Dynamic Histograms for Non-Stationary Updates

Elizabeth Lam
Equitrac Canada ULC
elizabethl@waterloo.equitrac.com

Kenneth Salem
School of Computer Science
University of Waterloo
kmsalem@uwaterloo.ca

Abstract

In this paper, we address the problem of incrementally maintaining a histogram in response to a non-stationary update process. In relational database systems, this problem can occur whenever relations model time-varying activities. We present a simple update model that is general enough to describe both stationary and non-stationary update processes, and we use it to show that existing histogram maintenance techniques can perform poorly when updates are non-stationary. We describe several techniques for solving this problem, and we use the update model to demonstrate that these techniques can effectively handle a broad range of update processes, including non-stationary ones.

1. Introduction

Database management systems maintain a variety of statistics, which are used to characterize the database. Such statistics are used for query optimization and other purposes. Histograms are widely used to characterize data distributions because they are easy to construct and they do not depend on *a priori* assumptions about the form of the distribution. However, as changes are made to the underlying database, histograms may become outdated. This problem can be addressed by periodically rebuilding histograms from scratch, but that can be expensive as it involves scanning the underlying data. An alternative is to incrementally maintain the histograms in response to updates.

Several incremental histogram maintenance techniques have been proposed – a brief overview of these techniques can be found in Section 2. These techniques have generally been shown to perform well with respect to stationary update workloads. In a stationary update workload, the probability that a particular value will be inserted into or removed from the underlying data set is stable over time. However, in some important situations, updates are not stationary. As an illustration of such a situation, consider the TPC-C transaction processing benchmark [13], which includes an ORDER table that contains a tuple for every customer order that has been entered in the system. The ORDER table includes an attribute called O_ENTRY_D, which records the order entry date. We expect that newer orders will have later dates than older orders. Thus, as new orders are entered into the ORDER table, the range of values found in O_ENTRY_D will gradually increase. This is an example of a non-stationary update workload. The ORDER table also includes an order identifier, O_ID. If order identifiers are correlated with the entry date (e.g., if the system assigns monotonically increasing order numbers), then O_ID, too, will experience non-stationary updates. Finally, consider the TPC-C NEW-ORDER table, which contains records only for recently entered transactions – tuples are added to the table as orders enter the system, and they are removed from the table as orders are fulfilled. Thus, in NEW-ORDER, we expect that both the minimum and the maximum order number will gradually increase over time. This is an example of a *rolling* update workload – one particular class of non-stationary workloads that we consider in this paper.

These examples are merely illustrations. Similar examples can be found in many other situations in which a relational attribute is used to represent time, or is correlated to such an attribute. We have found that existing incremental histogram maintenance techniques may perform poorly when faced with these kinds of update workloads. In this paper, we consider how to solve this problem. The main contributions of the paper are as follows.

- We propose a simple update model that is general enough to capture both stationary and non-stationary update processes. We use this model to generate synthetic update traces that can be used to test incremental histogram maintenance techniques under a wide variety of conditions.
- We show that existing incremental histogram maintenance techniques may not work well when

updates are non-stationary. We propose and evaluate several possible techniques for solving this problem. Our work is based on an existing incremental histogram maintenance algorithm called DADO [3, 2].

- We use the update model to evaluate our proposed techniques under a range of update workloads. Our experiments demonstrate that DADO-VRB, one of the proposed techniques, can provide effective incremental histogram maintenance across a broad range of stationary and non-stationary update processes.

2. Related Work

Most of the previous work in the literature on space constrained histograms focus on enhancing histogram accuracy through the proper placement of buckets or by considering alternative transformation methods [6, 12, 5, 11, 7]. In large, this work has deferred dealing with the issues of histogram maintenance. However, more recently several effective approaches to maintaining histograms incrementally have been proposed [4, 1, 3, 7, 2, 10]. In the rest of this section, we briefly describe the general approaches to incremental histogram maintenance previously considered and discuss their limitations with respect to non-stationary updates.

In the methods that examine data changes (i.e., insertions, modifications, and deletions) to maintain the histogram, the common approach is to update the appropriate bucket counter and then consider adjusting the bucket boundaries based on some chosen error criteria. Most proposed methods [4, 1, 3, 2] involve splitting buckets with high errors and merging similar adjacent buckets. The algorithms mainly differ in the error criteria used, the number of split-merge operations performed at a time, and the interval for considering such repartitioning operations.

