Efficient Computation of Frequent and Top-k Elements in Data Streams *

Ahmed Metwally **, Divyakant Agrawal, and Amr El Abbadi

Department of Computer Science
University of California, Santa Barbara
{metwally, agrawal, amr}@cs.ucsb.edu

Abstract. We propose an integrated approach for solving both problems of finding the most popular k elements, and finding frequent elements in a data stream. Our technique is efficient and exact if the alphabet under consideration is small. In the more practical large alphabet case, our solution is space efficient and reports both top-k and frequent elements with tight guarantees on errors. For general data distributions, our top-k algorithm can return a set of k' elements, where $k' \approx k$, which are guaranteed to be the top-k' elements; and we use minimal space for calculating frequent elements. For realistic Zipfian data, our space requirement for the frequent elements problem decreases dramatically with the parameter of the distribution; and for top-k queries, we ensure that only the top-k elements, in the correct order, are reported. Our experiments show significant space reductions with no loss in accuracy.

1 Introduction

Recently, online monitoring of data streams has emerged as an important data management problem. This new key research topic has its foundations and applications in many domains, including databases, data mining, algorithms, networking, theory and statistics. However, new challenges have emerged. Due to their vast sizes, some stream types should be mined fast before being deleted forever. Generally, the alphabet is too large to keep exact information for all elements. Conventional database, and mining techniques, though effective with stored data, are deemed impractical in this setting.

This work is primarily motivated by the setting of Internet advertising commissioners, who represent the middle persons between Internet publishers, and Internet advertisers. The file systems are bombarded continuously by streams of various types: advertisement rendering, clicks, sales, and leads; and each type is handled differently. For instance, before rendering an advertisement for a user, the clicks stream summary structure should be queried to determine what advertisements would suit the user's profile. If the user's profile indicates that

* This work was supported in part by NSF under grants EIA 00-80134, NSF 02-09112, and CNF 04-23336.

** Part of this work was done while the first author was at ValueClick, Inc.

(s)he is not a *frequent* "clicker", then this user, most probably, will not click any displayed advertisement. Thus, it can be more profitable to show Pay-Per-Impression advertisements, which generate revenue on rendering them. On the other hand, if the user's profile was found to be one of the *frequent* profiles, then, there is a good chance that this user will click some of the advertisements shown and potentially generate a sale/lead transaction. In this case, Pay-Per-Click advertisements should be displayed. Choosing what advertisements to display entails retrieving the *top* advertisement categories for this specific user profile.

From the above example we are motivated to solve two problems simultaneously. We would like to know if the user's profile is *frequent* in the click stream, and we need to identify the *top* advertisements for this specific profile. That is, we need to solve both the *frequent elements* and the *top-k* problems.

The problems of finding frequent [1] and top-k elements are closely related, yet, to the best of our knowledge, no integrated solution has been proposed. In this paper, we propose an integrated approach for solving both problems of finding the top-k elements, and finding frequent elements in a data stream. Our *Space-Saving* algorithm reports both top-k and frequent elements with tight guarantees on errors. For general data distributions, *Space-Saving* answers top-k queries by returning a set of k' elements, where $k' \approx k$, which are guaranteed to be the top-k' elements; and we use minimal space for calculating frequent elements. For realistic Zipfian data, our space requirement for the frequent elements problem decreases dramatically with the parameter of the distribution; and for top-k queries, we ensure that only the top-k elements, in the correct order, are reported.

The rest of the paper is organized as follows. Section 2 highlights the related work. In Section 3, we introduce our *Space-Saving* algorithm, and its associated data structure, followed by a discussion of query processing in Section 4. We comment on our experimental results in Section 5, and conclude in Section 6.

2 Background and Related Work

Formally, given an alphabet, A, a *frequent element*, E_i, is an element whose frequency, or number of hits, F_i, in a stream S of a given size N, exceeds a user specified support ϕN, where $0 \leq \phi \leq 1$; whereas the *top-k elements* are the k elements with highest frequencies. Since the space requirements for exact solutions of these problems are impractical [3], other relaxations of the original problems were proposed. The FindCandidateTop(S, k, l) problem was proposed in [3] to ask for l elements among which the top-k elements are concealed, with no guarantees on the rank of the remaining $(l-k)$ elements. The FindApproxTop(S, k, ϵ) [3] is a more practical approximation for the top-k problem. The user asks for a list of k elements such that every element, E_i, in the list has $F_i > (1-\epsilon)F_k$, where ϵ is a user-defined error, and $F_1 \geq F_2 \geq \ldots \geq F_{|A|}$, such that E_k is the element with k^{th} rank. The Hot Items [2] problem is a special case of the frequent

[1] The term "Heavy Hitters" was also used in [4].
[2] The term "Hot Items" was coined later in [5].

elements problem, proposed in [13], that asks for k elements, each of which has frequency more than $\frac{N}{k+1}$. This extends the early work done in [2], and [8] for identifying a majority element. The most popular variation of the frequent elements problem, finding the ϵ-Deficient Frequent Elements [11], asks for all the elements with fequency more than $(\phi - \epsilon)N$.

