Dynamic Maintenance of Wavelet-Based Histograms

Yossi Matias[*] *Jeffrey Scott Vitter*[†] *Min Wang*[‡]

Abstract

In this paper, we introduce an efficient method for the dynamic maintenance of wavelet-based histograms (and other transform-based histograms). Previous work has shown that wavelet-based histograms provide more accurate selectivity estimation than traditional histograms, such as equi-depth histograms. But since wavelet-based histograms are built by a nontrivial mathematical procedure, namely, wavelet transform decomposition, it is hard to maintain the accuracy of the histogram when the underlying data distribution changes over time. In particular, simple techniques, such as split and merge, which works well for equi-depth histograms, and updating a fixed set of wavelet coefficients, are not suitable here.

We propose a novel approach based upon probabilistic counting and sampling to maintain wavelet-based histograms with very little online time and space costs. The accuracy of our method is robust to changing data distributions, and we get a considerable improvement over previous methods for updating transform-based histograms. A very nice feature of our method is that it can be extended naturally to maintain multidimensional wavelet-based histograms, while traditional multidimensional histograms can be less accurate and prohibitively expensive to build and maintain.

[*]Department of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel. Also affiliated with Bell Laboratories, 600 Mountain Ave., Murray Hill, NJ 07974–0636. email: matias@math.tau.ac.il

[†]Center for Geometric Computing, Department of Computer Science, Duke University, Durham, NC 27708–0129. Part of this work was done while the author was on sabbatical at I.N.R.I.A. in Sophia Antipolis, France. Email: jsv@cs.duke.edu

[‡]Contact author. IBM T. J. Watson Research Center, 30 Saw Mill River Road, Hawthorne, NY 10532. Part of this work was done while the author was at Duke University and visiting Stanford University. Phone: (914) 784–6268, FAX: (914) 784-6079, Email: min@us.ibm.com

1 Introduction

Several important components in a database management system (DBMS) require accurate estimation of the selectivity of a given query. For example, query optimizers use the information to evaluate the costs of different query execution plans and choose the preferred one. Histograms are the major data structures used to capture the distribution of the data stored in a database and are used to guide selectivity estimation as well as approximate query processing, load balancing, etc..

In general, there are three issues to be considered when judging a histogram:

1. *Accuracy* of the histogram. Since the accuracy of selectivity estimation depends upon the histograms used, how to build better histograms that approximate the underlying data distributions more accurately is always important. A good histogram should characterize the underlying data distribution with a reasonable accuracy.
2. *Maintenance efficiency* of the histogram. When the underlying data distribution changes, the old histogram needs to be updated to reflect the new distribution; otherwise, significant estimation error could occur because of the outdated histogram. One way to do the update is to recompute and rebuild a new histogram based upon the new data distribution, from scratch, which is costly and still used in most commercial DBMSs. It is very preferable if a histogram can be maintained efficiently on-line. This issue has attracted much attention recently [GMP97, AC99, LKC99, DIR99].
3. *Extension to multidimensional* data. For multidimensional histograms that capture joint distribution of correlated attributes, achieving good accuracy and designing efficient maintenance methods becomes particularly difficult [MD88, PI97, MVW98, AC99, VWI98, VW99].

Equi-depth histograms [PSC84] are the most popular histograms and are used in many commercial DBMSs. They are also easy to implement. In some cases they provide good guidance in selectivity estimation and other data processing tasks. Several recent works have dealt with their maintenance [GMP97, AC99].

However, equi-depth histograms have major difficulties with complex queries, complex data distributions, and especially multidimensional data. They are built based upon data partitions and maintained by simple mechanisms to keep partition structures. It is very difficult to capture complex multidimensional data distributions with simple partitions. If more advanced partition methods are used, the accuracy becomes better, but the implementation and maintenance become difficult.

In [MVW98], we introduce a new type of histogram that is based upon the powerful mathemat-

ical tool of wavelets and multiresolution analysis. The wavelet-based histogram is fundamentally different from traditional approaches and offers noticeable improvements in accuracy over traditional equi-depth histogram and other leading histogram methods, like the ones in [PIHS96]. A nice feature of the wavelet-based histogram is that it can be naturally extended to the multidimensional case.

While wavelet-based histograms have very little CPU and storage cost at query optimization time, rebuilding a histogram when the underlying data distribution changes is usually very expensive since it involves scanning or sampling the data and performing the wavelet decomposition from scratch. To make the wavelet-based histogram the histogram of choice for future database optimizers, we have one more important step to go, namely, designing efficient dynamic maintenance methods for both one-dimensional and multidimensional wavelet-based histograms, and we accomplish that goal in this paper.

The rest of the paper is organized as follows: In the next two sections we review related work and formalize our problem. In Section 4, we describe the general ideas of doing dynamic maintenance for wavelet-based histograms and outline our method for one-dimensional case. Our method works for transform methods in general, although for brevity in this paper we confine ourselves to the case of wavelets. We extend our method to the dynamic maintenance of multidimensional wavelet-based histogram in Section 5. We present our experimental results in Section 6 and draw conclusions in Section 7.

2 Previous Work

Histograms are the most widely used form to store succinct statistical information of the data distribution in a database. Many different histograms have been proposed in the literature and some of them have been deployed in commercial DBMSs. However, almost all previous histograms have one thing in common, that is, they use buckets to partition the data, although in different ways.

The most popular histogram in today's DBMSs is equi-depth histogram [PSC84]. It partitions the interval between the minimum and maximum attribute value of an attribute into consecutive subintervals so that the total frequency of the attribute values for each subinterval is the same. Poosala et al. [PIHS96, Poo97] propose a taxonomy of partition-based histograms; new histograms types can be derived by combining effective aspects of different histogram methods. Among the histograms discussed in [PIHS96, Poo97], the MaxDiff(V, A) histogram gives the best overall per-

formance in terms of an accuracy/time trade-off.

When a query involves multiple attributes, its selectivity depends upon the joint distribution of all involved attributes. To simplify things, almost all commercial DBMSs make the *attribute value independent assumption*. This approach, although efficient computationally, is very inaccurate and often leads the query optimizer to generate poor query execution plans. To solve the multidimensional selectivity estimation problem in a better way, Muralikrishna and DeWitt [MD88] propose a spatial index partitioning technique for constructing multidimensional equi-depth histograms. Poosala and Ioannidis [PI97] give two effective alternatives.

All the histograms discussed so far follow the same general guideline, that is, they partition the underlying data in some particular fashion. The effectiveness and accuracy of various partition methods highly depend upon the data distributions and query types. If the data have a simple distribution, the histograms can capture the data distribution effectively. If the data have a very complex distribution, the construction becomes more difficult. In general, using a specific partition method to capture many unpredictable data distributions is a hard job, especially for multidimensional data. It is desirable to study general methods to construct histograms whose usability is more robust.

In [MVW98] we introduce a new type of histograms (wavelet-based histograms) based upon a multidimensional wavelet decomposition. A wavelet decomposition is performed on the underlying data distribution, and the most significant wavelet coefficients are chosen to compose the histogram. In other words, the original data are "compressed" into a set of numbers (wavelet coefficients) via a sophisticated mathematical transformation. Those coefficients constitute the final histogram. This approach offers more accurate selectivity estimation than traditional partition-based histogram methods and can be extended very naturally to efficiently compress the joint distribution of multiple attributes. Later, a similar approach was discussed using a different mathematical transformation procedure called discrete cosine transform (DCT) [LKC99].

