DIMACS Series in Discrete Mathematics
and Theoretical Computer Science

# Synopsis Data Structures for Massive Data Sets

Phillip B. Gibbons and Yossi Matias

Abstract. Massive data sets with terabytes of data are becoming commonplace. There is an increasing demand for algorithms and data structures that provide fast response times to queries on such data sets. In this paper, we describe a context for algorithmic work relevant to massive data sets and a framework for evaluating such work. We consider the use of "synopsis" data structures, which use very little space and provide fast (typically approximated) answers to queries. The design and analysis of effective synopsis data structures offer many algorithmic challenges. We discuss a number of concrete examples of synopsis data structures, and describe fast algorithms for keeping them up-to-date in the presence of online updates to the data sets.

## 1. Introduction

A growing number of applications demand algorithms and data structures that enable the efficient processing of data sets with gigabytes to terabytes to petabytes of data. Such massive data sets necessarily reside on disks or tapes, making even a few accesses of the base data set comparably slow (e.g., a single disk access is often $10,000$ times slower than a single memory access). For fast processing of queries to such data sets, disk accesses should be minimized.

This paper focuses on data structures for supporting queries to massive data sets, while minimizing or avoiding disk accesses. In particular, we advocate and study the use of small space data structures. We denote as *synopsis data structures* any data structures that are substantively smaller than their base data sets. Synopsis data structures have the following advantages over non-synopsis data structures:

- *Fast processing*: A synopsis data structure may reside in main memory, providing for fast processing of queries and of data structure updates, by avoiding disk accesses altogether.
- *Fast swap/transfer*: A synopsis data structure that resides on the disks can be swapped in and out of memory with minimal disk accesses, for the purposes of processing queries or updates. A synopsis data structure can

1991 *Mathematics Subject Classification.* Primary 68P05, 68P20; Secondary 68Q05, 68Q20, 62-04, 68Q25.

Research of the second author was supported in part by an Alon Fellowship, by the Israel Science Foundation founded by The Academy of Sciences and Humanities, and by the Israeli Ministry of Science.

be pushed or pulled remotely (e.g., over the internet) at minimal cost, since the amount of data to be transferred is small.

- *Lower cost*: A synopsis data structure has a minimal impact on the overall space requirements of the data set and its supporting data structures, and hence on the overall cost of the system.
- *Better system performance*: A synopsis data structure leaves space in the memory for other data structures. More importantly, it leaves space for other processing, since most processing that involves the disks uses the memory as a cache for the disks. In a data warehousing environment, for example, the main memory is needed for query-processing working space (e.g., building hash tables for hash joins) and for caching disk blocks. The importance of available main memory for algorithms can be seen from the external memory algorithms literature, where the upper and lower time bounds for many fundamental problems are inversely proportional to the logarithm of the available memory size [**Vit98**]. (See Section 2.3 for examples.) Thus although machines with large main memories are becoming increasingly commonplace, memory available for synopsis data structures remains a precious resource.
- *Small surrogate*: A synopsis data structure can serve as a small surrogate for the data set when the data set is currently expensive or impossible to access.

In contrast, linear space data structures for massive data sets can not reside in memory, have very slow swap and transfer times, can increase the space requirements and hence the overall cost of the system by constant factors, can hog the memory when they are in use, and can not serve as a small surrogate. Hence a traditional viewpoint in the algorithms literature — that a linear space data structure is a good one — is not appropriate for massive data sets, as such data structures often fail to provide satisfactory application performance.

On the other hand, since synopsis data structures are too small to maintain a full characterization of their base data sets, they must summarize the data set, and the responses they provide to queries will typically be approximate ones. The challenges are to determine (1) what synopsis of the full data set to keep in the limited space in order to maximize the accuracy and confidence of its approximate responses, and (2) how to efficiently compute the synopsis and maintain it in the presence of updates to the data set.

Due to their importance in applications, there are a number of synopsis data structures in the literature and in existing systems. Examples include uniform and biased random samples, various types of histograms, statistical summary information such as frequency moments, data structures resulting from lossy compression of the data set, etc. Often, synopsis data structures are used in a heuristic way, with no formal properties proved on their performance or accuracy, especially under the presence of updates to the data set. Our ongoing work since 1995 seeks to provide a systematic study of synopsis data structures, including the design and analysis of synopsis data structures with performance and accuracy guarantees, even in the presence of data updates.

In this paper, we describe a context for algorithmic work relevant to massive data sets and a framework for evaluating such work. In brief, we combine the PDM external memory model [**VS94**] with input/output conventions more typical for the study of (online) data structure problems. Two general scenarios are considered: one where the input resides on the disks of the PDM and one where the input arrives online in the PDM memory. We describe some of our work on synopsis data structures, and highlight results on three problem domains from the database literature: frequency moments, hot list queries, and histograms and quantiles.

*Outline.* Section 2 describes our framework in detail. Results on frequency moments, hot list queries and histograms are described in Sections 3, 4, and 5, respectively. Related work and further results are discussed in Section 6.

## 2. Framework

In this section, we first describe a context for data structure problems for massive data sets. We then introduce synopsis data structures and present a cost model for their analysis. Finally, we discuss two example application domains.

**2.1. Problem set-up.** In the data structure questions we consider, there are a number of data sets, $S_1, S_2, \ldots, S_\ell$, and a set of query classes, $Q_1, \ldots, Q_k$, on these data sets. The query classes are given a priori, and may apply to individual data sets or to multiple data sets. Data structure performance is analyzed on a model of computation that distinguishes between two types of storage, fast and slow, where the fast storage is of limited size. We equate the fast storage with the computer system's main memory and the slow storage with its disks, and use a relevant model of computation (details are in Section 2.3). However, the framework and results in this paper are also relevant to scenarios where (1) the fast storage is the disks and the slow storage is the tapes, or (2) the fast storage is the processor cache memory and the slow storage is the main memory.

In the *static* or *offline* scenario, the data sets are given as input residing on the disks. Given a class of queries $Q$, the goal is to design a data structure for the class $Q$ that minimizes the response time to answer queries from $Q$, maximizes the accuracy and confidence of the answers, and minimizes the preprocessing time needed to build the data structure.

In the *dynamic* or *online* scenario, which models the ongoing loading of new data into the data set, the data sets arrive online in the memory, and are stored on the disks. Specifically, the input consists of a sequence of operations that arrive online to be processed by the data structure, where an operation is either an insertion of a new data item, a deletion of an existing data item, or a query. Given a class of queries $Q$, the goal is to design a data structure for the class $Q$ that minimizes the response time to answer queries from $Q$, maximizes the accuracy and confidence of the answers, and minimizes the update time needed to maintain the data structure. As we are interested in the *additional* overheads for maintaining the data structure, there is no charge for updating the data sets.

This set-up reflects many environments for processing massive data sets. For example, it reflects most data warehousing environments, such as Walmart's multiterabyte warehouse of its sales transactions. For most data sets, there are far more insertions than deletions. An important exception is a "sliding-window" data set, comprised of the most recent data from a data source (such as the last 15 months of sales transactions). In such data sets, batches of old data are periodically deleted to make room for new data, making the number of insertions comparable to the number of deletions.

To handle many data sets and many query classes, a large number of synopsis data structures may be needed. Thus we will assume that when considering any one data structure problem in isolation, the amount of memory available to the data structure is a small fraction of the total amount of memory. We evaluate the effectiveness of a data structure as a function of its space usage or *footprint*. For example, it is common practice to evaluate the effectiveness of a histogram in range selectivity queries as a function of its footprint (see, e.g., [**PIHS96**]).

Finally, note that in some online environments, the data set is not stored alongside with the data structure, but instead resides in a remote computer system that may be currently unavailable [**FJS97**]. In such cases, the online view of the data is effectively the only view of the data used to maintain the data structure and answer queries. We denote this scenario the *purely online* scenario.

**2.2. Synopsis data structures.** The above set-up motivates the need for data structures with small footprints. We denote as *synopsis data structures* any data structures that are substantively smaller than their base data sets. Since such data structures are often too small to maintain a full characterization of their base data sets with regards to a class of queries, the responses they provide to queries will typically be approximate ones. Synopsis data structures seek to characterize the data using succinct representations.

A natural synopsis data structure is a uniform random sample, and indeed, it is well known that a random sample of a data set can be quite useful to support a variety of queries on the set. However, for many classes of queries, uniform sampling is not the best choice. A trivial example is the class of "number of items in the set" queries, for which a single counter is much better. More interesting examples can be found in the rest of this paper.

We define an $f(n)$-synopsis data structure as follows.

DEFINITION 2.1. An $f(n)$-*synopsis data structure* for a class $Q$ of queries is a data structure for providing (exact or approximate) answers to queries from $Q$ that uses $O(f(n))$ space for a data set of size $n$, where $f(n) = o(n^\epsilon)$ for some constant $\epsilon < 1$.

While any sublinear space data structure may be an important improvement over a linear space data structure, the above definition demands at least a polynomial savings in space, since only with such savings can most of the benefits of synopsis data structures outlined in Section 1 be realized. For example, massive data sets typically exceed the available memory size by a polynomial factor, so a data structure residing in memory must have a $o(n^\epsilon)$ footprint.

As with traditional data structures, a synopsis data structure can be evaluated according to five metrics:

- *Coverage*: the range and importance of the queries in $Q$.
- *Answer quality*: the accuracy and confidence of its (approximate) answers to queries in $Q$.
- *Footprint*: its space bound (smaller $f(n)$ is better).
- *Query time*.
- *Computation/Update time*: its preprocessing time in the static scenario, or its update time in the dynamic scenario.

Ideally, $f(n)$ is $\log^2 n$ or better, queries and updates require a constant number of memory operations and no disks operations, and the answers are exact.

**2.3. Cost model.** Query times and computation/update times can be analyzed on any of a number of models of computation, depending on the target computer system, including parallel or distributed models. For concreteness in this paper, we will use the parallel disk model (PDM) of Vitter and Shriver [**VS94, Vit98**], adapted to the scenarios discussed above.

In the PDM, there are $P$ processors, $D$ disks, and an (internal) memory of size $M$ (i.e., $M/P$ per processor). Each disk is partitioned into blocks of size $B$, and is of unbounded size. The input of size $N$ is partitioned (striped) evenly among the disks, $D_0, D_1, \ldots, D_{D-1}$, such that for $i = 0, 1, \ldots, N/B - 1$, the $i$th block of input data is the $\lfloor i/D \rfloor$th block of data on the $(i \bmod D)$th disk. The output is required to be similarly striped. The size parameters $N$, $M$, and $B$ are in units of the input data items, $M$ is less than $N$, and $1 \leq DB \leq M/2$. Thus the internal memory is too small to hold the input but sufficiently large to hold two blocks from each of the disks.

Algorithms are analyzed based on three metrics: the number of I/O operations, the processing time, and the amount of disk space used. In a single I/O read (I/O write), each of the $D$ disks can simultaneously transfer a block to (from, respectively) the internal memory. The processing time is analyzed assuming that each processor is a unit-cost RAM for its in-memory computation times, and that the processors are connected by an interconnection network whose properties depend on the setting. Most of the algorithmic work on the PDM has focused on reducing the number of I/O operations and proving matching lower bounds. As mentioned in the introduction, the I/O bounds are often inversely proportional to the logarithm of the available memory; specifically, they are inversely proportional to $\log(M/B)$. Examples discussed in [**Vit98**] include sorting, permuting, matrix transpose, computing the Fast Fourier Transform, and various batched problems in computational geometry. For other problems, such as matrix multiplication and LU factorization, the I/O bounds are inversely proportional to $\sqrt{M}$.