Several algorithms [3], [5], [6], [7], [9], [10], [11] have been proposed to handle the top-k, the frequent elements problems, and their variations. These techniques can be classified into *counter-based*, and *sketch-based* techniques.

Counter-based techniques keep an individual counter for each element in the monitored set, a subset of A. The counter of a monitored element, e_i, is updated when e_i occurs in the stream. If there is no counter kept for the observed ID, it is either disregarded, or some algorithm-dependent action is taken.

For solving the ϵ-Deficient Frequent Elements, algorithms *Sticky Sampling*, and *Lossy Counting* were proposed in [11]. The algorithms cut the stream into rounds. Though simple and intuitive, they suffer from zeroing too many counters at rounds' boundaries, and thus, they free space before it is really needed. In addition, answering a frequent elements query entails scanning all counters.

Demaine *et al.* proposed the *Frequent* algorithm to solve the Hot Items problem in [6]. Their algorithm, a re-discovery of the algorithm in [13], outputs a list of k elements with no guarantee on which elements, if any, have frequency more than $\frac{N}{k+1}$. The same algorithm was proposed independently by Karp *et al.* in [10]. *Frequent* extends the early work done in [2], and [8] for finding a majority item, using only one counter. *Frequent* [6] keeps k counters to monitor k elements. If a monitored element is observed, its counter is incremented, else all counters are decremented. In case any counter reaches 0, it is assigned the next observed element. When the algorithm terminates, the monitored elements are the candidate frequent elements. [6] proposed a lightweight data structure that can decrement all counters in $O(1)$ operations. The sampling algorithm *Probabilistic-InPlace* [6], which is similar to *Sticky Sampling* [11], solves FindCandidateTop(S, k, $\frac{m}{2}$). When queried, the algorithm returns the upper half of the counters, in the hope that they are the correct top-k. Again, the algorithm deletes half the counters at rounds' boundaries, which is $\Omega(|\text{distinct values of the deleted counters}|)$. In general, counter-based techniques exhibit fast per-item processing.

Sketch-based techniques do not monitor a subset of elements, rather provide, with less stringent guarantees, frequency estimation for all elements using bit-maps of counters. Usually, each element is hashed into the space of counters using a family of hash functions, and the hashed-to counters are updated for every hit of this element. Those "representative" counters are then queried for the element frequency with less accuracy, due to hashing collisions.

The *CountSketch* algorithm, proposed in [3], solves the FindApproxTop(S, k, ϵ) problem, with success probability $(1 - \delta)$. Its bottleneck is estimating the frequency of the element by finding the median of its representative counters.

The *GroupTest* algorithm, proposed in [5], answers queries about Hot Items, with a constant probability of failure, δ. A novel algorithm, *FindMajority*, was first devised to detect the majority element, assuming elements' IDs to be

$1 \ldots |A|$. Then *GroupTest*, a probabilistic generalization, was devised that employs several independent copies of *FindMajority*. *GroupTest* is generally accurate. However, its space complexity is large, and it offers no information about elements' frequencies or relative order. The *Multistage filters* approach proposed in [7], which was also independently proposed in [9], is very similar to *GroupTest*.

Sketch-based techniques monitor all elements. However, a hit entails expensive calculations. They do not offer guarantees about relative order or estimated frequencies, and their space usage are not bounded by the size of the alphabet.

3 Summarizing the Data Stream

The algorithms described in Section 2 handle individual problems. The main difficulty in devising an integrated solution is that queries of one type cannot serve as a pre-processing step for the other type of queries. For instance, the frequent elements receiving 1% or more of the total hits might constitute the top-100 elements, some of them, or none. In order to use frequent elements queries to pre-process the stream for a top-k query, several frequent elements queries have to be issued to reach a lower bound on the frequency of the k^{th} element; and in order to use top-k queries to pre-process the stream for a frequent elements query, several top-k queries have to be issued to reach an upper bound on the number of frequent elements. To offer an integrated solution, we have generalized both problems to *accurately estimate the frequencies of significant* [3] *elements, and store these frequencies in an always-sorted structure*. We, then, devise a generalized algorithm for the generalized problem.

3.1 The *Space-Saving* Algorithm

In this section, we propose our counter-based *Space-Saving* algorithm and its associated *Stream-Summary* data structure. The underlying idea is to maintain partial information of interest; i.e., we monitor only m elements. We update the counters in a way that accurately estimates the frequencies of the significant elements, and we use a lightweight data structure that keeps the elements sorted by their estimated frequencies. In an ideal situation, any significant element, E_i, with rank i, that has received F_i hits, should be accommodated in the i^{th} counter. However, due to errors in estimating the frequencies of the elements, the order of the elements in the data structure might not reflect their exact ranks. For this reason, we will denote the counter at the i^{th} position in the data structure $count_i$. The counter $count_i$ estimates the frequency f_i, of some element e_i. If the i^{th} position in the data structure has the right element, i.e., the element with the i^{th} rank, E_i, then $e_i = E_i$, and $count_i$ is an estimation of F_i.