Another technique involves keeping an auxiliary summary of the distribution on disk and updating it when the data change so that it can be used to maintain the main histogram. In [4], a uniform backing sample is used to rebuild the partition-based histogram from scratch when the split-merge techniques are no longer within acceptable error allowances. Similarly, in [10], which discusses the dynamic maintenance of wavelet-based histograms, both an activity log of updates and an auxiliary histogram of additional coefficients are kept on disk and used to maintain the main histogram. Unfortunately, the methods of [4] and [10] incur not only overhead storage costs but expensive disk I/O operations to maintain histograms. In addition, we note that the Approximate histograms of [4] were shown to have problems on heavy deletion loads. This is because numerous deletions deplete the backing sample rapidly making frequent scans of the relation necessary to maintain the uniform random nature of the backing sample.

The incremental algorithms [4, 3, 2, 10] that examine data changes are all empirically shown to be effective on random update processes with varying degrees of skew. But these incremental histograms have been shown to have problems on update streams with heavy deletion loads as well as sorted update streams. No evidence is available for broader classes of updates.

The maintenance techniques of [1] do not act in response to updates, but instead use feedback information from query execution engines on query workloads to refine the histograms. These histograms are referred to as self-tuning (ST) histograms because they are tuned on query feedback information. The idea behind such histograms is to finely tune the parts of the histogram where queries are concentrated.

3. Update Model

We are interested in modeling data distributions that evolve in response to a stream of updates. A data distribution is defined over some value domain. For simplicity, we assume that the value domain consists of non-negative integers in the range [1,S]. A data distribution D over such a domain is an S-element array of non-negative integers, in which D[i] ($1 \le i \le S$) represents the frequency of occurrence of value i. Thus, if we are modeling the values of the age attribute in an employee table, D[i] indicates the number of tuples (employees) for which the age is i. We define the size of a distribution to be the sum of its frequencies.

We consider two kinds of distribution updates: insertions and deletions. An insertion INSERT(i) applied to a distribution D results in a new distribution D' that is identical to D except that D'[i] = D[i] + 1. Similarly, an deletion DELETE(i) applied to D results in D' that is identical to D except that D'[i] = D[i] - 1. Since frequencies must be non-negative, we say that a delete operation DELETE(i) is valid on distribution D iff D[i] > 0. An update stream is a sequence of INSERT operations and valid DELETE operations.

3.1. Overview of the model

Our update model describes update streams as a random process characterized by the parameters shown in Table 1. It is a generative model, meaning that it describes how to produce the next update in the stream, given the updates that have already been produced. The model assumes that the update stream is applied to a distribution that is initially empty, i.e., a distribution for which **D[i] = 0** for all i in **[1,S].** A stream begins with $\mathbf{r_{init}}$ INSERT operations, which serve to initialize the distribution. These initialization updates are then followed by **L** insertion/deletion cycles, where each cycle consists of **r** INSERT operations followed by **r** DELETE operations. Thus, the total number of INSERT operations in an update stream, $\mathbf{n_{insert}}$, is given by $\mathbf{n_{insert} = r_{init} + Lr}$ and the total number of deletions is given by $\mathbf{n_{delete} = Lr}$. Furthermore, once the initialization INSERTs have been applied, the size of the distribution will cycle between a low of $\mathbf{r_{init}}$ and a high of $\mathbf{r_{init} + r}$, with a mean of $\mathbf{r_{init} + r/2}$. Note that we can model an INSERT-only update stream by setting **L = 0**.

To complete the definition of the model, we must describe how it determines the specific domain values that are inserted or deleted for the updates in the stream. We begin by describing the INSERT operations.

3.1.1. Modeling Insertions. The value to be inserted by an INSERT operation is determined randomly using an underlying probability mass function **g: [1, S] → [0, 1]**, which is defined over the value domain. This can be an arbitrary probability distribution, which can be used to model data skew. As discussed below, the inserted values are chosen in such a way that the expected number of INSERT(i) operations in the stream will be given by $\mathbf{g(i)n_{insert}}$. However, successive inserted values are not chosen independently, since we wish to be able to model non-stationary update streams. Instead, the values to be inserted are chosen using an insertion window of width $\mathbf{W_I}$, which slides across the value domain. Initially, the insertion window covers the value range $\mathbf{[1,W_I]}$. While the window is in this position, the model randomly and independently generates INSERT(i) operations, but only for those **i** in the range $\mathbf{[1,W_I]}$. After a certain number of updates have been generated with the window in this position, the window is shifted by one position, so that it covers $\mathbf{[2,W_I+1]}$. The model then randomly generates insertions of values that fall within this new range. This process continues until the window reaches its final position, $\mathbf{[S\text{-}W_I+1,S]}$. It should be clear that when $\mathbf{W_I = 1}$, the model will generate sorted insertions. Conversely, when $\mathbf{W_I = S}$, the model will generate independent random insertions, as the window will not slide at all. For values of $\mathbf{W_I}$ between **1** and **S**, the behavior will be between these extremes.