Our main deviation from the PDM is in the input and output requirements. Query times and computation/update times are analyzed on a PDM with input and output requirements adapted to the set-up described in Section 2.1. Our first deviation is to supplement the PDM with a write-only "output" memory, of unbounded size.

In our static scenario, the input resides on the disks as in the PDM, but we are allowed to preprocess the data and store the resulting data structures in the internal memory. In response to a query, the output is written to the output memory, in contrast to the PDM. Thus processing the query may incur no I/O operations.

In our dynamic scenario, the input arrives online in the internal memory in the form of insertions to the data set, deletions from the data set, or queries. Data structures are maintained for answering queries. As in the static scenario, data structures may be stored in the internal memory and responses to queries are written to the output memory, and hence queries may be answered without incurring any I/O operations.

Reducing the number of I/O operations is important since, as pointed out in Section 1, an I/O operation can take as much time as $10,000$ in-memory operations on modern computer systems.

Note that any insertions and deletions in the dynamic scenario are applied to the base data set, at no charge, so that the current state of the data set resides on the disks at all times. However, the cost of reading this data depends on the setting, and needs to be specified for algorithms that perform such reads. One can consider a variety of settings, such as cases where the base data is striped across the disks or cases where there are various indices such as B-trees that can be exploited. For the purely online scenario, the base data is unavailable.

With massive data sets, it will often be the case that the input size $N$ is not just larger than the memory size $M$, as assumed by the PDM, but is in fact polynomially larger: $N = M^c$, for a constant $c > 1$. Also, note that any algorithm for the dynamic scenario in which updates incur (amortized) processing time $t$ per update and no I/O operations yields an algorithm for computing the same synopsis data structure in the static scenario in one pass over the data set, i.e., $\frac{N}{DB}$ I/O operations and $Nt$ processing time.

For simplicity, in the remainder of this paper, we will assume that the PDM has only a single processor (i.e., $P = 1$).

**2.4. Applications: Approximate query answering and cost estimation.** An important application domain for synopsis data structures is approximate query answering for ad hoc queries of large data warehouses [**GM98**]. In large data recording and warehousing environments, it is often advantageous to provide fast, approximated answers to complex decision-support queries (see the TPC-D benchmark [**TPC**] for examples of such 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.

In the *Approximate query answering (Aqua)* project [**GMP97a, GPA$^+$98**] at Bell Labs, we seek to provide fast, approximate answers to queries using synopsis data structures. Unlike the traditional data warehouse set-up depicted in Figure 1, in which each query is answered exactly using the data warehouse, Aqua considers the set-up depicted in Figure 2. In this set-up, new data being loaded into the data warehouse is also observed by the approximate answer engine. This engine maintains various synopsis data structures, for use in answering queries.

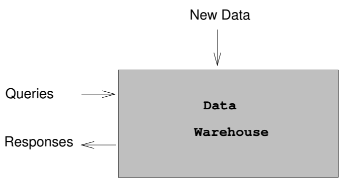


FIGURE 1. Traditional data warehouse

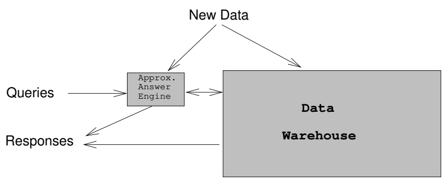


FIGURE 2. Data warehouse set-up for providing approximate query answers

Queries are sent to the approximate answer engine. Whenever possible, the engine promptly returns a response consisting of an approximated answer and a confidence measure (e.g., a 95% confidence interval). The user can then decide whether or not to have an exact answer computed from the base data, based on the user's desire for the exact answer and the estimated time for computing an exact answer as determined by the query optimizer and/or the approximate answer engine. There are a number of scenarios for which a user may prefer an approximate answer in a few seconds over an exact answer that requires tens of minutes or more to compute, e.g., during a drill down query sequence in data mining [**SKS97**].

Moreover, as discussed in Section 2.1, sometimes the base data is remote and currently unavailable, so that an exact answer is not an option, until the data again becomes available.

Another important application domain for synopsis data structures is cost estimation within a query optimizer. In commercial database systems, limited storage is set aside for synopsis data structures such as histograms; these are used by the query optimizer to estimate the cost of the primitive operations comprising a complex SQL query (i.e., estimates of the number of items that satisfy a given predicate, estimates of the size of a join operation [**GGMS96**], etc.). The query optimizer uses these cost estimates to decide between alternative query plans and to make more informed scheduling (allocation, load balancing, etc.) decisions in multi-query and/or multiprocessor database environments, in order to minimize query response times and maximize query throughput.

These two application domains highlight the fact that good synopsis data structures are useful either for providing fast approximate answers to user queries, or for speeding up the time to compute an exact answer, or for both.

The next three sections of this paper highlight in detail our work on synopsis data structures for three problem domains. These sections are not meant to be comprehensive, but instead provide a flavor of the difficulties and the techniques. Much of the details, including most of the proofs and the experimental results, are omitted; the reader is referred to the cited papers for these details. The first problem domain is that of estimating the frequency moments of a data set, such as the number of distinct values or the maximum frequency of any one value. The second problem domain is that of estimating the $m$ most frequently occurring values in a data set. The third problem domain is that of approximating the quantiles and other types of histograms of a data set. Note that the emphasis in these sections will be on what synopses to keep within the limited space, how to maintain these synopses, and what can be proved about the quality of the answers they provide; these are the challenges particular to synopsis data structures. Traditional data structure issues concerning the representation used to store the synopsis and its impact on query time and update time are important, but somewhat secondary to the main emphasis. As can be seen by the techniques presented in these sections, randomization and approximation seem to be essential features in the study of synopsis data structures for many problems, and have been proven to be essential for several problems.

## 3. Frequency moments

In this section, we highlight our results on synopsis data structures for estimating the frequency moments of a data set.

Let $A = (a_1, a_2, \ldots, a_n)$ be a sequence of elements, where each $a_i$ is a member of $U = \{1, 2, \ldots, u\}$. For simplicity of exposition, we assume that $u \leq n$.[1] Let $m_i = |\{j : a_j = i\}|$ denote the number of occurrences of $i$ in the sequence $A$, or the *frequency* of $i$ in $A$. The demographic information of the frequencies in the data set $A$ can be described by maintaining the full histogram over $U$: $H = (m_1, m_2, \ldots, m_u)$. However, when the desired footprint is substantially smaller than $u$, then a more succinct representation of the frequencies is required.

[1] A more detailed analysis would show that for the $f(n)$-synopsis data structures reported in this section, it suffices that $u \leq 2^{n^\epsilon}$ for some constant $\epsilon < 1$.

Define, for each $k \geq 0$, $F_k = \sum_{i=1}^{u} m_i^k$. In particular, $F_0$ is the number of distinct elements appearing in the sequence, $F_1$ ( $= n$) is the length of the sequence, and $F_2$ is the *repeat rate* or *Gini's index of homogeneity* needed in order to compute the *surprise index* of the sequence (see, e.g., [**Goo89**]). Also define $F_\infty^* = \max_{1 \leq i \leq u} m_i$. (Since the moment $F_k$ is defined as the sum of $k$-powers of the numbers $m_i$ and not as the $k$-th root of this sum the last quantity is denoted by $F_\infty^*$ and not by $F_\infty$.) The numbers $F_k$ are called the *frequency moments* of $A$.

The frequency moments of a data set represent useful demographic information about the data, for instance in the context of database applications. They indicate the degree of *skew* in the data, which is of major consideration in many parallel database applications. Thus, for example, the degree of the skew may determine the selection of algorithms for data partitioning, as discussed by DeWitt *et al* [**DNSS92**] (see also the references therein).

We discuss the estimation of frequency moments when the available memory is smaller than $u$ (i.e., when the full histogram $H$ is not available). We first consider the problem of estimating $F_0$, which demonstrates the advantages of viewing the input online versus ad hoc sampling from the data set. In particular, we present results showing that $F_0$ can be effectively estimated using a synopsis data structure with footprint only $O(\log u)$, but it cannot be effectively estimated based solely on a random sample of the data set unless $\Omega(u)$ memory is employed. We then discuss space-efficient algorithms for estimating $F_k$ for all $k \geq 2$, using $(n^{1-1/k} \log n)$-synopsis data structures, and an improved $(\log n)$-synopsis data structure for estimating $F_2$. Finally, lower bounds on the estimation of $F_k$ and $F_\infty^*$ are mentioned, as well as results showing that that both randomization and approximation are essential for evaluating $F_k$, $k \neq 1$.

**3.1. Estimating the number of distinct values.** Estimating the number of distinct values in a data set is a problem that frequently occurs in database applications, and in particular as a subproblem in query optimization. Indeed, Haas *et al* [**HNSS95**] claim that virtually all query optimization methods in relational and object-relational database systems require a means for assessing the number of distinct values of an attribute in a relation, i.e., the function $F_0$ for the sequence consisting of the attribute values for each item in the relation.

When no synopsis data structure is maintained, then the best methods for estimating $F_0$ are based on sampling. Haas *et al* [**HNSS95**] consider sampling-based algorithms for estimating $F_0$. They propose a hybrid approach in which the algorithm is selected based on the degree of skew of the data, measured essentially by the function $F_2$. However, they observe that fairly poor performance is obtained when using the standard statistical estimators, and remark that estimating $F_0$ via sampling is a hard and relatively unsolved problem. This is consistent with Olken's assertion [**Olk93**] that all known estimators give large errors on at least some data sets. In a recent paper, Chaudhuri *et al* [**CMN98**] show that "large error is unavoidable even for relatively large samples *regardless of the estimator used.* That is, there does not exist an estimator which can guarantee reasonable error with any reasonable probability unless the sample size is very close to the size of the database itself." Formally, they show the following.

THEOREM 3.1. [**CMN98**] *Consider any estimator $\hat{d}$ for the number of distinct values $d$ based on a random sample of size $r$ from a relation with $n$ tuples. Let the error of the estimator $\hat{d}$ be defined as $error(\hat{d}) = \max\{\hat{d}/d, d/\hat{d}\}$. Then for any*

$\gamma > e^{-r}$, *there exists a choice of the relation such that with probability at least* $\gamma$, $error(\hat{d}) \geq \sqrt{\frac{n \ln 1/\gamma}{r}}$.

In contrast, the algorithm given below demonstrates a $(\log n)$-synopsis data structure which enables estimation of $F_0$ within an arbitrary fixed error bound with high probability, *for any given data set.* Note that the synopsis data structure is maintained while observing the entire data set. In practice, this can be realized while the data set is loaded into the disks, and the synopsis data structure is maintained in main memory with very small overhead.

Flajolet and Martin [**FM83, FM85**] described a randomized algorithm for estimating $F_0$ using only $O(\log u)$ memory bits, and analyzed its performance assuming one may use in the algorithm an explicit family of hash functions which exhibits some ideal random properties. The $(\log n)$-synopsis data structure consists of a bit vector $V$ initialized to all 0. The main idea of the algorithm is to let each item in the data set select at random a bit in $V$ and set it to 1, with (quasi-)geometric distribution; i.e., $V[i]$ is selected with probability $(\approx)$ $1/2^i$. The selection is made using a random hash function, so that all items of the same value will make the same selection. As a result, the expected number of items selecting $V[i]$ is $\approx F_0/2^i$, and therefore $2^{i'}$, where $i'$ is the largest $i$ such that $V[i] = 1$, is a good estimate for $F_0$. Alon *et al* [**AMS96**] adapted the algorithm so that linear hash functions could be used instead, obtaining the following.

