed to capture the data distribution around each child bucket (line 6). In other words, we use child buckets’ information to estimate selectivity in bucket B . However, if the cover ratio C_r is greater than the threshold, we assume all the tuples in B are uniformly distributed and estimate the selectivity as $f_{\text{net}}(B)$ times the ratio of the volume of q falling in B over $\text{vol}_{\text{net}}(B)$ (line 7). We defer the discussion on density function $\varphi(x, y)$ to the next subsection.

4.1 Density Function

In this subsection, we deduce the density function $\varphi(x, y)$ used in Algorithm 2. Our idea is based on the following observation.

observation 3. *The difference between the data distribution in the area around a bucket and that in the bucket is unlikely to change dramatically.*

This observation is based on the fact that bucket boundaries are determined by user queries, rather than some optimal separation based on data distribution. This observation implies that information about data distribution in each bucket can be used to approximate the data distribution of the area nearby. Without a uniform distribution assumption, an appropriate density function needs to be identified to describe data distribution.

Define a *barycenter* (i.e., center of mass) of a bucket as the point in the bucket where the frequency of the bucket can be viewed as concentrated on that point. If data are uniformly distributed within the area bounded by the boundary of bucket, this barycenter should be the center of the bucket. However, when data are not uniformly distributed, i.e., the area has been separated by hole buckets, it is nontrivial to compute the barycenter, as the center moves with respect to each insertion, deletion and frequency update of a child bucket. To determine the barycenter $(\bar{x}_{B_i}, \bar{y}_{B_i})$ for bucket B_i in \mathcal{H} , we use the following propositions.

Proposition 1. *Suppose bucket A is the only child bucket of bucket B , $d(A)$ and $d(B)$ are their densities under uniformly distribution assumption respectively, then the barycenter of bucket B can be calculated as follows:*

$$\bar{x}_B = \frac{1}{2f(B)} \sum_{X \in \{A, B\}} (d(X) - d(P(X))) F_1(X)$$

$$\bar{y}_B = \frac{1}{2f(B)} \sum_{X \in \{A, B\}} (d(X) - d(P(X))) F_2(X)$$

where $P(X)$ is the parent of X , $d(P(B)) = 0$ and

$$F_1(X(v_l, v_u)) = (v_u^2 - v_l^2)([v_u^1]^2 - [v_l^1]^2),$$

$$F_2(X(v_l, v_u)) = (v_u^1 - v_l^1)([v_u^2]^2 - [v_l^2]^2).$$

Proof. According to the formula calculating the barycenter of an area in the plane with density function $\rho(x, y)$, we have

$$\bar{x}_B = \frac{1}{f(B)} \int \int_B x \rho(x, y) \, dx dy$$

$$= \frac{1}{f(B)} \left(\int \int_B x d(B) \, dx dy + \int \int_A x (d(A) - d(B)) \, dx dy \right)$$

The result of the expression is what we expected. The calculation of \bar{y}_B is similar. \square

Proposition 2. *Let B as a bucket in \mathcal{H} , A as an arbitrary hole bucket of B , $d(A)$ as density of A , and $p(A)$ as the parent of bucket A . By setting $d(P(B)) = 0$, the barycenter of B can be computed as follows under the uniformly distribution assumption:*

$$\bar{x}_B = \frac{1}{2f(B)} \sum_A (d(A) - d(P(A))) F_1(A)$$

$$\bar{y}_B = \frac{1}{2f(B)} \sum_A (d(A) - d(P(A))) F_2(A).$$

Proof. By proposition 1, we can use the induction on the number of levels in the bucket-tree rooted at B to verify this proposition easily. \square

