Fast Incremental Maintenance of Approximate Histograms

PHILLIP B. GIBBONS
Intel Research Pittsburgh
YOSSI MATIAS
Tel Aviv University
and
VISWANATH POOSALA
Bell Laboratories

Many commercial database systems maintain histograms to summarize the contents of large relations and permit efficient estimation of query result sizes for use in query optimizers. Delaying the propagation of database updates to the histogram often introduces errors in the estimation. This paper presents new sampling-based approaches for *incremental* maintenance of approximate histograms. By scheduling updates to the histogram based on the updates to the database, our techniques are the first to maintain histograms effectively up-to-date at all times and avoid computing overheads when unnecessary. Our techniques provide highly-accurate approximate histograms belonging to the *equi-depth* and *Compressed* classes. Experimental results show that our new approaches provide orders of magnitude more accurate estimation than previous approaches.

An important aspect employed by these new approaches is a *backing sample*, an up-to-date random sample of the tuples currently in a relation. We provide efficient solutions for maintaining a uniformly random sample of a relation in the presence of updates to the relation. The backing sample techniques can be used for any other application that relies on random samples of data.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems—*Query processing*

General Terms: Algorithms, Experimentation, Performance

Additional Key Words and Phrases: approximation, histograms, incremental maintenance, sampling, query optimization

1. INTRODUCTION

Most database management systems (DBMSs) maintain a variety of statistics on the contents of the database relations in order to estimate various quantities, such as selectivities within cost-based query optimizers. These statistics are typically used to approximate the

A preliminary version of this article appeared in *Proceedings of the 23rd International Conference on Very Large Data Bases* (Athens, Greece, August). 1997, pp. 466–475.
Authors' addresses: P. B. Gibbons, Intel Research Pittsburgh, 417 South Craig Street, Suite 300, Pittsburgh, PA 15213, e-mail: phillip.b.gibbons@intel.com; Y. Matias, Department of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel, e-mail: matias@math.tau.ac.il; V. Poosala, Bell Laboratories, Room 2A-212, 600 Mountain Avenue, Murray Hill, NJ 07974, e-mail: poosala@lucent.com.

distribution of data in the attributes of various database relations. It has been established that the validity of the optimizer's decisions may be critically affected by the quality of these approximations [Christodoulakis 1984; Ioannidis and Christodoulakis 1991]. This is becoming particularly evident in the context of increasingly complex queries (e.g., data analysis queries).

The most common technique used in practice for selectivity estimation is maintaining *histograms* on the frequency distribution of an attribute. A histogram groups attribute values into "buckets" (subsets) and approximates true attribute values and their frequencies based on summary statistics maintained in each bucket [Kooi 1980]. For most real-world databases, there exist histograms that produce low-error estimates while occupying reasonably small space (of the order of 1K bytes in a catalog) [Poosala 1997]. Histograms are used in IBM DB2, Informix, Ingres, Oracle, Microsoft SQL Server, Sybase, and Teradata. They are also being used in other areas, e.g., parallel join load balancing [Poosala and Ioannidis 1996] to provide various estimates.

Histograms are usually *precomputed* on the underlying data and used without much additional overhead inside the query optimizer. A drawback of using precomputed histograms is that they may get outdated when the data in the database is modified, and hence introduce significant errors in estimations. On the other hand, it is clearly impractical to compute a new histogram after every update to the database. Fortunately, it is not necessary to keep the histograms perfectly up-to-date at all times, because they are used only to provide *reasonably accurate* estimates (typically within 1–10%). Instead, one needs appropriate schedules and algorithms for propagating updates to histograms, so that the database performance is not affected.

Despite the popularity of histograms, issues related to their maintenance have only recently started receiving attention. Most of the work on histograms so far has focused on proper bucketizations of values in order to enhance the accuracy of histograms, and assumed that the database is not being modified. In our earlier work, we have introduced several classes of histograms that offer high accuracy for various estimation problems [Poosala et al. 1996]. We have also provided efficient sampling-based methods to construct various histograms, but ignored the problem of maintaining histograms. In a more general context, we can view histograms as materialized views, but they are different in two aspects. First, during utilization, they are typically maintained in main memory, which implies more constraints on space. Second, they need to be maintained only approximately, and can therefore be considered as *cached approximate materialized views*. We are not aware of any prior work on approximate materialized views.

The most common approach used to date for histogram updates, which is followed in nearly all commercial systems, is to recompute histograms periodically (e.g., every night or on demand). This approach has two disadvantages. First, any significant updates to the data between two recomputations could cause poor estimations in the optimizer. Second, recomputing a histogram from scratch by scanning the entire relation is computationally expensive for large relations.

In this paper, we present fast and effective procedures for maintaining two histogram classes used extensively in database management systems: *equi-depth* histograms (which are used in most DBMSs) and *Compressed* histograms (used in DB2). There are three key novel components to our approach:

(1) We introduce the notion of an *approximate* histogram that is maintained in the pres-

ence of database updates, and which provides bounds on its maximum deviation from the true histogram.

(2) We develop a split and merge technique for quickly adjusting histogram buckets in response to data updates.

(3) We introduce the notion of a "backing sample", a random sample of the data that is kept up-to-date in the presence of database updates. We demonstrate important advantages gained by using a backing sample when updating histograms, and present algorithms for its maintenance. We observe that the backing sample can be used in any application that requires uniform random samples of the current data in the database. For example, instead of dynamically computing samples at usage-time (which is a drawback of several sampling-based techniques), one can precompute the samples and use our techniques to maintain them efficiently[1].

The main advantages of our techniques are as follows:

— Our approach leads to approximate histograms that are close to the actual histogram belonging to the same class, with high probability, regardless of the data distribution.

— Our algorithms handle all forms of updates to the database (*insert, delete*, and *modify* operations). They are most efficient in insert-intensive environments or in data warehousing environments that house transactional information for sliding time windows.

— Our algorithms process the sequence of database updates; they almost never access the relation on disk (the only exception is when the size of the relation has shrunk dramatically due to deleting, say, half the tuples). For most insert operations, our algorithms do not access the backing sample. The sample nevertheless remains up to date at all times.

We conducted an extensive set of experiments studying our techniques and comparing them with the traditional approaches based on recomputation. The experiments confirm the theoretical findings and show that with a small amount of additional storage and CPU resources, our techniques maintain histograms nearly up-to-date at all times.

Recent work. Since the completion of our work discussed in this paper, there have been a number of important developments in the area of histogram maintenance and related topics. First, there has been some commercial acceptance of using sampling to speed up histogram recomputation. For example, when SQL Server recomputes a histogram, it first extracts a random sample from the relation and then computes the histogram from the sample (see [Chaudhuri et al. 1998]). Thus the extracted random sample serves the same function as a backing sample, for the restricted purpose of computing a new histogram from scratch. Sampling during recomputation has the advantage that there are no overheads at database update time (versus the minimal overheads with backing samples). On the other hand, as discussed in Section 3 and elsewhere in this paper, there are a number of advantages to having a precomputed and maintained backing sample, and we exploit these advantages in our algorithms.

Second, a *split and merge* approach has been used to incrementally maintain histograms in response to feedback from the query execution engine about the actual selectivities of range queries [Aboulnaga and Chaudhuri 1999]. Such histograms are called *self-tuning*

[1] If a sampling-based algorithm requires a sample that may be larger than what is maintained, as can be the case for adaptive sampling [Lipton et al. 1990], then some ad-hoc sampling may be unavoidable.

histograms, because they automatically adapt to changes in the database without looking at the updates and without recomputing from the database. Instead, the actual selectivity of the executed range query is compared with the histogram estimate of that selectivity, and the histogram bucket counts are adjusted by spreading any discrepancy over the buckets that lie within the query range. Buckets with large counts are split. Buckets of near-equal frequencies are merged. Since a backing sample (or any equivalent means) is not used, there are no bounds proved for the maximum deviation of a self-tuning histogram from the true histogram. On the other hand, experimental results reported by Aboulnaga and Chaudhuri [1999] showed the technique performs well for multi-dimensional data distributions with low-to-moderate skew. More recently, Bruno et al. [2001] applied a similar feedback-based technique to multi-dimensional histograms. A key feature of their approach is its flexible partitioning of the multi-dimensional space into buckets. Unlike the techniques presented in our paper, neither of these approaches use the contents of the data in a direct manner.

Finally, there have been a number of recent papers on approximate histograms, their maintenance, and their use in query result size estimation and in providing fast approximate answers to queries (e.g., [Blohsfeld et al. 1999; Deshpande et al. 2001; Gilbert et al. 2002; Gilbert et al. 2002; Greenwald and Khanna 2001; Gunopulos et al. 2000; Guha et al. 2001; Ioannidis and Poosala 1999; Jagadish et al. 1998; Konig and Weikum 1999; Matias et al. 1998; 2000; Poosala and Ioannidis 1997]). Moreover, the notion of a backing sample has been extended to the general notion of precomputed (and maintained) sampling-based *data synopses*, which have been shown to be effective for providing fast approximate answers to queries (c.f. [Acharya et al. 2000; Acharya et al. 1999; Chaudhuri et al. 2001; Ganti et al. 2000; Gibbons 2001; Gibbons and Matias 1998]).

Outline of the paper. In Section 2, we discuss histograms, approximate histograms, and histogram maintenance. Backing samples and their maintenance are described in Section 3. Sections 4 and 5 present our algorithms for incremental maintenance of approximate equi-depth histograms and Compressed histograms, respectively. Our experimental evaluation is in Section 6, followed by conclusions in Section 7. A number of the proofs are left to the appendix.

2. HISTOGRAMS AND THEIR MAINTENANCE

The *domain* $\mathcal{D}$ of an attribute X is the set of all possible values of X and the *value set* $\mathcal{V}$ ($\subseteq \mathcal{D}$) for a relation R is the set of values of X that are present in R. Let $\mathcal{V} = \{v_i : 1 \leq i \leq |\mathcal{V}|\}$, where $v_i < v_j$ when $i < j$ and $|\mathcal{V}|$ is the cardinality of the set $\mathcal{V}$. The *frequency* f_i of v_i is the number of tuples in R whose value for attribute X is v_i. The *data distribution* of X (in R) is the set of pairs $\mathcal{T} = \{(v_1, f_1), (v_2, f_2), \ldots, (v_{|\mathcal{V}|}, f_{|\mathcal{V}|})\}$.

A *histogram* on attribute X is constructed by partitioning the data distribution $\mathcal{T}$ into β (≥ 1) mutually disjoint subsets called *buckets* and approximating the values and frequencies in each bucket in some common fashion. Typically, a bucket is assumed to contain either all m values in $\mathcal{D}$ between the smallest and largest values in that bucket (the bucket's *range*), or just $k \leq m$ equi-distant values in the range, where k is the number of distinct values in the bucket. The former is known as the *continuous value assumption* [Selinger et al. 1979], and the latter is known as the *uniform spread assumption* [Poosala et al. 1996]. Let the *bucket frequency* f_B be the number of tuples in R whose value for attribute X is in bucket B.[2] The frequencies for values in a bucket B are approximated by their averages; i.e., by either f_B/m or f_B/k.

Different *classes* of histograms can be obtained by using different rules for partitioning values into buckets. In this paper, we focus on two important classes of histograms, namely the *equi-depth* and *Compressed(V,F)* (simply called *Compressed* in this paper) classes. In an *equi-depth* (or *equi-height*) histogram, contiguous ranges of attribute values are grouped into buckets such that the number of tuples, f_B, in each bucket B is the same. In a *Compressed(V,F)* histogram [Poosala et al. 1996], the n highest frequencies are stored separately in n singleton buckets; the rest are partitioned as in an equi-depth histogram. In our target Compressed histogram, the value of n adapts to the data distribution to ensure that no singleton bucket can fit within an equi-depth bucket and yet no single value spans an equi-depth bucket. We have shown in our earlier work [Poosala et al. 1996] that Compressed histograms are very effective in approximating distributions of low or high skew.

Equi-depth histograms are used in one form or another in nearly all commercial systems, except DB2 which uses the more accurate Compressed histograms.

Histogram storage and usage. For both equi-depth and Compressed histograms, we store for each bucket B the largest value in the bucket, B.maxval, and a count, B.count, that equals or approximates f_B. If B is a singleton bucket, then its range is the single value B.maxval. Otherwise, its range is from the B'.maxval of its preceding bucket (or the minimum value in the domain $\mathcal{D}$, if B is the first bucket) to B.maxval, excluding the value of each singleton bucket within this range (if any).

