DOI:10.1145/1562764.1562789

Finding the Frequent Items in Streams of Data

By Graham Cormode and Marios Hadjieleftheriou

Abstract

The frequent items problem is to process a stream of items and find all those which occur more than a given fraction of the time. It is one of the most heavily studied problems in mining data streams, dating back to the 1980s. Many other applications rely directly or indirectly on finding the frequent items, and implementations are in use in large-scale industrial systems. In this paper, we describe the most important algorithms for this problem in a common framework. We place the different solutions in their historical context, and describe the connections between them, with the aim of clarifying some of the confusion that has surrounded their properties.

To further illustrate the different properties of the algorithms, we provide baseline implementations. This allows us to give empirical evidence that there is considerable variation in the performance of frequent items algorithms. The best methods can be implemented to find frequent items with high accuracy using only tens of kilobytes of memory, at rates of millions of items per second on cheap modern hardware.

1. INTRODUCTION

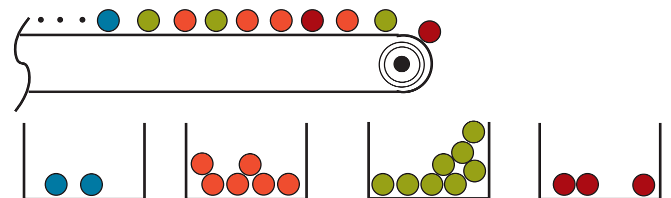
Many data generation processes can be modeled as *data streams*. They produce huge numbers of pieces of data, each of which is simple in isolation, but which taken together lead to a complex whole. For example, the sequence of queries posed to an Internet search engine can be thought of as a stream, as can the collection of transactions across all branches of a supermarket chain. In aggregate, this data can arrive at enormous rates, easily in the realm of hundreds of gigabytes per day or higher. While this data may be archived and indexed within a data warehouse, it is also important to process the data "as it happens," to provide up to the minute analysis and statistics on current trends. Methods to achieve this must be quick to respond to each new piece of information, and use resources which are very small when compared to the total quantity of data.

These applications and others like them have led to the formulation of the so-called "streaming model." In this abstraction, algorithms take only a single pass over their input, and must accurately compute various functions while using resources (space and time per item) that are strictly sublinear in the size of the input—ideally, polynomial in the logarithm of the input size. The output must be produced at the end of the stream, or when queried on the prefix of the stream that has been observed so far. (Other variations ask for the output to be maintained continuously in the presence of updates, or on a "sliding window" of only the most recent updates.) Some problems are simple in this model: for example, given a stream of transactions, finding the mean and standard deviation of the bill totals can be accomplished by retaining a few "sufficient statistics" (sum of all values, sum of squared values, etc.). Others can be shown to require a large amount of information to be stored, such as determining whether a particular search query has already appeared anywhere within a large stream of queries. Determining which problems can be solved effectively within this model remains an active research area.

The *frequent items problem* (also known as the *heavy hitters problem*) is one of the most heavily studied questions in data streams. The problem is popular due to its simplicity to state, and its intuitive interest and value. It is important both in itself, and as a subroutine within more advanced data stream computations. Informally, given a sequence of items, the problem is simply to find those items which occur most frequently. Typically, this is formalized as finding all items whose frequency exceeds a specified fraction of the total number of items. This is shown in Figure 1. Variations arise when the items are given weights, and further when these weights can also be negative.

This abstract problem captures a wide variety of settings. The items can represent packets on the Internet, and the weights are the size of the packets. Then the frequent items represent the most popular destinations, or the heaviest bandwidth users (depending on how the items are extracted from the flow identifiers). This knowledge can help in optimizing routing decisions, for in-network caching, and for planning where to add new capacity. Or, the items can represent queries

Figure 1. A stream of items defines a frequency distribution over items. In this example, with a threshold of $\phi = 20\%$ over the 19 items grouped in bins, the problem is to find all items with frequency at least 3.8—in this case, the green and red items (middle two bins).



A previous version of this paper was published in *Proceedings of the International Conference on Very Large Data Bases* (Aug. 2008).

made to an Internet search engine, and the frequent items are now the (currently) popular terms. These are not simply hypothetical examples, but genuine cases where algorithms for this problem have been applied by large corporations: AT&T[11] and Google,[23] respectively. Given the size of the data (which is being generated at high speed), it is important to find algorithms which are capable of processing each new update very quickly, without blocking. It also helps if the working space of the algorithm is very small, so that the analysis can happen over many different groups in parallel, and because small structures are likely to have better cache behavior and hence further help increase the throughput.

Obtaining efficient and scalable solutions to the frequent items problem is also important since many streaming applications need to find frequent items as a subroutine of another, more complex computation. Most directly, mining frequent *itemsets* inherently builds on finding frequent *items* as a basic building block. Finding the entropy of a stream requires learning the most frequent items in order to directly compute their contribution to the entropy, and remove their contribution before approximating the entropy of the residual stream.[8] The HSS (Hierarchical Sampling from Sketches) technique uses hashing to derive multiple substreams, the frequent elements of which are extracted to estimate the frequency moments of the stream.[4] The frequent items problem is also related to the recently popular area of Compressed Sensing.

Other work solves generalized versions of frequent items problems by building on algorithms for the "vanilla" version of the problem. Several techniques for finding the frequent items in a "sliding window" of recent updates (instead of all updates) operate by keeping track of the frequent items in many sub-windows.[2,13] In the "heavy hitters distinct" problem, with applications to detecting network scanning attacks, the count of an item is the number of *distinct* pairs containing that item paired with a secondary item. It is typically solved extending a frequent items algorithm with distinct counting algorithms.[25] Frequent items have also been applied to models of probabilistic streaming data,[17] and within faster "skipping" techniques.[3]

Thus the problem is an important one to understand and study in order to produce efficient streaming implementations. It remains an active area, with many new research contributions produced every year on the core problem and its variations. Due to the amount of work on this problem, it is easy to miss out some important references or fail to appreciate the properties of certain algorithms. There are several cases where algorithms first published in the 1980s have been "rediscovered" two decades later; existing work is sometimes claimed to be incapable of a certain guarantee, which in truth it can provide with only minor modifications; and experimental evaluations do not always compare against the most suitable methods.

