An Integrated Efficient Solution for Computing Frequent and Top-*k* Elements in Data Streams

AHMED METWALLY, DIVYAKANT AGRAWAL, and AMR EL ABBADI
University of California, Santa Barbara

We propose an approximate integrated approach for solving both problems of finding the most popular k elements, and finding frequent elements in a data stream coming from a large domain. Our solution is space efficient and reports both frequent and top-k elements with tight guarantees on errors. For general data distributions, our top-k algorithm returns k elements that have roughly the highest frequencies; and it uses limited space for calculating frequent elements. For realistic Zipfian data, the space requirement of the proposed algorithm for solving the exact frequent elements problem decreases dramatically with the parameter of the distribution; and for top-k queries, the analysis ensures that only the top-k elements, in the correct order, are reported. The experiments, using real and synthetic data sets, show space reductions with hardly any loss in accuracy. Having proved the effectiveness of the proposed approach through both analysis and experiments, we extend it to be able to answer continuous queries about frequent and top-k elements. Although the problems of incremental reporting of frequent and top-k elements are useful in many applications, to the best of our knowledge, no solution has been proposed.

Categories and Subject Descriptors: H.2.8 [**Database Management**]: Database Applications—*Data mining*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems; C.2.3 [**Computer-Communication Networks**]: Network Operations—*Network monitoring*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Data streams, advertising networks, frequent elements, top-k elements, Zipfian data, exact queries, continuous queries, approximate queries

1. INTRODUCTION

More than a decade ago, both industry and research communities realized the benefit of statistically analyzing vast amounts of historical data in order to

This work was supported in part by the NSF under grants EIA 00-80134, NSF 02-9112, and CNF 04-23336.
Part of this work was done while A. Metwally was at ValueClick, Inc.
Authors' addresses: Department of Computer Science, University of California, Santa Barbara, Santa Barbara, CA 93106; email: {mewally,agrawal,amr}@cs.ucsb.edu.

discover useful information. Data mining emerged as a very active research field that offered scalable data analysis techniques for large volumes of historical data. Data mining, a well-established key research area, has its foundations and applications in many domains, including databases, algorithms, networking, theory, and statistics.

However, new challenges have emerged as the data acquisition technology has evolved aggressively. For some applications, data is being generated at a rate high enough to make its long-term storage cost outweigh its benefits. Hence, such streams of data are stored temporarily, and should be mined quickly before they are lost forever. The data mining community adapted by devising novel approximate stream handling algorithms that incrementally analyze arriving data in one pass, answer approximate queries, and store summaries for future usage [Babcock et al. 2002].

There is a growing need to develop new techniques to cope with high-speed streams and to answer online queries. Currently, data stream management systems are used for monitoring click streams [Gunduz and Ozsu 2003], stock tickers [Chen et al. 2000; Zhu and Shasha 2002], sensor readings [Bonnet et al. 2001], telephone call records [Cortes et al. 2000], network packet traces [Demaine et al. 2002], auction bidding patterns [Arasu et al. 2003a], traffic management [Arasu et al. 2003b], network-aware clustering [Cormode et al. 2003], and security against DoS [Cormode et al. 2003]. Golab and Ozsu [2003] reviewed the literature.

Complying with this restricted environment, and motivated by the above applications, researchers started working on novel algorithms for analyzing data streams. Problems studied in this context include approximate frequency moments [Alon et al. 1996], differences [Feigenbaum et al. 1999], distinct values estimation [Flajolet and Martin 1985; Haas et al. 1995; Whang et al. 1990], bit counting [Datar et al. 2002], duplicate detection [Metwally et al. 2005a], approximate quantiles [Greenwald and Khanna 2001; Lin et al. 2004; Manku et al. 1999], histograms [Guha et al. 2001, 2002], wavelet-based aggregate queries [Gilbert et al. 2001; Matias et al. 2000], correlated aggregate queries [Gehrke et al. 2001], elements classification [Gupta and McKeown 1999], frequent elements [Bose et al. 2003; Cormode et al. 2004; Cormode and Muthukrishnan 2003; Demaine et al. 2002; Estan and Varghese 2003; Fang et al. 1998; Golab et al. 2003; Jin et al. 2003; Karp et al. 2003; Manku and Motwani 2002; Metwally et al. 2005b], and top-k queries [Babcock and Olston 2003; Charikar et al. 2002; Demaine et al. 2002; Gibbons and Matias 1998; Metwally et al. 2005b]. Earlier results on data streams were presented in Boyer and Moore [1981] and Fischer and Salzberg [1982].

This work is primarily motivated by the setting of Internet advertising. As the Internet continues to grow, the Internet advertising industry flourishes as a means of reaching focused market segments. The main coordinators in this setting are the Internet advertising commissioners, who are positioned as the brokers between Internet publishers and Internet advertisers. In a standard setting, an advertiser provides the advertising commissioner with its advertisements, and they agree on a commission for each action, for example, an impression (advertisement rendering) to a Web surfer, clicking an advertisement,

bidding in an auction, or making a sale. The publishers, being motivated by the commission paid by the advertisers, contract with the commissioner to display advertisements on their Web sites. Every time a surfer visits a publisher's Web page, after loading the page on the surfer's browser, the publisher's Web page has a script that refers the browser to the commissioner's server, which loads the advertisements and logs the advertisement impression. Whenever a Web surfer clicks an advertisement on a publisher's Web page, the surfer is referred again to the servers of the commissioner, who logs the click for accounting purposes, and *clicks-through* the surfer to the Web site of the advertiser, which loads its own Web page on the surfer's Browser. A commissioner earns a commission on the advertisers' payments to the publishers. Therefore, commissioners are generally motivated to show advertisements on publishers' Web pages that would maximize publishers' earnings. To achieve this goal, the commissioners have to analyze the traffic, and make use of prevalent trends. One way to optimize the rendering of advertisements is to show the right advertisements for the right type of surfers.

Since publishers prefer to be paid according to the advertising load on their servers, there are two main types of paying publishers, Pay-Per-Impression, and Pay-Per-Click. The revenue generated by Pay-Per-Impression advertisements is proportional to the number of times the advertisements are rendered. On the other hand, rendering Pay-Per-Click advertisements does not generate any revenue. They generate revenue according to the number of times Web surfers click them. On average, one click on a Pay-Per-Click advertisement generates 500 times as much revenue as Pay-Per-Impression renderings. Hence, to maximize the revenue of impressions and clicks, the commissioners should render a Pay-Per-Click advertisement when it is expected to be clicked. Otherwise, it should use the chance to display a Pay-Per-Impression advertisement that will generate small but guaranteed revenue.

To know when advertisements are more likely to be clicked, the commissioner has to know whether the surfer, to whom the advertisement is displayed, is a *frequent* "clicker" or not. To identify surfers, commissioners assign unique IDs in cookies set in the surfers' browsers. Before rendering an advertisement for a surfer, the summarization of the clicks stream should be queried to see if the surfer is a *frequent* "clicker" or not. If the surfer's is not found to be among the *frequent* "clickers," then (s)he will probably not click any displayed advertisement. Thus, it can be more profitable to show Pay-Per-Impression advertisements. On the other hand, if the surfer is found to be one of the *frequent* profiles, then, there is a good chance that (s)he will click some of the advertisements shown. In this case, Pay-Per-Click advertisements should be displayed. Keeping in mind the limited number of advertisements that can be displayed on a Web page, choosing what advertisements to display entails retrieving the *top* advertisements in terms of clicking.

This is one scenario that motivates solving two famous problems simultaneously. The commissioner should be able to query the click stream for frequent users and top-k advertisements before every impression. Exact queries about frequent and top-k elements are not scalable enough to handle this problem. An average-sized commissioner has around 120M unique monthly surfers, 50, 000

publisher sites, and 30, 000 advertisers' campaigns, each of which has numerous advertisements. Storing only the unique IDs assigned to the surfers requires 2 to 8 GB of main memory, since the IDs used are between 128 and 512 bits.

The size of the motivating problem poses challenges for answering exact queries about *frequent* and *top-k* elements in streams. The domains under consideration are too large to keep track of the exact frequencies of all the surfers or the advertisements. This motivated us to devise an approximate integrated approach for solving both problems. Approximately solving the queries would require less space than solving the queries exactly, and hence would be more feasible. However, the traffic rate entails performing an update and a query every 50 μs, since an average-sized commissioner receives around 70M records every hour. The already existing approximate solutions for frequent and top-k elements could be relatively slow for online decision making. To allow for online decisions on what advertisements should be displayed, we propose that the commissioner should keep a cache of the frequent users and the top-k advertisements. The set of frequent users and the top-k advertisements can change after every impression, depending on how the user reacts to the advertisements displayed. Therefore, the cache has to be updated efficiently after every user response to an impression. We propose updating the cache only whenever necessary. That is, the cache should serve as a materialization of the queries' answer sets, which is updated continuously.

The problems of approximately 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 article, we propose an integrated online streaming algorithm, called *Space-Saving*, for solving both the problem of finding the top-k elements and that of finding frequent elements in a data stream. Our *Space-Saving* algorithm reports both frequent and top-k elements with tight guarantees on errors. For general data distributions, *Space-Saving* answers top-k queries by returning k elements with roughly the highest frequencies in the stream; and it uses limited space for calculating frequent elements. For realistic Zipfian data, our space requirement for the exact 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. We are not aware of any other algorithms that solve the exact problems of finding frequent and top-k elements under any constraints. In addition, we slightly modify our baseline algorithm to answer continuous queries about frequent and top-k elements. Although answering such queries continuously is useful in many applications, we are not aware of any other existing solution.

The rest of the article is organized as follows. Section 2 highlights the related work. In Section 3, we introduce the *Space-Saving* algorithm, and its associated data structure, followed by a discussion of query processing in Section 4. We report the results of our experimental evaluation in Section 5. We describe how the proposed scheme can be extended to handle continuous queries about frequent and top-k elements in Section 6, and finally conclude the article in Section 7.

[1]The term *Heavy Hitters* was also used by Cormode et al. [2003].

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 whose current size is N, exceeds a user-specified support $\lceil \phi N \rceil$, where $0 \leq \phi \leq 1$; the *top-k elements* are the k elements with highest frequencies. The exact solutions of these problems require complete knowledge about the frequencies of all the elements [Charikar et al. 2002; Demaine et al. 2002], and hence are impractical for applications with large alphabets. Thus, several relaxations of the original problems were proposed.

