Proactive and Reactive Multi-dimensional Histogram Maintenance for Selectivity Estimation

Zhen He[1], Byung Suk Lee[2], X. Sean Wang[2]
[1]Department of Computer Science, La Trobe University, Bundoora VIC 3086, Australia*
[2]Department of Computer Science, University of Vermont, Burlington VT 05405, USA
Emails: {z.he@latrobe.edu.au, Byung.Lee@uvm.edu, Sean.Wang@uvm.edu}

Abstract

Many state-of-the-art selectivity estimation methods use query feedback to maintain histogram buckets, thereby using the limited memory efficiently. However, they are "reactive" in nature, that is, they update the histogram based on queries that have come to the system in the past for evaluation. In some applications, future occurrences of certain queries may be predicted and a "proactive" approach can bring much needed performance gain, especially when combined with the reactive approach. For these applications, this paper provides a method that builds customized proactive histograms based on query prediction and mergers them into reactive histograms when the predicted future arrives. Thus, the method is called the *Proactive and Reactive Histogram (PRHist)*. Two factors affect the usefulness of the proactive histograms and are dealt with during the merge process: the first is the predictability of queries and the second is the extent of data updates. PRHist adjusts itself to be more reactive or more proactive depending on these two factors. Through extensive experiments using both real and synthetic data and query sets, this paper shows that in most cases, PRHist outperforms STHoles, the state-of-the-art reactive method, even when only a small portion of the queries are predictable and a significant portion of data is updated.

1 Introduction

Extensive literature exists on the selectivity estimation problem. A variety of solutions have been proposed, but most popular ones are histogram-based solutions. Many state-of-the-art histogram methods adopt a "self-tuning" strategy, and construct and maintain histograms based on the feedback information provided in the form of query results [1, 2, 3]. This allows them to keep more histogram buckets in regions queried more frequently. Due to the common spatial and temporal locality of user queries, these methods usually make more efficient use of the limited memory than methods based solely on data distribution.

However, all existing self-tuning histograms are "reactive" in nature, that is, they are updated based on the feedback information of the *past* queries only. This reactive approach does not consider the possibility that certain *future* query occurrences can be predicted. In this paper, by building and using "proactive histograms" based on query prediction, we show that a proactive approach can make more efficient use of histogram memory, when combined with the reactive approach. We believe this paper is the first to introduce the idea of *proactive histograms* and to provide a method to take advantage of these histograms.

Query prediction may be obtained by inspecting the query log and identifying patterns in it, or may be derived directly from the business practice of the enterprise. For example, (1) in a university database, queries on students' course grades occur more frequently at the end of every semester, and (2) in a financial company database, queries on financial records occur more frequently at the end of every financial quarter.

Queries are predictable to a varying degree, either because they follow patterns to a varying degree or the patterns are recognizable to a varying degree. It is thus important to handle queries with different degrees of predictability. To this end, we equip our method with the ability to adjust itself to be more proactive

*This work was partially done while the author was at the Department of Computer Science, University of Vermont.

or reactive depending on query predictability. We call our method the *Proactive and Reactive Histogram (PRHist).*

Within the PRHist method, we need to deal with two main issues: (1) predicting future queries, and (2) using the predicted queries to improve selectivity estimation. In order to address the first issue, we have PRHist find patterns in the query log [1]. Looking for patterns of individual query occurrences is not feasible because those occurrences may rarely repeat themselves (the classical overfitting phenomenon). Therefore, PRHist groups queries into "clusters" and looks for patterns considering all queries in the same cluster as a whole. Section 4.1 provides a formal definition of a cluster of queries and cluster patterns.

In order to address the second issue, we have PRHist operate in two phases: off-line and on-line. In the off-line phase, we divide the future into a sequence of consecutive subintervals of time which we call *f-subintervals*. Then we predict a set of queries for each f-subinterval. Then, it builds a sequence of proactive histograms, one histogram for each of the f-subintervals. In the on-line phase, PRHist maintains an on-line reactive histogram using query feedback as done in the STHoles. At the beginning of each f-subinterval, it loads the proactive histogram built for the f-subinterval and merges it with the on-line reactive histogram. At the time of the merge, PRHist assigns a weight to each histogram based on its "confidence" in the proactive histogram, thereby becoming more reactive (when the confidence is low) or more proactive (otherwise). The confidence is measured as a combination of two factors: the predictability of queries and the extent of data updates.

Our method incurs very small additional run-time overhead over above existing reactive histogram techniques. This small overhead is for loading the off-line proactive histogram into memory and merging it with the on-line reactive histogram at the beginning of each f-subinterval. For example, if a proactive histogram is loaded every 30 minutes as is the case in our experiments with the real query set, this amounts to one or two extra page loadings[2] plus one merge operation for every 30 minutes. The merge operation is fast since the histograms often contain only a small number of buckets.

We have conducted extensive performance comparisons between PRHist and the state-of-the-art reactive histogram method STHoles, using both real and synthetic data sets. The results show that PRHist outperforms STHoles for most test cases even when only a very small portion of the queries are predictable or even when there are a high percentage of data updates.

We make two key contributions through this paper. We (1) develop a novel histogram maintenance method, which uses proactive histograms complemented with reactive histograms to improve selectivity estimation accuracy, and (2) conduct an extensive performance study to show the performance advantage of our method in a variety of situations.

The rest of the paper is organized as follows. Following this introduction, we discuss related work in Section 2, and formally define the problem addressed by PRHist in Section 3. We describe the off-line proactive histogram construction in Section 4 and the on-line PRHist maintenance in Section 5. In Section 6, we present the experiments, and with Section 7, we conclude the paper.

2 Related Work

In this section we discuss three areas of related work: histograms for selectivity estimation, forecasting methods for query prediction, and proactive optimization methods used in database systems.

2.1 Histograms

We classify histograms as shown in Figure 1. Data-driven histograms are built and/or maintained solely based on the distribution of the data [5, 6, 7, 8, 9], and are typically rebuilt or re-organized if the number of data updates exceeds a threshold or the estimation error is above a tolerance value [10, 11, 12, 13]. These histograms have the drawback of assuming that all queries are equally likely at all times, which is rarely true as certain regions may be queried more frequently than others at certain times.

Query-driven histograms are the ones considered in this paper. They are built and/or maintained based on the occurrences of queries. This allows for allocating more memory to regions queried more frequently,

[1]Deriving query patterns from the enterprise business practice is possible but beyond the scope of this paper.

[2]Histograms are typically very small, often fewer than one or two pages in size [1, 2, 4].

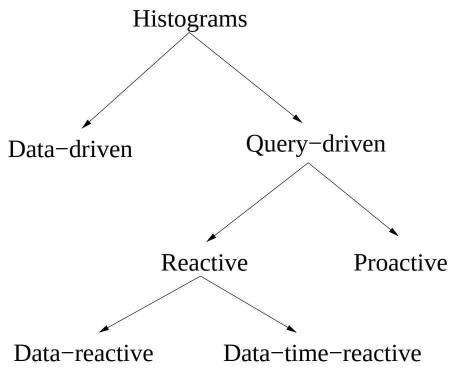


Figure 1: Taxonomy of histogram methods.

thereby using the memory more efficiently. These histograms are either reactive or proactive. Reactive histograms are responding to changes in either data only (called *data-reactive*) or both data and time (called *data-time-reactive*). Data-time-reactive histograms are maintained by taking the temporal locality of queries into consideration so newer query regions are modeled more accurately; in contrast, data-reactive histograms treat all queries equally without regard to their time of execution. There is no existing selectivity estimation method that uses proactive or data-time-reactive histograms.

Data-reactive histograms use query feedback to adapt to changing data distributions. There are two existing ones: STGrid [1] and STHoles [2]. STGrid uses query feedback to merge and split buckets of a grid-based histogram. STHoles improves on STGrid by allowing buckets to be created anywhere in the data space without any grid. It can create a child bucket (or "hole") inside a parent bucket, thus configuring the buckets into a tree structure. This hole approach implicitly allows non-rectangular buckets to be created, thereby modeling complex data distributions. It creates histogram buckets as query feedback arrives, and merges them when the memory is used up. Although both STGrid and STHoles use query feedback to maintain the histogram reactively, they do not age out buckets as they become old and not used anymore and, therefore, are not data-time-reactive.

To our knowledge, the only existing work that supports data-time-reactivity is LEO [3]. However, LEO is not a histogram-based technique per se, nor used specifically for selectivity estimation. It rather offers a comprehensive way of repairing incorrect statistics and cardinality estimates of a query execution plan by using feedback information from recent query executions.

We believe PRHist is the only query-driven histogram method that uses all three types of histograms under the query-drive category for selectivity estimation.

2.2 Forecasting methods

PRHist takes advantage of the forecasting ability of time series forecasting methods to predict future queries based on query clusters. There are a number of existing time series forecasting methods, including general exponential smoothing [14], Holt's linear trends model [15], Holt-Winters seasonal model (HWSM) [16, 17], parametric regression [18, 19], and Box-Jenkins [20]. These methods are designed to find patterns such as seasonal changes and trends.

In order to be usable by PRHist, however, they must be modified by incorporating the concept of query clustering. Our query prediction method clusters queries before looking for patterns using a forecasting method. We have chosen to use HWSM since empirical results suggest that simple methods like HWSM are robust to complex trends and seasonal effects and also achieve accuracy comparable to that of complex methods like Box Jerkins [21]. Moreover, HWSM is easy to automate and fast, hence ideal for finding a large number of patterns [21]. We will describe HWSM further in Section 4.2.2.

2.3 Proactive Optimization

We present three proactive optimization techniques: prefetching, speculative query processing and proactive query reoptimization. Prefetching is typically used in cache management [22, 23, 24, 25, 26]. It predicts items (e.g., disk page, web page, cache line) requested immediately next and preloads them into cache, thereby overlapping the fetch latency of IO, network, etc. with CPU time. Most existing prefetching methods use low order Markov Chains [22, 25], N-gram sequences [26], or structural information [23, 24] to make predictions. The prediction methods designed for prefetching are not suitable for our purpose of predicting items (i.e., queries) for the near future that is more distant from the immediate next. Predicting the immediate next queries may not be useful since it needs impractical *on-line* generation of proactive histograms.

