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

Graham Cormode
Center for Discrete Mathematics
and Computer Science (DIMACS)
Rutgers University, Piscataway NJ
graham@dimacs.rutgers.edu

S. Muthukrishnan
Department of Computer Science
Rutgers University, Piscataway NJ
muthu@cs.rutgers.edu

ABSTRACT

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 networking applications.

We present a new algorithm for dynamically determining the hot items at any time in the relation that is undergoing deletion operations as well as inserts. Our algorithm maintains a small space data structure that monitors the transactions on the relation, and when required, quickly outputs all hot items, without rescanning the relation in the database. With user-specified probability, it is able to report all hot items. Our algorithm relies on the idea of "group testing", is simple to implement, and has provable quality, space and time guarantees. Previously known algorithms for this problem that make similar quality and performance guarantees can not handle deletions, and those that handle deletions can not make similar guarantees without rescanning the database. Our experiments with real and synthetic data shows that our algorithm is remarkably accurate in dynamically tracking the hot items independent of the rate of insertions and deletions.

1. INTRODUCTION

One of the most basic statistics on a database relation is that of which items are *hot*, i.e., occur frequently.[1] This gives a useful measure of the skew of the data. High-biased and end-biased histograms [19, 20] specifically focus on hot items to summarize data distributions for selectivity estimation. *Iceberg queries* generalize the notion of hot items in the 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 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 context of anomaly detection [5, 9, 17, 21]. Our focus in this paper 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 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 and thoughtful discussion in [14] describes many applications for finding hot items and the challenges in maintaining them over a changing database relation. Also, [11] presents 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. The net occurrence of any item i at time t, denoted $n_i(t)$, 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 $f_i(t) = n_i(t)/\sum_{j=1}^{m} n_j(t)$. The most frequent item at time t is the one with $f_i(t) = \max_i f_i(t)$. The k most frequent items at time t are those with the k largest $f_i(t)$'s. We are interested in the related notion of frequent items that we call hot items. An item i is said to be a *hot item* if $f_i(t) > 1/(k+1)$, that is, it appears a significant fraction of the entire dataset; here

[1]They are also known as popular items, but we use "hot" to emphasize the notion of "current interest", since hot items may change from time to time.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PODS 2003, June 9-12, 2003, San Diego, CA.
Copyright 2003 ACM 1-58113-670-6/03/06 ...$5.00.

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 $n_i(t)$ for every item is non-negative (the number of deletions never exceeds the number of insertions).

Our main result is a highly efficient, randomized algorithm for maintaining hot items. This algorithm monitors the changes to the data distribution and maintains $O(k \log k \log m)$ space summary data structure. 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.[2] Thus our result here maintains a small summary data structure — significantly sublinear in the dataset size — and accurately detects hot items at any time in 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 can not 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 presence of deletes, in particular, in cases where the data distribution changes significantly over time, samples can not be maintained without rescanning the database relation.[3] Our 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 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 rest of the paper is organized as follows. In Section 2, we summarize previous work, which is rather extensive. In Section 3, we present our algorithms and prove our guarantees. In Section 4, we present a preliminary experimental study of our algorithm using synthetic data as well as real network data addressing the application domain cited earlier and show that our algorithm is effective and practical.

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 [1]. 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 paper [3], the authors proved that estimating $f^*(t) = \max_i f_i(t)$ 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 [3] 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 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 $S \subseteq \{1 \dots m\}$. Transform S into a stream of $n = |S|$ items by including i in the stream exactly once if and only if $i \in S$. Now process this stream with the proposed algorithm. We can then use the algorithm to extract whether $i \in S$ or not: for some i, insert $\lfloor n/k \rfloor$ copies of i. Suppose $i \notin S$, then the frequency of i 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 i will not be output. On the other hand, if $i \in S$ then $\lfloor n/k \rfloor + 1/(n + \lfloor n/k \rfloor) > (n/k)/(n+n/k) = 1/(k+1)$ and so i will be output. Hence, we can extract the set S, and so the space stored must be $\Omega(m)$. $\square$

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. To overcome this disadvantage, then in certain cases we may adopt a restriction based on the *small tail* assumption, to help with the analysis of the approach. Say $f_1 \geq \dots \geq f_m$ is the frequencies of items at any time in the *non-increasing* order. A set of frequencies is said to have a *small tail* if $(\sum_{i>k} f_i) \leq 1/(k+1)$, that is, the items except the top k do not amount to a significant count. If there are k hot items, then surely the small tail probability holds. If small tail probability holds then it is still possible some of the top k items are not hot. We shall analyze our solution in the presence and absence of this small tail property. We will later see that this condition is overly strong, and that in fact only a weaker condition is needed to guarantee no infrequent items are output.

2.1 Prior Work

Finding which items are the hot is a problem that has a history stretching back over two decades. The earliest work considered the problem of finding an item which occurred more than half of the time [6, 12]. 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 [24]. The total time to process n items is $O(n \log k)$, with space $O(k)$ (we assume throughout that any item label or counter can be stored in constant space). In their 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

[2] More formally, for any user specified probability δ, the algorithm succeeds with probability at least $1-\delta$, as is standard in randomized algorithms.