THEOREM 3.2. [**FM83, AMS96**] *For every* $c > 2$ *there exists an algorithm that, given a sequence* $A$ *of* $n$ *members of* $U = \{1, 2, \ldots, u\}$, *computes a number* $Y$ *using* $O(\log u)$ *memory bits, such that the probability that the ratio between* $Y$ *and* $F_0$ *is not between* $1/c$ *and* $c$ *is at most* $2/c$.

PROOF. Let $d$ be the smallest integer so that $2^d > u$, and consider the members of $U$ as elements of the finite field $F = GF(2^d)$, which are represented by binary vectors of length $d$. Let $a$ and $b$ be two random members of $F$, chosen uniformly and independently. When a member $a_i$ of the sequence $A$ appears, compute $z_i = a \cdot a_i + b$ , where the product and addition are computed in the field $F$. Thus $z_i$ is represented by a binary vector of length $d$. For any binary vector $z$, let $\rho(z)$ denote the largest $r$ so that the $r$ rightmost bits of $z$ are all 0 and let $r_i = \rho(z_i)$. Let $R$ be the maximum value of $r_i$, where the maximum is taken over all elements $a_i$ of the sequence $A$. The output of the algorithm is $Y = 2^R$. Note that in order to implement the algorithm we only have to keep (besides the $d = O(\log u)$ bits representing an irreducible polynomial needed in order to perform operations in $F$) the $O(\log u)$ bits representing $a$ and $b$ and maintain the $O(\log \log u)$ bits representing the current maximum $r_i$ value.

Suppose, now, that $F_0$ is the correct number of distinct elements that appear in the sequence $A$, and let us estimate the probability that $Y$ deviates considerably from $F_0$. The only two properties of the random mapping $f(x) = ax + b$ that maps each $a_i$ to $z_i$ we need is that for every fixed $a_i$, $z_i$ is uniformly distributed over $F$ (and hence the probability that $\rho(z_i) \geq r$ is precisely $1/2^r$), and that this mapping is pairwise independent. Thus, for every fixed distinct $a_i$ and $a_j$, the probability that $\rho(z_i) \geq r$ and $\rho(z_j) \geq r$ is precisely $1/2^{2r}$.

Fix an $r$. For each element $x \in U$ that appears at least once in the sequence $A$, let $W_x$ be the indicator random variable whose value is 1 if and only if $\rho(ax+b) \geq r$. Let $Z = Z_r = \sum W_x$, where $x$ ranges over all the $F_0$ elements $x$ that appear in the sequence $A$. By linearity of expectation and since the expectation of each $W_x$ is $1/2^r$, the expectation $\mathrm{E}(Z)$ of $Z$ is $F_0/2^r$. By pairwise independence, the variance of $Z$ is $F_0\frac{1}{2^r}(1-\frac{1}{2^r}) < F_0/2^r$. Therefore, by Markov's Inequality, if $2^r > cF_0$ then $\mathrm{Prob}(Z_r > 0) < 1/c$, since $\mathrm{E}(Z_r) = F_0/2^r < 1/c$. Similarly, by Chebyshev's Inequality, if $c2^r < F_0$ then $\mathrm{Prob}(Z_r = 0) < 1/c$, since $\mathrm{Var}(Z_r) < F_0/2^r = \mathrm{E}(Z_r)$ and hence $\mathrm{Prob}(Z_r = 0) \leq \mathrm{Var}(Z_r)/(\mathrm{E}(Z_r)^2) < 1/\mathrm{E}(Z_r) = 2^r/F_0$. Since our algorithm outputs $Y = 2^R$, where $R$ is the maximum $r$ for which $Z_r > 0$, the two inequalities above show that the probability that the ratio between $Y$ and $F_0$ is not between $1/c$ and $c$ is smaller than $2/c$, as needed. □

Thus we have a $(\log n)$-synopsis data structure for the class of $F_0$ queries, designed for the dynamic scenario of both insertions and queries. Analyzed on the cost model of Section 2, both the query and update times are only $O(1)$ processing time per query/update and no I/O operations.

**3.2. Estimating $F_k$ for $k \geq 2$.** Alon *et al* [**AMS96**] developed an algorithm which, for every sequence $A$ and a parameter $k$, can estimate $F_k$ within a small constant factor with high probability, using an $(n^{1-1/k}\log n)$-synopsis data structure. The description below is taken from [**AGMS97**], which considered implementation issues of the algorithm and showed how the algorithm, coined sample-count, could be adapted to support deletions from the data set.

The idea in the sample-count algorithm is rather simple: A random sample of locations is selected in the sequence of data items that are inserted into the data set. This random selection can be easily done as the items are being inserted. Once we reach an item that was chosen to be in the sample, we will count from now on the number of incoming items that have its value. It turns out that the count $r$ for each sample point is a random variable which satisfies $\mathrm{E}(nkr^{k-1}) \approx F_k$, and that the variance is reasonably small, for small $k$. The desired accuracy and confidence of the final estimate are obtained by applying averaging techniques over the counts of sample items.

More specifically, the number of memory words used by the algorithm is $s = s_1 \cdot s_2$, where $s_1$ is a parameter that determines the accuracy of the result, and $s_2$ determines the confidence; e.g., for any input set, the relative error of the estimate $Y$ for $F_2$ exceeds $4u^{1/4}/\sqrt{s_1}$ with probability at most $2^{-s_2/2}$. The algorithm computes $s_2$ random variables $Y_1, Y_2, \ldots, Y_{s_2}$ and outputs their median $Y$. Each $Y_i$ is the average of $s_1$ random variables $X_{ij} : 1 \leq j \leq s_1$, where the $X_{ij}$ are independent, identically distributed random variables. Averaging is used to reduce the variance, and hence the error (Chebyshev's inequality), and the median is used to boost the confidence (Chernoff bounds). Each of the variables $X = X_{ij}$ is computed from the sequence in the same way as follows:

- Choose a random member $a_p$ of the sequence $A$, where the index $p$ is chosen randomly and uniformly among the numbers $1, 2, \ldots, n$; suppose that $a_p = l\ (\in U = \{1, 2, \ldots, u\})$.
- Let $r = |\{q : q \geq p, a_q = l\}|$ be the number of occurrences of $l$ among the members of the sequence $A$ following $a_p$ (inclusive).
- Define $X = n(r^k - (r-1)^k)$, e.g., for $k = 2$ let $X = n(2r-1)$.

For $k = 2$, it is shown in [**AMS96**] that the estimate $Y$ computed by the above algorithm satisfies $\mathrm{E}(Y) = F_2$, and $\mathrm{Prob}\left(|Y - F_2| \leq 4u^{1/4}/\sqrt{s_1}\right) \geq 1 - 2^{-s_2/2}$. An accurate estimate for $F_2$ can therefore be guaranteed with high probability by selecting $s_1 = \Theta(\sqrt{u})$ and $s_2 = \Theta(\log n)$. More generally, by selecting $s_1 = \frac{8ku^{1-1/k}}{\lambda^2}$ and $s_2 = 2\log(1/\epsilon)$, one can obtain the following.

THEOREM 3.3. [**AMS96**] *For every $k \geq 1$, every $\lambda > 0$ and every $\epsilon > 0$ there exists a randomized algorithm that computes, given a sequence $A = (a_1, \ldots, a_n)$ of members of $U = \{1, 2, \ldots, u\}$, in one pass and using*

$$O\left(\frac{k\log(1/\epsilon)}{\lambda^2}u^{1-1/k}(\log u + \log n)\right)$$

*memory bits, a number $Y$ so that the probability that $Y$ deviates from $F_k$ by more than $\lambda F_k$ is at most $\epsilon$.*

Thus for fixed $k$, $\lambda$, and $\epsilon$, we have an $(n^{1-1/k}\log n)$-synopsis data structure for the class of $F_k$ queries, designed for the dynamic scenario. Waiting until query time to compute the averages $Y_i$ would result in $O(s_1) = O(n^{1-1/k})$ query time on our cost model. However, these averages can be maintained as running averages as updates arrive, resulting in $O(1)$ processing time per query, and no I/O operations. Moreover, by representing the samples $a_p$ as a concise sample (defined in Section 4) and using a dynamic dictionary data structure, the update time can likewise be reduced to $O(1)$ processing time per update and no I/O operations.

**3.3. Improved estimation for $F_2$.** An improved estimation algorithm for $F_2$ was also presented in [**AMS96**]. For every sequence $A$, $F_2$ can be estimated within a small constant factor with high probability, using a $(\log n)$-synopsis data structure. Again, the description below is taken from [**AGMS97**], which considers implementation issues of the algorithm and shows how the algorithm, coined tug-of-war, can be adapted to support deletions from the data set.

The tug-of-war algorithm can be illustrated as follows: Suppose that a crowd consists of several groups of varying numbers of people, and that our goal is to estimate the skew in the distribution of people to groups. That is, we would like to estimate $F_2$ for the set $\{a_i\}_{i=1}^n$, where $a_i$ is the group to which the $i$'th person belongs. We arrange a tug-of-war, forming two teams by having each group assigned at random to one of the teams. Equating the displacement of the rope from its original location with the difference in the sizes of the two teams, it is shown in [**AMS96**] that the expected square of the rope displacement is exactly $F_2$, and that the variance is reasonably small. This approach can be implemented in small memory, using the observation that we can have the persons in the crowd come one by one, and contribute their displacement in an incremental fashion. In addition to the updated displacements, the only thing that requires recording in the process is the assignment of groups to teams, which can be done succinctly using an appropriate pseudo-random hash function.

As with sample-count, the number of memory words used by tug-of-war is $s = s_1 \cdot s_2$, where $s_1$ is a parameter that determines the accuracy of the result, and $s_2$ determines the confidence. As before, the output $Y$ is the median of $s_2$ random variables $Y_1, Y_2, \ldots, Y_{s_2}$, each being the average of $s_1$ random variables $X_{ij} : 1 \leq j \leq s_1$, where the $X_{ij}$ are independent, identically distributed random variables. Each $X = X_{ij}$ is computed from the sequence in the same way, as follows:

- Select at random a 4-wise independent mapping $i \mapsto \epsilon_i$, where $i \in U = \{1, 2, \ldots, u\}$ and $\epsilon_i \in \{-1, 1\}$.
- Let $Z = \sum_{i=1}^{u} \epsilon_i m_i$.

- Let $X = Z^2$.

For accurate estimates for $F_2$ of fixed error with guaranteed fixed probability, constant values suffice for $s_1$ and $s_2$. Specifically, by selecting $s_1 = \frac{16}{\lambda^2}$ and $s_2 = 2\log(1/\epsilon)$, the following is obtained.

THEOREM 3.4. [**AMS96**] *For every $\lambda > 0$ and $\epsilon > 0$ there exists a randomized algorithm that computes, given a sequence $A = (a_1, \ldots, a_n)$ of members of $U$, in one pass and using $O\left(\frac{\log(1/\epsilon)}{\lambda^2}(\log u + \log n)\right)$ memory bits, a number $Y$ so that the probability that $Y$ deviates from $F_2$ by more than $\lambda F_2$ is at most $\epsilon$. For fixed $\lambda$ and $\epsilon$, the algorithm can be implemented by performing, for each member of the sequence, a constant number of arithmetic and finite field operations on elements of $O(\log u + \log n)$ bits.*

Thus for fixed $\lambda$ and $\epsilon$, we have a $(\log n)$-synopsis data structure for the class of $F_2$ queries, designed for the dynamic scenario. Both the query and update times are only $O(1)$ processing time per query/update and no I/O operations.