In speculative query processing, proposed by Polyzotis et. al.[27], the system monitors querying behaviors of the user and builds a user-behavior model. Then, using the model, the system expects certain features of partial queries the user is likely to ask soon and prepares the query processor by "issuing asynchronous manipulations that are likely to make the final query more efficient."[27] Their work has a different focus from ours. That is, they are focused on the entire query optimization process whereas we are focused on selectivity estimation only. Another difference is that we combine both reactive and proactive methods whereas they consider only reactive methods.

Babu et al.[28] propose a proactive approach to reducing the overheads of query reoptimization in conventional approaches. The key idea is to generate a set of robust and switchable execution plans of a query in a way to reduce the need for reoptimizing the query and the loss of pipelined execution. Specifically, plans are made robust, thus reducing the need for reopitmization, by allowing for some uncertainties in the cardinality estimations. Besides, a plan in a set of switchable plans is selected efficiently to allow the reuse of pipelined work. Their work has a different objective from ours. That is, their objective is efficient query re-optimization whereas our objective is accurate selectivity estimation.

3 Problem Formulation

In this section, we formally define the selectivity estimation problem addressed by PRHist.

3.1 Preliminaries

We define a data set D as a set of tuples. A query q is defined by the minimum and maximum bounds on the value of each queried attribute, i.e., $q \equiv \langle (v1_{min}, v1_{max}), (v2_{min}, v2_{max}), \cdots, (v_{d_{min}}, v_{d_{max}}) \rangle$ where each $(v_{i_{min}}, v_{i_{max}})$ denotes the bounds. A query log S_{log} is defined as a sequence of timestamped queries, i.e., $S_{log} \equiv \langle q_{t_1}, q_{t_2}, \cdots, q_{t_s} \rangle$ where $t_1, t_2, \cdots, t_s$ are timestamps.

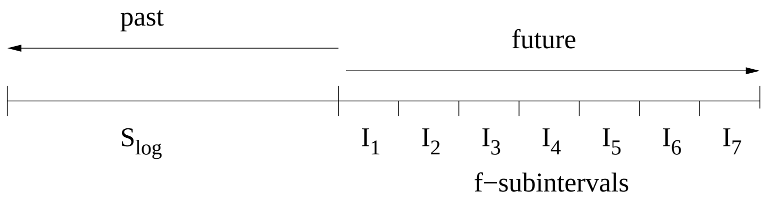


Figure 2: A query log (S_{log}) and f-subintervals ($\langle I_1, I_2, \cdots, I_h \rangle$).

We assume our whole time line includes the time covered by the S_{log} and the future time for which we want to make our predictions (see Figure 2). Then, we divide this whole time line into consecutive intervals, called *baseline intervals*, each of which is of the same length. We call a particular consecutive sequence of future baseline intervals *f-subintervals*, denoted $I \equiv \langle I_1, I_2, \cdots, I_h \rangle$. We want to predict the queries that will appear in these f-subintervals based on the queries in the query log.

3.2 Problem definition

PRHist addresses the problem of selectivity estimation in two phases: an off-line phase and an on-line phase. In the off-line phase, given a query log (S_{log}), a sequence of f-subintervals $I \equiv \langle I_1, I_2, \cdots, I_h \rangle$, a memory limit ($M$), and a data set ($D$), PRHist outputs a sequence of proactive histograms ($PHs \equiv \langle PH_1, PH_2, \cdots, PH_h \rangle$) customized to minimize the total selectivity estimation error for a sequence of actual query sets in I. Each PH_i must fit in the memory limit M. The main issues in the off-line phase are predicting the set of queries that will appear in each f-subinterval and determining how to build the optimal histogram for each of the predicted query sets.

In the on-line phase, PRHist merges each PH_i into an on-line reactive histogram (RH) at the beginning of I_i. Within each f-subinterval I_i, it uses RH to minimize the selectivity estimation error for queries arriving incrementally during I_i. The main issue in the on-line phase is how to best merge the proactive and reactive histograms in consideration for such query characteristics as the predictability and the locality.

4 Off-line Proactive Histogram Sequence Construction

Figure 3 shows the high-level algorithm used for off-line proactive histogram sequence construction. First, the query log is used to find a set of "useful" clusters (Line 1). These clusters are then used for predicting queries, which in turn are used to construct a sequence of proactive histograms (Line 2). Then, optimal weights are assigned to the buckets of the proactive and reactive histograms using an optimization algorithm (Line 3). (The weights are used during the on-line phase for histogram merging, to be described in Section 5.)

```
Construct_PHs($S_{log}$, $\langle I_1, I_2, \cdots, I_h \rangle$, $M$, $D$)
1. $C_{useful}$ := Find_useful_clusters($S_{log}$)  (§ 4.2).
2. Build_PHs($C_{useful}$, $S_{log}$, $\langle I_1, I_2, \cdots, I_h \rangle$, $M$, $D$) (§ 4.3).
3. $\mathbf{w}_{opt}$ := Optimal_bucket_weights($C_{useful}$, $S_{log}$, $M$, $D$) (§ 4.4).
```

Figure 3: High-level algorithm for proactive histogram sequence construction.

4.1 Basic concepts of query clusters

Our insight is that patterns can often be found from a group of queries rather than from individual queries. For example, consider queries executed by a particular real estate agent on a database of houses listed for sale. The agent may use the database only on Monday, Wednesday, and Friday afternoons, and each query for houses may use only a particular price range – \$350K $\sim$ \$400K for instance. There may not be any pattern identifiable for each particular query unless the query is repeated. However, if all the queries executed by the agent are examined together, a pattern may be identified. For example, we may find that queries for houses in the price range of \$300K $\sim$ \$600K occur on Monday, Wednesday, and Friday afternoons.

A group of queries form a *cluster*. A query cluster is defined as a group of queries that are close to one another according to a certain distance metric. The distance metric used in this paper is *cluster utility* (to be defined in Equation 10). A cluster may show a *cluster pattern* which is defined as a predictable sequence of the frequencies of query cluster appearances. The frequencies are modeled using the *discrete time-frequency distribution (DTFD)* (see Section 4.2.2). Note that a cluster pattern is a tool for predicting the future occurrences of queries in a cluster, and is not in itself a representation of a cluster. The prediction task is to predict the frequency at a cluster level for each subinterval.

Not all clusters are equally useful. A useful cluster should meet the following two criteria. First, its cluster pattern should be reliable for predicting the frequency of its queries for f-subintervals. Otherwise, inaccurate predictions may cause histogram buckets to be created in regions that are not queried and, thus, waste scarce memory resources. Second, its queries should cover a small region of the multi-dimensional space. Otherwise, the cluster would not be useful for predicting which region is likely to be queried. We refer to the first criterion as the *predictability* and the second criterion as the *spatial locality*. These two often contradict with each other, as a larger area often leads to higher predictability. How to establish a balance is the key. (Further details of these criteria are discussed in Section 4.2.3.)

4.2 Finding useful clusters

Finding the theoretically most useful clusters is very costly because calculating the usefulness of a cluster involves checking the two criteria mentioned above – predictability and spatial locality – while checking each criterion is an expensive operation (as will be shown in Section 4.2.3 and 4.2.3). We thus use a greedy algorithm based on the heuristics of finding one cluster at a time, from the current most useful one first (described below).

Figure 4 outlines the algorithm *Find_useful_clusters*. (The numbers prefixed with '§' refer to the sections where the topics are discussed in detail.) We first give a high-level overview of the algorithm here, and then elaborate on each step of the algorithm in the remainder of this subsection.

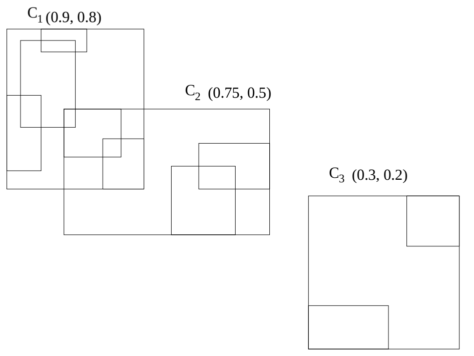
```
Find_useful_clusters($S_{log}$)
1. Cluster queries in $S_{log}$ into a group $g$ of overlapping clusters (§ 4.2.1).
2. Partition $S_{log}$ into $S_{train}$ and $S_{test}$.
3. repeat begin
4.    for each cluster $c$ in $g$ begin
5.        Extract cluster pattern $p$ from $c$ using $S_{train}$ (§ 4.2.2).
6.        Using $p$, predict the frequency of queries in $c$ during the time span
          of $S_{test}$ (§ 4.2.2).
7.        Estimate the predictability of $p$ for $c$ using $S_{test}$ (§ 4.2.3).
8.        Estimate the spatial locality of $c$ (§ 4.2.3).
9.        Compute the utility value of $c$ (§ 4.2.3).
10.   end for.
11.   Find the cluster $c_{hi}$ with the highest utility value among the clusters
       both of whose predictability and spatial locality are above their respective
      thresholds.
12.   if $c_{hi}$ is found then begin
13.       Save $c_{hi}$ into a set of useful clusters ($C_{useful}$) and remove it from $g$.
14.       Remove the queries in $c_{hi}$ from other clusters containing them.
15.       Remove empty clusters from $g$.
16.   end if
17. end repeat until $g$ has no cluster both of whose predictability and spatial
    locality are above their respective thresholds.
18. return $C_{useful}$.
```

Figure 4: Algorithm for finding useful clusters.

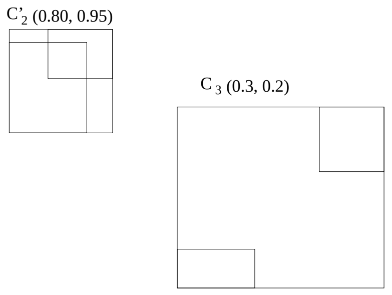
The algorithm first generates a set of possibly overlapping clusters (Line 1) and partitions the timestamped query sequence S_{log} into a training sequence and a testing sequence (Line 2). Then, it repeatedly searches through the clusters generated in Step 1, taking one cluster out each time (Lines 3 $\sim$ 17). Specifically, it first extracts a cluster pattern from each cluster (Line 5), and then predicts the frequency of queries in the cluster during the time interval covered by the testing query sequence (Line 6). Then, based on the heuristics, it chooses the cluster that gives the highest utility value (a measure of usefulness, defined in Section 4.2.3) among those whose patterns' prediction accuracies (Line 7) and spatial localities (Line 8) are above their respective thresholds (Line 11), and then saves the chosen cluster in the set of useful clusters (Line 13).