In this paper, we present the main ideas in this area, by describing some of the most significant algorithms for the core problem of finding frequent items using common notation and terminology. In doing so, we also present the historical development of these algorithms. Studying these algorithms is instructive, as they are relatively simple, but can be shown to provide formal guarantees on the quality of their output as a function of an accuracy parameter ε. We also provide baseline implementations of many of these algorithms against which future algorithms can be compared, and on top of which algorithms for different problems can be built. We perform experimental evaluation of the algorithms over a variety of data sets to indicate their performance in practice. From this, we are able to identify clear distinctions among the algorithms that are not apparent from their theoretical analysis alone.

2. DEFINITIONS

We first provide formal definition of the stream and the frequencies f_i of the items within the stream as the number of times item i is seen in the stream.

Definition 1. Given a stream S of n items $t_1 \ldots t_n$, the frequency of an item i is $f_i = |\{j | t_j = i\}|$. The *exact ϕ-frequent items* comprise the set $\{i | f_i > \phi n\}$.

Example. The stream $S = (a, b, a, c, c, a, b, d)$ has $f_a = 3$, $f_b = 2$, $f_c = 2$, $f_d = 1$. For $\phi = 0.2$, the frequent items are a, b, and c.

A streaming algorithm which finds the exact ϕ-frequent items must use a lot of space, even for large values of ϕ, based on the following information-theoretic argument. Given an algorithm that claims to solve this problem for $\phi = 50\%$, we could insert a set S of N items, where every item has frequency 1. Then, we could also insert $N - 1$ copies of item i. If i is now reported as a frequent item (occurring more than 50% of the time) then $i \in S$, else $i \notin S$. Consequently, since correctly storing a set of size N requires $\Omega(N)$ space, $\Omega(N)$ space is also required to solve the frequent items problem. That is, any algorithm which promises to solve the exact problem on a stream of length n must (in the worst case) store an amount of information that is proportional to the length of the stream, which is impractical for the large stream sizes we consider.

Because of this fundamental difficulty in solving the exact problem, an approximate version is defined based on a tolerance for error, which is parametrized by ε.

Definition 2. Given a stream S of n items, the ε-approximate frequent items problem is to return a set of items F so that for all items $i \in F$, $f_i > (\phi - \varepsilon)n$, and there is no $i \notin F$ such that $f_i > \phi n$.

Since the exact ($\varepsilon = 0$) frequent items problem is hard in general, we use "frequent items" or "the frequent items problem" to refer to the ε-approximate frequent items problem. A closely related problem is to estimate the frequency of items on demand.

Definition 3. Given a stream S of n items defining frequencies f_i as above, the frequency estimation problem is to process a stream so that, given any i, an $\hat{f}_i$ is returned satisfying $f_i \leq \hat{f}_i \leq f_i + \varepsilon n$.

A solution to the frequency estimation problem allows the frequent items problem to be solved (slowly): one can estimate the frequency of every possible item i, and report those i's whose frequency is estimated above $(\phi - \varepsilon)n$. Exhaustively enumerating all items can be very time consuming (or infeasible,

e.g., when the items can be arbitrary strings). However, all the algorithms we study here solve both the approximate frequent items problem and the frequency estimation problem at the same time. Most solutions are deterministic, but we also discuss randomized solutions, which allow a small user-specified probability of making a mistake.

3. FREQUENT ITEMS ALGORITHMS

We discuss two main classes of algorithms for finding the frequent items. Counter-based algorithms track a subset of items from the input, and monitor counts associated with these items. We also discuss sketch algorithms, which are (randomized) linear projections of the input viewed as a vector, and solve the frequency estimation problem. They therefore do not explicitly store items from the input. Furthermore, sketch algorithms can support deletion of items (corresponding to updates with a negative weight, discussed in more detail below), in contrast with counter-based schemes, at the cost of increased space usage and update time.

These are by no means the only solutions possible for this problem. Other solutions are based on various notions of randomly sampling items from the input, and of summarizing the distribution of items in order to find *quantiles*, from which the frequent items can be discovered. These solution types have attracted less interest for the frequent items problem, and are less effective based on our full experimental evaluations.[10]

3.1. Counter-based algorithms

Counter-based algorithms decide for each new arrival whether to store this item or not, and if so, what counts to associate with it. A common feature of these algorithms is that when given a new item, they test whether it is one of k being stored by the algorithm, and if so, increment its count. The cost of supporting this "dictionary" operation depends on the model of computation assumed. A simple solution is to use a hash table storing the current set of items, but this means that an otherwise deterministic solution becomes randomized in its time cost, since it takes *expected* $O(1)$ operations to perform this step. Other models assume that there is hardware support for these operations (such as Content Addressable Memory), or else that deterministic "dynamic dictionary algorithms" are used. We sidestep this issue in this presentation by just counting the number of "dictionary" operations in the algorithms.

Majority Algorithm: The problem of frequent items dates back at least to a problem proposed by Moore in 1980. It was published as a "problem" in the *Journal of Algorithms* in the June 1981 issue, as

> [J.Alg 2, P208–209] Suppose we have a list of n numbers, representing the "votes" of n processors on the result of some computation. We wish to decide if there is a majority vote and what the vote is.

Moore, with Boyer, also invented the MAJORITY algorithm in 1980, described in a technical report from early 1981.[6] To them, this was mostly of interest from the perspective of automatically proving the correctness of the solution (the details of this were published in 1991, along with a partial history[7]). In the December 1982, *Journal of Algorithms*, a solution provided by Fischer and Salzburg was published.[15] Their proposed algorithm, although presented differently, was essentially identical to MAJORITY, and was accompanied by an analysis of the number of comparisons needed to solve the problem. MAJORITY can be stated as follows: store the first item and a counter, initialized to 1. For each subsequent item, if it is the same as the currently stored item, increment the counter. If it differs, and the counter is zero, then store the new item and set the counter to 1; else, decrement the counter. After processing all items, the algorithm guarantees that if there is a majority vote, then it must be the item stored by the algorithm. The correctness of this algorithm is based on a pairing argument: if every nonmajority item is paired with a majority item, then there should still remain an excess of majority items. Although not posed as a streaming problem, the algorithm has a streaming flavor: it takes only one pass through the input (which can be ordered arbitrarily) to find a majority item. To verify that the stored item really is a majority, a second pass is needed to simply count the true number of occurrences of the stored item. Without this second pass, the algorithm has a partial guarantee: if there is an exact majority item, it is found at the end of the first pass, but the algorithm is unable to determine whether this is the case. Note that as the hardness results for Definition 1 show, no algorithm can correctly distinguish the cases when an item is just above or just below the threshold in a single pass without using a large amount of space.