The algorithm is straightforward. If we observe an element, e, that is monitored, we just increment its counter. If e is not monitored, give it the benefit of

[3] The significant elements are interesting elements that can be output in meaningful queries about top-k or frequent elements.

doubt, and replace e_m, the element that currently has the least estimated hits, min, with e. Assign $count_m$ the value $min + 1$. For each monitored element e_i, we keep track of its over-estimation, ε_i, resulting from the initialization of its counter when it was inserted into the list. That is, when starting to monitor e_i, set ε_i to the value of the evicted counter. The algorithm is depicted in Figure 1.

```
Algorithm: Space-Saving(m counters, stream S)
begin
  for each element, e, in S{
    If e is monitored,
      increment the counter of e;
    else{
      let e_m be the element with least hits, min
      Replace e_m with e;
      Increment count_m;
      Assign ε_m the value min;
    }
  }// end for
end;
```

Fig. 1. The *Space-Saving* Algorithm

In general, the top elements among non-skewed data are of no great significance. Hence, we concentrate on skewed datasets. The basic intuition is to make use of the skewed property of the data, since we expect a minority of the elements, the more frequent ones, to get the majority of the hits. Frequent elements will reside in the counters of bigger values, and will not be distorted by the ineffective hits of the infrequent elements, and thus, will never be replaced out of the monitored counters. Meanwhile, the numerous infrequent elements will be striving to reside on the smaller counters, whose values will grow slower than those of the larger counters. In addition, if the skew remains, but the popular elements change overtime, the algorithm adapts automatically. The elements that are growing more popular will gradually be pushed to the top of the list as they receive more hits. If one of the previously popular elements lost its popularity, it will receive less hits. Thus, its relative position will decline, as other counters get incremented, and it might eventually get dropped from the list.

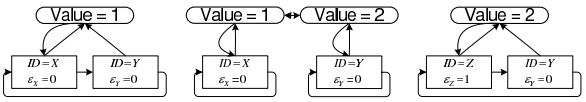
Even if the data is not skewed, the errors in the counters will be inversely proportional to the number of counters, as shown later. Keeping only a moderate number of counters will guarantee very small errors. This is because the more counters we keep, the less it is probable to replace elements, and thus, the smaller the over-estimation errors in counters' values.

To implement this algorithm, we need a data structure that cheaply increments counters without violating their order, and that ensures constant time retrieval. We propose the *Stream-Summary* data structure for these purposes.

In a *Stream-Summary*, all elements with the same counter value are linked together in a linked list. They all point to a parent bucket. The value of the parent bucket is the same as the counters' value of all of its elements. Every

bucket points to exactly one element among its child list, and buckets are kept in a doubly linked list, sorted by their values. Initially, all counters are empty, and are attached to a single parent bucket with value 0. The elements can be stored in a hash table for constant amortized access cost, or in an associative memory for constant worst case access cost. The *Stream-Summary* can be sequentially traversed as a sorted list, since the buckets' list is sorted. In case it is feasible to keep counters for all elements, *Stream-Summary* can be used to report both the most and the least significant elements. Reporting the least significant elements can be useful in some contexts, but it is beyond the scope of this paper.

The algorithm for counting elements' hits using *Stream-Summary* is straightforward. When an element's counter is updated, its bucket's neighbor with the larger value is checked. If it has a value equal to the new value of the element, then the element is detached from its current list, and is inserted in the child list of this neighbor. Otherwise, a new bucket with the correct value is created, and is attached to the bucket list in the right position; and this element is attached to this new bucket. The old bucket is deleted if it points to an empty child list. The worst case scenario costs 10 pointer assignments, and one heap operation.



(a) *Stream-Summary*, $S = X, Y$

(b) *Stream-Summary*, $S = X, Y, Y$

(c) *Stream-Summary*, $S = X, Y, Y, Z$

Fig. 2. *Space-Saving* updates to a *Stream-Summary* data structure as elements are observed.

Example 1. Assuming $m = 2$, and $A = \{X, Y, Z\}$. The stream $S = X, Y$ will yield the *Stream-Summary* in Figure 2(a), after the two counters accommodate the observed elements. When another Y arrives, a new bucket is created with value 2, and Y gets attached to it, as shown in Figure 2(b). When Z arrives, the element with the minimum counter, X, is replaced by Z. Z has $\varepsilon_Z = 1$, since that was the evicted counter. The final *Stream-Summary* is shown in Figure 2(c).

Stream-Summary is motivated by the work done in [6]. However, to look up a value of a counter using the data structure in [6], it takes $O(m)$, while *Stream-Summary* looks values up in $\Theta(1)$, for online queries about specific elements.

3.2 Properties of the *Space-Saving* Algorithm

To prove the space bounds in Section 4, we analyze some properties of *Space-Saving*, which will help us establish our space bounds. For space limitations, all proofs are omitted, and the reader is referred to the full version [12].

The strength behind our simple algorithm is that we keep the information until the space is absolutely needed, and we do not initialize counters in batches like other counter-based algorithms. This is what allowed us to prove these properties about the proposed algorithm.

A pivotal factor in our analysis is the value of min. The value of min is highly dynamic since it is dependent on the permutation of elements in S. We give an illustrative example. If $m = 2$, and $N = 4$. $S = X, Z, Y, Y$ yields $min = 1$, while $S = X, Y, Y, Z$ yields $min = 2$. Although it would be very useful to quantify min, we do not want to involve the order in which hits were received in our analysis, because predicating the analysis on all possible stream permutations will be intractable. Thus, we establish an upper bound on min.

