Aqua Project White Paper

Phillip B. Gibbons* Yossi Matias† Viswanath Poosala‡

Information Sciences Research Center
Bell Laboratories
600 Mountain Avenue
Murray Hill NJ 07974

December 30, 1997

Abstract

In large data recording and warehousing environments, it is often advantageous to provide fast, approximate answers to queries, whenever possible. The goal is to provide an estimated response in orders of magnitude less time than the time to compute an exact answer, by avoiding or minimizing the number of accesses to the base data.

This white paper describes the Approximate QUery Answering (AQUA) Project underway in the Information Sciences Research Center at Bell Labs. We present a framework for an approximate query engine that observes new data as it arrives and maintains small synopsis data structures on that data. These data structures are used to provide fast, approximate answers to a broad class of queries. We describe metrics for evaluating approximate query answers. We also present new synopsis data structures, and new techniques for approximate query answers. We report on the goals and status of the Aqua project, and plans for future work.

*Email: gibbons@research.bell-labs.com.

†Current address is Tel-Aviv University, Ramat Aviv, Tel-Aviv 69978 Israel. Email: matias@math.tau.ac.il.

‡Email: poosala@research.bell-labs.com.

1 Introduction

Traditional query processing has focused solely on providing exact answers to queries, in a manner that seeks to minimize response time and maximize throughput. However, there are a number of environments for which the response time for an exact answer is often slower than is desirable. First, in large data recording and warehousing environments, providing an exact answer to a complex query can take minutes to hours, due to the amount of disk I/O required. For environments with terabytes or more of data, even a single scan of the data can take tens of minutes[1]. Second, in distributed data recording and warehousing environments, some of the data may be remote, resulting in slow response times, and may even be currently unavailable, so that an exact answer is not an option until the data again becomes available [FJS97]. Finally, in environments with stringent response time requirements, even a single access at a particular level of the storage hierarchy may be unacceptably slow, e.g., for sub-millisecond response time, a single disk access is too slow.

Environments for which providing an exact answer results in undesirable response times motivate the study of techniques for providing *approximate* answers to queries. The goal is to provide an estimated response in orders of magnitude less time than the time to compute an exact answer, by avoiding or minimizing the number of accesses to the base data.

There are a number of scenarios for which an exact answer may not be required, and a user may prefer a fast, approximate answer. For example, during a drill-down query sequence in ad-hoc data mining, the earlier queries in the sequence are used solely to determine what the interesting queries are [GM95, HHW97]. An approximate answer can also provide feedback on how well-posed a query is. Moreover, it can provide a tentative answer to a query when the base data is unavailable. Another example is when the query requests numerical answers, and the full precision of the exact answer is not needed, e.g., a total, average, or percentage for which only the first few digits of precision are of interest (such as the leading few digits of a total in the millions, or the nearest percentile of a percentage). Finally, note that techniques for fast approximate answers can also be used in a more traditional role within the query optimizer to estimate plan costs; such an application demands very fast response times but not exact answers.

Despite some recent work in approximate query answers (e.g., [GM95, HHW97, GM97a, GP97]), the state-of-the-art is quite limited in its scope and accuracy, and further research is greatly needed.

This white paper describes the **Approximate QUery Answering (AQUA)** Project underway in the Information Sciences Research Center at Bell Labs. The goals of the project are to provide a framework for approximate query answering, new techniques for improving the speed and accuracy of approximate answers, and a working prototype. In this paper, we present a framework for an approximate query engine that observes new data as it arrives and maintains small synopsis data structures on that data. These data structures are used to provide fast, approximate answers to a broad class of queries. We describe metrics for evaluating approximate query answers. We also present new synopsis data structures, and new techniques for approximate query answers. Finally, we report on the status of the Aqua project and plans for future work.

[1] For example, scanning 3 TBs of data using parallel reads from 100 disks at a time with 20MB/s from each disk takes 25 minutes.

2 A framework for approximate query answering

Figure 1 depicts a traditional data warehouse set-up, in which the base data resides in a data warehouse that is updated as new data arrives, and each query is answered exactly using the data warehouse. In contrast, Figure 2 depicts a set-up for approximate query answering, which includes an approximate query engine in addition to the data warehouse. To facilitate in answering queries, the approximate query engine can store various summary information on the data, which we denote *synopsis data structures* [GM97b].[2] Examples of synopses for a relational data warehouse include histograms and sample rows of large relations and all the rows of small relations, projected on the columns of interest. These synopses can be maintained by:

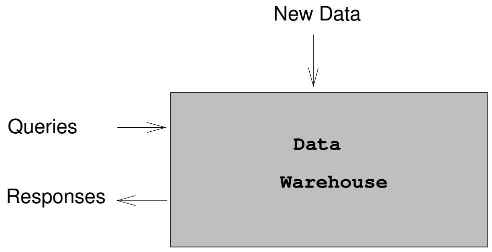

Figure 1: A traditional data warehouse.

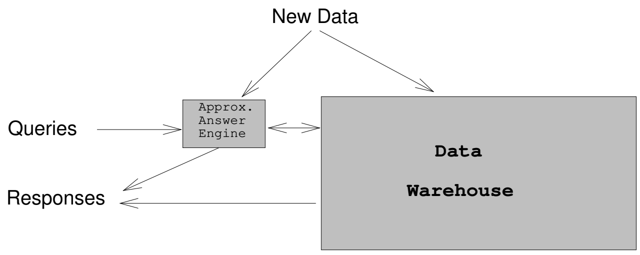

Figure 2: Data warehouse set-up for providing approximate query answers.

- observing the new data as it is loaded into the data warehouse,
- periodically returning to the data warehouse to update the information, and/or
- returning to the data warehouse at query time.

Queries are sent to the approximate answer engine. Whenever possible, the engine uses its synopsis data structures to promptly return a query response, consisting of an approximate answer and an accuracy measure (e.g., a 95% confidence interval for numerical answers).

- In *continuous reporting*, the engine proceeds to provide a series of ⟨approximate answer, accuracy measure⟩ pairs for the query, with each subsequent pair providing a more accurate answer (e.g. [HHW97]).
- In *discrete reporting*, only one or a few such pairs are provided by the engine.

The engine can also return an estimated time for computing an exact answer, as determined by the approximate answer engine and/or a traditional query optimizer. The user posing the query can decide whether to abort the query processing and be content with the current approximate answer or to proceed to the next approximation or to an exact answer from the base data.

There are further interactions between a user and an approximate query engine that can be explored. For instance, a user may use the approximate answer as a tentative answer, and go on (tentatively) to the next query in a drill-down sequence without aborting the previous query. The purpose of having the previous query continue is two-fold. First, it serves to verify the accuracy

[2] A synopsis data structure captures the important/highlight information on the data in a concise representation, i.e., it provides a "synopsis" of the data.

The exact answer			
region	**type**	**avg. sales**	**min. sales**
eastern	retail	12435	4035
eastern	outlet	7389	1227
central	retail	14837	3928
western	retail	16726	4399
western	outlet	8874	389

An approximate answer			
region	**type**	**avg. sales**	**min. sales**
eastern	retail	12000 ± 800	4100 ± 400
eastern	outlet	7200 ± 800	1200 ± 400
central	retail	14500 ± 800	3800 ± 400
central	outlet	< 500	< 500
western	retail	17000 ± 800	4100 ± 400
western	outlet	8900 ± 800	< 500

Figure 3: An example approximate answer for a group by query.

of the approximate answer. Second, a refined answer can be used to enable a correction process (e.g., additional queries for the missing part). For example, suppose that an approximate answer is 100 ± 7, and based on using 100 as the tentative answer, the next query is a selection query whose range is $[200, 300]$. If it turns out that 106 is the exact answer (or the more likely answer), the user may decide that the range of interest is actually $[200, 330]$, and can obtain the rest of the range by submitting a supplementary selection query whose range is $[300, 330]$.

2.1 What is an approximate answer?

For queries whose answer is an aggregate value (e.g., the result of avg, count, max, min, sum), the notion of an approximate answer is an intuitive one: it is simply an estimated value for the answer and an accuracy measure. This can be extended to a collection of aggregate values, such as arises with an SQL *group by* operation: an approximate answer is an ⟨estimated value, accuracy measure⟩ pair for each such aggregate value, labeled with the attributes that define the aggregate (the group). An example is given in Figure 3. In this example, the approximate answer provides accuracy measures as confidence intervals for each estimate, for some confidence probability that would also be specified (e.g., 95% confidence intervals). Note that in several cases an upper bound, denoted a *sanity bound*, is provided instead of an estimate. Finally, note that an approximate answer can include rows not in the exact answer, as in this example (for the *central outlet* group), and vice-versa.