To avoid double counting the queries, it removes the queries in the cluster from the remaining clusters (Line 14). This removal also has the advantage of making cluster patterns previously less predictable possibly more predictable, since queries showing conflicting patterns may be removed. In addition, previously less spatially local clusters may become more spatially local, since removing queries reduces the volume of the region covered by the remaining queries. If any cluster is made empty through this process, then the cluster is removed (Line 15). The search continues until no clusters remain or all the remaining clusters have either the predictability or spatial locality or both below their respective thresholds.

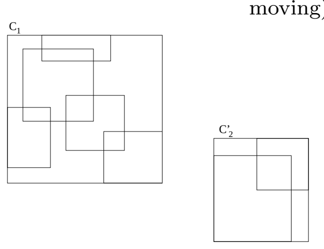
Figure 5 illustrates the algorithm *Find_useful_clusters*. Suppose there are three overlapping clusters, c_1, c_2, and c_3, in the group generated from queries in S_{log} (see Figure 5a). Then, the cluster c_1 is inserted into C_{useful} as its utility value 0.72 ($= 0.9 \times 0.8$; see Equation 10) is the highest among the three clusters. After removing c_1, the cluster c_2 becomes c'_2 as a result of removing the queries that belong to c_1. Figure 5(b) shows the remaining unselected clusters. Its new predictability, 0.80, and new spatial locality, 0.95, are both above the thresholds and the new utility value 0.76 ($= 0.80 \times 0.95$) is the highest among the remaining (two) clusters. Thus c'_2 is inserted into C_{useful}. There is no change to the predictability and spatial locality of c_3 and both are lower than their respective thresholds. Hence, c_3 is discarded. As a result, the algorithm returns $\{c_1, c'_2\}$ as C_{useful}, shown in Figure 5(c).



(a) Clusters generated from S_{log}.



(b) Clusters after selecting (and removing) c_1.



(c) Clusters in the resulting C_{useful}.

(c_i(pred, loc): pred $\equiv$ predictability of c_i; loc $\equiv$ spatial locality of c_i)
(predictability threshold = 0.7; spatial locality threshold = 0.7)

Figure 5: An illustration of the algorithm *Find_useful_clusters*.

4.2.1 Query clustering strategies

There is no restriction on how the query clusters can be generated. Ideally, the clusters should be generated by considering only a certain utility metric (like the one to be shown in Equation 10). If unlimited computational power were available, we could generate all possible overlapping subsets of the queries in S_{log}. Evidently, this is computationally expensive, and we thus opt for application specific clustering strategies to reduce the search space.

In our experiments in Section 6, we will assume the following three simple clustering strategies: clustering by user, clustering by region, and a combination of the two, namely, clustering by user-region. (The clustering by user has been mentioned as an example in Section 4.1.) More specifics of these strategies will appear in Section 6.1.1.

4.2.2 Cluster pattern extraction and query frequency prediction

After the query clusters are found, a cluster pattern is extracted from each cluster. We use the *discrete time-frequency distribution (DTFD)* for modeling the cluster pattern. To build the DTFD for a given cluster

(c), PRHist first discretizes the time span covered by S_{log} into a sequence of time slots $\Delta_1, \Delta_2, \cdots, \Delta_n$ of equal length such that (1) each Δ_i interval is entirely contained within a baseline interval, and (2) the union of all these Δ_i ($i = 1, \ldots, n$) intervals is equal to the union of a consecutive sequence of baseline intervals.

Then, it counts the frequency ($f_{\Delta_i}(c)$) of the queries in the cluster c within each $\Delta_i, i = 1, 2, \cdots, n$ (see Figure 6). The resulting DTFD can be considered a time series of query frequency, and thus opens a door to using a time series forecasting algorithm.

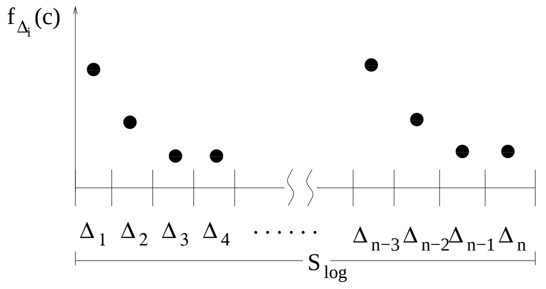


Figure 6: Discretizing the time span of S_{log} to build DTFD for cluster c.

As mentioned in Section 2.2, we use the *Holt-Winters seasonal model (HWSM)* [16, 17] because it is particularly suitable for our needs. HWSM uses exponential smoothing techniques and recurrence equations to model both seasonal and trend variations. Specifically, HWSM uses three recurrence equations, capturing the local level, growth, and season aspects of the time series. Each equation has one smoothing parameter. The equations are intended to give higher weights to recent observations and lower weights to observations further in the past. These weights are geometrically decreasing by a constant ratio.

There are two types of HWSM models: additive and multiplicative. The additive model is optimal for a seasonal auto-regressive integrated moving average (ARIMA) model of a complicated form [21]. The multiplicative model is near optimal for a class of non-linear state and space models. We use the additive model in our work, as it provides more accurate predictions than the multiplicative model in our experiments. The additive model is expressed as three recurrence equations involving what are called the level, L_i, trend, T_i, and seasonal index, I_i, of the time series:

$$L_i = \alpha(x_i - I_{i-s}) + (1-\alpha)(L_{i-1} + T_{i-1}) \quad (1)$$

$$T_i = \gamma(L_i - L_{i-1}) + (1-\gamma)T_{i-1} \quad (2)$$

$$I_i = \delta(x_i - L_i) + (1-\delta)I_{i-s} \quad (3)$$

where s is the order of seasonality and x_i is the i-th element of the training time series. Here, $s < i \leq n$ where n is the size (number of elements) of the training time series. The parameters α, γ, and δ are the smoothing parameters, whose values are determined as a result of the training. The starting values of L_i, T_i, and I_i (L_s, T_s, $\langle I_1, I_2, \cdots, I_s \rangle$) are computed using the following equations.

$$L_s = \frac{1}{s}(x_1 + x_2 + \cdots + x_s) \quad (4)$$

$$T_s = \frac{1}{s}\left(\frac{x_{s+1} - x_1}{s} + \frac{x_{s+2} - x_2}{s} + \cdots + \frac{x_{s+s} - x_s}{s}\right) \quad (5)$$

$$I_1 = x_1 - L_s,\ I_2 = x_2 - L_s,\ \cdots,\ I_s = x_s - L_s \quad (6)$$

The forecast of the j-th time element ($j = 1, 2, \cdots, h$) from the time of the n-th element is computed using the following equation.

$$\hat{x}_n(j) = L_n + jT_n + I_{n-s+j} \quad (7)$$

When applied to PRHist, x_i is $f_{\Delta_i}(c)$ (i.e., the frequency of queries in the cluster c within Δ_i), s is the number of time slots ($\Delta_1, \Delta_2, \cdots, \Delta_s$) in a period (e.g., $s = 24$ if the time series repeats every day and each Δ_i interval represents one hour), and $\hat{x}_n(j)$ is the query frequency within the j-th f-subinterval from the last time slot (Δ_n) of the query log.

In order to tune the parameters while training the model shown above (Equations 1, 2, and 3), PRHist uses the simulated annealing [29] heuristic search algorithm. For each enumerated value of the parameters, PRHist computes the forecasting (i.e., prediction) accuracy using the same metric as that used for computing the predictability of a cluster (see Equation 8 in Section 4.2.3 below).

Due to space constraint we do not elaborate further on HWSM but instead refer the reader to the articles by Chatfield and Yar [16] and by Ord, Koehler, and Snyder [17] for more detailed descriptions.

4.2.3 Predictability, spatial locality, and cluster utility

Predictability In order to know how effective a cluster pattern will be for accurately predicting query frequencies, PRHist uses a metric commonly applied to evaluating time series forecasting accuracy [21]. Specifically, it first partitions the baseline intervals that are covered by the queries in S_{log} into two parts based on one time point: all the baseline intervals before the time point are called S_{train} and after the time point are called S_{test}. The baseline intervals in S_{test} are called *test subintervals*: $I_{test_1}, I_{test_2}, \cdots, I_{test_h}$ (see Figure 7).

Then, given a cluster c generated from the training sequence, it extracts a cluster pattern $p(c)$ and uses it to predict the query frequency for each subinterval $I_{test_i}, i = 1, 2, \cdots, h$. It then uses the following equation to compute the predictability of the pattern for a cluster c, $\mathcal{P}(c)$.

$$\mathcal{P}(c) = 1 - \frac{\sum_{i=1}^{h} |f_{I_{test_i}}(c) - \hat{f}_{I_{test_i}}(p(c))|}{\sum_{i=1}^{h} max(f_{I_{test_i}}(c), \hat{f}_{I_{test_i}}(p(c)))} \tag{8}$$

where $f_{I_{test_i}}(c)$ is the actual frequency of queries in the cluster c occurring in I_{test_i} and $\hat{f}_{I_{test_i}}(p(c))$ is the corresponding predicted frequency obtained using the cluster pattern $p(c)$.

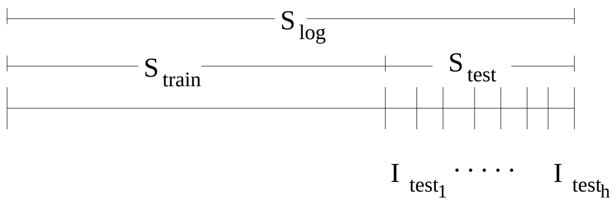


Figure 7: Partitioned S_{log} for predictability computation.

Spatial locality PRHist uses Equation 9 to compute the spatial locality, $\mathcal{L}(c, Q)$, for a cluster c generated from the set of queries Q in the query log.