It remains to specify the number of INSERT operations that the model should generate for each position of the window, as well as the manner in which values are randomly selected from within a window. Let $\mathbf{N_x}$ represent the number of INSERT operations that the model should generate while the window is in position $\mathbf{[x,x+W_I\text{-}1]}$. Let $\mathbf{p_x(i)}$ represent the probability that the model generates INSERT(i) given that the window is in that position. We would like to choose these values so that the following two conditions will hold:

- $\sum_x N_x = n_{insert} = r_{init} + Lr$
- For each i, $\sum_x N_x p_x(i) = g(i) n_{insert}$

The former condition ensures that the model generates the desired number of INSERT operations, and the latter condition ensures that the expected number of INSERT(i) operations will be determined by the underlying probability distribution. These conditions can be achieved as follows:

- $$N_x = (r_{init} + rL) \sum_{x \le k \le x+W_I-1} g(k)/m_k \qquad (1)$$
- For each i in the window $\mathbf{[x,x+W_I\text{-}1]}$,
$$p_x(i) = g(i)/m_i \Big/ \sum_{x \le k \le x+W_I-1} g(k)/m_k \qquad (2)$$
For each i outside of the window $\mathbf{[x,x+W_I\text{-}1]}$,
$$p_x(i) = 0$$

where

$$m_x = \begin{cases} \min(x, W_I), if\, 1 \le x \le S - W_I + 1, \\ \min(S - x + 1, S - W_I + 1), otherwise \end{cases}$$

3.1.2. Modeling Deletions. Deletions, like insertions, are controlled by a window that slides across the value domain. The deletion window is distinct from the insertion window. It has width $\mathbf{W_D}$ and it moves separately from the insertion window. Deletions are handled somewhat differently from insertions because we wish to ensure that deletions are valid, i.e., we do not wish to delete values that do not exist in the data distribution.

The deletion window starts at position $\mathbf{[1,W_D]}$ and, like the insertion window, slides "right". Suppose that the deletion window is at position $\mathbf{[x,x+W_D\text{-}1]}$, the current data distribution is **D**, and the next update operation to be generated by the model is a deletion.

Let $\mathbf{p_{x,D}(i)}$ represent the probability that the model generates DELETE(i). The probability $\mathbf{p_{x,D}(i)}$ is defined as follows:

- For each i in the window,
$$p_{x,D}(i) = D[i] \Big/ \sum_{x \le k \le x+W_D-1} D[k]$$

Table 1. Update model parameters

Parameter	Description of use	Valid values and restrictions
S	Size of the underlying value domain	$S \geq 1$, is an integer
g: [1, S] → [0, 1]	g is a probability mass function that represents the relative frequency distribution for the insertions	$\sum_{1 \leq i \leq S} g(i) = 1$, $0 \leq g(i) \leq 1$
$\mathbf{W_I}$	Width of the insertion window	W_I is an integer such that $1 \leq W_I \leq S$
$\mathbf{W_D}$	Width of the deletion window	W_D is an integer such that $1 \leq W_D \leq W_I$
$\mathbf{r_{init}}$	Length of the initial run of insertions	$r_{init} \geq 0$, is an integer
r	Number of insertions or deletions per cycle	$r \geq 0$, is an integer
L	Number of cycles	$L \geq 0$, is an integer

- For each i outside of the window,

$$p_{x,D}(i) = 0$$

This ensures that the model does not generate DELETE[i] unless **D[i] > 0**, so that all deletions are valid.

After generating a deletion, the model considers advancing the deletion window. The deletion window is advanced only if **D[x] = 0**. If the window is moved, its left edge is moved to the smallest value **y** such that **D[y] > 0**. In addition, the deletion window is constrained such that is left edge must be equal to or less than the left edge of the insertion window. If the deletion window cannot be moved without violating this constraint, then it is not moved.

3.2. Expressiveness of the model

The proposed update model can generate both stationary and non-stationary updates. In general, when one sets $W_I < S$ or $W_D < S$, the update model generates a non-stationary update process. If one sets $W_D = W_I = S$, the update model instead generates a stationary, random update process. Table 2 shows the model settings that can be used to generate types of update streams (of length r_{tot}) that have been considered in previous evaluations of incremental histogram maintenance techniques. An entry of "-" indicates that the parameter may be set to any valid value.