For more general queries, it is less intuitive what an approximate answer should be. We consider queries that return a set of tuples. Since the number of tuples in the exact answer may be quite large, we often do not want to return a tuple for each tuple in the exact answer. In order to ensure very fast response times, we seek to return only a small number of representative tuples, together with meta-information on the entire set of tuples. Thus an approximate answer consists of both estimates on meta-information for the exact answer, including a count or an estimate of the number of tuples in the exact answer, and representative tuples from the exact answer. Each meta-information estimate includes an accuracy measure. Representative tuples can be classified as:

- *actual*: the tuple is in the output of the exact answer, or
- *tentative*: the tuple may be in the output of the exact answer.

Tentative tuples are reported along with some measure of their similarity to actual tuples. Examples include tuples that are actual tuples with a given confidence probability or tuples that may not meet a selection criterion (such as min or max) that is computed by the query, but are close to it. Actual tuples are preferred to tentative tuples. Actual tuples can be classified as:

- *randomly-selected*: the tuples reported are a uniform random sample of the set of output tuples,
- *biased-selected*: the tuples reported are biased according to a specific criterion, or
- *arbitrary*.

Randomly-selected tuples have the advantage that they are uniformly representative of the entire set of output tuples. Biased-selected tuples have the advantage if the bias criterion is in line with the "most interesting" output tuples, e.g., the query requests tuples that lie above a certain threshold and the reported tuples are biased towards those that exceed the threshold by the largest amount. In such cases, biased-selected may be preferred to randomly-selected. On the other hand, if the criterion for what makes an output tuple interesting is not known, or there are conflicting criteria, then a uniform random sample is a natural choice.

Representative tuples may or may not contain all of the attributes in the full tuple.

There are a number of possible accuracy measures for an approximate answer, depending on the type of query. For numerical answers, a natural accuracy measure is a confidence interval, consisting of an accuracy interval $[a, b]$ and a confidence probability p. The confidence interval asserts that the exact value is between a and b with probability at least p. Confidence is ideally absolute, reporting that an approximate answer is surely within a given interval (the case where $p = 1$). Absolute confidence often arises when rounding techniques are in use (e.g., [MVY94]). In the context of synopsis data structures, we should expect that in most cases confidence would not be absolute, and the goal is to minimize the accuracy interval while maximizing the confidence probability. This trade-off can often be expressed parametrically, e.g., the exact value is $x \pm \beta y$ with probability at least $1 - e^{-\beta z}$ for all $\beta > 1$. It may also be useful to have the approximate answer be an *unbiased* estimator of the exact value, that is, the expected value of the approximate answer is equal to the exact value.

Accuracy measures and similarity measures can be classified as:

- *guaranteed*, or
- *heuristic*,

with guaranteed measures (such as confidence intervals) preferred. On the other hand, in some cases, approximate answers with high accuracy guaranteed measures may be too expensive to compute, and heuristic measures are more suitable.

Table 1 summarizes the requirements for approximate answers.

2.2 Metrics for evaluating approximate query engines

Approximate query engines can be evaluated according to the following five metrics:

- *coverage*: the range of queries for which approximate answers can be provided.

Table 1: Approximate answers.

Exact answer	Approximate answer
aggregate values	for each aggregate value: 1. estimated value or sanity bound 2. accuracy measure
set of tuples	1. estimated meta-information on the output, i.e., a collection of ⟨value, accuracy measure⟩ pairs. 2. representative tuples, either: a. randomly-selected actual b. biased-selected actual, with criterion c. arbitrary actual, or d. tentative, with similarity measure

- *response time*: the time to provide an approximate answer for a query.
- *accuracy*: the accuracy of the answers provided, and the confidence in that accuracy.
- *update time*: the overheads in keeping its synopses up-to-date.
- *footprint*: the storage requirements for its synopses.

Typically, there will be trade-offs among these metrics, e.g., with continuous reporting, the additional response time for each subsequent answer results in greater accuracy.

Synopsis data structures. The same metrics – coverage, response time, accuracy, update time and footprint – can be applied to each potential synopsis. In particular, small footprints are desirable in order to minimize response time and not to consume too much of the system memory. To handle many base tables and many types of queries, a large number of synopses may be needed. Moreover, for fast response times that avoid disk access altogether, synopsis data structures should be memory-resident.[3] Thus we evaluate the effectiveness of a synopsis as a function of its footprint. For example, it is common practice to evaluate the effectiveness of a histogram in estimating range selectivities as a function of the number of histogram buckets. Although machines with large main memories are becoming increasingly commonplace, this memory remains a precious resource, as it is needed for query-processing working space (e.g., building hash tables for hash joins) and for caching disk blocks. Moreover, small footprints are more likely to lead to effective use of the processor's L1 and/or L2 cache; e.g., a synopsis that fits entirely in the processor's cache enables even faster response times.

[3]Various synopses can be swapped in and out of memory as needed. For persistence and recovery, combinations of snapshots and/or logs can be stored on disk; alternatively, the synopsis can often be recomputed in one pass over the base data.

3 The Aqua approach

The Aqua project focuses on an approximate query engine that provides highly-accurate answers with minimal response time by:

- maintaining a number of synopses on the data,
- updating these synopses primarily by observing the new data as it is loaded into the data warehouse,
- providing discrete reporting,
- ensuring guaranteed accuracy measures, and
- having a footprint orders of magnitude smaller than the data warehouse.

A goal of the Aqua project is to develop effective synopsis data structures with minimal footprints and new techniques for using them for approximate query answers. Synopses are maintained primarily by observing the new data as it is loaded into the data warehouse, since this can enable highly-accurate answers with minimal response time. Observing the new data ensures that the synopses are kept up-to-date. Relying solely on periodic (e.g., daily) scans of the data warehouse to update the synopses would mean that the approximate answers would not take into account the most recent trends in the data (i.e., those occurring subsequent to the last scan), and hence could greatly decrease the accuracy guarantees (see, e.g., [GMP97]). Moreover, such scans can be quite time-consuming, and with the trend towards 7×24 operation of data warehouses, there is often no "idle" time during which the updates could be performed [Sch97].

Update time overheads in Aqua can be minimized in two ways:

1. Bulk updating. Aqua can buffer the new data as it arrives, and then periodically (e.g., every few minutes, or on demand in response to a query) update its synopses using the data in the buffer.

2. Update sampling. Aqua can sample from the sequence of updates, ignoring all but the sampled updates. For example, sampling 10% of the updates can reduce the update time overheads by a factor of 10. Note that we have decoupled the use of sampling as a means to lower update times from any use of sampling to maintain a small footprint, since the requirements differ: update sampling rates of 10% or higher are reasonable, while a footprint that is 10% or more of the data warehouse size is very likely unacceptable. We studied the effectiveness of update sampling, in the context of maintaining approximate histograms, in [GMP97].

Note that observing (nearly) all the data can be important for maintaining high-quality synopses. For example, the number of distinct values for an attribute in a relation of size n can be maintained quite accurately in $O(\lg n)$ bits of memory by observing all the data [FM85], but it is quite difficult to estimate using only samples of the data [HNSS95].

As discussed in the previous section, small footprints are good for response times and update times. A goal for Aqua is to have memory-resident any synopsis that is frequently updated and/or frequently used to respond to queries, in order to minimize these times.

Although heuristic accuracy measures for estimation procedures are common in commercial databases, they are not quite satisfying. This is particularly true for approximate query answers, since the approximate answer is reported to the user. One of the goals of the Aqua project is to push from heuristic confidence to guaranteed confidence.

Aqua provides for discrete reporting, instead of continuous reporting, as a trade-off for faster update and response times. Discrete reporting can be supported using an approximate engine with simpler synopses and a modest size footprint, since only a single level (or a few levels) of accuracy are to be reported. Continuous reporting, on the other hand, requires a means for reporting answers with increasing accuracy, resulting in greater complexity and either a sufficiently large footprint to support the highest level of accuracy or a very slow response time if the base data must be heavily accessed. We consider a user interface in which a query is posed, and the Aqua engine replies (whenever possible) with an approximate answer and an estimate of the time to compute an exact answer. The user can then decide whether or not to proceed to the exact answer.