When using the histograms to estimate range selectivities, we use the exact range information provided by singleton buckets and apply the continuous value assumption for equi-depth buckets. For equi-depth buckets, the uniform-spread assumption could be used instead, but it requires knowing the number of distinct values in each bucket, which is challenging to maintain (even approximately) under updates both to the database and to the histogram bucket boundaries.

2.1 Approximate histograms

An *approximate* class C histogram $\mathcal{H}^*$ on an attribute X for a relation R is a histogram that may deviate from the actual class C histogram $\mathcal{H}$ as R is updated. This deviation occurs because we cannot afford to recompute $\mathcal{H}$ each time R is updated. As R is modified, $\mathcal{H}^*$ may deviate from $\mathcal{H}$ in two ways:

(1) *Class Error:* first, $\mathcal{H}^*$ may no longer be the correct class C histogram for R, e.g., it may not have the same bucket boundaries as $\mathcal{H}$.

(2) *Distribution Error:* second, $\mathcal{H}^*$ may contain inaccurate information about X, e.g., it may not have the same bucket counts as $\mathcal{H}$.

The quality of an approximate histogram can be evaluated according to various error metrics defined based on the class and distribution errors.

[2] For any value v that is the right endpoint of ranges for $k \geq 1$ buckets, $B_i, B_{i+1}, \ldots, B_{i+k-1}$, there is ambiguity as to how to divide its frequency, f_v, in the entire relation among $f_{B_i}, f_{B_{i+1}}, \ldots, f_{B_{i+k}}$. In this paper, we select the following resolution to this ambiguity. If $f_v > (k-1)N/\beta$, we assign N/β of f_v to each bucket B_j, $j = i+1, \ldots, i+k-1$, with v for both endpoints, so that $f_{B_j} = N/\beta$. The remainder of f_v is assigned to f_{B_i}; none is assigned to $f_{B_{i+k}}$. If $f_v \leq (k-1)N/\beta$, we assign $f_{B_i} = 1$, $f_{B_{i+k}} = 0$, and, for $j = i+1, \ldots, i+k-1$, $f_{B_j} = (f_v - 1)/(k-1)$.

The μ_{count} error metric. As an example, consider the following distribution error metric, relevant to many histogram classes, which reflects the accuracy of the counts associated with each bucket. When R is modified, but the histogram is not, then there may be buckets B with B.count $\neq f_B$; the difference between f_B and B.count is the approximation error for B. We consider the error metric, μ_{count}, defined as follows:

$$\mu_{\text{count}} = \frac{\beta}{N}\sqrt{\frac{1}{\beta}\sum_{i=1}^{\beta}(f_{B_i} - B_i.\text{count})^2}\ , \tag{1}$$

where N is the number of tuples in R and β is the number of buckets. This is the standard deviation of the bucket counts from the actual number of elements in each bucket, normalized with respect to the mean bucket count (N/β).

2.2 Incremental histogram maintenance

The approach followed for maintenance in nearly all commercial systems is to recompute histograms periodically (e.g., every night), regardless of the number of updates performed on the database. This approach has two disadvantages. First, any significant updates to the data since the last recomputation could result in poor estimations by the optimizer. Second, because the histograms are recomputed from scratch by discarding the old histograms, the recomputation phase for the entire database can be computationally very intensive and may have to be performed when the system is lightly loaded or off-line.[3]

Instead, we propose an *incremental* technique, which maintains approximate histograms within specified errors bounds at all times with high probability and never accesses the underlying relations for this purpose. There are two components to our incremental approach: (i) maintaining a backing sample, and (ii) a framework for maintaining an approximate histogram that performs a few program instructions in response to each update to the database,[4] and detects when the histogram is in need of an adjustment of one or more of its bucket boundaries. Such adjustments make use of the backing sample. There is a fundamental distinction between the backing sample and the histogram it supports: the histogram is accessed far more frequently than the sample and uses less memory, and hence it can be stored in main memory while the sample is likely stored on disk. Figure 1 shows typical sizes of various entities relevant to our discussion.

Incremental histogram maintenance was previously studied in [Gibbons and Matias 1998], for the important case of a high-biased histogram, which is a Compressed histogram with $\beta - 1$ buckets devoted to the $\beta - 1$ most frequent values, and 1 bucket devoted to all the remaining values. This algorithm did not use the approach described above — for example, no backing sample was maintained or used.

In the next section, we describe how the backing sample is maintained in the context of our approach.

[3] To help alleviate this latter problem, some commercial systems such as SQL Server recompute (approximate) histograms by first sampling the data and then computing a histogram on the sampled data (as discussed in Section 1).

[4] To further reduce the overhead of our approach, the few program instructions can be performed only for a random sample of the database updates (as discussed in Section 6.2).

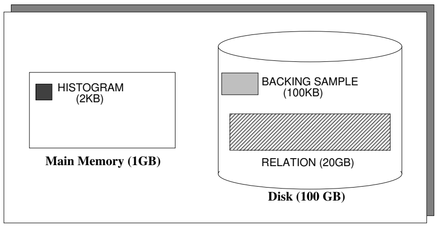

Fig. 1. Typical sizes of various entities

3. BACKING SAMPLE

A *backing sample* is a uniform random sample of the tuples in a relation that is kept up-to-date in the presence of updates to the relation. For each tuple, the sample contains the unique row id and one or more attribute values.

We argue that maintaining a backing sample is useful for histogram computation, selectivity estimation, etc. In most sampling-based estimation techniques, whenever a sample of size n is needed, either the entire relation is scanned to extract the sample, or several random disk blocks are read. In the latter case, the tuples in a disk block may be highly correlated, and hence to obtain a truly random sample, n disk blocks must be read. In contrast, a backing sample can be stored in consecutive disk blocks, and can therefore be scanned by reading sequential disk blocks. Moreover, for each tuple in the sample, only the unique row id and the attribute(s) of interest are retained. Thus the entire sample can be stored in only a small number of disk blocks, for even faster retrieval. Finally, an indexing structure for the sample can be created, maintained and stored; the index enables quick access to sample values within any desired range.

At any given time, the backing sample for a relation R needs to be equivalent to a random sample of the same size that would be extracted from R at that time. Thus the sample must be updated to reflect any updates to R, but without the overheads of such costly extractions. In this section, we present techniques for maintaining a provably random backing sample of R based on the sequence of updates to R, while accessing R very infrequently (R is accessed only when an update sequence deletes about half the tuples in R).

Let $\mathcal{S}$ be a backing sample maintained for a relation R. We first consider insertions to R. Our technique for maintaining $\mathcal{S}$ as a simple random sample in the presence of inserts is based on the *Reservoir Sampling* techniques due to Vitter [Vitter 1985]. Typically, in DBMSs, the reservoir sampling algorithm is used to obtain a sample of the data during a single scan of the relation without *a priori* knowledge about the number of tuples in the relation. The particular version described here (called Algorithm X in Vitter's paper), is as follows: The algorithm proceeds by inserting the first n tuples into a "reservoir." Then a random number of records are skipped, and the next tuple replaces a randomly selected tuple in the reservoir. Another random number of records are then skipped, and so forth, until the last record has been scanned. The distribution function of the length of each random skip depends explicitly on the number of tuples scanned so far, and is chosen such that each tuple in the relation is equally likely to be in the reservoir after the last tuple has been scanned. By treating the tuple being inserted in the relation as the next tuple in the scan of the relation, we essentially obtain a sample of the data in the presence of insertions.

Extensions to handle modify and delete operations. We extend Vitter's algorithm to handle modify and delete operations, as follows. Modify operations are handled by updating the value field, if the tuple is in the sample. Delete operations are handled by removing the tuple from the sample, if it is in the sample. However, such deletions decrease the size of the sample from the target size n, and moreover, it is not known how to use subsequent insertions to obtain a provably random sample of size n once the sample has dropped below n. Instead, we maintain a sample whose size is initially a prespecified upper bound U, and allow for it to decrease as a result of deletions of sample items down to a prespecified lower bound L. If the sample size drops below L, we rescan the relation to re-populate the random sample. In the appendix, we show that such rescans are expected to be infrequent for large relations, and moreover, for databases with infrequent deletions, no such rescans are expected. Even in the worst case where deletions are frequent, the cost of any rescans can be amortized against the cost of the (expected) large number of deletions required before a rescan becomes necessary.

Our algorithm, denoted **MaintainBackingSample**, is depicted in Figure 2. For each tuple selected for the backing sample $\mathcal{S}$, we store its (unique) row id and the value(s) of all attribute(s) of interest to any applications that will use the backing sample (e.g., for histograms, we store the value of the attribute on which the histogram is to be computed). For simplicity, we have shown in this figure only the case of a single attribute, X, of interest, and we have not shown any of the performance optimizations described below. The algorithm maintains the property that $\mathcal{S}$ is a uniform random sample of a relation R such that $\min(|R|, L) \leq |\mathcal{S}| \leq U$.

THEOREM 3.1. *Algorithm MaintainBackingSample maintains a uniform random sample of relation R.*

The proof appears in the appendix.

Optimizations. There are several techniques that can be applied to lower the overheads of the algorithm. First, a hash table of the row ids of the tuples in $\mathcal{S}$ can be used to speed up the test of whether or not an id is in $\mathcal{S}$. Second, if the primary source of delete operations is to delete from R all tuples before a certain date, as in the case of many data warehousing environments that maintain a sliding window of the most recent transactional data on disk, then such deletes can be processed in one step by simply removing all tuples in $\mathcal{S}$ that are before the target date. Third, and perhaps most importantly, we observe that the algorithm maintains a random sample independent of the order of the updates to the database. Thus we can "rearrange" the order to suit our needs, until an up-to-date sample is required by the application using the sample. We can use lazy processing of modify and delete operations, whereby such operations are simply placed in a buffer to be processed as a batch whenever the buffer becomes full or an up-to-date sample is needed. Likewise, we can postpone the processing of modify and delete operations until the next insert that is selected for $\mathcal{S}$. Specifically, instead of flipping a biased coin for each insert, we select a random number of inserts to skip, according to the criterion of Vitter's Algorithm X (this criterion is statistically equivalent to flipping the biased coin each insert). At that insert, we first process all modify and delete operations that have occurred since the last selected insert, then we have the new insert replace a randomly selected tuple in $\mathcal{S}$. Another random number of inserts are then skipped, and so forth. Note that postponing the modify and delete operations is important, since it reduces the problem to the insert-only case, and

MaintainBackingSample()

// $\mathcal{S}$ is the backing sample, R is the relation, X is the attribute of interest.
// L and U are prespecified lower and upper bounds for the size of the sample.

After an insert of a tuple τ with $\tau.ID = id$ and $\tau.X = v$ into R:
if $|\mathcal{S}| + 1 = |R| \leq U$ then
    $\mathcal{S} := \mathcal{S} + \{(id, v)\}$;
else with probability $|\mathcal{S}|/|R|$ do begin
    select a tuple (id', v') in $\mathcal{S}$ uniformly at random;
    $\mathcal{S} := \mathcal{S} + \{(id, v)\} - \{(id', v')\}$;
end;

After a modify to a tuple τ with $\tau.ID = id$ in R:
if the modify changes $\tau.X$ then do begin
    if id is in $\mathcal{S}$ then
        update the value field for tuple id in $\mathcal{S}$;
end;

After a delete of a tuple τ with $\tau.ID = id$ from R:
if id is in $\mathcal{S}$ then do begin
    remove the tuple id from $\mathcal{S}$;
    // This next conditional is expected to be true only when a constant
    // fraction of the database updates are delete operations.
    if $|\mathcal{S}| < \min(|R|, L)$ then do begin
        // Discard $\mathcal{S}$ and rescan R to compute a new $\mathcal{S}$.
        $\mathcal{S} := \emptyset$;
        rescan R, and for each tuple, apply the above procedure for inserts into R;
    end;
end;

Fig. 2. An algorithm for maintaining a backing sample of a relation under updates to the database

hence the criterion of Algorithm X can be applied to determine how many inserts to skip.

With these optimizations, insert and modify operations to attributes not of interest are processed with minimal overhead, whereas delete and modify operations to attributes of interest may require larger overhead (due to the batch processing of testing whether the id is in the sample). Thus the algorithm is best suited for insert-mostly databases or for the data warehousing environments discussed above.

4. FAST MAINTENANCE OF APPROXIMATE EQUI-DEPTH HISTOGRAMS

In this section, we demonstrate our approach for incremental histogram maintenance by considering a specific, important histogram: the equi-depth histogram. First, we present an algorithm for maintaining an approximate equi-depth histogram in the presence of insertions to the database; this algorithm has provable guarantees on its accuracy. Next, we show how heuristics can be used to modify the algorithm in order to minimize the overheads. Finally, we show how to extend both algorithms to handle modify and delete operations to the database. We assume throughout that a backing sample, $\mathcal{S}$, is being maintained using the algorithm of Figure 2.