In addition, the model can generate non-stationary update streams characterized by having value ranges that slide over time. We refer to processes exhibiting such trends as either "rolling" or "fuzzy rolling". A rolling process consists of sorted insertions intermixed with deletions of the older data in sequential order of insertion. For instance, a rolling process can be used to model timestamps in a database logs window. Similarly, a fuzzy rolling process also exhibits the general trend of the updates sliding across the value domain over time, but the order among the updates is more relaxed and not strictly sorted. These update streams can be found in a wide variety of real world applications where the recent data is of greater interest. Table 3 shows the model settings to generate update streams of length r_{tot} for these types of updates.

Table 2. Parameter settings for common types of update streams

Stream Type	W_I	W_D	r_{init}	r	L
Random Insertions	S	-	r_{tot}	0	0
Sorted Insertions	1	-	r_{tot}	0	0
Random Mixture	S	S	$r_{tot} - 2rL \geq 0$	$r > 0$	$L > 1$
Random Insertions followed by Random Deletions	S	S	$r_{tot} - 2r \geq 0$	$r > 0$	1
Sorted Insertions followed by Sorted Deletions	1	1	$r_{tot} - 2r \geq 0$	$r > 0$	1

Table 3. Parameter settings for additional types of update streams

Stream Type	W_I	W_D	r_{init}	r	L
Rolling Process	1	1	$r_{tot} - 2rL \geq 0$	$r > 0$	$L > 0$
Fuzzy Rolling Process	$1 < W_I << S$	$1 < W_D \leq W_I$	$r_{tot} - 2rL \geq 0$	$r > 0$	$L > 0$

4. Dynamic Average-Deviation Optimal (DADO) Algorithm

The techniques proposed in this paper are extensions of the DADO algorithm [3, 2], which builds and maintains a histogram dynamically without needing to directly access the underlying data on disk. Instead, the DADO histogram is continuously updateable in response to changes made to the underlying data. The goal of the DADO algorithm is to dynamically approximate the minimization of the overall sum of absolute values of deviations of frequencies from their average within each bucket, which is written as:

$$\varepsilon = \sum_{i=1}^{n} \sum_{j} \left| f_{ij} - \bar{f}_i \right| \tag{3}$$

where n denotes the number of buckets, f_{ij} denotes the frequency of the jth value in the ith bucket and $\bar{f}_i$ is the average frequency in the ith bucket. Here j is assumed to run over all possible domain values within the ith bucket. That is, the DADO algorithm uses the continuous values assumption to approximate the true values that are in each bucket.

The DADO algorithm cannot directly minimize (3) because the f_{ij}'s are unknown short of scanning the entire relation or storing all the frequencies. Instead, each bucket is divided into two parts of equal value-range width, called sub-buckets, each with its own count. The uniform frequency assumption is applied to each sub-bucket separately to approximate the corresponding frequencies by their average.

The DADO algorithm approximates the dynamic minimization of (3) by using the following split and merge operations:

- **Split operation:** A bucket is split along the sub-bucket border to generate two new buckets. For each new bucket, the sub-buckets have equal counts and the sub-bucket border is based on an equal-width partition of the corresponding original sub-bucket.
- **Merge operation:** Two adjacent buckets are merged to generate a single new bucket. The sub-bucket counts of the new bucket are calculated based on the counts and ranges of the original buckets. The sub-bucket border of the new bucket is based on an equal-width partition of the combined value range of the original buckets.

First, the DADO histogram is initialized by loading the first n distinct points into individual buckets, where n is the total number of buckets allowed with the available space. Then on each subsequent update, the algorithm adjusts the appropriate bucket counter after which it decides whether or not to repartition the bucket boundaries. Repartitioning in the DADO algorithm consists of splitting a bucket with frequencies that highly deviate from the bucket average and merging two adjacent buckets that are similar to each other.

In addition, an effective static histogram, the Successive Similar Bucket Merge (SSBM) histogram, based on the same merge error criteria as DADO, is introduced in [3]. The SSBM histogram is constructed by initially loading all the distinct values and the empty spaces between them into an exact histogram. The algorithm then successively merges adjacent buckets using the same criteria as DADO for best merge candidates, until the histogram size is reduced to the space allowed. In their experiments, the SSBM histogram is used as a measure to evaluate the performance of the dynamic histograms.

4.1. Performance of DADO

In [3], the DADO histogram is empirically compared against the Approximate Compressed (AC) histogram of [4] and shown to consistently outperform the AC histogram on a variety of update streams. The types of updates considered correspond to the common classes of updates listed in Table 2.

In particular, the DADO histogram is shown to be highly effective in capturing random updates; its performance approached that of the static SSBM histogram. However, the DADO algorithm was shown to have problems on sorted update streams.