Once the barycenters of buckets are calculated, we can use a parametric function to model the data distribution around these buckets as follows. Let B be a bucket of histogram \mathcal{H} , point (\bar{x}_B, \bar{y}_B) be its barycenter, ρ_B be the density of point (\bar{x}_B, \bar{y}_B) in B , and u_B be the shortest distance from (\bar{x}_B, \bar{y}_B) to the boundary of B . According to Observation 3, barycenter's density is supposed to be maximal and the density around this point decreases continuously in the manner characterized by a parametric function $g_\theta(t)$, where θ is the parameters to be determined and t represents the distance from the query to (\bar{x}_B, \bar{y}_B) . Several issues need to be considered. Firstly, the valid radius of parametric function $g_\theta(t)$ must be determined (within and only within this radius the barycenter of this bucket can be used for estimation). Secondly, the number of parameters in function $g_\theta(t)$ must be determined. And thirdly, the type of function should be determined. we discuss each of them below.

Let us consider the radius of the parametric function fist. Since the histogram is built from query feedback, the frequencies in each bucket and all its child buckets are known. That is, the data density of bucket B can be computed precisely and is not affected by any other buckets. Thus, the valid radius of each parametric function $g_\theta(t)$ cannot exceed the distance dis_B from (\bar{x}_B, \bar{y}_B) to its nearest barycenter. Hence, we get have the following condition (1).

$$g_\theta(dis_B) = 0 \quad (1)$$

Each parametric function can only be used to estimate the data distribution around its owner bucket. When a query is far away from all buckets, there is no buckets to be used for selectivity estimation. In this case, we follow the assumption of uniform density. The idea above is based on the principle "more information we known, better accuracy we can get."

Next, let us determine the number of parameters in function $g_\theta(t)$. Except condition (1), we only know the average data density d_B in bucket B . The exact density at the barycenter is not available. Generally, the influence form one barycenter reduces with the distance. Denote the density of point p , which is the nearest point on the boundary of B to its barycenter (\bar{x}_B, \bar{y}_B) , as $d(B)$. We get condition (2)

$$g_\theta(u_B) = d(B) \quad (2)$$

Thus, we have only two conditions to determine a parametric function $g_\theta(t)$ and the number of parameters needed for this function is at most 2.

It is more complex to select the type of the parametric function. A polynomial function with only two parameters such as $at + b$, $at^2 + bt$, $at^2 + b$, can be used. When the data distribution is known, a more accurate function can be applied. For example, we can choose ae^{-bt} as the function if we know that data follows Gaussian distribution approximately. After the function type is determined, function parameters can be determined by using condition (1) and condition (2). Denote the obtained function for bucket B as $g_B(t)$, $t \in [u_B, dis_B]$. This function can be easily extended to be defined for the whole space as below.

$$G_B(t) = \begin{cases} d(B) & t < u_B \\ g_B(t) & u_B < t \leq dis_B \\ 0 & otherwise \end{cases} \quad (3)$$

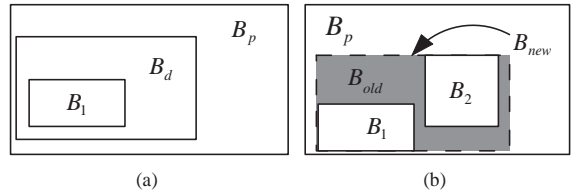


Figure 2: Buckets Update

If a query point p is covered by more than one bucket, its density estimation is a combination from the influences from all these buckets. Thus, this density function is

$$\varphi(x, y) = \sum_{B \in child(B)} G_B(dis((x, y), (\bar{x}_B, \bar{y}_B))).$$

4.2 Dynamic Maintenance of Histogram

With more buckets added, a histogram will reach the size of available memory allocated. In order to be adaptive to the changes of underlying data distribution and user's interest, some existing buckets need to be merged, by considering two different cases: parent-child buckets merge (where a parent bucket merges with one of its child bucket) and sibling-sibling merge (where two buckets under the same parent bucket merge).