4 Approximate answers using samples

In this section, we describe our base Aqua system, for a relational data warehouse. This system is used for comparison with versions of Aqua that employ the enhancements described in subsequent sections.

For the base system, we establish a threshold M on the maximum number of tuples/rows retained in the approximate query engine for any one relation. For each relation, we select which attributes/columns to retain. Examples of attributes not retained are descriptive strings such as street address or part description.

The base system maintains the following synopses:

- For each relation, we maintain a counter of the number of tuples in the relation.
- For relations with at most M tuples, we store all the tuples in the relation. For each such tuple, we store only the selected attributes.
- For relations with more than M tuples, we store a random sample of (target) size M of the tuples in the relation. For each such tuple, we store only the selected attributes.

The counters can be maintained by incrementing them as tuples are inserted and decrementing them as tuples are deleted. The random samples can be maintained as tuples are inserted and deleted using the algorithm discussed in Section 5.1.

Reporting approximate answers. In response to a query, the approximate query engine executes the query on the synopsis data, scales the results as needed, and reports the answer. The accuracy measure is calculated based on the type of query, the size of M, and the sizes of each relation in the query with more than M tuples. Note that the accuracy of the approximate answer improves with M, the size of the samples.

5 New maintenance techniques

In this section, we present a number of new techniques for maintaining important synopsis data structures as new data arrives.

5.1 Maintaining random samples

In most uses of random samples in estimation, whenever a sample of size n is needed, it is extracted from the base data: either the entire relation is scanned to extract the sample, or n random disk blocks must be read (since tuples in a disk block may be highly correlated). In Aqua, we eliminate these large response time overheads by maintaining a random sample at all times. As argued in [GMP97], maintaining a random sample allows for the sample to be packed into consecutive disk blocks or in consecutive pages of memory. Moreover, for each tuple in the sample, only the attribute(s) of interest are retained, for an even smaller footprint and faster retrieval.

A *backing sample* is a uniform random sample of 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. 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 [GMP97], we advocated the use of a backing sample and presented 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). The algorithm is presented next.

Let $\mathcal{S}$ be a backing sample of target size n maintained for a relation R. We first consider insertions to R. We use Vitter's *reservoir sampling* technique [Vit85]: The algorithm proceeds by inserting the first n tuples into a "reservoir." Then a random number of new tuples are skipped, and the next tuple replaces a randomly selected tuple in the reservoir. Another random number of tuples are then skipped, and so forth. The distribution function of the length of each random skip depends explicitly on the number of tuples so far, and is chosen such that at any point each tuple in the relation is equally likely to be in the reservoir. We extend this technique to handle modify and delete operations, as follows. Modify operations are handled by updating the value field(s), 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 [GMP97], we showed 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.

We proved in [GMP97] that this algorithm maintains the property that $\mathcal{S}$ is a uniform random sample of a relation R such that $\min(|R|, L) \leq |\mathcal{S}| \leq U$.

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 modifys and deletes 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 modifys and deletes is important, since it reduces the problem to the insert-only case, and hence the criterion of Algorithm X can be applied to determine how many inserts to skip.

With these optimizations, inserts and modifys to attributes not of interest are processed with minimal overhead, whereas deletes and modifys to attributes of interest can require a somewhat larger overhead (due to the batch processing of testing whether the id is in the sample).

5.2 Maintaining approximate histograms

The most common technique used in commercial database systems for estimating selectivities is maintaining *histograms* on the frequency distribution of an attribute. We describe new techniques, presented in [GMP97], for maintaining approximate equi-depth and Compressed histograms.

5.2.1 Equi-depth histograms

In an *equi-depth* histogram with β buckets, contiguous ranges of attributes are grouped into β buckets such that the number of tuples in each bucket is $|R|/\beta$, where $|R|$ is the number of tuples in the relation R. 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 associated with the buckets. Such histograms can be evaluated based on how close the buckets are to $|R|/\beta$ tuples and how close the counts are to the actual number of tuples in their respective buckets.

In [GMP97], we presented the first low overhead algorithm for maintaining highly-accurate approximate equi-depth histograms. The algorithm relies on using a backing sample $\mathcal{S}$ such that $|\mathcal{S}| \geq 4\beta \ln^2 \beta$. An approximate equi-depth histogram can be computed from $\mathcal{S}$ by sorting $\mathcal{S}$ and then taking every $(|\mathcal{S}|/\beta)$'th tuple as a bucket boundary, with the bucket counts set to $|R|/\beta$. Our algorithm minimizes the overheads by performing this recomputation from $\mathcal{S}$ only when necessary.

The algorithm works in phases. At each phase there is a threshold $T = \lceil (2+\gamma)N'/\beta \rceil$, where N' is the number of tuples in R at the beginning of the phase, and $\gamma > -1$ is a tunable performance parameter. Larger values for γ allow for greater imbalance among the buckets in order to have fewer bucket boundary changes and fewer recomputations from $\mathcal{S}$. 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 bucket's count reaches the threshold T, we split the bucket in half. In order to maintain the number of buckets β fixed, we merge two adjacent buckets whose total count is less than T, if such a pair of buckets can be found. When such a merge is not possible, we recompute the approximate equi-depth histogram from $\mathcal{S}$.

The operation of merging two adjacent buckets is quite simple: we sum the counts of the two buckets and dispose of the boundary 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. The split and merge operation is illustrated in Figure 4. Note that split and merge can occur only for $\gamma > 0$.

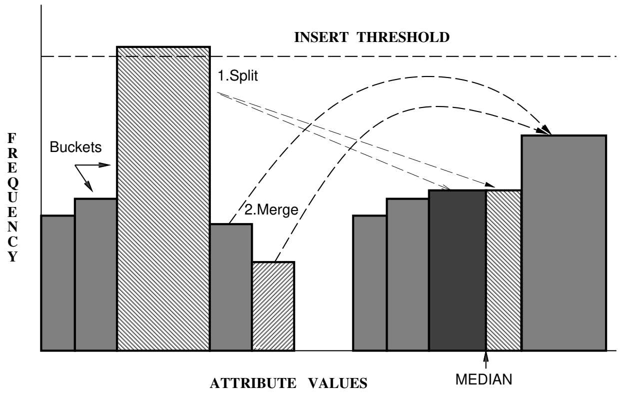

Figure 4: Split and merge operation during equi-depth histogram maintenance

Also at each phase, there is a lower threshold $T_\ell = \lfloor N'/(\beta(2+\gamma_\ell)) \rfloor$, where N' is the number of tuples in R at the beginning of the phase, and $\gamma_\ell > -1$ is a tunable performance parameter. The number of tuples in any given bucket is maintained above the threshold T_ℓ. As tuples are deleted from R, we decrement the counts of the appropriate buckets. If a bucket's count drops to the threshold T_ℓ, we merge the bucket with one of its adjacent buckets and then split the bucket B' with the largest count, as long as its count is at least $2(T_\ell + 1)$. (Note that B' may be the newly merged bucket.) If no such B' exists, we recompute the approximate equi-depth histogram from $\mathcal{S}$. The merge and split operation is illustrated in Figure 5.

For modify operations, if the modify results in the tuple changing buckets, then we update the histogram by treating the modify as a delete followed by an insert.

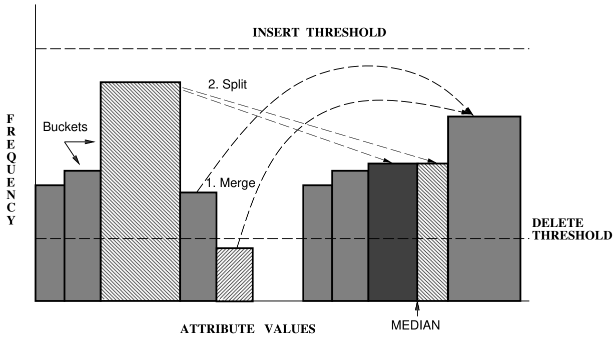

Figure 5: Merge and split operation during equi-depth histogram maintenance

5.2.2 Compressed histograms

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 by a single bucket with a single count, resulting in the same information within a smaller footprint. 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: (i) each equi-depth bucket has N'/β' tuples, where N' is the total number of tuples in equi-depth buckets, (ii) no single value "spans" an equi-depth bucket, i.e., the set of bucket boundaries are distinct, and conversely, (iii) the value in each singleton bucket has frequency $\geq N'/\beta'$. 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 associated with the buckets.