2.1 Variations of the Problems

The FindCandidateTop(S, k, l) problem was proposed by Charikar et al. [2002] 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) [Charikar et al. 2002] 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 $F_1 \geq F_2 \geq \cdots \geq F_{|A|}$, such that E_k is the element with the kth rank. That is, all the reported k elements are almost at least as frequent as the kth element. The Hot Items[2] problem is a special case of the frequent elements problem, proposed by Misra and Gries [1982], which asks for at most k elements, each of which has frequency more than $\frac{N}{k+1}$. This extends the early work done in Boyer and Moore [1981] and Fischer and Salzberg [1982] for identifying a majority element. The most popular variation of the frequent-elements problem, finding the ϵ-Deficient Frequent Elements [Manku and Motwani 2002], asks for all the elements with frequencies more than $\lceil \phi N \rceil$, such that no element reported can have a frequency of less than $\lceil (\phi - \epsilon)N \rceil$.

Several algorithms [Charikar et al. 2002; Cormode and Muthukrishnan 2003; Demaine et al. 2002; Estan and Varghese 2003; Jin et al. 2003; Karp et al. 2003; Manku and Motwani 2002] have been proposed to handle the top-k, the frequent elements problems, and their variations. In addition, a preliminary version of this work has been published in Metwally et al. [2005b]. These techniques can be classified into *counter-based* and *sketch-based* techniques.

2.2 Counter-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 incremented every time E_i is observed in the stream. If the observed element is not monitored, that is, if there is no counter kept for this element, it is either disregarded, or some algorithm-dependent action is taken.

The *Sticky Sampling* algorithm [Manku and Motwani 2002] slices S into rounds of nondecreasing length. The probability an element is added to the list of counters, that is, that it is being monitored, decreases as the round length increases. At rounds' boundaries, for every monitored element, a coin is tossed

[2]The term *Hot Items* was coined later by Cormode and Muthukrishnan [2003].

until a success occurs. The counter is decremented for every unsuccessful toss, and is deleted if it reaches 0. Thus, the probability of adding undeleted elements is constant throughout S. The simpler, and more famous *Lossy Counting* algorithm [Manku and Motwani 2002] breaks S up into equal rounds of length $\frac{1}{\epsilon}$. Throughout every round, nonmonitored elements are added to the list. At the end of each round, r, every element, E_i, whose estimated frequency is less than r is deleted. When a new element is added in round r, it is given the benefit of doubt, its initial count is set to $r-1$, and the maximum possible overestimation, $r-1$, is recorded for the new element. Both algorithms are simple and intuitive, though they zero too many counters at rounds' boundaries. In addition, answering a frequent elements query entails scanning all counters and reporting all elements whose estimated frequency is greater than $\lceil(\phi-\epsilon)N\rceil$.

Demaine et al. [2002] proposed the *Frequent* algorithm to solve the Hot Items problem, which asks for a maximum of k elements, each of which has frequency more than $\frac{N}{k+1}$. *Frequent*, a rediscovery of the algorithm proposed by Misra and Gries [1982], outputs a list of exactly 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. [2003]. *Frequent* extends the early work done in Boyer and Moore [1981] and Fischer and Salzberg [1982] for finding a majority element using only one counter. The algorithm in Boyer and Moore [1981] and Fischer and Salzberg [1982] monitors the first element in the stream. For each observation, the counter is incremented if the observed element is the monitored one, and is decremented otherwise. If the counter reaches 0, it is assigned the next observed element, and the algorithm is then repeated. When the algorithm terminates, the monitored element is the candidate majority element. A second pass is required to verify the result. *Frequent* [Demaine et al. 2002] keeps k counters to monitor k elements. If a monitored element is observed, its counter is incremented, else all counters are decremented in an $O(1)$ operation, using a lightweight data structure. In case any counter reaches 0, it is assigned the next observed element. The sampling algorithm *Probabilistic-InPlace* [Demaine et al. 2002] solves FindCandidateTop(S, k, $\frac{m}{2}$) by using m counters. The stream is divided into rounds of increasing length. At the beginning of each round, it assigns all empty counters to the first distinct elements. At the end of each round, it deletes the least $\frac{m}{2}$ counters. The algorithm returns the largest $\frac{m}{2}$ counters, in the hope that they contain the correct top-k. Although the algorithm is simple, deleting half the counters at rounds' boundaries is $\Omega(|\text{distinct values of the deleted counters}|)$.

In general, counter-based techniques have fast per-element processing, and provable error bounds.

2.3 Sketch-Based Techniques

Sketch-based techniques do not monitor a subset of elements, but rather provide, with less stringent guarantees, frequency estimation for all elements by using arrays 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. The "representative" counters are then queried for the element frequency with expected loss of accuracy due to hashing collisions.

The probabilistic *CountSketch* algorithm, proposed by Charikar et al. [2002], solves the FindApproxTop(S, k, ϵ) problem. The space requirements of *CountSketch* decrease as the data skew increases. The algorithm keeps a sketch structure to approximate, with probability $1 - \delta$, the count of any element up to an additive quantity of γ, where γ is a function of $F_{k+1} \cdots F_{|A|}$. The family of hash functions employed hashes every element to its representative counters, such that some counters are incremented and the others are decremented for every occurrence of this element. The approximate frequency of the element is estimated by finding the median from its representative counters. A heap of the top-k elements is maintained. If the estimated frequency of the observed element exceeds the smallest estimated counter in the heap, the least frequent element is replaced by the observed element.

The *GroupTest* algorithm, proposed by Cormode and Muthukrishnan [2003], answers queries about Hot Items, with a constant probability of failure, δ. A novel algorithm, *FindMajority*, was first devised to detect the majority element, by keeping a system of a global counter and $\lceil \log(|A|) \rceil$ counters. Elements' IDs are assumed to be $1 \cdots |A|$. A hit to element E is handled by updating the global counter, and all counters whose index corresponds to a 1 in the binary representation of E. At any time, counters whose value are more than half the global counter correspond to the 1s in the binary representation of the candidate majority element, if it exists. A deterministic generalization for the Hot k elements keeps $\lceil \log \binom{|A|}{k} \rceil$ counters, with elements' IDs mapped to superimposed codes. A simpler generalized solution, *GroupTest*, is proposed that keeps only $O(\frac{k}{\delta} \ln k)$ of such systems, and uses a family of hash functions to map each element to $O(\frac{\log k}{\delta})$ *FindMajority* systems that monitor the occurrences of the element. When queried, the algorithm discards systems with more than one, or with no Hot Items. Also proposed was an elegant scheme for suppressing false positives by checking that all the systems a Hot Item belongs to are hot. Thus, *GroupTest* is, in general, accurate. However, its space complexity is large, and it offers no information about elements' frequencies or order.

The *Multistage filters* approach, proposed by Estan and Varghese [2003], which was also independently proposed by Jin et al. [2003], is similar to *GroupTest*. Using the idea of Bloom's Filters [Bloom 1970], the *Multistage filters* algorithm hashes every element to a number of counters that are incremented every time the element is observed in the stream. The element is considered to be frequent if the smallest of its representative counters satisfies the user-required support. The algorithm by Estan and Varghese [2003] judges an element to be frequent or not while updating its counters. If a counter is estimated to be frequent, it is added to a specialized set of counters for monitoring frequent elements, the *Flow Memory*. To decrease the false positives, Estan and Varghese [2003] proposed some techniques to reduce the overestimation errors in counters. Once an element is added to the Flow Memory, its counters are not monitored anymore by *Multistage filters*. In addition, Estan and Varghese [2003] proposed incrementing only the counter(s) of the minimum value.

The *hCount* algorithm [Jin et al. 2003], does not employ the error reduction techniques employed by Estan and Varghese [2003]. However, it keeps a number of imaginary elements, which have no hits. At the end of the algorithm, all

the elements in the alphabet are checked for being frequent, and the overestimation error for each of the elements is estimated to be the average number of hits for the imaginary elements.

Sketch-based techniques monitor all elements. They are less affected by the ordering of elements in the stream. On the other hand, they are more expensive than the counter-based techniques. A hit or a query entails calculations across several counters. They do not offer guarantees about frequency estimation errors, and thus, can answer only a limited number of query types.

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 preprocessing step for the other type of queries, given no information about the data distribution. For instance, for general data distribution, 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 preprocess 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 kth element; and in order to use top-k queries to preprocess 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 devised a generalized algorithm for the generalized problem.

The integrated problem of finding *significant element* is intriguing. In addition to applications like advertising networks, where both the frequent elements and the top-k problems need to be solved, the integrated problem serves the purpose of exploratory data management. The user may not have a panoramic understanding of the application data to issue meaningful queries. Often, the user issues queries about top-k elements, and then discovers that the returned elements have insignificant frequencies. Sometimes a query for frequent elements above a specific threshold returns very few or no elements. Having one algorithm that solves the integrated problem of significant elements using only one underlying data structure facilitates exploring the data samples and understanding prevalent properties.

3.1 The *Space-Saving* Algorithm

In this section, we describe counter-based *Space-Saving* algorithm and its associated *Stream-Summary* data structure. The underlying idea is to maintain partial information of interest; that is, only m elements are monitored. The counters are updated in a way that accurately estimates the frequencies of the significant elements, and a lightweight data structure is utilized to keep the elements sorted by their estimated frequencies.

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

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

Fig. 1. The *Space-Saving* algorithm.

In an ideal situation, any significant element, E_i, with rank i, that has received F_i hits, should be accommodated in the ith 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 denote the counter at the ith position in the data structure as $count_i$. The counter $count_i$ estimates the frequency f_i, of some element e_i. Since m elements are monitored, the element estimated to be the least frequent is denoted e_m. Its frequency is estimated by $count_m$, which has a value of min. If the ith position in the data structure has the right element, that is, the element with the ith rank, E_i, then $e_i = E_i$, and $count_i$ is an estimation of F_i.

The algorithm is straightforward. If a monitored element is observed, the corresponding counter is incremented. If the observed element e is not monitored, give it the benefit of doubt, and replace e_m, the element that currently has the least estimated hits, min, with e. Increment $count_m$ to min $+1$, since the new element e could have actually occurred between 1 and min $+1$ times.

For each monitored element e_i, we keep track of its maximum overestimation, ε_i, resulting from the initialization of its counter when it was inserted into the list. That is, when starting to monitor e by counter $count_m$, set its maximum overestimation error ε_m to the counter value that was evicted. Keeping track of the overestimation error for each elements is mainly useful for giving some guarantees about the output of the algorithm, as will become clear in Section 4. The algorithm is depicted in Figure 1.

We choose never to underestimate frequencies, since the algorithm is designed to err only on the positive side, that is, to never miss a frequent element. The intuition behind replacing e_m with the observed element e is sacrificing information about the element with the least estimated frequency. Hence, the algorithm looses the least possible amount of information about the history of the stream, while retaining information about possibly more *significant* elements.