The "Frequent" Algorithm: Twenty years later, the problem of streaming algorithms was an active research area, and a generalization of the Majority algorithm was shown to solve the problem of finding all items in a sequence whose frequency exceeds a $1/k$ fraction of the total count.[14, 18] Instead of keeping a single counter and item from the input, the FREQUENT algorithm stores $k - 1$ (item, counter) pairs. The natural generalization of the Majority algorithm is to compare each new item against the stored items T, and increment the corresponding counter if it is among them. Else, if there is some counter with a zero count, it is allocated to the new item, and the counter set to 1. If all $k - 1$ counters are allocated to distinct items, then all are decremented by 1. A grouping argument is used to argue that any item which occurs more than n/k times must be stored by the algorithm when it terminates. Figure 2 illustrates some of the operations on this data structure. Pseudocode to illustrate this algorithm is given in Algorithm 1, making use of set notation to represent the dictionary operations on the set of stored items T: items are added and removed from this set using set union and set subtraction, respectively, and we allow ranging over the members of this set (any implementation will have to choose how to support these operations). We also assume that each item j stored in T has an associated counter c_j. For items not stored in T, then c_j is implicitly defined as 0 and does not need to be explicitly stored.

In fact, this generalization was first proposed by Misra and Gries as "Algorithm 3"[22] in 1982: the papers published in 2002 (which cite Fischer[15] but not Misra[22]) were actually rediscoveries of their algorithm. In deference to its initial discovery, this algorithm has been referred to as the "Misra–Gries" algorithm

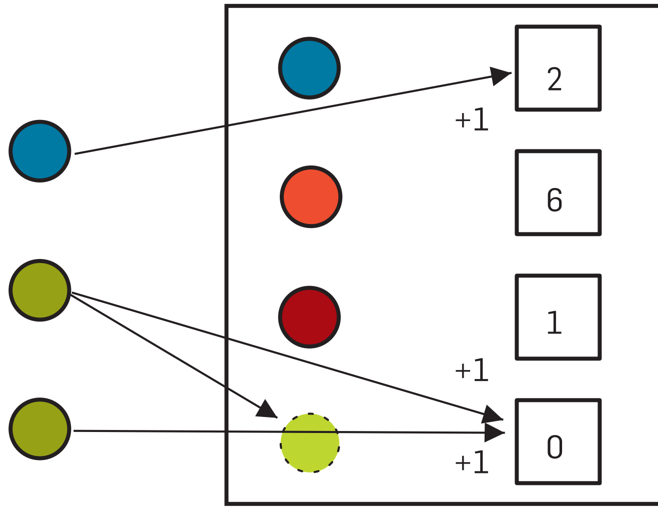


Figure 2. Counter-based data structure: the blue (top) item is already stored, so its count is incremented when it is seen. The green (middle) item takes up an unused counter, then a second occurrence increments it.

in more recent work on streaming algorithms. In the same paper, an "Algorithm 2" correctly solves the problem but has only speculated worst-case space bounds. Some works have asserted that the FREQUENT algorithm does not solve the frequency estimation problem accurately, but this is erroneous. As observed by Bose et al.,[5] executing this algorithm with $k = 1/\varepsilon$ ensures that the count associated with each item on termination is at most εn below the true value.

The time cost of the algorithm is dominated by the $O(1)$ dictionary operations per update, and the cost of decrementing counts. Misra and Gries use a balanced search tree, and argue that the decrement cost is amortized $O(1)$; Karp et al. propose a hash table to implement the dictionary[18]; and Demaine et al. show how the cost of decrementing can be made worst-case $O(1)$ by representing the counts using offsets and maintaining multiple linked lists.[14]

LossyCounting: The LOSSYCOUNTING algorithm was proposed by Manku and Motwani in 2002,[19] in addition to a randomized sampling-based algorithm and techniques for extending from frequent items to frequent itemsets. The algorithm stores tuples which comprise an item, a lower bound on its count, and a "delta" (Δ) value which records the difference between the upper bound and the lower bound. When processing the ith item in the stream, if information is currently stored about the item then its lower bound is increased by one; else, a new tuple for the item is created with the lower bound set to one, and Δ set to $\lfloor i/k \rfloor$. Periodically, all tuples whose upper bound is less than $\lfloor i/k \rfloor$ are deleted. These are correct upper and lower bounds on the count of each item, so at the end of the stream, all items whose count exceeds n/k must be stored. As with FREQUENT, setting $k = 1/\varepsilon$ ensures that the error in any approximate count is at most εn. A careful argument demonstrates that the worst-case space used by this algorithm is $O(\frac{1}{\varepsilon}\log \varepsilon n)$, and for certain time-invariant input distributions it is $O(\frac{1}{\varepsilon})$.

Storing the delta values ensures that highly frequent items which first appear early on in the stream have very accurate approximated counts. But this adds to the storage cost. A variant of this algorithm is presented by Manku in a presentation of the paper,[20] which dispenses with explicitly storing the delta values, and instead has all items sharing an implicit value of $\Delta(i) = \lfloor i/k \rfloor$. The modified algorithm stores (item, count) pairs. For each item in the stream, if it is stored, then the count is incremented; otherwise, it is initialized with a count of 1. Every time $\Delta(i)$ increases, all counts are decremented by 1, and all items with zero count are removed from the data structure. The same proof suffices to show that the space bound is $O(\frac{1}{\varepsilon}\log \varepsilon n)$. This version of the algorithm is quite similar to Algorithm 2 presented in Misra[22]; but in Manku,[20] a space bound is proven. The time cost is $O(1)$ dictionary operations, plus the periodic compress operations which require a linear scan of the stored items. This can be performed once every $O(\frac{1}{\varepsilon}\log \varepsilon n)$ updates, in which time the number of items stored has at most doubled, meaning that the amortized cost of compressing is $O(1)$. We give pseudocode for this version of the algorithm in Algorithm 2, where again T represents the set of currently monitored items, updated by set operations, and c_j are corresponding counts.