In [GMP97], we presented the first low overhead algorithm for maintaining highly-accurate approximate Compressed histograms. The algorithm relies on using a backing sample $\mathcal{S}$, as with the equi-depth algorithm. An approximate Compressed histogram can be computed from $\mathcal{S}$ as follows. Let m, initially $|\mathcal{S}|$, be the number of tuples tentatively in equi-depth buckets. We consider the $\beta - 1$ most frequent values occurring in $\mathcal{S}$, in order of maximum frequency. For each such value, if the frequency f of the value is at least m divided by the number of equi-depth buckets, we create a singleton bucket for the value with count $f|R|/|\mathcal{S}|$, and decrease m by f. Otherwise, we stop creating singleton buckets and produce an equi-depth histogram on the remaining values, using the approach of the previous subsection, but setting the bucket counts to $(|R|/|\mathcal{S}) \cdot (m/\beta')$. Our algorithm minimizes the overheads by performing this recomputation from $\mathcal{S}$ only when necessary.

Similar to the equi-depth algorithm, the algorithm works in phases, where each phase has an upper threshold for triggering equi-depth bucket splits and a lower threshold for triggering bucket merges. The steps for updating the bucket boundaries are similar to those for an equi-depth histogram, 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 the equi-depth algorithm). 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.

The requirements for when a bucket can be split or when two buckets can be merged are more involved than in the equi-depth algorithm: A bucket B is a *candidate split bucket* if it is an equi-depth bucket whose count is at least $2(T_\ell + 1)$ or a singleton bucket whose count is bounded by $2(T_\ell + 1)$ and $T/(2+\gamma)$. 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) the sum of their counts is less than T. When there are more than one candidate split bucket (candidate merge pair), the algorithm selects the one with the largest (smallest combined, respectively) bucket count.

5.2.3 Analytical and experimental studies

In [GMP97], we presented analytical and experimental studies of these algorithms, which showed the following:

- 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 overheads decrease.

- 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.

- Histograms maintained using these techniques remain highly effective in estimating range selectivities, unlike all previous approaches.

The CPU, I/O and storage requirements for these techniques are negligible for insert-mostly and sliding window data warehouses.

5.3 Maintaining frequency moments

Consider a data set R of size n with values from a domain D, and for each $i \in D$, let m_i be the frequency of value i in R. For each $k \geq 0$, the *kth frequency moment*, F_k, is $\sum_{i \in D} m_i^k$. In particular, F_0 is the number of distinct values in R, $F_1 = n$ is the size of the data set, and F_2 is the *repeat rate* or *self-join size* of R. The *maximum frequency*, F_∞, is $\max_{i \in D}\{m_i\}$.

In [AMS96], we obtained tight bounds for the minimum possible footprint required to approximate the numbers F_k to within a fixed constant, as follows. Let $m = |D|$. We proved that for every $k > 0$ and any constant success probability greater than $1/2$, F_k can be approximated to within a fixed relative constant randomly using at most $O(m^{1-1/k} \lg m)$ memory bits. We further showed that for $k \geq 6$, any (randomized) approximation algorithm for F_k requires at least $\Omega(m^{1-5/k})$ memory bits.

In addition, we showed that for any nonnegative integer $k \neq 1$, both approximation and randomization are necessary for approximating F_k to within a relative constant if only $o(\min(n, m))$ memory bits are used. We also presented an $\Omega(\lg m)$ lower bound on the memory bits required to approximate F_0 to within a relative constant, matching the upper bound of [FM83, FM85]. Moreover, we presented an $\Omega(\lg \lg n)$ lower bound on the memory bits required to approximate F_1 to within a relative constant, matching the upper bound of [Mor78].

In [AGMS96], we extended the algorithm for maintaining F_k in the presence of inserts to the data set to handle deletions as well. The algorithm maintains a constant number of random samples of target size $t = \Theta(m^{1-1/k})$. For each item selected for a sample, we maintain a count of the number of items with the same value that are subsequently inserted into R minus the number of items with the same value that are subsequently deleted. Note that the same value may occur multiple times in the same sample and in different samples, with a separate counter for each such occurrence. For each sample S_i, let $c_1, c_2, \ldots, c_t$ be the counts associated with the sample, and let $X_i = \frac{n}{t} \sum_{j=1}^{t}(c_j^k - (c_j - 1)^k)$. We estimate F_k by reporting the median of the X_i over all the samples. We proved that for each counter, c_j, the expected value of $n(c_j^k - (c_j - 1)^k)$ is F_k; the additional counters are used solely to reduce the variance in the estimator.

Maintaining the self-join size F_2. The algorithm presented above for general F_k provides an algorithm for F_2 that uses $O(\sqrt{m}(\lg m + \lg n))$ memory bits. In [AMS96], we presented another algorithm, the *tug-of-war* algorithm, for approximating F_2 in the presence of insertions to R using only $O(\lg m + \lg \lg n)$ memory bits. This matches the $\Omega(\lg m + \lg \lg n)$ lower bound we showed. In [AGMS96], we extended the algorithm to handle deletions, and presented experimental results comparing this algorithm with the algorithm in the preceding paragraph.

In the tug-of-war algorithm, we consider the family $\mathcal{H}$ of 4-wise independent mappings from the domain D to $\{-1, 1\}$. For each randomly selected $h_i \in \mathcal{H}$, we maintain a sum Z_i as follows. On an insert to R of value v, we add $h_i(v)$ to Z_i; on a delete, we subtract $h_i(v)$ from Z_i. We proved that F_2 is the expected value of Z_i^2. To reduce the variance and obtain an estimate that is provably within a constant factor with probability $> 1/2$, we select a constant number of h_i and estimate F_2 by reporting the median of the average of the Z_i^2, similar to above. We use $O(\lg m)$ bits to represent and compute the functions h_i; the number of bits for the sums Z_i can be reduced to $O(\lg \lg n)$ by using approximate counting [Mor78].

Maintaining the maximum frequency F_∞. We also showed in [AMS96] that any (deterministic or randomized) algorithm for approximating the frequency of the mode F_∞ of a given data set to within a constant factor (with probability $> 1/2$), using only a constant number of passes over the data, requires $\Omega(m)$ memory bits in the worst case. This lower bound was obtained by reducing the problem to an appropriate multi-party communication complexity problem, and then proving the result for the communication complexity problem.

6 New synopsis data structures

In this section we describe two new sampling-based synopses, *concise samples* and *counting samples*, and present techniques for their fast incremental maintenance regardless of the data distribution. These results are reported in detail in [GM97a].

Both concise samples and counting samples can greatly increase the number of sample points over traditional samples for the same footprint. Since the accuracy of an approximate answer improves with the size of the samples, using concise or counting samples can significantly improve the accuracy over using traditional samples.

6.1 Concise samples

Consider the class of queries that ask for the frequently occurring values for an attribute in a relation of size n. One possible synopsis is a uniform random sample of the tuples in the relation projected on the attribute. An approximate answer to the query would then be $\langle$value, count$\rangle$ pairs for any value occurring frequently in the sample, where the reported counts are n/m times the frequency in a sample of size m (for example, by Chernoff bounds, any value occurring $k = \Omega(\lg n)$ times in a sample of size m can be accurately estimated to occur $k \cdot n/m$ times in the data set w.h.p.). However, note that any value occurring frequently in the sample is a wasteful use of the footprint. We can represent k copies of the same value v as the pair $\langle v, k\rangle$, and (assuming that values and counts use one unit of space each), we have freed up space for $k-2$ additional sample points. This simple observation leads to the following new sampling-based synopsis:

- A *concise sample* is a uniform random sample of the data set such that values appearing more than once in the sample are represented as a value and a count.

This simple idea is quite powerful, and to the best of our knowledge, has never before been studied. For simplicity, we describe concise samples in terms of a single attribute, although the approach applies equally well to pairs of attributes, etc.

Since a concise sample represents sample points occurring more than once as $\langle$value, count$\rangle$ pairs, the true sample size may be much larger than its footprint (it is never smaller). Define the *sample-size* of a concise sample $S = \{\langle v_1, c_1\rangle, \ldots, \langle v_j, c_j\rangle, v_{j+1}, \ldots, v_\ell\}$ to be $\ell - j + \sum_{i=1}^{j} c_i$. For simplicity, we assume that values and counts contribute one unit of space each to the footprint, so that the footprint of S is $\ell + j$. In general, variable-length encoding could be used for the counts, so that only $\lceil \lg x \rceil$ bits are needed to store x as a count, further reducing the footprint.[4]

Note that if a data set of size n has at most $m/2$ distinct values, then a concise sample of sample-size n has a footprint at most m (i.e., in this case, the concise sample is the exact histogram

[4] Approximate counts [Mor78] could be used as well, for an even smaller footprint.

of all $\langle$value, count$\rangle$ pairs for the data set). Thus, the sample-size of a concise sample may be arbitrarily larger than its footprint: For any footprint $m \geq 2$, there exists data sets for which the sample-size of a concise sample is n/m times larger than its footprint, where n is the size of the data set.