In general, the top elements among nonskewed data are of no great significance. Hence, we concentrate on skewed data sets, where a minority of the elements, the more frequent ones, get the majority of the hits. The basic intuition is to make use of the skewed property of the data by assigning counters to

distinct elements, and keep monitoring the fast-growing elements. If we keep a sufficient number of counters, frequent elements will reside in the counters of bigger values and will not be distorted by the ineffective hits of the infrequent elements; thus they will never be replaced out of the monitored counters. Meanwhile, the numerous infrequent elements will be striving to reside in the smaller counters, whose values grow slower than those of the frequent elements.

In addition, if the skew remains, but the popular elements change over time, 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 loses its popularity, it will receive fewer 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 are inversely proportional to the number of counters, as shown later. Keeping only a moderate number of counters guarantees very small errors, since as proved later and illustrated through experiments, *Space-Saving* is among the most efficient techniques in terms of space. The reason is that, the more counters are kept, the less it is probable to replace elements, and thus the smaller the overestimation 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 *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.

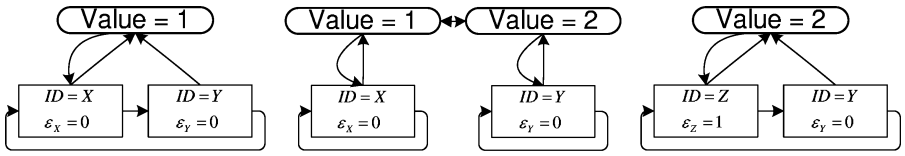
The elements can be stored in a hash table for constant amortized access cost, or in an associative memory for constant worst-case access cost. *Stream-Summary* can be sequentially traversed as a sorted list, since the buckets' list is sorted.

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; then this element is attached to this new bucket. The old bucket is deleted if it points to an empty child list. With some optimization, the worst case scenario costs 10 pointer assignments, and one heap operation. The *Increment-Counter* algorithm is sketched in Figure 2.

Example 3.1. Assuming $m = 2$, and $A = \{X, Y, Z\}$. The stream $S =$"X, Y" will yield the *Stream-Summary* data structure shown in Figure 3(a), after the two counters accommodate the observed elements. When another Y arrives,

```
Algorithm: Increment-Counter(counter count_i)
begin
  let Bucket_i be the Bucket of count_i
  let Bucket_i^+ be Bucket_i's neighbor of larger value
  Detach count_i from Bucket_i's child-list;
  count_i ++;
  //Finding the right bucket for count_i
  If (Bucket_i^+ does exist AND count_i = Bucket_i^+)
    Attach count_i to Bucket_i^+'s child-list;
  else{
    //A new bucket has to be created
    Create a new Bucket Bucket_new;
    Assign Bucket_new the value of count_i;
    Attach count_i to Bucket_new's child-list;
    Insert Bucket_new after Bucket_i;
  }
  //Cleaning up
  If Bucket_i's child-list is empty{
    Detach Bucket_i from the Stream-Summary;
    Delete Bucket_i;
  }
end;
```

Fig. 2. The *Increment-Counter* algorithm.



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

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

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

Fig. 3. Example of updates to *Stream-Summary* with $m = 2$.

a new bucket is created with value 2, and Y gets attached to it, as shown in Figure 3(b). When Z arrives, the element with the minimum counter, X, is replaced by Z. Z has $\varepsilon_Z = 1$, since that was the count of X when evicted. The final *Stream-Summary* is shown in Figure 3(c).

Stream-Summary is motivated by the work done by Demaine et al. [2002]. However, to look up a value of a counter using the data structure proposed by Demaine et al. [2002], it takes $O(m)$, while *Stream-Summary* lookups are in $\Theta(1)$ for online queries about specific elements. Online queries about specific elements is crucial for our motivating application, to check whether an element is frequent or not. Moreover, looking up the frequencies of specific elements in constant time makes *Space-Saving* more efficient when answering continuous queries, as shown later in Section 6.

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 establish its space bounds. The strength behind the simplicity of the algorithm is that it keeps information until the space is absolutely

needed, and that it does not initialize counters in batches like other counter-based algorithms. These characteristics are key to proving the space saving properties of the proposed algorithm.

LEMMA 3.2. *The length, N, of the stream is equal to the sum of all the counters in the* Stream-Summary *data structure. That is, $N = \sum_{i \leq m}(count_i)$*

PROOF. Every hit in S increments only one counter among the m counters. This is true even when a replacement happens, that is, the observed hit e was not previously monitored, and it replaces another counter e_m. This is because $count_m$ gets incremented. Therefore, at any time, the sum of all counters is equal to the length of the stream observed so far. □

A pivotal factor in the 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 $N = 4$, a stream of $S =$"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.

We assume the number of distinct elements in S is more than m. Thus, all m counters are occupied. Otherwise, all counts are exact, and the problem is trivial. Hence, from Lemma 3.2 we deduce the following.

LEMMA 3.3. *The minimum counter value,* min*, is no greater than $\lfloor \frac{N}{m} \rfloor$.*

PROOF. Lemma 3.2 can be rewritten as

$$\min = \frac{N - \sum_{i \leq m}(count_i - \min)}{m}. \tag{1}$$

All the terms in the summation of Equation (1) are nonnegative, that is, all counters are no smaller than min; hence $min \leq \lfloor \frac{N}{m} \rfloor$. □

We are interested in min since it represents an upper bound on the overestimation error in any counter in *Stream-Summary*. This relation is established in Lemma 3.4.

LEMMA 3.4. *For any element e_i in the* Stream-Summary*, $0 \leq \varepsilon_i \leq min$, that is, $f_i \leq (f_i + \varepsilon_i) = count_i \leq f_i + min$.*

PROOF. From the algorithm, the overestimation of e_i, ε_i, is nonnegative, because any observed element is always given the benefit of doubt. The overestimation ε_i is always assigned the value of the evicted counter, which is the minimum counter just before e_i started being monitored. Since the value of the minimum counter monotonically increases over time until it reaches the current min, then for all monitored elements $\varepsilon_i \leq min$. □

Therefore, the overestimation error cannot exceed $\lfloor \frac{N}{m} \rfloor$. Moreover, any element E_i, with frequency $F_i >$ min, is guaranteed to be monitored, as shown next.

THEOREM 3.5. *Any element, E_i, with $F_i >$* min *is present in* Stream-Summary.

PROOF. The proof is by contradiction. Assume E_i is not in the *Stream-Summary*. Then, it was evicted previously. Since $F_i >$ min, then F_i is more than the minimum counter value at any previous time, because the minimum counter value increases monotonically. Therefore, from Lemma 3.4, when E_i was last evicted, its estimated frequency was greater than the minimum counter value at that time. This contradicts the *Space-Saving* algorithm that evicts the element with the least counter to accommodate a new element. □

From Theorem 3.5 and Lemma 3.3, any element e_i that has occurred more than once every m observations throughout the stream must have always been monitored. Since e_i was never evicted, $\varepsilon_i = 0$. Keeping other factors constant, this holds better as f_i increases, since e_i has less chance of not being monitored. This inverse proportion between f_i and ε_i ensures that more frequent elements are less susceptible to overestimation.

From Theorem 3.5 and Lemma 3.4, we can infer an interesting general rule about the overestimation of elements' counters. For any element E_i, with rank $i \leq m$, the frequency of E_i, F_i, is no more than $count_i$, the counter occupying the ith position in the *Stream-Summary*. For instance, $count_{10}$, the counter at position 10 of the *Stream-Summary*, is an upper bound on F_{10}, even if the tenth position of the *Stream-Summary* is not occupied by E_{10}.

THEOREM 3.6. $F_i \leq count_i$.

PROOF. There are four possibilities for the position of E_i.

—The element E_i is not monitored. Thus, from Theorem 3.5, $F_i \leq min$. Thus any counter in the *Stream-Summary* is no smaller than F_i.

—The element E_i is at position j, such that $j > i$. From Lemma 3.4, the estimated frequency of E_i is no smaller than than F_i. Since j is greater than i, then the estimated frequency of e_i is no smaller than $count_j$, the estimated frequency of E_i. Thus, $count_i \geq F_i$.

—The element E_i is at position i. From Lemma 3.4, $count_i \geq f_i = F_i$.

—The element E_i is at position j, such that $j < i$. Thus, at least one element E_x with rank $x < i$ is located in some position y, such that $y \geq i$. Since the estimated frequency of E_x is no smaller than its frequency, F_x, from Lemma 3.4, and $x < i$, then the estimated frequency of E_x is no smaller than F_i. Since $y \geq i$, then $count_i \geq count_y$, which is equal to the estimated frequency of E_x. Therefore, $count_i \geq F_i$.