We conducted further evaluations of DADO on several types of non-stationary update streams that were not considered in the earlier study such as rolling update streams [8]. We found that DADO has problems with both sorted insertions and rolling updates where the data from one end of the value domain is deleted over time. We also found that the performance of DADO greatly deteriorates with larger domain sizes when the updates are in sorted order. The details of our study are available in [8].

5. New Incremental Histogram Maintenance Techniques

The types of non-stationary update processes that DADO has been shown to perform poorly on include those with variable value range over time as well as those with large sparse value domain. In general, merge operations lose information and so reduce histogram accuracy. Unfortunately, the frequent expansion and contraction of the value range, as results from sorted update patterns, causes frequent merges involving the end buckets.

In this section, we propose several techniques to address the problems exhibited by DADO on certain classes of non-stationary updates. These techniques could be extended to maintain other types of partition-based histograms where buckets are repartitioned based on some criteria in response to updates.

5.1. Adapting to Variable Value Ranges

Methods to enable the histogram to expand and contract its range dynamically would increase its ability to capture non-stationary distributions with variable value ranges over time, such as sorted updates. We refer to the proposed changes regarding range expansion and contraction, when applied to DADO, as the DADO Variable Range (DADO-VR) algorithm.

- **Range Expansion:** Currently, when an out of range insertion x occurs, DADO extends the range of the histogram buckets by temporarily assigning x its own bucket. However, any empty space between x and the closest end-point of the histogram is not explicitly represented. This is a problem because DADO does not store the maximum value (right border) of each bucket, but assumes it to be one less than the minimum value of the right neighboring bucket based on the continuous values assumption. As a result, any new empty space generated from an out of range insertion is implicitly merged with existing values in the original end bucket. Therefore, additional errors are introduced, which are further propagated in subsequent histogram repartitions.

Our approach to the problem is to create an additional bucket to represent the generated empty space. We then perform an additional merge. This method is preferable to DADO's approach because it performs two explicit merges based on the least error criterion. However, this approach may take slightly more time than the original DADO algorithm because of the extra merge.

- **Range Contraction:** Deletions to the underlying data can contract the range of the data distribution. Unfortunately, the DADO algorithm is unable to detect precisely when the range of the underlying data contracts because the actual data on disk is not directly accessed. Currently the DADO algorithm does not permit the range of the histogram to contract for deletions. This is a problem for update streams where data from one end is deleted over time. For such update streams, histogram accuracy deteriorates over time as space is wasted on approximating the distribution of values no longer found in the underlying data.

Our approach to contracting the histogram range is to delete an end bucket whenever a deletion falls in that bucket and causes it to become empty. Any recovered bucket allows us to split a non-empty bucket. Note that only empty end buckets are deleted to recover space and allow other buckets to be split. Therefore, the goal of approximating the minimization of the error quantity (3) is not compromised when an empty end bucket is deleted to contract the range of the histogram. However, with this approach it is possible to have subsequent out of range deletions. This is possible because the histogram is an approximation and although its frequency count for a value may be zero, the underlying frequency may be positive. Such deletions must be removed from somewhere to preserve the total mass of the histogram accurately. An out of range deletion is handled by decrementing the appropriate sub-bucket counter of the nearest bucket to the deleted value point.

- **Alternative approach to range expansion:** An alternative approach to expanding the range of the histogram is to simply store the right border of each bucket as well. However, this method requires more space per bucket. We refer to this method as the sparse DADO algorithm. We expect this method to perform worse than DADO on updates with few out of range insertions because it results in approximately 25% fewer buckets than DADO for the same amount of space.

5.2. Batch Processing of Updates

Update streams from real world applications are likely to contain insertions or deletions of identical values that occur in close succession. The updates are then said to occur in batches. We have seen DADO perform poorly on batched data such as sorted data. DADO performs poorly on batched updates because it adjusts the histogram on each individual update and hence is slow in recognizing significant trends. This is a problem for non-stationary batched updates because the algorithm may perform excessive bucket repartitions and thus lose a significant amount of information. We propose to process the updates in batches instead of individually to improve histogram accuracy. We refer to the proposed changes when applied to DADO as the DADO Batch Update algorithm.

To facilitate histogram adaptation to batched updates, we track individual recently updated values separately from the main histogram. From the total space available, we propose to set aside space equivalent to $k \geq 1$, singleton buckets to track the k most recent values. The singleton buckets store the value and a count representing the value's net frequency from insertions and deletions. The corresponding singleton bucket counter is incremented or decremented by one for an insertion or a deletion of the value, respectively. We allow the count to become negative because at all times, we use both the frequency approximation from the main histogram and the corresponding singleton bucket count to arrive at an estimation of the value's frequency. If the frequency estimation from the combined information is negative, the frequency of the value is then approximated as zero and the outstanding counts are moved to neighboring buckets.