Despite the popularity of histograms, the important issue of their maintenance has attracted attention only recently, and most of the work has been on the maintenance of the traditional partition-based histograms. Gibbons et al. [GMP97] propose sampling-based approaches for incremental maintenance of one-dimensional equi-depth and compressed histograms. In [AC99], a novel approach of building self-tuning histograms is introduced and it can be effectively used for maintaining both one-dimensional and multidimensional equi-depth histograms. Basically, the approach in [AC99] is not about building a new type of histogram, but about adjusting a given partition-based histogram through a learning procedure to be more accurate.

While wavelet-based histograms offer more accuracy than all traditional partition-based histograms, their maintenance is much more difficult because they are built through a nontrivial mathematical transformation procedure. For equi-depth histograms, any change of the data distribution can be easily recorded by updating the summary frequency for the corresponding bucket. The job of the maintenance is to balance the size of the buckets over time through simple operations like splitting and merging [GMP97, AC99]. On the other hand, for wavelet-based histograms, when there is an update in the underlying data distribution, several wavelet coefficients change their values. A significant coefficient could become insignificant over time, and vice versa. Since the histogram should maintain the most significant coefficients in order to ensure accuracy in a robust manner, we must keep track of the most significant coefficient set, and this is a non-trivial job.

The DCT-based histogram method discussed by Lee et al. [LKC99] faces exactly the same problem. The approach in [LKC99] is to maintain a static set of coefficients and update their values in response to the data updates. The same method can be applied to wavelet-based histograms. It is claimed in [LKC99] that any data change will be reflected immediately in the histogram. However, such a claim is true only if the updates follow the same or very similar distribution as the base data. Their method maintains a fixed set of DCT coefficients and only keeps track of the value changes of those coefficients. It never considers the fact that the set of significant coefficients should change over time, and as we shall see in Section 6, this type of dynamic maintenance of a static set of coefficients can introduce very big errors in estimation.

The method of Lee et al. [LKC99] has another fundamental problem. The set of DCT coefficients, which is statically chosen, is not even chosen with respect to the initial set of data, but rather is chosen *a priori* independently of the data by means of static geometrical zonal sampling. The coefficients chosen using the predefined geometrical zonal sampling method may not be significant coefficients, thus resulting in bad histograms.

3 Preliminaries and Problem Formulation

3.1 Wavelet Decomposition

Wavelets are a mathematical tool for the hierarchical decomposition of functions in a space-efficient manner. Wavelets represent a function in terms of a coarse overall shape, plus details that range from coarse to fine. Regardless of whether the function of interest is an image, a curve, or a surface, wavelets offer an elegant technique for representing the various levels of detail of the function in a space-efficient manner.

For readers who are not familiar with wavelets, we borrow a simple example from [VW99] to illustrate the wavelet decomposition procedure. To start the wavelet decomposition procedure, we first need to choose the wavelet basis functions. Haar wavelets are conceptually the simplest wavelet basis functions, and for purposes of exposition in this paper, we focus our discussion on Haar wavelets. They are fastest to compute and easiest to implement. To illustrate how Haar wavelets work, we start with a simple example which will be used throughout the paper. (A detailed treatment of wavelets can be found in any standard reference on the subject, e.g., [JS94, SDS96].) Suppose we have a one-dimensional "signal" of $N = 8$ data items:

$$S = [2,\ 2,\ 0,\ 2,\ 3,\ 5,\ 4,\ 4].$$

We perform a wavelet transform on it. We first average the signal values, pairwise, to get the new lower-resolution signal with values

$$[2,\ 1,\ 4,\ 4].$$

That is, the first two values in the original signal (2 and 2) average to 2, and the second two values 0 and 2 average to 1, and so on. Clearly, some information is lost in this averaging process. To recover the original signal from the four averaged values, we need to store some *detail coefficients*, which capture the missing information. Haar wavelets store the pairwise differences of the original values (divided by 2) as detail coefficients. In the above example, the four detail coefficients are $(2-2)/2 = 0$, $(0-2)/2 = -1$, $(3-5)/2 = -1$, and $(4-4)/2 = 0$. It is easy to see that the original values can be recovered from the averages and differences.

We have succeeded in decomposing the original signal into a lower-resolution version of half the number of entries and a corresponding set of detail coefficients. By repeating this process recursively on the averages, we get the full decomposition:

Resolution	Averages	Detail Coefficients
8	[2, 2, 0, 2, 3, 5, 4, 4]	
4	[2, 1, 4, 4]	[0, −1, −1, 0]
2	$[1\frac{1}{2},\ 4]$	$[\frac{1}{2},\ 0]$
1	$[2\frac{3}{4}]$	$[-1\frac{1}{4}]$

We define the *wavelet transform* (also called *wavelet decomposition*) of the original eight-value signal to be the single coefficient representing the overall average of the original signal, followed by the detail coefficients in the order of increasing resolution. Thus, for the one-dimensional Haar basis, the wavelet transform of our original signal is given by

$$\widehat{S} = [2\tfrac{3}{4},\ -1\tfrac{1}{4},\ \tfrac{1}{2},\ 0,\ 0,\ -1,\ -1,\ 0]. \tag{1}$$

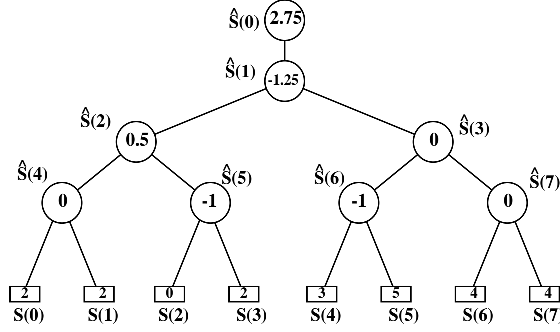


Figure 1: Error tree for $N = 8$

The individual entries are called the *wavelet coefficients*. The wavelet decomposition is very efficient computationally, requiring only $O(N)$ CPU time and $O(N/B)$ I/Os to compute for a signal of N values.

No information has been gained or lost by this process. The original signal has eight values, and so does the transform. Given the transform, we can reconstruct the exact signal by recursively adding and subtracting the detail coefficients from the next-lower resolution.

For compression reasons, the detail coefficients at each level of the recursion are often normalized; the coefficients at the lower resolutions are weighted more heavily than the coefficients at the higher resolutions. One advantage of the normalized wavelet transform is that in many cases a large number of the detail coefficients turn out to be very small in magnitude. Truncating these small coefficients from the representation (i.e., replacing each one by 0) introduces only small errors in the reconstructed signal. We can approximate the original signal effectively by keeping only the most significant (normalized) coefficients.

The wavelet decomposition procedure can be represented by an *error tree* [MVW98]. The error tree is built based upon the wavelet transform procedure. Figure 1 is the error tree for the example in this section. Each internal node is associated with a wavelet coefficient value, and each leaf is associated with an original signal value. (For purposes of exposition, the wavelet coefficients are unnormalized, but in the implementation the values are normalized and the algorithm is modified appropriately.) Internal nodes and leaves are labeled separately. Their labels are in the domain $\{0, 1, \ldots, N-1\}$ for a signal of length N. For example, the root is an internal node with label 0 and its node value is 2.75 in Figure 1. For convenience, we shall use "node" and "node value" interchangeably.