The standard algorithm for constructing an (exact) equi-depth histogram first sorts the tuples in the relation by attribute value, and then selects tuples $\lfloor i \cdot N/\beta \rfloor$, for $i = 1, \ldots, \beta$. However, for large relations, this algorithm is quite slow because the sorting may involve multiple I/O scans of the relation.

```
EquiDepthSampleCompute();
    // S is the sample to be used to compute the histogram, sorted on the attribute value X.
    // N is the total number of tuples in R.
    // β is the desired number of buckets.
For i := 1 to β do begin
    τ := the ⌊i · |S|/β⌋'th tuple in S;
    B_i.maxval := τ.X;
    B_i.count := ⌊i · N/β⌋ − ⌊(i − 1) · N/β⌋;
end;
return ({B_1, B_2, ..., B_β})
```

Fig. 3. Procedure for computing an approximate equi-depth histogram from a random sample

An *approximate* equi-depth histogram approximates the exact histogram by relaxing the requirement on the number of tuples in a bucket and/or the accuracy of the counts. Such histograms can be evaluated based on how close the buckets are to N/β tuples and how close the counts are to the actual number of tuples in their respective buckets.

A class error metric for equi-depth histograms. Consider an approximate equi-depth histogram with β buckets for a relation of N tuples. We consider an error metric, μ_{ed}, that reflects the extent to which the histogram's bucket boundaries succeed in evenly dividing the tuples in the relation:

$$\mu_{\text{ed}} = \frac{\beta}{N}\sqrt{\frac{1}{\beta}\sum_{i=1}^{\beta}\left(f_{B_i} - \frac{N}{\beta}\right)^2}\ . \tag{2}$$

This is the standard deviation of the buckets' sizes from the mean bucket size, normalized with respect to the mean bucket size.

Computing approximate equi-depth histograms from a random sample. Given a random sample, an approximate equi-depth histogram can be computed by constructing an equi-depth histogram on the sample but setting the bucket counts to be N/β ([Poosala et al. 1996]). This algorithm, denoted **EquiDepthSampleCompute**, is depicted in Figure 3.

Section 4.1 presents an incremental algorithm that occasionally uses EquiDepthSample-Compute. The accuracy of the approximate histogram maintained by the incremental algorithm depends on the accuracy resulting from this procedure, which is stated in the following theorem[5]. The statement of the theorem is in terms of a sample size m. To ensure such a sample size for the backing sample we maintain, we set L to be at least m in MaintainBackingSample.

THEOREM 4.1. *Let $\beta \geq 3$. Let $m = (c \ln^2 \beta)\beta$, for some $c \geq 4$. Let $\mathcal{S}$ be a random sample of size m of values drawn uniformly from a set of size $N \geq m^3$, either with or without replacement. Let $\alpha = (c \ln^2 \beta)^{-1/6}$. Then EquiDepthSampleCompute computes an approximate equi-depth histogram such that with probability at least $1 - \beta^{-(\sqrt{c}-1)} - N^{-1/3}$, $\mu_{\text{ed}} = \mu_{\text{count}} \leq \alpha$.*

[5]Even though the computation of approximate histograms from a random sample of a fixed relation R has been considered in the past, we are not aware of a similar analysis.

EquiDepthSimple()

// R is the relation, X is the attribute of interest.
// $\mathcal{H}$ is the ordered set of β buckets in the current histogram.
// T is the current upper bound threshold on a bucket count.
// $\gamma > -1$ is a tunable performance parameter.

After an insert of a tuple τ with $\tau.X = v$ into R:
Determine the bucket $B \in \mathcal{H}$ whose interval contains v;
B.count := B.count $+ 1$;

if B.count $= T$ then do begin
  $\mathcal{H}$:= *EquiDepthSampleCompute*(); // (See Figure 3).
  $T := \lceil (2+\gamma) \cdot |R|/\beta \rceil$;
end;

Fig. 4. An algorithm for maintaining an approximate equi-depth histogram under insertions to the database

The proof is given in the appendix.

4.1 Maintaining equi-depth histograms using a backing sample

Given our backing sample, we can compute an approximate equi-depth histogram at any time, using EquiDepthSampleCompute. To maintain approximate histograms in the presence of database updates, one could invoke this procedure whenever the backing sample is modified. However, the overheads of this approach may be too large, and we would like instead to have a procedure that can maintain the histogram while only occasionally going to the backing sample to perform a full recomputation.

To this end, we devise an algorithm that monitors the accuracy of the histogram, and performs (partial) recomputation only when the approximation error exceeds a prespecified tolerance parameter. Figure 4 depicts the new algorithm, denoted **EquiDepthSimple**.

The algorithm proceeds in a series of phases. At each phase we maintain a threshold $T = \lceil (2+\gamma)N'/\beta \rceil$, where N' is the number of tuples in the relation R at the beginning of the phase, and $\gamma > -1$ is a tunable performance parameter. The threshold is set at the beginning of each phase. The number of tuples in any given bucket is maintained below the threshold T. (Recall that the ideal target number for a bucket size would be $|R|/\beta$.) As new tuples are added to the relation, we increment the counts of the appropriate buckets. When a count exceeds the threshold T, the entire equi-depth histogram is recomputed from the backing sample using EquiDepthSampleCompute, and a new phase is started.

Performance analysis. We first consider the accuracy of the above algorithm, and show that with very high probability it is guaranteed to be a good approximation for the equi-depth histogram. The following theorem shows that the error parameter μ_{count} remains unchanged, whereas the error parameter μ_{ed} may grow by an additive factor of at most $(1+\gamma)$, the tolerance parameter. The statement of the theorem is in terms of a sample size m.

THEOREM 4.2. *Let $\beta \geq 3$. Let $m = (c \ln^2 \beta)\beta$, for some $c \geq 4$. Consider EquiDepthSimple applied to a sequence of $N \geq m^3$ inserts of tuples into an initially empty relation. Let S be a random sample of size m of tuples drawn uniformly from the relation, either with or without replacement. Let $\alpha = (c \ln^2 \beta)^{-1/6}$. Then EquiDepthSimple computes an approximate equi-depth histogram such that with probability at least*

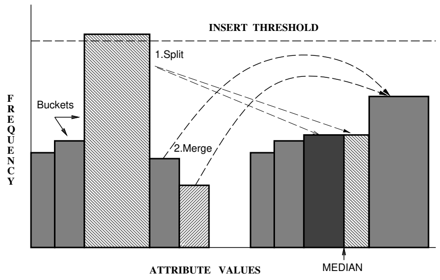

Fig. 5. Split and merge operation during equi-depth histogram maintenance

$1-\beta^{-(\sqrt{c}-1)}-(N/(2+\gamma))^{-1/3}$, $\mu_{\text{ed}} \leq \alpha+(1+\gamma)$ *and* $\mu_{\text{count}} \leq \alpha$.

The proof appears in the appendix.

We now consider the performance of the algorithm in terms of its computational overhead. Consider the cost of the calls to EquiDepthSimple. It is dominated by the cost of reading from disk a relation of size $|\mathcal{S}|$, in order to extract the β sample quantiles. This procedure is called at the beginning of each phase. It is easy to see that if the relation size is N at the beginning of the phase, then the number of insertions before the phase ends is at least $(1+\gamma)N/\beta$. Also the relation size at the end of the phase is at least $(1+(1+\gamma)/\beta)N$. These observations can be used to prove the following lemma, which bounds the total number of calls to EquiDepthSampleCompute as a function of the final relation size and the tolerance parameter γ.

LEMMA 4.3. *Let $\alpha = 1+(1+\gamma)/\beta$. If a total of N tuples are inserted in all, then the number of calls to EquiDepthSampleCompute is at most* $\min(\log_\alpha N, N)$.

4.2 The split&merge algorithm

In this section we modify the previous algorithm in order to reduce the number of recomputations from the sample, by trying to balance the buckets using a local, inexpensive procedure, before resorting to EquiDepthSampleCompute. When a bucket count reaches the threshold, T, we split the bucket in half instead of recomputing the entire histogram from the backing sample. In order to maintain the number of buckets, β, fixed, we merge two adjacent buckets whose total count does not exceed T, if such a pair of buckets can be found. Only when a merge is not possible do we recompute from the backing sample. As before, we define a *phase* to be the sequence of operations between consecutive recomputations. Figure 6 depicts our new algorithm, denoted **EquiDepthSplitMerge**.

The operation of merging two adjacent buckets is quite simple; it merely involves adding the counts of the two buckets and disposing of the boundary (quantile) between them. The splitting of a bucket is less straightforward; an approximate median value in the bucket is selected to serve as the bucket boundary between the two new buckets, using the backing sample. In particular, we select the median value among all tuples in the backing sample that fall within the bucket being split. To minimize disk accesses when determining the median value in a bucket, we keep the backing sample organized on disk according to histogram bucket. (Note that we visit the backing sample each time we split or merge, so we can readily maintain this organization). The split and merge operation is illustrated in Figure 5. Note that split and merge can occur only for $\gamma > 0$.

```
EquiDepthSplitMerge()

    // R is the relation, X is the attribute of interest.
    // H is the ordered set of β buckets in the current histogram.
    // T is the current threshold for splitting a bucket.
    // γ > −1 is a tunable performance parameter.

After an insert of a tuple τ with τ.X = v into R:
Determine the bucket B ∈ H whose interval contains v;
B.count := B.count + 1;

if B.count = T then do begin
    if ∃ buckets B_i and B_{i+1} such that B_i.count + B_{i+1}.count < T then do begin
        // Merge buckets B_i and B_{i+1}.
        B_{i+1}.count := B_i.count + B_{i+1}.count;

        // Split bucket B using the backing sample; use B_i for the first half of B's tuples.
        m := median value among all tuples in S associated with bucket B.
        B_i.maxval := m;
        B_i.count := ⌊T/2⌋;
        B.count := ⌈T/2⌉;
        Reshuffle equi-depth buckets in H back into sorted order;
    end;

    else do begin
        // No buckets suitable for merging, so recompute the histogram from S.
        H := EquiDepthSampleCompute();    // (See Figure 3).
        T := ⌈(2 + γ) · |R|/β⌉;
    end;
end;
```

Fig. 6. The Split&Merge algorithm for maintaining an approximate equi-depth histogram under insertions

The tolerance parameter γ determines how often a recomputation from the backing sample occurs. Consider the extreme case of $\gamma \approx -1$. Here EquiDepthSplitMerge recomputes the histogram with each database update, i.e., there are $\Theta(|R|)$ phases. Consider the other extreme, of setting $\gamma > |R|$. Then the algorithm simply sticks to the original buckets, and is therefore equivalent to the trivial algorithm which does not employ any balancing operation. Thus, the setting of the performance parameter γ gives a spectrum of algorithms, from the most efficient one which provides very poor accuracy performance, to the relatively accurate algorithm which has a rather poor efficiency performance. By selecting a suitable intermediate value for γ, we can obtain an algorithm with good performance, both in accuracy as well as in efficiency. For instance, setting $\gamma = 1$ will result with an algorithm whose imbalance factor is bounded by about 3 (since each phase begins with roughly $|R|/\beta$ tuples per bucket, by Theorem 4.2, and the threshold T for splitting a bucket is 3 times that number), and the number of phases is $O(\log N)$ (as will be shown in Theorem 4.6 below).

The following lemma establishes a bound on the number of splits in a phase, as a function of γ. We prove it for the range $\gamma \leq 2$, in which we are particularly interested.

LEMMA 4.4. *Let $\gamma \leq 2$. The number of splits that occur in a phase is at most β.*

PROOF. Let a bucket be denoted as *intact* if it was involved in neither a bucket split nor a bucket merge since the beginning of a phase. We claim that at every merge of two adjacent buckets, at least one of these buckets must be intact. Indeed, note that a bucket that has participated in a bucket split has a count of at least $T/2$. Further, note that a bucket

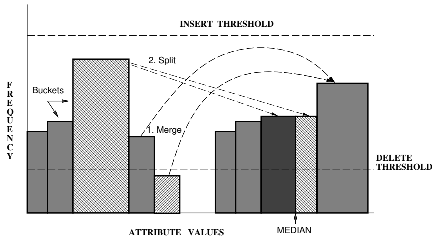

Fig. 7. Merge and split operation during equi-depth histogram maintenance

that has participated in a bucket merge has a count of at least $2 \cdot T/(2+\gamma) \geq T/2$, since the bucket counts at the beginning of the phase were $T/(2+\gamma)$, and $\gamma \leq 2$. The claim follows from the observation that at least one of the merging buckets must have a count smaller than $T/2$.