Since we have a finite number k of singleton buckets to track recent values, when all k tracking buckets are in use and a different value occurs, the least recently updated tracking bucket c_{LRU} is integrated into the main histogram and the new value is given its own (singleton) tracking bucket. The integration of c_{LRU} is accomplished by partitioning the bucket that contains the tracked value, c_{LRU}.value, into at most four distinct parts: the sub-bucket values less than c_{LRU}.value, c_{LRU}.value, the sub-bucket values greater than c_{LRU}.value and the other sub-bucket that c_{LRU}.value does not fall within. Each of these partitions is temporarily given its own bucket. Then we merge best candidate adjacent buckets until the histogram size is restored. At most three merges are required since at most four new buckets are created.

By using a queue of singleton buckets to track a number of most recent values, the algorithm is able to see a larger window of changes between successive bucket repartitions. However, accuracy is not compromised since at any given time, all the summary information available in both the main histogram and the k tracking buckets are used for query selectivity estimation. In addition, by separating the value that actually occurred in updates from other values in the same bucket, the frequency prediction of that value can be precisely updated. This allows values with high frequencies to be approximated more accurately and hence leads to improved histogram accuracy. However, a drawback of this approach is that fewer regular buckets are available because some of the space is used instead to track the k most recent values. The required space for tracking the k recent values is roughly equivalent to 2/3 k regular DADO buckets.

6. Experimental Evaluation

We have proposed several techniques to address existing histogram maintenance problems on certain classes of non-stationary updates. We combine the techniques and apply them to the DADO algorithm to derive the following new dynamic histograms:

- Sparse DADO
- DADO Variable Range (DADO-VR)
- DADO Variable Range Batch Update (DADO-VRB)

The name of the algorithm indicates which techniques have been combined. In the rest of this section, we present and analyze the results of our empirical evaluation of the proposed dynamic histograms.

6.1. Experimental Environment

First, we describe the experimental environment of our empirical study.

Error Measure: We use the KS statistic [9] as an error metric to evaluate the quality of a histogram as is done in [3]. The KS statistic for two distributions is defined as

$$t = \max_{-\infty<x<\infty} \left| P_1(x) - P_2(x) \right| \qquad (4)$$

where $P_1(x)$ and $P_2(x)$ are cumulative distribution functions.

The KS statistic has an intuitive interpretation for range predicates. The selectivity of a range predicate is the fraction of tuples in the relation that satisfy the range predicate. The selectivity estimate of any range predicate posed against the histogram rather than the original data will have an absolute error that is less than or equal to 2t.

Relative Insertion Distributions: Past experimental work [4, 1, 3, 10] has used permuted Zipf [14] distributions to evaluate proposed histograms. The Zipf distribution is widely used because many real-world data appear to follow Zipf laws. In our experiments we also used permuted Zipf distributions to determine the underlying data distribution g of our model. Specifically, the set of frequencies and values are chosen independently using Zipf, and then the frequencies are randomly mapped to the values.

- **Frequency Sets:** The frequencies are generated following a Zipf distribution with z = 1 (i.e., skew of 1).
- **Value Sets:** The values are determined from the individual spreads, which are the distances between successive values. The spreads are generated following a Zipf distribution with z = 1 (i.e., skew of 1) and then randomly reordered to obtain the values.

Update Streams: We used the update model to generate the update streams tested on. Full details of the parameter settings used are available in [8]. Unless otherwise noted, the number of distinct values |V| is fixed at 1000.

Methodology and Histogram Settings: We compare the performance of the dynamic histograms not only against each other, but against the statically built SSBM histogram. All the dynamic histograms are initially empty and are populated entirely based on the updates. The SSBM histogram is built on the final data distribution after all the updates are applied to the underlying distribution. After the entire update stream is processed, each histogram is compared to the real (net) distribution using the KS statistic as an error metric. Every test configuration was generated multiple times (by starting from a different random seed for the random number generators used in generating the update streams) and evaluated based on the average of the measured KS statistics.

For fair comparison, all histograms were given the same amount of memory. In our experiments, the default amount of space given to each histogram is 1 KB, which is the same default used in [3]. For the DADO-VRB histogram, the default amount of space allocated for tracking individual recent values is 5% of the total amount of space available. This corresponds to six singleton buckets in our experiments.