Therefore, in all cases, $count_i \geq F_i$. □

Theorem 3.6 establishes bounds on the rank of an element. The rank of an element e_i has to be less than j if the guaranteed hits of e_i are less than the counter at position j. That is, $count_j < (count_i - \varepsilon_i) \Rightarrow rank(e_i) < j$. Conversely, the rank of an element e_i is greater than the number of elements having guaranteed hits more than $count_i$. That is, $rank(e_i) > Count(e_j | (count_j - \varepsilon_j) >$

```
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. 4. Reporting frequent elements.

$count_i$). Thus, Theorem 3.6 helps establishing the order-preservation property among the top-k, as discussed later.

In the next section, we use these properties to derive a bound on the space requirements for solving the frequent elements and the top-k problems.

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, the algorithm sequentially traverses *Stream-Summary* as a sorted list until an element with frequency less than the user support is reached. Thus, frequent elements are reported in $\Theta(|\text{frequent elements}|)$. An element, e_i, is *guaranteed to be a frequent element* if its guaranteed number of hits, $count_i - \varepsilon_i$, exceeds $\lceil \phi N \rceil$, the minimum support. If for each reported element e_i, $count_i - \varepsilon_i > \lceil \phi N \rceil$, 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. The *QueryFrequent* algorithm is given in Figure 4.

4.1.1 *The General Case.* In the general case, *Space-Saving* solves the ϵ-Deficient Frequent Elements problem, for a user-specified ϵ, by reporting all the elements with frequencies more than $\lceil \phi N \rceil$, such that no reported element has frequency less than $\lceil (\phi - \epsilon) N \rceil$. We will analyze the space requirements assuming no specific data distribution.

THEOREM 4.1. *Assuming no specific data distribution,* Space-Saving *uses a number of counters of* $\min(|A|, \lceil \frac{1}{\epsilon} \rceil)$ *to find all* ϵ*-Deficient Frequent Elements. Any element,* E_i*, with frequency* $F_i > \lceil \phi N \rceil$ *is guaranteed to be reported.*

PROOF. Since, for any monitored element, the upper bound of ε_i is min, from Lemma 3.3, it follows that $\varepsilon_i \leq min \leq \lfloor \frac{N}{m} \rfloor$. If we set $min = \lceil \epsilon N \rceil$ in this inequality, then $m \geq \lceil \frac{1}{\epsilon} \rceil$ guarantees an error rate of ϵ. From Theorem 3.5, any

element E_i whose $F_i > min$ is guaranteed to be in the *Stream-Summary*. Since $\phi \geq \epsilon$, from Theorem 3.5, any element with frequency greater than $\lceil \phi N \rceil$ is monitored in the *Stream-Summary*, and hence is guaranteed to be reported. □

The bound of Theorem 4.1 is tight. For instance, this can happen if all the elements in the stream are distinct. In addition, Theorem 4.1 shows that the space consumption of *Space-Saving* is within a constant factor of the lower bound on the space of any deterministic counter-based algorithm, as shown in Theorem 4.2.

THEOREM 4.2. *Any deterministic counter-based algorithm uses a number of counters of at least* $\min(|A|, \lceil \frac{1}{2\epsilon} \rceil)$ *to find all* ϵ*-Deficient Frequent Elements.*

PROOF. The proof is similar to that given by Bose et al. [2003]. Given two streams S_1 and S_2, of length $L(m+1)+1$ for an arbitrary large multiple L. The two streams have the same first $L(m+1)$ elements, where $m+1$ elements occur L times each. After observing the $L(m+1)$ stream elements, any counter-based algorithm with m counters will be monitoring only m elements. The last element is the only difference between S_1 and S_2. S_1 ends with an element e_1 that was never observed before, and S_2 ends with an element e_2 that has occurred before but is not monitored by the algorithm. Any deterministic algorithm should handle the last element of S_1 and S_2 in the same manner, since it has no record of its previous hits. If the algorithm estimated the previous hits of the last element to be 1, then the algorithm will have an error rate of $\frac{1}{m+1}$ in the case of S_2. On the other hand, if the algorithm estimated the previous hits of the last element to be L, then the algorithm will have an error rate of $\frac{1}{m+1}$ in the case of S_1. The estimation that results in the least error in both cases is $\frac{1}{2(m+1)}$. Therefore, the least number of counters to guarantee an error rate of ϵ is $\lceil \frac{1}{2\epsilon} \rceil$. □

4.1.2 *Zipf Distribution Analysis.* A Zipfian [Zipf 1949] data set, with parameter α, has the frequency, F_i, of an element, E_i, with the ith rank, such that $F_i = \frac{N}{i^\alpha \zeta(\alpha)}$, where

$$\zeta(\alpha) = \sum_{i=1}^{|A|} \frac{1}{i^\alpha}.$$

$\zeta(\alpha)$ converges to a small constant inversely proportional to α, except for $\alpha \leq 1$. For instance, $\zeta(1) \approx \ln(1.78|A|)$. As $|A|$ grows to infinity, $\zeta(2) \approx 1.645$, and $\zeta(3) \approx 1.202$. We assume $\alpha \geq 1$, to ensure that the data is skewed, and hence that it is worth analyzing. As noted before, we do not expect the popular elements to be of great importance if the data is uniform or weakly skewed.

To analyze the Zipfian case, we need to introduce some new notation. Among all the possible permutations of S, the maximum possible min is denoted $\min_{\max}$, and among all the elements with hits more than $\min_{\max}$, the element with least hits is denoted E_r, for some rank r. Thus, we can deduce from Theorem 3.5 that

LEMMA 4.3. *Any element,* E_i*, whose* $F_i > \min_{\max}$*, is guaranteed to be monitored, if and only if* $i \leq r$*, regardless of the ordering of* S*.*

Now, $\min_{\max}$, and E_r can be used to establish an upper bound on the space requirements for processing Zipfian data.

THEOREM 4.4. *Assuming noiseless Zipfian data with parameter α, to find all ϵ-Deficient Frequent Elements, the number of counters used by* Space-Saving *is bounded by*

$$\min\left(|A|, \left\lceil\left(\frac{1}{\epsilon}\right)^{\frac{1}{\alpha}}\right\rceil, \left\lceil\frac{1}{\epsilon}\right\rceil\right).$$

This is regardless of the stream permutation.

PROOF. From Equation (1), and Lemma 3.4, $min_{max} \geq \frac{N-\sum_{i\leq m} f_i}{m}$, from which it can be ascertained that

$$min_{max} \geq \frac{N - \sum_{i\leq m} F_i}{m}.$$

From Lemma 4.3, substitute $F_r > min_{max}$. Rewriting frequencies in their Zipfian form yields

$$\frac{1}{r^\alpha} > \frac{1}{m} * \sum_{i=m+1}^{|A|} \frac{1}{i^\alpha}.$$

This can be approximated to $\frac{1}{r^\alpha} > \frac{1}{m^\alpha} * \sum_{i=2}^{|A|/m} \frac{1}{i^\alpha}$, which can be simplified to

$$m > r * \left(\sum_{i=2}^{|A|/m} \frac{1}{i^\alpha}\right)^{\frac{1}{\alpha}}.$$

Since $\left(\sum_{i=2}^{|A|/m} \frac{1}{i^\alpha}\right)^{\frac{1}{\alpha}}$ has no closed form, m is set to satisfy the stronger constraint

$$m > r(\zeta(\alpha) - 1)^{\frac{1}{\alpha}}.$$

Since $F_{r+1} = \frac{N}{(r+1)^\alpha \zeta(\alpha)} < min_{max} < \epsilon N$, then the smaller the error bound ϵ, the smaller the value of $\min_{\max}$, the larger r should be, and the larger m should be. Therefore, r is chosen to satisfy

$$r \geq \left(\frac{1}{\epsilon\zeta(\alpha)}\right)^{\frac{1}{\alpha}}.$$

Combining this result with the relation between m and r established above implies that to guarantee an error that is bound by ϵ, m should satisfy

$$m > \left(\frac{\zeta(\alpha) - 1}{\epsilon\zeta(\alpha)}\right)^{\frac{1}{\alpha}}.$$

If $\alpha > 1$, the upper bound on ϵ will be enforced by satisfying $m = \lceil(\frac{1}{\epsilon})^{\frac{1}{\alpha}}\rceil$. Otherwise, the bound of $m \geq \lceil\frac{1}{\epsilon}\rceil$ will apply from the general case discussed previously in Theorem 4.1. □

Having established the bounds of *Space-Saving* for both the general and the Zipf distributions, we compare these bounds to other algorithms. In addition, we comment on some practical issues, which can not be directly inferred from the theoretical bounds.

4.1.3 *Comparison with Similar Work.* The bound of Theorem 4.1 is tighter than those guaranteed by the algorithms in Estan and Varghese [2003], Jin et al. [2003], and Manku and Motwani [2002]. *Sticky Sampling* [Manku and Motwani 2002] has a space bound of $\frac{2}{\epsilon}\ln(\frac{1}{\phi\delta})$, where δ is the failure probability. *Lossy Counting* [Manku and Motwani 2002] has a bound of $\frac{1}{\epsilon}\ln(\epsilon N)$. Both the *hCount* algorithm [Jin et al. 2003], and the *Multistage filters* [Estan and Varghese 2003] require a number of counters bounded by $\frac{e}{\epsilon} * \ln(\frac{-|A|}{\ln\delta})$. Furthermore, *Space-Saving* has a tighter bound than *GroupTest* [Cormode and Muthukrishnan 2003], whose bound is $O(\frac{1}{\phi}\ln(\frac{1}{\delta\phi})\ln(|A|))$, for a large range of practical values of the parameters $|A|$, ϵ, and ϕ. For example, for $N = 10^{10}$, $|A| = 10^7$, $\phi = 10^{-1}$, $\epsilon = 10^{-2}$, and $\delta = 10^{-1}$, and making no assumptions about the data distribution, *Space-Saving* needs only 100 counters, while *Sticky Sampling* needs 922 counters, *Lossy Counting* needs 1843 counters, *hCount* and *Multistage filters* need 4155 counters, and *GroupTest* needs $C * 743$ counters, where $C \geq 1$.

Frequent [Demaine et al. 2002] has a similar space bound to *Space-Saving* in the general case. Using m counters, the elements' under-estimation error in *Frequent* is bounded by $\frac{N-1}{m}$. This is close to the theoretical underestimation error bound, as proved by Bose et al. [2003]. However, there is no straightforward feasible extension of the algorithm to track the underestimation error for each counter, since the current form of the algorithm does not support estimating the missed hits for an element that is starting being monitored. In addition, every observation of a nonmonitored element increases the errors for all the monitored elements, since their counters get decremented. Therefore, elements of high frequency are highly 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 by Demaine et al. [2002] 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 in Section 5.

The number of counters used in *GroupTest* [Cormode and Muthukrishnan 2003] 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 does not reveal the relative order of the elements. In addition, its assumption that elememts' IDs are $1 \cdots |A|$ can only be enforced by building an indexed lookup table that maps every element to a unique number in the range $1 \cdots |A|$. Thus, in practice, *GroupTest* needs $O(|A|)$ space, which is infeasible in most cases. The *hCount* algorithm makes a similar assumption about the alphabet. In addition, it has to scan the entire alphabet domain for identifying the frequent elements. This is true even if a small portion of the elements' IDs were observed in the stream. This is in contrast to *Space-Saving*, which only requires the m elements' IDs to fit in memory.

For the Zipfian case, we are not aware of a similar analysis. For the numerical example given above, if the data is Zipfian with $\alpha = 2$, *Space-Saving* would need only 10 counters, instead of 100, to guarantee the same error of 10^{-2}.

4.2 Top-k Elements

For the top-k elements, the algorithm can output the first k elements. From Theorem 3.6, $count_{k+1}$, the overestimated number of hits for the element in position $k+1$, is an upper bound on F_{k+1}, the hits of the element E_{k+1}, which is of rank $k+1$. Therefore, an element, e_i, is *guaranteed to be among the top-k* if its guaranteed number of hits, $count_i - \varepsilon_i$, exceeds $count_{k+1}$.

We call the results to have *guaranteed top-k*, if by simply inspecting the results, the algorithm can determine that the reported top-k elements are correct. *Space-Saving* reports a guaranteed top-k if, for all i, $(count_i - \varepsilon_i) \geq count_{k+1}$, where $i \leq k$. That is, all the reported k elements are guaranteed to be among the top-k elements.

All guaranteed top-i subsets, for all i, can be reported in $\Theta(m)$, by iterating on all the counters $1 \cdots m-1$. During each iteration, i, the first i elements are guaranteed to be the top-i elements if the minimum value of $(count_j - \varepsilon_j)$ found so far is no smaller than $count_{i+1}$, where $j \leq i$. 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, for all i, where $i \leq k$, $count_i - \varepsilon_i \geq count_{i+1}$. 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 overestimated hits of the next element. Thus, the order is guaranteed if the algorithm guarantees the top-i, for all $i \leq k$. The algorithm *QueryTop-k* is given in Figure 5.

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 for as many iterations such that the total inspected elements k' are guaranteed to be the top-k', where $k' > k$.

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 article, we assume that the algorithm outputs only the first k elements, that is, the second loop is not executed. Next, we look at the space requirements of the algorithm.

4.2.1 *The General Case.* For the guaranteed top-k case, it is widely accepted that the space requirements are $\Theta(|A|)$ [Charikar et al. 2002; Demaine et al. 2002] for solving the exact problem, with no assumptions on the data distribution. Since, for general data distribution, we are not able to solve the exact problem, we restrict the discussion to the relaxed version, FindApproxTop(S,

```
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. 5. Reporting top-k.

k, ϵ) [Charikar et al. 2002], which is to find a list of k elements, each of which has frequency more than $(1-\epsilon)F_k$.