The claim implies that for every bucket merge, the number of intact buckets decreases by at least one, and hence the total number of possible bucket merges in a phase is at most β. The lemma now follows since each bucket split occurs after a bucket merge. □

The number of phases is bounded as follows:

LEMMA 4.5. *Let $\alpha = 1 + \gamma/2$ if $\gamma > 0$, and otherwise let $\alpha = 1 + (1+\gamma)/\beta$. If a total of N tuples are inserted in all, then the number of calls to EquiDepthSampleCompute is at most* $\min(\log_\alpha N, N)$.

The proof appears in the appendix.

We can now conclude:

THEOREM 4.6. *Consider EquiDepthSplitMerge with β buckets and performance parameter $-1 < \gamma \leq 2$ applied to a sequence of N inserts. Then the total number of phases is at most* $\log_\alpha N$*, and the total number of splits is at most* $\beta \log_\alpha N$*, where $\alpha = 1 + \gamma/2$ if $\gamma > 0$, and otherwise $\alpha = 1 + (1+\gamma)/\beta$.*

4.3 Extensions to handle modify and delete operations

Consider first the EquiDepthSimple algorithm. To handle deletions to the database, we extend it as follows. Deletions can decrease the number of elements in a bucket relative to other buckets, so we use an additional threshold, T_ℓ, that serves as a lower bound on the count in a bucket. At the start of each phase, we set $T_\ell = \lfloor |R|/(\beta(2+\gamma_\ell)) \rfloor$, where $\gamma_\ell > -1$ is a tunable parameter. We also set T as before. Consider a deletion of a tuple τ with $\tau.X = v$ from R. Let B be the bucket in the histogram $\mathcal{H}$ whose interval contains v. We decrement B.count, and if now B.count $= T_\ell$ then we recompute $\mathcal{H}$ from the backing sample, and update both T and T_ℓ.

For modify operations, we observe that if the modify does not change the value of attribute X, or if it changes the value of X such that the old value is in the same bucket as the new value, then $\mathcal{H}$ remains unchanged. Else, we update $\mathcal{H}$ by treating the modify as a delete followed by an insert.

Note that the presence of delete and modify operations does not affect the accuracy of the histogram computed from the backing sample. Moreover, the upper and lower thresholds control the imbalance among buckets during a phase, so the histograms remain quite accurate. On the other hand, the number of phases can be quite large in the worst case. By repeatedly inserting items into the same bucket until T is reached, and then deleting these same items, we can force the algorithm to perform many recomputations from the backing sample. However, if the sequence of updates to a relation R is such that $|R|$ increases at a steady rate, then the number of recomputes can be bounded by a constant factor times the bound given in Lemma 4.3, where the constant depends on the rate of increase.

Now consider the EquiDepthSplitMerge algorithm. The extensions to handle delete operations are identical to those outlined above, with the following additions to handle the split and merge operations, as illustrated in Figure 7. If B.count $= T_\ell$, we merge B with one of its adjacent buckets and then split the bucket B' with the largest count, as long as B'.count $\geq 2(T_\ell + 1)$. (Note that B' might be the newly merged bucket.) If no such B' exists, then we recompute $\mathcal{H}$ from the backing sample. Modify operations are handled as outlined above.

Figure 8 depicts the full split&merge algorithm, denoted **EquiDepthSplitMerge2**, for maintaining an approximate equi-depth histogram under insert, delete, and modify operations.

5. FAST MAINTENANCE OF APPROXIMATE COMPRESSED HISTOGRAMS

In this section, we consider another important histogram type, the *Compressed(V, F)* histogram. We first present a "split&merge" algorithm for maintaining a Compressed histogram in the presence of database insertions, and then show how to extend the algorithm to handle database modify and delete operations. To simplify the presentation, we will omit any explicit rounding of quotients to the next smaller or larger integer. We assume throughout that a backing sample, $\mathcal{S}$, is being maintained using MaintainBackingSample.

Definitions. Consider a relation of (a priori unknown) size N. In an equi-depth histogram, values with high frequencies can span a number of buckets; this is a waste of buckets since the sequence of spanned buckets for a value can be replaced with a single bucket with a single count. A Compressed histogram has a set of such singleton buckets and an equi-depth histogram over values not in singleton buckets. Our target Compressed histogram with β buckets has β' equi-depth buckets and $\beta - \beta'$ singleton "high-biased" buckets, where $1 \leq \beta' \leq \beta$, such that the following requirements hold: (R1) each equi-depth bucket has N'/β' tuples, where N' is the total number of tuples in equi-depth buckets, (R2) no single value "spans" an equi-depth bucket, i.e., the set of bucket boundaries are distinct, and conversely, (R3) the value in each singleton bucket has frequency $\geq N'/\beta'$. Associated with each bucket B is a maximum value B.maxval (either the singleton value or the bucket boundary) and a count, B.count.

An *approximate* Compressed histogram approximates the exact histogram by relaxing one or more of the three requirements above and/or the accuracy of the counts.

Class error metrics. Consider an approximate Compressed histogram $\mathcal{H}$ with equi-depth buckets $B_1, \ldots, B_{\beta'}$ and singleton buckets $B_{\beta'+1}, \ldots, B_\beta$. Recall that f_B is defined to be the number of tuples in a bucket B. Let N' be the number of tuples in equi-depth buckets, i.e., $N' = \sum_{i=1}^{\beta'} f_{B_i}$. We define two class error metrics, μ_{ed} and μ_{hb} (μ_{ed} is as

EquiDepthSplitMerge2()

// R is the relation, X is the attribute of interest.
// $\mathcal{H}$ is the ordered set of β buckets in the current histogram.
// T (T_ℓ) is the current upper bound (lower bound, resp.) threshold on a bucket count.
// $\gamma > -1$ and $\gamma_\ell > -1$ are tunable performance parameters.

After an insert of a tuple τ with $\tau.X = v$ into R:
Determine the bucket $B \in \mathcal{H}$ whose interval contains v; B.count := B.count $+ 1$;

if B.count $= T$ then do begin
  if $\exists$ buckets B_i and B_{i+1} such that B_i.count $+$ B_{i+1}.count $< T$ then do begin
    B_{i+1}.count := B_i.count $+$ B_{i+1}.count; // Merge buckets B_i and B_{i+1}.
    m := median value among all tuples in $\mathcal{S}$ associated with bucket B. // Split bucket B.
    B_i.maxval := m; B_i.count := $\lfloor T/2 \rfloor$; B.count := $\lceil T/2 \rceil$;
    Reshuffle equi-depth buckets in $\mathcal{H}$ back into sorted order;
  end;

  else do begin // No buckets suitable for merging, so recompute $\mathcal{H}$ from $\mathcal{S}$.
    $\mathcal{H}$:= *EquiDepthSampleCompute*(); // (See Figure 3).
    T := $\lceil (2+\gamma) \cdot |R|/\beta \rceil$; T_ℓ := $\lfloor (|R|/\beta)/(2+\gamma_\ell) \rfloor$;
  end;
end;

After a modify of a tuple τ in R, with $\tau.X = v$ before the modify and $\tau.X = v'$ after the modify:
if $v \neq v'$ then do begin
  Determine the buckets for v and v';
  If v and v' belong to different buckets then do begin
    Apply the procedure below for deleting τ with $\tau.X = v$ from R;
    Apply the procedure above for inserting τ with $\tau.X = v'$ into R;
  end;
end;

After a delete of a tuple τ with $\tau.X = v$ from R:
Determine the bucket $B \in \mathcal{H}$ whose interval contains v; B.count := B.count $- 1$;

if B.count $= T_\ell$ then do begin // Merge bucket B with one of its adjacent buckets.
  B_i := an adjacent bucket to B;
  B_i.maxval := $\max(B$.maxval, B_i.maxval); B_i.count := B.count $+$ B_i.count;

  B' := a bucket such that $\forall j$: B'.count $\geq$ B_j.count;
  if B'.count $\geq 2(T_\ell + 1)$ then do begin // Split bucket B'.
    m := median value among all tuples in $\mathcal{S}$ associated with bucket B'.
    B.maxval := m; B.count := $\lfloor B'$.count$/2 \rfloor$; B'.count := $\lceil B'$.count$/2 \rceil$;
    Reshuffle equi-depth buckets in $\mathcal{H}$ back into sorted order;
  end;

  else do begin // No bucket suitable for splitting, so recompute $\mathcal{H}$ from $\mathcal{S}$.
    $\mathcal{H}$:= *EquiDepthSampleCompute*(); // (See Figure 3).
    T := $\lceil (2+\gamma) \cdot |R|/\beta \rceil$; T_ℓ := $\lfloor (|R|/\beta)/(2+\gamma_\ell) \rfloor$;
  end;
end;

Fig. 8. The Split&Merge algorithm for maintaining an approximate equi-depth histogram under updates

CompressedSampleCompute();

// $\mathcal{S}$ is the random sample of R used to compute the histogram.
// β is the desired number of buckets.

// Compute the scaling factor for the bucket counts, and initialize β' and m'.
$\lambda := |R|/|\mathcal{S}|$; $\beta' := \beta$; $m' := |\mathcal{S}|$;

For each value v in $\mathcal{S}$ compute $f_v^{\mathcal{S}}$, the frequency of v in $\mathcal{S}$;
Let $v_1, v_2, \ldots, v_{\beta-1}$ be the $\beta - 1$ most frequent values in nonincreasing order.

For $i := 1$ to $\beta - 1$ while $f_{v_i}^{\mathcal{S}} \geq m'/\beta'$ do begin

// Create a singleton bucket for v_i.
$B_{\beta'}$.maxval := v_i; $B_{\beta'}$.count := $\lambda \cdot f_v^{\mathcal{S}}$;
$m' := m' - f_v^{\mathcal{S}}$; $\beta' = \beta'$ - 1;
end;

Let $\mathcal{S}'$ be the tuples in $\mathcal{S}$ whose values are not in singleton buckets, sorted by value;

// Create β' equi-depth buckets from $\mathcal{S}'$.
For $i := 1$ to β' do begin
u := the value of tuple $i \cdot m'/\beta'$ in $\mathcal{S}'$;
B_i.maxval := u; B_i.count := $\lambda \cdot m'/\beta'$;
end;

return $(\left\{B_1, B_2, \ldots, B_\beta\right\}, \lambda \cdot m', \beta')$;

Fig. 9. Procedure for computing an approximate Compressed histogram from a random sample

defined in Section 4 but applied only to the equi-depth buckets):

$$\mu_{\text{ed}} = \frac{\beta'}{N'} \sqrt{\frac{1}{\beta'} \sum_{i=1}^{\beta'} \left(f_{B_i} - \frac{N'}{\beta'}\right)^2} \tag{3}$$

$$\mu_{\text{hb}} = \frac{\beta'}{N'} \sum_{v \in U} \left| f_v - \frac{N'}{\beta'} \right| \tag{4}$$

where U is the set of values that violate requirement (R2) or (R3). This metric penalizes for mistakes in the choice of high-biased buckets in proportion to how much the true frequencies deviate from the target threshold, N'/β', normalized with respect to this threshold.

Computing approximate Compressed histograms from a random sample. Given a random sample, an approximate Compressed histogram can be computed by constructing a Compressed histogram on the sample but scaling the bucket counts by the scaling factor $|R|/|\mathcal{S}|$. This algorithm, denoted **CompressedSampleCompute**, is depicted in Figure 9. This is a new algorithm for computing an approximate version of our target Compressed histogram, and can be used as well to compute our exact target Compressed histogram by taking $\mathcal{S}$ to be all of R.

Note that if the counts in the sample $\mathcal{S}$ accurately reflect the counts in the set R, then the condition $f_{v_i}^{\mathcal{S}} \geq m'/\beta'$ of the first loop addresses Requirement (R3). The size of the error μ_{hb} will depend on how accurately $f_{v_i}^{\mathcal{S}}$ represents f_{v_i}, as well as the magnitude of the residual sample size m'. The latter is the number of items left in the sample after removing all copies of items v_j, $j \leq i$. The accuracy directly depends on m'/β', and for good accuracy we should aim at having $m'/\beta' \geq \lambda$, for a suitable choice of λ. For example, $\lambda \geq 5$ ensures good accuracy with reasonably high confidence. Problems arise with highly skewed data. For example, if a single value were sufficiently popular such that all the remaining values together were only a fraction $\alpha < \frac{1}{2}$ of the entire relation, and the backing sample size was such that $|\mathcal{S}| < \lambda(\beta - 1)/\alpha$, then after the first iteration, we have

$$\frac{m'}{\beta'} = \frac{m'}{\beta - 1} \approx \frac{\alpha|\mathcal{S}|}{\beta - 1} < \lambda \ .$$

This implies that having a backing sample that is sufficiently large to satisfy Requirement (R3) for highly-skewed data will be very wasteful for more uniform data.