[3] For example, the entire set of sampled values may get erased from the relation by a sequence of deletes.

Algorithm	Type	Time Per Item	Space	Reference
Lossy Counting	Deterministic	$O(\log(n/k))$ amortized	$\Omega(k \log(n/k))$	[23]
Misra-Gries	Deterministic	$O(\log k)$ amortized	$O(k)$	[24]
Frequent	Randomized (LV)	$O(1)$ expected	$O(k)$	[9, 21]
Charikar *et al*	Approximate, Randomized (MC)	$\Omega(k/\epsilon^2 \log n)$	$\Omega(k/\epsilon^2 \log n)$	[8]

Table 1: Summary of previous results. LV (Los Vegas) and MC (Monte Carlo) are types of randomized algorithms. See [25] for details.

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 amongst 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 kind of problem. Several authors [9, 21] have rediscovered the algorithm of Misra and Gries, and using more sophisticated data structures they are 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 is used by Manku and Motwani [23] 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 $O(\frac{1}{\epsilon} \log \epsilon n)$ — 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 is that of Babcock and Olston [4]. This is not immediately comparable to our work, since their focus is on maintaining the top-k items in a distributed environment, and the goal is to minimize communication. Counts of all items are maintained exactly at each location, so the memory space is $\Omega(m)$. All of these mentioned algorithms are deterministic in their operation: the output is solely a function of the input stream and the parameter k. We summarize the prior work in Table 1.

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 a 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 to that reached without the insertions and deletions of the item.

Previous work that studied hot items in presence of both of inserts and deletes is sparse [13, 14]. These papers propose methods to maintain a sample and count of times the sample appears in the data set. These methods work provably for insert-only case, but provide no guarantees for the fully dynamic case with deletions. However, the authors study how effective these samples are for the deletion case through calculations and experiments. [15] presents methods to maintain various histograms in presence of inserts and deletes using "backing sample", but these methods too need access to large portion of the data periodically in presence of deletes. A recent theoretical work presented provable algorithms for maintaining histograms with guaranteed accuracy and small space [16]. The methods in this paper may 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 paper — maintaining the sum of items in various groups may be thought of as the "sketching" developed in [16], but our overall methods are much simpler and more efficient. Finally, recent work in maintaining quantiles [18] is similar to ours since it keeps the sum of items in random subsets. However, the result here is, of necessity, more involved, involving a random group generation phase based on group testing which is not needed in [18]. Also, once such groups are generated, we maintain sums of deterministic sets (in contrast to the random sets as in [18]) given again by error correcting codes. Finally, our algorithm is more efficient than the $\Omega(k^2 \log^2 m)$ space and time algorithms therein.

2.2 Sketch based methods

An important recent result is that of Charikar *et al* [8], who gave an algorithm to find k items whose frequency is 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 $O(k/\epsilon^2 \log n/\delta)$. The method works as follows: a randomized "sketch" structure is constructed and updated which guarantees with high probability that the count of any item can be approximated up to an additive quantity of γ, where γ is a function of the frequencies of the infrequent items. As each item is encountered, the structure is updated, and the approximate frequency of the item is extracted. A heap of frequent items is kept, and if the current items exceeds the threshold, then the least frequent item in the heap is ejected, and the current item inserted. It is possible to amend this method so that the approximate counts can reflect item deletions. However, it is important to realize that this does *not* allow the current hot items to be found with this method: suppose that an item which is currently frequent is subject to a number of deletions so that it is not longer among the most frequent. In this case, it is not possible using the method of [8] to retrieve items from the past which have consequently *become* frequent.

We will now propose a novel algorithm which makes use of approximate methods such as those in [8], and which can

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

Figure 1: Divide and conquer algorithm to find hot items using a dyadic range sum oracle

handle insertions and deletions. We shall analyze its space requirements in detail, and show that these are significantly larger than those of the algorithm which we propose here. We shall make use of the fact that the range of items can be mapped onto the integers $1 \dots m$. We will initially describe the method in terms of an oracle, and then show how this oracle can be realized in a variety of ways.

Definition 1. A *dyadic range sum oracle* returns the (approximate) sum of the counts of items in the range $(i2^j + 1) \dots (i+1)2^j$ 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 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 two. 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. The procedure is presented in Figure 1.