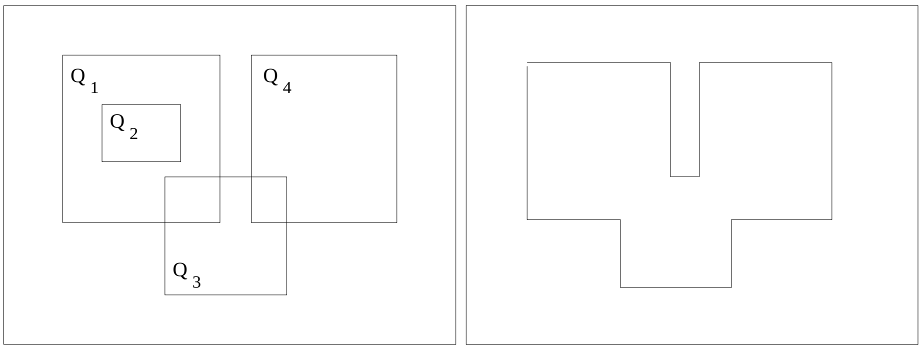
$$\mathcal{L}(c, Q) = 1 - \frac{vol(\bigcup_{q_j \in c} q_j)}{vol(\bigcup_{q_i \in Q} q_i)} \tag{9}$$

where $vol(\bigcup_{q_i \in Q} q_i)$ is the volume of the region covered by the union of all queries in Q and $vol(\bigcup_{q_j \in c} q_j)$ is the volume of the region covered by the union of all queries in c. Figure 8(b) illustrates the region covered by the union of all queries within a cluster shown in Figure 8(a).

Cluster utility PRHist computes the utility of a cluster c, $\mathcal{U}(c)$, as a product of the two key metrics – predictability and spatial locality.

$$\mathcal{U}(c) = \mathcal{P}(c) \times \mathcal{L}(c) \tag{10}$$

As mentioned in Section 4.1, there is often a trade-off between the two metrics. For example, a cluster with a single small query gives a very high spatial locality value, but the query may not occur regularly enough to be predictable.



(a) Queries in a cluster. (b) Union of queries in the cluster.

Figure 8: Region covering the queries in a cluster.

4.3 Constructing a proactive histogram sequence

Figure 9 outlines the algorithm *Build_PHs* which constructs a sequence of proactive histogram for a given sequence of f-subintervals. The algorithm has two steps. In the first step (Lines 1∼4), using the query log S_{log}, it extracts a cluster pattern from each useful cluster (in C_{useful}) provided by the algorithm *Find_useful_clusters* (Line 2), and then predicts the query frequency of the cluster for each f-subinterval using the cluster pattern (Line 3). In the second step (Lines 5∼11), for each f-subinterval, it combines the predicted queries of each useful cluster and inserts them into a set of queries if the query frequency of the cluster is above the minimum frequency threshold (Lines 7 ∼ 9), and then, for the queries thus collected, it creates histogram buckets within the memory limit M, using the data set D to compute the statistics (Line 10).

```
Build_PHs($C_{useful}$, $S_{log}$, $\langle I_1, I_2, \cdots, I_h \rangle$, $M$, $D$)
   // Predict the query frequency for each f-subinterval.
1. for each cluster $c$ in $C_{useful}$ begin
2.    Extract a cluster pattern $p$ for $c$ using $S_{log}$ (§ 4.2.2).
3.    Using $p$, predict the query frequency $f$ of $c$ for each f-subinterval $I_i$ (§ 4.2.2).
4. end for.
   // Build a proactive histogram for each f-subinterval.
5. for each $I_i (i = 1, 2, \cdots, h)$ begin
6.     combined query set $Q := \{ \}$.
       // Combine queries.
7.     for each cluster $c$ in $C_{useful}$ begin
8.        if the query frequency of $c$ is above the minimum frequency threshold
          then insert the queries in $c$ into $Q$.
9.     end for.
       // Construct a proactive histogram.
10.    Create histogram buckets for the queries in $Q$ within the memory limit $M$,
       using $D$ to compute the statistics.
11. end for.
```

Figure 9: Algorithm for constructing a proactive histogram sequence.

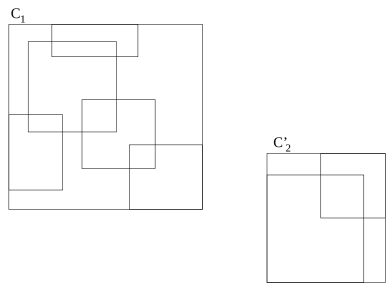
Note that the clusters whose query frequencies are below the minimum frequency threshold are discarded. The reason is that including clusters with low query frequency decreases the spatial locality of the combined queries, while there is only a relatively small corresponding increase in estimation accuracy. For example, if a cluster has ten queries but the queries are predicted to occur only twice during an f-subinterval (that is, query frequency is two), then including this cluster of queries to build proactive histogram may give rise to ten histogram buckets, while only two of the buckets may be used by future queries.

The time complexity of the query frequency prediction (Lines 1∼4 in Figure 9) is $O(n_c(n + p_n + h))$, where n_c is the number of useful clusters, n is the number of Δ time slots in S_{log}, p_n is the time used to build the HWSM model for a cluster with n input values, and h is the number of f-subintervals. A detailed analysis, for each useful cluster, is as follows. First, the starting values L_s, T_s, $\langle I_1, I_2, \cdots, I_s\rangle$ are computed with the complexity $O(s)$ using the first s elements of the time series (see Equations 4, 5, and 6). Then, the next $n - s$ values are computed with the complexity of $O(n - s)$ ($O(1)$ each; see Equations 1, 2, and 3). We leave p_n unspecified as it depends on the specific method used to build the HWSM model, and with the simulated annealing we used in the experiments, p_n is rather small but varies a lot for different clusters. Then, forecasting the h f-subintervals is computed with the complexity $O(h)$ ($O(1)$ each; see Equation 7). Thus, the total is $O(n_c(s + (n - s) + p_n + h)) = O(n_c(n + p_n + h))$.

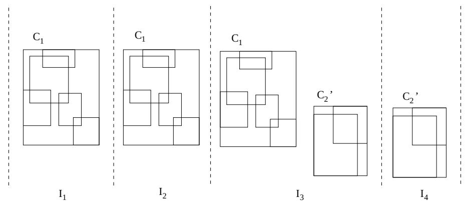
The running time complexity of proactive histogram construction (Lines 5∼11 in Figure 9) is $O(h(n_c + n_q b_M^2))$, where n_q is the number of queries in Q and b_M is the maximum number of buckets that can be contained in a histogram occupying the memory of size M. The term $n_q b_M^2$ is for the time needed to construct a histogram (of b_M buckets) with the memory limit M given n_q queries. This term is derived from the STHoles histogram construction algorithm; the time complexity of this algorithm is dominated by the time complexity of bucket merging which involves comparing the cost-benefit ratio for each pair of the b_M buckets, thus taking time proportional to b_M^2, and this bucket merging needs to be done for each query inserted, thus the merging time multiplied by n_q. (More details can be found from the article by Bruno et al. [2].)

The proactive histogram construction can be done straightforwardly by inserting predicted queries into the empty reactive histogram. In our work, we have used STHoles [2], but any other reactive histogram (like STGrid [1]) may be used as well.

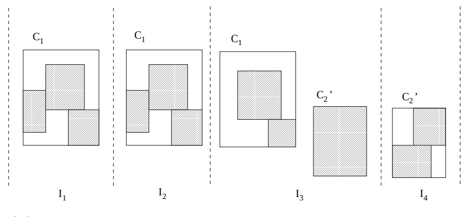
Figure 10 illustrates the algorithm *Build_PHs* given the C_{useful} shown in Figure 10(a) (continuing from the example in Figure 5). Suppose the following query frequency is predicted for each of c_i and c_2' for a sequence of four f-subintervals $\langle I_1, I_2, I_3, I_4\rangle$: $\langle 4, 3, 2, 1\rangle$ for c_1 and $\langle 0, 1, 2, 2\rangle$ for c_2'. Additionally, suppose the minimum frequency threshold is 2. Then, the combined query set, Q, includes queries in the following clusters for each f-subinterval: $\{c_1\}$ for I_1, $\{c_1\}$ for I_2, $\{c_1, c_2'\}$ for I_3, and $\{c_2'\}$ for I_4 (see Figure 10(b)). Suppose the memory is limited to store only four buckets. Then, only four buckets remain for each f-subinterval as a result of inserting all the buckets in Figure 10(b). Figure 10(c) shows one possible result. The figure assumes STHoles, for which the root (i.e., outermost box) is considered a bucket as well. (The particulars of the resulting buckets depend on the reactive histogram technique used to insert the buckets.)



(a) Clusters in C_{useful}.



(b) Predicted queries in each f-subinterval.



(c) Buckets created within the memory limit M (=4 buckets).

Figure 10: An illustration of the algorithm *Build_PHs*.

4.4 Finding optimal bucket weights

As mentioned in the Introduction, PRHist automatically adjusts itself to be more proactive or reactive depending on the predictability of queries and the extent of data updates. This is done by assigning different weights to the buckets of the reactive and proactive histograms and using the weights when merging the two types of histograms during the on-line phase. The weights thus indirectly affect the estimated selectivity through its (direct) effect on the histogram construction.

Bucket weights are initially computed off-line based on only the predictability of queries. Then, they are adjusted during the on-line phase based on the extent of data updates. In this subsection, we discuss how the initial weights are computed off-line. The on-line adjustment will be discussed in Section 5.1.

PRHist supports buckets of all three histogram types classified in Section 2.1. We categorize them into proactive, data-reactive, and data-time-reactive buckets following their histogram type names. That is, buckets of a proactive histogram are labeled as proactive buckets; buckets created by incorporating dynamically arriving user queries are labeled as reactive buckets; a reactive bucket is labeled as data-reactive by default, and labeled as data-time-reactive if "touched" (i.e., used or created) during the previous f-subinterval.

We use a simple weighting scheme that assigns the same weight to all buckets of the same type. Intuitively, higher weights should be assigned to proactive buckets if the predictability is higher, to data-*time*-reactive buckets if temporal locality is higher (so more recently touched buckets are preferred to those giving higher selectivity estimation accuracies), and to data-reactive buckets otherwise.