A possible solution is to replace the random sample with a *concise sample*, as defined in [Gibbons and Matias 1998]. A concise sample represents multiple sample items having the same value v as a single pair $\langle v, c(v) \rangle$, where $c(v)$ is the number of sample items with value v (tuple IDs are not retained). Thus each value in the sample uses only constant space, regardless of its popularity. This enables a larger uniform sample to be stored within the given space bound, and in particular, the frequency $f^{\mathcal{S}}_{v_i}$ in a concise sample is essentially indifferent to the popularity of other values v_j, $j \neq i$. As a result we can set the space bound for $\mathcal{S}$ such that $\mathcal{S}$ will be sufficient but not wasteful across all distributions, regardless of skew.

5.1 A split&merge algorithm for Compressed histograms

In this section, we show how the approach in EquiDepthSplitMerge can be extended to handle Compressed histograms.

On an insertion of a tuple with value v into the relation, the (singleton or equi-depth) bucket, B, for v is determined, and the count is incremented. If B is an equi-depth bucket, then as in EquiDepthSplitMerge, we check to see if its count now equals the threshold T for splitting a bucket, and if it does, we update the bucket boundaries. The steps for updating the Compressed histogram are similar to those in EquiDepthSplitMerge, but must address several additional concerns:

(1) New values added to the relation may be skewed, so that values that did not warrant singleton buckets before may now belong in singleton buckets.
(2) The threshold for singleton buckets grows with N', the number of tuples in equi-depth buckets. Thus values rightfully in singleton buckets for smaller N' may no longer belong in singleton buckets as N' increases.
(3) Because of concerns 1 and 2 above, the number of equi-depth buckets, β', grows and shrinks, and hence we must adjust the equi-depth buckets accordingly.
(4) Likewise, the number of tuples in equi-depth buckets grows and shrinks dramatically as sets of tuples are removed from and added to singleton buckets. The ideal is to maintain N'/β' tuples per equi-depth bucket, but both N' and β' are growing and shrinking.

Briefly and informally, our algorithm addresses each of these four concerns as follows. To address concern 1, we use the fact that a large number of updates to the same value v will suitably increase the count of the equi-depth bucket containing v so as to cause a bucket split. Whenever a bucket is split, if doing so creates adjacent bucket boundaries with the same value v, then we know to create a new singleton bucket for v. To address concern 2, we allow singleton buckets with relatively small counts to be merged back into the equi-depth buckets. As for concerns 3 and 4, we use our procedures for splitting and merging buckets to grow and shrink the number of buckets, while maintaining approximate equi-depth buckets, until we recompute the histogram. The imbalance between the equi-depth buckets is controlled by the thresholds T and T_ℓ (which depend on the tunable performance parameters γ and γ_ℓ, as in EquiDepthSplitMerge). When we convert an equi-depth bucket into a singleton bucket or vice-versa, we ensure that at the time, the bucket is within a constant factor of the average number of tuples in an equi-depth bucket (sometimes additional splits and merges are required). Thus the average is roughly maintained as such equi-depth buckets are added or subtracted.

Figure 10 depicts the new algorithm, denoted **CompressedSplitMerge**.

The requirements for when a bucket can be split or when two buckets can be merged are more involved than in EquiDepthSplitMerge: A bucket B is a *candidate split bucket* if it is an equi-depth bucket with B.count $\geq 2(T_\ell + 1)$ or a singleton bucket such that $T/(2+\gamma) \geq B$.count $\geq 2(T_\ell + 1)$. A pair of buckets, B_i and B_j, is a *candidate merge pair* if (1) either they are adjacent equi-depth buckets or they are a singleton bucket and the equi-depth bucket in which its singleton value belongs, and (2) B_i.count $+ B_j$.count $< T$. When there are more than one candidate split bucket (candidate merge pair), the algorithm selects the one with the largest (smallest combined, resp.) bucket count.

LEMMA 5.1. *Algorithm CompressedSplitMerge maintains the following invariants. (1) For all buckets B, B.count $> T_\ell$. (2) For all equi-depth buckets B, B.count $< T$. (3) All bucket boundaries (B.maxval) are distinct. (4) Any value v belongs to either one singleton bucket, one equi-depth bucket or two adjacent equi-depth buckets (in the latter case, any subsequent inserts or deletes are targeted to the first of the two adjacent buckets).*

Thus the set of equi-depth buckets have counts that are within a factor of $T/T_\ell = (2+\gamma)(2+\gamma_\ell)$, which is a small constant independent of $|R|$.

5.2 Extensions to handle modify and delete operations

We now discuss how to extend CompressedSplitMerge to handle deletions to the database. Deletions can decrease the number of tuples in a bucket relative to other buckets, resulting in a singleton bucket that should be converted to an equi-depth bucket or vice-versa. A deletion can also drop a bucket count to the lower threshold T_ℓ.

Consider a deletion of a tuple τ with $\tau.X = v$ from R. Let B be the bucket in the histogram $\mathcal{H}$ whose interval contains v. We decrement B.count, and if B.count $= T_\ell$, we do the following. If B is part of some candidate merge pair, we merge the pair with the smallest combined count and then split the candidate split bucket B' with the largest count. (Note that B' might be the newly merged bucket.) If no such B' exists, then we recompute $\mathcal{H}$ from the backing sample. Likewise, if B is not part of some candidate merge pair, we recompute $\mathcal{H}$ from the backing sample. As in the insertion-only case, the conversion of buckets from singleton to equi-depth and vice-versa is primarily handled by detecting the need for such conversions when splitting or merging buckets.

For modify operations, we observe as before that if the modify does not change the value of attribute X, or it changes the value of X such that the old value is in the same bucket as the new value, then $\mathcal{H}$ remains unchanged. Else, we update $\mathcal{H}$ by treating the modify as a delete followed by an insert.

The invariants in Lemma 5.1 hold for the version of the algorithm that incorporates these extensions for modify and delete operations.

CompressedSplitMerge()

// R is the relation, A is the attribute of interest, $\mathcal{S}$ is the backing sample.
// $\mathcal{H}$ is the set of $\beta' \geq 1$ equi-depth buckets (sorted by value) and
// $\beta - \beta'$ singleton buckets in the current histogram.
// T is the current threshold for splitting an equi-depth bucket.
// T_ℓ is the current threshold for merging a bucket.
// $\gamma > -1$ and $\gamma_\ell > -1$ are tunable performance parameters.

After an insert of a tuple τ with $\tau.A = v$ into R:
Determine the bucket $B \in \mathcal{H}$ for v;
B.count := B.count $+ 1$;

if B is an equi-depth bucket and B.count $= T$ then
    SplitBucket(B);

SplitBucket(B) // This procedure either splits B or recomputes $\mathcal{H}$ from $\mathcal{S}$.

m := median value among all tuples in $\mathcal{S}$ associated with bucket B.
Let B_p be the bucket preceding B among the equi-depth buckets.

if $m \neq B_p$.maxval and $m \neq B$.maxval then
    if $\exists$ buckets B_i and B_j that are a candidate merge pair then do begin
        B_j.count := B_i.count $+$ B_j.count; // Merge B_i into B_j.
        B_i.maxval := m; B_i.count := $T/2$; B.count := $T/2$; // Split B.
    end;
    else do begin // No suitable merge pair, so recompute $\mathcal{H}$ from $\mathcal{S}$.
        $(\mathcal{H}, \hat{N}', \beta')$:= *CompressedSampleCompute*(); // (see Figure 9).
        T := $(2+\gamma)\cdot \hat{N}'/\beta'$; T_ℓ := $\hat{N}'/((2+\gamma_\ell)\beta')$; // Update thresholds.
    end;
else if $m = B_p$.maxval then do begin
    // Create a singleton bucket for the value m.
    B.count := B_p.count $+$ B.count $- f_m^{\mathcal{S}} \cdot |R|/|\mathcal{S}|$; // First use B for $B \cup B_p - m$.
    B_p.maxval := m; B_p.count := $f_m^{\mathcal{S}} \cdot |R|/|\mathcal{S}|$; // Then use B_p for m.

    if B.count $\geq T$ then // The merged bucket (without m) is too big.
        SplitBucket(B);
    else if B.count $\leq T_\ell$ then do begin // The merged bucket (without m) is too small.
        if $\exists$ buckets B_i and B_j that are a candidate merge pair such that $B = B_i$
         or $B = B_j$ and $\exists$ bucket B_s that is a candidate split bucket then
            B_j.count := B_i.count $+$ B_j.count; *SplitBucket*(B_s); // Merge and split.
        else do begin
            $(\mathcal{H}, \hat{N}', \beta')$:= *CompressedSampleCompute*(); // (see Figure 9).
            T := $(2+\gamma)\cdot \hat{N}'/\beta'$; T_ℓ := $\hat{N}'/((2+\gamma_\ell)\beta')$; // Update thresholds.
        end;
    end;
else if $m = B$.maxval then
    // This case is similar to the previous case, focusing on B and the bucket after it.

Fig. 10. An algorithm for maintaining an approximate Compressed histogram under insertions

6. EXPERIMENTAL EVALUATION

In this section, we experimentally study the effectiveness of our histogram maintenance techniques and their efficiency. First, we describe the experiment testbed.

Database. We model the base data already in the database independently from the update data. Both are modeled using an extensive set of Zipfian [Zipf 1949] data distributions. The z value was varied from 0 to 4 to vary the skew ($z = 0$ corresponds to the *uniform* distribution). The number of tuples (T) in the relation was $100K$ to start with and the number of distinct values (D) was varied from 200 to 1000. Since the exact attribute values do not affect the relative quality of our techniques, we chose the integer value domain. Finally, the frequencies were mapped to the values in different *orders* – decreasing (*decr*), increasing (*incr*), and random (*random*) – thereby generating a large collection of data distributions. We refer to a Zipf distribution with the parameter z and order x as the $zipf(z, x)$ distribution.

Histograms. The equi-depth and Compressed histograms consisted of 20 buckets and were computed from a sample of 2000 tuples, which was also the size of the backing sample.

Updates. We used three classes of updates, described below, based on the mix of insert, delete, and modify operations. In each case, the update data was taken from a Zipf distribution. By varying the z parameter, we can vary the skew in the updates. The number of updates was increased up to $400K$ (four times the relation size).

(1) *Insert:* The first class of updates consists of just insert operations. Since our algorithms are most efficient for such an environment, they are studied in most detail.

(2) *Warehouse:* This class contains an alternating sequence of a set of inserts followed by a set of deletes. This pattern is common in data warehouses keeping transactional information during sliding time windows (loading fresh data and discarding very old data, when loaded close to capacity).

(3) *Mixed:* This class contains a uniform mixture of insert, delete, and modify operations occurring in random order.

Unless otherwise specified, experimental results are for the *Insert* class of updates.

Techniques. We studied several variants of old and new techniques which are described below in terms of their operations for a single insert (operations for delete are similar in principle).

(1) *Fixed-Histogram:* The sum of frequencies in each bucket is incremented by $\frac{1}{\beta}$ so that the total sum of the frequencies increases by 1. This is essentially the technique in use in nearly all systems prior to our work, in that they update the number of tuples but do not update the histogram.

(2) *Periodic-Sample-Compute:* This (expensive) technique requires recomputing the histogram from the backing sample after each insertion into the sample, while the total sum of frequencies is incremented as in the above technique.

(3) *SplitMerge:* This is the class of techniques corresponding to the algorithms proposed in this paper.

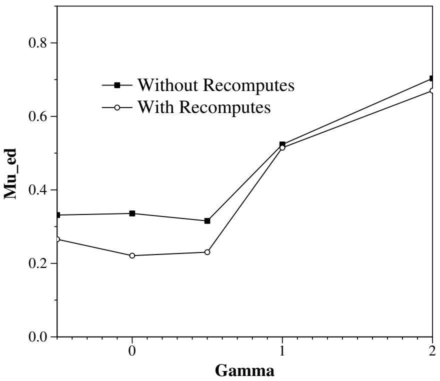

Fig. 11. Effect of γ and recomputation on μ_{ed} errors

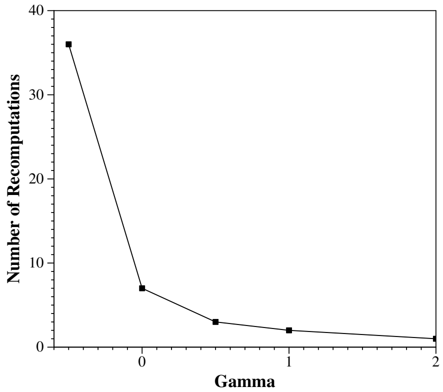

Fig. 12. Effect of γ on the number of recomputations

(4) *No-Recompute:* This technique differs from SplitMerge by not performing the recomputations and simply increasing the split threshold when a merge can not be performed.

