What's Hot and What's Not: Tracking Most Frequent Items Dynamically

GRAHAM CORMODE and S. MUTHUKRISHNAN
Rutgers University

Most database management systems maintain statistics on the underlying relation. One of the important statistics is that of the "hot items" in the relation: those that appear many times (most frequently, or more than some threshold). For example, end-biased histograms keep the hot items as part of the histogram and are used in selectivity estimation. Hot items are used as simple outliers in data mining, and in anomaly detection in many applications.

We present new methods for dynamically determining the hot items at any time in a relation which is undergoing deletion operations as well as inserts. Our methods maintain small space data structures that monitor the transactions on the relation, and, when required, quickly output all hot items without rescanning the relation in the database. With user-specified probability, all hot items are correctly reported. Our methods rely on ideas from "group testing." They are simple to implement, and have provable quality, space, and time guarantees. Previously known algorithms for this problem that make similar quality and performance guarantees cannot handle deletions, and those that handle deletions cannot make similar guarantees without rescanning the database. Our experiments with real and synthetic data show that our algorithms are accurate in dynamically tracking the hot items independent of the rate of insertions and deletions.

Categories and Subject Descriptors: H.2.8 [**Database Management**]: Database Applications

General Terms: Algorithms, Measurement

Additional Key Words and Phrases: Data stream processing, approximate query answering.

1. INTRODUCTION

One of the most basic statistics on a database relation is that of which items are *hot*, that is, they occur frequently, but the set of hot items can change over time. This gives a useful measure of the skew of the data. High-biased and end-biased histograms [Ioannidis and Christodoulakis 1993; Ioannidis and Poosala 1995] specifically focus on hot items to summarize data distributions for selectivity

The first author was supported by NSF ITR 0220280 and NSF EIA 02-05116; the second author was supported by NSF EIA 0087022, NSF ITR 0220280, and NSF EIA 02-05116.
This is an extended version of an article which originally appeared as Cormode and Muthukrishnan [2003].
Authors' current addresses: G. Cormode, Room 2B-315, Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974; email: graham@dimacs.rutgers.edu; S. Muthukrishnan, Room 319, CoRE Building, Department of Computer and Information Sciences, 110 Frelinghuysen Road, Piscataway, NJ 08854; email: muthu@cs.rutgers.edu.

estimation. *Iceberg queries* generalize the notion of hot items in relation to aggregate functions over an attribute (or set of attributes) in order to find aggregate values above a specified threshold. Hot item sets in market data are influential in decision support systems. They also influence caching, load balancing, and other system performance issues. There are other areas—such as data warehousing, data mining, and information retrieval—where hot items find applications. Keeping track of hot items also arises in application domains outside traditional databases. For example, in telecommunication networks such as the Internet and telephone, it is of great importance for network operators to see meaningful statistics about the operation of the network. Keeping track of which network addresses are generating the most traffic allows management of the network, as well as giving a warning sign if this pattern begins to change unexpectedly. This has been studied extensively in the context of anomaly detection [Barbara et al. 2001; Demaine et al. 2002; Gilbert et al. 2001; Karp et al. 2003].

Our focus in this article is on dynamically maintaining hot items in the presence of delete and insert transactions. In many of the motivating applications above, the underlying data distribution changes, sometimes quite rapidly. Transactional databases undergo insert and delete operations, and it is important to propagate these changes to the statistics maintained on the database relations in a timely and accurate manner. In the context of continuous iceberg queries, this is apt since the iceberg aggregates have to reflect new data items that modify the underlying relations. In the networking application cited above, network connections start and end over time, and hot items change over time significantly. A thorough discussion by Gibbons and Matias [Gibbons and Matias 1999] described many applications for finding hot items and the challenges in maintaining them over a changing database relation. Also, Fang et al. [1998] presented an influential case for finding and maintaining hot items and, more generally, iceberg queries.

Formally, the problem is as follows. We imagine that we observe a sequence of n transactions on items. Without loss of generality, we assume that the item identifiers are integers in the range 1 to m. Throughout, we will assume the RAM model of computation, where all quantities and item identifiers can be encoded in one machine word. The net occurrence of any item x at time t, denoted $n_x(t)$, is the number of times it has been inserted less the number of times it has been deleted. The current frequency of any item is then given by $f_x(t) = n_x(t)/\sum_{y=1}^{m} n_y(t)$. The most frequent item at time t is the one with $f_x(t) = \max_y f_y(t)$. The k most frequent items at time t are those with the k largest $f_x(t)$'s. We are interested in the related notion of frequent items that we call *hot items*. An item x is said to be a *hot item* if $f_x(t) > 1/(k+1)$, that is, if it appears a significant fraction of the entire dataset; here k is a parameter. Clearly, there can be at most k hot items, and there may be none. We assume throughout that a basic integrity constraint is maintained, that $n_x(t)$ for every item is nonnegative (the number of deletions never exceeds the number of insertions). From now on, we drop the index t, and all occurrences will be treated as being taken at the current timestep t.

Our main results are highly efficient, randomized algorithms for maintaining hot items. There are three important characteristics to consider: the space used, the time to update the data structure following each transaction (the update time), and the time to produce the hot items (the query time). Our algorithms monitor the changes to the data distribution and maintain $O(k \log(k) \log(m))$ space summary data structures. Processing each transaction takes time $O(\log(k) \log(m))$. When queried, we can find all hot items in time $O(k \log(k) \log(m))$ from the summary data structure, without scanning the underlying relation. Additionally, given a user-specified parameter ϵ, the algorithms return no items whose frequency is less than $\frac{1}{k+1} - \epsilon$. More formally, for any user-specified probability δ, the algorithm succeeds with probability at least $1 - \delta$, as is standard in randomized algorithms.

Since k is typically very small compared to the size of the data, our results here maintain small summary data structures—significantly sublinear in the dataset size—and accurately detect hot items at any time in the presence of the full repertoire of inserts and deletes. Despite extensive work on this problem (which will be summarized in Section 2), most of the prior work with comparable guarantees works only for insert-only transactions. Prior work that deals with the fully general situation where both inserts and deletes are present cannot provide the guarantees we give, without rescanning the underlying database relation. Thus, our result is the first provable result for maintaining hot items, with small space.

A common approach to summarizing data distribution or finding hot items relies on keeping samples on the underlying database relation. These samples—deterministic or randomized—can be updated if data items are only inserted. Samples can then faithfully represent the underlying data relation. However, in the presence of deletes, in particular cases where the data distribution changes significantly over time, samples cannot be maintained without rescanning the database relation. For example, the entire set of sampled values may get erased from the relation by a sequence of deletes if there are very many deletions.

We present two different approaches for solving the problem. Our first result here relies on random sampling to construct groups ($O(k \log(k))$ sets) of items, but we further group such sets deterministically into a small number ($\log m$) of subgroups. Our summary data structure comprises a sum of the items in each group and subgroup. The grouping is based on error-correcting codes, and the entire procedure may be thought of as "group testing," which is described in more detail later. The second result makes use of $\log m$ small space "sketches" to act as oracles to approximate the count of any item or certain groups of items, and uses an intuitive divide and conquer approach to find the hot items. This is a different style of group testing, and the two methods give different guarantees for the problem. We also give additional time and space tradeoffs for both methods, where the time to process each update can be reduced by constant factors, at the cost of devoting extra space to the data structures. We perform a set of experiments on large datasets, which allow us to characterize further the advantages of each approach. We also see that, in practice, the methods given outperform their theoretical guarantees, and can operate very quickly using a small amount of space but still give almost perfect results.

Once the hot items have been identified, a secondary problem is to approximate the counts n_x of these items. We do not focus on this problem, since there are many existing solutions which can be applied to the problem of, given x, estimate n_x, in the presence of insertions and deletions [Gilbert et al. 2002b; Charikar et al. 2002; Cormode and Muthukrishnan 2004a]. However, we observe that for the solutions we propose, no additional storage is needed, since the information needed to make estimates of the count of items is already present in the data structures that we propose. We will show how to estimate the counts of individual items, but we do not give experimental results since experiments for these estimators can be found in prior work.

The rest of the article is organized as follows. In Section 2, we summarize previous work, which is rather extensive. In Section 3 and Section 4 we present our algorithms and prove their guarantees, and compare the different approaches in Section 5. In Section 6, we present an experimental study of our algorithms using synthetic data as well as real network data addressing the application domain cited earlier and show that our algorithms are effective and practical. Conclusions and closing remarks are given in Section 7.

2. PRELIMINARIES

If one is allowed $O(m)$ space, then a simple heap data structure will process each insert or delete operation in $O(\log m)$ time and find the hot items in $O(k \log m)$ time in the worst case [Aho et al. 1987]. Our focus here is on algorithms that only maintain a summary data structure, that is, one that uses sublinear space as it monitors inserts and deletes to the data.

In a fundamental article, Alon et al. [1996] proved that estimating $f^*(t) = \max_x f_x(t)$ is impossible with $o(m)$ space. Estimating the k most frequent items is at least as hard. Hence, research in this area studies related, relaxed versions of the problems. For example, finding hot items, that is, items each of which has frequency above $1/(k+1)$, is one such related problem. The lower bound of Alon et al. [1996] does not directly apply to this problem. But a simple information theory argument suffices to show that solving this problem exactly requires the storage of a large amount of information if we give a strong guarantee about the output. We provide the simple argument here for completeness.

LEMMA 2.1. *Any algorithm which guarantees to find all and only items which have frequency greater than* $1/(k+1)$ *must store* $\Omega(m)$ *bits.*

PROOF. Consider a set $S \subseteq \{1 \cdots m\}$. Transform S into a sequence of $n = |S|$ insertions of items by including x exactly once if and only if $x \in S$. Now process these transactions with the proposed algorithm. We can then use the algorithm to extract whether $x \in S$ or not: for some x, insert $\lfloor n/k \rfloor$ copies of x. Suppose $x \notin S$; then the frequency of x is $\lfloor n/k \rfloor / (n + \lfloor n/k \rfloor) = \lfloor n/k \rfloor / \lfloor n(k+1)/k \rfloor \leq \lfloor n/k \rfloor / (k+1) \lfloor n/k \rfloor = 1/(k+1)$, and so x will not be output. On the other hand, if $x \in S$ then $(\lfloor n/k \rfloor + 1)/(n + \lfloor n/k \rfloor) > (n/k)/(n + n/k) = 1/(k+1)$ and so x will be output. Hence, we can extract the set S, and so the space stored must be $\Omega(m)$ since, by an information theoretic argument, the space to store an arbitrary subset S is m bits. □

Table I. Summary of Previous Results on Insert-Only Methods (LV (Las Vegas) and MC (Monte Carlo) are types of randomized algorithms. See Motwani and Raghavan [1995] for details.)