Lemma 1. *The minimum counter value, min, is less than or equal to* $\lfloor \frac{N}{m} \rfloor$.

We assume that the number of distinct elements in S to be more than m. Thus, all m counters are occupied. Otherwise, all counts are exact. We are interested in min since it represents an upper bound on the over-estimation in any counter in *Stream-Summary*. Moreover, any element e_i, with frequency $f_i > min$, is guaranteed to be monitored, as shown next.

Theorem 1. *An element* e_i *with* $f_i > min$, *must exist in the* Stream-Summary.

We can infer an interesting general rule about the over-estimation of elements' counters. For any element E_i, with rank $i \leq m$, the frequency F_i, is no more than $count_i$, the counter occupying the i^{th} position in the *Stream-Summary*.

Theorem 2. *Whether or not* E_i *occupies the* i^{th} *position,* $count_i \geq F_i$.

4 Processing Queries

In this section, we discuss query processing using the *Stream-Summary* data structure. We also analyze the space requirements for both the general case, where no data distribution is assumed, and the more interesting Zipfian case.

4.1 Frequent Elements

In order to answer queries about the frequent elements, we sequentially traverse *Stream-Summary* as a sorted list until an element with frequency less than the user support is reached. Thus, we report frequent elements in $\Theta(|\text{frequent elements}|)$. If for each reported element e_i, $count_i - \varepsilon_i > \phi N$, then the algorithm **guarantees that all, and only the frequent elements** are reported. This guarantee is conveyed through the boolean parameter `guaranteed`. The number of counters, m, should be specified by the user according to the data properties, the required error rate and/or the available memory on the server. The *QueryFrequent* algorithm is given in Figure 3.

```
Algorithm: QueryFrequent(m counters, support φ)
begin
  Bool guaranteed = true;
  Integer i = 1;
  while (count_i > φN AND i ≤ m){
    output e_i;
    If ((count_i − ε_i) < φN)
      guaranteed = false;
    i++;
  }// end while
  return( guaranteed )
end;
```

Fig. 3. Reporting Frequent Elements

The General Case. We will analyze the space requirements for the general case of the data distribution.

Theorem 3. *Assuming no specific data distribution, or user-supplied support, to find all frequent elements with error* ϵ*,* Space-Saving *uses a number of counters that is bounded by* $\min(|A|, \frac{1}{\epsilon})$*. Any element,* e_i*, with frequency* $f_i > \epsilon N$ *is guaranteed to be in the* Stream-Summary*.*

Zipf Distribution Analysis. Assuming Zipfian data [14], with parameter α, $F_i = \frac{N}{i^\alpha \zeta(\alpha)}$, where $\zeta(\alpha) = \sum_{i=1}^{|A|} \frac{1}{i^\alpha}$ converges to a small constant inversely proportional to α, except for $\alpha \leq 1$. For instance, $\zeta(1) \approx \ln(1.78|A|)$. We assume $\alpha \geq 1$, to ensure that the data is worth analyzing. As noted before, we do not expect the popular elements to be of great importance if the data is weakly skewed.

Theorem 4. *Assuming Zipfian data with parameter* α*, to calculate the frequent elements with error rate* ϵ*,* Space-Saving *uses only* $\min(|A|, \left(\frac{1}{\epsilon}\right)^{\frac{1}{\alpha}}, \frac{1}{\epsilon})$ *counters. This is regardless of the stream permutation.*

Having established the bounds of *Space-Saving* for both the general, and the Zipf distributions, we compare them to other algorithms. We also comment on some practical issues, that can not be directly inferred from the bounds.

Comparison with Similar Work. The above bounds are better than those guaranteed in [11]. *Sticky Sampling* has a bound of $\frac{2}{\epsilon}\log(\frac{1}{\phi\delta})$. *Lossy Counting* has a bound of $\frac{1}{\epsilon}\log(\epsilon N)$. Furthermore, *Space-Saving* has a better bound than *GroupTest* [5], whose bound is $O(\frac{1}{\phi}\log(\frac{1}{\delta\phi})\log(|A|))$, which is less scalable than ours. For example, for $N = 10^6$, $|A| = 10^4$, $\phi = 10^{-1}$, $\epsilon = 10^{-2}$, and $\delta = 10^{-1}$, we need 100 counters, *Sticky Sampling* needs 700 counters, *Lossy Counting* needs 1000 counters, and *GroupTest* needs $C * 930$ counters, where $C \geq 1$.

Frequent [6] has a similar bound in the general case. Using m counters, the elements' under-estimation error is bounded by $\frac{N-1}{m}$. Although this is close to the

theoretical under-estimation error bound, as proved in [1], there is no straightforward feasible extension of the algorithm to track the under-estimation error for each counter. In addition, every observation of a non-monitored element increases the errors for all the monitored elements, since their counters get decremented. Therefore, elements of higher frequency are more error prone, and thus, it is still difficult to guess the frequent elements, which is not the case for *Space-Saving*. Even more, the structure proposed in [6] is built and queried in a way that does not allow the user to specify an error threshold, ϵ. Thus, the algorithm has only one parameter, the support ϕ, which increases the number of false positives dramatically, as will be clear from the experimental results in Section 5.