Theorem 1. *Calling* DivideAndConquer$(1, m, n/(k+1))$ *will output all and only hot items. A total of* $O(k \log m/k)$ *calls will be made to the oracle.*

Proof. The procedure rejects dyadic intervals whose range sum is less than the threshold, $n/(k+1)$. These cannot contain a hot item, since negative counts are not possible. For each length, there can be at most k dyadic regions with weight greater than $n/(k+1)$, since the total weight must sum to n. Because there are $\log m$ levels, this bounds the number of calls to the oracle as $O(k \log m)$. A tighter analysis improves this to $O(k \log m/k)$. □

Recent work has given several methods of implementing the dyadic range sum oracle approximately, which can be updated to reflect the arrival or departure of any item. We now list three examples of these:

1. The technique of [2], as refined in [18], allows arbitrary range sums to be computed for a range of length l accurate up to an additive factor of $\epsilon n \sqrt{l}$ with probability at least $1-\delta$ using space $O(1/\epsilon^2 \log 1/\delta)$. Applying this here, we are interested in ranges up to length m/k (above this length, we can store the $O(k)$ range sums exactly), and we will want to set $\epsilon n \sqrt{l}$ to be less than $n/(k+1)$. This will require us to make $\epsilon = O(1/\sqrt{kn})$, and consequently the space required will be $\Omega(km)$.

2. The method of Random Subset Sums described in [18] allows finding the frequency of a single item, up to additive error of ϵn using $O(1/\epsilon^2 \log 1/\delta)$ space. It succeeds with probability at least $1-\delta$. This can be transformed into a method for finding the approximate dyadic range sum by keeping $\log m$ independent copies of the procedure, one for each length 2^i. When a new item arrives or departs, each of the $\log m$ copies is updated, for each of the $\log m$ dyadic ranges which the item falls in. When setting ϵ, a trivial bound is that $\epsilon \leq \gamma/(k+1)$ for some small constant γ (say, $\gamma = 1 - 10\%$). Overall, this transformation requires a total of $O(k^2/\gamma^2 \log m \log k/\delta)$ space. This guarantees with probability at least $1-\delta$ that we output all items with frequency at least $(1+\gamma)/(k+1)$, and we output no items with frequency less than $(1-\gamma)/(k+1)$.

3. The method of Charikar *et al* builds a structure which allows the approximation of the count of any item correct up to an additive quantity of $\gamma n/(k+1)$ for q queries in space $O(\sum_{i=k+1}^{m} n_i(t)^2/(\gamma n/(k+1))^2 \log q/\delta)$. In our situation, we know that $n_i(t) < n/(k+1)$, so this quantity is $O(k/\gamma^2 \log q/\delta)$. We use the same observation as above to reduce dyadic range sum queries to point queries in $\log m$ levels. The overall space requirement is $O(k/\gamma^2 \log m \log k/\delta)$, since we make $q \leq 2k$ queries at each of $\log m$ levels. The big-O notation here hides large constant multiples of several hundred, according to [8]. This guarantees with probability at least $1-\delta$ that we output all items with frequency at least $(1+\gamma)/(k+1)$, and we output no items with frequency less than $(1-\gamma)/(k+1)$.

The most promising of these approaches is based on the structure of [8], which has space cost of $O(k/\gamma^2 \log m \log k/\delta)$. The big-O notation hides large constant factors and makes this appear competitive with our approach, so we give an example showing the difference. For $k = 100, \gamma = 10\%, m = 2^{32}$ and $\delta = 0.1$, this method takes approximately 800Mb (storing counts exactly would take several gigabytes). We will propose a method using a simple randomized construction, which has a similar asymptotic memory cost. However, the constant factors are much smaller (we shall make these explicit in our description of the algorithm). For the same parameters as above, the method we propose requires under 128Kb.

2.3 Our Approach

We propose a new approach to this problem, based on ideas from group testing and error correcting codes. We do not keep counters for any individual item but rather for subsets of items, and the pattern of which items are monitored is fixed in advance, and so does not vary whatever the input distribution. As a consequence, it can find hot items given an input which is a sequence of insertions and deletions with only a small amount of storage.

3. OUR ALGORITHMS

Our algorithms depend on ideas drawn from Group Testing [10]. The idea of (non-adaptive) Group Testing is to design 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". This maps neatly onto our goal of finding up to k hot items. 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

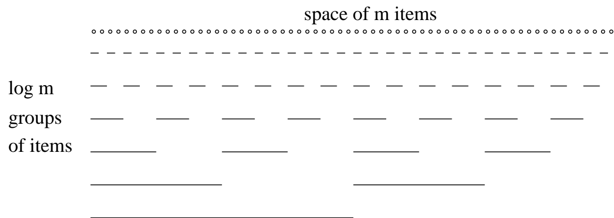

Figure 2: Each test includes half of the range $[1 \ldots m]$, corresponding to the binary representation of values

good coins is an example of group testing. The goal is to minimize the number of tests, where each test in group testing consists of a subset of the items.

Our general procedure is as follows: for each transaction on an item i, we determine which subsets it is included in (denote these $S(i)$). Each subset is associated with a counter, and for an insertion, we increment the counter for all $S(i)$; for a deletion, we correspondingly decrement these counters. The test will be whether the count for a subset exceeds a certain threshold: under certain restrictions, this will inform us that there is 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 subsets required (2) Finding a concise representation of the subsets (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.