6.2. Experimental Results

In our experiments, we varied S to study the effect of spreads on histogram performance and varied W_I and W_D to evaluate how well the histograms perform on updates that fall in between the extremes of random and sorted orders.

Performance on Sorted Updates: For sorted updates, we are interested in studying the effect of spreads (i.e., distances) between values on histogram performance. Figure 1 shows results for insertions only. The results for mixtures of sorted insertions and deletions are similar and available in [8].

In the experiments depicted in Figure 1, we fixed the number of distinct values to 1000 and varied S from 5000 to 40000. Since the number of distinct values is fixed, by varying S we vary the magnitudes of the spreads. From the results, it is clear that the accuracy of DADO deteriorates greatly as S increases. In contrast, the accuracy of the other dynamic histograms just modestly degrades with increasing S. We note that DADO-VRB not only significantly outperforms the other dynamic histograms, but it also comes close to the performance of the statically built SSBM histogram (on the final data distribution) at every magnitude of S tested. Clearly, the proposed batching and variable range techniques are highly effective for capturing sorted updates.

Performance on Fuzzy Updates: We study the effectiveness of the dynamic histograms for capturing non-stationary updates that fall in between the extremes of random and sorted orders by varying W_I and W_D, the window widths for generating insertions and deletions, respectively. In our experiments, we set $W_I = W_D$ and varied their values as percentages of S, the maximum domain value. Random updates correspond to $W_I = W_D$ = 100% of S, while strictly sorted updates are achieved as the window size approaches zero. Figure 2 shows results for insertions only. The results for mixtures of insertions and deletions are similar and available in [8].

In Figure 2, we notice that all the dynamic histograms perform significantly better on the random update streams than on the non-stationary fuzzy update streams (i.e., updates that fall in between the extremes of random and sorted orders) tested. However, the performance of each dynamic histogram does not appear to vary significantly across the intermediate sized window widths (i.e., window widths from 10% - 80% of S). We also note that for smaller window widths, DADO-VRB clearly outperforms the other dynamic histograms because of batching.

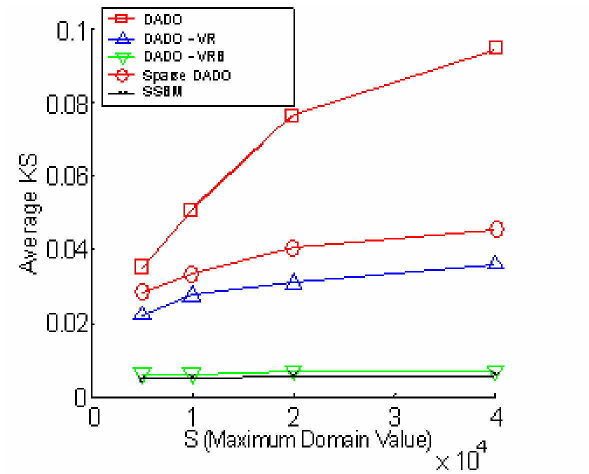

Figure 1. Error vs. Domain Size
Sorted insertions, Value range = [1, S], |V| = 1000, Stream size = 100K, Memory = 1 KB

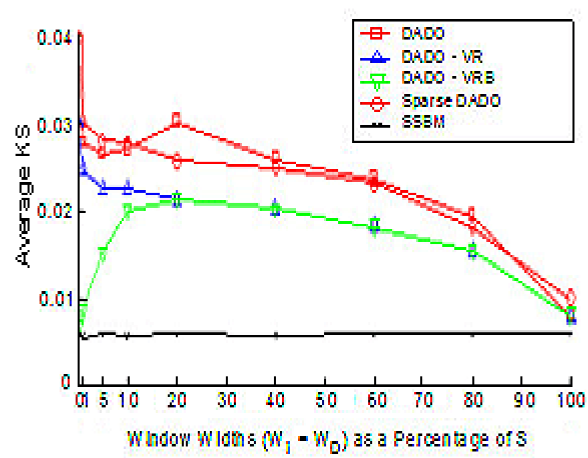

Figure 2. Error vs. Window Widths
Insertions, S = 20000, Value range = [1, S], |V| = 1000, $W_I = W_D$, Stream size = 100K, Memory = 1 KB

Histogram performance over time: In Figure 3, we measured the KS statistic of each proposed dynamic histogram at different fractions of the update stream loaded. We can see that the performance of each dynamic histogram degrades as more updates are processed. However, for all the dynamic histograms the rate of decline in accuracy appears to stabilize.