The number of counters used in *GroupTest* [5] depends on the failure probability, δ, as well as the support, ϕ. Thus, it does not suffer from the single-threshold drawback of *Frequent*. However, it does not output frequencies at all, and reveals nothing about the relative order of the elements. In addition, its assumption that elememts' IDs are $1 \dots |A|$ can only be enforced by building an indexed lookup table that maps every ID to a unique number in the range $1 \dots |A|$. Thus, in practice , *GroupTest* needs $O(|A|)$ space, which is infeasible in most cases. Meanwhile, we only require the m IDs to fit in memory.

For the Zipfian case, we make no comparison to other works, since we are not aware of a similar analysis. For the numerical example given above, if $\alpha = 2$, we would need only 10 counters, instead of 100, to guarantee the same error.

4.2 Top-k Elements

For the top-k elements, the algorithm can output the first k elements. An element, e_i, is **guaranteed to be among the top-k** if its guaranteed number of hits, $count_i - \varepsilon_i$, exceeds $count_{k+1}$, the over-estimated number of hits for the element in position $k+1$. We call the results to have **guaranteed top-k** if by looking at the results only, we can tell that the reported top-k elements are correct. *Space-Saving* reports a guaranteed top-k if $\forall_{i \leq k}$, $count_i - \varepsilon_i \geq count_{k+1}$. That is, all the reported k elements are guaranteed to be among the top-k.

All guaranteed top-i subsets, for all i, can be reported in $\Theta(m)$, by iterating on all the counters $1 \dots m-1$. At each iteration, i, the $\min_{\forall_{j \leq i}}(count_j - \varepsilon_j)$ is compared to $count_{i+1}$. The first i elements are guaranteed to be the top-i elements if this minimum is no smaller than $count_{i+1}$. The algorithm guarantees the top-m if in addition to this condition, $\varepsilon_m = 0$; which is only true if the number of distinct elements in the stream is at most m.

Similarly, we call the top-k to have **guaranteed order** if $\forall_{i \leq k}$, $count_i - \varepsilon_i \geq count_{i+1}$. That is, in addition to having guaranteed top-k, the order of elements among the top-k elements are guaranteed to hold, if the guaranteed hits for every element in the top-k are more than the over-estimated hits of the next element. Thus, the order is guaranteed if the algorithm guarantees the top-i, $\forall_{i \leq k}$.

This is the first algorithm that can give guarantees about its output. For top-k queries, we can guarantee which reported elements are among the top-k. Even if we cannot guarantee all the top-k elements, we can guarantee top-k'

elements, where $k' \approx k$. For the case of Zipfian data, we guarantee that $k' = k$, as shown later in the section. The algorithm *QueryTop-k* is given in Figure 4.

```
Algorithm: QueryTop-k(m counters, Integer k)
begin
  Bool order = true;
  Bool guaranteed = false;
  Integer min-guar-freq = ∞;
  for i = 1...k{
    output e_i;
    If ((count_i − ε_i) < min-guar-freq)
      min-guar-freq = (count_i − ε_i);
    If ((count_i − ε_i) < count_{i+1})
      order = false;
  }// end for
  If (count_{k+1} ≤ min-guar-freq){
    guaranteed = true;
  }else{
    output e_{k+1};
    for i = k + 2...m{
      If ((count_{i−1} − ε_{i−1}) < min-guar-freq)
        min-guar-freq = (count_{i−1} − ε_{i−1});
      If (count_i ≤ min-guar-freq){
        guaranteed = true;
        break;
      }
      output e_i;
    }
  }
  return( guaranteed, order )
end;
```

Fig. 4. Reporting Top-k

The algorithm consists of two loops. The first loop outputs the top-k candidates. At each iteration the order of the elements reported so far is checked. If the order is violated, `order` is set to false. At the end of the loop, the top-k candidates are checked to be the guaranteed top-k, by checking that all of these candidates have guaranteed hits that exceed the overestimated counter of the $k+1$ element, $count_{k+1}$. If this does not hold, the second loop is executed to search for the next k', where $k' \approx k$, such that the top-k' are guaranteed.

The algorithm can also be implemented in a way that only outputs the first k elements, or that outputs k' elements, such that k' is the closest possible to k, regardless of whether k' is greater than k, or vice versa. Throughout the rest of the paper, we assume that the algorithm outputs only the first k elements. Next, we look at the space requirements for solving this problem.

The General Case. We start by considering data which is not as skewed as Zipf of parameter 1. We deal with skewed data later. We also restrict the discussion

to the relaxed version, FindApproxTop(S, k, ϵ) [3], which is finding a list of k elements, each of which has frequency more than $(1-\epsilon)F_k$.

Theorem 5. *Regardless of the data distribution, to solve the FindApproxTop(S, k, ϵ) problem,* Space-Saving *uses* $\min(|A|, \frac{N}{\epsilon F_k})$ *counters. Any element with frequency more than* $(1-\epsilon)F_k$ *is guaranteed to be monitored.*

For the general case, we were not able to solve the exact problem, and we only considered a relaxed version, since it is widely accepted that solving the exact problem requires $\Theta(|A|)$ space [3].