For the first case, we scan all the buckets' gap value and find out the minimum gap value. Denote the bucket with minimum gap value G_{min} as B_{min} . Data density in B_{min} is the closest to that of its parent bucket. This means least penalty will be caused if B_{min} is merged with its parent bucket. Figure 2 (a) illustrates an example. After B_1 is created in B_d , the actual density of B_d is similar to B_p . Thus, B_d needs to be merged with B_p .

For the other case, to compare two sibling buckets (B_1 and B_2) on the same level under parent bucket B_p , we find the smallest hyperrectangle B_{new} that encloses both B_1 and B_2 (Bruno et al. 2001). Figure 2 (b) illustrates an example. In general, B_{new} contains old part B_{old} (the shadowed region in the example) from B_p . As we are able to calculate the area of B_{old} , we can easily get the frequency of B_{old} approximately by using

$$f(B_{old}) = f(B_p) \frac{V(B_{old})}{V(B_p)}$$

In the mean while, B_p becomes $B_{p'}$, volume and frequency are updated respectively. Having $B_{p'}$ and B_{new} 's volume and frequency, we get a new gap value G_{new} for B_{new} . There are two drawbacks in sibling-sibling merge. On one side, the identification of B_{new} is very expensive because we must test if any two sibling buckets forms as the configuration in figure 2 (b) without any other sibling bucket falling into the shadowed region. On the other side, $f(B_{old})$ is an estimation number, which may lead to an misleading.

Based on the above discussion, we give parent-child merge a higher priority than sibling merge. That is, we determine a parent-child merge candidate bucket B_{min} globally and then to determine whether there is a better sibling merge candidate bucket B_{new} such that $G_{new} < G_{min}$. If so, we will merge

some two sibling buckets into B_{new} ; otherwise, we merge B_{min} with its parent.

After two buckets are merged, the barycenter of each bucket needs to be updated, as well as the gap values of the buckets surrounding the merged bucket. Suppose the barycenter of a bucket B is (\bar{x}_B, \bar{y}_B) , and a new query causes updating the histogram by inserting a new bucket B_I somewhere in bucket B and deleting a bucket B_D in bucket B_1 . It is possible that $B = B_1$ (see Figure 2). Now we need calculate the barycenter of the updated bucket B and B_1 . Since if $B = B_1$, we can finish this update by first calculating the updated barycenter for an insertion in B following by calculating the updated barycenter for a deletion in B . So, we only consider the update caused by a single operation, insertion or deletion.

Proposition 3. *Let B is a bucket of histogram \mathcal{H} , $d(B)$ is the density of B and (\bar{x}_B, \bar{y}_B) is the barycenter of B . If a bucket B_I with density $d(B_I)$ is insert into B and causes the density of B changed as $d'(B)$, then the barycenter of B can be updated as follows.*

$$\begin{aligned}\bar{x} &= \bar{x} + (d'(B) - d(B)) \left[\sum_{X \in \text{child}(B)} F_1(X) - F_1(B) \right] \\ &\quad + (d(B_I) - d(P(B))) F_1(B_I) \\ \bar{y} &= \bar{y} + (d'(B) - d(B)) \left[\sum_{X \in \text{child}(B)} F_2(X) - F_2(B) \right] \\ &\quad + (d(B_I) - d(P(B))) F_2(B_I)\end{aligned}$$

Proof. Calculate the barycenter of B before and after insertion with proposition 2, and subtract the former from the latter them. \square

Proposition 4. *Let B is a bucket of histogram \mathcal{H} , $d(B)$ is the density of B and (\bar{x}_B, \bar{y}_B) is the barycenter of B . If a child bucket B_D of B with density $d(B_D)$ is merged with bucket B and causes the density of B changed as $d'(B)$, then the barycenter of B can be updated as follows.*

$$\begin{aligned}\bar{x} &= \bar{x} + (d'(B) - d(B)) [F_1(B) \\ &\quad - \sum_{X \in \text{child}(B) \setminus \{B_D\}} F_1(X)] - (d(B_D) - d'(B)) F_1(B_D) \\ \bar{y} &= \bar{y} + (d'(B) - d(B)) [F_2(B) \\ &\quad - \sum_{X \in \text{child}(B) \setminus \{B_D\}} F_2(X)] - (d(B_D) - d'(B)) F_2(B_D)\end{aligned}$$

The proof of this proposition is similar as above.