Algorithm	Type	Time per item	Space
Lossy Counting [Manku and Motwani 2002]	Deterministic	$O(\log(n/k))$ amortized	$\Omega(k \log(n/k))$
Misra-Gries [Misra and Gries 1982]	Deterministic	$O(\log k)$ amortized	$O(k)$
Frequent [Demaine et al. 2002]	Randomized (LV)	$O(1)$ expected	$O(k)$
Count Sketch [Charikar et al. 2002]	Approximate, randomized (MC)	$O(\log(1/\delta))$	$\Omega(k/\epsilon^2 \log n)$

This also applies to randomized algorithms. Any algorithm which guarantees to output all hot items with probability at least $1 - \delta$, for some constant δ, must also use $\Omega(m)$ space. This follows by observing that the above reduction corresponds to the Index problem in communication complexity [Kushilevitz and Nisan 1997], which has one-round communication complexity $\Omega(m)$. If the data structure stored was $o(m)$ in size, then it could be sent as a message, and this would contradict the communication complexity lower bound.

This argument suggests that, if we are to use less than $\Omega(m)$ space, then we must sometimes output items which are not hot, since we will endeavor to include every hot item in the output. In our guarantees, we will instead guarantee that (with arbitrary probability) all hot items are output and no items which are far from being hot will be output. That is, no item which has frequency less than $\frac{1}{k+1} - \epsilon$ will be output, for some user-specified parameter ϵ.

2.1 Prior Work

Finding which items are hot is a problem that has a history stretching back over two decades. We divide the prior results into groups: those which find frequent items by keeping counts of particular items; those which use a filter to test each item; and those which accommodate deletions in a heuristic fashion. Each of these approaches is explained in detail below. The most relevant works mentioned are summarized in Table I.

2.1.1 *Insert-Only Algorithms with Item Counts.* The earliest work on finding frequent items considered the problem of finding an item which occurred more than half of the time [Boyer and Moore 1982; Fischer and Salzberg 1982]. This procedure can be viewed as a two-pass algorithm: after one pass over the data, a candidate is found, which is guaranteed to be the majority element if any such element exists. A second pass verifies the frequency of the item. Only a constant amount of space is used. A natural generalization of this method to find items which occur more than n/k times in two passes was given by Misra and Gries [1982]. The total time to process n items is $O(n \log k)$, with space $O(k)$

(recall that we assume throughout that any item label or counter can be stored in constant space). In the Misra and Gries implementation, the time to process any item is bounded by $O(k \log k)$ but this time is only incurred $O(n/k)$ times, giving the amortized time bound. The first pass generates a set of at most k candidates for the hot items, and the second pass computes the frequency of each candidate exactly, so the infrequent items can be pruned out. It is possible to drop the second pass, in which case at most k items will be output, among which all hot items are guaranteed to be included.

Recent interest in processing data streams, which can be viewed as one-pass algorithms with limited storage, has reopened interest in this problem (see surveys such as those by Muthukrishnan [2003] and Garofalakis et al. [2002]). Several authors [Demaine et al. 2002; Karp et al. 2003] have rediscovered the algorithm of Misra and Gries [1982], and using more sophisticated data structures have been able to process each item in expected $O(1)$ time while still keeping only $O(k)$ space. As before, the output guarantees to include all hot items, but some others will be included in the output, about which no guarantee of frequency is made. A similar idea was used by Manku and Motwani [2002] with the stronger guarantee of finding all items which occur more than n/k times and not reporting any that occur fewer than $n(\frac{1}{k} - \epsilon)$ times. The space required is bounded by $O(\frac{1}{\epsilon} \log \epsilon n)$—note that $\epsilon \leq \frac{1}{k}$ and so the space is effectively $\Omega(k \log(n/k))$. If we set $\epsilon = \frac{c}{k}$ for some small c then it requires time at worst $O(k \log(n/k))$ per item, but this occurs only every $1/k$ items, and so the total time is $O(n \log(n/k))$. Another recent contribution was that of Babcock and Olston [2003]. This is not immediately comparable to our work, since their focus was on maintaining the top-k items in a distributed environment, and the goal was to minimize communication. Counts of all items were maintained exactly at each location, so the memory space was $\Omega(m)$. All of these mentioned algorithms are deterministic in their operation: the output is solely a function of the input stream and the parameter k.

All the methods discussed thus far have certain features in common: in particular, they all hold some number of counters, each of which counts the number of times a single item is seen in the sequence. These counters are incremented whenever their corresponding item is observed, and are decremented or reallocated under certain circumstances. As a consequence, it is not possible to directly adapt these algorithms to the dynamic case where items are deleted as well as inserted. We would like the data structure to have the same contents following the deletion of an item, as if that item had never been inserted. But it is possible to insert an item so that it takes up a counter, and then later delete it: it is not possible to decide which item would otherwise have taken up this counter. So the state of the algorithm will be different from that reached without the insertions and deletions of the item.

2.1.2 *Insert-Only Algorithms with Filters.* An alternative approach to finding frequent items is based on constructing a data structure which can be used as a filter. This has been suggested several times, with different ways

to construct such filters being suggested. The general procedure is as follows: as each item arrives, the filter is updated to reflect this arrival and then the filter is used to test whether this item is above the threshold. If it is, then it is retained (for example, in a heap data structure). At output time, all retained items can be rechecked with the filter, and those which pass the filter are output. An important point to note is that, in the presence of deletions, this filter approach cannot work directly, since it relies on testing each item as it arrives. In some cases, the filter can be updated to reflect item deletions. However, it is important to realize that this does *not* allow the current hot items to be found from this: after some deletions, items seen in the past may become hot items. But the filter method can only pick up items which are hot when they reach the filter; it cannot retrieve items from the past which have since *become* frequent.

The earliest filter method appears to be due to Fang et al. [1998], where it was used in the context of iceberg queries. The authors advocated a second pass over the data to count exactly those items which passed the filter. An article which has stimulated interest in finding frequent items in the networking community was by Estan and Varghese [2002], who proposed a variety of filters to detect network addresses which are responsible for a large fraction of the bandwidth. In both these articles, the analysis assumed very strong hash functions which exhibit "perfect" randomness. An important recent result was that of Charikar et al. [2002], who gave a filter-based method using only limited (pairwise) independent hash functions. These were used to give an algorithm to find k items whose frequency was at least $(1-\epsilon)$ times the frequency of the kth most frequent item, with probability $1-\delta$. If we wish to only find items with count greater than $n/(k+1)$ then the space used is $O(\frac{k}{\epsilon^2}\log(n/\delta))$. A heap of frequent items is kept, and if the current items exceed the threshold, then the least frequent item in the heap is ejected, and the current item inserted. We shall return to this work later in Section 4.1, when we adapt and use the filter as the basis of a more advanced algorithm to find hot items. We will describe the algorithm in full detail, and give an analysis of how it can be used as part of a solution to the hot items problem.

2.1.3 *Insert and Delete Algorithms.* Previous work that studied hot items in the presence of both of inserts and deletes is sparse [Gibbons and Matias 1998, 1999]. These articles have proposed methods to maintain a sample of items and count of the number of times each item occurs in the data set, and focused on the harder problem of monitoring the k most frequent items. These methods work provably for the insert-only case, but provide no guarantees for the fully dynamic case with deletions. However, the authors studied how effective these samples are for the deletion case through experiments. Gibbons et al. [1997] presented methods to maintain various histograms in the presence of inserts and deletes using a "backing sample," but these methods too need access to large portion of the data periodically in the presence of deletes.

A recent theoretical work presented provable algorithms for maintaining histograms with guaranteed accuracy and small space [Gilbert et al. 2002a].

The methods in this article can yield algorithms for maintaining hot items, but the methods are rather sophisticated and use powerful range summable random variables, resulting in $k \log^{O(1)} n$ space and time algorithms where the $O(1)$ term is quite large. We draw some inspiration from the methods in this article—we will use ideas similar to the "sketching" developed in Gilbert et al. [2002a], but our overall methods are much simpler and more efficient. Finally, recent work in maintaining quantiles [Gilbert et al. 2002b] is similar to ours since it keeps the sum of items in random subsets. However, our result is, of necessity, more involved, involving a random group generation phase based on group testing, which was not needed in [Gilbert et al. 2002b]. Also, once such groups are generated, we maintain sums of deterministic sets (in contrast to the random sets as in Gilbert et al. [2002b]), given again by error correcting codes. Finally, our algorithm is more efficient than the $\Omega(k^2 \log^2 m)$ space and time algorithms given in Gilbert et al. [2002b].

2.2 Our Approach

We propose some new approaches to this problem, based on ideas from group testing and error-correcting codes. Our algorithms depend on ideas drawn from group testing [Du and Hwang 1993]. The idea of group testing is to arrange a number of tests, each of which groups together a number of the m items in order to find up to k items which test "positive." Each test reports either "positive" or "negative" to indicate whether there is a positive item among the group, or whether none of them is positive. The familiar puzzle of how to use a pan balance to find one "positive" coin among n good coins, of equal weight, where the positive coin is heavier than the good coins, is an example of group testing. The goal is to minimize the number of tests, where each test in the group testing is applied to a subset of the items (a group). Our goal of finding up to k hot items can be neatly mapped onto an instance of group testing: the hot items are the positive items we want to find.

Group testing methods can be categorized as *adaptive* or *nonadaptive*. In adaptive group testing, the members of the next set of groups to test can be specified after learning the outcome of the previous tests. Each set of tests is called a *round*, and adaptive group testing methods are evaluated in terms of the number of rounds, as well as the number of tests, required. By contrast, nonadaptive group testing has only one round, and so all groups must be chosen without any information about which groups tested positive. We shall give two main solutions for finding frequent items, one based on nonadaptive and the other on adaptive group testing. For each, we must describe how the groups are formed from the items, and how the tests are performed. An additional challenge is that our tests here are not perfect, but have some chance of failure (reporting the wrong result). We will prove that, in spite of this, our algorithms can guarantee finding all hot items with high probability. The algorithms we propose differ in the nature of the guarantees that they give, and result in different time and space guarantees. In our experimental studies, we were able to explore these differences in more detail, and to describe the different situations which each of these algorithms is best suited to.