We deal with skewed data later, in Section 4.2.2, where we provide the first proven space bound for the guaranteed solution of the exact top-k problem, for Zipfian data distribution.

THEOREM 4.5. *Regardless of the data distribution, to solve the FindApproxTop(S, k, ϵ) problem,* Space-Saving *uses only* $\min(|A|, \lceil \frac{N}{\epsilon F_k} \rceil)$ *counters. Any element with frequency more than* $(1-\epsilon)F_k$ *is guaranteed to be monitored.*

PROOF. This is another form of Theorem 4.1, but $min = \lceil \epsilon F_k \rceil$, instead of $min = \lceil \epsilon N \rceil$. By the same token, we set $m = \lceil \frac{1}{\epsilon} * \frac{N}{F_k} \rceil$ so that $\varepsilon_i \leq \epsilon F_k$ is guaranteed. □

4.2.2 *Zipf Distribution Analysis.* To answer exact top-k queries for Zipf distribution, ϵ can be automatically set to less than $F_k - F_{k+1}$. Thus, *Space-Saving* guarantees correctness, and order.

THEOREM 4.6. *Assuming noiseless Zipfian data with parameter* $\alpha > 1$*, to calculate the exact top-k, the number of counters used by* Space-Saving *is bounded by*

$$\min\left(|A|, O\left(\left(\frac{k}{\alpha}\right)^{\frac{1}{\alpha}} k\right)\right).$$

When $\alpha = 1$, the space complexity is

$$\min(|A|, O(k^2 \ln(|A|))).$$

This is regardless of the stream permutation. Also, the order among the top-k elements is preserved.

PROOF. From Equation (1), Lemma 3.4, and Lemma 4.3, we can deduce that for the maximum possible value of min, min_{max}, and the least frequent element that is guaranteed to be monitored, E_r, it is true that

$$min_{max} \leq \frac{N - \sum_{i \leq r}(F_i - min_{max})}{m}.$$

With some simplification, and substituting $F_{r+1} \leq min_{max}$, from Lemma 4.3, it follows $F_{r+1} \leq \frac{N - \sum_{i \leq r} F_i}{m-r}$. Rewriting frequencies in their Zipfian form yields

$$m - r \leq (r+1)^{\alpha} \sum_{i=r+1}^{|A|} \frac{1}{i^{\alpha}}.$$

This can be approximated to $m - r < (r+1) * \sum_{i=1}^{|A|/(r+1)} \frac{1}{i^{\alpha}}$, which simplifies to

$$\frac{1}{r} < \frac{\zeta(\alpha) + 1}{m - \zeta(\alpha)}.$$

To guarantee that the first k slots are occupied by the top-k, we have to make sure that the difference between F_k and F_{k+1} is more than min_{max}, since from Lemma 3.4, $0 \leq \varepsilon_i \leq min_{max}$ for all monitored elements. That is, the condition $min_{max} < F_k - F_{k+1}$ has to be enforced. Thus, $min_{max} < \frac{N}{\zeta(\alpha)} * \frac{(k+1)^{\alpha} - k^{\alpha}}{(k+1)^{\alpha} k^{\alpha}}$. Enforcing a tighter condition, F_r is set to satisfy $F_r < \frac{N}{\zeta(\alpha)} * \frac{\alpha}{(k+1)^{\alpha} k}$. Enforcing an even tighter condition by combining this with the relation between m and r established above, it is sufficient to satisfy

$$\frac{N}{\zeta(\alpha)} * \left(\frac{\zeta(\alpha) + 1}{m - \zeta(\alpha)}\right)^{\alpha} < \frac{N}{\zeta(\alpha)} * \frac{\alpha}{(k+1)^{\alpha} k}.$$

After some manipulation, a lower bound is reached on m to guarantee top-k correctness: $[(\zeta(\alpha) + 1)(\frac{k}{\alpha})^{\frac{1}{\alpha}}(k+1)] + \zeta(\alpha) < m$.

If $\alpha = 1$, then $\zeta(\alpha) = \zeta(1) \approx \ln(1.78|A|)$, and the complexity reduces to

$$\min(|A|, O(k^2 \ln(|A|))).$$

If $\alpha > 1$, then $\zeta(\alpha)$ converges to a small constant inversely proportional to α, and the complexity reduces to

$$\min\left(|A|, O\left(\left(\frac{k}{\alpha}\right)^{\frac{1}{\alpha}} k\right)\right).$$

This establishes the space bound. We now prove the order-preserving property. If the data distribution is Zipfian, then, $(F_i - F_{i+1}) > (F_{i+1} - F_{i+2})$. Since $min_{max} < (F_k - F_{k+1})$, then, $\forall_{i \leq k}, min_{max} < (F_i - F_{i+1})$. Since $\forall_{i \leq m}, \varepsilon_i \leq min_{max}$,

then, the over-estimation errors are not effective enough to change the order among the top-k elements. □

In addition to solving the ϵ-Deficient Frequent Elements problem in Section 4.1.2, from Theorem 4.6 we can establish a bound on the space needed for the exact solution of the frequent elements problem in case of Zipfian data. Given noise-free Zipfian data with parameter $\alpha \geq 1$, *Space-Saving* can report the elements that satisfy the user support $\lceil \phi N \rceil$, with very small errors in their frequencies.

COROLLARY 4.7. *Assuming noiseless Zipfian data with parameter $\alpha > 1$, to calculate the exact frequent elements, the number of counters used by* Space-Saving *is bounded by*

$$min\left(|A|, O\left(\left(\frac{1}{\phi}\right)^{\frac{\alpha+1}{\alpha^2}}\right)\right).$$

When $\alpha = 1$, the space complexity is

$$min\left(|A|, O\left(\frac{1}{\phi^2 \ln(|A|)} + \ln(|A|)\right)\right).$$

This is regardless of the stream permutation.

PROOF. Assuming Zipf distribution, it is possible to map a frequent elements query into a top-k elements query. Since the support is known, it is possible to know the rank of the least frequent element that satisfies the support. That is, if $\lceil \phi N \rceil < \frac{N}{i^\alpha \zeta(\alpha)}$, where i is the rank of the least frequent element that satisfies the support, then

$$i > \left\lfloor \left(\frac{1}{\zeta(\alpha)\phi}\right)^{\frac{1}{\alpha}} \right\rfloor.$$

From Theorem 4.6, the number of counters, m, needed to calculate the exact top-i elements is $m > [(\zeta(\alpha)+1)(\frac{i}{\alpha})^{\frac{1}{\alpha}}(i+1)] + \zeta(\alpha)$. Substituting $i = \lfloor \frac{1}{\zeta(\alpha)\phi}^{\frac{1}{\alpha}} \rfloor + 1$ yields

$$m > \left[(\zeta(\alpha)+1)\left(\frac{\left\lfloor \left(\frac{1}{\zeta(\alpha)\phi}\right)^{\frac{1}{\alpha}} \right\rfloor + 1}{\alpha}\right)^{\frac{1}{\alpha}} \left(\left\lfloor \left(\frac{1}{\zeta(\alpha)\phi}\right)^{\frac{1}{\alpha}} \right\rfloor + 2\right)\right] + \zeta(\alpha).$$

If $\alpha = 1$, then $\zeta(\alpha) = \zeta(1) \approx \ln(1.78|A|)$, and the space complexity reduces to

$$min\left(|A|, O\left(\frac{1}{\phi^2 \ln(|A|)} + \ln(|A|)\right)\right).$$

If $\alpha > 1$, then $\zeta(\alpha)$ converges to a small constant inversely proportional to α, and the space complexity reduces to

$$min\left(|A|, O\left(\left(\frac{1}{\phi}\right)^{\frac{\alpha+1}{\alpha^2}}\right)\right).$$

This establishes the space bound. □

To the best of our knowledge, this is the first work to look at the space bounds for answering exact queries, 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.

4.2.3 *Comparison with Similar Work.* These bounds are tighter than the bounds guaranteed by the best known algorithm, *CountSketch* [Charikar et al. 2002], 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 $O\left(\log(\frac{N}{\delta})(k + \frac{1}{(\epsilon F_k)^2}\sum_{i=k+1}^{|A|} {F_i}^2)\right)$, with a large constant hidden in the big-O notation [Charikar et al. 2002; Cormode and Muthukrishnan 2003]. The bound of *Space-Saving* for the relaxed problem is $\lceil \frac{N}{\epsilon F_k} \rceil$, with a 0-failure probability. For instance, assuming no specific data distribution, for $N = 10^{10}$, $|A| = 10^7$, $k = 100$, and $\epsilon = \delta = 10^{-1}$, *Space-Saving* requires 10^6 counters, while *CountSketch* needs $C * 3.6 * 10^{10}$ 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 of such elements.

In the 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 nonskewed distribution, the top few elements have a less significant share in the stream than in the case of Zipf(1), and less frequent elements will have a higher share in S than they would have had if the distribution is Zipf(1). Using this assumption, we rewrite the bound of *Space-Saving* as $O(\frac{k*\ln(N)}{\epsilon})$; while the bound in [Charikar et al. 2002] can be rewritten as

$$O\left(\log\left(\frac{N}{\delta}\right) * \left(k + \frac{k^2}{\epsilon^2}\left(\frac{1}{k+1} - \frac{1}{|A|}\right)\right)\right) \approx O\left(\frac{k}{\epsilon^2}\log\left(\frac{N}{\delta}\right)\right).$$

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

In the case of Zipf Distribution, the bound of Charikar et al. [2002] is $O\left(k \log(\frac{N}{\delta})\right)$. For $\alpha > 1$, the bound of *Space-Saving* is $O\left((\frac{k}{\alpha})^{\frac{1}{\alpha}}k\right)$. Only when $\alpha = 1$, the space complexity is $O\left(k^2 \ln(|A|)\right)$, and thus *Space-Saving* requires less space for cases of skewed data, long streams/windows, and has a 0-failure probability. In addition, *Space-Saving* preserves the order of the top-k elements.