5 Experimental Evaluation

In this section, we compare the performance of our proposed methods with *STHoles* in the context of changing data distribution and changing workload patterns. In addition to comparing their estimation accuracy, two other impotent factors are also considered: convergence speed and memory sizes. We call our technique *BQHist* hereafter.

In our experiment, a real-world dataset is used: the State Regional Ecosystem (SRE) dataset with 398, 464 polygons representing different species' coverage. Synthetic datasets of 2- and 3-dimensions are also used: one with Gauss distribution and one with logistic distribution. The Gauss distribution data

consists of a number of Gaussian bells with a pre-determined standard deviation. The total number of tuples is 1,600,000. A Zipf distribution regulates the number of tuples in each Gaussian bell. The logistic dataset is generated in a similar way.

5.1 Estimation Accuracy

For different datasets, we examine estimation accuracy of our algorithm using parametric or uniform estimation at different cover ratios, as detailed in different types of workload. The capacity of the query buffer is set to 200 throughout the experiment (i.e., at any time no more than 200 queries can be cached in the buffer for scheduling).

Workload 1: This workload pattern is designed to compare estimation accuracy. It consists of 1050 queries, with 50, 100, 300 and 600 queries to retrieve 2%, 1%, 0.4% and 0.3% of the whole dataset respectively. These queries are distributed uniformly in the region, and their combined query areas cover 50% of the total area.

Result: Figure 3 shows the accuracy after 250 queries are executed on SRE dataset. *BQHist* with uniform estimation performs at least 30% more accurate than *STHoles*. This figure also shows that *BQHist* with uniform estimation performs better than *BQHist* with parametric estimation, which is consistent with what we discussed before, as this workload distribution is not skewed much.

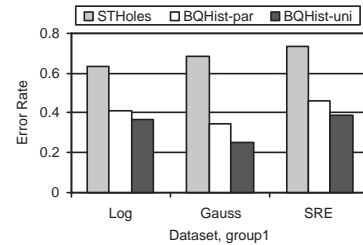


Figure 3: Accuracy of different estimation methods

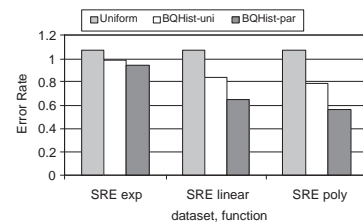


Figure 4: Accuracy of different functions

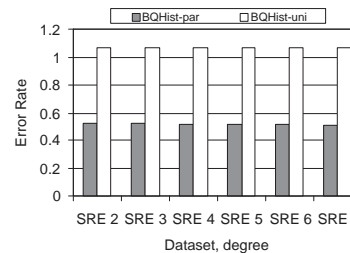


Figure 5: Accuracy of different polynomial powers

Workload 2: This workload is designed to examine the cases where query areas are concentrated to a

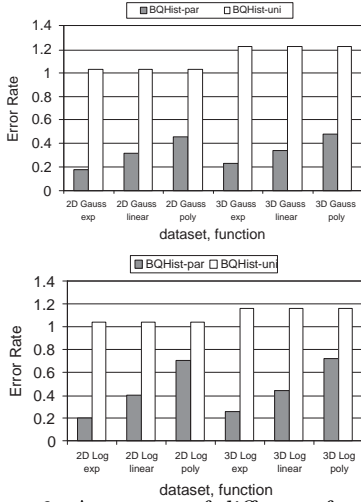


Figure 6: Accuracy of different functions

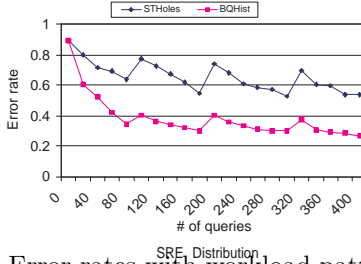


Figure 7: Error rates with workload pattern change

small area. It still uses 1050 queries but the total area covered by these queries occupy just 10% of the total area. Further, these queries are nested to 10 levels, with 10 top-level queries retrieving 3% of tuples from the dataset, and each query is followed by two child queries which retrieve 10% of the data in its parent.