3. NONADAPTIVE GROUP TESTING

Our general procedure is as follows: we divide all items up into several (overlapping) groups. For each transaction on an item x, we determine which groups it is included in (denoting these $G(x)$). Each group is associated with a counter, and for an insertion we increment the counter for all $G(x)$; for a deletion, we correspondingly decrement these counters. The test will be whether the count for a subset exceeds a certain threshold: this is evidence that there may a hot item within the set. Identifying the hot items is a matter of putting together the information from the different tests to find an overall answer.

There are a number of challenges involved in following this approach: (1) bounding the number of groups required; (2) finding a concise representation of the groups; and (3) giving an efficient way to go from the results of tests to the set of hot items. We shall be able to address all of these issues. To give greater insight into this problem, we first give a simple solution to the $k = 1$ case, which is to find an item that occurs more than half of the time. Later, we will consider the more general problem of finding $k > 1$ hot items, which will use the procedure given below as a subroutine.

3.1 Finding the Majority Item

If an item occurs more than half the time, then it is said to be the *majority item*. While finding the majority item is mostly straightforward in the insertions-only case (it is solved in constant space and constant time per insertion by the algorithms of Boyer and Moore [1982] and Fischer and Salzberg [1982]), in the dynamic case, it looks less trivial. We might have identified an item which is very frequent, only for this item to be the subject of a large number of deletions, meaning that some other item is now in the majority.

We give an algorithm to solve this problem by keeping $\lceil \log_2 m \rceil + 1$ counters. The first counter, c_0, merely keeps track of $n(t) = \sum_x n_x(t)$, which is how many items are "live": in other words, we increment this counter on every insert, and decrement it on every deletion. The remaining counters are denoted $c_1 \cdots c_j$. We make use of the function $bit(x, j)$, which reports the value of the jth bit of the binary representation of the integer x; and $gt(x, y)$, which returns 1 if $x > y$ and 0 otherwise. Our procedures are as follows:

Insertion of item x: increment each counter c_j such that $bit(x, j) = 1$ in time $O(\log m)$.
Deletion of x: decrement each counter c_j such that $bit(x, j) = 1$ in time $O(\log m)$.
Search: if there is a majority, then it is given by $\sum_{j=1}^{\log_2 m} 2^j gt(c_j, n/2)$, computed in time $O(\log m)$.

The arrangement of the counters is shown graphically in Figure 1. The two procedures of this method—one to process updates, another to identify the majority element—are given in Figure 2 (where *trans* denotes whether the transaction is an insertion or a deletion).

THEOREM 3.1. *The algorithm in Figure 2 finds a majority item if there is one with time* $O(\log m)$ *per update and search operation.*

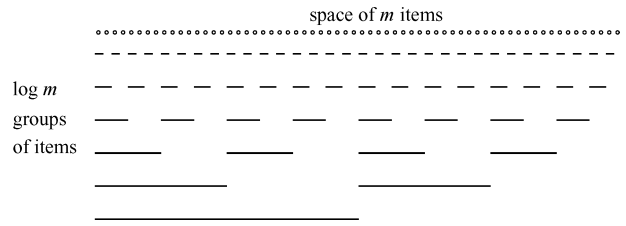

Fig. 1. Each test includes half of the range $[1 \cdots m]$, corresponding to the binary representation of values.

```
UpdateCounters(x, trans, c[0 ... log m])
  if (trans = insertion) then
    d ← 1
  else
    d ← −1
  c[0] ← c[0] + d
  for j = 1 to log m
    c[j] ← c[j] + bit(x, j) ∗ d
```

```
FindMajority(c[0 ... log m])
  x ← 0
  t ← 1
  for j ← 1 to log m
    if (c[j] > c[0]/2) then
      x ← x + t
    t ← 2 ∗ t
  return(x)
```

Fig. 2. Algorithm to find the majority element in a sequence of update.

PROOF. We make two observations: first, that the state of the data structure is equivalent to that following a sequence of c_0 insertions only, and second, that in the insertions only case, this algorithm identifies a majority element. For the first point, it suffices to observe that the effect of each deletion of an element x is to precisely cancel out the effect of a prior insertion of that element. Following a sequence of I insertions and D deletions, the state is precisely that obtained if there had been $I - D = n$ insertions only.

The second part relies on the fact that if there is an item whose count is greater than $n/2$ (that is, it is in the majority), then for any way of dividing the elements into two sets, the set containing the majority element will have weight greater than $n/2$, and the other will have weight less than $n/2$. The tests are arranged so that each test determines the value of a particular bit of the index of the majority element. For example, the first test determines whether its index is even or odd by dividing on the basis of the least significant bit. The $\log m$ tests with binary outcomes are necessary and sufficient to determine the index of the majority element. □

Note that this algorithm is completely deterministic, and guarantees always to find the majority item if there is one. If there is no such item, then still *some* item will be returned, and it will not be possible to distinguish the difference based on the information stored. The simple structure of the tests is standard in group testing, and also resembles the structure of the Hamming single error-correcting code.

3.2 Finding *k* Hot Items

When we perform a test based on comparing the count of items in two buckets, we extract from this a single bit of information: whether there is a hot item present in the set or not. This leads immediately to a lower bound on the number

of tests necessary: to locate k items among m locations requires $\log_2\binom{m}{k} \geq k\log(m/k)$ bits.

We make the following observation: suppose we selected a group of items to monitor which happened to contain exactly one hot item. Then we could apply the algorithm of Section 3.1 to this group (splitting it into a further $\log m$ subsets) and, by keeping $\log m$ counters, identify which item was the hot one. We would simply have to "weigh" each bucket, and, providing that the total weight of other items in the group were not too much, the hot item would always be in the heavier of the two buckets.

We could choose each group as a completely random subset of the items, and apply the algorithm for finding a single majority item described at the start of this section. But for a completely random selection of items then in order to store the description of the groups, we would have to list every member of every group explicitly. This would consume a very large amount of space, at least would be linear in m. So instead, we shall look for a concise way to describe each group, so that given an item we can quickly determine which groups it is a member of. We shall make use of hash functions, which will map items onto the integers $1 \cdots W$, for some W that we shall specify later. Each group will consist of all items which are mapped to the same value by a particular hash function. If the hash functions have a concise representation, then this describes the groups in a concise fashion. It is important to understand exactly how strong the hash functions need to be to guarantee good results.

3.2.1 *Hash Functions.* We will make use of universal hash functions derived from those given by Carter and Wegman [1979]. We define a family of hash functions $f_{a,b}$ as follows: fix a prime $P > m > W$, and draw a and b uniformly at random in the range $[0 \cdots P-1]$. Then set

$$f_{a,b}(x) = ((ax + b \mod P) \mod W).$$

Using members of this family of functions will define our groups. Each hash function is defined by a and b, which are integers less than P. P itself is chosen to be $O(m)$, and so the space required to represent each hash function is $O(\log m)$ bits.

Fact 3.2 (*Proposition 7 of Carter and Wegman* [1979]). Over all choices of a and b, for $x \neq y$, $\Pr[f_{a,b}(x) = f_{a,b}(y)] \leq 1/W$.

We can now describe the data structures that we will keep in order to allow us to find up to k hot items.

3.2.2 *Nonadaptive Group Testing Data Structure.* The group testing data structure is initialized with two parameters W and T, and has three components:

— a three-dimensional array of counters c, of size $T \times W \times (\log(m)+1)$;
— T universal hash functions h, defined by $a[1 \cdots T]$ and $b[1 \cdots T]$ so $h_i = f_{a[i],b[i]}$;
— the count n of the current number of items.

```
INITIALIZE(T, W)
  n ← 0
  for i = 1 to T
    for j = 0 to W − 1
      for k = 0 to log m
        c[i][j][k] = 0
    a[i] = Random(0, P − 1)
    b[i] = Random(0, P − 1)

PROCESSITEM(x, tt, T, W)
  if (tt = insertion) then
    n ← n + 1
  else
    n ← n − 1
  for i ← 1 to T
    UPDATECOUNTERS(x, tt, c[i][h_i(x)])

GROUPTEST(T, W, k)
  for i ← 1 to T
    for j ← 0 to W − 1
      r ← 1; t ← n/(k + 1); x ← 0
      if (c[i][j][0]) > t then
        for l ← 1 to log m
          p ← c[i][j][l]; q ← c[i][j][0] − p;
          if (p ≤ t ∧ q ≤ t) ∨ (p > t ∧ q > t)
          then
            skip to next value of i
          if (p > t) then
            x ← x + r
          r ← 2 ∗ r
        if h_i(x) = j then
          for l = 1 to T
            check c[l][h_l(x)][0] > t
          if checks passed then
            output(x)
```

Fig. 3. Procedures for finding hot items using nonadaptive group testing.

The data structure is initialized by setting all the counters, $c[1][0][0]$ to $c[T][W-1][\log m]$, to zero, and by choosing values for each entry of a and b uniformly at random in the range $[0 \cdots P-1]$. The space used by the data structure is $O(TW \log m)$. We shall specify values for W and T later. We will write h_i to indicate the ith hash function, so $h_i(x) = a[i] * x + b[i] \bmod P \bmod W$. Let $G_{i,j} = \{x | h_i(x) = j\}$ be the (i, j)th group. We will use $c[i][j][0]$ to keep the count of the current number of items within the $G_{i,j}$. For each such group, we shall also keep counts for $\log m$ *subgroups*, defined as $G_{i,j,l} = \{x | x \in G_{i,j} \wedge bit(x, l) = 1\}$. These correspond to the groups we kept for finding a majority item. We will use $c[i][j][l]$ to keep count of the current number of items within subgroup $G_{i,j,l}$. This leads to the following update procedure.

3.2.3 *Update Procedure.* Our procedure in processing an input item x is to determine which groups it belongs to, and to update the $\log m$ counters for each of these groups based on the bit representation of x in exactly the same way as the algorithm for finding a majority element. If the transaction is an insertion, then we add one to the appropriate counters, and subtract one for a deletion. The current count of items is also maintained. This procedure is shown in pseudocode as PROCESSITEM (x, *trans*, T, W) in Figure 3. The time to perform an update is the time taken to compute the T hash functions, and to modify $O(T \log m)$ counters.