To show the difference in space requirements, consider the following example. For $N = 10^{10}$, $|A| = 10^7$, $k = 100$, $\alpha = 2$, and $\delta = 10^{-1}$, *Space-Saving*'s space requirements are only 708 counters, while *CountSketch* needs $C * 3655$ counters, where $C \gg 1$.

This is the first algorithm that can give guarantees about its output. For top-k queries, *Space-Saving* specifies the guaranteed elements among the top-k. Even if it cannot guarantee all the top-k elements, it can guarantee the top-k' elements.

5. EXPERIMENTAL RESULTS

To evaluate the capabilities of *Space-Saving*, we conducted a comprehensive set of experiments, using both real and synthetic data. We tested the performance of *Space-Saving* for finding both the frequent and the top-k elements under different parameter settings. We compared the results to those of the algorithms whose theoretical bounds are not worse than those of *Space-Saving*. We were interested in the *recall*, the number of correct elements found as a percentage of the number of correct elements; and the *precision*, the number of correct elements found as a percentage of the entire output [Cormode and Muthukrishnan 2003]. It is worth noting that an algorithm will have a recall and a precision of 1, if it outputs all and nothing but the correct set of elements. Superfluous output reduces precision, while failing to identify all correct elements reduces recall.

We also measured the run time and space used by each algorithm, which are good indicators of its capability to handle high-speed streams, and to reside on servers with limited memories. Notice that we included the size of the hash tables used in the algorithms for fair comparisons of the space usages.

For the frequent elements problem, we compared *Space-Saving* to *GroupTest* [Cormode and Muthukrishnan 2003], and *Frequent* [Demaine et al. 2002]. For *GroupTest* and *Frequent*, we used the C code available on the Web site of the first author of Cormode and Muthukrishnan [2003]. For the top-k problem, we implemented *Probabilistic-InPlace* [Demaine et al. 2002], and *CountSketch* [Charikar et al. 2002]. For *CountSketch* [Charikar et al. 2002], we implemented the median algorithm by Hoare [Hoare 1961] with Median-of-three partition, which has a linear run time, in the average case [Kirschenhofer et al. 1997]. Instead of maintaining a heap as suggested by Charikar et al. [2002], we kept a *Stream-Summary* of fixed length k. This guarantees constant time update for elements that are in the *Stream-Summary*, while a heap would entail $O(\log(k))$ operations. The difference in space usage between a heap and a *Stream-Summary* of size k is negligible, when compared to the size of the hash space used by *CountSketch*. For the hidden constant of the space bounds given in Charikar et al. [2002], we ran *CountSketch* several times, and estimated that a factor of 16 would enable *CountSketch* to give results comparable to *Space-Saving* in terms of precision and recall. For the probabilistic algorithms, *GroupTest* and *CountSketch*, we set the probability of failure, δ, to 0.01, which is a typical value for δ. All the algorithms were compiled using the same compiler,

and were run on a Pentium IV 2.66-GHz PC, with 1.0 GB of RAM, and 80 GB of hard disk.

5.1 Synthetic Data

We generated several synthetic Zipfian data sets with the Zipf parameter varying from 0.5, which is very slightly skewed, to 3.0, which is highly skewed, with a fixed increment of $\frac{1}{2}$. The size of each data set, N, is 10^8 hits, and the alphabet was of size $5 * 10^6$. We conducted two sets of experiments. In the first set, we varied the Zipf parameter, α, and measured how the performances of the algorithms change for the same set of queries. In the second set of experiments, we used a data set with a realistic skew ($\alpha = 1.5$), and compared the results of the algorithms as we varied the parameters of the queries.

5.1.1 *Varying the Data Skew.* In this set of experiments, we varied the Zipf parameter, α, and measured how the performances of the algorithms change, for the same set of queries. This set of experiments measure how the algorithms adapt to, and make use of the data skew.

5.1.1.1 *The Frequent Elements Problem.* The query issued for *Space-Saving*, *GroupTest*, and *Frequent* was to find all elements with frequency at least $\frac{N}{10^2}$. For *Space-Saving*, we assigned enough counters to guarantee correct results from Corollary 4.7. When the Zipf parameter was 0.5, we assigned the same number of counters as in the case when the Zipf parameter was 1.0. The results comparing the recall, precision, time, and space used by the algorithms are summarized in Figure 6.

Although *Frequent* ran up to six times faster than *Space-Saving* and had a constant recall of 1, as reported in Figures 6(a) and 6(c), 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 0.5, the algorithm reported 16 elements, and actually there were no elements satisfying the support. For the rest of the experiments in Figure 6(b), the precision achieved by *Frequent* ranged from 0.049 to 0.158. The space used ranged from one-tenth to four times the space of *Space-Saving*, as shown in Figure 6(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. *Frequent* did not always output exactly 100 elements for each experiment, since when it decrements the lowest counter, more than one element sharing that counter could potentially be deleted if it reaches 0.

From Figure 6(a), the ratio in run time between *Space-Saving* and *GroupTest* changed from 1:0.73, when the Zipf parameter was 0.5, to 1:1.9, when the data was highly skewed. When the Zipf parameter was 0.5, there were no frequent elements, and both algorithms identified none. We report this fact for both algorithms as having a precision and recall of 1 in Figures 6(b) and 6(c), respectively. However, when the Zipf parameter was 1, the difference in precision between the two algorithms was 14%, since *GroupTest* was not able to prune out all the false positives due to the weak data skew. For values of the

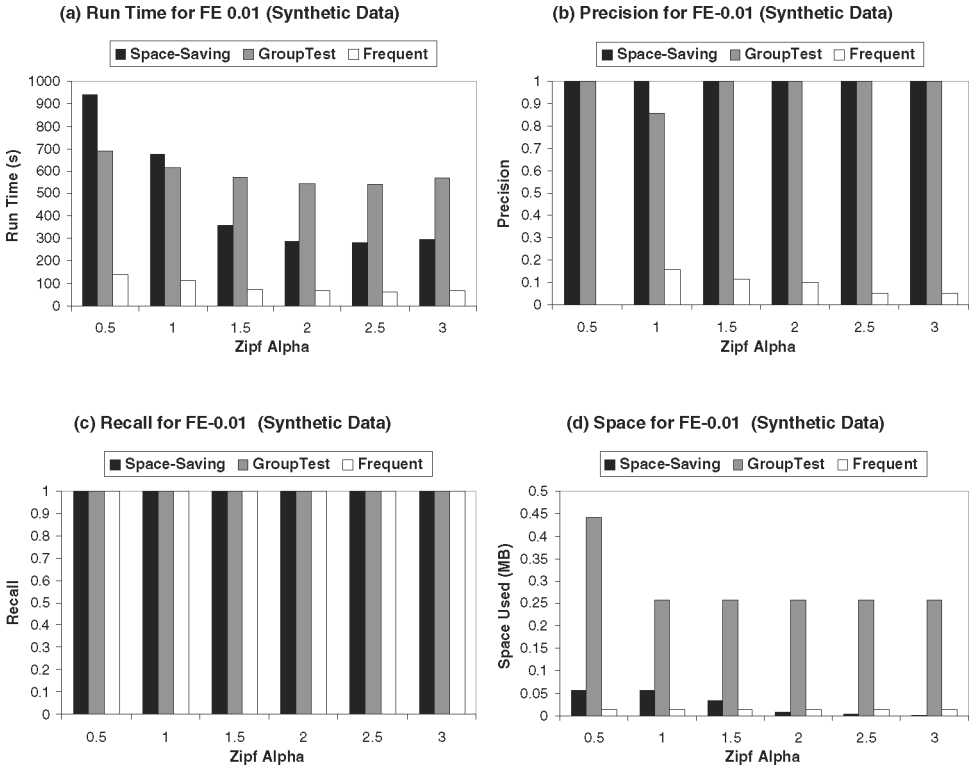


Fig. 6. Performance comparison for the frequent elements problem using synthetic Zipfian data—varying data skew.

Zipf parameter larger than 1.0, the precisions of both algorithms were constant at 1, as reported in Figure 6(b). The recalls of both algorithms were constant at 1 for all values of the Zipf parameter, as is clear from Figure 6(c). The advantage of *Space-Saving* is evident in Figure 6(d), which shows that *Space-Saving* achieved a reduction in the space used by a factor ranging from 8 when the Zipf parameter was 0.5 up to 200 when the Zipf parameter was 3.0. This shows that *Space-Saving* adapts well to the data skew.

5.1.1.2 *The Top-k Problem.* *Space-Saving*, *CountSketch*, and *Probabilistic-InPlace* were used to identify the top-50 elements. *Space-Saving* monitored enough elements to guarantee that the top-50 elements are correct and reported in the right order, as illustrated in Theorem 4.6. For $\alpha = 0.5$, the same number of counters were monitored as in the case of $\alpha = 1.0$. Both *Space-Saving* and *Probabilistic-InPlace* were allowed the same number of counters. We were not able to make *Probabilistic-InPlace* produce results comparable to the quality of the results of *Space-Saving*. If *Probabilistic-InPlace* is given $2k$ counters so that it outputs only k elements, its recall is unsatisfactory. If it is allowed a large number of counters, its recall increases, due to tighter estimation; but the precision drops dramatically, since a lot of superfluous elements are output. Thus, we allowed it to run using the same number of counters as *Space-Saving*,

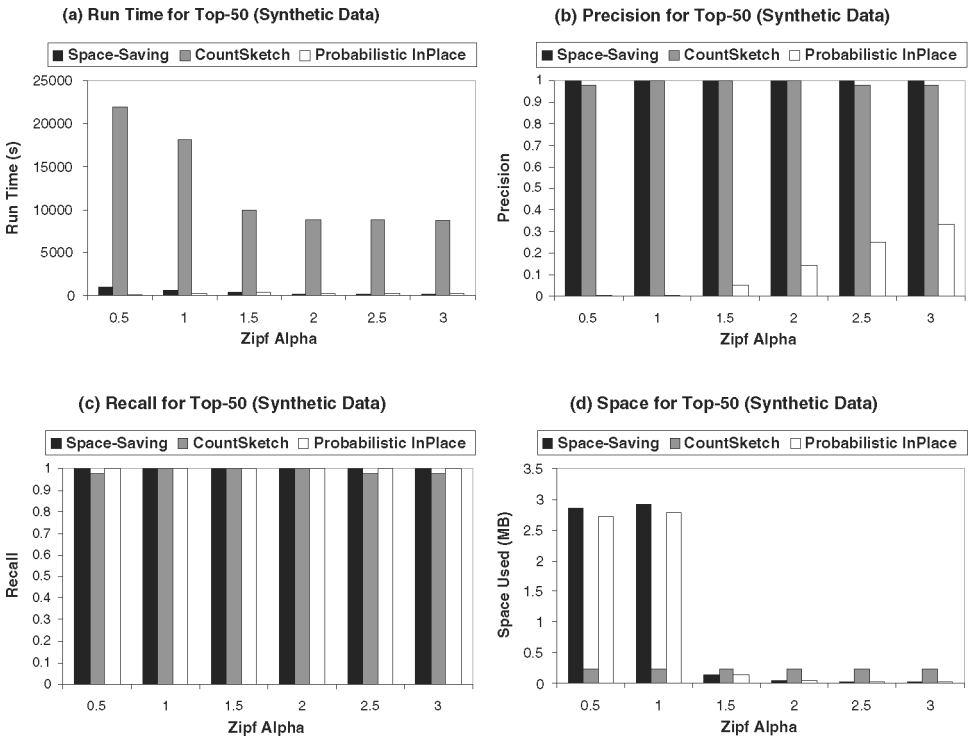


Fig. 7. Performance comparison for the top-k problem using synthetic Zipfian data—varying data skew.

and the time, precision, and recall were measured. The results are summarized in Figure 7.