**3.4. Lower bounds.** We mention lower bounds given in [**AMS96**] for the space complexity of randomized algorithms that approximate the frequency moments $F_k$. The lower bounds are obtained by reducing the problem to an appropriate communication complexity problem [**Yao83, BFS86, KS87, Raz92**], a set disjointness problem, obtaining the following.

THEOREM 3.5. [**AMS96**] *For any fixed $k > 5$ and $\gamma < 1/2$, any randomized algorithm that outputs, given one pass through an input sequence $A$ of at most $n$ elements of $U = \{1, 2, \ldots, n\}$, a number $Z_k$ such that* $\text{Prob}(|Z_k - F_k| > 0.1F_k) < \gamma$, *requires $\Omega(n^{1-5/k})$ memory bits.*

THEOREM 3.6. [**AMS96**] *Any randomized algorithm that outputs, given one pass through an input sequence $A$ of at most $2n$ elements of $U = \{1, 2, \ldots, n\}$, a number $Y$ such that* $\text{Prob}(|Y - F^*_\infty| \geq F^*_\infty/3) < \gamma$, *for some fixed $\gamma < 1/2$, requires $\Omega(n)$ memory bits.*

The first theorem above places a lower bound on the footprint of a synopsis data structure that can estimate $F_k$ to within constant factors in the purely online scenario, over all distributions. The second theorem shows that *no synopsis data structure exists* for estimating $F^*_\infty$ to within constant factors in the purely online scenario, over all distributions. As will be discussed in the next section, good synopsis data structures exist for skewed distributions, which may be of practical interest.

The number of elements $F_1$ can be computed deterministically and exactly using a $(\log n)$-synopsis data structure (a simple counter). The following two theorems show that for all $k \neq 1$, both randomization and approximation are essential in evaluating $F_k$.

THEOREM 3.7. [**AMS96**] *For any nonnegative integer $k \neq 1$, any randomized algorithm that outputs, given one pass through an input sequence $A$ of at most $2n$ elements of $U = \{1, 2, \ldots, n\}$ a number $Y$ such that $Y = F_k$ with probability at least $1 - \epsilon$, for some fixed $\epsilon < 1/2$, requires $\Omega(n)$ memory bits.*

THEOREM 3.8. [**AMS96**] *For any nonnegative integer $k \neq 1$, any deterministic algorithm that outputs, given one pass through an input sequence $A$ of $n/2$ elements* *of* $U = \{1, 2, \ldots, n\}$*, a number* $Y$ *such that* $|Y - F_k| \leq 0.1 F_k$ *requires* $\Omega(n)$ *memory bits.*

Proof. Let $G$ be a family of $t = 2^{\Omega(n)}$ subsets of $U$, each of cardinality $n/4$ so that any two distinct members of $G$ have at most $n/8$ elements in common. (The existence of such a $G$ follows from standard results in coding theory, and can be proved by a simple counting argument). Fix a deterministic algorithm that approximates $F_k$ for some fixed nonnegative $k \neq 1$. For every two members $G_1$ and $G_2$ of $G$ let $A(G_1, G_2)$ be the sequence of length $n/2$ starting with the $n/4$ members of $G_1$ (in a sorted order) and ending with the set of $n/4$ members of $G_2$ (in a sorted order). When the algorithm runs, given a sequence of the form $A(G_1, G_2)$, the memory configuration after it reads the first $n/4$ elements of the sequence depends only on $G_1$. By the pigeonhole principle, if the memory has less than $\log t$ bits, then there are two distinct sets $G_1$ and $G_2$ in $G$, so that the content of the memory after reading the elements of $G_1$ is equal to that content after reading the elements of $G_2$. This means that the algorithm must give the same final output to the two sequences $A(G_1, G_1)$ and $A(G_2, G_1)$. This, however, contradicts the assumption, since for every $k \neq 1$, the values of $F_k$ for the two sequences above differ from each other considerably; for $A(G_1, G_1)$, $F_0 = n/4$ and $F_k = 2^k n/4$ for $k \geq 2$, whereas for $A(G_2, G_1)$, $F_0 \geq 3n/8$ and $F_k \leq n/4 + 2^k n/8$. Therefore, the answer of the algorithm makes a relative error that exceeds 0.1 for at least one of these two sequences. It follows that the space used by the algorithm must be at least $\log t = \Omega(n)$, completing the proof. □

## 4. Hot list queries

In this section, we highlight our results on synopsis data structures for answering hot list and related queries.

A *hot list* is an ordered set of $m$ ⟨value, count⟩ pairs for the $m$ most frequently occurring "values" in a data set, for a prespecified $m$. In various contexts, hot lists are denoted as *high-biased histograms* [**IC93**] of $m + 1$ buckets, the first $m$ mode statistics, or the $m$ largest itemsets [**AS94**]. Hot lists are used in a variety of data analysis contexts, including:

- Best sellers lists ("top ten" lists): An example is the top selling items in a database of sales transactions.
- Selectivity estimation in query optimization: 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**]).
- Load balancing: 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.
- Market basket analysis: Given a sequence of sets of values, the goal is to determine the most popular *k-itemsets*, i.e., $k$-tuples of values that occur together in the most sets. Hot lists can be maintained on $k$-tuples of values for any specified $k$, and indicate a positive correlation among values in itemsets in the hot list. These can be used to produce association rules, specifying a (seemingly) causal relation among certain values [**AS94, BMUT97**]. An example is a grocery store, where for a sequence of customers, a set of the

items purchased by each customer is given, and an association rule might be that customers who buy bread typically also buy butter.
- "Caching" policies based on most-frequently used: The goal is to retain in the cache the most-frequently-used items and evict the least-frequently-used whenever the cache is full. An example is the most-frequently-called countries list in caller profiles for real-time telephone fraud detection [**Pre97**], and in fact an early version of the hot list algorithm described below has been in use in such contexts for several years.

As these examples suggest, the input need not be simply a sequence of individual values, but can be tuples with various fields such that for the purposes of the hot list, both the value associated with a tuple and the contribution by that tuple to that value's count are functions on its fields. However, for simplicity of exposition, we will discuss hot lists in terms of a sequence of values, each contributing one to its value's count.

Hot lists are trivial to compute and maintain given sufficient space to hold the full histogram of the data set. However, for many data sets, such histograms require space linear in the size of the data set. Thus for synopsis data structures for hot list queries, a more succinct representation is required, and in particular, counts cannot be maintained for each value. Note that the difficulty in maintaining hot lists in the dynamic scenario is in detecting when values that were infrequent become frequent due to shifts in the distribution of arriving data. With only a small footprint, such detection is difficult since there is insufficient space to keep track of all the infrequent values, and it is expensive (or impossible, in the purely online scenario) to access the base data once it is on the disks.

A related, and seemingly simpler problem to hot list queries is that of "popular items" queries. A *popular items* query returns a set of ⟨value, count⟩ pairs for all values whose frequency in the data set exceeds a prespecified threshold, such as 1% of the data set. Whereas hot list queries prespecify the number of pairs to be output but not a frequency lower bound, popular items queries prespecify a frequency lower bound but not the number of pairs. An approximate answer for a popular items query can be readily obtained by sampling, since the sample size needed to obtain a desired answer quality can be predetermined from the frequency threshold. For example, if $p < 1$ is the prespecified threshold percentage, then by Chernoff bounds, any value whose frequency exceeds this threshold will occur at least $c/2$ times in a sample of size $c/p$ with probability $1 - e^{-c/8}$. A recent paper by Fang *et al* [**FSGM$^+$98**] presented techniques for improving the accuracy and confidence for popular items queries. They considered the generalization to tuples and functions on its fields mentioned above for hot list queries, and denoted this class of queries as *iceberg queries*. They presented algorithms combining sampling with the use of multiple hash functions to perform coarse-grained counting, in order to significantly improve the answer quality over the naive sampling approach given above.

In the remainder of this section, we describe results in [**GM98**] presenting and studying two synopsis data structures, *concise samples* and *counting samples*. As mentioned in Section 3.4, there are no synopsis data structures for estimating the count of the most frequently occurring value, $F^*_\infty$, to within constant factors in the purely online scenario, over all distributions. Hence, no synopsis data structure exists for the more difficult problem of approximating the hot list in the purely online scenario, over all distributions. On the other hand, concise samples and counting samples are shown in [**GM98**] both analytically and experimentally to produce more accurate approximate hot lists than previous methods, and perform quite well for the skewed distributions that are of interest in practice.

**4.1. Concise samples.** Consider a hot list query on a data set of size $n$. One possible synopsis data structure is the set of values in a uniform random sample of the data set, as was proposed above for popular items queries. The $m$ most frequently occurring values in the sample are returned in response to the query, with their counts scaled by $n/m$. However, note that any value occurring frequently in the sample is a wasteful use of the available space. We can represent $k$ copies of the same value $v$ as the pair $\langle v, k\rangle$, and (assuming that values and counts use the same amount of space), we have freed up space for $k-2$ additional sample points. This simple observation leads to the following synopsis data structure.

DEFINITION 4.1. In a *concise representation* of a multiset, values appearing more than once in the multiset are represented as a value and a count. A *concise sample* of size $m$ is a uniform random sample of the data set whose concise representation has footprint $m$.

We can quantify the advantage of concise samples over traditional samples in terms of the number of additional sample points for the same footprint. Let $S = \{\langle v_1, c_1\rangle, \ldots, \langle v_j, c_j\rangle, v_{j+1}, \ldots, v_\ell\}$ be a concise sample of a data set of $n$ values. We define *sample-size*$(S)$ to be $\ell - j + \sum_{i=1}^{j} c_i$. Note that the footprint of $S$ depends on the number of bits used per value and per count. For example, variable-length encoding could be used for the counts, so that only $\lceil \log x\rceil$ bits are needed to store $x$ as a count; this reduces the footprint but complicates the memory management. Approximate counts [**Mor78**] could be used as well, so that only $\lceil \log\log x\rceil$ bits are needed to store $x$ to within a power of two. For simplicity of exposition, we will consider only fixed-length encoding of $\log n$ bits per count and per value, including any bits needed to distinguish values from counts, so that the footprint of $S$ is $(\ell + j)\log n$. For a traditional sample with $m$ sample points, the sample-size is $m$ and the footprint is $m \log n$.

Concise samples are never worse than traditional samples (given the encoding assumptions above), and can be exponentially or more better depending on the data distribution. For example, if there are at most $m/(2\log n)$ distinct values in the data set, then a concise sample of size $m$ would have sample-size $n$ (i.e., in this case, the concise sample is the full histogram). Thus, the sample-size of a concise sample may be arbitrarily larger than its footprint:

LEMMA 4.2. [**GM98**] *For any footprint $m \geq 2\log n$, 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.*

For exponential distributions, the advantage is exponential:

LEMMA 4.3. [**GM98**] *Consider the family of exponential distributions: for $i = 1, 2, \cdots$, $\Pr(v = i) = \alpha^{-i}(\alpha - 1)$, for $\alpha > 1$. For any $m \geq 2$, the expected sample-size of a concise sample with footprint $m \log n$ is at least $\alpha^{m/2}$.*

PROOF. Let $x = m/2$. Note that we can fit at least $x$ values and their counts within the given footprint. The expected sample-size can be lower bounded by the expected number of randomly selected tuples before the first tuple whose attribute value $v$ is greater than $x$. The probability of selecting a value greater than $x$ is $\sum_{i=x+1}^{\infty} \alpha^{-i}(\alpha-1) = \alpha^{-x}$, so the expected number of tuples selected before such an event occurs is $\alpha^x$. $\square$

The expected gain in using a concise sample over a traditional sample for arbitrary data sets is a function of the frequency moments $F_k$, for $k \geq 2$, of the data set. Recall from Section 3 that $F_k = \sum_j m_j^k$, where $j$ is taken over the values represented in the set and $m_j$ is the number of set elements of value $j$.