(5) *Fixed-Buckets:* This technique differs from SplitMerge by not attempting to split any bucket. But, unlike the Fixed-Histogram algorithm, the size of the bucket containing the inserted value is correctly incremented.

Error metrics. The following error metrics are used: μ_{count} (Eq. 1), μ_{ed} (Eq. 2 and 3), and μ_{hb} (Eq. 4). In addition a new metric μ_{range} is defined, which captures the accuracy of histograms in estimating the result sizes of range predicates (of the form $X \leq a$). The query-set contains range predicates over all possible values in the joint value domain. For each query, we find the error as a percentage of the result size. μ_{range} is defined as the average of these errors over the query-set.

All our experiments were conducted five times to reduce the accidental effects of samples and had similar results in each instance. Hence, we present the results of one of the runs.

6.1 Effects of recomputation and γ

Figure 11 depicts the errors (μ_{ed}) of the equi-depth histogram obtained at the end of $400K$ insertions as a function of γ, under the SplitMerge and No-Recompute techniques. The base data distribution for this case was *uniform* and the update distribution was *zipf(2,decr)*. It is clear that SplitMerge outperforms the technique without recomputations. Also, the errors due to the techniques are lowest for low values of γ and increase rapidly as γ increases. This is because for low values of γ, the histogram is recomputed more often and the bucket sizes do not exceed a low threshold, thus keeping the μ_{ed} small.

On the other hand, small values of γ result in a larger number of disk accesses (for the backing sample). Figure 12 shows the effect of γ on the number of recomputations. It is clear that too small values of γ result in a large number of recomputations. Based on similar sets of experiments conducted over the entire set of data distributions, we concluded that $\gamma = 0.5$ is a reasonable value for limiting the number of computations as well as for decreasing errors; we use this setting in all the remaining experiments.

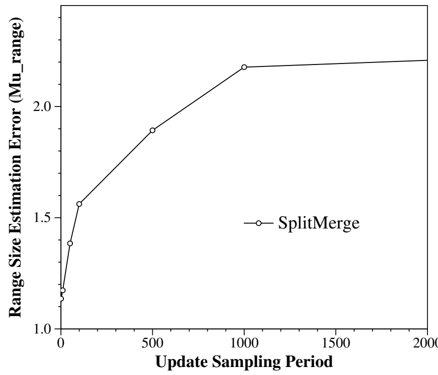

Fig. 13. Effect of update sampling

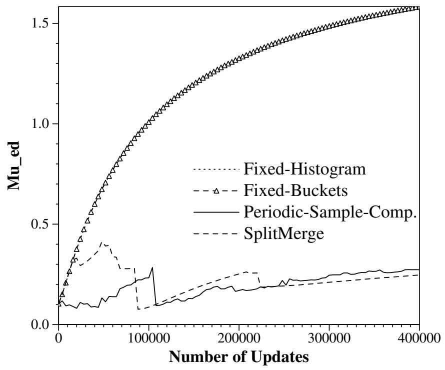

Fig. 14. μ_{ed} errors (equi-depth histograms)

6.2 Update sampling

Nearly all the experiments in this paper were conducted by considering every insertion in the database. In some update-intensive databases this could result in intolerable performance degradation. Hence we propose uniformly sampling the updates with a certain probability and modifying the histograms only for the sampled updates. In this experiment, we study the effect of the update sampling probability on histogram performance. The base and update distributions are chosen to be *zipf(1,incr)* and *zipf(0.5, random)* respectively, and the histogram is Compressed. Figure 13 depicts the errors due to the SplitMerge technique for various sampling probabilities. The x-axis represents the average number of updates that are skipped and the y-axis represents the errors incurred by the histogram resulting at the end of $400K$ inputs in estimating the result sizes of range queries (μ_{range}). It is clear from this figure that the accuracy depends on the number of updates sampled; as long as not too many updates are skipped (say, at most 100 in this experiment), the errors are reasonably small.

6.3 Approximation of equi-depth histograms

We compare the effectiveness of various techniques in approximating equi-depth histograms under insertions into the database. The results are presented for *uniform* base data and *zipf(2,incr)* update data and are fairly consistent over most other combinations. Figures 14 through 16 depict various error measures as a function of the number of insertions. For this experiment, the SplitMerge technique performed just 2 recomputations from the backing sample while Periodic-Sample-Compute performed 3276.

It is clear from Figure 14 that the SplitMerge technique is nearly identical to the more expensive Periodic-Sample-Compute technique in maintaining the histogram close to equi-depth. The Periodic-Sample-Compute technique does not maintain a perfectly equi-depth histogram because it is recomputed from the backing sample which may not reflect all the insertions. The other two techniques clearly result in a very poor equi-depth histogram because they do not perform any splits of the over-populated buckets. Figure 15 shows that the SplitMerge and Fixed-Buckets techniques are very accurate in reflecting the accurate counts, because their bucket sizes are correctly updated after every insertion. For the other two techniques, the size of a bucket is always equal to N/β, hence the μ_{count} and μ_{ed} measures are identical. Finally, it is clear from Figure 16 that the SplitMerge technique

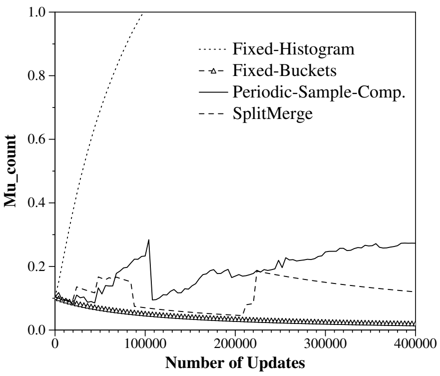

Fig. 15. μ_{count} errors (equi-depth histograms)

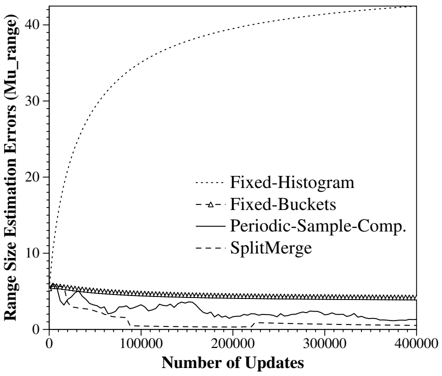

Fig. 16. μ_{range} errors (equi-depth histograms)

offers the best performance in estimating range query result sizes as well.

6.4 Approximation of Compressed histograms

We compare the effectiveness of various techniques in maintaining approximating Compressed histograms. The base data distribution is *zipf(1,incr)* (a skewed distribution was chosen so that the Compressed histogram will contain a few high-biased buckets) and the update distribution is *zipf(2,random)*, which introduces skew at different points in the relation's distribution. Figures 17 and 18 depict the μ_{hb} and μ_{range} errors on the y-axes respectively and the number of insertions on the x-axes. The results for the other two metrics are similar to the equi-depth case and consistently demonstrate the accuracy of the SplitMerge technique, hence are not presented. Once again, the SplitMerge technique performed just 2 recomputations from the sample, while Periodic-Sample-Compute performed 3274 recomputations.

It can be seen from Figure 17 that the Periodic-Sample-Compute and SplitMerge techniques result in almost zero errors in capturing the high frequency values in the updated relation, even when these values were not frequent in the base relation. In the beginning, the updates do not create a new high frequency value and all techniques perform well. But once a new value becomes frequent, it is clear that the other two techniques fail to characterize it as such and hence incur high errors.

Figure 18 shows that the errors in range size estimation follow the similar pattern as the equi-depth case. Also, as expected from our earlier work [Poosala et al. 1996], the Compressed histograms are observed to incur smaller errors than the equi-depth histograms from Figure 16.

6.5 Effect of skew in the updates

High skew in the update data can alter the overall data distribution dramatically, and hence requires effective histogram maintenance techniques. In Figure 19 we depict the performance of various Compressed histograms resulting from the techniques at the end of $400K$ insertions to the database. The x-axis represents the z parameter values and the y-axis represents the errors in estimating range query result sizes (μ_{range}). The Fixed-Histogram technique fails very quickly because it assumes that the updates are uniform and hence does not update the high-biased part correctly. It is clear from this figure that the Split-

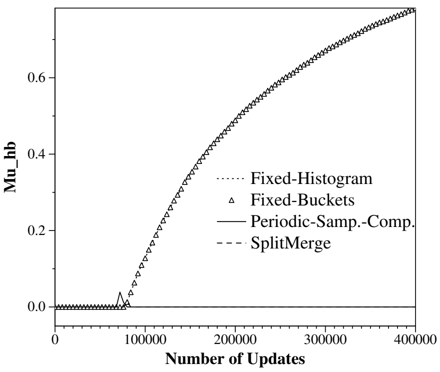

Fig. 17. μ_{hb} errors (Compressed histograms)

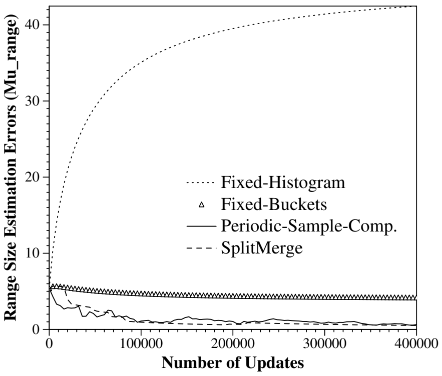

Fig. 18. μ_{range} errors (Compressed histograms)

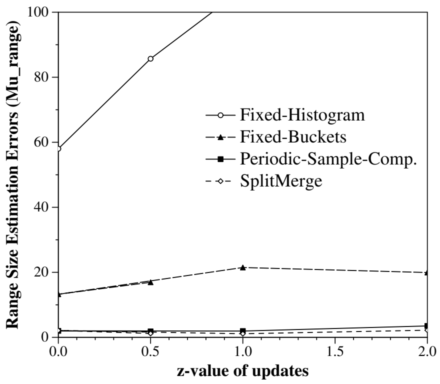

Fig. 19. Effect of skew in the updates

Merge technique performs consistently well for all levels of skew and is always better than the other techniques, because it approximates the equi-depth part well using splits and recomputations, and approximates the high-biased part well by dynamically detecting high-frequency values.

6.6 Effect of update nature

The updates in all the experiments studied thus far consisted of inserts only (the *Insert* update set). In this section we study the performance of various maintenance techniques in the presence of delete and modify operations. The performance of Compressed histograms maintained using various maintenance techniques is depicted in Figures 20 and 21 for the *Warehouse* and *Mixed* update sets respectively. These graphs show range size estimation errors as a function of the number of updates. The same conclusions as in the previous experiments were derived for other error metrics and for equi-depth histograms. Hence, we do not present those results here.

Note that the Periodic-Sample-Compute and SplitMerge techniques successfully limit the range size estimation errors to a small constant value in the presence of both kinds of updates. On the other hand, depending on the fluctuating relation size, the errors due to

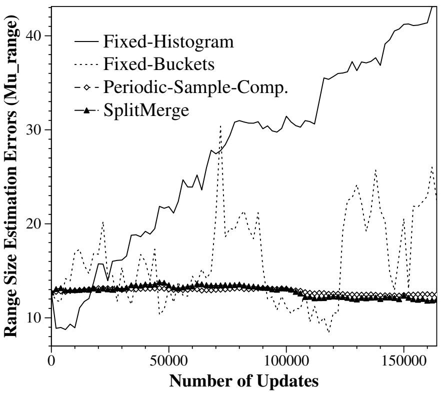

Fig. 20. Errors under Warehouse updates

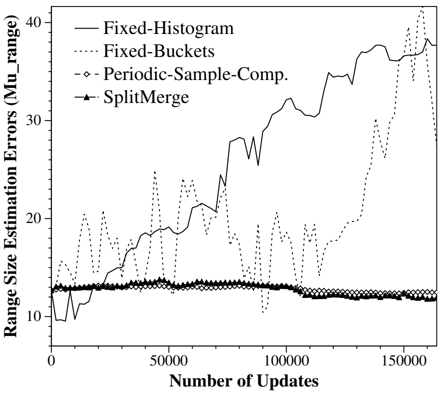

Fig. 21. Errors under Mixed updates

Fixed-Buckets increase and decrease, but overall they increase because the approximate histogram deviates too far from the actual data distribution. The performance of Fixed-Buckets differs from the experiments on Insert data because the insert, delete, and modify streams are skewed at different attribute values and hence vary the data distribution drastically and require significant bucket changes in order to capture it accurately. As in the earlier experiments, Fixed-Histogram performs poorly because it fails to capture the varying shape of the distribution.