The construction of the error tree exactly mirrors the wavelet transform procedure. It is a bottom-up process. First, leaves are assigned original signal values from left to right. Then wavelet

coefficients are computed, level by level, and assigned to internal nodes.

As the figure shows, the unnormalized value of each internal node j is denoted by $\widehat{S}(j)$, and the value of each leaf i is denoted by $S(i)$.

3.2 Problem Formulation

In [MVW98], the wavelet-based histogram is built by first performing wavelet decomposition on the *extended cumulative data distribution* (partial sums) of an attribute, resulting in a sequence of wavelet coefficients. Then the top m coefficients with the largest absolute values are chosen to compose the histogram. (The parameter m depends upon the available storage space for the histogram.) A similar scenario arises in the other transform-based methods, such as those based upon the discrete cosine transform (DCT).

For dynamic maintenance of the wavelet-based histogram, do we have to consider partial-sum based wavelet decomposition, or should we just do the decomposition using the raw data distribution? To answer this question, we need to resolve two issues. First, what would be the accuracy difference between using partial sums and raw data? If the partial-sum-based method gives much higher accuracy for selectivity estimation than the raw-data-based method, we'd better do dynamic maintenance for the partial-sum-based method. The first issue raises the second, that is, intuitively, it would be much harder to do maintenance for the partial-sum-based method because local data changes will propagate globally.

To resolve the first issue, we did an extensive experimental study to compare the accuracy of the two methods using a large collection of Zipf data sets [Zip49]. It turns out that the accuracy of the two methods varies with data distribution, query type, and the choice of different error measurements. We cannot say that one method is alway better than another. In general, the two methods are comparable in accuracy when considering the overall performance. (A similar observation is made in [VW99] for high-dimensional sparse data cube approximation.) The experimental result dismisses the first concern, and consequently, we do not have to consider the second issue. From now on, we shall focus on the dynamic maintenance of histograms based upon raw data distributions.

We formalize the dynamic maintenance problem using the error tree structure for one-dimensional case, as follows. Let T_0 be the initial error tree constructed from the initial data distribution S_{t_0} of an attribute at time t_0. The wavelet-based histogram H_0 is a subset of internal nodes of T_0. Each element of H_0 is of the form $\big(j, \widehat{S}(j)\big)$. Specifically, for a histogram of size m, H_0 contains the

top m internal nodes of T_0 (ranking by the absolute values of the normalized coefficients).

Suppose during a time interval from t_0 to t_1, the data distribution changes from S_{t_0} to S_{t_1}. Based upon S_{t_1} we can construct a new error tree T_1, and consequently we obtain a new histogram H_1. Our *dynamic maintenance problem* is the following: *At any time t_1, how can we efficiently maintain a histogram $\widetilde{H}_1$ that is a good approximation of H_1?*

There are three possible changes in the distribution of an attribute: insertion, deletion, and update. Consider a period from t_0 to t_1. During this time interval, a sequence of insertions, deletions, and updates occur at the leaves, and we can represent them by a list:

$$\Delta = \{\Delta(0), \ldots, \Delta(N-1)\}, \tag{2}$$

where $\Delta(i)$ is the value change at leaf i. The Δ sequence is mainly introduced for simplifying our discussions. We only need to store the non-zero entries in the sequence. (The zero entries correspond to the leaf values that remain unchanged from time t_0 to time t_1.)

4 Dynamic Maintenance of One-Dimensional Wavelet-Based Histograms

In this section, we describe the general ideas of doing dynamic maintenance for wavelet-based histograms and outline our method for one-dimensional case.

4.1 Properties of Wavelet-Based Histograms

A value change at a leaf will affect all its ancestors in the error tree. The problem is how those ancestors are affected. It turns out that the error tree has some nice properties and we can use them to figure out the exact changes occurred at internal nodes upon any leaf change.

Inserting a value i is equivalent to increasing $S(i)$ by one. Similarly, deleting a value i is equivalent to decreasing $S(i)$ by one. An update can be considered as one deletion followed by one insertion. (For simplicity, we restrict our attention in this paper to individual value changes of $+1$ and -1, but the method can be extended naturally to general updates.)

Consider an internal node j that is the parent of a leaf node i. At time t_0, leaf i's value is $S_{t_0}(i)$. At time t_1, its value changes to $S_{t_1}(i)$. Letting

$$\Delta(i) = S_{t_1}(i) - S_{t_0}(i), \tag{3}$$

we have

$$\widehat{S}_{t_1}(j) = \begin{cases} \widehat{S}_{t_0}(j) + \dfrac{\Delta(i)}{2} & \text{if leaf } i \text{ is the left child of node } j; \\ \widehat{S}_{t_0}(j) - \dfrac{\Delta(i)}{2} & \text{if leaf } i \text{ is the right child of node } j. \end{cases} \tag{4}$$

From (4) we can see that from time t_0 to time t_1, the magnitude of $\widehat{S}(j)$ may increase or decrease, depending upon $\widehat{S}_{t_0}(j)$, $\Delta(i)$, and the relative position of j with respect to i (i.e., if i is the left or right child of node j.)

In the above example, node j's subtree only contains two leaves, and we are only considering single leaf value change. Now we look at the more general cases. For any leaf i, we use $path(i)$ to denote the set of the internal nodes along the path from i to the root. We use $left(j)$ and $right(j)$ to denote the left and right child of any node j, and we use $leaves(j)$ to denote the set of leaves in the subtree rooted at j. For any internal node j, we define its *height* as

$$height(j) = \begin{cases} \log N - \lfloor \log j \rfloor & \text{for } 1 \leq j < N; \\ \log N & \text{for } j = 0. \end{cases} \tag{5}$$

(By convention, we say that node 0 and 1 have height $\log N$, which is the only difference with the picture in Figure 1.)

Lemma 1 *For any leaf i, we consider the effects of the its value change (3). For each $j \in path(i)$, we have*

$$\widehat{S}_{t_1}(j) = \begin{cases} \widehat{S}_{t_0}(j) + \dfrac{\Delta(i)}{2^{height(j)}} & \text{if } i \in leaves\big(left(j)\big); \\ \widehat{S}_{t_0}(j) - \dfrac{\Delta(i)}{2^{height(j)}} & \text{if } i \in leaves\big(right(j)\big). \end{cases} \tag{6}$$

From Lemma 1 we obtain the following

Theorem 1 *From a single leaf value change we can compute the value change of each of its ancestors in constant time using a closed-form formula.*

We can generalize Lemma 1 to the case when there are multiple leaf value changes:

Lemma 2 *For any internal node j, consider each leaf node i in $leaves(j)$. During the time interval $[t_0, t_1]$, the value change of i is $\Delta(i)$. We use $ave_val_change(j)$ to denote the average of value changes for all leaves in $leaves(j)$. We have*

$$\widehat{S}_{t_1}(j) = \widehat{S}_{t_0}(j) + \frac{ave_val_change\big(left(j)\big) - ave_val_change\big(right(j)\big)}{2}. \tag{7}$$

Consider the Δ sequence (2). It represents all the leaf value changes from time t_0 to t_1, and it contains N_z non-zero elements. Conceptually, using all the entries in (2) as the leaf values, we may construct a new error tree, and we call it the Δ error tree, denoted by T_Δ. The tree T_Δ has the same structure as the error tree T_0, and it contains at most $\min\{N_z \log N, N\}$ non-zero internal nodes. All the non-zero leaves of T_Δ together with their ancestors compose a partial error tree. We can do a node-wise tree addition between T_0 and T_Δ, that is, we add T_0 and T_Δ, as follows: We construct a new tree with exactly the same structure as T_0. For each node j, its value in the new tree is the sum of the corresponding node values in T_0 and T_Δ. We denote this tree addition by $T_0 + T_\Delta$.