Figure 11 outlines an algorithm for finding the optimal weight combination $\mathbf{w}_{opt}$ for the three bucket types. Since this is a costly operation, PRHist performs it off-line. It first partitions S_{log} into a training sequence S_{train} and a testing sequence S_{test} in the same manner as in the algorithm Find_useful_clusters (Figure 4) (Line 1). Then, it further partitions S_{test} into a sequence of subintervals (Line 2). It then constructs a proactive histogram sequence (using the algorithm *Build_PHs* in Figure 9) for the sequence of subintervals spanning S_{test} while using S_{train}, instead of S_{log}, as the query log (Line 3). A heuristic search is then performed to find the optimal weight combination (Lines 4 $\sim$ 9). In our experiments we have used the simulated annealing heuristic search technique. It is appropriate for our purpose because it can avoid local optima, which may not be the global optimum, without an exhaustive search. But other techniques may be used as well. The optimality criterion is to minimize the sum of squared errors of selectivity estimation obtained by executing queries in S_{test} reactively in the on-line PRHist maintenance mode (to be described in Section 5) (Line 7).

```
Optimal_off-line_bucket_weights($C_{useful}$, $S_{log}$, $M$, $D$)
1. Partition $S_{log}$ into $S_{train}$ and $S_{test}$.
2. Partition the time interval spanning $S_{test}$ into a sequence of subintervals
   $I_{test_1}, I_{test_2}, \cdots, I_{test_n}$.
3. $PHs$ := Build_PHs($C_{useful}$, $S_{train}$, $\langle I_{test_1}, I_{test_2}, \cdots, I_{test_n} \rangle$, $M$,$D$).
4. For each weight combination $\mathbf{w}$ enumerated by an
   optimization algorithm begin
5.    Initialize PRHist as the empty histogram.
6.    Update PRHist reactively using queries in $S_{train}$.
7.    Compute the $SSE$ of selectivity estimation while updating PRHist reactively
      using queries in $S_{test}$ and merging each $PH_i (i = 1, 2, \cdots, n) \in PHs$ into
      PRHist at the beginning of each subinterval $I_{test_i}$ $(i = 1, 2, \cdots, n)$ using $\mathbf{w}$
      (§ 5.2).
8.   if $SSE < min_{SSE}$ then $\mathbf{w}_{opt} := \mathbf{w}$.
9. end for.
10. return $\mathbf{w}_{opt}$.
```

Figure 11: Algorithm for finding optimal off-line weights of buckets.

5 On-line PRHist Maintenance

Figure 12 gives a high-level overview of the on-line maintenance of PRHist. A proactive histogram built off-line is loaded and merged with an on-line reactive histogram at the beginning of every f-subinterval. At this point, the bucket weights are adjusted depending on the percentage of data updates. Then, during the f-subinterval, the histogram is updated reactively with the feedback from incrementally occurring queries.

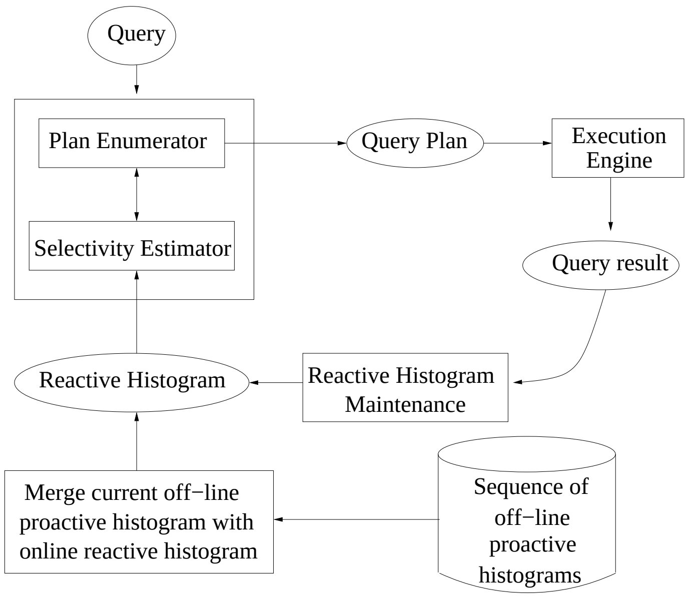

Query
Plan Enumerator
Selectivity Estimator
Query Plan
Execution Engine
Query result
Reactive Histogram
Reactive Histogram Maintenance
Merge current off−line proactive histogram with online reactive histogram
Sequence of off−line proactive histograms

Figure 12: On-Line PRHist maintenance.

5.1 Adjusting the bucket weights

In case the data are updated significantly between the off-line construction of proactive histograms and their merge with the on-line reactive histogram, the tuple density of the proactive histogram buckets may no longer be accurate. We have identified two alternative resolutions to this potential problem.

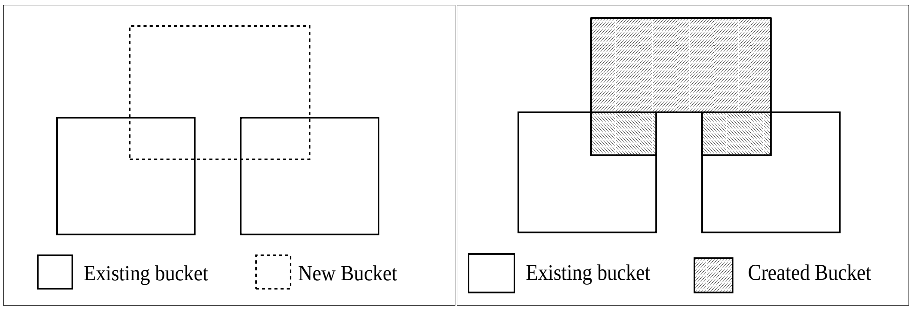
The first approach, used by PRHist, is to adjust the weights of the proactive and reactive buckets at the time of the merge based on the percentage of recorded updates. Specifically, the weights are scaled using a simple heuristic which reflects that reactive buckets are adaptive to the changes whereas proactive buckets (built off-line) are not. That is, the initial weights (determined off-line using the algorithm *Optimal_off-line_bucket_weights* in Figure 11) are scaled by (1 + data update percentage / 100), and proactive buckets are scaled by (1 - data update percentage / 100). For example, if zero percent of data have been updated, then the bucket weights are unchanged, whereas if 100 percent of the data have been updated, the proactive buckets are given the weight of zero. This is the approach used in our experiments.

The second approach is to update the number of tuples associated with the proactive buckets to reflect the changes in data distribution at the time of the merge. A possible way of implementing this approach is to extend existing methods designed for rebuilding single dimensional data-driven histograms [11, 12] to work with multidimensional proactive histograms. We leave this approach as future work.

5.2 Merging proactive and reactive histograms

In merging proactive and reactive histograms, the goal is to produce a set of buckets that have high weights and that show highly contrasting tuple densities between the inside and the outside. Buckets with higher weights are more likely to be used to answer queries. Buckets with higher contrast in tuple densities make more difference in the selectivity estimation accuracy for queries overlapping the bucket significantly.

We handle the histogram merge in two steps: bucket insertions and bucket merging. First, buckets from the two histograms are inserted into an empty histogram in the descending order of their weight (possibly using some extra memory space temporarily). The reason for this ordered insertion is that a bucket inserted later is more likely to be fragmented into small buckets, as illustrated in Figure 13 using STHoles as an example. The figure shows that holes (i.e., buckets) are "drilled" in existing buckets when a new bucket overlapping the existing buckets is inserted. (Readers are referred to [2] for more details.) The fragmentation causes some of the original volume to be lost. Fragmented buckets created as a result of a bucket insertion are assigned the same weight as the bucket inserted.



(a) Inserting a new bucket. (b) Buckets created as a result of insertion.

Figure 13: Fragmentation of a bucket after insertion.

Second, the inserted buckets are merged so the resulting buckets fit within the memory limit. We take advantage of STHoles's merge algorithms except that we scale the merge penalty by the weight of the bucket that is merged out. In STHoles, the merge penalty is computed as the difference in the estimation accuracies between the old histogram, in which both buckets are separate, and the new histogram in which the two or more buckets are merged into one. In our case, for a parent-child merge we scale the merge penalty by the weight of the child bucket, which is the one removed, and for a sibling-sibling merge, by the sum of the weights of the sibling buckets that are merged.

6 Experiments

We evaluate PRHist by comparing it with STHoles using both real and synthetic query/data sets. In this section, we first describe the experimental setup, and then present the results from five experiments.

6.1 Experimental setup

6.1.1 Query clustering strategies

We use the three clustering strategies mentioned in Section 4.2.1: by user, by query region, and by user-region. We use all three strategies together to create a large pool of overlapping clusters. Clustering by user is done by grouping all queries by the same user together and is suitable for finding patterns in which the level of activity for a user is predictable. Clustering by query region uses the K means algorithm [30] to group all queries based on spatial distance between their centers. This clustering strategy is useful for finding clusters of high spatial locality.

Clustering by user-region is done by first clustering some queries by the same user together (e.g., all queries of the same user in one Δ_i interval) and then uses the K means clustering algorithm to further cluster based on the spatial distance between the centers of the queries in the same user-id group. An example pattern that can be found using this clustering strategy is that a user (e.g., a real estate agent) issues queries for a particular region of the data space (e.g., price range of \$300K $\sim$ \$600K) every Monday and a different region (e.g., price range of \$150K $\sim$ \$250K) every Tuesday.

6.1.2 Evaluation method

For performance evaluation, we use the tail portion of a query log, covering one day in length, as the *evaluation query sequence* (i.e., used for performance evaluation) and all the queries before that tail portion in the query log as the *training query log* (i.e., S_{log} defined in Section 3). Then, for PRHist, we use the time interval covered by the test query sequence as the "future time interval"(I), and use the query sequence in the training query log to generate the sequence of proactive histograms for the future interval (without looking at the evaluation query sequence). We initialize STHoles by starting with an empty histogram and then running the STHoles query insertion algorithm on the sequence of queries in the training query log. We initialize PRHist with a proactive histogram built for the first f-subinterval.

6.1.3 Real query set and data set

We used the University of Vermont student administration database as the source of queries and data. This database is used by various departments of the university for managing information about student course registrations, student course grades, etc. We used a query log spanning 19 days. The query log contains 204,549 query execution records. In order to speed up the experiments (so we can test more cases), we used 4,091 queries sampled (through stratified sampling) from the full log. The data contains a table storing the course descriptions, course codes, date course was taught, etc. The table stores 15,037 tuples.