6.7 Practicality considerations

In this section we discuss the costs of using our histogram maintenance techniques in a DBMS. Our techniques require the following resources:

CPU. For most updates the only CPU-intensive operation one needs to perform is incrementing the bucket count. This is further reduced by sampling the updates. Complete recomputations of histograms from the backing sample are more expensive (on the order of tens of milliseconds [Poosala et al. 1996]), but happen rarely (twice during a five-fold increase in the size of a database). Also, since histograms are no longer read-only data structures, one now needs some form of concurrency control mechanism for accessing them. We suggest using inexpensive latches for updating the histograms and allowing inconsistent reads (which is often fine in an estimation application).

I/O. For inserts, the backing sample on the disk is accessed only during splits, sample updates, and recomputations. In the last case, the entire backing sample has to be accessed, which may require fetching a small number of disk pages, but this is very rare. As shown in our theorems, one needs a very large number of inserts in order to perform a split, hence the split costs are also quite negligible. Similarly, for a large relation, the backing sample is also updated very rarely. On the other hand, our techniques for arbitrary delete and modify operations require accessing the backing sample on every sampled update, which may make the techniques expensive in some environments. For delete operations in a data warehouse environment, which house transactional information for sliding time windows, the techniques are I/O effective.

Space. Our techniques do not require any additional data in memory other than what is already present in a histogram. On the other hand, they need disk space (on the order of

1–10 pages) for the backing sample. In comparison with typical relation and disk sizes, this is clearly negligible (note Figure 1 for an illustration).

Overall, the above argues qualitatively that the resource requirements of the maintenance algorithms are negligibly small in most situations. Further quantitative studies in real database systems are needed in order to measure the overheads occurring in practice.

7. CONCLUSIONS

This paper proposed a novel approach for maintaining histograms and samples up-to-date in the presence of updates to the database. This is critical for various DBMS components (primarily query optimizers) that rely on estimates requiring information about the current data. Algorithms were proposed for the widely used equi-depth histograms and the highly accurate class of Compressed histograms. We introduced the following innovations:

—The notion of a *backing sample*, with its advantages over previous approaches to obtaining samples, and techniques for its maintenance.
—The idea of maintaining histograms incrementally by making use of the backing sample. The backing sample can be much larger than the histogram and reside on the disk; it is accessed very rarely in support of the histogram, which is typically in main memory.
—Split and merge techniques on histogram buckets, which drastically reduce accesses to the disk for the backing sample.

Next, we conducted a large set of experiments to demonstrate the effectiveness of our algorithms in maintaining histograms. Our conclusions are as follows:

—The new techniques are very effective in approximating equi-depth and Compressed histograms. They are equally effective for relations orders of magnitude larger. In fact, as the relation size grows, the relative overhead of maintaining a backing sample with equal accuracy becomes even smaller.
—Very few recomputations from the backing sample are incurred for a large number of updates, proving that our split&merge techniques are quite effective in minimizing the overheads due to recomputation.
—The experiments clearly show that histograms maintained using these techniques remain highly effective in result size estimation, unlike the previous approaches.

The CPU, I/O and storage requirements for these techniques are negligible for insert-mostly databases and for data warehousing environments.

Based on our results, we recommend that these techniques be used in most DBMSs, for effective incremental maintenance of approximate histograms.

APPENDIX

A. PROOFS FROM SECTION 3

In this section, we prove the correctness of MaintainBackingSample, as well as various properties about the algorithm. An important assumption we use is that the sequence of database operations is independent of the random choices made by our algorithm.

Let S be a set of size N. A *sample* of size $n \leq N$ (without replacement) from S is a subset of size n of the elements in S. There are $\binom{N}{n}$ possible samples of S of size n. A *random* sample of size n is a sample of S selected with probability $1/\binom{N}{n}$.

Our algorithm treats database insertions as in Vitter's algorithm [Vitter 1985], so we will use the following fact shown in [Vitter 1985], restated using the terminology of this paper:

OBSERVATION A.1. *Let S be a set of size N, and let x be an element not in S. Let S_1 be a random sample of size n of S, and let u be an element selected uniformly at random from S_1. Let S_2 be constructed as follows:*

$$S_2 = \begin{cases} S_1 + \{x\} - \{u\} & \textit{with probability } \frac{n}{(N+1)} \\ S_1 & \textit{otherwise} \end{cases}$$

Then S_2 is a random sample of size n of $S + \{x\}$.

Next consider deletions. For an element just deleted from the relation R, if the element is in the backing sample, $\mathcal{S}$, our algorithm deletes it from $\mathcal{S}$, else it leaves $\mathcal{S}$ unchanged. Lemma A.2 establishes that this maintains the property that S is a random sample.

LEMMA A.2. *Let S be a set of size t, and let y be an element in S. Let S_1 be a random sample of size s of S. Then if y is not in S_1 then $S_2 = S_1 - \{y\} = S_1$ is a random sample of size s of $S - \{y\}$. Else if y is in S_1 then $S_2 = S_1 - \{y\}$ is a random sample of size $s - 1$ of $S - \{y\}$.*

PROOF. In the former case (y is not in S_1), there are $\binom{t-1}{s}$ possible samples of size s of S that do not contain y, each of which is equally likely to be selected for S_1. Since $S_2 = S_1$, there is a 1-1 correspondence between samples S_1 not containing y and samples S_2. Thus, for any S_2, $\Pr[S_2 \text{ selected}] = \Pr[S_1 \text{ selected} \mid \text{select a sample without } y] = \Pr[S_1 \text{ selected}] / \Pr[\text{ select a sample without } y]$, which equals

$$\frac{\frac{1}{\binom{t}{s}}}{\frac{\binom{t-1}{s}}{\binom{t}{s}}} = \frac{1}{\binom{t-1}{s}} \quad .$$

In the latter case (y is in S_1), there are $\binom{t}{s} - \binom{t-1}{s} = \binom{t-1}{s-1}$ possible samples size s of S that contain y, each of which is equally likely to be selected for S_1. There is a 1-1 correspondence between samples S_1 containing y and samples S_2. Thus, for any S_2, the probability S_2 is selected is $1/\binom{t-1}{s-1}$. ☐

We now proceed to prove Theorem 3.1.

PROOF OF THEOREM 3.1. Consider a sequence of insert, modify and delete operations for an initially empty relation R. Let S be the sample resulting from applying Maintain-BackingSample to the sequence, with a given L and U, $1 \leq L \leq U$. We first prove the claim that the ids in S are a random sample of the ids in R.

The proof is by induction on the length, k, of the sequence. For a sequence of length k, let R_k be the set of ids in R resulting from applying the updates in the sequence to an initially empty relation R, and let S_k be the set of ids in S resulting from applying the algorithm in response to the sequence.

For the base case $k = 1$, the first update must be an insert for some id. This id is added to S, so $S_1 = \{id\}$ is a random sample of $R_1 = \{id\}$.

Assume the claim is true for $k \geq 1$, and consider an arbitrary sequence, A_{k+1}, of length $k+1$. The sequence A_{k+1} consists of a sequence A_k of length k followed by a single update operation (either an insert, a modify or a delete). Let R_k and S_k be defined according to A_k.

First, consider the case where the $(k+1)$st update is an insert of a new element id. Thus $R_{k+1} = R_k + \{id\}$. If $|S_k| + 1 = |R_{k+1}| \leq U$ then $|S_k| = |R_k|$ and both sets contain the same ids. By the algorithm, $S_{k+1} = S_k + \{id\}$, and the claim holds by the inductive assumption. Else the claim holds by the inductive assumption and Observation A.1.

Second, consider the case where the $(k+1)$st update is a modify of one or more of the values of an element id in R_k. By the algorithm, $R_{k+1} = R_k$ and $S_{k+1} = S_k$, so the claim holds by the inductive assumption.

Third, consider the case where the $(k+1)$st update is a delete of an element id in R_k. If $id \notin S_k$, then the claim holds by the inductive assumption and Lemma A.2. If $id \in S_k$, then also by the inductive assumption and Lemma A.2, the ids in $S_k - \{id\}$ are a random sample of the ids in R_{k+1}. If the algorithm scans R_{k+1}, then consider some sequence, A', of $|R_{k+1}| < k$ inserts, one insert for each element in R_{k+1}. By the inductive assumption applied to A', the ids in S_{k+1} are a random sample of the ids in R_{k+1}.

Since in all three cases, the claim holds for an arbitrary A_{k+1}, the claim is maintained for all sequences of length $k+1$, and hence by induction holds for all finite length sequences.

Finally, consider the relation R after an arbitrary sequence of update operations and a sample S generated by the algorithm. Since the set of ids in S is a random sample of the set of ids in R, and the updates are independent of the random choices made by the algorithm, then for any attribute X of R, the set $V_s = \bigcup_{id \in S} id.X$ of values is a random sample of the set $V_r = \bigcup_{id \in R} id.X$ of values. The theorem follows. □

MaintainBackingSample maintains a backing sample, $\mathcal{S}$, such that $\min(|R|, L) \leq |\mathcal{S}| \leq U$, where L and U are prespecified upper and lower bounds. It populates $\mathcal{S}$ up to U elements and then a series of $U - L + 1$ deletes of sample elements are needed in order to force the algorithm to rescan the relation R in order to re-populate $\mathcal{S}$. The next lemma shows that rescans are expected to be infrequent for large relations.

LEMMA A.3. *Consider an initial relation R and a backing sample, $\mathcal{S}$, for R of size U. Consider any sequence of updates to R and let $N_0 \geq U$ be a lower bound on the size of R after each such update. (There is no upper bound imposed on $|R|$.) Then at most 1 rescan is expected every $N_0(U - L + 1)/U$ updates.*

PROOF. Initially, $|\mathcal{S}| = U$, so a series of $U - L + 1$ deletes of sample elements are needed in order to force a rescan of R. At any time, the probability that an id selected for deletion is in $\mathcal{S}$ is $|\mathcal{S}|/|R|$ (since $\binom{|R|-1}{|\mathcal{S}|-1}/\binom{|R|}{|\mathcal{S}|} = |\mathcal{S}|/|R|$). Since $|S|/|R| \leq U/N_0$, the expected number of deletions needed is at least $N_0(U - L + 1)/U$. After the rescan, $|\mathcal{S}| = U$, and the argument can be repeated. □

As an example, consider a relation of size $N = 2N_0$ and $L = U/2 + 1$. Then in order to force a rescan, we must delete half of the relation. Moreover, in such cases, the number of tuples to be rescanned is $N/2$, which can be amortized against the $N/2$ deletions needed to force the rescan.

B. PROOFS FROM SECTION 4

To simplify the presentation of the proofs that follow, we will ignore the use of floors and ceilings in the algorithms.

PROOF OF THEOREM 4.1. Since each bucket count is set to N/β, $\mu_{\text{ed}} = \mu_{\text{count}}$ for this algorithm. We assume without loss of generality that all values are distinct; this can be accomplished by appending to each original value a unique label.

The probability that sampling with replacement will pick m distinct elements is

$$\frac{N(N-1)(N-2)\cdots(N-m+1)}{N^m} \geq \left(\frac{N-m}{N}\right)^m = \left(1-\frac{m}{N}\right)^m \geq 1-\frac{m^2}{N} \quad .$$

Since $m \leq N^{1/3}$, the probability is at least $1 - 1/N^{1/3}$. Thus we will ignore the difference between sampling with and without replacement by considering whichever one is more suited to the analysis, and compensate by subtracting $N^{-1/3}$ from the probability of obtaining the stated bounds. (This can be argued formally using conditioned events.)