SpaceSaving: All the deterministic algorithms presented so far have a similar flavor: a set of items and counters are kept, and various simple rules are applied when a new item arrives (as illustrated in Figure 2). The SPACESAVING algorithm introduced in 2005 by Metwally et al. also fits this template.[21] Here, k (item, count) pairs are stored, initialized by the first k distinct

Algorithm 1: FREQUENT(k)

```
n ← 0;
T ← ∅;
foreach i do
    n ← n + 1;
    if i ∈ T then
        c_i ← c_i + 1;
    else if |T| < k–1 then
        T ← T ∪ {i};
        c_i ← 1;
    else forall j ∈ T do
        c_j ← c_j – 1;
        if c_j = 0 then T ← T\{j};
```

Algorithm 2: LOSSYCOUNTING(k)

```
n ← 0; Δ ← 0; T ← ∅;
foreach i do
    n ← n + 1;
    if i ∈ T then c_i ← c_i + 1;
    else
        T ← T ∪ {i};
        c_i ← 1+ Δ;
if ⌊n/K⌋ ≠ Δ then
    Δ ← ⌊n/k⌋;
    forall j ∈ T do
        if c_j < Δ then T ← T \ {j};
```

Algorithm 3: SPACESAVING(k)

```
n ← 0;
T ← ∅;
foreach i do
    n ← n + 1;
    if i ∈ T then c_i ← c_i + 1;
    else if |T| < k then
        T ← T ∪ {i};
        c_i ← 1;
    else
        j ← arg min_{j∈T} c_j;
        c_i ← c_j + 1;
        T ← T ∪ {i} \ {j};
```

items and their exact counts. As usual, when the next item in the sequence corresponds to a monitored item, its count is incremented; but when the next item does not match a monitored item, the (item, count) pair with the smallest count has its item value replaced with the new item, and the count incremented. So the space required is $O(k)$ (respectively $O(\frac{1}{\varepsilon})$), and a short proof demonstrates that the counts of all stored items solve the frequency estimation problem with error n/k (respectively εn). It also shares a useful property with LossyCounting, that items which are stored by the algorithm early in the stream and are not removed have very accurate estimated counts. The algorithm appears in Algorithm 3. The time cost is bounded by the dictionary operation of finding if an item is stored, and of finding and maintaining the item with minimum count. Simple heap implementations can track the smallest count item in $O(\log 1/\varepsilon)$ time per update. When all updates are unitary (+1), a faster approach is to borrow ideas from the Demaine et al. implementation of Frequent, and keep the items in groups with equal counts. By tracking a pointer to the group with smallest count, the find minimum operation takes constant time, while incrementing counts take $O(1)$ pointer operations (the "StreamSummary" data structure desribed by Metwally et al.[21]).

3.2. Sketch algorithms

Here, we use the term "sketch" to denote a data structure which can be thought of as a linear projection of the input. That is, if we imagine the stream as implicitly defining a vector whose ith entry is f_i, the sketch is the product of this vector with a matrix. For the algorithm to use small space, this matrix will be implicitly defined by a small number of bits. The sketch algorithms described here use hash functions to define a (very sparse) linear projection. Both views (hashing or linear projection) can be helpful in explaining the methods, and it is usually possible to alternate between the two without confusion. Because of their linearity, it follows immediately that updates with negative values can easily be accommodated by such sketching methods. This allows us to model the removal of items (to denote the conclusion of a packet flow; or the return of a previously bought item, say) as an update with negative weight.

The two sketch algorithms outlined below solve the frequency estimation problem. They need additional data information to solve the frequent items problem, so we also describe algorithms which augment the stored sketch to find frequent items quickly. The algorithms are randomized, which means that in addition to the accuracy parameter ε, they also take a failure probability δ so that (over the random choices made in choosing the hash functions) the probability of failure is at most δ. Typically, δ can be chosen to be very small (e.g., 10^{-6}) while keeping the space used by the sketch low.

CountSketch: The first sketch in the sense that we use the term was the AMS or Tug-of-war sketch due to Alon et al.[1] This was used to estimate the second frequency moment, $F_2 = \Sigma_i f_i^2$. It was subsequently observed that the same data structure could be used to estimate the inner product of two frequency distributions, i.e., $\Sigma_i f_i f_i'$ for two distributions given (in a stream) by f_i and f_i'. But this means that if f_i is defined by a stream, at query time we can find the product with $f_i' = 1$ and $f_j' = 0$ for all $j \neq i$. Then, the true answer to the inner product should be exactly f_i. The error guaranteed by the sketch turns out to be $\varepsilon F_2^{1/2} \leq \varepsilon n$ with probability of at least $1 - \delta$ for a sketch of size $O(\frac{1}{\varepsilon^2} \log 1/\delta)$. The ostensibly dissimilar technique of "Random Subset Sums"[16] (on close inspection) turns out to be isomorphic to this instance of the algorithm.

Maintaining the AMS data structure is slow, since it requires updating the whole sketch for every new item in the stream. The CountSketch algorithm of Charikar et al.[9] dramatically improves the speed by showing that the same underlying technique works if each update only affects a small subset of the sketch, instead of the entire summary. The sketch consists of a two-dimensional array C with d rows of w counters each. There are two hash functions for each row, h_j which maps input items onto $[w]$, and g_j which maps input items onto $\{-1, +1\}$. Each input item i causes $g_j(i)$ to be added on to entry $C[j, h_j(i)]$ in row j, for $1 \leq j \leq d$. For any row j, the value $g_j(i) C[j, h_j(i)]$ is an unbiased estimator for f_i. The estimate $\hat{f}_i$ is the median of these estimates over the d rows. Setting $d = \log \frac{4}{\delta}$ and $w = O(\frac{1}{\varepsilon^2})$ ensures that f_i has error at most $\varepsilon F_2^{1/2} \leq \varepsilon n$ with probability of at least $1 - \delta$. This guarantee requires that the hash functions are chosen randomly from a family of "four-wise independent" hash functions.[24] The total space used is $O(\frac{1}{\varepsilon^2} \log \frac{1}{\delta})$, and the time per update is $O(\log \frac{1}{\delta})$ worst case. Figure 3 shows a schematic of the data structure under the update procedure: the new item i gets mapped to a different location in each row, where $g_j(i)$ is added on to the current counter value in that location. Pseudocode for the core of the update algorithm is shown in Algorithm 4.