The queries used in the experiment are to find all course instances taught before a specified date and to return the description of the most recently taught course instance. Here, a course instance is uniquely identified by the course code and the date the course was taught. The queries we used perform two-dimensional selections on the data, with the following two conditions in the 'where' clauses: the course code must be equal to a user-specified code and the date the course was taught must be before a user-specified date.

6.1.4 Synthetic query sets and data sets

A synthetic query set is made of three types of queries: *patterned (PAT)* queries, showing predictable patterns and spatial locality, used to test the proactive part of PRHist, (2) *local (LOC)* queries, showing temporal locality, used to test the data-time-reactive part of PRHist, and (3) *non-local (NON)* queries, showing no temporal (nor spatial) locality, used to test the data-reactive part of PRHist. Queries of these three types are mixed to test the adaptability of PRHist.

PAT queries are generated in groups so that each group encompasses five Gaussian (standard deviation = 4) regions and is populated with queries around the Gaussian centers. The Gaussian distribution simulates the spatial locality of queries. We use three group types based on the three clustering strategies (see Section 6.1.1): by user, by query region, and by user-region. In the first type, all the queries in a group are from the same user and distributed randomly to the five regions. In the second type, all the queries in a group are from the same user and distributed to the five regions in a repeating pattern (e.g., queries timestamped between 12:30 p.m. and 1:30 p.m. on Monday of every week are distributed to the region number one and five). In the third type, all the queries in a group are from randomly chosen users. This simulates many users querying shared data at similar times. Note that these groups may overlap in terms of both region and user. This allows us to test how well PRHist can find clusters despite the overlaps.

In addition, each group is assigned with a randomly generated cluster pattern that has both seasonal change components and trend components. The seasonal component is simulated by repeating the same pattern periodically (i.e., every day or week); the trend component is simulated by monotonously increasing (or decreasing) the frequency at a rate randomly selected between 0 and 1.

LOC queries and NON queries are generated by first generating 100 Gaussian centers, and then randomly picking one of them and generating a certain number (N_{LOC}) of queries around it. This process repeats, with one Gaussian center randomly picked each time. This creates a temporal order for the queries, where a timestamp is assigned to each query. Thus, N_{LOC} controls the temporal locality of the queries, that is, NON if $N_{LOC} = 1$ and when N_{LOC} increases, the query set exhibits more locality. We set N_{LOC} to 50 for LOC and to 1 for NON.

All the synthetic query sets used in the experiments consist 40% PAT, 30% LOC, and 30% NON (except for those used in Section 6.2.2 and Section 6.2.3 where we vary the ratio among PAT, LOC, and NON).

Synthetic query sets are labeled *daily* if their patterns repeat daily (i.e., the period of the pattern spans one day) and *daily-weekly* if repeat both daily and weekly.

We created three query logs. The first one (called *daily-2D*) consists of daily query sets spanning over six days, with 1,700 queries, executed against a two-dimensional data set (see below). The second (called *daily-weekly-2D*) and the third (called *daily-weekly-3D*) consist of daily-weekly query sets spanning over three weeks, with 3,500 queries, executed against a two-dimensional data set and a three-dimensional data set, respectively. Daily-weekly query sets are assigned with more queries because they cover longer time intervals.

All synthetic query sets occupy 1% of the volume of the entire multi-dimensional space, similarly to that used in [2].

The synthetic data sets are generated around 100 randomly generated Gaussian centers with the standard deviation of 2. Each data set consists of 20,000 data points, with 200 data points created around each of the 100 Gaussian centers. The data sets are either two-dimensional or three-dimensional.

6.1.5 Parameter settings

We set the memory limit (M) to 1 KB, similarly to that used in other works [2, 8], except in the experiment for varying memory limit. For PRHist, we set the following threshold values needed by the algorithm *Find_useful_clusters*: predictability threshold to 0.2 for synthetic query/data set and to 0.1 for the real query/data set, and spatial locality threshold to 0.95 for synthetic query/data sets and to 0.1 for the real query/data set.[3] (Section 6.2.5 examines the effects of varying these two thresholds.) Additionally, we set the minimum query frequency threshold (needed by the algorithm *Build_PHs*) to 5. The length of each f-subinterval ($I_1, I_2, \cdots, I_h$) of the future interval is set to 30 minutes. The length of the testing query sequence, used for both the predictability computation and the automatic off-line (initial) weight selection (see Section 4.2.3 and 4.4), is set to one day, and the length of each subinterval of the testing query sequence is set to 30 minutes. We used an additive HWSM model for forecasting the query frequencies since we found it gives higher prediction accuracy.

6.1.6 Performance metric

We use the same *normalized absolute error* as in [2] to measure the selectivity estimation accuracy of various histogram maintenance methods. Given a data set D, a histogram maintenance method H, and a testing query sequence S_{test}, the normalized absolute error NAE is defined as follows:

$$NAE(D, H, S_{test}) = \frac{\sum_{q \in S_{test}} |\hat{\rho}_H(q) - \rho(D,q)|}{\sum_{q \in S_{test}} |\hat{\rho}_u(D,q) - \rho(D,q)|} \quad (11)$$

where $\hat{\rho}_H(q)$ is the selectivity of query q estimated using H, $\hat{\rho}_u(D,q)$ is the selectivity of q estimated assuming the uniform distribution for D, and $\rho(D,q)$ is the actual selectivity of q. This metric is normalized with respect to how much the data deviates from the uniform distribution, and this allows us to better compare the results across data sets of different distributions. The uniform distribution is suitable as the reference distribution because, in a multi-dimensional case as PRHist is designed for, data rarely show the uniform distribution across the whole data set.

We have chosen NAE over the conventional absolute error ($\sum_{q \in S_{test}} |\hat{\rho}_H(q) - \rho(D,q)|$) or relative error ($\sum_{q \in S_{test}} \frac{|\hat{\rho}_H(q) - \rho(D,q)|}{\rho(D,q)}$) for the following reasons. An absolute error may vary significantly across different data sets, and is not so meaningful without reference to the range of the actual selectivity $\rho(D,q)$. A relative error biases the comparison against a few test queries with very low actual selectivity (i.e., $\rho(D,q)$ in the denominator).

[3]The predictability and locality vary widely among different data sets. So, we have set those threshold values through manual tuning while not necessarily seeking the optimal values. In our experiments, PRHist outperforms STHoles in the entire range of threshold values.

6.2 Experimental results

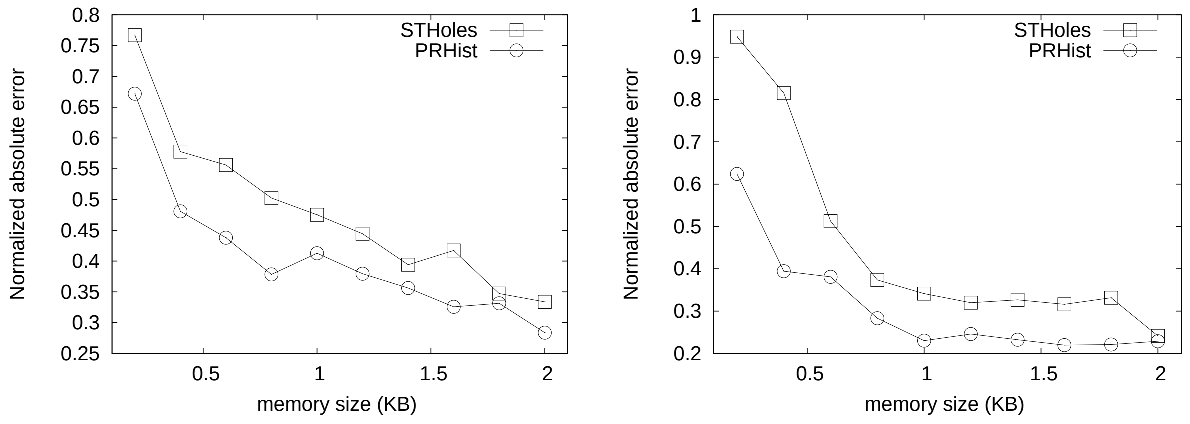
We conducted five different experiments to compare the effects of (1) varying memory limit, (2) using different bucket weighting schemes for PRHist, (3) varying the three query characteristics (PAT, LOC, NON), (4) varying the data update percentage, and (5) varying threshold values.

Due to space constraints, we cannot show the results from all possible combinations of different query sets, different data sets, and different parameter values. Thus, we omit some of those that show similar trends as those presented here.

6.2.1 Varying memory limit

This experiment examines the effects of varying the memory limit on the accuracy of STHoles and PRHist.

Figures 14(a) and (b) show the results for the synthetic query sets and data sets. We see that PRHist consistently performs better than STHoles. Considering that only 40% are PAT queries (see Section 6.1.4), and thus 60% of the queries are not predictable, this result indicates PRHist's good performance even with a small portion of PAT queries. This can be credited to three features of PRHist: using only useful clusters, using automatic off-line weighting of buckets, using on-line reactive histogram to compensate for unpredictable queries.



(a) Daily-2D (b) Daily-weekly-2D

Figure 14: Results of varying memory size when using synthetic query set and data set.

Figure 15 shows the results for the real query set and data set. It shows PRHist performs better than STHoles by up to a factor of 2.6 for the memory size between 0.6 KB and 1.4 KB and equal to STHoles outside the range. A closer examination reveals that with PRHist, the useful clusters collectively contain 15% of the queries in the real data set. This indicates that predictable patterns do exist in the real query set, and the patterns can indeed be used to improve performance.

6.2.2 Comparing different bucket weighting schemes

For this experiment, we add the following three variants of PRHist to the methods to be compared: (1) *PRH-D-reactive*, assigning weight 1 to data-reactive buckets and 0 to the proactive and data-time-reactive buckets, (2) *PRH-DT-reactive*, assigning weight 1 to data-time-reactive buckets and 0 to the proactive and data-reactive buckets, and (3) *PRH-proactive*, assigning weight 1 to proactive buckets and 0 to the reactive buckets.

In this experiment, we vary the percentages of LOC and NON to observe the effects of different weighting schemes on varying amounts of temporal locality. For example, PRH-DT-reactive should work well when there is significant temporal locality (as is the case when LOC is high).