Result: Figure 4 shows the normalized absolute errors of using different parametric functions and uniformity based estimation on SRE dataset. This figure clearly shows that the *BQHist* can achieve a much higher accuracy comparing to the uniformity-based estimation, for all the three sample functions used; and for the case of polynomial function, a 50% improvement has been achieved. The reason here is that SRE is not a highly skewed dataset, which means that the frequencies of the areas around a bucket do not change dramatically from that in the bucket. A polynomial function reflects this behavior well, as its value does not drop quickly in a short distance away from the starting point. A uniformity-based estimation, on the other side, results in an underestimation around a bucket's child buckets and an overestimation at the area far from the child buckets.

Figure 5 shows that the differences among polynomial's powers (on SRE dataset) do not have a significant impact on estimation accuracy. This insensitivity is a nice property, as these powers are typically not easy to determine. Figure 6 shows the normalized absolute errors of using different parametric functions on the four synthetic datasets. Since the distribution of the synthetic data is known in prior, one can predict which function has better performance, as the more similar they are, the better accuracy the result is. Because exponential function is the same as Gauss distribution, it leads to the best accuracy. Since logistics distribution is the most similar among three

functions, it delivers good results as well. However, the polynomial function is constantly robust across all the tests with error rate less than 50% of uniform distribution error rate. Furthermore, we notice that with the number of dimension increasing to there, the error rates only change slightly.

5.2 Changing Workload Patterns

Workload 3: This workload is designed to test the impact of changing workload patterns. Four clusters of queries are used, where the areas for the queries in one cluster do not overlap with the queries from another cluster. For each cluster, 400 queries with a uniform selectivity of 0.25% are randomly distributed in the area of the cluster. During the tests, only 100 queries in a cluster are processed, the queries from another cluster are used straightaway). The normalized error is calculated with the previously tested queries.

Result: Figure 7 shows the results on SRE (experimental results on other datasets, which confirm a similar trend, are omitted due to space limit). It is clear that *BQHist* results in a lower error rate than *STHoles* consistently, especially when the workload pattern moves from one cluster to another.

5.3 Convergence Speed and Cache Capacity

Workload 4: This workload is designed to test the convergence speed and the impact of cache capacity (as it can be argued that the additional memory required by *BQHist* can be used by *STHoles* to improve its performance by using more buckets). 50 queries from Workload 1 and other two group of queries are used as training queries, and 1050 validation queries following the same distribution as the training queries are used to measure the normalized absolute error. The initial queries of both sequential and batched order are the same. We use three groups of 1050 queries with 25% cover ratio distributed in the region following the Gaussian, Zipfian (with the z parameter, which indicates distribution skewness, set to 2) and uniform distribution. The selectivity ranges from 1% (50 queries), 0.25% (200 queries), to 0.05% (800 queries).

Result: Figure 8 shows the results on different datasets, methods and cache capacities. In general, the error rates drop sharply in around first 300 queries when system is able to cache all the queries (*BQHist-full*). When the *QT* tree is flat (i.e., too many query nodes are on top level), *BQHist* becomes less effective (shown in the second part of Figure 8) as the gap value's function is not fully utilized. However, the result is still about 20% better than *STHoles*. The performance of *BQHist-300* is between *STHoles* and *BQHist-full*, which indicates that a larger cache leads to a better performance. Figure 9 demonstrates that *BQHist* is at least 10% more accurate than *STHoles* with randomly distributed queries.