THEOREM 4.4. [**GM98**] *For any data set, when using a concise sample $S$ with sample-size $m$, the expected gain is*

$$E[m - \textit{number of distinct values in } S] = \sum_{k=2}^{m} (-1)^k \binom{m}{k} \frac{F_k}{n^k} \ .$$

PROOF. Let $p_j = m_j/n$ be the probability that an item selected at random from the set is of value $j$. Let $X_i$ be an indicator random variable so that $X_i = 1$ if the $i$th item selected to be in the traditional sample has a value not represented as yet in the sample, and $X_i = 0$ otherwise. Then, $\Pr(X_i = 1) = \sum_j p_j(1-p_j)^{i-1}$, where $j$ is taken over the values represented in the set (since $X_i = 1$ if some value $j$ is selected so that it has not been selected in any of the first $i-1$ steps). Clearly, $X = \sum_{i=1}^{m} X_i$ is the number of distinct values in the traditional sample. We can now evaluate $E$[number of distinct values] as

$$\begin{aligned}
E[X] &= \sum_{i=1}^{m} E[X_i] = \sum_{i=1}^{m} \sum_j p_j(1-p_j)^{i-1} = \sum_j \sum_{i=1}^{m} p_j(1-p_j)^{i-1} \\
&= \sum_j p_j \frac{1-(1-p_j)^m}{1-(1-p_j)} = \sum_j \left(1-(1-p_j)^m\right) \\
&= \sum_j \left(1 - \sum_{k=0}^{m} (-1)^k \binom{m}{k} p_j^k\right) = \sum_j 1 - \sum_{k=0}^{m} (-1)^k \binom{m}{k} \sum_j p_j^k \\
&= \sum_{k=1}^{m} (-1)^{k+1} \binom{m}{k} \frac{F_k}{n^k} \ .
\end{aligned}$$

$\square$

Note that the footprint for a concise sample is at most $2\log n$ times the number of distinct values, whereas the footprint for a traditional sample of sample-size $m$ is $m \log n$.

*Maintaining concise samples.* We describe next the algorithm given in [**GM98**] for maintaining a concise sample within a given footprint bound as new data is inserted into the data set. Since the number of sample points provided by a concise sample depends on the data distribution, the problem of maintaining a concise sample as new data arrives is more difficult than with traditional samples. For traditional samples, the reservoir sampling algorithm of Vitter [**Vit85**] can be used to maintain a sample in the presence of insertions of new data (see Section 5.1 for details). However, this algorithm relies heavily on a priori knowledge of the target sample-size (which, for traditional samples, equals the footprint divided by $\log n$). 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. Let $\tau$ be an entry threshold (initially 1) for new data to be selected for the sample. Let $S$ be the current concise sample and consider an insertion of a data item with value $v$. With probability $1/\tau$, add $v$ to $S$, preserving the concise representation. If the footprint for $S$ now exceeds the prespecified footprint bound, raise the threshold to some $\tau'$ and then subject each sample point in $S$ to this higher threshold. Specifically, each of the sample-size($S$) sample points is evicted with probability $\tau/\tau'$. It is expected that sample-size($S$) $\cdot (1 - \tau/\tau')$ sample points will be evicted. Note that the footprint is only decreased when a ⟨value, count⟩ pair reverts to a singleton or when a value is removed altogether. If the footprint has not decreased, repeat with a higher threshold.

There is complete flexibility in this algorithm in selecting the sequence of increasing thresholds, and [**GM98**] discussed a variety of approaches and their trade-offs, as well as ways to improve the constant factors.

THEOREM 4.5. [**GM98**] *The above algorithm maintains a concise sample within a prespecified size bound in constant amortized expected update time per insert, and no I/O operations.*

PROOF. The algorithm maintains a uniform random sample since, whenever the threshold is raised, it preserves the invariant that each item in the data set has been treated (probabilistically) as if the threshold were always the new threshold. The look-ups can be done in constant expected time using a dynamic dictionary data structure such as a hash table. Raising a threshold costs $O(m')$ processing time, where $m'$ is the sample-size of the concise sample before the threshold was raised. For the case where the threshold is raised by a constant factor each time, we expect there to be a constant number of coin tosses resulting in sample points being retained for each sample point evicted. Thus we can amortize the retained against the evicted, and we can amortize the evicted against their insertion into the sample (each sample point is evicted only once). □

**4.2. 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 set after the value was selected for the sample. Their definition is motivated by a sampling&counting process of this type from a static data set:

DEFINITION 4.6. A *counting sample* for a data set $A$ with threshold $\tau$ is any subset of $A$ stored in a concise representation (as defined in Definition 4.1) that is obtained by a process that is probabilistically equivalent to the following process: For each value $v$ occurring $c > 0$ times in $A$, 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 $i$th coin toss is heads, then $v$ occurs $c - i + 1$ times in the subset, else $v$ is not in the subset.

A counting sample differs from the approach used in Section 3.2 in not allowing multiple counts for the same value and in its use of a threshold (that will adapt to a data distribution) versus a prespecified sample size. Although counting samples are not uniform random samples of the data set, 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.

*Maintaining counting samples.* The following algorithm is given in [**GM98**] for maintaining a counting sample within a given footprint bound for the dynamic scenario. Let $\tau$ be an entry threshold (initially 1) for new data to be selected for the sample. Let $S$ be the current counting sample and consider an insertion of a data item with value $v$. If $v$ is represented by a $\langle$value, count$\rangle$ pair in $S$, increment its count. If $v$ is a singleton in $S$, create a pair with count set to 2. Otherwise, add $v$ to $S$ with probability $1/\tau$. If the footprint for $S$ now exceeds the prespecified footprint bound, raise the threshold to some $\tau'$ and then subject each value in $S$ to this higher threshold. Specifically, for each value in the counting sample, flip a biased coin, decrementing its observed 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 $\tau/\tau'$, 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.

An advantage of counting samples over concise samples is that one can maintain counting samples in the presence of deletions to the data set. 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.[2] With counting samples, we do not have this difficulty. For a delete of a value $v$, it suffices to reverse the increment procedure by decrementing a count, converting a pair to a singleton, or removing a singleton, as appropriate.

As with concise samples, there is complete flexibility in this algorithm in selecting the sequence of increasing thresholds, and [**GM98**] discussed a variety of approaches and their tradeoffs, as well as ways to improve the constant factors.

THEOREM 4.7. [**GM98**] *For any sequence of insertions and deletions in the dynamic scenario, the above algorithm maintains a counting sample within a prespecified footprint in constant amortized expected update time and no I/O operations.*

PROOF. We must show that the requirement in the definition of a counting sample is preserved when an insert occurs, a delete occurs, or the threshold is raised. Let $A$ be the data set and $S$ be the counting sample.

An insert of a value $v$ increases by one its count in $A$. If $v$ is in $S$, then one of its coin flips to date was heads, and we increment the count in $S$. Otherwise, none of its coin flips to date were heads, and the algorithm flips a coin with the appropriate probability. All other values are untouched, so the requirement is preserved.

A delete of a value $v$ decreases by one its count in $A$. If $v$ is in $S$, then the algorithm decrements the count (which may drop the count to 0). Otherwise, $c$ coin flips occurred to date and were tails, so the first $c-1$ were also tails, and the value remains omitted from $S$. All other values are untouched, so the requirement is preserved.

Consider raising the threshold from $\tau$ to $\tau'$, and let $v$ be a value occurring $c > 0$ times in $A$. If $v$ is not in $S$, there were the equivalent of $c$ coin flips with heads probability $1/\tau$ that came up tails. Thus the same $c$ probabilistic events would fail to come up heads with the new, stricter coin (with heads probability only $1/\tau'$). If

[2] For some applications of random samples, an effective alternative approach is to collect and make use of two uniform samples: one for the inserted data and one for the deleted data.

$v$ is in $S$ with count $c'$, then there were the equivalent of $c - c'$ coin flips with heads probability $1/\tau$ that came up tails, and these same probabilistic events would come up tails with the stricter coin. This was followed by the equivalent of a coin flip with heads probability $1/\tau$ that came up heads, and the algorithm flips a coin with heads probability $\tau/\tau'$, so that the result is equivalent to a coin flip with probability $(1/\tau) \cdot (\tau/\tau') = (1/\tau')$. If this coin comes up tails, then subsequent coin flips for this value have heads probability $1/\tau'$. In this way, the requirement is preserved for all values.

The update time bounds are argued as in the proof of Theorem 4.5. □

Note that although both concise samples and counting samples have $O(1)$ amortized update times, counting samples are slower to update than concise samples, since, unlike concise sample, they perform a look-up (into the counting sample) at each update to the data set. On the other hand, with counting samples, the guarantees on the counts are stronger, since exact counting is used on values already in the sample.

**4.3. Application to hot list queries.** Consider a hot list query requesting $k$ pairs. Given a concise sample $S$ of footprint $m \log n$, $m \geq 2k$, an approximate hot list can be reported by computing the $k$'th largest count $c_k$ (using a linear time selection algorithm), and then reporting all pairs with counts at least $\max(c_k, \delta)$, scaling the counts by $n/m'$, where $\delta \geq 1$ is a confidence threshold and $m'$ = sample-size($S$). Note that when $\delta = 1$, $k$ pairs will be reported, but with larger $\delta$, fewer than $k$ may be reported. The response time for reporting is $O(m)$ processing time and no I/O operations. Alternatively, we can trade-off update time versus query time by keeping the concise sample sorted by counts. This allows for reporting in $\Theta(k)$ time.

Given a counting sample $S$ of footprint $m \log n$ with threshold $\tau$, an approximate hot list can be reported by computing the $k$'th largest count $c_k$, and then reporting all pairs with counts at least $\max(c_k, \tau - \hat{c})$, where $\hat{c}$ is a compensation added to each reported count that serves to compensate for inserts of a value into the data set prior to the successful coin toss that placed it in the counting sample. An analysis in [**GM98**] argued for $\hat{c} = \tau\left(\frac{e-2}{e-1}\right) - 1 \approx .418 \cdot \tau - 1$. Given the conversion of counting samples into concise samples discussed in Section 4.2, this can be seen to be similar to taking $\delta = 2 - \frac{\hat{c}+1}{\tau} \approx 1.582$.

Analytical bounds and experimental results are presented in [**GM98**] quantifying the accuracy of the approximate hot lists reported using concise samples or counting samples. An example plot from that paper is given in Figure 3, where the data is drawn from a Zipf distribution with parameter 1.5 and the footprint is measured in memory words.