Lemma 3 (Additive property) *If we construct an error tree T_Δ from array Δ (we call T_Δ the Δ error tree), then we can add the values of the corresponding nodes in T_0 and T_Δ to obtain T_1.*

The above lemma summarizes the changes in (2) in a batch. Equivalently, starting from the initial error tree T_0, we can just apply each leaf node change directly and incrementally using (6) to obtain T_1. Alternately, we can also obtain T_1 by updating the internal nodes of T_0 using (7).

4.2 The Difficulties in Dynamic Maintenance

From the discussions in Section 4.1, we can see some difficulties in maintaining wavelet-based histograms. To keep track of the top m coefficients exactly, we must keep all the internal nodes of the error tree, and we must keep their values up to date all the time. Suppose we have kept the top m coefficients at time t_0 in H_0. For any single insertion after t_0, there are $1 + \log N$ ancestor coefficients in the error tree that will be affected. We can easily use (6) to update those ancestor coefficients that are among H_0. The hard part is how to deal with the coefficients that are *not* in H_0. The only thing we know about those coefficients is that they were not significant enough to be include in H_0 at time t_0. But when the data distribution changes (i.e., after the insertion), they may become significant. Although we can use (6) to compute the *value changes* at those non-H_0 nodes, we still do not know what to do with them because we have no idea about their original values.

A naive way to solve the problem is to recompute the wavelet decomposition from scratch based upon the new data distribution and to choose the top m coefficients whenever the data distribution changes (i.e., when any leaf value changes in the error tree). This approach is not feasible in practice since it invokes too much overhead at query optimization time, even though it maintains the quality of the histogram.

Another approach is to trade accuracy for efficiency. We could build the histogram H_0 at time t_0 and thereafter just keep the values of the coefficients in H_0 up to date using (6). This approach would work fine if the statistics of the underlying data distribution do not change or just have minor change. This obvious approach is the one used in [LKC99], although the approach in [LKC99] is less sophisticated in its initial choice of coefficients, in that they are chosen independently of the data. The problem is that there is no guarantee that the histogram will still contain the most significant coefficients after the underlying data distribution has changed even modestly. As we shall see in Section 6, histograms maintained using this simple approach often incur very big errors in estimations.

Our goal in this paper is to maintain the histograms efficiently without sacrificing accuracy. We want the histograms to be robust to changing distributions, yet roughly as accurate as a fixed histogram for the case in which the update data follow the same or similar distribution as the base data.

4.3 Our Method

Let the initial error tree at time t_0 be T. We consider the top $m + m'$ internal nodes of T, where $m + m' \ll N$. From those $m + m'$ nodes, we choose the top m to form the histogram H. The other m' nodes are kept as an *auxiliary histogram* H' on disk. We use an *activity log* L to log the insert, delete, and update activities; the log has a maximal size of *Max_Log_Size*.

We shall focus on insertions because deletions are similar and update is one insertion followed by one deletion (or vice versa). We consider a sequence of insertions. When an insertion happens to a leaf node i, according to Lemma 1, the values of all the internal nodes in $path(i)$ change. For any internal node j in $path(i)$, we characterize it into one of three types: (a) $j \in H$, (b) $j \in H'$, and (c) $j \notin H \cup H'$. We will handle different types of nodes differently.

First, we write an entry i into the log L. Then we consider all the internal nodes $j \in path(i)$. If j is a type (a) node, we update its value immediately according to (6). (We have $\Delta(i) = 1$ for an insertion and $\Delta(i) = -1$ for a deletion.) If j is not a type (a) node, we do nothing.

When the number of entries in log L reaches *Max_Log_Size*, we process the entries in the log. For any entry i in L, we update all the corresponding type (b) nodes according to (6). For a type (c) node j, we flip a coin with probability $p(j)$ of heads. If the coin flips a head, we set node j's magnitude $\widehat{S}(j)$ to be $v(j)$ (a value to be determined later), and we replace the smallest node (in magnitude) in H' with node j.

When all the entries in the log L have been processed, we adjust H and H'. Whenever the magnitude of the largest node in H' exceeds a threshold value H'*_Thresh* (to be determined later), we will switch the smallest node in H with it.

Now we are done with the current log L and we can start to process new insertions and start over to form a new fresh log L. At any given time, H is the histogram that is used for selectivity estimation.

The logic behind our method is as follows:

- By updating the coefficients in H promptly, our histogram H will always keep the coefficient values up to date.
- By keeping and updating the coefficients in H' in a batched fashion, we have reasonable amount of extra information on the possible candidates that are likely to get into H later.
- By using *probabilistic counting* technique for all type (c) nodes, we can detect any "surprising" candidates for H that may become significant later even though they do not look significant at all initially.

Now let us consider the parameters *Max_Log_Size*, H'*_Thresh*, $p(j)$, and $v(j)$. The parameter *Max_Log_Size* specifies how often we want to check into the situation where we need to do a shake up to include some new nodes into H and exclude some old nodes from H when the data distribution changes. The smaller the *Max_Log_Size* value is, the more often the shake-up will occur, and the more chance H will get to update. The frequent updating is good for accuracy, but frequent log processing causes slow performance. Besides, it is unlikely that a node that was not in H would become significant after a very small number of insertions. In our experiments, we choose *Max_Log_Size* to be 1%–5% of the base data size.

The parameter H'*_Thresh* specifies how aggressive we are in adjusting H over time. Denote the magnitude of the minimum coefficient in H by $\min(H)$. A reasonable setting would be H'*_Thresh* $= \min(H)$. On the other hand, if the magnitudes of two coefficients are very close, it does not really matter which one is in H, because both would be (approximately) equally important. In our experiments, we set H'*_Thresh* $= c_1 \times \min(H)$, where c_1 is a constant (typically in the range $[1.0, 3.0]$).

Now let us consider how we should set $p(j)$ and $v(j)$. Intuitively, we want $1/p(j)$ to correspond to the number of insertions needed at leaf i to bring the magnitude of node j from its initial value of zero to $v(j)$, so we need to set $v(j)$ first. The parameter $v(j)$ is similar to the parameter H'*_Thresh*, and we can set $v(j) = c_2 \times \min(H)$, where c_2 is a constant, typically in the range $[0.2, 0.8]$. The

value $v(j)$ is independent of j and we denote $v = c_2 \times \min(H)$. An alternative that also works well is to define $v = c_1 \times \min(H')$. We can easily derive

$$p(j) = \frac{1}{v \times 2^{height(j)}}$$

according to (6). The exact value of node j depends upon its position. When the coin flips a head for node j, the value should be set to v if leaf i is in the left subtree of j and $-v$ otherwise.