Maintaining concise samples. We describe a fast algorithm for maintaining a concise sample within a given footprint bound as new data is inserted into the data warehouse. Since the number of sample points provided by a concise sample depends on the data distribution, unlike traditional samples, the problem of maintaining a concise sample as new data arrives is more difficult than with traditional samples. The approach described in Section 5.1 relies heavily on the fact that we know in advance the sample-size (which, for traditional samples, equals the footprint). With concise samples, the sample-size depends on the data distribution to date, and any changes in the data distribution must be reflected in the sampling frequency.

Our maintenance algorithm is as follows. We set up an entry threshold τ (initially 1) for new tuples to be selected for the sample. Let S be the current concise sample and consider a new tuple being inserted into the data warehouse, with value v. With probability $1/\tau$, we add v to S. To add v to S, we first do a look-up on v in S. If it is represented by a pair, we increment its count. Otherwise, if v is a singleton in S, we create a pair, or if it is not in S, we create a singleton. In these latter two cases we have increased the footprint by 1, so if the footprint for S was already equal to the prespecified footprint bound, then we need to evict existing sample points to create room.

In order to create room, we raise the threshold to some τ' and then subject each sample point in S to this higher threshold. Specifically, if m is the sample-size of S then each of the m sample points in S is evicted with probability τ/τ'. We expect to have $m(1 - \tau/\tau')$ sample points evicted. Note that the footprint is only decreased when a $\langle$value, count$\rangle$ pair reverts to a singleton or when a value is removed altogether. If the footprint has not decreased, we raise the threshold and try again. Subsequent inserts are selected for the sample with probability $1/\tau'$.

In [GM97a], we proved that for any sequence of insertions to the data warehouse, the above algorithm maintains a concise sample. The algorithm maintains a concise sample regardless of the sequence of increasing thresholds used. Thus, there is complete flexibility in deciding, when raising the threshold, what the new threshold should be. A large raise may evict more than is needed to reduce the sample's footprint below its upper bound, resulting in a smaller sample-size than when the sample's footprint matches the upper bound. On the other hand, evicting more than is needed creates room for subsequent additions to the concise sample, so the procedure for creating room runs less frequently. A small raise also increases the likelihood that the footprint will not decrease at all, and the procedure will need to be repeated with a higher threshold.

Note that instead of flipping a coin for each insert into the data warehouse, we can flip a coin that determines how many such inserts can be skipped before the next insert that must be placed in the sample (as in Vitter's reservoir sampling described in Section 5.1): the probability of skipping over exactly i elements is $(1 - 1/\tau)^i \cdot (1/\tau)$. Likewise, since the probability of evicting a sample point is typically small (i.e., τ'/τ is a small constant), we can save on coin flips and decrease the update time by using a similar approach when evicting. We proved in [GM97a] that this algorithm incurs a constant amortized expected update time per new tuple inserted into the data warehouse, regardless of the data distribution.

The expected sample-size increases with the skew in the data. We showed above that the advantage is unbounded for certain distributions. In [GM97a], we further quantified the advantage in sample-size, both analytically and experimentally. For example, we showed that for the family of exponential distributions,

$$\Pr(v = i) = \alpha^{-i}(\alpha - 1), \quad i = 1, 2, \cdots,$$

$\alpha > 1$, and any footprint $m \geq 2$, the expected sample-size of a concise sample with footprint m is at least $\alpha^{m/2}$. Thus for exponential distributions, the advantage is exponential. Figures 6 and 7 depict experimental results quantifying the advantage for a wide range of Zipf distributions. These plots show the sample-sizes for traditional samples (i.e., 1000), for concise samples produced by the above algorithm, and for concise samples produced by an offline algorithm that extracts sample points from the data set until the target footprint of 1000 is exceeded. Thus in addition to showing the large advantage of concise samples over traditional samples, the figures demonstrate that our maintenance algorithm yields concise samples whose sizes are nearly optimal.

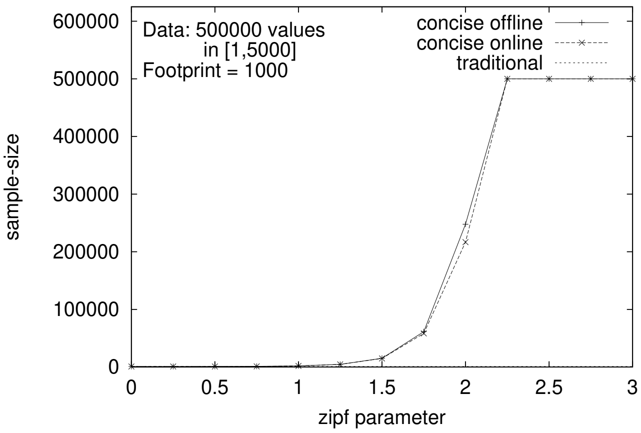

Figure 6: Concise vs. traditional

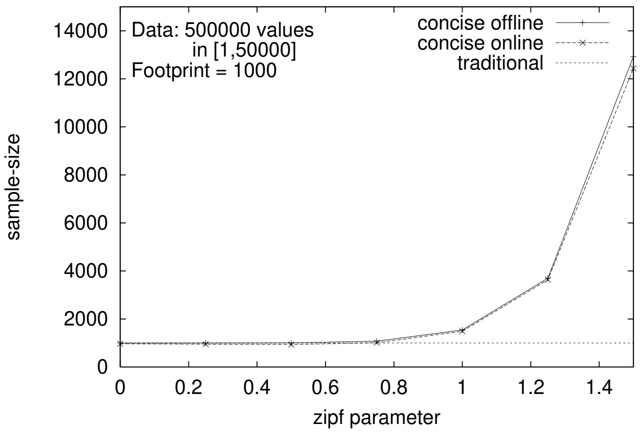

Figure 7: Concise vs. traditional

6.2 Counting samples

Counting samples are a variation on concise samples in which the counts are used to keep track of all occurrences of a value inserted into the data warehouse since the value was selected for the sample.[5] Their definition is motivated by a sampling-and-counting process of this type from a static data warehouse:

- A *counting sample* with threshold τ for a data set R is any subset of R obtained as follows: For each value v occurring $c > 0$ times in R, we flip a coin with probability $1/\tau$ of heads until the first heads, up to at most c coin tosses in all; if the ith coin toss is heads, then v occurs $c - i + 1$ times in the subset, else v is not in the subset. Each value v occurring $c > 1$ times in the subset is represented as a pair $\langle v, c \rangle$, and each value v occurring exactly once is represented as a singleton v.

[5] In other words, since we have set aside space for a count, why not count the subsequent occurrences exactly?

Extracting a concise sample from a counting sample. Although a counting sample is not in general a uniform random sample of the data set R, it can be used to obtain such a sample without any further access to R. Specifically, a concise sample can be obtained from a counting sample by considering each pair $\langle v, c\rangle$ in the counting sample in turn, and flipping a coin with probability $1/\tau$ of heads $c-1$ times and reducing the count by the number of tails. The footprint decreases by one for each pair whose coin flips are all tails.

Maintaining counting samples. We describe a fast algorithm for maintaining a counting sample within a given footprint bound as data is inserted into or deleted from the data warehouse.

We set up an entry threshold τ (initially 1) for new tuples to be selected for the counting sample. Let S be the current counting sample and consider a new tuple being inserted into the data warehouse, with value v. We do a look-up on v in S. If it is represented by a pair, we increment its count. If it is a singleton in S, we create a pair. Otherwise, v is not in S and we add it to S with probability $1/\tau$.

If the footprint for S now exceeds the prespecified footprint bound, then we need to evict existing values to create room. As with concise samples, we raise the threshold to some τ', and then subject each value in S to this higher threshold. The process is slightly different for counting samples, since the counts are different. For each value in the counting sample, we flip a biased coin, decrementing its count on each flip of tails until either the count reaches zero or a heads is flipped. The first coin toss has probability of heads τ/τ', and each subsequent coin toss has probability of heads $1/\tau'$. Values with count zero are removed from the counting sample; other values remain in the counting sample with their (typically reduced) counts. The overall number of coin tosses can be reduced to a constant per value using an approach similar to that described for concise samples, since we stop at the first heads (if any) for each value.