By Lemma 7.1 in [Reif and Valiant 1987], for each bucket B_i,

$$\Pr\left\{ f_{B_i} \in \left[\left(1 \pm \left(\frac{m}{\beta}\right)^{-\frac{1}{6}}\right)\frac{N}{\beta}\right]\right\} \geq 1 - \frac{2}{\sqrt{c}\ln\beta}\beta^{-\sqrt{c}} \quad ,$$

which is greater than $1-\beta^{-\sqrt{c}}$. Thus f_{B_i} is within the above range for all i with probability at least $1-\beta^{-(\sqrt{c}-1)}$. This implies that, with the same probability,

$$\mu_{\text{ed}} \leq \frac{\beta}{N}\sqrt{\frac{1}{\beta}\sum_{i=1}^{\beta}\left(\frac{N}{\beta}\left(\frac{m}{\beta}\right)^{-\frac{1}{6}}\right)^2} = \left(\frac{m}{\beta}\right)^{-\frac{1}{6}} \quad .$$

The theorem follows. □

PROOF OF THEOREM 4.2. Consider some phase in the EquiDepthSimple algorithm, at which the relation is of size N. At the beginning of the phase, let N' be the size of the relation, and let μ'_{count} and μ'_{ed} be the errors μ_{count} and μ_{ed}, respectively. Let $\rho' = 1-\beta^{-(\sqrt{c}-1)} - (N')^{-1/3}$, and let $\rho = 1-\beta^{-(\sqrt{c}-1)} - (N/(2+\gamma))^{-1/3}$. Since during a phase $N \leq N'(2+\gamma)$, we have $\rho \leq \rho'$. By Theorem 4.1, $\mu'_{\text{ed}} = \mu'_{\text{count}} \leq \alpha$ with probability at least ρ', and hence at least ρ.

During a phase, a value inserted to bucket B_i increments both f_{B_i} and B_i.count. Therefore, by the definition of μ_{count} (Eq. 1), its value does not change during a phase, and hence at any time during the phase $\mu_{\text{count}} = \mu'_{\text{count}} \leq \alpha$ with probability ρ. It remains to bound μ_{ed}.

Let f'_{B_i} and B_i.count$'$ be the values of f_{B_i} and B_i.count, respectively, at the beginning of the phase. Let $\Delta'_i = f'_{B_i} - N'/\beta$, and let $\Delta_i = f_{B_i} - N/\beta$. We claim that $|\Delta_i - \Delta'_i| \leq (1+\gamma)N'/\beta$. Note that $|\Delta_i - \Delta'_i| \leq \max(f_{B_i} - f'_{B_i}, N/\beta - N'/\beta)$. The claim follows since $f_{B_i} - f'_{B_i} = B_i.\text{count} - B_i.\text{count}' \leq T - B_i.\text{count}' = (2+\gamma)N'/\beta - N'/\beta$, and $N - N' \leq \beta(B_i.\text{count} - B_i.\text{count}')$.

By the claim,

$$\Delta_i{}^2 \leq \left(\Delta'_i + (1+\gamma)N'/\beta\right)^2 = \Delta'^2_i + 2\Delta'_i(1+\gamma)N'/\beta + ((1+\gamma)N'/\beta)^2 \quad .$$

Note that $\sum_{i=1}^{\beta}\Delta'_i = \sum_{i=1}^{\beta}(f_{B_i} - N'/\beta) = 0$. Hence, substituting for Δ_i^2 in the definition of μ_{ed} (Eq. 2) we obtain

$$\mu_{\text{ed}} = \frac{\beta}{N}\sqrt{\frac{1}{\beta}\left(\sum_{i}^{\beta}\Delta'^2_i + \sum_{i=1}^{\beta}((1+\gamma)N'/\beta)^2\right)} \leq \mu'_{\text{ed}} + \frac{\beta}{N}(1+\gamma)N'/\beta \leq \mu'_{\text{ed}} + (1+\gamma) \quad .$$

The theorem follows. □

PROOF OF LEMMA 4.5. Let N' be the total number of elements at the beginning of a phase t. Note that the sum of the bucket counts at the start of phase t is N'. Each new element increases this sum by one, and both splitting and merging have no effect on this sum. Thus throughout phase t, the sum of the bucket counts is always exactly the number of elements.

Consider first the case that $\gamma > 0$. Recall that a phase ends when there is no pair of adjacent buckets B_i and B_{i+1} such that B_i.count + B_{i+1}.count $< T$. Therefore, summing over the pairs, $\{B_{2j-1}, B_{2j}\}$ for $j = 1, 2, \ldots, \beta/2$ we obtain that the sum of the bucket counts (and hence the total number of elements) at the end of phase t is at least $(\beta/2) \cdot T = \beta/2 \cdot (2 + \gamma) \cdot N'/\beta = (1 + \gamma/2)N'$.

For the case $-1 < \gamma \leq 0$ we note that a bucket can get to be of size T only after getting $(1+\gamma)N'/\beta$ new elements. Therefore, the total number of elements at the end of the phase is at least $(1 + (1 + \gamma)/\beta)N'$.

Thus in either case, the number of phases after N inserts is at most $\log_\alpha N$. The lemma follows because the number of phases is also upper bounded by the number of inserts. □

ACKNOWLEDGMENTS

This work was performed while all three authors were with the Information Sciences Research Center, Bell Laboratories. We acknowledge the contributions of Andy Witkowski to the algorithm for maintaining approximate equi-depth histograms. We also thank Nabil Kahale and Sridhar Rajagopalan for discussions related to this work. Finally, we thank the anonymous referees' for their helpful comments on improving the paper.

REFERENCES

ABOULNAGA, A. AND CHAUDHURI, S. 1999. Self-tuning histograms: building histograms without looking at data. In *Proc. ACM SIGMOD International Conf. on Management of Data*. ACM, Philadelphia, PA, 181–192.

ACHARYA, S., GIBBONS, P. B., AND POOSALA, V. 2000. Congressional samples for approximate answering of group-by queries. In *Proc. ACM SIGMOD International Conf. on Management of Data*. ACM, Dallas, TX, 487–498.

ACHARYA, S., GIBBONS, P. B., POOSALA, V., AND RAMASWAMY, S. 1999. Join synopses for approximate query answering. In *Proc. ACM SIGMOD International Conf. on Management of Data*. ACM, Philadelphia, PA, 275–286.

BLOHSFELD, B., KORUS, D., AND SEEGER, B. 1999. A comparison of selectivity estimators for range queries on metric attributes. In *Proc. ACM SIGMOD International Conf. on Management of Data*. ACM, Philadelphia, PA, 238–250.

BRUNO, N., CHAUDHURI, S., AND GRAVANO, L. 2001. Stholes: A multidimensional workload-aware histogram. In *Proc. ACM SIGMOD International Conf. on Management of Data*. ACM, Santa Barbara, CA, 211–222.

CHAUDHURI, S., DAS, G., AND NARASAYYA, V. 2001. A robust, optimization-based approach for approximate answering of aggregate queries. In *Proc. ACM SIGMOD International Conf. on Management of Data*. ACM, Santa Barbara, CA, 295–306.

CHAUDHURI, S., MOTWANI, R., AND NARASAYYA, V. 1998. Random sampling for histogram construction: How much is enough? In *Proc. ACM SIGMOD International Conf. on Management of Data*. ACM, Seattle, WA, 436–447.

CHRISTODOULAKIS, S. 1984. Implications of certain assumptions in database performance evaluation. *ACM TODS 9,* 2 (June), 163–186.

DESHPANDE, A., GAROFALAKIS, M., AND RASTOGI, R. 2001. Independence is good: Dependency-based histogram synopses for high-dimensional data. In *Proc. ACM SIGMOD International Conf. on Management of Data*. ACM, Santa Barbara, CA, 199–210.

GANTI, V., LEE, M.-L., AND RAMAKRISHNAN, R. 2000. ICICLES: self-tuning samples for approximate query answering. In *Proc. 26th International Conf. on Very Large Data Bases*. Morgan Kaufman, Cairo, Egypt, 176–187.

GIBBONS, P. B. 2001. Distinct sampling for highly-accurate answers to distinct values queries and event reports. In *Proc. 27th International Conf. on Very Large Data Bases*. Morgan Kaufman, Rome, Italy, 541–550.

GIBBONS, P. B. AND MATIAS, Y. 1998. New sampling-based summary statistics for improving approximate query answers. In *Proc. ACM SIGMOD International Conf. on Management of Data*. ACM, Seattle, WA, 331–342.

GILBERT, A., GUHA, S., INDYK, P., KOTIDIS, Y., MUTHUKRISHNAN, S., AND STRAUSS, M. J. 2002. Fast small-space algorithms for approximate histogram maintenance. In *Proc. 34th ACM Symp. on the Theory of Computing*. ACM, Montréal, Québec, Canada.

GILBERT, A. C., KOTIDIS, Y., MUTHUKRISHNAN, S., AND STRAUSS, M. J. 2002. How to summarize the universe: Dynamic maintenance of quantiles. In *Proc. 28th International Conf. on Very Large Data Bases*. Morgan Kaufman, Hong Kong.

GREENWALD, M. AND KHANNA, S. 2001. Space-efficient online computation of quantile summaries. In *Proc. ACM SIGMOD International Conf. on Management of Data*. ACM, Santa Barbara, CA, 58–66.

GUHA, S., KOUDAS, N., AND SHIM, K. 2001. Data-streams and histograms. In *Proc. 33rd ACM Symp. on Theory of Computing*. ACM, Hersonissos, Crete, Greece, 471–475.

GUNOPULOS, D., KOLLIOS, G., TSOTRAS, V. J., AND DOMENICONI, C. 2000. Approximating multi-dimensional aggregate range queries over real attributes. In *Proc. ACM SIGMOD International Conf. on Management of Data*. ACM, Dallas, TX, 463–474.

IOANNIDIS, Y. AND CHRISTODOULAKIS, S. 1991. On the propagation of errors in the size of join results. In *Proc. of the ACM SIGMOD Conf. on Management of Data*. ACM, Denver, CO, 268–277.

IOANNIDIS, Y. AND POOSALA, V. 1999. Histogram-based techniques for approximating set-valued query-answers. In *Proc. 25th International Conf. on Very Large Databases*. Morgan Kaufman, Edinburgh, Scotland, UK, 174–185.

JAGADISH, H. V., KOUDAS, N., MUTHUKRISHNAN, S., POOSALA, V., SEVCIK, K., AND SUEL, T. 1998. Optimal histograms with quality guarantees. In *Proc. 24th International Conf. on Very Large Data Bases*. Morgan Kaufman, New York, NY, 275–286.

KONIG, A. C. AND WEIKUM, G. 1999. Combining histograms and parametric curve fitting for feedback-driven query result-size estimation. In *Proc. 25th International Conf. on Very Large Databases*. Morgan Kaufman, Edinburgh, Scotland, UK, 423–434.

KOOI, R. P. 1980. The optimization of queries in relational databases. Ph.D. thesis, Case Western Reserver University.

LIPTON, R. J., NAUGHTON, J. F., AND SCHNEIDER, D. A. 1990. Practical selectivity estimation through adaptive sampling. In *Proc. of the ACM SIGMOD Conf. on Management of Data*. ACM, Atlantic City, NJ, 1–11.

MATIAS, Y., VITTER, J. S., AND WANG, M. 1998. Wavelet-based histograms for selectivity estimation. In *Proc. ACM SIGMOD International Conf. on Management of Data*. ACM, Seattle, WA, 448–459.

MATIAS, Y., VITTER, J. S., AND WANG, M. 2000. Dynamic maintenance of wavelet-based histograms. In *Proc. 26th International Conf. on Very Large Data Bases*. Morgan Kaufman, Cairo, Egypt, 101–110.

POOSALA, V. 1997. Histogram-based estimation techniques in database systems. Ph.D. thesis, Univ. of Wisconsin-Madison.

POOSALA, V. AND IOANNIDIS, Y. 1996. Estimation of query-result distribution and its application in parallel-join load balancing. In *Proc. of the 22nd Int. Conf. on Very Large Databases*. VLDB, Bombay, India, 448–459.

POOSALA, V. AND IOANNIDIS, Y. 1997. Selectivity estimation without the attribute value independence assumption. In *Proc. 23rd International Conf. on Very Large Data Bases*. Morgan Kaufman, Athens, Greece, 486–495.

POOSALA, V., IOANNIDIS, Y., HAAS, P., AND SHEKITA, E. 1996. Improved histograms for selectivity estimation of range predicates. In *Proc. of the ACM SIGMOD Conf. on Management of Data*. ACM, Montreal, Quebec, Canada, 294–305.

REIF, J. H. AND VALIANT, L. G. 1987. A logarithmic time sort for linear size networks. *Journal of the ACM 34,* 1, 60–76.

SELINGER, P. G., ASTRAHAN, M. M., CHAMBERLIN, D. D., LORIE, R. A., AND PRICE, T. T. 1979. Access path selection in a relational database management system. In *Proc. of the ACM SIGMOD Conf. on Management of Data*. ACM, Boston, MA, 23–34.

VITTER, J. S. 1985. Random sampling with a reservoir. *ACM Trans. Math. Software 11*, 37–57.

ZIPF, G. K. 1949. *Human behaviour and the principle of least effort*. Addison-Wesley, Reading, MA.

RECEIVED month year; REVISED month year; ACCEPTED month year