## 5. Histograms and quantiles

Histograms approximate a data set by grouping values into "buckets" (subsets) and approximating the distribution of values in the data set based on summary statistics maintained in each bucket (see, e.g., [**PIHS96**]). Histograms are commonly used in practice in various databases (e.g., in DB2, Informix, Ingres, Oracle, Microsoft SQL Server, Sybase, and Teradata). They are used for selectivity estimation purposes within a query optimizer and in query execution, and there is work in progress on using them for approximate query answering.

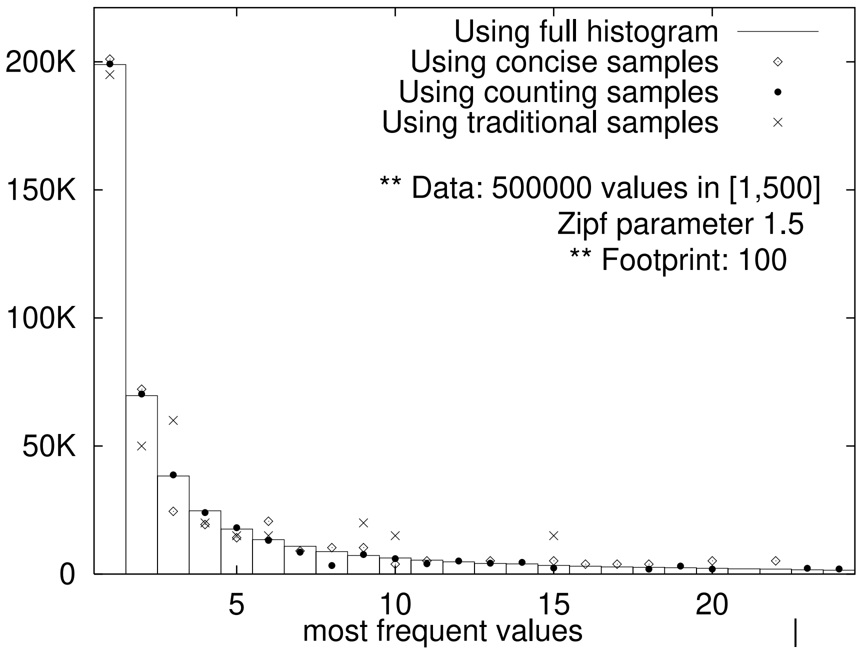


FIGURE 3. Comparison of algorithms for a hot list query, depicting the frequency of the most frequent values as reported using a full histogram, using a concise sample, using a counting sample, and using a traditional sample

Two histogram classes used extensively in database systems are equi-depth histograms and compressed histograms. In an *equi-depth* or *equi-height* histogram, contiguous ranges of values are grouped into buckets such that the number of data items falling into each bucket is the same. The endpoints of the value ranges are denoted the bucket boundaries or *quantiles*. In a *compressed* histogram [**PIHS96**], the highest frequency values are stored separately in single-valued buckets; the rest are partitioned as in an equi-depth histogram. Compressed histograms typically provide more accurate estimates than equi-depth histograms.

A common problem with histograms is their dynamic maintenance. As a data set is updated, its distribution of values might change and the histogram (which is supposed to reflect the distribution) should change as well, since otherwise estimates based on the histogram will be increasingly inaccurate. In this section, we describe our work in [**GMP97b**] on algorithms for maintaining approximate equi-depth and compressed histograms as synopsis data structures in the dynamic scenario. We also discuss recent related work by Manku *et al* [**MRL98**] on computing approximate quantiles.

Another concern for histograms is their construction costs in the static scenario. Sampling can be used to improve the construction times (see, e.g., [**PIHS96**]), and we discuss recent work by Chaudhuri *et al* [**CMN98**] on using sampling to construct approximate equi-depth histograms in the static scenario.

An important feature of our algorithms for maintaining approximate histograms is the use of a "backing sample". Backing samples are interesting for two reasons:

they can be used to convert sampling-based algorithms for the static scenario into algorithms for the dynamic scenario, and their use is an example of a hierarchical approach to synopsis data structures.

**5.1. Backing samples.** A *backing sample* for a data set, $A$, is a uniform random sample of $A$ that is kept up-to-date in the presence of updates to $A$ [**GMP97b**]. In most sampling-based estimation techniques, whenever a sample of size $m$ is needed, either the entire relation is scanned to extract the sample, or several random disk blocks are read. In the latter case, the values in a disk block may be highly correlated, and hence to obtain a truly random sample, $m$ disk blocks may need to be read, with only a single value used from each block. In contrast, a backing sample is a synopsis data structure that may reside in main memory, and hence be accessed with no I/O operations. Moreover, if, as is typically the case in databases, each data item is a record ("tuple") comprised of fields ("attributes"), then only the fields desired for the sampling need be retained in the synopsis. In the case of using samples for histograms, for example, only the field(s) needed for the histogram need be retained. If the backing sample is stored on the disks, it can be packed densely into disk blocks, allowing it to be more quickly swapped in and out of memory. Finally, an indexing structure for the sample can be maintained, which would enable fast access of the sample values within a certain range.

Clearly, a backing sample of $m$ sample points can be used to convert a sampling-based algorithm requiring $\frac{m}{D}$ I/O operations for its sampling into an algorithm that potentially requires no I/O operations.

*Maintaining backing samples.* A uniform random sample of a target size $m$ can be maintained under insertions to the data set using Vitter's *reservoir sampling* technique [**Vit85**]: The algorithm proceeds by inserting the first $m$ items into a "reservoir." Then a random number of new items are skipped, and the next item replaces a randomly selected item in the reservoir. Another random number of items are then skipped, and so forth. The distribution function of the length of each random skip depends explicitly on the number of items so far, and is chosen such that at any point each item in the data set is equally likely to be in the reservoir. Specifically, when the size of the data set is $n$, the probability for an item to be selected for the backing sample of size $m$ is $m/n$. Random skipping is employed in order to reduce constant factors in the update times compared with the approach of flipping a coin for each new item. Reservoir sampling maintains a traditional random sample as a backing sample; an alternative is to use a concise sample or a counting sample as a backing sample, and maintain them as discussed in Section 4.

As discussed in Section 4.2, there are difficulties in maintaining uniform random samples under deletions to the data set, with two possible solutions being counting samples and deletion samples. In [**GMP97b**], we assumed that each data item has a unique id (namely, its row id in the database table in which it resides), so that a deletion removes a unique item from the data set. We retained the row id with the sample point (which precludes the use of concise samples or counting samples). With row ids, deletions can be handled by removing the item from the sample, if it is in the sample. However, such deletions decrease the size of the sample from the target size $m$, and moreover, it is not apparent how to use subsequent insertions to obtain a provably random sample of size $m$ once the sample has dropped below $m$. Instead, we maintained a sample whose size is initially a prespecified upper bound

$U$, and allowed for it to decrease as a result of deletions of sample items down to a prespecified lower bound $L$. If the sample size dropped below $L$, the data set is read from the disks in order to re-populate the random sample, either by rereading all the data or by reading $U - L + 1$ random disk blocks. Since the sampling is independent of the deletions, the deletion of a fraction of the sample is expected to occur only after the deletion of the same fraction of the data set.

We presented in [**GMP97b**] several techniques for reducing constant factors in the update times. For example, since the algorithm maintains a random sample independent of the order of the updates to the data set, we postponed the processing of deletes until the next insert selected for the backing sample. This reduced the maintenance to the insert-only case, for which random skipping can be employed (having deletions intermixed with insertions foils random skipping).

Note that since a backing sample is a fixed sample of a prespecified size, it may be desirable to augment the sample and/or refresh the sample, as appropriate for a particular application.

*Backing samples in a synopsis hierarchy.* In [**GMP97b**], we used a backing sample in support of dynamically maintaining histograms. In the scenario we considered, the histogram resided in main memory whereas the backing sample, being somewhat larger than the histogram, resided on the disks. The goal was to maintain the histogram under the dynamic scenario, while minimizing the accesses and updates to the backing sample, in order to minimize the number of I/O operations. The backing sample was a traditional random sample maintained using reservoir sampling. When the size of the data set is $n$, the probability for an item to be selected for the backing sample of size $m$ is $m/n$, and hence in maintaining the backing sample an I/O operation is expected only once every $\Omega(n/m)$ insertions. Therefore, over the process of maintaining the backing sample, while the data set grows from $m$ to $n$, an I/O operation is expected (on the average) only once every $\Omega(n/(m\log(n/m)))$ insertions. Thus, since this overhead is small for large $n$ and small $m$, the goal became to design an algorithm for maintaining histograms that minimized the number of accesses to a given backing sample.

**5.2. Equi-depth histograms.** An equi-depth histogram partitions the range of possible values into $\beta$ buckets such that the number of data items whose value falls into a given bucket is the same for all buckets. An *approximate* equi-depth histogram approximates the exact histogram by relaxing the requirement on the number of data items falling in a bucket and/or the accuracy of the counts associated with the buckets. Let $N$ be the number of items in the data set, let $B$.count be the count associated with a bucket $B$, and let $f_B$ be the number of items falling in a bucket $B$.[3] In [**GMP97b**], we defined two error metrics for evaluating approximate equi-depth histograms. Our first metric, $\mu_{\mathrm{ed}}$, was defined to be the standard deviation of the bucket sizes from the mean bucket size, normalized with respect to the mean bucket size:

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

[3]For simplicity in this paper, we ignore issues of how to attribute items to buckets for items whose data value is equal to one or more bucket boundaries; such issues are addressed in [**GMP97b**].

Our second error metric, $\mu_{\text{count}}$, was defined to be the standard deviation of the bucket counts from the actual number of items in each bucket, normalized with respect to the mean bucket count:

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

In [**GMP97b**], we presented the first low overhead algorithms for maintaining highly-accurate approximate equi-depth histograms. Each algorithm relied on using a backing sample, $S$, of a fixed size dependent on $\beta$.

Our simplest algorithm, denoted Equi-depth_Simple, worked as follows. At the start of each phase, compute an approximate equi-depth histogram from $S$ by sorting $S$ and then taking every $(|S|/\beta)$'th item as a bucket boundary. Set the bucket counts to be $N'/\beta$, where $N'$ is the number of items in the data set at the beginning of the phase. Let $T = \lceil (2+\gamma)N'/\beta \rceil$, where $\gamma > -1$ is a tunable performance parameter. Larger values for $\gamma$ allow for greater imbalance among the buckets in order to have fewer phases. As each new item is inserted into the data set, increment the count of the appropriate bucket. When a count exceeds the threshold $T$, start a new phase.

THEOREM 5.1. [**GMP97b**] *Let $\beta \geq 3$. Let $m = (c\ln^2\beta)\beta$, for some $c \geq 4$. Consider Equi-depth_Simple applied to a sequence of $N \geq m^3$ inserts of items into an initially empty data set. 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 Equi-depth_Simple computes an approximate equi-depth histogram such that with probability at least $1 - \beta^{-(\sqrt{c}-1)} - (N/(2+\gamma))^{-1/3}$, $\mu_{\text{ed}} \leq \alpha + (1+\gamma)$ and $\mu_{\text{count}} \leq \alpha$.*

PROOF. Let $H$ be an approximate equi-depth histogram computed by the Equi-depth_Simple algorithm after $N$ items have been inserted into the data set. Let $\Phi$ be the current phase of the algorithm, and let $N' \leq N$ be the number of items in the data set at the beginning of phase $\Phi$. Let $\mu'_{\text{count}}$ and $\mu'_{\text{ed}}$ be the errors $\mu_{\text{count}}$ and $\mu_{\text{ed}}$, respectively, resulting after extracting an approximate histogram from $S$ at the beginning of phase $\Phi$. Finally, 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 phase $\Phi$, we have that $N \leq N'(2+\gamma)$, it follows that $\rho \leq \rho'$. We show in [**GMP97b**] that $\mu'_{\text{ed}} = \mu'_{\text{count}} \leq \alpha$ with probability at least $\rho'$, and hence at least $\rho$.