CountMin Sketch: The CountMin sketch algorithm of Cormode and Muthukrishnan[12] can be described in similar terms to CountSketch. An array of $d \times w$ counters is maintained, along with d hash functions h_j. Each update is mapped onto d entries in the array, each of which is incremented. Now $\hat{f}_i = \min_{1 \leq j \leq d} C[j, h_j(i)]$. The Markov inequality is used to show that the estimate for each j overestimates by less than n/w, and repeating d times reduces the probability of error exponentially. So setting $d = \log \frac{1}{\delta}$ and $w = O(\frac{1}{\varepsilon})$ ensures that $\hat{f}_i$ has error at most εn with probability of at least $1 - \delta$. Consequently, the space is $O(\frac{1}{\varepsilon} \log \frac{1}{\delta})$ and the time per update is $O(\log \frac{1}{\delta})$. The data structure and update procedure is consequently much like that illustrated for the CountSketch in Figure 3, with $g_j(i)$ always equal to 1. The update algorithm is shown in Algorithm 5.

Finding Frequent Items Using a Hierarchy: Since sketches solve the case when item frequencies can decrease, more

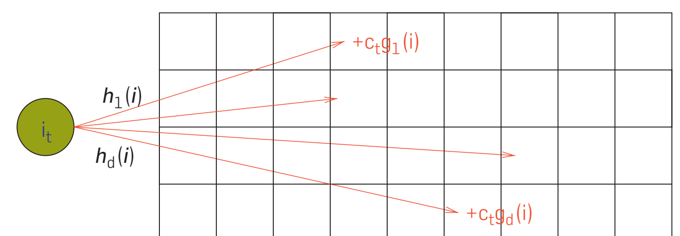


Figure 3. Sketch data structure: each new item is mapped to a set of counters, which are incremented.

Algorithm 4: CountSketch(w, d)

$C[1,1] \ldots C[d,w] \leftarrow 0$;
for $j \leftarrow 1$ **to** d **do**
  Initialize g_j, h_j;
foreach i **do**
  $n \leftarrow n+1$;
  for $j \leftarrow 1$ **to** d **do**
    $C[j, h_j(i)] \leftarrow C[j, h_j(i), j] + g_j(i)$;

Algorithm 5: CountMin(w, d)

$C[1,1] \ldots C[d,w] \leftarrow 0$;
for $j \leftarrow 1$ **to** d **do**
  Initialize g_j;
foreach i **do**
  $n \leftarrow n+1$;
  for $j \leftarrow 1$ **to** d **do**
    $C[j, h_j(i)] \leftarrow C[j, h_j(i)] + 1$;

complex algorithms are needed to find the frequent items. Here, we assume a "strict" case, where negative updates are possible but no negative frequencies are allowed. In this strict case, an approach based on divide-and-conquer will work: additional sketches are used to determine which *ranges* of items are frequent.[12] If a range is frequent, then it can be split into b nonoverlapping subranges and the frequency of each subrange estimated from an appropriate sketch, until a single item is returned. The choice of b trades off update time against query time: if all items $i \in \{1 \ldots U\}$, then $\lceil \log_b U \rceil$ sketches suffice, but each potential range is split into $b > 1$ subranges when answering queries. Thus, updates take $O(\log_b U \log \frac{1}{\delta})$ hashing operations, and $O(1)$ counter updates for each hash. Typically, moderate constant values of b are used (between 2 and 256, say); choosing b to be a power of two allows fast bit-shifts to be used in query and update operations instead of slower divide and mod operations. This results in CountMin sketch Hierarchical and CountSketch Hierarchical algorithms.

Finding Frequent Items Using Group Testing: An alternate approach is based on "combinatorial group testing" (CGT), which randomly divides the input into buckets so that we expect at most one frequent item in each group. Within each bucket, the items are divided into subgroups so that the "weight" of each group indicates the identity of the frequent item. For example, separating the counts of the items with odd identifiers and even identifiers will indicate whether the heavy item is odd or even; repeating this for all bit positions reveals the full identity of the item. This can be seen as an extension of the CountMin sketch, since the structure resembles the buckets of the sketch, with additional information on subgroups of each bucket (based on the binary representation of items falling in the bucket); further, the analysis and properties are quite close to those of a Hierarchical CountMin sketch. This increases the space to $O(\frac{1}{\varepsilon} \log U \log \delta)$ when the binary representation takes log U bits. Each update requires $O(\log \frac{1}{\delta})$ hashes as before, and updating $O(\log U)$ counters per hash.

4. EXPERIMENTAL COMPARISON

4.1. Setup

We compared these algorithms under a common implementation framework to test as accurately as possible their relative performance. All algorithms were implemented using C++, and used common subroutines for similar tasks (e.g., hash tables) to increase comparability. We ran experiments on a 4 Dual Core Intel(R) Xeon(R) 2.66 GHz with 16GB of RAM running Windows 2003 Server. The code was compiled using Microsoft's Visual C++ 2005 compiler and g++ 3.4.4 on cygwin. We did not observe significant differences between the two compilers. We report here results obtained using Visual C++ 2005. The code is available from http://www.research.att.com/~marioh/frequent–items/.

For every algorithm we tested a number of implementations, using different data structures to implement the basic set operations. For some algorithms the most robust implementation choice was clear; for others we present results of competing solutions. For counter-based algorithms we examine: Frequent using the Demaine et al. implementation technique of linked lists (F), LossyCounting keeping separate delta values for each item (LCD), LossyCounting without deltas (LC), SpaceSaving using a heap (SSH), and SpaceSaving using linked lists (SSL). We also examine sketch-based methods: hierarchical CountSketch (CS), hierarchical CountMin sketch (CMH), and the CGT variant of CountMin.