Workload 5: A scheduling algorithm is proposed in subsection 3.2 to select the top k queries from the query tree according to their hotness value. This workload is designed to test the impact of k values on our the performance of our method. In the first experiment, 400 queries in a single cluster from Workload 3 are used. In the second experiment, queries from all Workload 3 clusters are involved.

Result: Recall that these top k queries are selected according to the hotness value in the query

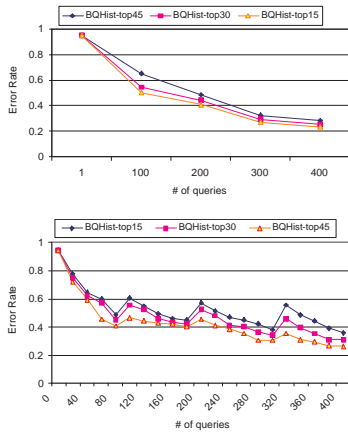


Figure 8: Convergence rates

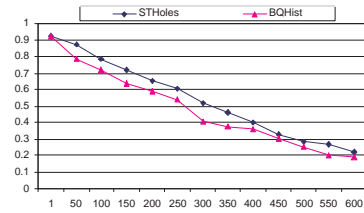


Figure 9: Convergence for random queries

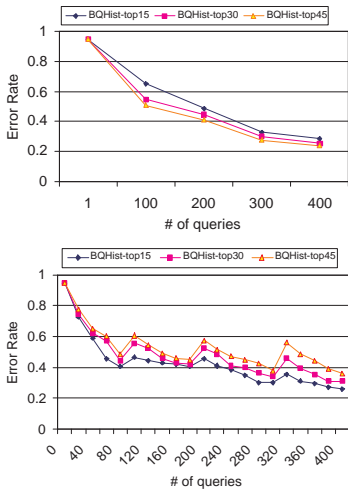


Figure 10: Accuracy using hotness and gap

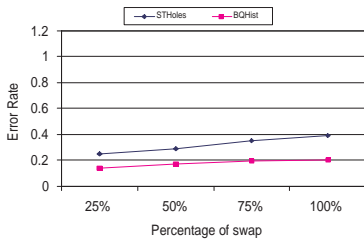


Figure 11: Accuracy after dataset updates

tree. Thus, a larger k value deemphasize the importance of the hotness value, giving more room to select query execution according to their gap values. The first figure in Figure 10 shows that the convergence speed increases when the k value decreases (as all queries are from the same cluster, thus it is more important to cover disproportional area than hot areas). In other words, the gap value is a better performance indicator than the hotness indicator. On the contrary, the second figure shows that a bigger k value can make our method more adaptive to workload pattern changes, since the hot spots, which change when workload patterns change, have a better chance to be visited than those disproportional areas.

5.4 Effect of Updates

One of the benefits of building histograms based on batch query feedback is that such a histogram keeps updating when the data distribution changes. This final set of tests evaluates how *BQHist* adapts to data distribution changes.

Workload 6: In this experiment, we progressively swap the Gaussian and Logistics datasets and test the accuracy. We start with the Gaussian dataset and test with 350 and 500 queries respectively. Then we swap a certain percentage of data with the Logistic dataset and test with another 350 and 500 queries respectively. Finally we use validation queries to test the accuracy. To simulate the reality that queries keep coming in, we set up a query pool with 2100 queries with random order. For *STHoles*, we choose the first 350 and 500 queries as validation queries. For *BQHist*, we organize the first 1050 queries in the *QT* tree for the first half of workload, but we only use 350 and 500 queries among them respectively. Then, we add another 1050 queries in *QT* and test the same amount as first half in second half workload.

Result: Figure 11 shows the accuracy of histogram adapts to data distribution changing from the Gaussian to Logistic datasets. One can observe that *BQHist* is more adaptive than *STHoles* with data distribution update, as its line is smoother than *STHoles* in both cases.