At any point, we can search the data structure to find hot items. Various checks are made to avoid including in the output any items which are not hot. In group testing terms, the test that we will use is whether the count for a group or subgroup exceeds the threshold needed for an item to be hot, which is $n/(k+1)$. Note that any group which contains a hot item will pass this test, but that it is possible that a group which does not contain a hot item can also pass this test. We will later analyze the probability of such an event, and show that it can be made quite small.

3.2.4 *Search Procedure.* For each group, we will use the information about the group and its subgroups to test whether there is a hot item in the group, and if so, to extract the identity of the hot item. We process each group $G_{i,j}$ in turn. First, we test whether there can be a hot item in the group. If $c[i][j][0] \leq n/(k+1)$ then there cannot be a hot item in the group, and so the group is rejected. Then we look at the count of every subgroup, compared to the count of the whole group, and consider the four possible cases:

$c[i][j][l] > \frac{n}{k+1}$?	$c[i][j][0] - c[i][j][l] > \frac{n}{k+1}$?	Conclusion
No	No	Cannot be a hot item in the group, so reject group
No	Yes	If a hot item x is in group, then $bit(l, x) = 0$
Yes	No	If a hot item x is in group, then $bit(l, x) = 1$
Yes	Yes	Not possible to identify the hot item, so reject group

If the group is not rejected, then the identity of the candidate hot item, x, can be recovered from the tests. Some verification of the hot items can then be carried out.

— The candidate item must belong to the group it was found in, so check $h_i(x) = j$.
— If the candidate item is hot, then every group it belongs in should be above the threshold, so check that $c[i][h_i(x)][0] > n/(k+1)$ for all i.

The time to find all hot items is $O(T^2 W \log m)$. There can be at most TW candidates returned, and checking them all takes worst-case time $O(T)$ each. The full algorithms are illustrated in Figure 3. We now show that for appropriate choices of T and W we can first ensure that all hot items are found, and second ensure that no items are output which are far from being hot.

LEMMA 3.3. *Choosing $W \geq 2k$ and $T = \log_2(\frac{k}{\delta})$ for a user chosen parameter δ ensures that the probability of all hot items being output is at least $1 - \delta$.*

PROOF. Consider each hot item x, in turn, remembering that there are at most k of these. Using Fact 3.2 about the hash functions, then the probability for any other item falling into the same group as x under the ith hash function is given by $1/W \leq \frac{1}{2k}$. Using linearity of expectation, then the expectation of the total frequency of other items which land in the same group as item x is

$$\mathsf{E}\left(\sum_{y \neq x, h_i(y)=h_i(x)} f_y\right) = \sum_{y \neq x} f_y \cdot \mathsf{Pr}[h_i(y) = h_i(x)] \leq \sum_{y \neq x} \frac{f_y}{2k} \leq \frac{1 - f_x}{2k} \leq \frac{1}{2(k+1)}. \tag{1}$$

Our test cannot fail if the total weight of other items which fall in the same bucket is less than $1/(k+1)$. This is because each time we compare the counts of items in the group we conclude that the hot item is in the half with greater count. If the total frequency of other items is less than $1/(k+1)$, then the hot

item will always be in the heavier half, and so, using a similar argument to that for the majority case, we will be able to read off the index of the hot item using the results of $\log m$ groups. The probability of failing due to the weight of other items in the same bucket being more than $1/(k+1)$ is bounded by the Markov inequality as $\frac{1}{2}$, since this is at least twice the expectation. So the probability that we fail on every one of the T independent tests is less than $\frac{1}{2}^{\log(k/\delta)} = \delta/k$. Using the union bound, then, over all hot items, the probability of any of them failing is less than δ, and so each hot item is output with probability at least $1-\delta$. □

LEMMA 3.4. *For any user specified fraction $\epsilon \leq \frac{1}{k+1}$, if we set $W \geq \frac{2}{\epsilon}$ and $T = \log_2(k/\delta)$, then the probability of outputting any item y with $f_y < \frac{1}{k+1} - \epsilon$ is at most δ/k.*

PROOF. This lemma follows because of the checks we perform on every item before outputting it. Given a candidate item, we check that every group it is a member of is above the threshold. Suppose the frequency of the item y is less than $(\frac{1}{k+1} - \epsilon)$. Then the frequency of items which fall in the same group under hash function i must be at least ϵ, to push the count for the group over the threshold for the test to return positive. By the same argument as in the above lemma, the probability of this event is at most $\frac{1}{2}$. So the probability that this occurs in all groups is bounded by $\frac{1}{2}^{\log k/\delta} = \delta/k$. □

Putting these two lemmas together allows us to state our main result on nonadaptive group testing:

THEOREM 3.5. *With probability at least $1-\delta$, then we can find all hot items whose frequency is more than $\frac{1}{k+1}$, and, given $\epsilon \leq \frac{1}{k+1}$, with probability at least $1-\delta/k$ each item which is output has frequency at least $\frac{1}{k+1} - \epsilon$ using space $O(\frac{1}{\epsilon}\log(m)\log(k/\delta))$ words. Each update takes time $O(\log(m)\log(k/\delta))$. Queries take time no more than $O(\frac{1}{\epsilon}\log^2(k/\delta)\log m)$.*

PROOF. This follows by setting $W = \frac{2}{\epsilon}$ and $T = \log(k/\delta)$, and applying the above two lemmas. To process an item, we compute T hash functions, and update $T\log m$ counters, giving the time cost. To extract the hot items involves a scan over the data structure in linear time, plus a check on each hot item found that takes time at most $O(T)$, giving total time $O(T^2 W \log m)$. □

Next, we describe additional properties of our method which imply its stability and resilience.

COROLLARY 3.6. *The data structure created with $T = \log(k/\delta)$ can be used to find hot items with parameter k' for any $k' < k$ with the same probability of success $1-\delta$.*

PROOF. Observe in Lemma 3.3 that, to find k' hot items, we required $W \geq 2k'$. If we use a data structure created with $W \geq 2k$, then $W \geq 2k > 2k'$, and so the data structure can be used for any value of k less than the value it was created for. Similarly, we have more tests than we need, which can only

help the accuracy of the group testing. All other aspects of the data structure are identical. So, if we run the procedure with a higher threshold, then with probability at least $1 - \delta$, we will find the hot items. □

This property means that we can fix k to be as large as we want, and are then able to find hot items with any frequency greater than $1/(k+1)$ determined at query time.

COROLLARY 3.7. *The output of the algorithm is the same for any reordering of the input data.*

PROOF. During any insertion or deletion, the algorithm takes the same action and does not inspect the contents of the memory. It just adds or subtracts values from the counters, as a function solely of the item value. Since addition and subtraction commute, the corollary follows. □

3.2.5 *Estimation of Count of Hot Items.* Once the hot items have been identified, we may wish to additionally estimate the count, n_x, of each of these items. One approach would be to keep a second data structure enabling the estimation of the counts to be made. Such data structures are typically compact, fast to update, and give accurate answers for items whose count is large, that is, hot items [Gilbert et al. 2002b; Charikar et al. 2002; Cormode and Muthukrishnan 2004a]. However, note that the data structure that we keep embeds a structure that allows us to compute an estimate of the weight of each item [Cormode and Muthukrishnan 2004a].

COROLLARY 3.8. *Computing* $\min_i c[i][h_i(x)][0]$ *gives a good estimate for* n_x *with probability at least* $1 - (\delta/k)$.

PROOF. This follows from the proofs of Lemma 3.3 and Lemma 3.4. Each estimate $c[i][h_i(x)][0] = n_x + \sum_{y \neq x, h_i(x)=h_i(y)} n_y$. But by Lemma 3.3, this additional noise is bounded by ϵn with constant probability at least $\frac{1}{2}$, as shown in Equation (1). Taking the minimum over all estimates amplifies this probability to $1 - (\delta/k)$. □

3.3 Time-Space Tradeoff

In certain situations when transactions are occurring at very high rates, it is vital to make the update procedure as fast as possible. One of the drawbacks of the current procedure is that it depends on the product of T and $\log m$, which can be slow for items with large identifiers. For reducing the time dependency on T, note that the data structure is intrinsically parallelizable: each of the T hash functions can be applied in parallel, and the relevant counts modified separately. In the experimental section we will show that good results are observed even for very small values of T; therefore, the main bottleneck is the dependence on $\log m$.

The dependency on $\log m$ arises because we need to recover the identifier of each hot item, and we do this 1 bit at a time. Our observation here is that we can find the identifier in different units, for example, 1 byte at a time, at the expense of extra space usage. Formally, define $dig(x, i, b)$ to be the ith digit

in the integer x when x is written in base $b \geq 2$. Within each group, we keep $(b-1) \times \log_b m$ subgroups: the i, jth subgroup counts how many items have $dig(x, i, b) = j$ for $i = 1 \cdots \log_b m$ and $j = 1 \cdots b-1$. We do not need to keep a subgroup for $j = 0$ since this count can be computed from the other counts for that group. Note that $b = 2$ corresponds to the binary case discussed already, and $b = m$ corresponds to the simple strategy of keeping a count for every item.

THEOREM 3.9. *Using the above procedure, with probability at least* $1-\delta$*, then we can find all hot items whose frequency is more than* $\frac{1}{k+1}$*, and with probability at least* $1-(\delta/k)$*, each item which is output has frequency at least* $\frac{1}{k+1} - \epsilon$ *using space* $O(\frac{b}{\epsilon}\log_b(m)\log(k/\delta))$ *words. Each update takes time* $O(\log_b(m)\log(k/\delta))$ *and queries take* $O(\frac{b}{\epsilon}\log_b(m)\log^2(k/\delta))$ *time.*

PROOF. Each subgroup now allows us to read off one digit in the base-b representation of the identifier of any hot item x. Lemma 3.3 applies to this situation just as before, as does Lemma 3.4. This leads us to set W and T as before. We have to update one counter for each digit in the base b representation of each item for each transaction, which corresponds to $\log_b m$ counters per test, giving an update time of $O(T\log_b(m))$. The space required is for the counters to record the subgroups of TW groups, and there are $(b-1)\log_b(m)$ subgroups of every group, giving the space bounds. □

For efficient implementations, it will generally be preferable to choose b to be a power of 2, since this allows efficient computation of indices using bit-level operations (shifts and masks). The space cost can be relatively high for speedups: choosing $b = 2^8$ means that each update operation is eight times faster than for $b = 2$, but requires 32 times more space. A more modest value of b may strike the right balance: choosing $b = 4$ doubles the update speed, while the space required increases by 50%. We investigate the effects of this tradeoff further in our experimental study.