We ran experiments using 10 million packets of HTTP traffic, representing 24 hours of traffic from a backbone router in a major network. Experiments on other real and synthetic datasets are shown in an extended version of this article.[10] We varied the frequency threshold ϕ, from 0.0001 to 0.01. In our experiments, we set the error guarantee $\varepsilon = \phi$, since our results showed that this was sufficient to give high accuracy in practice.

We compare the efficiency of the algorithms with respect to

- Update throughput, measured in number of updates per millisecond.
- Space consumed, measured in bytes.
- Recall, measured in the total number of true heavy hitters reported over the number of true heavy hitters given by an exact algorithm.
- Precision, measured in total number of true heavy hitters reported over the total number of answers reported. Precision quantifies the number of false positives reported.
- Average relative error of the reported frequencies: We measure separately the average relative error of the frequencies of the true heavy hitters, and the average relative error of the frequencies of the false positive answers. Let the true frequency of an item be f and the measured frequency $\tilde{f}$. The absolute relative error is defined by $\delta f = \frac{|f - \tilde{f}|}{f}$. We average the absolute relative errors over all measured frequencies.

For all of the above, we perform 20 runs per experiment (by dividing the input data into 20 chunks and querying the algorithms once at the end of each run). We report averages on all graphs, along with the 5th and 95th percentiles as error bars.

4.2. Counter-based algorithms

Space and Time Costs: Figure 4(a) shows the update throughput of the algorithms as a function of increasing frequency threshold (ϕ). The SpaceSaving and Frequent algorithms are fastest, while the two variations of LossyCounting are appreciably slower. On this data set, SSL and SSH are equally very fast, but on some other data sets SSL was significantly faster than SSL, showing how data structure choices can affect performance. The range of frequency thresholds (ϕ) considered did not affect update throughput (notice the log scale on the horizontal axis). The space used by these algorithms at the finest accuracy level was less than 1MB. SSL used 700KB for $\phi = 0.0001$, while the other algorithms all required approximately 400KB. Since the space cost varies with $1/\phi$, for $\phi = 0.01$, the cost was 100 times less, i.e., a matter of kilobytes. This range of sizes is small enough to fit within a mod ern second level cache, so there is no obvious effect due to crossing memory boundaries on the architectures tested on. A naive solution that maintains one counter per input item would consume many megabytes (and this grows linearly with the input size). This is at least 12 times larger than SSH for $\phi = 0.0001$ (which is the most robust algorithm in terms of space), and over a thousand times larger than all algorithms for $\phi = 0.01$. Clearly, the space benefit of these algorithms, even for small frequency thresholds, is substantial in practice.

Precision, Recall, and Error: All algorithms tested guarantee perfect recall (they will recover every item that is frequent). Figure 4(b) plots the *precision*. We also show the 5th and 95th percentiles in the graphs as error bars. Precision is the total number of true answers returned over the total number of answers. Precision is an indication of the number of false positives returned. Higher precision means smaller number of false positive answers. There is a clear distinction between different algorithms in this case. When using $\varepsilon = \phi$, F results in a very large number of false positive answers, while LC and LCD result in approximately 50% false positives for small ϕ parameters, but their precision improves as skewness increases. Decreasing ε relative to ϕ would improve this at the cost of increasing the space used. However, SSL and SSH yield 100% accuracy in all cases (i.e., no false positives), with about the same or better space usage. Note that these implement the same algorithm and so have the same output, only differing in the underlying implementation of certain data structures. Finally, notice that by keeping additional per-item information, LCD can sometimes distinguish between truly frequent and potentially frequent items better than LC.

Figure 4(c) plots the average relative error in the frequency estimation of the truly frequent items. The graph also plots the 5th and 95th percentiles as error bars. The relative error of F decreases with ϕ, while the error of LossyCounting increases with ϕ. Note that F always returns an underestimate of the true count of any item; LC and LCD always return overestimates based on a Δ value, and so yield inflated estimates of the frequencies of infrequent items.

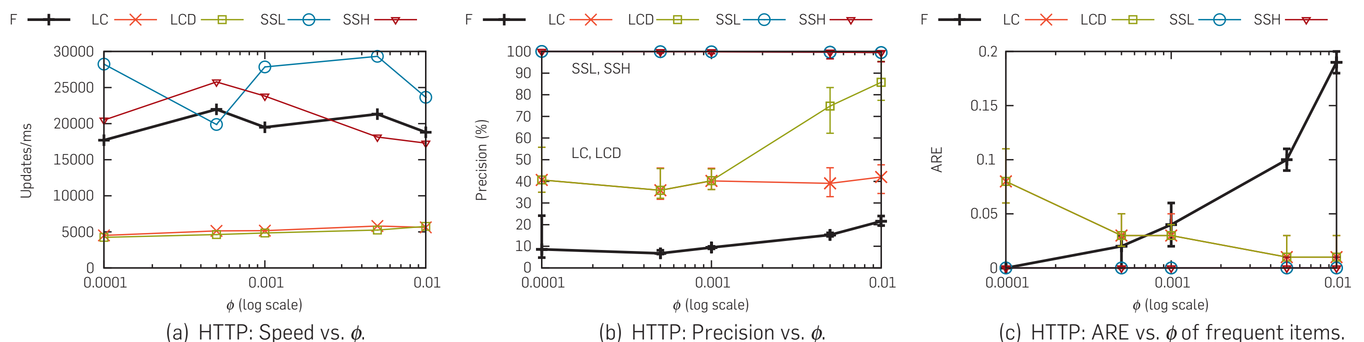
Conclusions: Overall, the SpaceSaving algorithm appears conclusively better than other counter-based algorithms, across a wide range of data types and parameters. Of the two implementations compared, SSH exhibits very good performance in practice. It yields very good estimates, typically achieving 100% recall and precision, consumes very small space, and is fairly fast to update (faster than LC and LCD). Alternatively, SSL is the fastest algorithm with all the good characteristics of SSH, but consumes twice as much space on average. If space is not a critical issue, SSL is the implementation of choice.