Zipf Distribution Analysis. To answer exact top-k queries, ϵ can be automatically set less than $F_k - F_{k+1}$. Thus, we guarantee correctness, and order.

Theorem 6. *Assuming the data is Zipfian with parameter* $\alpha > 1$*, to calculate the exact top-k,* Space-Saving *uses* $\min(|A|, O((\frac{k}{\alpha})^{\frac{1}{\alpha}}k))$ *counters. When* $\alpha = 1$*, the space complexity is* $\min(|A|, O(k^2 \log(|A|)))$*. This is regardless of the stream permutation. Also, the order among the top-k elements is preserved.*

To the best of our knowledge, this is the first work to look at the space bounds for solving the exact problem, in the case of Zipfian data, with guaranteed results. Having established the bounds of *Space-Saving* for both the general, and the Zipf distributions, we compare these bounds to other algorithms.

Comparison with Similar Work. These bounds are better than the bounds guaranteed by the best known algorithm, *CountSketch* [3], for a large range of practical values of the parameters $|A|$, ϵ, and k. *CountSketch* solves the relaxed version of the problem, FindApproxTop(S, k, ϵ), with failure probability δ, using space of $O(\log(\frac{N}{\delta})(k + \frac{1}{(\epsilon F_k)^2}\sum_{i=k+1}^{|A|} {F_i}^2))$, with a large constant hidden in the big-O notation, according to [3], and [5]. The bound of *Space-Saving* for the relaxed problem is $\frac{N}{\epsilon F_k}$, with a 0-failure probability. For instance, for $N = 10^6$, $|A| = 10^4$, $k = 100$, and $\epsilon = \delta = 10^{-1}$, and a uniformly distributed data, we require 10^3 counters, while *CountSketch* needs $C * 2.3 * 10^7$ counters, where $C \gg 1$, which is more than the entire stream. In addition, *Space-Saving* guarantees that any element, e_i, whose $f_i > (1-\epsilon)F_k$ belongs to the *Stream-Summary*, and does not simply output a random k selection of these elements.

In case of a non-Zipf distribution, or a weakly skewed Zipf distribution with $\alpha < 1$, for all $i \geq k$, we will assume that $F_i \geq \frac{N}{\zeta(1)} * \frac{1}{i}$. This assumption is justified. Since we are assuming a non-skewed distribution, the top few elements have a less significant share in the stream than in the case of Zipf(1), and thus, less frequent elements will have a larger share. Using this assumption, we rewrite the bound of *Space-Saving* as $O(\frac{k*\log(N)}{\epsilon})$; while the bound in [3] can be rewritten as $O(\log(\frac{N}{\delta}) * (k + \frac{k^2}{\epsilon^2}\left(\frac{1}{k+1} - \frac{1}{|A|}\right))) \approx O(\frac{k}{\epsilon^2}\log(\frac{N}{\delta}))$. Even more, depending

on the data distribution, *Space-Saving* can guarantee the reported top-k, or a subset of them, to be correct; while *CountSketch* does not offer any guarantees.

We can assume that field experts know whether the data is skewed enough to be considered Zipf(1) or not. Even if this is not applicable, we can start by analyzing a sample from the data, and then re-size the structure accordingly.

In the case of Zipf Distribution, the bound of [3] is $O(k \log(\frac{N}{\delta}))$. For $\alpha > 1$, the bound of *Space-Saving* is $O((\frac{k}{\alpha})^{\frac{1}{\alpha}} k)$. Only when $\alpha = 1$, the space complexity is $O(k^2 \log(|A|))$, and thus, *Space-Saving* is better for cases of skewed data, long streams/windows, and has a 0-failure probability. In addition, we preserve the order of the top-k elements. For example, for $N = 10^6$, $|A| = 10^4$, $k = 10$, $\alpha = 2$, and $\delta = 10^{-1}$ our space requirements are only 66 counters, while [3] needs $C*230$ counters, where $C \gg 1$.

5 Experimental Results

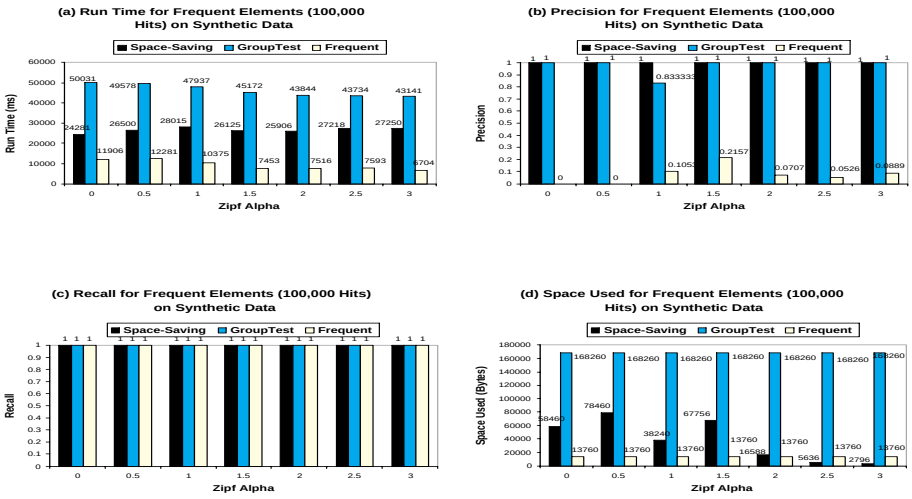


Fig. 5. Performance Comparison for the Frequent Elements Problem Using Synthetic Zipfian Data