The following pseudo code summarizes our algorithm:

```
Procedure Dynamic_Maintenance(H, H', L, m, m', Max_Log_Size, c1, c2)
  Compute wavelet decomposition of the base data distribution;
  Compose H by choosing the top m wavelet coefficients;
  Compose H' by choosing the (m+1)st to (m+m')th coefficients;
  while (insert value i to the relation)
    write i to log L;
    log_size++;
    path(i) = { j | j is an internal node on the path from leaf i to the root};
    for each j ∈ path(i) ∩ H
      update Ŝ(j) using (6);
    if (log_size == Max_Log_Size)
      Process_Log(L, H', H, log_size, c1, c2);
      log_size = 0;

Procedure Process_Log(L, H', H, log_size, c1, c2)
  for k = 1 to log_size do
    i = kth entry in L;
    v = c2 × min_coeff_magnitude(H);
    path(i) = {j | j is an internal node on the path from leave i to the root};
    for each j ∈ path(i) ∩ H'
      update Ŝ(j) using (6);
    for each j ∈ (path(i) − (H ∩ H'))
      p = 1 / (v × 2^height(j));
      flip a coin with probability p of head;
      if (coin flips a head)
        if (i ∈ leaves(left(j)))
          Ŝ(j) = v;
        else
          Ŝ(j) = −v;
        replace the minimum coefficient in H' with (j, Ŝ(j));
  while (max_coeff_magnitude(H') > c1 × min_coeff_magnitude(H))
    switch the maximum coefficient in H' with the minimum coefficient in H;
```

Function *min_coeff_magnitude*(X) returns the magnitude of the smallest coefficient in coefficient set X. Function *max_coeff_magnitude*(X) returns the magnitude of the largest coefficient in coefficient set X.

A sequence of deletions can be handled in a similar way. For a mixed sequence of deletions and insertions, we may maintain two separate logs, one for insertions and another for deletions. When the total size of these two logs reaches *Max_Log_Size*, we start to process them, one by one. Or, we just maintain one log for both insertions and deletions. Later, when we need to use the log information to update H and H', we can either split it into two parts, or we just process the single log and handle insertion and deletion differently in our algorithm.

The activity log L is a simplified version of the database transaction log that is used widely in commercial DBMSs. In the above algorithm, we write the inserted entries to the log directly. In our implementation, we use a double buffer in memory to keep the inserted entries instead of writing each entry to the log directly. When the double buffer becomes half full, we write all the entries in the buffer to the log on disk. The number of disk I/O is hence reduced dramatically. (The double-buffer mechanism is indeed used in the implementation of the real database transaction log.) To further facilitate the functions *min_coeff_magnitude*(X) and *max_coeff_magnitude*(X), we maintain H and H' as priority queues throughout the process.

The only storage overhead invoked by our method is the auxiliary histogram H', which is stored on disk. The auxiliary histogram H' is not part of the true histogram used by the optimizer, and it is needed only when we do batch processing of the activity log. Therefore, we can store H' together with the activity log. As we show in Section 6, the overhead can be kept low while achieving very good histogram quality. The time overhead of our method composes two parts: the time for changing the coefficient values for the coefficients in the present histogram and the time for processing the log. The first part is fixed since we can compute each change in constant time using the properties of the error tree structure. Use of random sampling or batch processing of the updates, as discussed in the previous paragraph, would speed up processing even further, at a slight cost in accuracy. The second part, although taking longer time than the first part, is only invoked in batch mode after a significant number of updates and can be tuned by the user.

One way to improve the time performance of our algorithm is to preprocess the log whenever it is full before we invoke the procedure *Process_Log*. We combine multiple insertions of the same value into a single entries of the form (*value*, *number_of_insertions*). The preprocessing can speed up *Process_Log* significantly when the number of combined entries are much smaller than the original log size, which is likely to happen when *Max_Log_Size* is set to a big value.

Another speed improvement is to randomly sample the events and process only a random sample. The updates done on the leaves should be adjusted normalized appropriately. Yet another speed improvement is to process the events, whether sampled or not, in batch mode, much like the

entries in the log, but at more frequent intervals.

Our method is very stable for various update distributions. The static method in [LKC99] performs reasonably well only when the distribution of the update sequence is *very close* to that of the original base data. Otherwise, the accuracy of the maintained histogram degrades dramatically since the histogram does not contain the set of most significant coefficients anymore.

5 Maintenance of Multidimensional Histograms

The one-dimensional wavelet decomposition and reconstruction procedure in Section 3.1 can be extended naturally to the multidimensional case. One way to do a multidimensional wavelet decomposition is by a series of one-dimensional decompositions. For example, in the two-dimensional case, we first apply the one-dimensional wavelet transform to each row of the data. Next, we treat these transformed rows as if they were themselves the original data, and we apply the one-dimensional transform to each column. So we can easily build multidimensional wavelet-based histograms.

Our method for maintaining one-dimensional wavelet-based histograms can also be extended naturally to multidimensional case. Suppose we want to build a d-dimensional wavelet-based histogram for d attributes. Let $D = \{D_1, D_2, \ldots, D_d\}$ denote the set of attributes. We represent the joint distribution of these attributes by a d-dimensional array S of size $|D_1| \times |D_2| \times \cdots \times |D_d|$, where $|D_i|$ is the size of the domain for D_i. Without loss of generality, we assume that each attribute D_i has domain $\{0, 1, \ldots, |D_i|-1\}$. An array element $S(i_1, i_2, \ldots, i_d)$ is the frequency of the corresponding value combination $(i_1, i_2, \ldots, i_d)$ of the attributes. The wavelet coefficients obtained by performing a d-dimensional wavelet decomposition on S can be represented by a d-dimensional array $\widehat{S}$ of size $|D_1| \times |D_2| \times \cdots \times |D_d|$.

We first extend Lemma 1 to general d-dimensional case.

Lemma 4 *For any array element $S(i_1, i_2, \ldots, i_d)$, we consider the effects of its value change*

$$\Delta(i_1, i_2, \ldots, i_d) = S_{t_1}(i_1, i_2, \ldots, i_d) - S_{t_0}(i_1, i_2, \ldots, i_d). \tag{8}$$

For each $\widehat{S}(j_1, j_2, \ldots, j_d)$ where $j_k \in path(i_k)$ in the error tree of D_k, for $1 \leq k \leq d$, we have

$$\widehat{S}_{t_1}(j_1, j_2, \ldots, j_d) = \widehat{S}_{t_0}(j_1, j_2, \ldots, j_d) + \frac{\Delta(i_1, i_2, \ldots, i_d)}{\prod_{k=1}^{d} sign(i_k, j_k) 2^{height(j_k)}}, \tag{9}$$

where

$$sign(i_k, j_k) = \begin{cases} 1 & \textit{if } i_k \in leaves\big(left(j_k)\big) \textit{ in the error tree of } D_k; \\ -1 & \textit{if } i \in leaves\big(right(j_k)\big) \textit{ in the error tree of } D_k. \end{cases} \quad (10)$$

The following algorithm that extends our algorithm in Section 4 to d-dimensional case follows naturally.