4.3. Sketch algorithms

The advantage of sketches is that they support deletions, and hence are the only alternative in fully dynamic environments. This comes at the cost of increased space consumption and slower update performance. We used a hierarchy with branching factor $b = 16$ for all algorithms, after running experiments with several values and choosing the best trade-off between speed, size, and precision. The sketch depth is set to $d = 4$ throughout, and the width to $w = 2/\phi$, based on the analysis of the CountMin sketch. This keeps the space usage of CS and CMH relatively similar, and CGT is larger by constant factors.

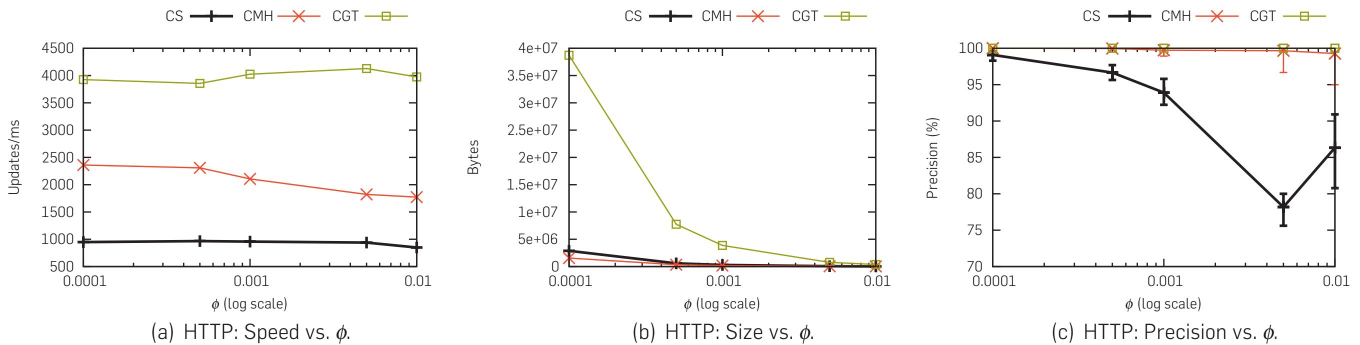
Space and Time Cost: Figure 5(a) shows the update throughput of the algorithms. Update throughput is mostly unaffected by variations in ϕ, though CMH does seem to become slower for larger values of ϕ. CS has the slowest update rate among all algorithms, due to the larger number of hashing operations needed. Still, the fastest sketch algorithm is from 5 up to 10 times slower than the fastest counter-based algorithm. Figure 5(b) plots the space consumed. The size of the sketches is fairly large compared to counter-based algorithms: of the order of several megabytes

Figure 4. Performance of counter-based algorithms on real network data (a) speed, (b) precision, and (c) average relative error.



(a) HTTP: Speed vs. ϕ. (b) HTTP: Precision vs. ϕ. (c) HTTP: ARE vs. ϕ of frequent items.

Figure 5. Performance of sketch algorithms on real network data (a) speed, (b) size, and (c) precision.



(a) HTTP: Speed vs. ϕ.

(b) HTTP: Size vs. ϕ.

(c) HTTP: Precision vs. ϕ.

for small values of ϕ. CMH is the most space efficient sketch and still consumes space three times as large as the least space efficient counter-based algorithm.

Precision, Recall, and Error: The sketch algorithms all have near perfect recall, as is the case with the counter-based algorithms. Figure 5(c) shows that they also have good precision, with CS reporting the largest number of false positives. Nevertheless, on some other datasets we tested (not shown here), the results were reversed. We also tested the average relative error in the frequency estimation of the truly frequent items. For sufficiently skewed distributions all algorithms can estimate item frequencies very accurately, and the results from all sketches were similar since all hierarchical sketch algorithms essentially correspond to a single instance of a CountSketch or CountMin sketch of equal size.

Conclusions: There is no clear winner among the sketch algorithms. CMH has small size and high update throughput, but is only accurate for highly skewed distributions. CGT consumes a lot of space but it is the fastest sketch and is very accurate in all cases, with high precision and good frequency estimation accuracy. CS has low space consumption and is very accurate in most cases, but has slow update rate and exhibits some random behavior.

5. CONCLUDING REMARKS

We have attempted to present algorithms for finding frequent items in streams, and give an experimental comparison of their behavior to serve as a baseline for comparison. For insert-only streams, the clear conclusion of our experiments is that the SpaceSaving algorithm, a relative newcomer, has surprisingly clear benefits over others. We observed that implementation choices, such as whether to use a heap or lists of items grouped by frequencies, trade-off speed, and space. For sketches to find frequent items over streams including negative updates, there is not such a clear answer, with different algorithms excelling at different aspects of the problem. We do not consider this the end of the story, and continue to experiment with other implementation choices. Our source code and experimental test scripts are available from http://www.research.att.com/~marioh/frequent–items/ so that others can use these as baseline implementations.

We conclude by outlining some of the many variations of the problem

- In the *weighted input* case, each update comes with an associated weight (such as a number of bytes, or number of units sold). Here, sketching algorithms directly handle weighted updates because of their linearity. The SpaceSaving algorithm also extends to the weighted case, but this is not known to be the case for the other counter-based algorithms discussed.
- In the *distributed data* case, different parts of the input are seen by different parties (different routers in a network, or different stores making sales). The problem is then to find items which are frequent over the union of all the inputs. Again due to their linearity properties, sketches can easily solve such problems. It is less clear whether one can merge together multiple counter-based summaries to obtain a summary with the same accuracy and worst-case space bounds.
- Often, the item frequencies are known to follow some statistical distribution, such as the Zipfian distribution. With this assumption, it is sometimes possible to prove a smaller space requirement on the algorithm, as a function of the amount of "skewness" in the distribution.[9, 21]
- In some applications, it is important to find how many *distinct* observations there have been, leading to a *distinct heavy hitters* problem. Now the input stream S is of the form (i, j), and f_i is defined as $|\{j|(i, j) \in S\}|$. Multiple occurrences of (i, j) only count once towards f_i. Techniques for "distinct frequent items" rely on combining frequent items algorithms with "count distinct" algorithms.[25]
- While processing a long stream, it may be desirable to weight more recent items more heavily than older ones. Various models of *time decay* have been proposed to achieve this. In a sliding window, only the most recent items are considered to define the frequent items.[2] More generally time decay can be formalized via a function which assigns a weight to each item in the stream as a function of its (current) age, and the frequency of an item is the sum of its decayed weights.