3.1 Deterministic algorithm for maintaining 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 [6], and Salzberg and Fischer [12]), 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 merely keeps track of $n(t) = \sum_i n_i(t)$ 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 labeled 1 up to $\lceil \log_2 m \rceil$. We make use of the function $bit(i, j)$, which reports the value of the jth bit of the binary representation of the integer i; and $gt(i, j)$, which returns 1 if $i > j$ and 0 otherwise. Our procedures are as follows:

Insertion of item i: Increment each counter c_j such that $bit(i, j) = 1$ in time $O(\log m)$.
Deletion of i: Decrement each counter c_j such that $bit(i, j) = 1$ in time $O(\log m)$.
Query: If there is a majority, then it is given by $\sum_{j=1}^{\log_2 m} 2^j gt(c_j, c/2)$, computed in time $O(\log m)$.

The two procedures of this method — one to process updates, another to identify the majority element — are given in Figure 3. The arrangement of the counters is shown graphically in Figure 2.

THEOREM 2. *The Algorithm in Figure 3 finds a majority item if there is one with time $O(\log m)$ per operation.*

PROOF. We make two observations: firstly, that the state of the data structure is equivalent to that following a sequence of c insertions only, and secondly 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 i 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 $I - D = c$ insertions only.

The second part relies on the fact that if there is an item whose count is greater than $c/2$ (that is, it is in the majority), then for any way of dividing the elements into two sets, then the set containing the majority element will have weight greater than $c/2$, and the other will have weight less than $c/2$. 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. □

The simple structure of the tests is standard in group testing, and also resembles the structure of the Hamming single error-correcting code. This is no coincidence, and we shall now see the relation between error correcting codes and the style of group testing that we wish to apply.

3.2 Intuition: Algorithms using error-correcting codes

When we perform a test based on 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 amongst m locations requires $\log_2 \binom{m}{k} \geq k \log(m/k)$ bits.

One way to generate the groups is to use ideas from error correcting codes. Consider a code with m codewords each of length l. We will keep l counters, each corresponding to one bit of the codeword. Any codeword consists of a sequence of 0s and 1s. A 1 in location j in the ith codeword tells us that when item i is encountered we should increment counter j for an insertion, and decrement if for a deletion. *Superimposed codes* of order k have the property that the sum (bitwise-or) of any k codewords is distinct from any other such sum [22]. From a sum, it is possible to uniquely identify which codewords contributed to the sum. Deterministic constructions of superimposed codes are known, based on Reed-Solomon codes, which have length $l = k^2 \log^2 m$. However, while the construction of any individual codeword may be made efficiently in small space, we do not know of any construction of superimposed codes that allows for the efficient decoding of the superimposition of several codewords. Existing methods rely on an exhaustive comparison with all m codewords, something that is not practical in our model, where m can be very large. We leave the construction of efficient deterministic solutions as an open problem, and instead consider

```
UpdateCounters(i, transtype, c[0 ... log m])
c[0] ← total + diff
for j = 1 to log m do
  if (transtype = insertion) then
    c[j] ← c[j] + bit(j, i)
  else
    c[j] ← c[j] − bit(j, i)
```

```
FindMajority(c[0 ... log m])
position = 0; t = 1;
for j = 1 to log m do
  if (c[j] ≥ c[0]/2) then
    position ← position + t
  t ← 2 * t
return(position)
```

Figure 3: Algorithm to find the majority element in a sequence of updates

a class of randomized constructions of subsets which, with arbitrary probability, have the desired properties.

3.3 Randomized constructions for finding hot items

We make the following observation: suppose we selected a subset of items to monitor which happened to contain exactly one hot item. Then we could apply the algorithm of Section 3.1 to this subset (splitting it into a further $\log m$ subsets) and, by keeping $\log m$ counters, identify which item was the hot one. We simply have to "weigh" each bucket, and (with some caveats), the hot item is always in the heavier of the two buckets. This makes use of the algorithm for finding a single majority item described at the start of this section. In this section, we will show that a choice of enough subsets at random will allow us to apply this procedure, and that these subsets can be described in a way that will require a small amount of space.

Definition 2. Let $F \subseteq [1 \dots m]$ denote the set of hot items. We have that $|F| \leq k$. We define a *good subset* as a set $S \subseteq [1 \dots m]$ such that $|S \cap F| = 1$.

Theorem 3. *Picking $O(k \ln k)$ subsets by drawing (m/k) items uniformly at random from the items $[1 \dots m]$ means that with constant probability we have included k subsets $S_1 \dots S_k$ such that each S_i is a good subset, and $\cup_i (F \cap S_i) = F$.*

Proof. If there are k hot items and we pick any item from the space of m items then the probability of picking a member of F is k/m. If we repeat this procedure m/k times, then the probability of picking exactly 1 member of F is $p = \frac{m}{k} \cdot \frac{k}{m}(1 - \frac{k}{m})^{\frac{m}{k}-1} = \frac{m}{m-k}(1 - \frac{k}{m})^{\frac{m}{k}}$. For all non-trivial cases, then $1 \leq k \leq m/2$ and so $\frac{1}{4} \leq p \leq \frac{2}{e} \leq \frac{3}{4}$ — in other words, the probability of a randomly chosen set being "good" like this is a constant. We now have the coupon collector's problem [25]: how many sets do we need to collect in order to guarantee with constant probability that we have a set that is good for *each* hot item? Picking $O(k \ln k)$ sets gives us this result. □