An advantage of counting samples over concise samples is that we can maintain counting samples in the presence of deletions to the data warehouse. Maintaining concise samples in the presence of such deletions is difficult: If we fail to delete a sample point in response to the delete operation, then we risk having the sample fail to be a subset of the data set. On the other hand, if we always delete a sample point, then the sample may no longer be a random sample of the data set. With counting samples, we do not have this difficulty. For a delete of a value v, we do a look-up to see if v is in the counting sample, and decrement its count if it is.

In [GM97a], we proved that for any sequence of insertions or deletions, the above algorithm maintains a counting sample. The algorithm maintains a counting sample regardless of the sequence of increasing thresholds used. If the threshold is raised by a constant factor each time, we showed in [GM97a] that this algorithm incurs a constant amortized expected update time per tuple inserted into or deleted from the data warehouse, regardless of the data distribution.

Let R be a data set and τ be the current threshold for a counting sample S. We proved in [GM97a] the following three properties of counting samples: (i) Any value v that occurs at least τ times in R is expected to be in S, (ii) any value v that occurs f times in R will be in S with probability $1-(1-\frac{1}{\tau})^f$, and (iii) for all $\alpha > 1$, if $f \geq \alpha \cdot \tau$, then with probability $\geq 1 - e^{-\alpha}$, the value will be in S and its count will be at least $f - \alpha\tau$.

6.3 Accurate and efficient histogram techniques

In Section 5.2, we presented results on maintaining two important classes of histograms: equi-depth and Compressed histograms on individual attributes. In this section, we discuss the need for new classes of histograms, and efficient techniques for computing them.

In their current form, histograms suffer from the following drawbacks:

- They are mostly used for approximating individual attributes and hence do not provide accurate estimates for multi-attribute queries.
- There has been very little work on accuracy measures (error analysis) for histogram-based estimates, i.e., estimates are often provided with no knowledge about their accuracy.
- There are no known efficient techniques for computing the important class of *V-Optimal* histograms, which have been shown to be highly accurate for several estimation problems [PIHS96].

We have worked on all the above aspects in extending the work on histograms [MPS97a, MPS97b]. Our main contributions are as follows:

1. We have identified novel and highly-accurate classes of multidimensional histograms and provided a uniform framework for computing *optimal* histograms (i.e., those minimizing an error metric for a given footprint). The main feature of this framework is that it can be used for optimizing a large class of error metrics, even in the presence of a combination of approximation techniques (e.g., histograms and splines in different regions of the data).
2. We have also devised solutions for the dual problem, namely, identifying histograms for a given bound on the error. In addition, we have incorporated auxiliary information into histogram buckets in order to provide better accuracy measures for an estimation.
3. We have designed the first known efficient algorithm for computing one-dimensional V-Optimal histograms (a direct outcome of our framework mentioned above).

6.4 Statistics selection and data cube approximation

Users of on-line analytical processing (OLAP) systems find it useful to organize data along several dimensions of a multidimensional *data cube* and perform aggregate analysis on (possibly subsets of) the dimensions [GBLP96]. The cells of the data cube contain the corresponding value of a *measured attribute*. In [GP97], we proposed and studied the use of histograms to summarize the multidimensional data cube in order to provide quick and approximate answers to aggregate queries.

An important issue in the use of any synopsis for the data cube is determining a configuration that maximizes estimation accuracy for queries over the data cube, i.e, selecting the attribute combinations to build statistics on and the types of statistics, and allocating the resources (mainly, the available footprint) among them. This is particularly critical for data cube approximation due to the large data volumes and the potentially very high accuracy requirements.

In [GP97], we studied the histogram configuration problem for minimizing the average and maximum errors in answering aggregate queries involving selections and projections. We also studied the dual problem of minimizing the space required to satisfy *a priori* error bounds. We provided efficient space allocation algorithms for many of these problems with guarantees on the quality of their solutions and exploited the interactions between attribute combinations to reduce the space requirements of the configurations. Experiments on the TPC-D and synthetic databases showed that these algorithms result in highly-accurate and concise histogram configurations, and justified the use of histograms for approximate query answering.

7 Improvements in approximate query answering: An example

In this section, we consider a concrete example of how Aqua can provide highly-accurate approximate answers to an important class of queries recognized as difficult to approximate. Specifically, we consider the problem of *hot list* queries, i.e., queries that request an ordered set of ⟨value, count⟩ pairs for the k most frequently occurring data values, for some k.

An example hot list is the top selling items in a database of sales transactions. In various contexts, hot lists of k pairs are denoted as *high-biased histograms* [IC93] of $k+1$ buckets, the first k mode statistics, or the k largest itemsets [AS94]. Hot lists can be maintained on singleton values, pairs of values, triples, etc. — e.g., they can be maintained on c-itemsets for any specified c, and used to produce association rules [AS94, BMUT97]. Hot lists capture the most skewed (i.e., popular) values in a relation, and hence have been shown to be quite useful for estimating predicate selectivities and join sizes (see [Ioa93, IC93, IP95]). In a mapping of values to parallel processors or disks, the most skewed values limit the number of processors or disks for which good load balance can be obtained. Hot lists are also quite useful in data mining contexts for real-time fraud detection in telecommunications traffic [Pre97], and in fact an early version of the algorithm described below has been in use in such contexts for over a year.

Hot list queries can be answered exactly by maintaining a histogram of ⟨value, count⟩ pairs for all distinct values in the data set. However, the number of distinct values can often be quite large, so that most of the histogram should be or must be stored on disk. This implies high update time overheads, as each update to the data set can require a separate disk access to update the histogram. Since the number of distinct values can be on the order of the size of the data set, the histogram can incur a large disk footprint. The response time can be minimized by storing the top k pairs as a synopsis within the approximate answer engine.

Our goal is to provide, to a certain accuracy, an ordered set of ⟨value, count⟩ pairs for the most frequently occurring values in a data set, in potentially orders of magnitude smaller footprint than needed to maintain the counts for all values. Note that the difficulty in incremental maintenance of hot lists is in detecting when values that were infrequent become frequent ("hot") due to a shift in the distribution of the newer data. Such detection is difficult since the synopsis maintains information on only a few values in the data set, in order to remain within the footprint bound.

Approximate answers to hot list queries. In [GM97a], we presented the first low overhead algorithms for providing highly-accurate approximate answers to hot list queries. One algorithm is based on using concise samples and another is based on using counting samples. These are compared with an algorithm using traditional samples. All three algorithms maintain their accuracy in the presence of ongoing insertions to the data warehouse; the algorithm using counting samples maintains its accuracy also in the presence of deletions.

For concise and traditional samples, we report ⟨value, count⟩ pairs for the k pairs with highest counts in the sample, as long as the counts are greater than or equal to a confidence threshold $\delta \geq 1$. If m is the sample-size, then the reported counts are scaled by $|R|/m$. For counting samples, we augment the counts by $\hat{c}$ instead of scaling them, where $\hat{c}$ is roughly half the current threshold τ for the counting sample. This augmentation of the counts serves to compensate for inserts of a value into the data warehouse prior to the successful coin toss that placed it in the counting sample. We report ⟨value, count⟩ pairs for the k pairs with highest counts in the counting sample, as long as the augmented counts are at least τ.

Our analytical and experimental studies in [GM97a] of the three algorithms showed the following:

- Using counting samples is the most accurate, and far superior to using traditional samples; using concise samples falls in between, nearly matching counting samples at high skew but nearly matching traditional samples at very low skew.
- On the other hand, the overheads are the smallest using traditional samples, and the largest using counting samples.