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

Let $f'_{B_i}$ and $B_i.\text{count}'$ be the values of $f_{B_i}$ and $B_i$.count, respectively, at the beginning of phase $\Phi$. 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,

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

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

$$\begin{aligned}\mu_{\rm ed} &= \frac{\beta}{N}\sqrt{\frac{1}{\beta}\left(\sum_i^{\beta} {\Delta_i'}^2 + \sum_{i=1}^{\beta}((1+\gamma)N'/\beta)^2\right)} \\ &\leq \mu_{\rm ed}' + \frac{\beta}{N}(1+\gamma)N'/\beta) \leq \mu_{\rm ed}' + (1+\gamma)\ .\end{aligned}$$

The theorem follows. □

A second algorithm from [**GMP97b**] reduced the number of recomputations from $S$ by trying to balance the buckets using a local, less expensive procedure. The algorithm, denoted Equi-depth_SplitMerge, worked in phases. As in Equi-depth_Simple, at each phase there is a threshold $T = \lceil (2+\gamma)N'/\beta \rceil$. As each new item is inserted into the data set, increment the count of the appropriate bucket. When a count exceeds the threshold $T$, split the bucket in half. In order to maintain the number of buckets $\beta$ fixed, 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, recompute the approximate equi-depth histogram from $S$.

To merge two buckets, sum the counts of the two buckets and dispose of the boundary between them. To split a bucket $B$, select an approximate median in $B$ to serve as the bucket boundary between the two new buckets, by selecting the median among the items in $S$ that fall into $B$. The split and merge operation is illustrated in Figure 4. Note that split and merge can occur only for $\gamma > 0$.

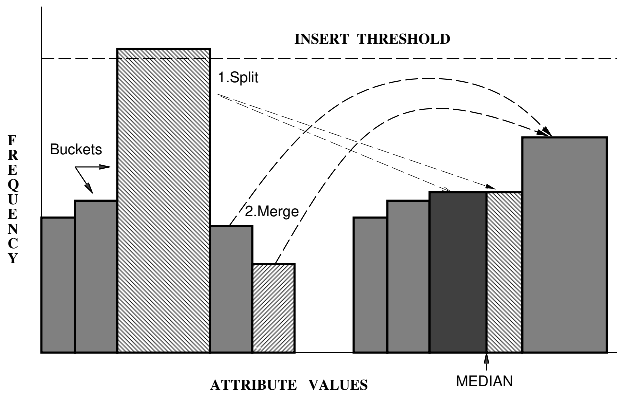


FIGURE 4. Split and merge operation during equi-depth histogram maintenance

The number of splits and the number of phases can be bounded as follows.

THEOREM 5.2. [**GMP97b**] *Consider Equi-depth_SplitMerge with $\beta$ 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 $\alpha = 1 + (1+\gamma)/\beta$.*

To handle deletions to the data set, let $T_\ell = \lfloor N'/(\beta(2+\gamma_\ell)) \rfloor$ be a lower threshold on the bucket counts, where $\gamma_\ell > -1$ is a tunable performance parameter. When an item is deleted from the data set, decrement the count of the appropriate bucket. If a bucket's count drops to the threshold $T_\ell$, 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, recompute the approximate equi-depth histogram from $S$. The merge and split operation is illustrated in Figure 5.

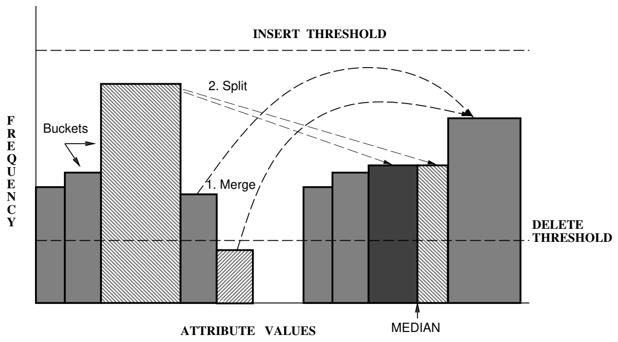


FIGURE 5. Merge and split operation during equi-depth histogram maintenance

*Related work.* A recent paper by Manku *et al* [**MRL98**] presented new algorithms for computing approximate quantiles of large data sets in a single pass over the data and with limited main memory. Whereas in an equi-depth histogram, the desired ranks for the quantiles are at regular intervals, their paper considered arbitrary prespecified ranks. Compared to an earlier algorithm of Munro and Paterson [**MP80**], their deterministic algorithm restricts attention to a single pass and improves the constants in the memory requirements. Specifically, let an item be an $\epsilon$-approximate $\phi$-quantile in a data set of $N$ items if its rank in the sorted data set is between $\lceil(\phi - \epsilon)N\rceil$ and $\lceil(\phi + \epsilon)N\rceil$. Manku *et al* presented a deterministic algorithm that, given $\phi_1, \ldots, \phi_k \in [0, 1]$, computes $\epsilon$-approximate $\phi_i$-quantiles for $i = 1, \ldots, k$ in a single pass using only $O(\frac{1}{\epsilon}\log^2(\epsilon N))$ memory. Note that this algorithm performs $\frac{N}{DB}$ I/O operations in the static scenario, for a class of queries where the ranks of the desired quantiles are prespecified.

Manku *et al* also analyzed the approach of first taking a random sample and then running their deterministic algorithm on the sample, in order to reduce the memory requirements for massive data sets. They did not explicitly consider dynamic maintenance of quantiles, and indeed they have not attempted to minimize the query time to output their approximate quantiles, since their output operation occurs only once, after the pass over the data. However, by using a backing sample residing in memory, their algorithm can be used in the dynamic scenario with no I/O operations at update time or query time.

A recent paper by Chaudhuri *et al* [**CMN98**] studied the problem of how much sampling is needed to guarantee an approximate equi-depth histogram of a certain accuracy. The error metric they used to evaluate accuracy is the maximum over all buckets $B$ of $|f_B - \frac{N}{\beta}|$, where $N$ is the number of data items, $\beta$ is the number of buckets, and $f_B$ is the number of items falling into $B$. As argued in the paper, this error metric seems more appropriate than the $\mu_{\text{ed}}$ metric considered above, for providing guarantees on the accuracy of approximate answers to range queries. (See also [**JKM$^+$98**] for another approach to providing improved quality guarantees when using histograms to answer range queries.) Chaudhuri *et al* provided a tighter analysis than in [**GMP97b**] for analyzing the accuracy of equi-depth histograms computed from a sample. The paper studied only the static scenario of constructing an equi-depth histogram, including a discussion of techniques for extracting multiple sample points from a sampled disk block. However, by using a backing sample, such issues are no longer a concern, and their analysis can be used to improve the guarantees of the algorithms in [**GMP97b**] for maintaining equi-depth histograms in the dynamic scenario.

**5.3. 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 [**PIHS96**]. Our target compressed histogram with $\beta$ buckets has $\beta'$ equi-depth buckets and $\beta - \beta'$ singleton buckets, where $1 \leq \beta' \leq \beta$, such that the following requirements hold: (i) each equi-depth bucket has $N'/\beta'$ tuples, where $N'$ is the total number of data items 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 [**GMP97b**], we presented the first low overhead algorithm for maintaining highly-accurate approximate compressed histograms. As in the equi-depth case, the algorithm relied on using a backing sample $S$. An approximate compressed histogram can be computed from $S$ as follows. Let $m$, initially $|S|$, be the number of items tentatively in equi-depth buckets. Consider the $\beta - 1$ most frequent values occurring in $S$, 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, create a singleton bucket for the value with count $fN/|S|$, and decrease $m$ by $f$. Otherwise, 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 $(N/|S|) \cdot (m/\beta')$. Our algorithm reduced the number of such recomputations from $S$ by employing a local procedure for adjusting bucket boundaries.

Similar to the equi-depth algorithm, the algorithm worked 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 data set 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 items 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, $\beta'$, grows and shrinks, and hence we must adjust the equi-depth buckets accordingly.

4. Likewise, the number of items in equi-depth buckets grows and shrinks dramatically as sets of items are removed from and added to singleton buckets. The ideal is to maintain $N'/\beta'$ items per equi-depth bucket, but both $N'$ and $\beta'$ are growing and shrinking.

Briefly and informally, the algorithm in [**GMP97b**] addressed each of these four concerns as follows. To address concern 1, it used 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 a new singleton bucket for $v$ must be created. To address concern 2, the algorithm allowed singleton buckets with relatively small counts to be merged back into the equi-depth buckets. As for concerns 3 and 4, it used our procedures for splitting and merging buckets to grow and shrink the number of buckets, while maintaining approximate equi-depth buckets, until the histogram is recomputed from $S$. The imbalance between the equi-depth buckets is controlled by the thresholds $T$ and $T_\ell$ (which depend on the tunable performance parameters $\gamma$ and $\gamma_\ell$, as in the equi-depth algorithm). When an equi-depth bucket is converted into a singleton bucket or vice-versa, the algorithm ensured that at the time, the bucket is within a constant factor of the average number of items 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 selected the one with the largest (smallest combined, respectively) bucket count.

In [**GMP97b**], we presented analytical and experimental studies of the algorithms discussed above for maintaining equi-depth histograms and for maintaining compressed histograms in the dynamic scenario.

## 6. Related work and further results

A concept related to synopsis data structures is that of condensed representations, presented by Mannila and Toivonen [**MT96, Man97**]: Given a class of structures $D$, a data collection $d \in D$, and a class of patterns $P$, a *condensed representation* for $d$ and $P$ is a data structure that makes it possible to answer queries of the form "How many times does $p \in P$ occur in $d$" approximately correctly and more efficiently than by looking at $d$ itself. Related structures include the data cube [**GCB$^+$97**], pruned or cached data structures considered in machine learning [**Cat92, ML97**], and $\epsilon$-nets widely used in computational geometry [**Mul94**]. Mannila and Toivonen also proposed an approximation metric for their structures, denoted an $\epsilon$-adequate representation.

Approximate data structures that provide fast approximate answers were proposed and studied by Matias *et al* [**MVN93, MVY94, MSY96**]. 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. The data structures considered have linear space footprints, so are not synopsis data structures. However, they can be adapted to provide a synopsis approximate priority queue, where the footprint is determined by the approximation error.

There have been several papers discussing systems and techniques for providing approximate query answers without the benefit of precomputed/maintained synopsis data structures. 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 at query time and the approximate answer for an aggregation query is updated as the scan proceeds. Bayardo and Miranker [**BM96**] devised techniques for "fast-first" query processing, whose goal is to quickly provide a few tuples of the query answer from the base data. The Oracle Rdb system [**AZ96**] also provides support for fast-first query processing, by running multiple query plans simultaneously. Vrbsky and Liu [**VL93**] (see also the references therein) described a query processor that provides approximate answers to queries in the form of subsets and supersets that converge to the exact answer. The query processor uses various class hierarchies to iteratively fetch blocks of the base data that are relevant to the answer, producing tuples certain to be in the answer while narrowing the possible classes that contain the answer. Since these approaches read from the base data at query time, they incur multiple I/O operations at query time.