4. ADAPTIVE GROUP TESTING

The more flexible model of adaptive group testing allows conceptually simpler choices of groups, although the data structures required to support the tests become more involved. The idea is a very natural "divide-and-conquer" style approach, and as such may seem straightforward. We give the full details here to emphasize the relation between viewing this as an adaptive group testing procedure and the above nonadaptive group testing approach. Also, this method does not seem to have been published before, so we give the full description for completeness.

Consider again the problem of finding a majority item, assuming that one exists. Then an adaptive group testing strategy is as follows: test whether the count of all items in the range $\{1 \cdots m/2\}$ is above $n/2$, and also whether the count of all items in the range $\{m/2+1 \cdots m\}$ is over the threshold. Recurse on whichever half contains more than half the items, and the majority item is found in $\lceil \log_2 m \rceil$ rounds.

The question is: how to support this adaptive strategy as transactions are seen? As counts increase and decrease, we do not know in advance which queries

```
AdaptiveUpdateItem(x, tt, T, W)
  if tt = insertion then
    d ← 1
  else
    d ← −1
  n ← n + d
  for i ← 1 to log m
    for j ← 1 to T
      t[i][j][h_i(x)] ← t[i][j][h_i(x)] + g_i(x) * d
```

```
Adaptive(l, r, thresh)
  if oracle(l, r) > thresh then
    if (l = r) then
      output(l);
  else
    Adaptive(l, (l + r − 1)/2, thresh);
    Adaptive((l + r + 1)/2, r, thresh);

AdaptiveGroupTest(k)
  call Adaptive(1, m, n/(k + 1))
```

Fig. 4. Adaptive group testing algorithms.

will be posed, and so the solution seems to be to keep counts for every test that could be posed—but there are $\Omega(m)$ such tests, which is too much to store. The solution comes by observing that we do not need to know counts exactly, but rather it suffices to use approximate counts, and these can be supported using a data structure that is much smaller, with size dependent on the quality of approximation. We shall make use of the fact that the range of items can be mapped onto the integers $1 \cdots m$. We will initially describe an adaptive group testing method in terms of an oracle that is assumed to give exact answers, and then show how this oracle can be realized approximately.

Definition 4.1. A *dyadic range sum oracle* returns the (approximate) sum of the counts of items in the range $l = (i2^j + 1) \cdots r = (i+1)2^j$ for $0 \leq j \leq \log m$ and $0 \leq i \leq m/2^j$.

Using such an oracle, which reflects the effect of items arriving and departing, it is possible to find all the hot items, with the following binary search divide-and-conquer procedure. For simplicity of presentation, we assume that m, the range of items, is a power of 2. Beginning with the full range, recursively split in two. If the total count of any range is less than $n/(k+1)$, then do not split further. Else, continue splitting until a hot item is found. It follows that $O(k \log(m/k))$ calls are made to the oracle. The procedure is presented as AdaptiveGroupTest on the right in Figure 4.

In order to implement dyadic range sum oracles, define an approximate count oracle to return the (approximate) count of the item x. A dyadic range sum oracle can be implemented using $j = 0 \cdots \log m$ approximate count oracles: for each item in the stream x, insert $\lfloor \frac{x}{2^j} \rfloor$ into the jth approximate count oracle, for all j. Recent work has given several methods of implementing the approximate count oracle, which can be updated to reflect the arrival or departure of any item. We now list three examples of these and give their space and update time bounds:

— The "tug of war sketch" technique of Alon et al. [1999] uses space and time $O(\frac{1}{\epsilon^2} \log \frac{1}{\delta})$ to approximate any count up to ϵn with a probability of at least $1 - \delta$.
— The method of random subset sums described in Gilbert et al. [2002b] uses space and time $O(\frac{1}{\epsilon^2} \log \frac{1}{\delta})$.
— The method of Charikar et al. [2002]. builds a structure which can be used to approximate the count of any item correct upto ϵn in space $O(\frac{1}{\epsilon^2} \log \frac{1}{\delta})$ and time per update $O(\log \frac{1}{\delta})$.

The fastest of these methods is that of Charikar et al. [2002], and so we shall adopt this as the basis of our adaptive group testing solution. In the next section we describe and analyze the data structure and algorithms for our purpose of finding hot items.

4.1 CCFC Count Sketch

We shall briefly describe and analyze the CCFC count sketch.[1] This is a different and shorter analysis compared to that given in Charikar et al. [2002], since here the goal is to estimate each count to within an error in terms of the total count of all items rather than in the count of the kth most frequent item, as was the case in the original article.

4.1.1 *Data Structure.* The data structure used consists of a table of counters t, with width W and height T, initialized to zero. We also keep T pairs of universal hash functions: $h_1 \cdots h_T$, which map items onto $1 \cdots W$, and $g_1 \cdots g_T$, which map items onto $\{-1, +1\}$.

4.1.2 *Update Routine.* When an insert transaction of item x occurs, we update $t[i][h_i(x)] \leftarrow t[i][h_i(x)] + g_i[x]$ for all $i = 1 \cdots T$. For a delete transaction, we update $t[i][h_i(x)] \leftarrow t[i][h_i(x)] - g_i[x]$ for all $i = 1 \cdots T$.

4.1.3 *Estimation.* To estimate the count of x, compute $\text{median}_i(t[i][h_i(x)] \cdot g_i(x))$.

4.1.4 *Analysis.* Use the random variable X_i to denote $t[i][h_i(x)] \cdot g_i(x)$. The expectation of each estimate is

$$\mathsf{E}(X_i) = n_x + \sum_{y \neq x} \mathsf{Pr}[h_i(y) = h_i(x)] \cdot (\mathsf{Pr}[g_i(x) = g_i(y)] - \mathsf{Pr}[g_i(x) \neq g_i(y)]) = n_x$$

since $\mathsf{Pr}[g_i(x) = g_i(y)] = \frac{1}{2}$. The variance of each estimate is

$$\begin{aligned}
\mathsf{Var}(X_i) &= \mathsf{E}(X_i^2) - \mathsf{E}(X_i)^2 && (2)\\
&= \mathsf{E}(g_i(x)^2 (t[i][h_i(x)])^2) - n_x^2 && (3)\\
&= 2 \sum_{y \neq x, z} n_y n_z \mathsf{Pr}[h_i(y) = h_i(z)](\mathsf{Pr}[g_i(x) = g_i(y)] - \mathsf{Pr}[g_i(x) \neq g_i(y)]) && (4)\\
&\quad + n_x^2 + \sum_{y \neq x} g_i^2(y) n_y^2 \mathsf{Pr}[h_i(y) = h_i(x)] - n_x^2 && (5)\\
&= \sum_{y \neq x} \frac{n_y^2}{W} \leq \frac{n^2}{W}. && (6)
\end{aligned}$$

Using the Chebyshev inequality, it follows that $\mathsf{Pr}[|X_i - x| > \frac{\sqrt{2}n}{\sqrt{W}}] < \frac{1}{2}$. Taking the median of T estimates amplifies this probability to $2^{T/4}$, by a standard Chernoff bounds argument [Motwani and Raghavan 1995].

[1] CCFC denotes the initials of the authors of Charikar et al. [2002].

4.1.5 *Space and Time.* The space used is for the WT counters and the $2T$ hash functions. The time taken for each update is the time to compute the $2T$ hash functions, and update T counters.

THEOREM 4.2. *By setting* $W = \frac{2}{\epsilon^2}$ *and* $T = 4 \log \frac{1}{\delta}$ *then we can estimate the count of any item up to error* $\pm \epsilon n$ *with probability at least* $1 - \delta$.

4.2 Adaptive Group Testing Using CCFC Count Sketch

We can now implement an adaptive group testing solution to finding hot items. The basic idea is to apply the adaptive binary search procedure using the above count sketch to implement the dyadic range sum oracle. The full procedure is shown in Figure 4.

THEOREM 4.3. *Setting* $W = \frac{2}{\epsilon^2}$ *and* $T = \log \frac{k \log m}{\delta}$ *allows us to find every item with frequency greater than* $\frac{1}{k+1} + \epsilon$, *and report no item with frequency less than* $\frac{1}{k+1} - \epsilon$, *with a probability of at least* $1 - \delta$. *The space used is* $O(\frac{1}{\epsilon^2} \log(m) \log \frac{k \log m}{\delta})$ *words, and the time to perform each update is* $O(\log(m) \log \frac{k \log m}{\delta})$. *The query time is* $O(k \log m \log \frac{k \log m}{\delta})$ *with a proabability of at least* $1 - \delta$.

PROOF. We set the probability of failure to be low ($\frac{\delta}{k \log m}$), so that for the $O(k \log m)$ queries that we pose to the oracle, there is probability at most δ of *any* of them failing, by the union bound. Hence, we can assume that with a probability of at least $1 - \delta$, all approximations are within the $\pm \epsilon n$ error bound. Then, when we search for hot items, any range containing a hot item will have its approximate count reduced by at most ϵn. This will allow us to find the hot item, and output it if its frequency is at least $\frac{1}{k+1} + \epsilon$. Any item which is output must pass the final test, based on the count of just that item, which will not happen if its frequency is less than $\frac{1}{k+1} - \epsilon$.

Space is needed for $\log(m)$ sketches, each of which has size $O(TW)$ words. For these settings of T and W, we obtain the space bounds listed in the theorem. The time per update is that needed to compute $2T \log(m)$ hash values, and then to update up to this many counters, which gives the stated update time. □

4.2.1 *Hot Item Count Estimation.* Note that we can immediately extract the estimated counts for each hot item using the data structure, since the count of item x is given by using the lowest-level approximate count. Hence, the count n_x is estimated with error at most ϵn in time $O(\log(m) \log \frac{k \log m}{\delta})$.