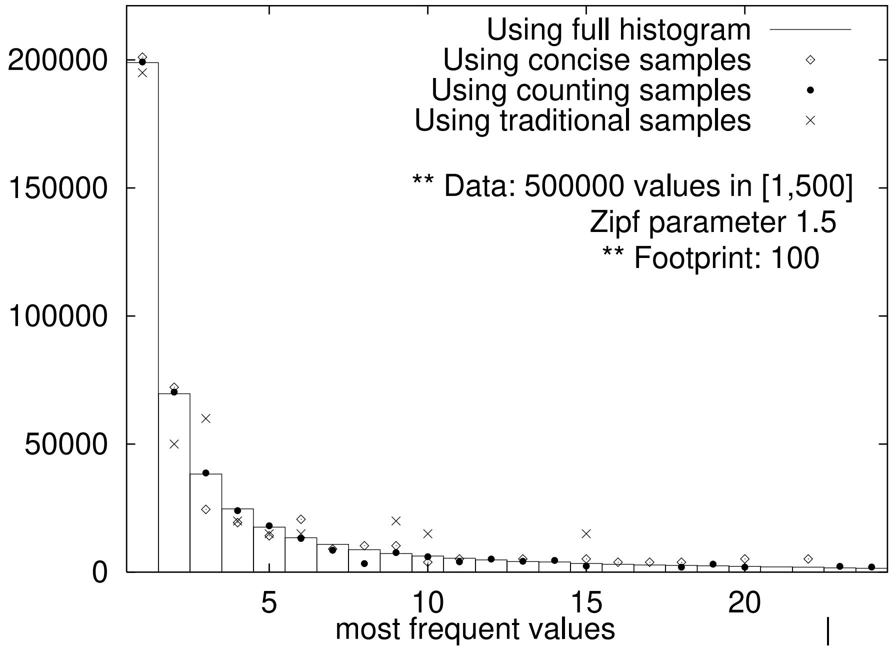

Figure 8: Comparison of algorithms for a hot list query

An example of the relative accuracy of the algorithms is depicted in Figure 8. This figure plots the most frequent values in the data warehouse in order of nonincreasing counts, together with their counts. The x-axis depicts the rank of a value (the actual values are irrelevant here); the y-axis depicts the count for the value with that rank. The exact counts are plotted as histogram boxes.

Our algorithms report less than k values for certain data distributions. The lower bound in Section 5.3 implies that even for $k = 1$, *any* algorithm for answering approximate hot list queries based on a synopsis whose footprint is sublinear in the number of distinct values will fail to be accurate for certain data distributions. Thus in order to report only highly-accurate answers, it is inevitable that less than k values are reported for certain distributions.

Note that the problematic data distributions are the nearly-uniform ones with relatively small maximum frequency (this is the case in which the lower bound in Section 5.3 is proved). Fortunately, it is the skewed distributions, not the nearly-uniform ones, that are of interest, and the algorithms above report good results for skewed distributions.

8 Previous related work

Hellerstein *et al.* [HHW97] proposed a framework for approximate answers of aggregation queries called *online aggregation*, in which the base data is scanned in a certain order and the approximate answer for an aggregation query is updated as the scan proceeds. A graphical display depicts the answer and a (decreasing) confidence interval as the scan proceeds, so that the user may stop the process at any time. Special treatment is given to small sets in group by operations to ensure that tuples in such sets are observed early in the scan order. In the taxonomy of Section 2, this work provides continuous reporting for aggregation queries. The only synopses maintained are the indexes to enable the special treatment of small sets, so the footprint can be quite small. The reported tuples are retrieved from the base data at query time. Thus the response time will be orders of magnitude slower than in Aqua. If the scan order for a group is random, then randomly-selected actual tuples with guaranteed accuracy measures will be reported. Moreover, considering all groups, biased-selected actual tuples will be reported with the bias in favor of the small sets, as desired. The disadvantage of a random scan order is that the response time is even slower. If the scan order is the order of the data on the disks, then the response time is faster than with random order, but now the reported tuples are arbitrary actual tuples with heuristic accuracy measures (which can be quite inaccurate). In summary, the coverage is aggregation queries, the response time is quite slow, the accuracy is good only if a random order scan is used, the update time is fast, and the footprint is small.

Olken and Rotem [OR92] presented techniques for maintaining random sample views. They did not consider concise or counting samples.

Matias *et al.* [MVN93, MVY94, MSY96] proposed and studied *approximate data structures* that provide fast approximate answers. For example, a priority queue data structure supports the operations insert, findmin, and deletemin; their approximate priority queue supports these operations with smaller overheads while reporting an approximate min in response to findmin and deletemin operations. These data structures have linear space footprints.

The design of sampling-based estimation algorithms is a popular area of research [HÖT88, HÖT89, LN89, LN90, LNS90, HÖD91, HS92, LS92, LNSS93, HNSS93, HNS94, LN95, HNSS95, GGMS96]. Results in [LNS90, HÖD91, HS92, HNS94] and elsewhere demonstrate the practicality of estimation procedures based on sampling by showing that the time taken to compute the estimate is a small fraction of the time taken to compute the actual query.

Studies of the relative merits of various types of histograms in estimating selectivities and join sizes include [MO79, Koo80, Chr83, PSC84, KK85, Lyn88, MCS88, MD88, Ioa93, IC93, IP95, PIHS96]. Using histograms for estimating selectivities in multi-dimensional queries was studied in [MD88, PI97].

A number of probabilistic techniques have been previously proposed for various counting problems. Morris [Mor78] (see also [Fla85], [HK95]) showed how to approximate the sum of a set of n values in $[1..m]$ using only $O(\lg \lg m + \lg \lg n)$ bits of memory. Flajolet and Martin [FM83, FM85] designed an algorithm for approximating the number of distinct values in a relation in a single pass through the data and using only $O(\lg n)$ bits of memory. Other algorithms for approximating the number of distinct values in a relation include [WVZT90, HNSS95]. Probabilistic techniques for fast parallel estimation of the size of a set were studied in [Mat92].

None of this previous work uses the new techniques described in this paper.

9 Project status and future directions

We are in the process of implementing the base Aqua system. A simple query processor has been implemented for comparing approximate answers against actual answers.

We have also begun work on various enhancements beyond the base Aqua system. We have implemented our new techniques for maintaining samples (Section 5.1), approximate equi-depth and Compressed histograms (Section 5.2), and second frequency moments (Section 5.3). We have also implemented concise samples (Section 6.1) and counting samples (Section 6.2) and their uses in hot list queries (Section 7). Finally, we have implemented the histogram techniques described in Section 6.3 and Section 6.4.

Techniques for adding skew to the TPC-D benchmark data have been developed and implemented, to test the effectiveness of our techniques under the more realistic scenario in which the data is skewed.

The next steps are:

1. Complete the implementation of the base Aqua system.
2. Determine an accuracy measure calculus for the base Aqua system.
3. Measure the accuracy of the base Aqua system on the TPC-D benchmark queries.
4. Incorporate the implemented enhancements into the Aqua system.
5. Measure the accuracy of the enhanced Aqua system on the TPC-D benchmark queries, quantifying the effectiveness of the various enhancements relative to each other and the base system.

Future work includes:

- Look at each TPC-D query and see where additional types of synopses (either known ones or new ones) would improve accuracy and coverage.
- Develop and study improved techniques for maintaining important synopses with lower overheads.

Acknowledgements

The Aqua project is part of the umbrella **Databases Incorporating Guaranteed EStimation Techniques (DIGEST)** project by the authors (http://www.bell-labs.com/project/digest/). Noga Alon and Mario Szegedy were collaborators on the work reported in Section 5.3. S. Muthukrishnan and Torsten Suel were collaborators on the work reported in Section 6.3. Venkatesh Ganti was a collaborator on the work reported in Section 6.4.

References

[AGMS96] N. Alon, P. B. Gibbons, Y. Matias, and M. Szegedi. Dynamic, probabilistic maintenance of self-join sizes in limited storage. Manuscript, March 1996.

[AMS96] N. Alon, Y. Matias, and M. Szegedi. The space complexity of approximating the frequency moments. In *Proc. 28th ACM Symp. on the Theory of Computing*, pages 20–29, May 1996.

[AS94] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proc. 20th International Conf. on Very Large Data Bases*, pages 487–499, September 1994.

[BMUT97] S. Brin, R. Motwani, J. D. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. In *Proc. ACM SIGMOD International Conf. on Management of Data*, pages 255–264, May 1997.

[Chr83] S. Christodoulakis. Estimating record selectivities. *Information Systems*, 8(2):105–115, 1983.

[FJS97] C. Faloutsos, H. V. Jagadish, and N. D. Sidiropoulos. Recovering information from summary data. In *Proc. 23rd International Conf. on Very Large Data Bases*, pages 36–45, August 1997.

[Fla85] P. Flajolet. Approximate counting: a detailed analysis. *BIT*, 25:113–134, 1985.

[FM83] P. Flajolet and G. N. Martin. Probabilistic counting. In *Proc. 24th IEEE Symp. on Foundations of Computer Science*, pages 76–82, October 1983.

[FM85] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *J. Computer and System Sciences*, 31:182–209, 1985.