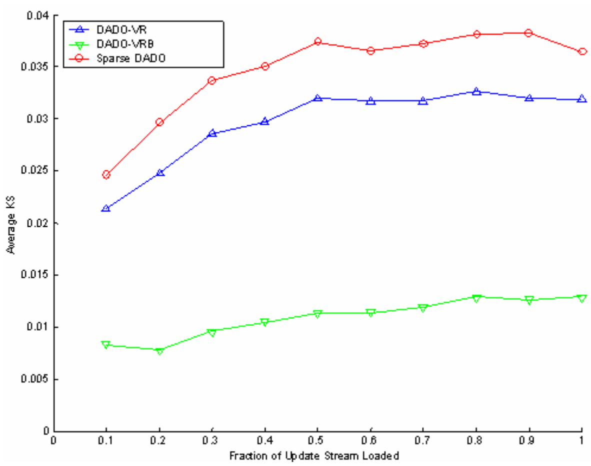

Figure 3. Error vs. Fraction of stream loaded

Fuzzy insertions and deletions, S = 60000, Value range = [1, S], |V| = 3000, $W_I = W_D = 0.01S$, Stream size = 500K, Memory = 1 KB

Other Experiments: In other experiments, we observed that on random updates the performances of the proposed dynamic histograms did not vary significantly from that of DADO. We also varied the amount of tracking space for DADO-VRB and found that its performance initially improves when the amount of tracking space is increased from low levels, but eventually it worsens with increased levels because of the trade-off of having fewer regular buckets. More details are available in [8].

7. Conclusions

In this paper, we introduced several techniques for capturing broader classes of non-stationary updates and applied them in effective ways to the DADO algorithm. In addition, we introduced a general but still realistic update model for database changes. Our experimental results show that the proposed dynamic histograms offer greater accuracy than DADO for capturing broader classes of updates than were considered in previous studies. In particular, the DADO-VRB histogram is shown to consistently outperform the others.

8. References

[1] A. Aboulnaga and S. Chaudhuri. Self-tuning Histograms: Building Histograms Without Looking at Data. In *Procreedings of the 1999 ACM SIGMOD International Conference on Management of Data*, pages 181-192, 1999.

[2] D. Donjerkovic, Y. Ioannidis, and R. Ramakrishnan. Dynamic Histograms: Capturing Evolving Data Sets. In *Proceedings of the 16th International Conference on Data Engineering*, page 86, 2000.

[3] D. Donjerkovic, Y. Ioannidis, and R. Ramakrishnan. Dynamic Histograms: Capturing Evolving Data Sets. *Technical Report* CS-TR-99-1396, University of Wisconsin-Madison, 1999.

[4] P. B. Gibbons, Y. Matias, and V. Poosala. Fast Incremental Maintenance of Approximate Histograms. In *Proceedings of the 23rd International Conference on Very Large Databases,* pages 466-475, 1997.

[5] H. V. Jagadish, N. Koudas, S. Muthukrishnan, V. Poosala, K. Sevcik, and T. Suel. Optimal Histograms with Quality Guarantees. In *Proceedings of the 24th International Conference on Very Large Databases,* pages 275-286, 1998.

[6] R. P. Kooi. *The optimization of queries in relational databases.* PhD thesis, Case Western Reserve University, September 1980.

[7] J. H. Lee, D. H. Kim, and C. W. Chung. Multi-dimensional Selectivity Estimation Using Compressed Histogram Information. In *Procreedings of the 1999 ACM SIGMOD International Conference on Management of Data,* pages 205-214, 1999.

[8] E. Lam and K. Salem. Dynamic Histograms for Non-Stationary Updates. *Technical Report* CS-2005-18, School of Computer Science, University of Waterloo, May 2005.

[9] F. J. Massey. The Kolmogorov-Smirnov test for goodness-of-fit. *Journal of the American Statistical Association*, 46: 68-78, 1951.

[10] Y. Matias, J. S. Vitter, and M. Wang. Dynamic Maintenance of Wavelet-Based Histograms. In *Proceedings of the 26th International Conference on Very Large Databases,* pages 101-110, 2000.

[11] Y. Matias, J. S. Vitter, and M. Wang. Wavelet-Based Histograms for Selectivity Estimation. In *Procreedings of the 1998 ACM SIGMOD International Conference on Management of Data,* pages 448-459, 1998.

[12] V. Poosala, Y. E. Ioannidis, P. J. Haas, and E. J. Shekita. Improved Histograms for Selectivity Estimation of Range Predicates. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 294-305, 1996.

[13] TPC Benchmark C, Revision 5.0. Transaction Processing Performance Council, 2001.

[14] G. K. Zipf. *Human Behavior and the Principle of Least Effort.* Addison-Wesley, Reading, MA, 1949.