We conducted a set of experiments, using both real and synthetic data. For space constraints, we summarize our synthetic data results here. The real data experimental results agree with those presented here, and the reader is referred to [12] for a full analysis on both the synthetic and real data experimental results. We generated several synthetic Zipfian datasets with the zipf parameter, α, varying from 0, which is uniform, to 3, which is highly skewed, on a fixed

interval of $\frac{1}{2}$. The size of each dataset, N, is 10^7 hits. This set of experiments measure how the algorithms adapt to, and make use of data skew.

The algorithms were run on a Pentium IV 2.66 GHz, with 1.0 GB RAM. The stream was input and processed by each algorithm, and then a query was issued, and we recorded the *recall*, the number of correct elements found as a percentage of the number of actual correct elements; and the *precision*, the number of correct elements found as a percentage of the entire output [5]. We also measured the run time and space used by each algorithm, which indicates the capability to deal with high-speed streams, and to reside on servers with limited memories.

For the frequent elements problem, we compared our results with those of *GroupTest* [5], and *Frequent* [6]. For *GroupTest*, and *Frequent*, we used the C code available on the web-site of the first author of [5]. *Space-Saving*, *GroupTest*, and *Frequent* were queried for the frequent elements with support, ϕ, of 10^{-2}. We set ϵ, the error, to be one hundredth of ϕ, the required support; and δ, the failing probability, to be 0.01. Although *Frequent* ran up to four times faster than *Space-Saving*, as clear from Figure 5(a), its results were not competitive in terms of precision. Since it is not possible to specify an ϵ parameter for the algorithm, its precision was very low in all the runs. When the Zipf parameter was 0.0, and 0.5, the algorithm reported 28, and 19 elements, respectively, and actually there were no elements satisfying the support. For the rest of the experiments in Figure 5(b), the precision achieved by *Frequent* ranged from 0.053 to 0.216. The space used ranged from one fifth to five times the space of *Space-Saving*, as shown in Figure 5(d). It is interesting to note that as the data became more skewed, the space advantage of *Space-Saving* increased, while *Frequent* was not able to exploit the data skew to reduce its space requirements. Compared to *GroupTest*, from Figure 5(a), *Space-Saving* ran 1.5 to 2 times faster than *GroupTest*. The precision of the proposed algorithm was always 1; while *GroupTest* precision depended on α, with a precision of 0.83 when $\alpha = 1$, as sketched in Figure 5(b). The recalls of both algorithms were constant at 1, as clear from Figure 5(c). The advantage of *Space-Saving* is clear in Figure 5(d), which shows that *Space-Saving* achieved a reduction in the space used by a factor ranging from 2 up to 60.

For the top-k problem, we implemented *Probabilistic-InPlace* [6], and *CountSketch* [3]. The *Space-Saving*, *CountSketch*, and *Probabilistic-InPlace* algorithms were used to identify the top-100 elements. For *CountSketch*, we set the probability of failure, δ, to 0.01. Both the *Space-Saving*, and the *Probabilistic-InPlace* were allowed the same number of counters; and thus, their run time and space usages were comparable, as clear from Figure 6(a), and Figure 6(d), respectively. The precision of *Probabilistic-InPlace* increased from 0.02 to 0.36 as the skew increased; and finally reached 1, when $\alpha \geq 2.5$, as indicated in Figure 6(b). On the contrary, from Figure 6(c), the recall of *Probabilistic-InPlace* was very high throughout the entire range of α. The precision and recall of *Space-Saving* were constant at 1. From Figure 6(a), the time reductions of *Space-Saving* over *CountSketch* ranged from a factor of 30, to 82. Although we used a hidden factor of 16, as indicated earlier, *CountSketch* failed to attain a recall and precision of 1, for all the experiments. *CountSketch* had a very low precision and recall

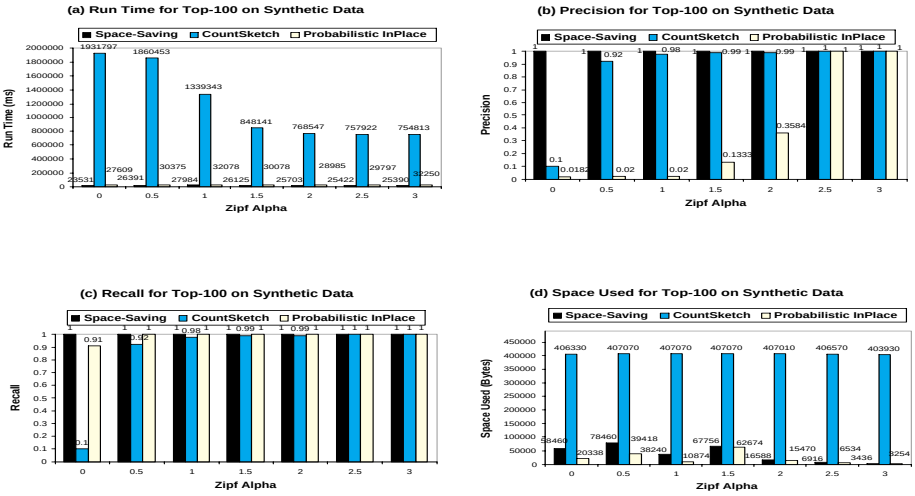


Fig. 6. Performance Comparison for the Top-k Problem Using Synthetic Zipfian Data