[GBLP96] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tabs, and sub-totals. In *Proc. 12th IEEE International Conf. on Data Engineering*, pages 152–159, February-March 1996.

[GGMS96] S. Ganguly, P. B. Gibbons, Y. Matias, and A. Silberschatz. Bifocal sampling for skew-resistant join size estimation. In *Proc. 1996 ACM SIGMOD International Conf. on Management of Data*, pages 271–281, June 1996.

[GM95] P. B. Gibbons and Y. Matias, August 1995. Presentation and feedback during a Bell Labs-Teradata presentation to Walmart scientists and executives on proposed improvements to the Teradata DBS.

[GM97a] P. B. Gibbons and Y. Matias. New sampling-based summary statistics for improving approximate query answers. Technical report, Bell Laboratories, Murray Hill, New Jersey, November 1997.

[GM97b] P. B. Gibbons and Y. Matias. Synopsis data structures, concise samples, and mode statistics. Manuscript, July 1997.

[GMP97] P. B. Gibbons, Y. Matias, and V. Poosala. Fast incremental maintenance of approximate histograms. In *Proc. 23rd International Conf. on Very Large Data Bases*, pages 466–475, August 1997.

[GP97] V. Ganti and V. Poosala. Space-efficient approximation of the data cube. Technical report, Bell Laboratories, Murray Hill, New Jersey, November 1997.

[HHW97] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *Proc. ACM SIGMOD International Conf. on Management of Data*, pages 171–182, May 1997.

[HK95] M. Hofri and N. Kechris. Probabilistic counting of a large number of events. Manuscript, 1995.

[HNS94] P. J. Haas, J. F. Naughton, and A. N. Swami. On the relative cost of sampling for join selectivity estimation. In *Proc. 13th ACM Symp. on Principles of Database Systems*, pages 14–24, May 1994.

[HNSS93] P. J. Haas, J. F. Naughton, S. Seshadri, and A. N. Swami. Fixed-precision estimation of join selectivity. In *Proc. 12th ACM Symp. on Principles of Database Systems*, pages 190–201, May 1993.

[HNSS95] P. J. Haas, J. F. Naughton, S. Seshadri, and L. Stokes. Sampling-based estimation of the number of distinct values of an attribute. In *Proc. 21st International Conf. on Very Large Data Bases*, pages 311–322, September 1995.

[HÖD91] W.-C. Hou, G. Özsoyoğlu, and E. Dogdu. Error-constrained COUNT query evaluation in relational databases. In *Proc. ACM SIGMOD International Conf. on Management of Data*, pages 278–287, May 1991.

[HÖT88] W.-C. Hou, G. Özsoyoğlu, and B. K. Taneja. Statistical estimators for relational algebra expressions. In *Proc. 7th ACM Symp. on Principles of Database Systems*, pages 276–287, March 1988.

[HÖT89] W.-C. Hou, G. Özsoyoğlu, and B. K. Taneja. Processing aggregate relational queries with hard time constraints. In *Proc. ACM SIGMOD International Conf. on Management of Data*, pages 68–77, June 1989.

[HS92] P. J. Haas and A. N. Swami. Sequential sampling procedures for query size estimation. In *Proc. ACM SIGMOD International Conf. on Management of Data*, pages 1–11, June 1992.

[IC93] Y. E. Ioannidis and S. Christodoulakis. Optimal histograms for limiting worst-case error propagation in the size of join results. *ACM Transactions on Database Systems*, 18(4):709–748, 1993.

[Ioa93] Y. E. Ioannidis. Universality of serial histograms. In *Proc. 19th International Conf. on Very Large Data Bases*, pages 256–267, August 1993.

[IP95] Y. E. Ioannidis and V. Poosala. Balancing histogram optimality and practicality for query result size estimation. In *Proc. ACM SIGMOD International Conf. on Management of Data*, pages 233–244, May 1995.

[KK85] N. Kamel and R. King. A model of data distribution based on texture analysis. In *Proc. ACM SIGMOD International Conf. on Management of Data*, pages 319–325, May 1985.

[Koo80] R. P. Kooi. *The Optimization of Queries in Relational Databases.* PhD thesis, Case Western Reserve University, September 1980.

[LN89] R. J. Lipton and J. F. Naughton. Estimating the size of generalized transitive closures. In *Proc. 15th International Conf. on Very Large Data Bases*, pages 165–172, August 1989.

[LN90] R. J. Lipton and J. F. Naughton. Query size estimation by adaptive sampling. In *Proc. 9th ACM Symp. on Principles of Database Systems*, pages 40–46, April 1990.

[LN95] R. J. Lipton and J. F. Naughton. Query size estimation by adaptive sampling. *J. Computer and System Sciences*, 51(1):18–25, 1995.

[LNS90] R. J. Lipton, J. F. Naughton, and D. A. Schneider. Practical selectivity estimation through adaptive sampling. In *Proc. ACM SIGMOD International Conf. on Management of Data*, pages 1–12, May 1990.

[LNSS93] R. J. Lipton, J. F. Naughton, D. A. Schneider, and S. Seshadri. Efficient sampling strategies for relational database operations. *Theoretical Computer Science*, 116(1-2):195–226, 1993.

[LS92] Y. Ling and W. Sun. A supplement to sampling-based methods for query size estimation in a database system. *SIGMOD Record*, 21(4):12–15, 1992.

[Lyn88] C. A. Lynch. Selectivity estimation and query optimization in large databases with highly skewed distributions of column values. In *Proc. 14th International Conf. on Very Large Data Bases*, pages 240–251, August 1988.

[Mat92] Y. Matias. *Highly Parallel Randomized Algorithmics.* PhD thesis, Tel Aviv University, Israel, 1992.

[MCS88] M. V. Mannino, P. Chu, and T. Sager. Statistical profile estimation in database systems. *ACM Computing Surveys*, 20(3):191–221, 1988.

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

[MO79] T. H. Merrett and E. J. Otoo. Distribution models of relations. In *Proc. 5th International Conf. on Very Large Data Bases*, pages 418–425, October 1979.

[Mor78] R. Morris. Counting large numbers of events in small registers. *Communications of the ACM*, 21:840–842, 1978.

[MPS97a] S. Muthukrishnan, V. Poosala, and T. Suel. Highly efficient partitioning schemes for approximating low-dimensional data. Manuscript, November 1997.

[MPS97b] S. Muthukrishnan, V. Poosala, and T. Suel. On histogram-based selectivity estimation with bounded errors. Manuscript, November 1997.

[MSY96] Y. Matias, S. C. Sahinalp, and N. E. Young. Performance evaluation of approximate priority queues. Presented at *DIMACS Fifth Implementation Challenge: Priority Queues, Dictionaries, and Point Sets*, organized by D. S. Johnson and C. McGeoch, October 1996.

[MVN93] Y. Matias, J. S. Vitter, and W.-C. Ni. Dynamic generation of discrete random variates. In *Proc. 4th ACM-SIAM Symp. on Discrete Algorithms*, pages 361–370, January 1993.

[MVY94] Y. Matias, J. S. Vitter, and N. E. Young. Approximate data structures with applications. In *Proc. 5th ACM-SIAM Symp. on Discrete Algorithms*, pages 187–194, January 1994.

[OR92] F. Olken and D. Rotem. Maintenance of materialized views of sampling queries. In *Proc. 8th IEEE International Conf. on Data Engineering*, pages 632–641, February 1992.

[PI97] V. Poosala and Y. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *Proc. 23rd International Conf. on Very Large Data Bases*, pages 486–495, August 1997.

[PIHS96] V. Poosala, Y. E. Ioannidis, P. J. Haas, and E. J. Shekita. Improved histograms for selectivity estimation of range predicates. In *Proc. ACM SIGMOD International Conf. on Management of Data*, pages 294–305, June 1996.

[Pre97] D. Pregibon. Mega-monitoring: Developing and using telecommunications signatures, October 1997. Invited talk at the *DIMACS Workshop on Massive Data Sets in Telecommunications*.

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

[Sch97] D. Schneider. The ins & outs (and everything in between) of data warehousing, August 1997. Tutorial in the *23rd International Conf. on Very Large Data Bases.*

[Vit85] J. S. Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software*, 11(1):37–57, 1985.

[WVZT90] K.-Y. Whang, B. T. Vander-Zanden, and H. M. Taylor. A linear-time probabilistic counting algorithm for database applications. *ACM Transactions on Database Systems*, 15(2):208–229, 1990.