Figure 16(a) shows the results of varying LOC. For this experiment, we have set the percentage of NON to be 5% of the queries and assigned varying portions of the remaining 95% of queries to PAT and LOC.

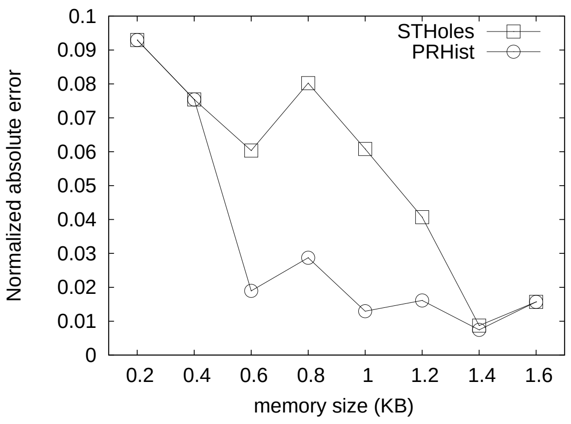
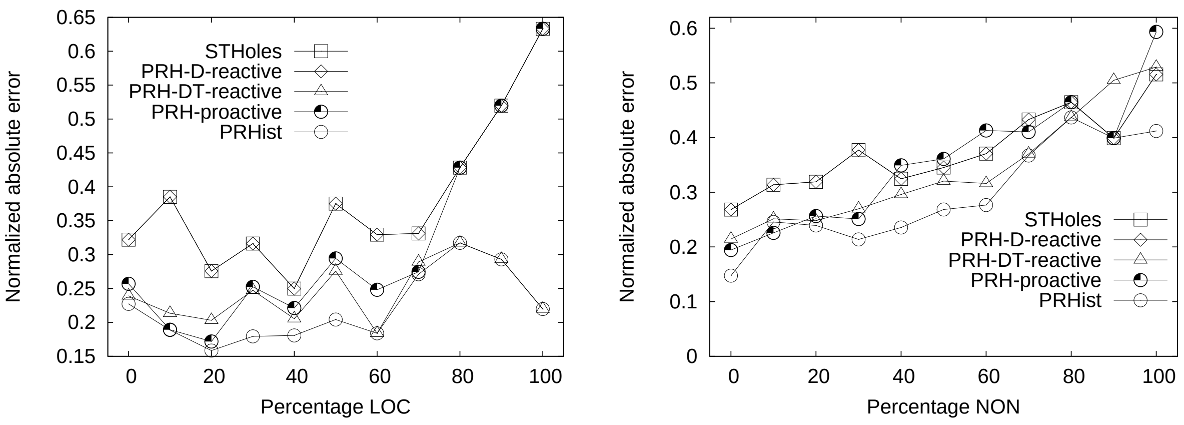


Figure 15: Results of varying memory size when using real query set and data set.

The figure shows that PRHist consistently outperforms the other weighting schemes. This demonstrates its ability to be more proactive or reactive depending on which is more beneficial.



(a) Varying percentage of LOC

(b) Varying percentage of NON

Figure 16: Results of various weighting schemes

In the same figure, the PRHist variants' weighting schemes show the expected trends as follows. First, PRH-D-reactive performs essentially the same as STHoles. This is because it does not age out old buckets and, thus, maintains the histogram using the same algorithm as STHoles. Second, PRH-DT-reactive outperforms the other methods (except PRHist) by an increasing margin as LOC percentage increases. This is because the queries are increasingly more temporally-local, which in turn benefits from PRH-DT-reactive's tendency of aging out old buckets. Third, PRH-proactive performs increasingly worse as LOC percentage increases and eventually performs the same as STHoles. This is because, as the number of PAT queries decreases, PRH-proactive is not able to find useful clusters (due to low predictability) and, thus, produces empty proactive histograms; In this case, its performance is no different from that of STHoles.

Figure 16(b) shows the results for varying NON. For this experiment, we used zero percent LOC. We can make the same observations as those for varying LOC. Due to space constraints we do not elaborate any further.

6.2.3 Varying the three query characteristics

In this experiment, we compare PRHist with STHoles across the full spectrum of PAT, LOC, and NON. This allows us to observe how consistently PRHist outperforms STHoles across different query characteristics. We use a three-dimensional plot to present the results. In the x-axis, we vary the percentage of PAT queries, and, in the y-axis, we vary the N_{LOC} parameter as specified in Section 6.1.4.

Figure 17 shows the results as the ratio of the NAEs between PRHist and STHoles. We see that PRHist outperforms STHoles consistently for the full spectrum of query characteristics.

6.2.4 Varying data update percentage

In this experiment we compare the abilities of the two histogram methods to adjust to varying update percentage. To do this, we have changed the original data to be used by queries in the test query sequence after constructing the proactive histogram sequence. The changes have been made by generating new data points around 100 randomly generated Gaussian centers (with the standard deviation of 2) and, then, randomly replacing a certain percentage of the original data points with those randomly picked among the new data points.

Figure 18 shows the results. We see that PRHist always performs better than or the same as STHoles, sometimes even when the update percentage is high. This performance advantage of PRHist stems from its ability to adjust the bucket weights depending on the percentage of data updates (see Section 5.1).

6.2.5 Varying threshold values

In this experiment, we test the effect of varying the spatial locality threshold and the predictability threshold of PRHist.

Figure 19(a) shows the results for varying spatial locality threshold. It shows that the spatial locality threshold value giving the lowest error is not at the two extremes of 0 and 1. This is because, at a low locality threshold values, clusters showing low locality can still be included as useful clusters. However, the proactive histograms built using these clusters are not concentrated in certain regions and, hence, their utility values diminish. On the other hand, when locality threshold is close to 1, there are not enough clusters selected and thus the proactive histograms built do not cover many of the queries that actually occur.

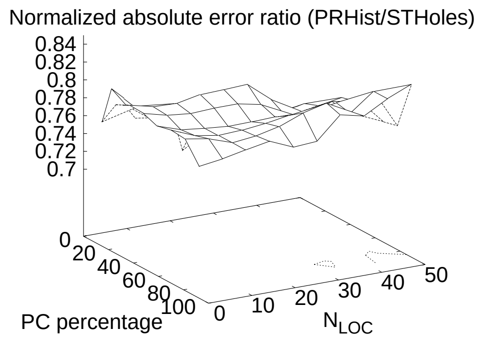
Figure 19(b) shows the results for varying predictability threshold. Again, the lowest error result does not occur at the two extremes. This is because, at low prediction threshold values, clusters that have poor prediction accuracies are included, which in turn causes poor-quality proactive histograms to be built. On the other hand, when predictability threshold value is high, few clusters are chosen and, thus, the proactive histograms built do not cover many of the queries that actually occur.

7 Conclusions

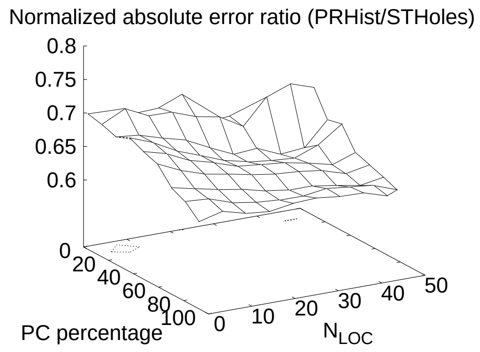
In this paper we have developed a novel multi-dimensional histogram method, called PRHist. We believe it is the first method that incorporates proactive histograms for selectivity estimation. PRHist has an off-line and an on-line phase. In the off-line phase it *proactively* builds customized histograms for query sets predicted for future time intervals. To accommodate the fact that predictability of queries may vary, PRHist uses a weighted combination of reactive and proactive histograms during the on-line phase. The weight is automatically determined based on the predictability of the queries and adjusted based on the extent of data updates.

We have performed extensive experiments comparing PRHist with the state-of-the-art reactive method STHoles, using both real and synthetic query/data sets. The results show PRHist gives lower estimation error than STHoles for most of the test cases. We have found PRHist outperforms STHoles even when only a small portion of the queries are predictable or a large portion of data are updated.

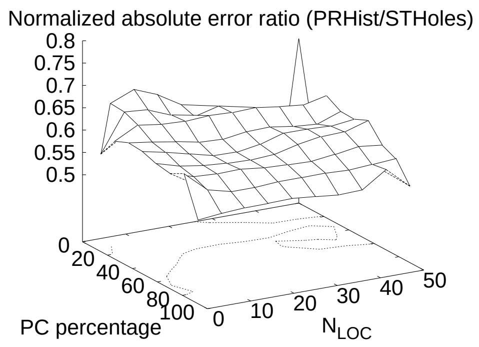
Our method can be easily incorporated into an existing database management system by extending it to handle the on-line loading and merging of proactive histograms. The reactive operation has already proven to be readily incorporable in [2]. The necessary extension is straightforward and incurs minimum additional run-time overhead.



(a) Daily-2D

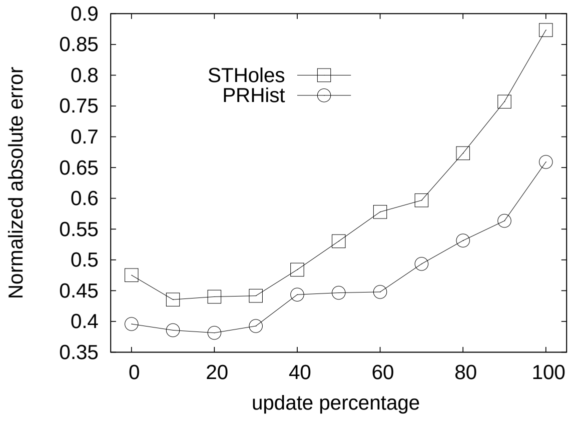


(b) Daily-weekly-2D

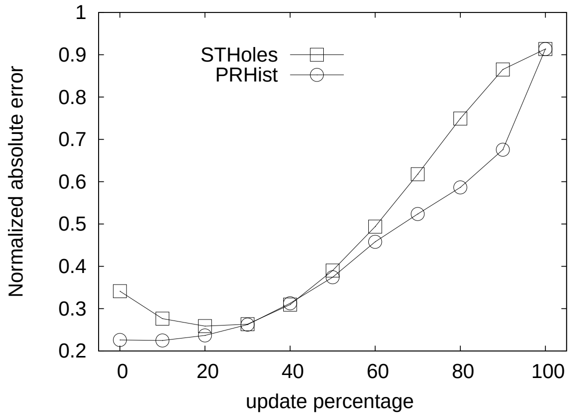


(c) Daily-weekly-3D

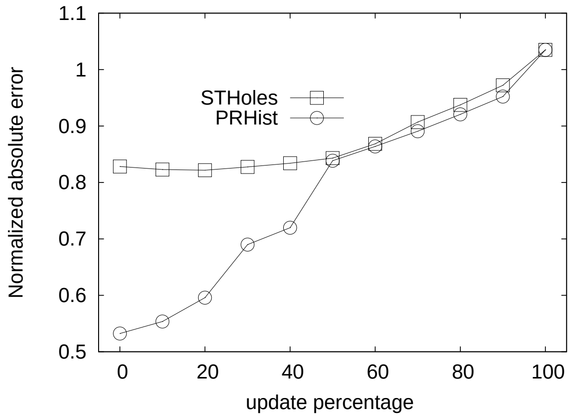
Figure 17: Results of the varying PAT, LOC, and NON.