Each of these problems has also led to considerable effort from the research community to propose and analyze algorithms. This research is ongoing, cementing the position of the frequent items problem as one of the most enduring and intriguing in the realm of algorithms for data streams.

References

1. Alon, N., Matias, Y., Szegedy, M. The space complexity of approximating the frequency moments. In *ACM Symposium on Theory of Computing*, (1996), 20–29. Journal version in *J. Comp. Syst. Sci. 58* (1999), 137–147.
2. Arasu, A., Manku, G.S. Approximate counts and quantiles over sliding windows. In *ACM Principles of Database Systems* (2004).
3. Bhattacharrya, S., Madeira, A., Muthukrishnan, S., Ye, T. How to scalably skip past streams. In *Scalable Stream Processing Systems (SSPS) Workshop with ICDE 2007* (2007).
4. Bhuvanagiri, L., Ganguly, S., Kesh, D., Saha, C. Simpler algorithm for estimating frequency moments of data streams. In *ACM-SIAM Symposium on Discrete Algorithms* (2006).
5. Bose, P., Kranakis, E., Morin, P., Tang, Y. Bounds for frequency estimation of packet streams. In *SIROCCO* (2003).
6. Boyer, R.S., Moore, J.S. A fast majority vote algorithm. Technical Report ICSCA-CMP-32, Institute for Computer Science, University of Texas (Feb. 1981).
7. Boyer, R.S., Moore, J.S. MJRTY—a fast majority vote algorithm. In *Automated Reasoning: Essays in Honor of Woody Bledsoe*, Automated Reasoning Series. Kluwer Academic Publishers, 1991, 105–117.
8. Chakrabarti, A., Cormode, G., McGregor, A. A near-optimal algorithm for computing the entropy of a stream. In *ACM-SIAM Symposium on Discrete Algorithms* (2007).
9. Charikar, M., Chen, K., Farach-Colton, M. Finding frequent items in data streams. In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP)* (2002).
10. Cormode, G., Hadjieleftheriou, M. Finding frequent items in data streams. In *International Conference on Very Large Data Bases* (2008).
11. Cormode, G., Korn, F., Muthukrishnan, S., Johnson, T., Spatscheck, O. Srivastava, D. Holistic UDAFs at streaming speeds. In *ACM SIGMOD International Conference on Management of Data* (2004), 35–46.
12. Cormode, G., Muthukrishnan, S. An improved data stream summary: The count-min sketch and its applications. *J. Algorithms 55*, 1 (2005), 58–75.
13. Datar, M., Gionis, A., Indyk, P., Motwani, R. Maintaining stream statistics over sliding windows. In *ACM-SIAM Symposium on Discrete Algorithms* (2002).
14. Demaine, E., López-Ortiz, A., Munro, J.I. Frequency estimation of internet packet streams with limited space. In *European Symposium on Algorithms (ESA)* (2002).
15. Fischer, M., Salzburg, S. Finding a majority among *n* votes: Solution to problem 81–5. *J. Algorithms 3*, 4 (1982), 376–379.
16. Gilbert, A.C., Kotidis, Y., Muthukrishnan, S., Strauss, M. How to summarize the universe: Dynamic maintenance of quantiles. In *International Conference on Very Large Data Bases* (2002), 454–465.
17. Jayram, T.S., McGregor, A., Muthukrishnan, S., Vee, E. Estimating statistical aggregates on probabilistic data streams. In *ACM Principles of Database Systems* (2007).
18. Karp, R., Papadimitriou, C., Shenker, S. A simple algorithm for finding frequent elements in sets and bags. *ACM Trans. Database Syst. 28* (2003), 51–55.
19. Manku, G., Motwani, R. Approximate frequency counts over data streams. In *International Conference on Very Large Data Bases* (2002).
20. Manku, G.S. Frequency counts over data streams. http://www.cse.ust.hk/vldb2002/VLDB2002–proceedings/slides/S10P03slides.pdf (2002).
21. Metwally, A., Agrawal, D., Abbadi, A.E. Efficient computation of frequent and top-k elements in data streams. In *International Conference on Database Theory* (2005).
22. Misra, J., Gries, D. Finding repeated elements. *Sci. Comput. Programming 2* (1982), 143–152.
23. Pike, D., Dorward, S., Griesemer, R., Quinlan, S. Interpreting the data: Parallel analysis with sawzall. *Dyn. Grids Worldwide Comput. 13*, 4 (2005), 277–298.
24. Thorup, M., Zhang, Y. Tabulation-based 4-universal hashing with applications to second moment estimation. In *ACM-SIAM Symposium on Discrete Algorithms* (2004).
25. Venkataraman, S., Song, D.X., Gibbons, P.B., Blum, A. New streaming algorithms for fast detection of superspreaders. In *Network and Distributed System Security Symposium NDSS* (2005).

Graham Cormode and Marios Hadjieleftheriou ({graham,marioh}@research.att.com), AT&T Labs—Research, Florham Park, NJ.

© 2009 ACM 0001-0782/09/1000 $10.00

Take Advantage of
ACM's Lifetime Membership Plan!

- **ACM Professional Members** can enjoy the convenience of making a single payment for their entire tenure as an ACM Member, and also be protected from future price increases by taking advantage of **ACM's Lifetime Membership** option.
- **ACM Lifetime Membership** dues may be tax deductible under certain circumstances, so becoming a Lifetime Member can have additional advantages if you act before the end of 2009. (Please consult with your tax advisor.)
- Lifetime Members receive a certificate of recognition suitable for framing, and enjoy all of the benefits of **ACM Professional Membership**.

Learn more and apply at:
http://www.acm.org/life

Association for Computing Machinery

Advancing Computing as a Science & Profession