From Figure 7(b), the outputs of *Space-Saving* and *CountSketch* were much better than *Probabilistic-InPlace*, in terms of precision. On the contrary, from Figure 7(c), the recall of *Probabilistic-InPlace* was constant at 1 throughout the entire range of α. On the whole, the run time and space usages of both *Probabilistic-InPlace* and *Space-Saving* were comparable. Nevertheless, from Figure 7(a), we notice that the run time of *Probabilistic-InPlace* was longer than that of *Space-Saving* for $\alpha \geq 1.5$, due to the unnecessary deletions at the boundaries of rounds.

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.[4] *CountSketch* had precision and recall varying between 0.98 and 1.0, as is clear from Figures 7(b) and 7(c). From Figure 7(d), the space reductions of *Space-Saving* become clear only for skewed data. The ratio in space used by *Space-Saving* and *CountSketch* ranged from 10:1 when the data was weakly skewed, to 1:10 when the data was highly skewed. This is because *Space-Saving* takes advantage of the skew of the data to minimize the number of counters it needs to

[4]*CountSketch* and *Space-Saving* have the precision equal to recall, for any query, since exactly k elements are output.

keep, while the proved bound on the space used by *CountSketch* is fixed for $\alpha > \frac{1}{2}$ [Charikar et al. 2002]. The reductions of *Space-Saving* in time, when compared with *CountSketch*, are significant. From Figure 7(a), *Space-Saving* run-time, though almost constant, was 22 times smaller when the data was weakly skewed, and 33 times smaller when the data was highly skewed. The run-time of *CountSketch* decreased as α increased, since the number of times *CountSketch* estimated the frequency of an element decreased, which is the bottleneck in *CountSketch*. However, the run-time of *Space-Saving* dropped faster as the data became more skewed, since the gap between the significant buckets' values increased, and it grew less likely that any two elements in the top-k shared the same bucket. This reduced the work to increment the top-k elements.

We can easily see that running on a 2.66-GHz machine enables *CountSketch* to handle streams with a rate not higher than 5 hits/ms, since when the data was almost uniform, *CountSketch* took 219 μs, on average, to process each observation in the stream. Since, in real life, the traffic rate entails processing each data entry in at most 50 μs, *CountSketch* is rendered unsuitable for our application.

5.1.2 *Varying the Query Parameters.* This set of experiments measured how the algorithms perform under different realistic query parameters, keeping the data skew parameter constant at a realistic value. The data set with the Zipf parameter 1.5 was used for this purpose.

5.1.2.1 *The Frequent Elements Problem.* The query issued for *Space-Saving*, *GroupTest*, and *Frequent* was to find all elements with frequency at least $\lceil \frac{N}{\phi} \rceil$. The support ϕ was varied from 0.001 to 0.01. The results are summarized in Figure 8.

From Figure 8(c), *Frequent* was able to attain a recall of 1, for all the queries issued. From Figure 8(a), *Frequent*'s run time was up to five times faster than *Space-Saving*. In addition, the space usage of *Frequent* dropped to $\frac{2}{5}$ that of *Space-Saving*, as is clear from Figure 8(d). However, *Frequent* has a precision ranging from 0.087 to 0.115, as indicated by Figure 8(b), which is a significant drawback of this algorithm. This is due to its inability to prune out false positives.

Both *GroupTest* and *Space-Saving* were able to attain a value of 1 for recall for all the values of support, as is clear from Figure 8(c). However, from Figure 8(b), the precision of *GroupTest* dropped to 0.952 when ϕ was $\frac{1}{250}$. Figure 8(d) shows that *Space-Saving* used space ranging from 8 to 18 times less than that of *GroupTest*, and ran twice as fast, as shown in Figure 8(a).

In conclusion, we can see that *Space-Saving* combined the lightweight advantage of *Frequent* and the precision advantage of *GroupTest*.

5.1.2.2 *The Top-k Problem.* *Space-Saving*, *CountSketch*, and *Probabilistic-InPlace* were used to identify the top-k elements in the stream. The parameter k was varied, and the results are shown in Figure 9.

Probabilistic-InPlace had run-time and space usage that were very close to *Space-Saving*, as illustrated in Figures 9(a) and 9(d). *Probabilistic-InPlace* was

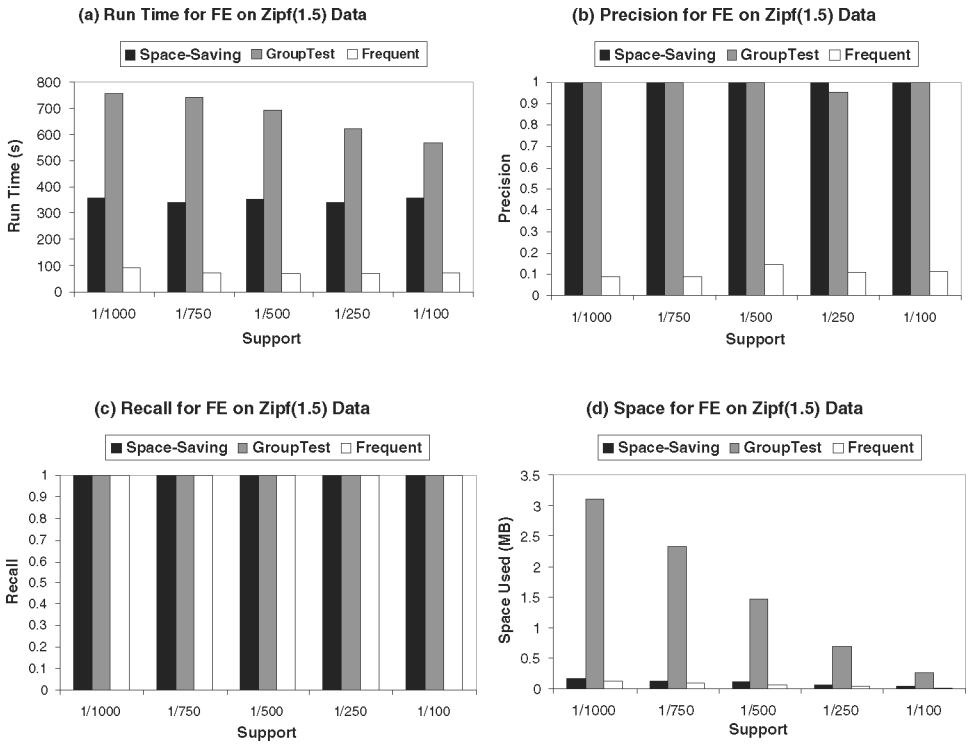


Fig. 8. Performance comparison for the frequent elements problem using synthetic Zipf(1.5) data—varying the support.

able to attain a recall of 1 throughout this set of experiments, as is clear from Figure 9(c). However, it had very low precision, as shown in Figure 9(b). Its highest precision was 0.133, and thus the algorithm seems impractical for real-life applications.

Space-Saving has a precision and recall of 1 for the entire range of k, as is clear from Figures 9(b) and 9(c). Meanwhile, *CountSketch* had recall/precision values ranging from 0.987 for top-75 to 1 for top-10, top-25, and top-50, which is satisfactory for real-life applications. However, Figures 9(a) and 9(d) show that *Space-Saving*'s run-time was 28 to 31 times less than that of *CountSketch*, while *Space-Saving*'s space was up to five times smaller.

Again, *Space-Saving* combined the lightweight property of *Probabilistic-InPlace*, and had better precision than *CountSketch*.

5.2 Real Data

For real data experiments, we used a click stream from Anonymous.com. The stream size was 25,000,000 hits, and the alphabet size was 4,235,870. The data was fairly skewed, but it was difficult to estimate the Zipf parameter. The sum of the counts of the frequent elements was small when compared to the length of the stream. For instance, the most frequent element, the top-10, the top-50, and the top-100 elements occurred 619,310, 1,726,609, 2,596,833, and 3,130,639

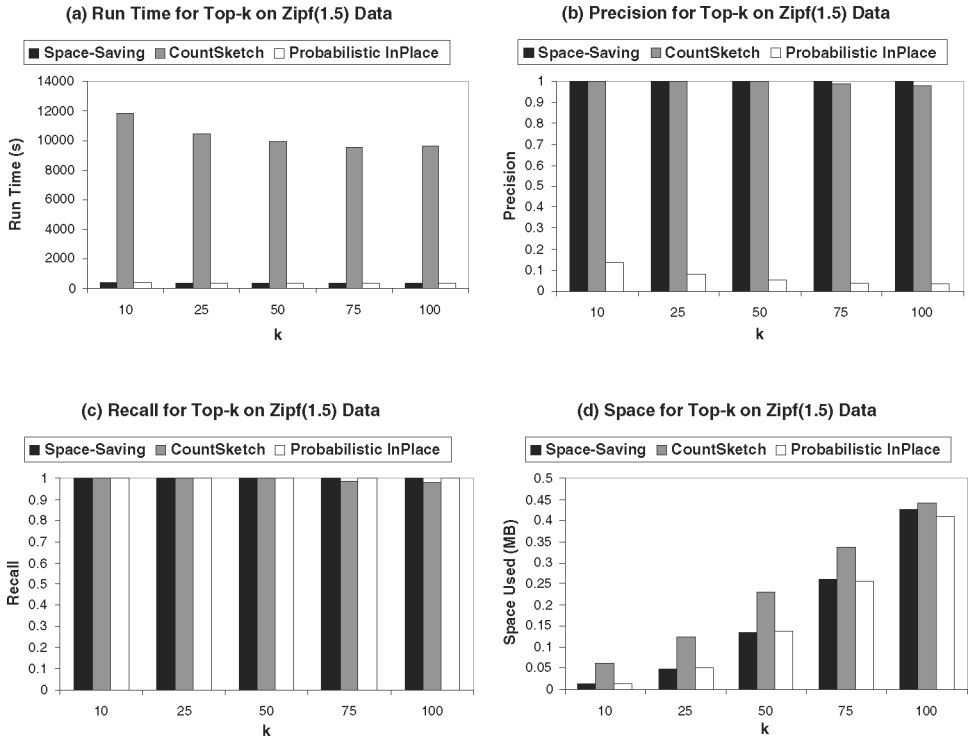


Fig. 9. Performance comparison for the top-k problem using synthetic Zipf(1.5) data—varying the k parameter.