Procedure *Multidimensional_Dynamic_Maitenance*$(H, H', L, m, m', Max_Log_Size, c_1, c_2, d)$
  Compute the wavelet decomposition of the joint data distribution of the base data;
  Compose H by choosing the top m wavelet coefficients;
  Compose H' by choosing the $(m+1)$st to $(m+m')$th coefficients;
  $log_size = 0$;
  while (insert value $(i_1, i_2, \ldots, i_d)$ to the relation)
    write $(i_1, i_2, \ldots, i_d)$ to log L;
    log_size++;
    for $k = 1$ **to** d **do**
      $path(i_k)$=$\{j_k \mid j_k$ is an internal node on the path from i_k to the root in the error tree of $D_k\}$;
    for each $(j_1, j_2, \ldots, j_d) \in \big(path(i_1) \times path(i_2) \times \cdots \times path(i_d)\big) \cap H$
      update $\widehat{S}(j_1, j_2, \ldots, j_d)$ using (9);
    if $(log_size == Max_Log_Size)$
      Multidimensional_Process_Log$(L, H', H, log_size, c_1, c_2, d)$;
      $log_size = 0$;

Procedure *Multidimensional_Process_Log*$(L, H', H, log_size, c_1, c_2, d)$
  for $k = 1$ **to** log_size **do**
    $(i_1, i_2, \ldots, i_d) = k$th entry in B;
    $v = c_2 \times min_coeff_magnitude(H)$;
    for $k = 1$ **to** d **do**
      $path(i_k)$=$\{j_k \mid j_k$ is an internal node on the path from i_k to the root in the error tree of $D_k\}$;
    for each $(j_1, j_2 \ldots j_d) \in \big(path(i_1) \times path(i_2) \times \cdots \times path(i_d)\big) \cap H'$
      update $\widehat{S}(j_1, j_2, \ldots, j_d)$ using (9);
    for each $(j_1, j_2, \ldots, j_d) \in \big(path(i_1) \times path(i_2) \times \cdots \times path(i_d)\big) - (H \cup H')$
      $p = 1/v \prod_{k=1}^{d} 2^{height(j_k)}$;
      flip a coin with probability p of head;
      if (coin flips a head)
        $\widehat{S}(j_1, j_2, \ldots, j_d) = v \times \prod_{l=1}^{d} sign(i_l, j_l)$;
        replace the minimum coefficient in H' with $\big(j_1, j_2, \ldots, j_d, \widehat{S}(j_1, j_2, \ldots, j_d)\big)$;
  while $\big(max_coeff_magnitude(H') > c_1 \times min_coeff_magnitude(H)\big)$;
    switch the maximum coefficient in H' with the minimum coefficient in H;

6 Experimental Results

6.1 Methods of Comparisons

We implement the following three methods for dynamic maintenance of wavelet-based histograms and we compare their performance.

Exact Method. This *expensive* method corresponds to *recomputing* the histogram from scratch whenever *any* update happens to the data distribution.

Probabilistic Counting Method. This is the method we introduced in Sections 4 and 5.

Static Method. At initial time t_0, we compute the exact histogram H_0 for the base data distribution. From then on, the content (the coefficients) of H_0 no longer changes. We only change the values of those coefficients to make them up to date.

At any given time, the histogram maintained using the exact method contains exactly the set of most significant coefficients for the data distribution at that time. Thus, this method usually provides the best accuracy in selectivity estimations.

The static method is similar to the method proposed in [LKC99], except that we start with the "right" set of coefficients, chosen in a way corresponding to the distribution of the base data. Its implementation is very simple: After building the initial histogram H_0 based upon the base data distribution at time t_0, in the event of any frequency change for value i, we just need to update the relevant coefficient values for the coefficients that are in $path(i) \cap H_0$ according to (6). For the multidimensional case, when the frequency changes at $(i_1, i_2, \ldots, i_d)$, we use (9) to update the coefficient values for the coefficients that are in $\big(path(i_1) \times path(i_2) \times \cdots \times path(i_d)\big) \cap H_0$.

6.2 Error Measures

To define the proper measures for the accuracy of various methods, we first look into the logic in choosing the top m coefficients to form the exact wavelet-based histograms.

With proper normalization (which we always do), the Haar basis is orthonormal. For any orthonormal wavelet basis, choosing the m largest (in absolute value) wavelet coefficients is provably optimal in minimizing the 2-norm of the absolute error *when considering the reconstruction of the original signal values* [SDS96]. Note that the 2-norm of the absolute error for the reconstructed data values corresponds to the 2-norm of the absolute error for all the equal queries. Thus, a useful measurement to evaluate the accuracy of a dynamic maintenance method is the 2-norm average

absolute error for all the equal queries using the maintained histogram after a sequence of updates. By using this measure, we can clearly see how "close" a histogram maintained using a certain method is to that maintained using the (expensive) optimal method.

In our experiments, we also use other error measures proposed in [VWI98] and we report the results for one-dimensional case. For the multidimensional case, we only report the results using the 2-norm absolute error for all equal queries. For all the experiments in this section, the errors reported for our probabilistic counting method are the averages over multiple different runs.

6.3 Data Distributions

The data we used in our experiments are similar to those of [GMP97]. For one-dimensional data, we model the base data originally in the database and the update data sequence using an extensive set of Zipf distributions. The number of distinct values varies from 128 to 1024. Without loss of generality, we use the integer value domain. The z value for the frequency distribution is chosen from 0.0 to 3.0 to vary the skew. The frequencies are mapped to the values according to three different types of correlations: *positive* (the bigger the value, the higher the frequency), *negative* (the bigger the value, the lower the frequency), and *random*. We refer to a Zipf distribution with parameter z and correlation X as the *Zipf* (z, X) distribution.

We extend Zipf data distributions to the multidimensional case. To generate a multidimensional data set, we first generate the frequencies for a one-dimensional data set of appropriate size using the Zipf distribution. We then logically map the one-dimensional values to the multidimensional values according to certain dimension order. For example, we can use the row-major ordering to do the mapping between two-dimensional values and one-dimensional values.

When we model a sequence of insertions, we generate two data sets, one to represent the base data and another to represent the insertion sequence. In our experiments, the size of the insertion sequence varies from the base data size to four times the base data size.

For deletions, we first generate two data sets A and B, as for the insertion case, and we combine A and B to obtain the base data set $A \cup B$. Then we use the second data set B as our deletion sequence. In our experiments, the size of the deletion sequence varies from one quarter of the base data size to three quarters of the base data size.

6.4 Parameter Settings

Parameters c_1 and c_2 are two very important parameters for our algorithm. They define how aggressive we are in shaking up the coefficients in the maintained histogram. The smaller c_1 and c_2 are, the more aggressive we are in shaking up the histogram. Intuitively, it helps to know the relationship between the distribution of the base data and that of the update sequence in order to choose the proper c_1 and c_2 values. For example, if the two distributions are similar, we should set c_1 and c_2 to bigger values in order to save time.

In our experiments, we find that our algorithm performs very well and is stable for a variety of update distributions as long as we choose c_1 and c_2 from reasonable ranges, and we do not need any *a priori* knowledge on the distribution of the update sequence to guarantee good performance of our algorithm. The appropriate ranges for c_1 and c_2 are 1.0–3.0 and 0.2–0.8, respectively. In our experiments, we use the default values $c_1 = 1.5$ and $c_2 = 0.5$.

The other important parameter is m', the size of the auxiliary histogram. It is obvious that a bigger m' value will give better accuracy but slower performance since we have more extra information in adjusting the histogram. However, our experiments show that it is not necessary to keep a very big auxiliary histogram to achieve good accuracy. For example, in most cases, setting $m' = m/2$ or $m' = 2m$ will not change the accuracy significantly. We use $m' = m/2$ as the default value in our experiments.