6 Related Work

Several selectivity estimation techniques have been proposed in the past, including sampling (Wu, Agrawal & Abadi 2002, Lipton, Naughton & Schneider 1990, Wu, Agrawal & Abadi 2001), histograms (Abounaga & Chaudhuri 1999, Bruno et al. 2001, An, Yang & Sivasubramaniam 2001, Chen & Roussopoulos 1994, Donjerkovic, Ioannidis & Ramakrishnan 2000, Ioannidis & Pooala 1995, Jin, An & Sivasubramaniam 2000) and parametric technique (Konig & Weikum 1999). The tuple sampling technique (Lipton et al. 1990, Wu et al. 2001) summarizes a relation by taking uniform samples from the tuples in the relation. When a query is posed to an estimator, the estimator considers the sample size and sampling result and produces results. Parametric technique, also known as the curve-fitting technique, approximates data distributions using distribution functions with a limited number of parameters. In (Chen & Roussopoulos 1994, Sun, Ling, Rishe & Deng 1993), a general polynomial function and least squares fitting are used to choose its coefficients, while (Konig & Weikum 1999) represents the distribution

as a linear combination of some mathematical functions. The coefficients of the functions are adjusted using feedback information. The main problem with this approach is that usually it is very difficult to find a function to describe arbitrary data distribution.

Histograms can be constructed either statically or dynamically (see (Ioannidis 2003) for a survey). Static histograms have been well studied in literature, such as wavelet based histograms (Matias, Vitter & Wang 1998) and the V -optimal(F, F) (Poosala, Ioannidis, Haas & Shekita 1996) and V -optimal(V, F) (Jagadish, Koudas, Muthukrishnan, Poosala, Sevcik & Suel 1998) family of histograms. After histograms are built using such static approaches, buckets and frequencies remain fixed regardless of any changes in the dataset. The histograms are rebuilt when the error of selectivity estimation reaches some threshold. On the other side, dynamic histograms can capture the changes in the data distribution. This type of histograms works well for close to uniform tuple density by considering query workload information and query execution feedback to progressively refine histogram buckets. Buckets with non-uniform density can be detected and split into smaller and more accurate buckets. Adjacent buckets of similar data distribution can also be detected and merged to recuperate space for more critical regions. *STGrid* histograms use query workloads to refine a grid-based histogram structure (Aboulnaga & Chaudhuri 1999). A problem of this method is that the grid partitioning strategy can be too rigid, as data distributions generally form clusters which leads to many not-so-useful buckets. In (Lim et al. 2003), SASH is proposed to use a two-phase method to automatically build and maintain an optimal set of histograms using query feedback information. A sampling-based approach for incremental maintenance of approximate histograms is reported in (Gibbons, Mattias & Poosala 1997). In (Thaper, Guha, Indyk & Koudas 2002), dynamic histogram on data stream is addressed.

(Bruno et al. 2001) proposed a histogram called *STHoles*. It is a ‘workload aware’ histogram technique allows nested histogram buckets to capture regions with nearly uniform data distribution. To construct an *STHoles* histogram, one can start with an empty histogram or a single bucket histogram that covers the entire domain. The actual result of each query in the workload is intercepted from the query processor and is used to refine the histogram. By computing the overlapping regions of a query with each histogram bucket, one can refine the histogram by ‘drilling holes’ or zooming into the buckets that cover the query region. If the total number of buckets exceeds the fixed storage constraint, one can merge adjacent similar buckets, which results in the smallest penalty on the query estimation accuracy.