We do not wish to pick the sets and store them explicitly: this would consume $\Omega(k \log k \log \binom{m}{m/k}) = \Omega(m \log^2 k)$ space, which is much too large (depending superlinearly on m). Instead, we choose the sets in a pseudo-random fashion using universal hash functions derived from those given by Carter and Wegman [7]. We define a family of hash functions $h_{a,b}$ as follows: fix a prime $P > 2k$, and draw a and b uniformly at random in the range $[0..P-1]$. Then set $h_{a,b}(x) = ((ax + b \mod P) \mod 2k)$. We use members of this family of functions to define our sets: $S_{a,b,i} = \{x | h_{a,b}(x) = i\}$.

Fact 1. *Proposition 7 of [7] Over all choices of a and b, for $x \neq y$, $\Pr(h_{a,b}(x) = h_{a,b}(y)) \leq 1/2k$*

This means that for any pair of items, the probability that they are both in the same set is at most $1/2k$. We will choose $T = O \log k/\delta$ values of a and b. which has the effect of creating $2Tk = 2k \log k/\delta$ sets — each pair a, b defines $2k$ sets. Our procedure in processing an input item i is to determine which T sets it is a member of, and for these to update $\log m$ counters based on the bit representation of i in exactly the same way as the algorithm for finding a majority element.

Lemma 2. *The probability of each hot item being in at least one good set is at least $1 - \delta$.*

Proof. Consider each hot item in turn, remembering that there are at most k of these. For each of T independent repetitions, we choose to put it in one of $2k$ buckets. By the pairwise independence, we expect the total frequency of other items which land in the same bucket as item j to be $\sum_{i \neq j} f_i/2k < 1/2(k+1)$. 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 set, 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 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 $1/2$, since this is twice the expectation. So the probability that we fail on every one of the T independent tests is less than $1/2^{\log k/\delta} = \delta/k$. 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 in at least one good set with probability $1 - \delta$. □

The space required to represent the set is that necessary to store 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 set is $O(\log m)$ bits. The total space requirements is then $k \log k/\delta(1 + \lceil \log m \rceil)$ counters, plus $\log k/\delta$ values of a and b. The main component of this cost is the counters, which can typically be represented as a machine word, so the space required is close to $2k \log k/\delta \log 2m$ words.

We also maintain a count, c, of the number of active items, which is maintained by adding one for every insertion, and subtracting one for every deletion. The idea is that we can use these counters to identify any hot items in a subset at query time, since such an item will be the majority if the set is a good subset. If the subset is not good, then we can

```
ProcessItem(i, transtype, T, k)
c = 0
Initialize c[0...2Tk][0...log m] = 0
Draw a[1...T], b[1...T] from 0...P-1
for all (i, transtype) do
  if (transtype = insertion) then
    c ← c + 1
  else
    c ← c - 1
  for x = 1 to T do
    index = 2(x-1) + (i * a[x] + b[x] mod P mod k)
    UpdateCounters(i, transtype, c[index])
```

```
GroupTest(T, k, φ)
for i = 1 to T do
  if c[i][0] > cφ then
    position = 0; t = 1;
    for j = 1 to log m do
      if (c[i][j] > cφ and
      (c[i][0] - c[i][j] > cφ) then
        Skip to next value of i
      if (c[i][j] ≥ cφ) then
        position ← position + t
      t ← 2 * t
    output(position)
```

Figure 4: Procedures for finding hot items using Group Testing

detect this and discard the set from our calculations when locating hot items, as the following Lemma shows.

Lemma 3. *Given a subset $S_{a,b,i}$, and its associated set of counters $c_0 \ldots c_{\log m}$, it is possible to detect deterministically whether $S_{a,b,i}$ is a good subset, given the small tail property.*

Proof. There are two ways that a subset may fail to be good: it might have no hot items, or it might have more than one hot item. We can detect both cases. (1) If there are no hot items, then the count for the subset will be less than $c/(k+1)$, due to the small tail property. (2) If there are two or more hot items in the set, then there will be a division of the set into two where both halves have a hot item in. Hence (regardless of the small tail property) there will be some j for which $c_j > c/(k+1)$ and $c_0 - c_j > c/(k+1)$. If either condition is met, then we know for sure that the test is bad and can reject it. □

The full algorithms are illustrated in Figure 4. On receiving each item i and the information about whether this is an insertion or deletion, we pass this to the ProcessItem routine. At any point we can search for hot items by calling the GroupTest procedure with parameters $(T, k, 1/(k+1))$.