(a) Daily-2D

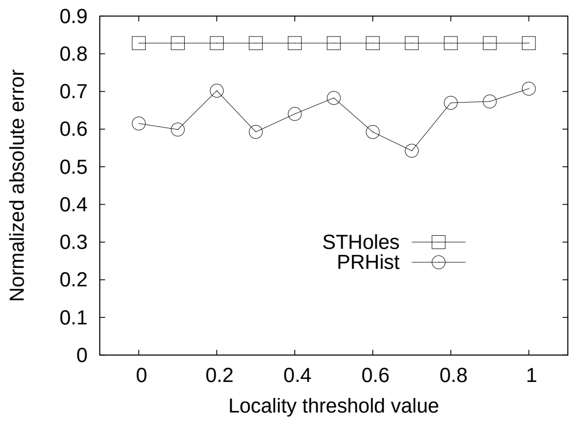


(b) Daily-weekly-2D

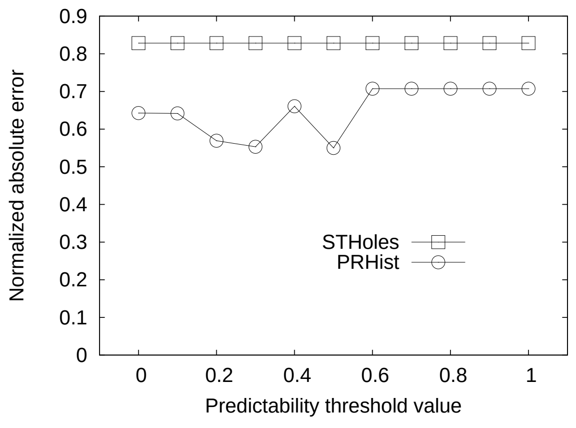


(c) Daily-weekly-3D

Figure 18: Results of varying data update percentage.



(a) Varying locality threshold



(b) Varying predictability threshold

Figure 19: Results of varying predictability and spatial locality thresholds using real queries.

We believe the key concept of PRHist, that is, predicting query patterns and using it to proactively manage a model coupled with reactive on-line maintenance, is applicable to other database applications like aggregation query answering[31, 32, 33], index selection[34], view materialization[35, 36, 37], etc. We plan to investigate these applications in our future work. Additionally, it may be possible to extend other selectivity estimation models like wavelets and sampling instead of histograms to be proactive and reactive. This is in our future work as well.

In addition, we plan to study algorithms for finding optimal f-subintervals given a future time interval and the algorithms for finding query clusters without application-specific strategies (Section 4.2.1). Furthermore, we plan to investigate the effects of using different forecasting methods (such as those described in Section 2.2) for cluster pattern extraction and query frequency prediction (Section 4.2.2). Lastly, we will explore *automated* techniques for determining the best threshold values of predictability and locality (Section 6.1.5).

Acknowledgments

We thank Dennis Fuchs for providing us with the implementation of STHoles. We also thank Nancy Snow Littlefield, Michael Grundhauser, and Keith Kennedy for providing us with the real data and query sets. This research has been supported by the US National Science Foundation through Grant No. IIS-0415023 and the US Department of Energy through Grant No. DE-FG02-ER45962.

References

[1] A. Aboulnaga, S. Chaudhuri, Self-tuning histograms: building histograms without looking at data, in: ACM SIGMOD, 1999, pp. 181–192.

[2] N. Bruno, S. Chaudhuri, L. Gravano, STHoles: a mulidimensional workload-aware histogram, in: ACM SIGMOD, 2001, pp. 211–222.

[3] M. Stillger, G. Lohman, V. Markl, M. Kandil, LEO - DB2's LEarning optimizer, in: VLDB, 2001, pp. 19–28.

[4] V. Poosala, Y. Ioannidis, Selectivity estimation without the attribute value independence assumption, in: VLDB, 1997, pp. 486–495.

[5] D. Gunopulos, G. Kollios, V. J. Tsotras, C. Domeniconi, Approximating multi-dimensional aggregate range queries over real attributes, in: ACM SIGMOD, 2000, pp. 463–474.

[6] J. Lee, D. Kim, C. Chung, Multi-dimensional selectivity estimation using compressed histogram information, in: ACM SIGMOD, 1999, pp. 205–214.

[7] M. Muralikrishna, D. J. DeWitt, Equi-depth histograms for estimating selectivity factors for multidimensional queries, in: ACM SIGMOD, 1988, pp. 28–36.

[8] V. Poosala, Y. Ioannidis, P. Haas, E. Shekita, Improved histograms for selectivity estimation of range predicates, in: ACM SIGMOD, 1996, pp. 294–305.

[9] J. S. Vitter, M. Wang, Approximate computation of multi-dimensional aggregates of sparse data using wavelets, in: ACM SIGMOD, 1999, pp. 193–204.

[10] D. Donjerkovic, Y. Ioannidis, R. Ramakrishnan, Dynamic histograms: Capturing evolving data sets, in: ICDE, 2000, pp. 411–422.

[11] P. B. Gibbons, Y. Matias, V. Poosala, Fast incremental maintenance of approximate histograms, in: VLDB, 1997, pp. 466–475.

[12] Y. Matias, J. S. Vitter, M. Wang, Dynamic maintenance of wavelet-based histograms, in: VLDB, 2000, pp. 101–110.

[13] N. Thaper, S. Guha, P. Indyk, N. Koudas, Dynamic multi-dimensional histograms, in: ACM SIGMOD, 2002, pp. 428–439.

[14] G. R. Brown, Smoothing, Forecasting and Prediction, Prentice-Hall, 1963.

[15] S. E. Gardner, E. McKenzie, Forecasting trends in time series, Management Science (1985) 1237–1246.

[16] C. Chatfield, M. Yar, Holt-winters forecasting: Some practical issues, The Statistician (1988) 129–140.

[17] K. J. Ord, B. A. Koehler, D. R. Snyder, Estimation and prediction for a class of dynamic nonlinear statistical models, Journal of American Statistical Association 92 (1997) 1621–1629.

[18] R. N. Farnum, W. L. Stanton, Quantitative Forecasting Methods, PWS-Kent, 1989.

[19] J. Fox, Applied Regression Analysis, Linear Models, and Related Methods, Sage Publications, Thousand Oaks, CA, 1997.

[20] P. Box, M. Jenkins, C. Reinsel, Time Series Analysis, Forecasting and Control, Prentice-Hall, 1994.

[21] C. Chatfield, Time-Series Forecasting, Chapman and Hall/CRC, 2001.

[22] K. M. Curewitz, P. Krishnan, J. S. Vitter, Proceedings of practical prefetching via data compression, in: ACM SIGMOD, 1993, pp. 43–53.

[23] C. A. Gerlhof, A. Kemper, A multi-threaded architecture for prefetching in object bases, in: EDBT, 1994, pp. 351–364.

[24] N. Knafla, A prefetching technique for object-oriented databases, in: British National Conference on Advances in Databases, 1997, pp. 154–168.

[25] N. Knafla, Analysing object relationships to predict page access for prefetching, in: Int. Workshop on Persistent Object Systems: Design, Implementation and Use, 1998, pp. 160–170.

[26] Z. Su, Q. Yang, Y. Lu, H. Zhang, What next: A prediction system for web requests using n-gram sequence models, in: Conference on Web Information Systems Engineering, 2000, pp. 214–222.

[27] N. Polyzotis, Y. Ioannidis, Speculative query processing, in: Processings of CIDR, 2003.

[28] S. Babu, P. Bizarro, D. DeWitt, Proactive re-optimization, in: Processings of SIGMOD, 2005, pp. 107–118.

[29] S. Kirkpatrick, C. D. Gelatt, M. P. Vecchi, Optimization by simulated annealing, Science 220 (4598) (1983) 671–680.

[30] J. MaxQueen, Some methods for classification and analysis of multivariate observations, in: Berkeley Symposium on Mathematical Statistics and Probability, 1967, pp. 281–297.

[31] A. Acharya, P. B. Gibbons, V. Poosala, Congressional samples for approximate answering of group-by queries, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, 2000, pp. 487–498.

[32] B. Babcock, S. Chaudhuri, G. Das, Dynamic sample selection for approximate query processing, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, 2003, pp. 539–550.

[33] S. Chaudhuri, G. Das, V. Narasayya, A robust, optimization-based approach for approximate answering of aggregate queries, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, 2001, pp. 295–306.

[34] G. Valentin, M. Zuliani, D. C. Zilio, G. Lohman, DB2 advisor: An optimizer smart enough to recommend its own indexes, in: Proceedings of IEEE International Conference on Data Engineering, 2000, pp. 101–110.

[35] D. Agrawal, A. E. Abbadi, A. K. Singh, T. Yurek, Efficient view maintenance at data warehouses, in: Proceedings of SIGMOD, 1997, pp. 417–427.

[36] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, H. Trickey, Algorithms for deferred view maintenance, in: Proceedings of SIGMOD, 1996, pp. 469–480.

[37] K. Salem, K. S. Beyer, R. Cochrane, B. G. Lindsay, How to roll a join: Asynchronous incremental view maintenance, in: Proceedings of SIGMOD, 2000, pp. 129–140.