for uniform data. From Figure 6(b), and Figure 6(c), the precision and recall of *CountSketch* did not reach 1 except for $\alpha \geq 2.5$. The space reductions of *Space-Saving* ranged from a factor of 5, to 117, as manifested in Figure 6(d). The performance gap increased with the data skew.

6 Discussion

This paper has devised an integrated approach for solving an interesting family of problems in data streams. The *Stream-Summary* data structure was proposed, and utilized by the *Space-Saving* algorithm to guarantee strict error bounds for approximate counts of elements, using very limited space. We showed that *Space-Saving* can handle both the frequent elements and top-k problems because it efficiently estimates the elements' frequencies. The memory requirements were analyzed with special attention to the case of skewed data. We validated the theoretical analysis by experimental evaluation.

This is the first algorithm, to the best of our knowledge, that guarantees the order of the top-k elements. Even when it cannot guarantee the top-k, the algorithm outputs guaranteed top-k' elements, where $k' \approx k$.

In practice, if the alphabet is too large, like in the case of IP addresses, only a subset of this alphabet is observed in the stream, and not all the 2^{32} addresses. Our space bounds are actually a function of the number of distinct elements observed in the stream. However, in our analysis, we have assumed that the entire alphabet is observed in the stream, which is the worst case for *Space-Saving*. Yet, our space bounds are still better than those of other algorithms.

The main practical strengths of *Space-Saving* is that it can use whatever space is available on the server to estimate the elements' frequencies, and provide guarantees on its results whenever possible. Even when analysts are not sure about the appropriate parameters, the algorithm can run in the available memory, and the results can be analyzed for further adaptation. It is interesting that running the algorithm on the available space ensures that more important elements are less susceptible to noise.

References

1. P. Bose, E. Kranakis, P. Morin, and Y. Tang. Bounds for Frequency Estimation of Packet Streams. In *Proceedings of the 10th International Colloquium on Structural Information and Communication Complexity*, pages 33–42, 2003.
2. R. Boyer and J. Moore. A Fast Majority Vote Algorithm. Technical Report 1981-32, Institute for Computing Science, University of Texas, Austin, February 1981.
3. M. Charikar, K. Chen, and M. Farach-Colton. Finding Frequent Items in Data Streams. In *Proceedings of the 29th International Colloquium on Automata, Languages and Programming*, pages 693–703. Springer-Verlag, 2002.
4. G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava. Finding Hierarchical Heavy Hitters in Data Streams. In *Proceedings of the 29th International Conference on Very Large Data Bases*, pages 464–475, 2003.
5. G. Cormode and S. Muthukrishnan. Whats Hot and Whats Not: Tracking Most Frequent Items Dynamically. In *Proceedings of the 22nd Symposium on Principles of Databse Systems*, pages 296–306, June 2003.
6. E. D. Demaine, A. Lpez-Ortiz, and J. I. Munro. Frequency Estimation of Internet Packet Streams with Limited Space. In *Proceedings of the 10th Annual European Symposium on Algorithms*, pages 348–360, 2002.
7. C. Estan and G. Varghese. New Directions in Traffic Measurement and Accounting: Focusing on the Elephants, Ignoring the Mice. *ACM Trans. Comput. Syst.*, 21(3):270–313, 2003.
8. M. J. Fischer and S. L. Salzberg. Finding a Majority Among N Votes: Solution to Problem 81-5. *Journal of Algorithms*, 3:376–379, 1982.
9. C. Jin, W. Qian, C. Sha, J. X. Yu, and A. Zhou. Dynamically Maintaining Frequent Items over A Data Stream. In *Proceedings of the Twelfth International Conference on Information and Knowledge Management*, pages 287–294. ACM Press, 2003.
10. R. Karp, S. Shenker, and C. Papadimitriou. A Simple Algorithm for Finding Frequent Elements in Streams and Bags. *ACM Transactions on Database Systems*, 28(1):51–55, 2003.
11. G. Manku and R. Motwani. Approximate Frequency Counts over Data Streams. In *Proceedings of the 28th International Conference on Very Large Data Bases*, pages 346–357, 2002.
12. A. Metwally, D. Agrawal, and A. El Abbadi. Efficient Computation of Frequent and Top-k Elements in Data Streams. Technical Report 2005-23, University of California, Santa Barbara, September 2005.
13. J. Misra and D. Gries. Finding Repeated Elements. *Science of Computer Programming*, 2:143–152, November 1982.
14. G.K. Zipf. *Human Behavior and The Principle of Least Effort.* Addison-Wesley, 1949.