Theorem 4. *With probability at least $1-\delta$, calling the* GroupTest$(\log k/\delta, k, 1/(k+1))$ *procedure finds all the hot items using $O(k(\log k + \log 1/\delta))$ space. The time for an update is $O(\log k/\delta \log m)$ and the time to list all hot items is $O(k \log k/\delta \log m)$.*

Proof. By Lemma 2, we know that we will have sufficient good sets to cover the k hot items with arbitrary probability, and by Theorem 2, we know that we can find a hot item in a set if there is one, since it must be in the majority for that set. To process an item, we compute T hash functions, and update $T \log m$ counters, giving the time cost. □

Corollary 1. *If we have the small tail property, then we will output no items which are not hot.*

This corollary follows by observing that, using Lemma 3, we know that we can detect whether a set is good or not. So if we have the small tail property then anything we output must be a hot item. □

This is an important feature of our method: without any mention of the small tail property, we are guaranteed of including every hot item in the output with arbitrarily high probability. The small tail property comes into play to guarantee that no infrequent items will be output. Without it, it is possible that items which are not hot will be included in the output. We summarize the results: On any input, all hot items will be included in the output with probability $1-\delta$. An infrequent item can only be output if there are no frequent items in a set: If there are two or more frequent items in a set then we detect this and output nothing for the set. If there is one frequent item, then we will only output an infrequent item if for some split, the weight of the half containing only infrequent items is greater than the half with the hot item. But then both halves with exceed the threshold, and we will reject the set. Lemma 3 bounds the chances of this event happening. This leaves the case where there are no hot items in a set. We will only output an item for this set if in every of the $\log m$ ways we divide the set in two, one half has weight more than $1/(k+1)$ and the other has weight less than this threshold. This event is unlikely, but unwieldy to analyze. Instead, we observe that with the small tail property, this event cannot happen, and so in this case we can guarantee that we will output no infrequent items (with probability 1). We will see in practice through our experiments that on data sets without this property, remarkably few infrequent items are output.

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

Corollary 2. *The set of counters 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 $1-\delta$ by calling* GroupTest$(\log k/\delta, k, 1/(k'+1))$.

Proof. Observe in the proof of Lemma 2 that to find k' hot items, we require that the expected frequency of items falling in the same bucket be at most $1/2(k'+1)$. This is achieved if the number of buckets increases above $2k'$: the more buckets there are, the lower the probability of the set being bad. 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.

Lemma 4. *The output of the algorithm is the same for any reordering of the input data.*

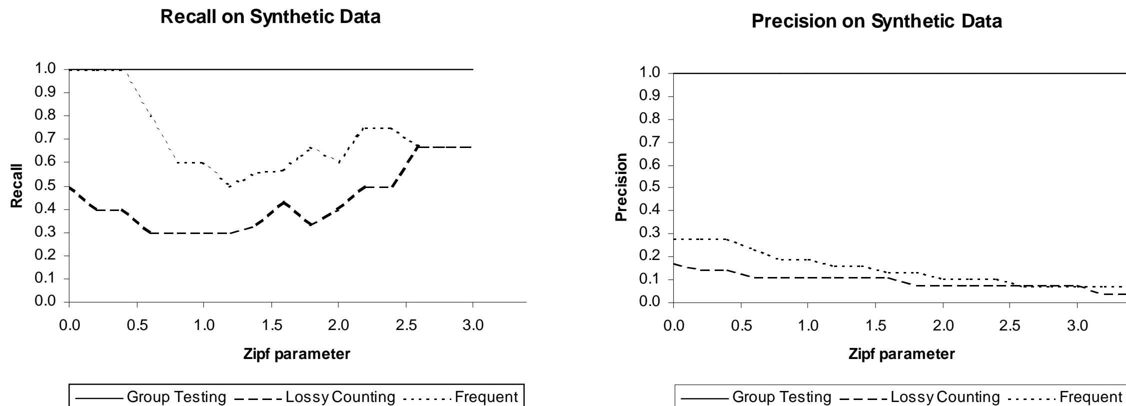

Figure 5: Experiments on synthetic data consisting of 10^6 transactions. Left: testing recall (proportion of the hot items reported). Right: testing precision (proportion of the output items which are hot)

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 lemma follows. □

4. EXPERIMENTS

To evaluate our approach, we implemented our Group Testing algorithm in C. We also implemented two algorithms which operate on non-dynamic data, the algorithm Lossy Counting [23] and Frequent [9]. 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 is deleted. When an item without a counter is deleted, then we take no action.[4] This modification ensures that when the algorithms encounter an inserts-only dataset, then their action is the same as the original algorithms. We ran tests on both synthetic and real data, and measure the two standard qualities of the output: the *recall* and the *precision*.

Definition 3. 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 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 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 bound the precision, whereas for Lossy Counting, the parameter ϵ affects the precision indirectly (depending on the properties of the sequence). Meanwhile, given a sequence with the small tail property, Group Testing guarantees that no infrequent items will be output and for general sequences it is unlikely to output infrequent items.