7 Conclusions

In this paper, we have proposed two effective methods to make dynamic histograms adaptive to changing workload patterns. We have addressed the problems in *STHoles* which is a representative self-tuning histograms by using an online scheduling algorithm that orders query execution such that histograms built from the feedback can adapt much more rapidly but not oversensitively to the changes of the underlying workload patterns. A selectivity estimation algorithm has also been proposed to improve estimation

accuracy for queries in the areas close to but not covered by high quality buckets, using a technique based on parametric approximation. Both the online scheduling algorithm and selectivity estimation algorithms can be incrementally maintained. Our experiments have demonstrated that our methods can improve consistently the range query selectivity estimation accuracy by nearly 50%, comparing *STHoles* which is a highly effective and practical histogram method. Our future work include investigating the problem of non-aligned window query selectivity estimation, and extending our techniques to join selectivity estimation.

References

- Aboulnaga, A. & Chaudhuri, S. (1999), Self-tuning histograms: Building histograms without looking at data, *in* ‘SIGMOD’.
- An, N., Yang, Z. Y. & Sivasubramaniam, A. (2001), Selectivity estimation for spatial joins, *in* ‘ICDE’.
- Bruno, N., Chaudhuri, S. & Gravano, L. (2001), Stholes: A multidimensional workload-aware histogram, *in* ‘SIGMOD’.
- Chen, C. M. & Roussopoulos, N. (1994), Adaptive selectivity estimation using query feedback, *in* ‘SIGMOD’.
- Donjerkovic, D., Ioannidis, Y. & Ramakrishnan, R. (2000), Dynamic histograms: Capturing evolving data sets, *in* ‘SIGMOD’.
- Gibbons, P., Mattias, Y. & Poosala, V. (1997), Fast incremental maintenance of approximate histograms, *in* ‘VLDB’.
- Ioannidis, Y. E. & Poosala, V. (1995), Balancing histogram optimality and practicality for query result size estimation, *in* ‘SIGMOD’.
- Jagadish, H. V., Koudas, N., Muthukrishnan, S., Poosala, V., Sevcik, K. C. & Suel, T. (1998), Optimal histograms with quality guarantees, *in* ‘VLDB’.
- Jin, J., An, N. & Sivasubramaniam, A. (2000), Analyzing range queries on spatial data, *in* ‘ICDE’.
- Konig, A. C. & Weikum, G. (1999), Combining histograms and parametric curve fitting for feedback-driven query result-size estimation, *in* ‘VLDB’.
- Lim, L., Wang, M. & Vitter, J. S. (2003), Sash: A self-adaptive histogram set for - dynamically changing workloads, *in* ‘VLDB’.
- Lipton, R. J., Naughton, J. F. & Schneider, D. A. (1990), Practical selectivity estimation through adaptive sampling, *in* ‘SIGMOD’.
- Ioannidis, Y. (2003), The history of histograms (abridged), *in* ‘VLDB’.
- Matias, Y., Vitter, J. S. & Wang, M. (1998), Wavelet-based histograms for selectivity estimation, *in* ‘SIGMOD’.
- Poosala, V., Ioannidis, Y. E., Haas, P. J. & Shekita, E. J. (1996), Improved histograms for selectivity estimation of range predicates, *in* ‘SIGMOD’.
- Srivastava, U., Haas, P. J., Markl, V., Megiddo, N., Kutsch, M. & Tran, T. M. (2006), Isomer: Consistent histogram construction using query feedback, *in* ‘ICDE’.
- Sun, W., Ling, Y., Rishé, N. & Deng, Y. (1993), An instant and accurate size estimation method for joins and selections in an retrieval-intensive environment, *in* ‘SIGMOD’.
- Thaper, N., Guha, S., Indyk, P. & Koudas, N. (2002), Dynamic multidimensional histogram, *in* ‘SIGMOD’.
- Wu, Y. L., Agrawal, D. & Abbadi, A. E. (2001), Applying the golden rule of sampling for query estimation, *in* ‘SIGMOD’.
- Wu, Y. L., Agrawal, D. & Abbadi, A. E. (2002), Query estimation by adaptive sampling, *in* ‘ICDE’.