The parameter *Max_Log_Size* is relevant to the size of the base data. In our experiments, we set *Max_Log_Size* to be between 1% and 5% of the base data size. The default value for *Max_Log_Size* is 1% of the base data size.

6.5 Accuracy for Maintaining One-Dimensional Histograms

We experimentally study the accuracy of various methods for a wide range of base data and update data distributions. We use $m = 20$ as the size of our wavelet-based histogram for all the experiments. The size of the value set is 512.

In the first set of experiments, we compare the accuracy of the three methods for the cases when the distribution of the update sequence is the same or very similar to that of the base data. For example, the base data and the update sequence follow the $\mathit{Zipf}(1.0, \mathit{positive})$ and $\mathit{Zipf}(1.2, \mathit{positive})$ distributions, respectively. In this case, the set of significant coefficients will either not change or else have some minor changes. As long as we keep the values up to date for the coefficients in the original histogram H_0 (which were built upon the base data), the histogram maintained using the static

Error Norm	*Exact*	*Probabilistic Counting*	*Static*
$\|e^{\text{abs}}\|_1$	52.13	52.42	52.13
$\|e^{\text{abs}}\|_2$	88.70	88.80	88.70
$\|e^{\text{rel}}\|_1$	0.17	0.17	0.17
$\|e^{\text{comb}}\|_1,\ \ \alpha = 1,\, \beta = 100$	16.71	16.72	16.71

Table 1: Average errors of various methods for all equal queries.

method is almost the same as the exact histogram. On the other hand, our probabilistic counting method could give less accurate results since it might unnecessarily shake up the coefficients in the histogram because of the probabilistic nature of the method.

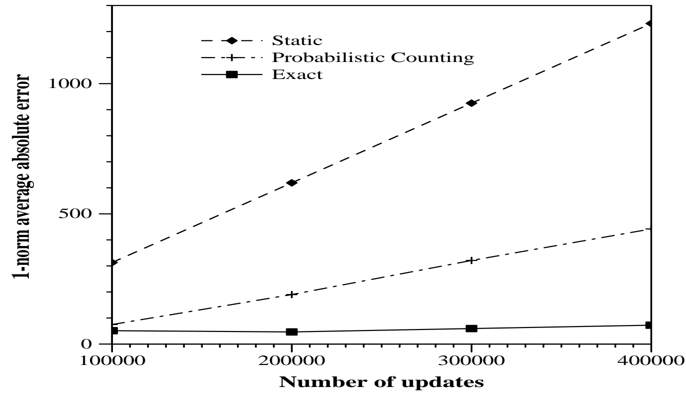
Our results show that our method gives almost the same accuracy as the exact method even when the distribution of the update sequence is the same as that of the base data. Tables 1–2 show the accuracy of the three methods for one typical case. The base data contain 100K tuples from the *Zipf*(1.0, *negative*) distribution; the update data come from the *Zipf*(1.2, *negative*) distribution. We measure the errors for different types of queries using the error measures defined in [MVW98] after 400K insertions. (Note that the number of insertions is four times the base data size.)

Error Norm	*Exact*	*Probabilistic Counting*	*Static*
$\|e^{\text{abs}}\|_1$	1679	1696	1679
$\|e^{\text{abs}}\|_2$	1982	1991	1982
$\|e^{\text{rel}}\|_1$	0.0036	0.0036	0.0036
$\|e^{\text{comb}}\|_1,\ \ \alpha = 1,\, \beta = 100$	0.36	0.36	0.36

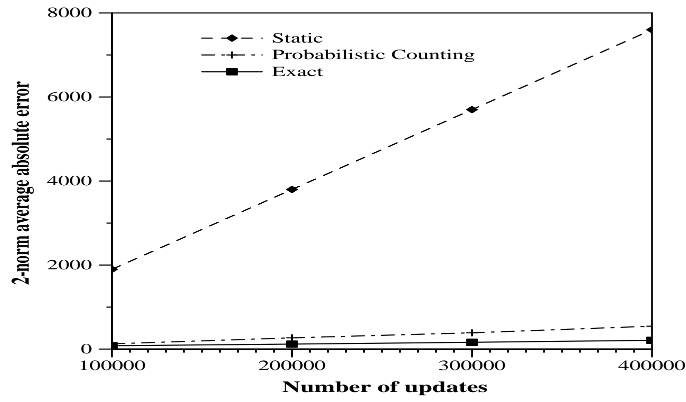
Table 2: Average errors of various methods for all one-side range queries.

Now we look into the more interesting cases where the distribution of the update sequence are different from or unrelated to that of the base data. Figure 2 plots the accuracy of different methods for a typical case. In this case, the base data contain 100K tuples and follow the *Zipf*(1.0, *negative*) distribution. The insertions follow the *Zipf*(1.5, *random*) distribution. We measure the accuracy of the different methods when the insertion sequence contains 100K, 200K, 300K, and 400K entries. The results show that the probabilistic counting method is much better than the static method.

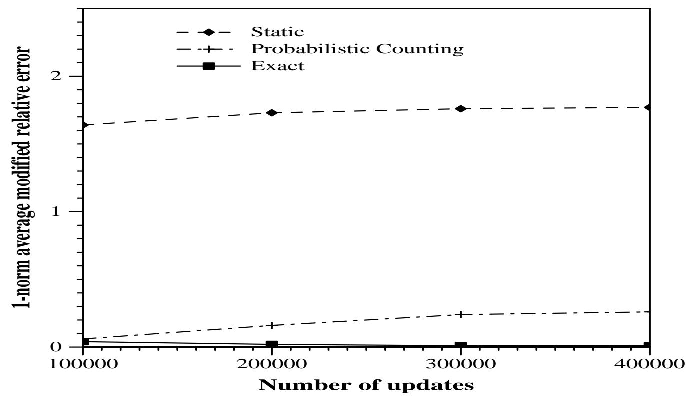
We also compare the methods for deletions. Tables 3–4 show a typical case. In this case, we generate the base data distribution by combining two data sets; one follows *Zipf*(1.0, *negative*) and the other follows *Zipf*(1.5, *positive*). Each data set contains 100K entries. The deletion sequence follows the same data distribution as the second data set, i.e., *Zipf*(1.5, *positive*). Tables 3–4 show the accuracy of the three methods when 100K deletions have been performed.



(a) 1-norm average absolute error for all equal queries



(b) 2-norm average absolute error for all equal queries



(c) 1-norm average modified relative error for all one-side range queries

Figure 2: Accuracy of various methods for insertions.

Error Norm	*Exact*	*Probabilistic Counting*	*Static*
$\|e^{\text{abs}}\|_1$	51.3	55.8	313.9
$\|e^{\text{abs}}\|_2$	81.7	103.1	1899.3
$\|e^{\text{rel}}\|_1$	0.33	0.37	2.20
$\|e^{\text{m_rel}}\|_1$	0.46	0.59	2.55
$\|e^{\text{comb}}\|_1,\ \alpha = 1, \beta = 100$	27.8	29.5	153.0

Table 3: Average errors of various methods for all equal queries for deletions.