4.1 Synthetic Data

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 was drawn from a zipf distribution with varying parameter; lastly, a sequence of deletes was distributed uniformly over the same range as the starting sequence. The net effect of this sequence is that first and last groups of transactions should (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 datasets of 1,000,000 items so it was possible to compute the exact answers in order to compare, and searched for the $k = 50$ hot items while varying the zipf parameter of the signal from 0 (uniform) to around 3 (highly skewed). The results are shown in Figure 5, with the recall plotted on the left, and the precision on the right.

We immediately see that existing algorithms perform very badly on this data set including deletions. Lossy counting performs worst on both recall and precision, while Frequent, which does reasonably well when zipf parameter of the signal is low (all hot items have about the same frequency) does less well as the distribution becomes more skewed (when the lower ranked frequent items are pushed closer to the threshold. Group Testing finds all hot items, and only hot items in every case, even though the zipf distribution does not satisfy the small tail assumptions: this property is slightly stronger than is needed, and so Group Testing is able to succeed on data sets on which it does not hold. Even when the recall of the other algorithms is reasonably good (finding around three-quarters of the hot items), their precision

[4]Many variations of this theme are possible. Our experimental results here that compare our algorithms to modifications of Lossy Counting [23] and Frequent [9] should be considered proof-of-concept only.

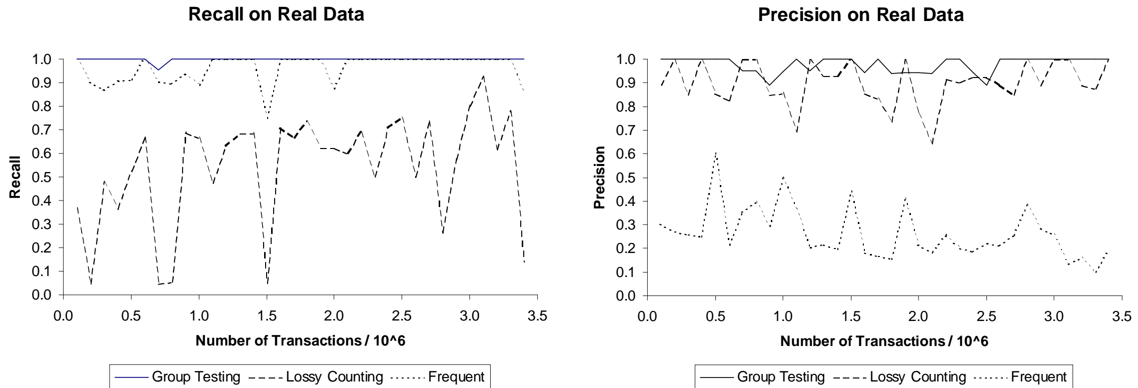

Figure 6: Performance results on real data

is very poor: for every hot item that is reported, around ten infrequent items are also included in the output, and we cannot distinguish between these two types.

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: Group Testing used 17Kb, Lossy Counting used 6Kb on average, and Frequent needed only 3Kb. Additional speed could be achieved by a more optimized implementation, with more attention paid to making the hash function fast to evaluate. It is also trivial to parallelize the Group Testing algorithm, since each test can be done separately.

4.2 Real Data

We obtained data from one of AT&Ts networks for part of a day, totaling around 100Mb. 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 many tens of thousands 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 current frequency greater than 1%. We were grouping connections by their regional code, giving many millions of possible pairs, m, although we discovered that 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 6.

The Group Testing approach is shown to be justified here on real data, which has no guarantee of having the small tail property. In terms of both recall and precision, it is near perfect. On one occasion, it overlooked a hot item, and a few times it includes items which are 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. Lossy Counting performs 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 looks reasonably good especially as time progresses, but its precision, which begins poorly, appears to degrade further. One possible explanation is that the algorithm is collecting all items which are are ever hot, and outputting these whether they are hot or not. Certainly, it outputs between two to three times as many items as are currently hot, meaning that its output will necessarily contain many infrequent items.

Lastly, we ran tests which demonstrated the flexibility of our approach. As noted in Section 3.3, if we have created a set of counters 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 re-computation. 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 7. As predicted, the recall level was the same (100% throughout), and precision was high, with a few non-hot 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. Results maintained an

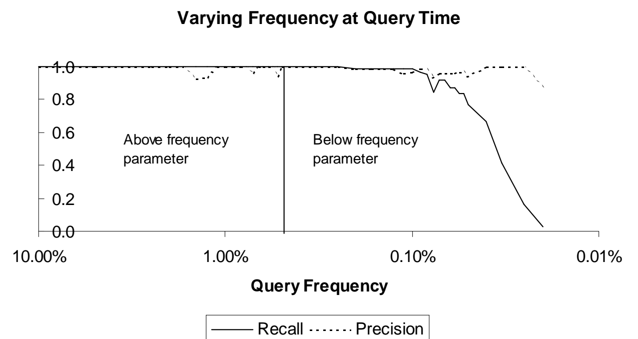

Figure 7: Choosing the frequency level at query time: the data structure was built for queries at the 0.5% level, but was then tested with queries ranging from 10% to 0.02%

impressive level of recall and precision down to around the 0.1% 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). We were unable to compare with other algorithms, since not only are these not designed to cope with dynamic transaction sequences, but they are also designed around a fixed frequency threshold, which has to be supplied in advance, and cannot be chosen at query time.