4.3 Time-Space Tradeoffs

As with the nonadaptive group testing method, the time cost for updates depends on T and $\log m$. Again, in practice we found that small values of T could be used, and that computation of the hash functions could be parallelized for extra speedup. Here, the dependency on $\log m$ is again the limiting factor. A similar trick to the nonadaptive case is possible, to change the update time dependency to $\log_b m$ for arbitrary b: instead of basing the oracle on dyadic ranges, base it on b-adic ranges. Then only $\log_b m$ sketches need to be updated for each transaction. However, under this modification, the same guarantees

do not hold. In order to extract the hot items, many more queries are needed: instead of making at most two queries per hot item per level, we make at most b queries per hot item per level, and so we need to reduce the probability of making a mistake to reflect this. One solution would be to modify T to give a guarantee—but this can lose the point of the exercise, which is to reduce the cost of each update. So instead we treat this as a heuristic to try out in practice, and to see how well it performs.

A more concrete improvement to space and time bounds comes from observing that it is wasteful to keep sketches for high levels in the hierarchy, since there are very few items to monitor. It is therefore an improvement to keep exact counts for items at high levels in the hierarchy.

5. COMPARISON BETWEEN METHODS AND EXTENSIONS

We have described two methods to find hot items after observing a sequence of insertion and deletion transactions, and proved that they can give guarantees about the quality of their output. These are the first methods to be able to give such guarantees in the presence of deletions, and we now go on to compare these two different approaches. We will also briefly discuss how they can be adapted when the input may come in other formats.

Under the theoretical analysis, it is clear that the adaptive and nonadaptive methods have some features in common. Both make use of universal hash functions to map items to counters where counts are maintained. However, the theoretical bounds on the adaptive search procedure look somewhat weaker than those on the nonadaptive methods. To give a guarantee of not outputting items which are more than ϵ from being hot items, the adaptive group testing depends on $1/\epsilon^2$ in space, whereas nonadaptive testing uses $1/\epsilon$. The update times look quite similar, depending on the product of the number of tests, T, and the bit depth of the universe, $\log_b(m)$. It will be important to see how these methods perform in practice, since these are only worst-case guarantees. In order to compare these methods in concrete terms, we shall use the same values of T and W for adaptive and nonadaptive group testing in our tests, so that both methods are allocated approximately the same amount of space.

Another difference is that adaptive group testing requires many more hash function evaluations to process each transaction compared to nonadaptive group testing. This is because adaptive group testing computes a different hash for each of $\log m$ prefixes of the item, whereas nonadaptive group testing computes one hash function to map the item to a group, and then allocates it to subgroups based on its binary representation. Although the universal hash functions can be implemented quite efficiently [Thorup 2000], this extra processing time can become apparent for high transaction rates.

5.1 Other Update Models

In this work we assume that we modify counts by one each time to model insertions or deletions. But there is no reason to insist on this: the above proofs work for arbitrary count distributions; hence it is possible to allow the counts to be modified by arbitrary increments or decrements, in the same update time

bounds. The counts can even include fractional values if so desired. This holds for both the adaptive and nonadaptive methods. Another feature is that it is straightforward to combine the data structures for the merge of two distributions: providing both data structures were created using the same parameters and hash functions, then summing the counters coordinatewise gives the same set of counts as if the whole distribution had been processed by a single data structure. This should be contrasted to other approaches [Babcock and Olston 2003], which also compute the overall hot items from multiple sources, but keep a large amount of space at each location: instead the focus is on minimizing the amount of communication. Immediate comparison of the approaches is not possible, but for periodic updates (say, every minute) it would be interesting to compare the communication used by the two methods.

6. EXPERIMENTS

6.1 Evaluation

To evaluate our approach, we implemented our group testing algorithms in C. We also implemented two algorithms which operate on nondynamic data, the algorithm Lossy Counting [Manku and Motwani 2002] and Frequent [Demaine et al. 2002]. Neither algorithm is able to cope with the case of the deletion of an item, and there is no obvious modification to accommodate deletions and still guarantee the quality of the output. We instead performed a "best effort" modification: since both algorithms keep counters for certain items, which are incremented when that item is inserted, we modified the algorithms to decrement the counter whenever the corresponding item was deleted. When an item without a counter was deleted, then we took no action.[2] This modification ensures that when the algorithms encounter an inserts-only dataset, then their action is the same as the original algorithms. Code for our implementations is available on the Web, from `http://www.cs.rutgers.edu/~muthu/massdal-code-index.html`.

6.1.1 *Evaluation Criteria.* We ran tests on both synthetic and real data, and measured time and space usage of all four methods. Evaluation was carried out on a 2.4-GHz desktop PC with 512-MB RAM. In order to evaluate the quality of the results, we used two standard measures: the *recall* and the *precision*.

Definition 6.1. The *recall* of an experiment to find hot items is the proportion of the hot items that are found by the method. The *precision* is the proportion of items identified by the algorithm which are hot items.

It will be interesting to see how these properties interact. For example, if an algorithm outputs every item in the range $1 \cdots m$ then it clearly has perfect recall (every hot item is indeed included in the output), but its precision is very poor. At the other extreme, an algorithm which is able to identify only the

[2]Many variations of this theme are possible. Our experimental results here that compare our algorithms to modifications of Lossy Counting [Manku and Motwani 2002] and Frequent [Demaine et al. 2002] should be considered proof-of-concept only.

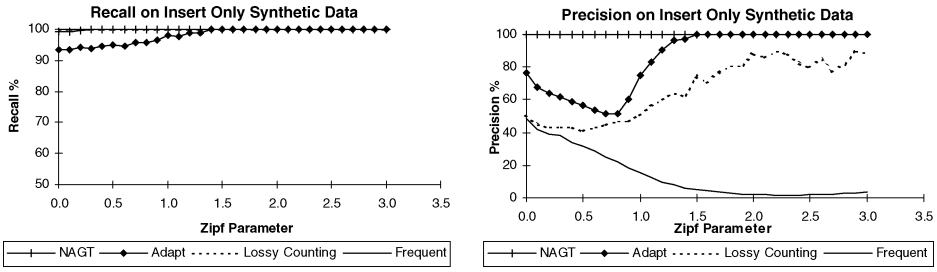

Fig. 5. Experiments on a sequence of 10^7 insertion-only transactions. Left: testing recall (proportion of the hot items reported). Right: testing precision (proportion of the output items which were hot).

most frequent item will have perfect precision, but may have low recall if there are many hot items. For example, the Frequent algorithm gives guarantees on the recall of its output, but does not strongly bound the precision, whereas, for Lossy Counting, the parameter ϵ affects the precision indirectly (depending on the properties of the sequence). Meanwhile, our group testing methods give probabilistic guarantees of perfect recall and good precision.

6.1.2 *Setting of Parameters.* In all our experiments, we set $\epsilon = \frac{1}{k+1}$ and hence set $W = \frac{2}{k+1}$, since this keeps the memory usage quite small. In practice, we found that this setting of ϵ gave quite good results for our group testing methods, and that smaller values of ϵ did not significantly improve the results. In all the experiments, we ran both group testing methods with the same values of W and T, which ensured that on most base experiments they used the same amount of space. In our experiments, we looked at the effect of varying the value of the parameters T and b. We gave the parameter ϵ to each algorithm and saw how much space it used to give a guarantee based on this ϵ. In general, the deterministic methods used less space than the group testing methods. However, when we made additional space available to the deterministic methods equivalent to that used by the group testing approaches, we did not see any significant improvement in their precision and we saw a similar pattern of dependency on the Zipf parameter.

6.2 Insertions-Only Data

Although our methods have been designed for the challenges of transaction sequences that contain a mix of insertions and deletions, we first evaluated a sequence of transactions which contained only insertions. These were generated by a Zipf distribution, whose parameter was varied from 0 (uniform) to 3 (highly skewed). We set $k = 1000$, so we were looking for all items with frequency 0.1% and higher. Throughout, we worked with a universe of size $m = 2^{32}$. Our first observation on the performance of group testing-based methods is that they gave good results with very small values of T. The plots in Figure 5 show the precision and recall of the methods with $T = 2$, meaning that each item was placed in two groups in nonadaptive group testing, and two estimates were computed for each count in adaptive group testing. Nonadaptive group

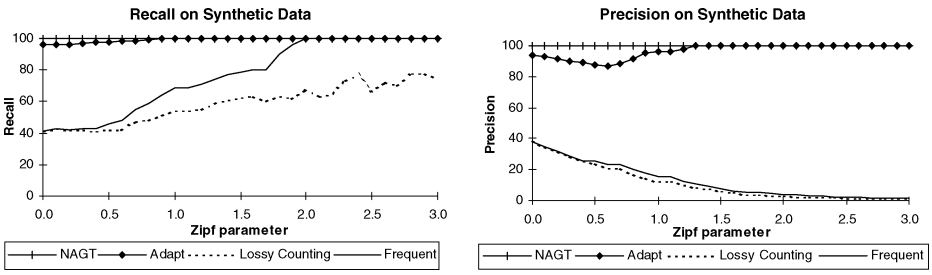

Fig. 6. Experiments on synthetic data consisting of 10^7 transactions.

testing is denoted as algorithm "NAGT," and adaptive group testing as algorithm "Adapt." Note that, on this data set, the algorithms Lossy Counting and Frequent both achieved perfect recall, that is, they returned every hot item. This is not surprising: the deterministic guarantees ensure that they will find all hot items when the data consists of inserts only. Group testing approaches did pretty well here: nonadaptive got almost perfect recall, and adaptive missed only a few for near uniform distributions. On distributions with a small Zipf parameter, many items had counts which were close to the threshold for being a hot item, meaning that adaptive group testing can easily miss an item which is just over the threshold, or include an item which is just below. This is also visible in the precision results: while nonadaptive group testing included no items which were not hot, adaptive group testing did include some. However, the deterministic methods also did quite badly on precision, frequently including many items which were not hot in its output while, for this value of ϵ, Lossy Counting did much better than Frequent, but consistently worse than group testing. As we increased T, both nonadaptive and adaptive group testing got perfect precision and recall on all distributions. For the experiment illustrated, the group testing methods both used about 100 KB of space each, while the deterministic methods used a smaller amount of space (around half as much).