A recent survey by Barbará *et al.* [**BDF$^+$97**] describes the state of the art in *data reduction* techniques, for reducing massive data sets down to a "big picture" and for providing quick approximate answers to queries. The data reduction techniques surveyed by the paper are singular value decomposition, wavelets, regression, log-linear models, histograms, clustering techniques, index trees, and sampling. Each technique is described briefly (see the references therein for further details on these techniques and related work) and then evaluated *qualitatively* on to its effectiveness and suitability for various data types and distributions, on how well it can be maintained under insertions and deletions to the data set, and on whether it supports answers that progressively improve the approximation with time.

The list of data structures work that could be considered synopsis data structures is extensive. We have described a few of these works in the paper; here we mention several others. Krishnan *et al* [**KVI96**] proposed and studied the use of a compact suffix tree-based structure for estimating the selectivity of an alphanumeric predicate with wildcards. Manber [**Man94**] considered the use of concise "signatures" to find similarities among files. Broder *et al* [**BCFM98**] studied the use of (approximate) min-wise independent families of permutations for signatures in a related context, namely, detecting and filtering near-duplicate documents. Our work on synopsis data structures also includes the use of multi-fractals and wavelets for synopsis data structures [**FMS96, MVW98**] and *join samples* for queries on the join of multiple sets [**GPA$^+$98**].

## 7. Conclusions

This paper considers synopsis data structures as an algorithmic framework relevant to massive data sets. For such data sets, the available memory is often substantially smaller than the size of the data. Since synopsis data structures are too small to maintain a full characterization of the base data sets, the responses they provide to queries will typically be approximate ones. The challenges are to determine (1) what synopsis of the data to keep in the limited space in order to maximize the accuracy and confidence of its approximate responses, and (2) how to efficiently compute the synopsis and maintain it in the presence of updates to the data set.

The context of synopsis data structures presents many algorithmic challenges. Problems that may have easy and efficient solutions using linear space data structures may be rather difficult to address when using limited-memory, synopsis data structures. We discussed three such problems: frequency moments, hot list queries, and histograms. Different classes of queries may require different synopsis data structures. While several classes of queries have been recently considered, there is a need to consider many more classes of queries in the context of synopsis data structures, and to analyze their effectiveness in providing accurate or approximate answers to queries. We hope that this paper will motivate others in the algorithms community to study these problems. Due to the increasing prevalence of massive data sets, improvements in this area will likely find immediate applications.

*Acknowledgments.* We thank Andy Witkowski and Ramesh Bhashyam for discussions on estimation problems in database systems such as the NCR Teradata DBS. We also thank those who collaborated on the results surveyed in this paper. The research described in Section 3 is joint work with Noga Alon and Mario Szegedy. The research described in Section 5 is joint work with Vishy Poosala. The Aqua project, discussed briefly in Section 2.4, is joint work with Swarup Acharya, Vishy Poosala, Sridhar Ramaswamy, and Torsten Suel with additional contributions by Yair Bartal and S. Muthukrishnan. Other collaborators on the synopsis data structures research mentioned briefly in Section 6 are Christos Faloutsos, Avi Silberschatz, Jeff Vitter, and Min Wang. Finally, we thank Torsten Suel for helpful comments on an earlier draft of this paper.

## References

[AGMS97] N. Alon, P. B. Gibbons, Y. Matias, and M. Szegedy, *Dynamic probabilistic maintenance of self-join sizes in limited storage*, Manuscript, February 1997.

[AMS96] N. Alon, Y. Matias, and M. Szegedi, *The space complexity of approximating the frequency moments*, Proc. 28th ACM Symp. on the Theory of Computing, May 1996, Full version to appear in JCSS special issue for STOC'96, pp. 20–29.

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

[AZ96] G. Antoshenkov and M. Ziauddin, *Query processing and optimization in Oracle Rdb*, VLDB Journal **5** (1996), no. 4, 229–237.

[BCFM98] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher, *Min-wise independent permutations*, Proc. 30th ACM Symp. on the Theory of Computing, May 1998, pp. 327–336.

[BDF+97] D. Barbará, W. DuMouchel, C. Faloutsos, P. J. Haas, J. M. Hellerstein, Y. Ioannidis, H. V. Jagadish, T. Johnson, R. Ng, V. Poosala, K. A. Ross, and K. C. Sevcik, *The New Jersey data reduction report*, Bulletin of the Technical Committee on Data Engineering **20** (1997), no. 4, 3–45.

[BFS86] L. Babai, P. Frankl, and J. Simon, *Complexity classes in communication complexity theory*, Proc. 27th IEEE Symp. on Foundations of Computer Science, October 1986, pp. 337–347.

[BM96] R. J. Bayardo, Jr. and D. P. Miranker, *Processing queries for first-few answers*, Proc. 5th International Conf. on Information and Knowledge Management, November 1996, pp. 45–52.

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

[Cat92] J. Catlett, *Peepholing: Choosing attributes efficiently for megainduction*, Machine Learning: Proc. 9th International Workshop (ML92), July 1992, pp. 49–54.

[CMN98] S. Chaudhuri, R. Motwani, and V. Narasayya, *Random sampling for histogram construction: How much is enough?*, Proc. ACM SIGMOD International Conf. on Management of Data, June 1998, pp. 436–447.

[DNSS92] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri, *Practical skew handling in parallel joins*, Proc. 18th International Conf. on Very Large Data Bases, August 1992, pp. 27–40.

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

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

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

[FMS96] C. Faloutsos, Y. Matias, and A. Silberschatz, *Modeling skewed distribution using multifractals and the '80-20' law*, Proc. 22rd International Conf. on Very Large Data Bases, September 1996, pp. 307–317.

[FSGM+98] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. D. Ullman, *Computing iceberg queries efficiently*, Proc. 24th International Conf. on Very Large Data Bases, August 1998, pp. 299–310.

[GCB+97] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh, *Data cube: A relational aggregation operator generalizing group-by, cross-tabs, and sub-totals*, Data Mining and Knowledge Discovery **1** (1997), no. 1, 29–53.

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

[GM98] P. B. Gibbons and Y. Matias, *New sampling-based summary statistics for improving approximate query answers*, Proc. ACM SIGMOD International Conf. on Management of Data, June 1998, pp. 331–342.

[GMP97a] P. B. Gibbons, Y. Matias, and V. Poosala, *Aqua project white paper*, Tech. report, Bell Laboratories, Murray Hill, New Jersey, December 1997.

[GMP97b] P. B. Gibbons, Y. Matias, and V. Poosala, *Fast incremental maintenance of approximate histograms*, Proc. 23rd International Conf. on Very Large Data Bases, August 1997, pp. 466–475.

[Goo89] I. J. Good, *Surprise indexes and p-values*, J. Statistical Computation and Simulation **32** (1989), 90–92.

[GPA+98] P. B. Gibbons, V. Poosala, S. Acharya, Y. Bartal, Y. Matias, S. Muthukrishnan, S. Ramaswamy, and T. Suel, *AQUA: System and techniques for approximate query answering*, Tech. report, Bell Laboratories, Murray Hill, New Jersey, February 1998.

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

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

[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** (1993), no. 4, 709–748.

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

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

[JKM+98] H. V. Jagadish, N. Koudas, S. Muthukrishnan, V. Poosala, K. Sevcik, and T. Suel, *Optimal histograms with quality guarantees*, Proc. 24th International Conf. on Very Large Data Bases, August 1998, pp. 275–286.

[KS87] B. Kalyanasundaram and G. Schnitger, *The probabilistic communication complexity of set intersection*, Proc. 2nd Structure in Complexity Theory Conf., June 1987, pp. 41–49.

[KVI96] P. Krishnan, J. S. Vitter, and B. Iyer, *Estimating alphanumeric selectivity in the presence of wildcards*, Proc. ACM SIGMOD International Conf. on Management of Data, June 1996, pp. 282–293.

[Man94] U. Manber, *Finding similar files in a large file system*, Proc. Usenix Winter 1994 Technical Conf., January 1994, pp. 1–10.

[Man97] H. Mannila, *Inductive databases and condensed representations for data mining*, Proc. International Logic Programming Symposium, 1997, pp. 21–30.

[ML97] A. Moore and M. S. Lee, *Cached sufficient statistics for efficient machine learning with large datasets*, Tech. Report CMU-RI-TR-97-27, Robotics Institute, Carnegie-Mellon University, 1997, To appear in *J. Artificial Intelligence Research*.

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

[MP80] J. I. Munro and M. S. Paterson, *Selection and sorting with limited storage*, Theoretical Computer Science **12** (1980), no. 3, 315–323.

[MRL98] G. S. Manku, S. Rajagopalan, and B. G. Lindsley, *Approximate medians and other quantiles in one pass and with limited memory*, Proc. ACM SIGMOD International Conf. on Management of Data, June 1998, pp. 426–435.

[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.

[MT96] H. Mannila and H. Toivonen, *Multiple uses of frequent sets and condensed representations*, Proc. 2nd International Conf. on Knowledge Discovery and Data Mining, August 1996, pp. 189–194.

[Mul94] K. Mulmuley, *Computational geometry: An introduction through randomized algorithms*, Prentice Hall, Englewood Cliffs, NJ, 1994.

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

[MVW98] Y. Matias, J. S. Vitter, and M. Wang, *Wavelet-based histograms for selectivity estimation*, Proc. ACM SIGMOD International Conf. on Management of Data, June 1998, pp. 448–459.

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

[Olk93] F. Olken, *Random sampling from databases*, Ph.D. thesis, Computer Science, U.C. Berkeley, April 1993.

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

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

[Raz92] A. A. Razborov, *On the distributional complexity of disjointness*, Theoretical Computer Science **106** (1992), no. 2, 385–390.

[SKS97] A. Silberschatz, H. F. Korth, and S. Sudarshan, *Database system concepts*, third ed., McGraw-Hill, New York, 1997.

[TPC] *TPC-Committee. Transaction processing council (TPC)*, `http://www.tpc.org` .

[Vit85] J. S. Vitter, *Random sampling with a reservoir*, ACM Transactions on Mathematical Software **11** (1985), no. 1, 37–57.

[Vit98] J. S. Vitter, *External memory algorithms*, Proc. 17th ACM Symp. on Principles of Database Systems, June 1998, pp. 119–128.

[VL93] S. V. Vrbsky and J. W. S. Liu, *Approximate—a query processor that produces monotonically improving approximate answers*, IEEE Trans. on Knowledge and Data Engineering **5** (1993), no. 6, 1056–1068.

[VS94] J. S. Vitter and E. A. M. Shriver, *Algorithms for parallel memory I: Two-level memories*, Algorithmica **12** (1994), no. 2–3, 110–147.

[Yao83] A. C. Yao, *Lower bounds by probabilistic arguments*, Proc. 24th IEEE Symp. on Foundations of Computer Science, November 1983, pp. 420–428.

Information Sciences Research Center, Bell Laboratories, Room 2A-341, Lucent Technologies, 600 Mountain Avenue, Murray Hill, New Jersey 07974

*E-mail address*: `gibbons@research.bell-labs.com`

*URL*: `http://www.bell-labs.com/~pbgibbons/`

Department of Computer Science, Tel-Aviv University, Tel-Aviv 69978 Israel, and Information Sciences Research Center, Bell Laboratories

*E-mail address*: `matias@math.tau.ac.il`

*URL*: `http://www.math.tau.ac.il/~matias/`