5. CONCLUSIONS

We proposed a new method for identifying hot items which occur more than some frequency threshold. This is the first method which can cope with dynamic datasets, that is, the removal as well as the addition of items. It performs to a high degree of recall in practice, as guaranteed by our analysis of the algorithm, and is simple to implement.

There are also some future extensions of this work, which we outline here. The algorithm here allows us to aggregate information from separate sources by summing the counters from two or more copies (for example, to compute the new hot items when merging two sets). This should be contrasted to other approaches [4], which also compute the overall hot items from multiple sources, but keeps 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.

In future work, it will be interesting to attempt to use our structure in order to compare the difference in frequencies between difference datasets (which items have the biggest frequency *change* between two datasets?). This is of interest in a number of scenarios, such as trend analysis, financial data sets and anomaly detection [26].

Our approach of group testing may have application to other problems, notably in designing summary data structures for 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 subsets, in order to give a fully deterministic construction for maintaining hot items. The 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.

6. ACKNOWLEDGMENTS

The first author is supported in part by DIMACS through grant number NSF EIA 02-05116

7. REFERENCES

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data structures and algorithms.* Addison Wesley, 1987.

[2] N. Alon, P. Gibbons, Y. Matias, and M. Szegedy. Tracking join and self-join sizes in limited storage. In *Proceedings of the Eighteenth ACM Symposium on Principles of Database Systems (PODS '99)*, pages 10–20, 1999.

[3] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing*, pages 20–29, 1996.

[4] B. Babcock and C. Olston. Distributed top-k monitoring. In *Proceedings of ACM SIGMOD*, 2003.

[5] D. Barbara, N. Wu, and S. Jajodia. Detecting novel network intrusions using bayes estimators. In *Proceedings of the first SIAM International Conference on Data Mining*, 2001.

[6] B. Boyer and J. Moore. A fast majority vote algorithm. Technical Report 35, Institute for Computer Science, University of Texas, 1982.

[7] J. L. Carter and M. N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, 1979.

[8] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. In *Procedings of the International Colloquium on Automata, Languages and Programming (ICALP)*, pages 693–703, 2002.

[9] E. Demaine, A. López-Ortiz, and J. I. Munro. Frequency estimation of internet packet streams with limited space. In *Proceedings of the 10th Annual European Symposium on Algorithms*, volume 2461 of *Lecture Notes in Computer Science*, pages 348–360, 2002.

[10] D.-Z. Du and F. Hwang. *Combinatorial Group Testing and Its Applications*, volume 3 of *Series on Applied Mathematics.* World Scientific, 1993.

[11] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. D. Ullman. Computing iceberg queries efficiently. In *Proceedings of the Twenty-fourth International Conference on Very Large Databases*, pages 299–310, 1998.

[12] M. Fischer and S. Salzberg. Finding a majority among n votes: Solution to problem 81-5. *Journal of Algorithms*, 3(4):376–379, 1982.

[13] P. Gibbons and Y. Matias. New sampling-based summary statistics for improving approximate query answers. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD-98)*, volume 27 of *ACM SIGMOD Record*, pages 331–342, 1998.

[14] P. Gibbons and Y. Matias. Synopsis structures for massive data sets. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, A, 1999.

[15] P. B. Gibbons, Y. Matias, and V. Poosala. Fast incremental maintenance of approximate histograms. In *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB '97)*, pages 466–475, 1998.

[16] A. Gilbert, S. Guha, P. Indyk, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Fast, small-space algorithms for approximate histogram maintenance. In *Proceedings of the 34th ACM Symposium on Theory of Computing*, 2002.

[17] A. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. QuickSAND: Quick summary and analysis of network data. Technical Report 2001-43, DIMACS, 2001.

[18] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. How to summarize the universe: Dynamic

maintenance of quantiles. In *Proceedings of 28th International Conference on Very Large Data Bases*, 2002.

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

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

[21] R. Karp, C. Papadimitriou, and S. Shenker. A simple algorithm for finding frequent elements in sets and bags. *ACM Transactions on Database Systems*, 2003.

[22] W. Kautz and R. Singleton. Nonrandom binary superimposed codes. *IEEE Transactions on on Information Theory*, 10:363–377, 1964.

[23] G. Manku and R. Motwani. Approximate frequency counts over data streams. In *Proceedings of 28th International Conference on Very Large Data Bases*, pages 346–357, 2002.

[24] J. Misra and D. Gries. Finding repeated elements. *Science of Computer Programming*, 2:143–152, 1982.

[25] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[26] B.-K. Yi, N. Sidiropoulos, T. Johnson, H. Jagadish, C. Faloutsos, and A. Biliris. Online data mining for co-evolving time sequences. In *16th International Conference on Data Engineering (ICDE' 00)*, pages 13–22, 2000.