6.3 Synthetic Data with Insertions and Deletions

We created synthetic datasets designed to test the behavior when confronted with a sequence including deletes. The datasets were created in three equal parts: first, a sequence of insertions distributed uniformly over a small range; next, a sequence of inserts drawn from a Zipf distribution with varying parameters; last, a sequence of deletes distributed uniformly over the same range as the starting sequence. The net effect of this sequence should be that the first and last groups of transactions would (mostly) cancel out, leaving the "true" signal from the Zipf distribution. The dataset was designed to test whether the algorithms could find this signal from the added noise. We generated a dataset of 10,000,000 items, so it was possible to compute the exact answers in order to compare, and searched for the $k = 1000$ hot items while varying the Zipf parameter of the signal. The results are shown in Figure 6, with the recall plotted on the left and the precision on the right. Each data point comes from one trial, rather than averaging over multiple repetitions.

The purpose of this experiment was to demonstrate a scenario where insert-only algorithms would not be able to cope when the dataset included many deletes (in this case, one in three of the transactions was a deletion). Lossy Counting performed worst on both recall and precision, while Frequent managed to get good recall only when the signal was very skewed, meaning the hot items had very high frequencies compared to all other items. Even when the recall of the other algorithms was reasonably good (finding around three-quarters of the hot items), their precision was very poor: for every hot item that was reported, around 10 infrequent items were also included in the output, and we could not distinguish between these two types. Meanwhile, both group testing approaches succeeded in finding almost all hot items, and outputting few infrequent items.

There is a price to pay for the extra power of the group testing algorithm: it takes longer to process each item under our implementation, and requires more memory. However, these memory requirements are all very small compared to the size of the dataset: both group testing methods used 187 kB—Lossy Counting allocated 40 kB on average, and Frequent used 136 kB.[3] In a later section, we look at the time and space costs of the group testing methods in more detail.

6.4 Real Data with Insertions and Deletions

We obtained data from one of AT&Ts networks for part of a day, totaling around 100 MB. This consisted of a sequence of new telephone connections being initiated, and subsequently closed. The duration of the connections varied considerably, meaning that at any one time there were huge numbers of connections in place. In total, there were 3.5 million transactions. We ran the algorithms on this dynamic sequence in order to test their ability to operate on naturally occurring sequences. After every 100,000 transactions we posed the query to find all (source, destination) pairs with a current frequency greater than 1%. We were grouping connections by their regional codes, giving many millions of possible pairs, m, although we discovered that geographically neighboring areas generated the most communication. This meant that there were significant numbers of pairings achieving the target frequency. Again, we computed recall and precision for the three algorithms, with the results shown in Figure 7: we set $T = 2$ again and ran nonadaptive group testing (NAGT) and adaptive group testing (Adapt).

The nonadaptive group testing approach is shown to be justified here on real data. In terms of both recall and precision, it is nearly perfect. On one occasion, it overlooked a hot item, and a few times it included items which were not hot. Under certain circumstances this may be acceptable if the items included are "nearly hot," that is, are just under the threshold for being considered hot. However, we did not pursue this line. In the same amount of space, adaptive group testing did almost as well, although its recall and precision were both

[3]These reflected the space allocated for the insert-only algorithms based on upper bounds on the space needed. This was done to avoid complicated and costly memory allocation while processing transactions.

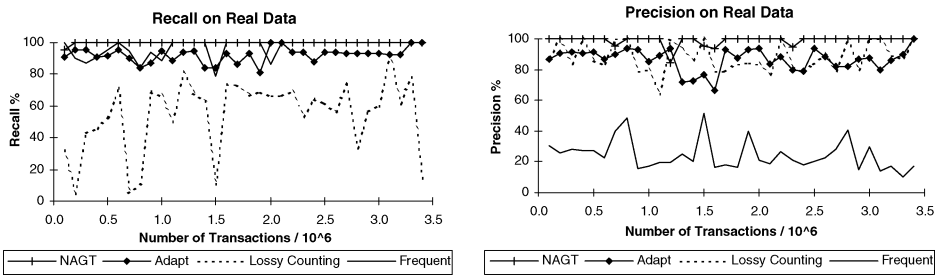

Fig. 7. Performance results on real data.

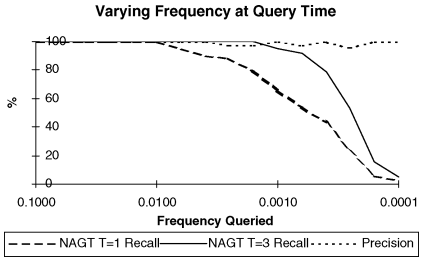

Fig. 8. Choosing the frequency level at query time: the data structure was built for queries at the 0.5% level, but was then tested with queries ranging from 10% to 0.01%.

less good overall than nonadaptive. Both methods reached perfect precision and recall as T was increased: nonadaptive group testing achieved perfect scores for $T = 3$, and adaptive for $T = 7$.

Lossy Counting performed generally poorly on this dynamic dataset, its quality of results swinging wildly between readings but on average finding only half the hot items. The recall of the Frequent algorithm looked reasonably good, especially as time progressed, but its precision, which began poorly, appeared to degrade further. One possible explanation is that the algorithm was collecting all items which were ever hot, and outputting these whether they were hot or not. Certainly, it output between two to three times as many items as were currently hot, meaning that its output necessarily contained many infrequent items.

Next, we ran tests which demonstrated the flexibility of our approach. As noted in Section 3.2, if we create a set of counters for nonadaptive group testing for a particular frequency level $f = 1/(k+1)$, then we can use these counters to answer a query for a higher frequency level without any need for recomputation. To test this, we computed the data structure for the first million items of the real data set based on a frequency level of 0.5%. We then asked for all hot items for a variety of frequencies between 10% and 0.5%. The results are shown in Figure 8. As predicted, the recall level was the same (100% throughout), and precision was high, with a few nonhot items included at various points. We then examined how much below the designed capability we could push the group testing algorithm, and ran queries asking for hot items with progressively lower frequencies. For nonadaptive group testing with $T = 1$, the quality of the recall began deteriorating after the query frequency descended below 0.5%, but

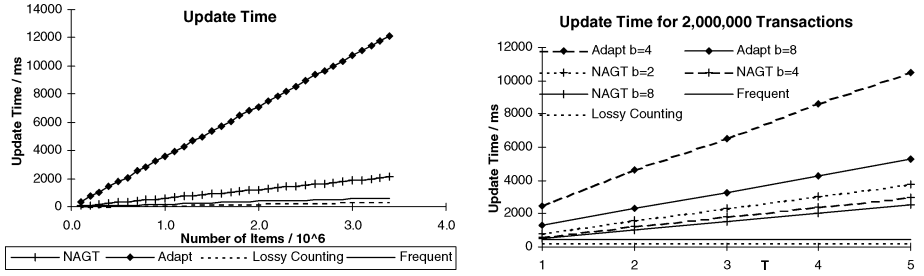

Fig. 9. Timing results on real data.

for $T = 3$ the results maintained an impressive level of recall down to around the 0.05% level, after which the quality deteriorated (around this point, the threshold for being considered a hot item was down to having a count in single figures, due to deletions removing previously inserted items). Throughout, the precision of both sets of results were very high, close to perfect even when used far below the intended range of operation.

6.5 Timing Results

On the real data, we timed how long it took to process transactions, as we varied certain parameters of the methods. We also plotted the time taken by the insert-only methods for comparison. Timing results are shown in Figure 9. On the left are timing results for working through the whole data set. As we would expect, the time scaled roughly linearly with the number of transactions processed. Nonadaptive group testing was a few times slower than for the insertion-only methods, which were very fast. With $T = 2$, nonadaptive group testing processed over a million transactions per second. Adaptive group testing was somewhat slower. Although asymptotically the two methods have the same update cost, here we see the effect of the difference in the methods: since adaptive group testing computes many more hash functions than nonadaptive (see Section 5), the cost of this computation is clear. It is therefore desirable to look at how to reduce the number of hash function computations done by adaptive group testing. Applying the ideas discussed in Sections 3.3 and 4.3, we tried varying the parameter b from 2.

The results for this are shown on the right in Figure 9. Here, we plot the time to process two million transactions for different values of b against T, the number of repetitions of the process. It can be seen that increasing b does indeed bring down the cost of adaptive and nonadaptive group testing. For $T = 1$, nonadaptive group testing becomes competitive with the insertion methods in terms of time to process each transaction. We also measured the output time for each method. The adaptive group testing approach took an average 5 ms per query, while the nonadaptive group testing took 2 ms. The deterministic approaches took less than 1 ms per query.

6.6 Time-Space Tradeoffs

To see in more detail the effect of varying b, we plotted the time to process two million transactions for eight different values of b (2, 4, 8, 16, 32, 64, 128, and

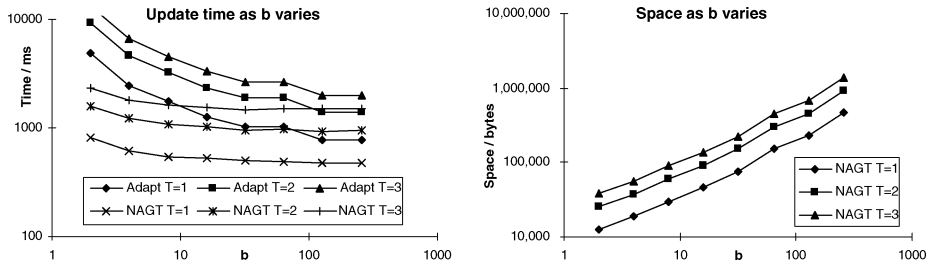

Fig. 10. Time and space costs of varying b.

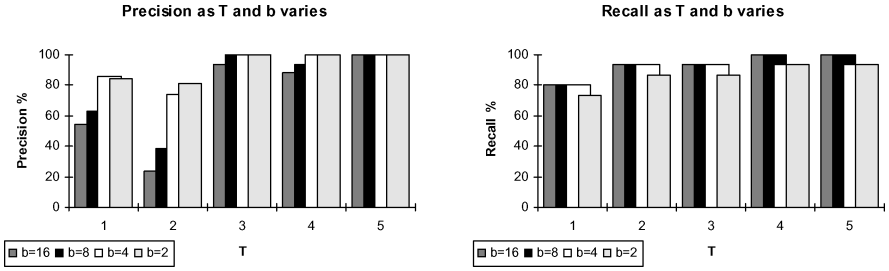

Fig. 11. Precision and recall on real data as b and T vary.