Error Norm	*Exact*	*Probabilistic Counting*	*Static*
$\|e^{\text{abs}}\|_1$	4162	6776	22242
$\|e^{\text{abs}}\|_2$	4829	7772	35633
$\|e^{\text{rel}}\|_1$	0.04	0.07	0.21
$\|e^{\text{m_rel}}\|_1$	0.04	0.07	0.21
$\|e^{\text{comb}}\|_1,\ \alpha = 1, \beta = 100$	4.6	7.1	21.4

Table 4: Average errors of various methods for all one-side range queries for deletions.

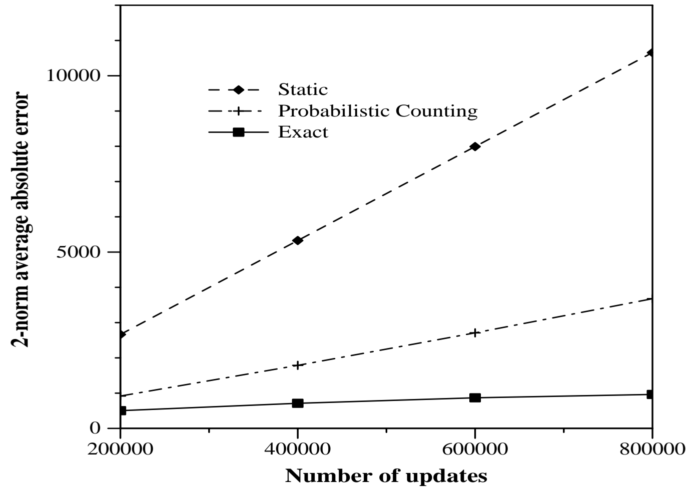


Figure 3: Accuracy of various methods for all equal queries for two-dimensional data

6.6 Accuracy for Maintenance of Multidimensional Histograms

Figure 3 and Figure 4 depict the accuracy of different methods for typical two-dimensional and three-dimensional data, respectively.

For the two-dimensional case, the value set size is 32×32. The base data follows the distribution *Zipf* $(1.0, negative)$ and its size is 200K to start with. The distributions of the inserted data follows *Zipf* $(1.5, random)$. We keep $m = 40$ coefficients in the histogram during the process. Figure 3 plot the 2-norm absolute errors of different methods for all equal queries when 200K, 400K, 600K, and 800K entries are inserted.

For the three-dimensional case, the value set size is $16 \times 16 \times 16$. The base data follows the

distribution *Zipf*(1.0, *positive*) and its size is 200K to start with. The distribution of the inserted data follows *Zipf*(1.5, *random*). We keep $m = 80$ coefficients in the histogram during the process. Figure 4 plot the 2-norm absolute error of different methods for all equal queries when 200K, 400K, 600K, and 800K entries are inserted.

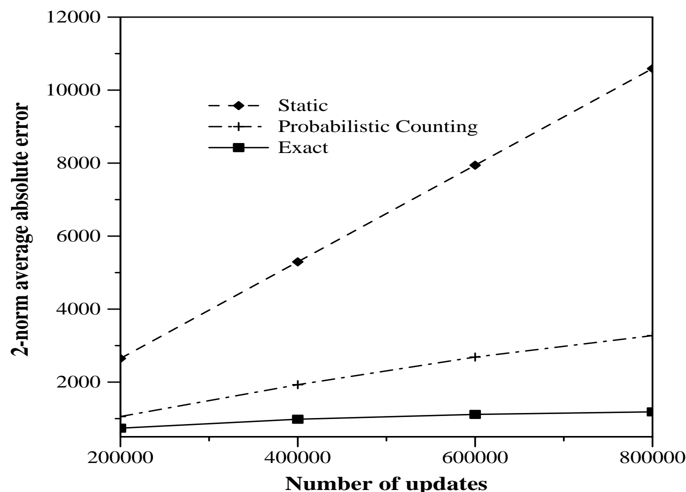


Figure 4: Accuracy of various methods for all equal queries for three-dimensional data

7 Conclusions

In this paper, we present a novel method based upon probabilistic counting to maintain wavelet-based histograms. Experiments show that our method can effectively maintain wavelet-based histograms for a wide variety of update sequences with very little online time and space costs. Our techniques provide much more accurate results than the static maintenance method [LKC99] used in previous work. The accuracy of the histogram maintained using our method is very close to that of the optimal histogram obtained by rebuilding from scratch upon any update.

We are working on extending our method to maintain the wavelet-based compact data cube proposed in [VW99]. When the underlying data are very large in size and have extremely high dimensionality, our algorithm as stated in Section 5 will invoke high CPU and I/O costs beccause each leaf update affects many internal node values. The main challenge in dynamic maintenance of the data cube is how to improve the method to achieve efficiency.

References

[AC99] A. Aboulnaga and S. Chaudhuri. Self-tuning histograms: Building histograms without looking at data. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of*

Data, pages 181–192, Phildelphia, June 1999.

[DIR99] D. Donjerkovic, Y. Ioannidis, and R. Ramakrishnan. Dynamic histograms: Capturing evolving data sets. Technical report, Department of Computer Science, University of Wisconsin-Madison, 1999.

[GMP97] P. B. Gibbons, Y. Matias, and V. Poosala. Fast incremental maintenance of approximate histograms. In *Proceedings of the 1997 International Conference on Very Large Databases*, Athens, Greece, August 1997.

[JS94] B. Jawerth and W. Sweldens. An overview of wavelet based multiresolution analyses. *SIAM Rev.*, 36(3):377–412, 1994.

[LKC99] J. Lee, D. Kim, and C. Chung. Multi-dimensional selectivity estimation using compressed histogram information. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, pages 205–214, Phildelphia, June 1999.

[MD88] M. Muralikrishna and D. J. DeWitt. Equi-depth histograms for estimating selectivity factors for multi-dimensional queries. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, pages 28–36, 1988.

[MVW98] Y. Matias, J. S. Vitter, and M. Wang. Wavelet-based histograms for selectivity estimation. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, pages 448–459, Seattle, WA, June 1998.

[PI97] V. Poosala and Y. E. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *Proceedings of the 1997 International Conference on Very Large Databases*, Athens, Greece, August 1997.

[PIHS96] V. Poosala, Y. E. Ioannidis, P. J. Haas, and E. Shekita. Improved histograms for selectivity estimation of range predicates. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, Montreal, Canada, May 1996.

[Poo97] V. Poosala. *Histogram-Based Estimation Techniques in Database Systems*. Ph. D. dissertation, University of Wisconsin-Madison, 1997.

[PSC84] G. Piatetsky-Shapiro and C. Connell. Accurate estimation of the number of tuples satisfying a condition. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pages 256–276, 1984.

[SDS96] E. J. Stollnitz, T. D. Derose, and D. H. Salesin. *Wavelets for Computer Graphics*. Morgan Kaufmann, 1996.

[VW99] J. S. Vitter and M. Wang. Approximate computation of multidimensional aggregates of sparse data using wavelets. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, pages 193–204, Phildelphia, June 1999.

[VWI98] J. S. Vitter, M. Wang, and B. Iyer. Data cube approximation and histograms via wavelets. In *Proceedings of Seventh International Conference on Information and Knowledge Management*, pages 96–104, Washington D.C., November 1998.

[Zip49] G. K. Zipf. *Human Behaviour and the Principle of Least Effort*. Addison-Wesley, Reading, MA, 1949.