256) and three values of T (1, 2, 3) at $k = 100$. The results are shown in Figure 10. Although increasing b does improve the update time for every method, the effect becomes much less pronounced for larger values of b, suggesting that the most benefit is to be had for small values of b. The benefit seems strongest for adaptive group testing, which has the most to gain. Nonadaptive group testing still computes T functions per item, so eventually the benefit of larger b is insignificant compared to this fixed cost.

For nonadaptive group testing, the space must increase as b increases. We plotted this on the right in Figure 10. It can be seen that the space increases quite significantly for large values of b, as predicted. For $b = 2$ and $T = 1$, the space used is about 12 kB, while for $b = 256$, the space has increased to 460 kB. For $T = 2$ and $T = 3$, the space used is twice and three times this, respectively.

It is important to see the effect of this tradeoff on accuracy as well. For nonadaptive group testing, the precision and recall remained the same (100% for both) as b and T were varied. For adaptive group testing, we kept the space fixed and looked at how the accuracy varied for different values of T. The results are given in Figure 11. It can be seen that there is little variation in the recall with b, but it increases slightly with T, as we would expect. For precision, the difference is more pronounced. For small values of T, increasing b to speed up processing has an immediate effect on the precision: more items which are not hot are included in the output as b increases. For larger values of T, this effect is reduced: increasing b does not affect precision by as much. Note that the transaction processing time is proportional to $T/\log(b)$, so it seems that good tradeoffs are achieved for $T = 1$ and $b = 4$ and for $T = 3$ and $b = 8$ or 16. Looking at Figure 10, we see that these points achieve similar update times, of approximately one million items per second in our experiments.

7. CONCLUSIONS

We have proposed two new methods for identifying hot items which occur more than some frequency threshold. These are the first methods which can cope with dynamic datasets, that is, the removal as well as the addition of items. They perform to a high degree of accuracy in practice, as guaranteed by our analysis of the algorithm, and are quite simple to implement. In our experimental analysis, it seemed that an approach based on nonadaptive group testing was slightly preferable to one based on adaptive group testing, in terms of recall, precision, and time.

Recently, we have taken these ideas of using group testing techniques to identify items of interest in small space, and applied them to other problems. For example, consider finding items which have the biggest frequency difference between two datasets. Using a similar arrangement of groups but a different test allows us to find such items while processing transactions at very high rates and keeping only small summaries for each dataset [Cormode and Muthukrishnan 2004b]. This is of interest in a number of scenarios, such as trend analysis, financial datasets, and anomaly detection [Yi et al. 2000]. One point of interest is that, for that scenario, it is straightforward to generalize the nonadaptive group testing approach, but the adaptive group testing approach cannot be applied so easily.

Our approach of group testing may have application to other problems, notably in designing summary data structures for the maintenance of other statistics of interest and in data stream applications. An interesting open problem is to find combinatorial designs which can achieve the same properties as our randomly chosen groups, in order to give a fully deterministic construction for maintaining hot items. The main challenge here is to find good "decoding" methods: given the result of testing various groups, how to determine what the hot items are. We need such methods that work quickly in small space.

A significant problem that we have not approached here is that of continuously monitoring the hot items—that is, to maintain a list of all items that are hot, and keep this updated as transactions are observed. A simple solution is to keep the same data structure, and to run the query procedure when needed, say once every second, or whenever n has changed by more than k. (After an item is inserted, it is easy to check whether it is now a hot item. Following deletions, other items can become hot, but the threshold of $n/(k+1)$ only changes when n has decreased by $k+1$.) In our experiments, the cost of running queries is a matter of milliseconds and so is quite a cheap operation to perform. In some situations this is sufficient, but a more general solution is needed for the full version of this problem.

ACKNOWLEDGMENTS

We thank the anonymous referees for many helpful suggestions.

REFERENCES

AHO, A. V., HOPCROFT, J. E., AND ULLMAN, J. D. 1987. *Data structures and algorithms*. Addison-Wesley, Reading, MA.

ALON, N., GIBBONS, P., MATIAS, Y., AND SZEGEDY, M. 1999. Tracking join and self-join sizes in limited storage. In *Proceedings of the Eighteenth ACM Symposium on Principles of Database Systems*. 10–20.

ALON, N., MATIAS, Y., AND SZEGEDY, M. 1996. The space complexity of approximating the frequency moments. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing*. 20–29. Journal version in *J. Comput. Syst. Sci.*, *58*, 137–147, 1999.

BABCOCK, B. AND OLSTON, C. 2003. Distributed top-*k* monitoring. In *Proceedings of ACM SIGMOD International Conference on Management of Data*.

BARBARA, D., WU, N., AND JAJODIA, S. 2001. Detecting novel network intrusions using Bayes estimators. In *Proceedings of the First SIAM International Conference on Data Mining*.

BOYER, B. AND MOORE, J. 1982. A fast majority vote algorithm. Tech. Rep. 35. Institute for Computer Science, University of Texas, at Austin, Austin, TX.

CARTER, J. L. AND WEGMAN, M. N. 1979. Universal classes of hash functions. *J. Comput. Syst. Sci. 18*, 2, 143–154.

CHARIKAR, M., CHEN, K., AND FARACH-COLTON, M. 2002. Finding frequent items in data streams. In *Procedings of the International Colloquium on Automata, Languages and Programming* (ICALP). 693–703.

CORMODE, G. AND MUTHUKRISHNAN, S. 2003. What's hot and what's not: Tracking most frequent items dynamically. In *Proceedings of ACM Conference on Principles of Database Systems*. 296–306.

CORMODE, G. AND MUTHUKRISHNAN, S. 2004a. An improved data stream summary: The count-min sketch and its applications. *J. Algorithms*. In press.

CORMODE, G. AND MUTHUKRISHNAN, S. 2004b. What's new: Finding significant differences in network data streams. In *Proceedings of IEEE Infocom*.

DEMAINE, E., LÓPEZ-ORTIZ, A., AND MUNRO, J. I. 2002. Frequency estimation of Internet packet streams with limited space. In *Proceedings of the 10th Annual European Symposium on Algorithms*. Lecture Notes in Computer Science, vol. 2461. Springer, Berlin, Germany, 348–360.

DU, D.-Z. AND HWANG, F. 1993. *Combinatorial Group Testing and Its Applications*. Series on Applied Mathematics, vol. 3. World Scientific, Singapore.

ESTAN, C. AND VARGHESE, G. 2002. New directions in traffic measurement and accounting. In *Proceedings of ACM SIGCOMM. Journal version in Comput. Commun. Rev. 32*, 4, 323–338.

FANG, M., SHIVAKUMAR, N., GARCIA-MOLINA, H., MOTWANI, R., AND ULLMAN, J. D. 1998. Computing iceberg queries efficiently. In *Proceedings of the International Conference on Very Large Data Bases*. 299–310.

FISCHER, M. AND SALZBERG, S. 1982. Finding a majority among *n* votes: Solution to problem 81-5. *J. Algorith. 3*, 4, 376–379.

GAROFALAKIS, M., GEHRKE, J., AND RASTOGI, R. 2002. Querying and mining data streams: You only get one look. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

GIBBONS, P. AND MATIAS, Y. 1998. New sampling-based summary statistics for improving approximate query answers. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, *Journal version in ACM SIGMOD Rec. 27*, 331–342.

GIBBONS, P. AND MATIAS, Y. 1999. Synopsis structures for massive data sets. DIMACS Series in Discrete Mathematics and Theoretical Computer Science A.

GIBBONS, P. B., MATIAS, Y., AND POOSALA, V. 1997. Fast incremental maintenance of approximate histograms. In *Proceedings of the International Conference on Very Large Data Bases*. 466–475.

GILBERT, A., GUHA, S., INDYK, P., KOTIDIS, Y., MUTHUKRISHNAN, S., AND STRAUSS, M. 2002a. Fast, small-space algorithms for approximate histogram maintenance. In *Proceedings of the 34th ACM Symposium on the Theory of Computing*. 389–398.

GILBERT, A., KOTIDIS, Y., MUTHUKRISHNAN, S., AND STRAUSS, M. 2001. QuickSAND: Quick summary and analysis of network data. DIMACS Tech. Rep. 2001–43, Available online at http://dimacs.crutgers.edu/Techniclts.

GILBERT, A. C., KOTIDIS, Y., MUTHUKRISHNAN, S., AND STRAUSS, M. 2002b. How to summarize the universe: Dynamic maintenance of quantiles. In *Proceedings of the International Conference on Very Large Data Bases*. 454–465.

IOANNIDIS, Y. E. AND CHRISTODOULAKIS, S. 1993. Optimal histograms for limiting worst-case error propagation in the size of the join radius. *ACM Trans. Database Syst. 18*, 4, 709–748.

IOANNIDIS, Y. E. AND POOSALA, V. 1995. Balancing histogram optimality and practicality for query result size estimation. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*. 233–244.

KARP, R., PAPADIMITRIOU, C., AND SHENKER, S. 2003. A simple algorithm for finding frequent elements in sets and bags. *ACM Trans. Database Syst. 28*, 51–55.

KUSHILEVITZ, E. AND NISAN, N. 1997. *Communication Complexity*. Cambridge University Press, Cambridge, U.K.

MANKU, G. AND MOTWANI, R. 2002. Approximate frequency counts over data streams. In *Proceedings of the International Conference on Very Large Data Bases*. 346–357.

MISRA, J. AND GRIES, D. 1982. Finding repeated elements. *Sci. Comput. Programm. 2*, 143–152.

MOTWANI, R. AND RAGHAVAN, P. 1995. *Randomized Algorithms*. Cambridge University Press, Cambridge, U.K.

MUTHUKRISHNAN, S. 2003. Data streams: Algorithms and applications. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms*. Available online at `http://athos.rutgers.edu/~muthu/stream-1-1.ps`.

THORUP, M. 2000. Even strongly universal hashing is pretty fast. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms*. 496–497.

YI, B.-K., SIDIROPOULOS, N., JOHNSON, T., JAGADISH, H., FALOUTSOS, C., AND BILIRIS, A. 2000. Online data mining for co-evolving time sequences. In *Proceedings of the 16th International Conference on Data Engineering* (ICDE' 00). 13–22.

Received October 2003; revised June 2004; accepted September 2004