times, respectively. Thus, it was very difficult to estimate the α from which we can a priori calculate a bound on the number of counters to be used. We made use of this set of experiments to evaluate the performance of *Space-Saving* on real data that has no known distribution. We varied the number of counters, m, with the query parameters. Surprisingly, in very restricted space, *Space-Saving* achieved substantial gains in run time and space with hardly any loss in precision and recall. On the whole, the results were very similar to those of the synthetic data experiments when the query parameters were varied. We will start by comparing the results of the algorithms when varying the query parameters, and will then comment on how *Space-Saving* guarantees its output.

5.2.1 *Varying the Query Parameters.* This set of experiments measured how the algorithms perform under different realistic query parameters.

5.2.1.1 *The Frequent Elements Problem.* For the frequent elements, the algorithms were used to find elements with minimum frequency $\lceil \phi N \rceil$. The parameter ϕ was varied from 0.001 to 0.01, and the number of elements monitored by space saving was fixed at $\frac{10}{\phi}$. The results are summarized in Figure 10.

From Figure 10(a), the run time of *Frequent* was consistently faster than *Space-Saving*, and *Space-Saving* used five times more space than *Frequent*, as

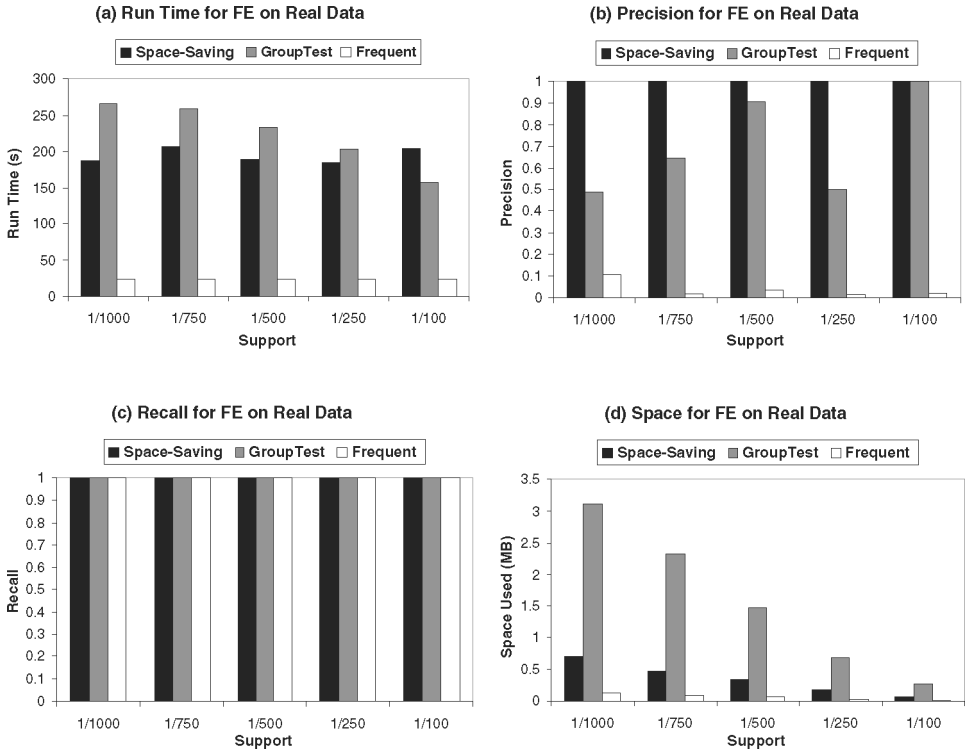


Fig. 10. Performance comparison for the frequent elements problem using a real click stream.

is clear from Figure 10(d). However, because of the excessive number of false positives reported by *Frequent*, its precision ranged from 0.011 to 0.035, as indicated by Figure 10(b).

For *GroupTest*, all the IDs of the alphabet were mapped to the range $1 \cdots 4,235,870$ so as to be able to compare it with *Space-Saving*, though we did not count the mapping lookup table as part of *GroupTest*'s space requirements. Despite the restricted space condition we imposed on *Space-Saving*, the algorithm was able to attain a value of 1 for precision and recall for all support levels, as is clear from Figures 10(b), and 10(c). However, *GroupTest* had a precision ranging from 0.486 to 1. On the other hand, from Figure 10(d), *Space-Saving* used space up to five times less than *GroupTest*, and ran faster most of the time, as shown in Figure 10(a).

5.2.1.2 *The Top-k Problem.* *Space-Saving*, *CountSketch*, and *Probabilistic-InPlace* were used to identify the top-k elements in the stream. The parameter k was varied, and the number of elements monitored by *Space-Saving* and *Probabilistic-InPlace* was fixed at $100k$. The results are shown in Figure 11.

Although *Probabilistic-InPlace* had good recall, as shown in Figures 11(c), its precision, as is clear from Figure 11(b), was worse than the two other

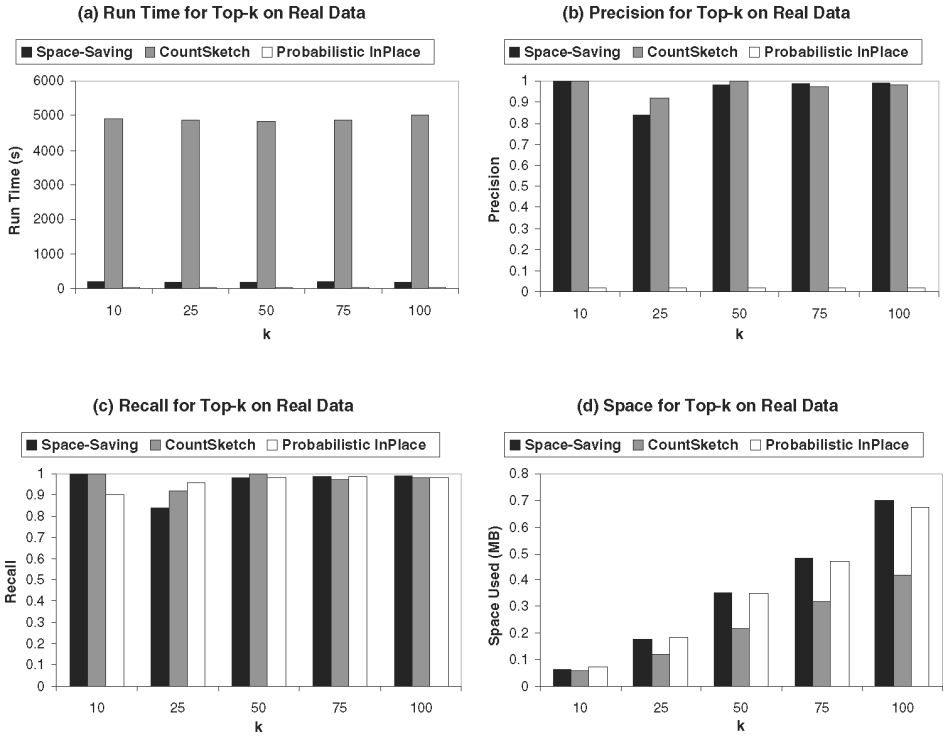


Fig. 11. Performance comparison for the top-*k* problem using a real click stream.

algorithms, since its highest precision was 0.02. The run-time of *Probabilistic-InPlace* was four to five times less than that of *Space-Saving*, and their space usages were close.

Interestingly, Figures 11(b) and 11(c) show that *Space-Saving* and *CountSketch* had very close recall and precision. The average precision and recall of *Space-Saving* and *CountSketch* were 0.96 and 0.97, respectively. However, Figure 11(a) shows that *Space-Saving*'s run-time was 25 times less than that of *CountSketch*. *Space-Saving*'s space requirements were 1.1 to 1.6 times larger, as shown in Figure 11(d).

5.2.2 *Measuring the* Guarantee *of* Space-Saving. We now introduce a new measure, *guarantee*. The guarantee metric is very close to precision, but is only measurable for algorithms that can offer guarantees about their output. Guarantee is the number of guaranteed correct elements as a percentage of the entire output, that is, the percentage of the output whose correctness is guaranteed. For instance, if an algorithm outputs 50 elements, from which it guarantees 42 to be correct, then the guarantee of this algorithm is 84%, even though some of the remaining eight elements might still be correct. Thus, the guarantee of a specific answer set is no greater than the precision, which is based on the number of correct, and not necessarily guaranteed, elements in the output.

Table I. *Space-Saving* Guarantee for the Frequent Elements Problem Using a Real Click Stream

Support	Number of Frequent Elements	Size of Output	Number of Guaranteed Frequent Elements	Guarantee	Precision
1/1000	18	18	18	1.0	1.0
1/750	11	11	11	1.0	1.0
1/500	10	10	10	1.0	1.0
1/250	2	2	2	1.0	1.0
1/100	2	2	2	1.0	1.0

Table II. *Space-Saving* Guarantee for the Top-k Problem Using a Real Click Stream

Number of Top-k	Size of Output	Number of Guaranteed Top-k Elements	Guarantee	Precision
10	10	10	1.0	1.0
25	25	20	0.80	0.84
50	50	46	0.92	0.98
75	75	72	0.96	0.9867
100	100	98	0.98	0.99

In the context of the frequent elements problem, the guarantee of *Space-Saving* is the number of elements whose guaranteed hits exceeds the user support, as a percentage of the entire output. Formally, this is equal to

$$\frac{Count(e_i|(count_i - \varepsilon_i) > \lceil \phi N \rceil)}{Count(e_i|count_i > \lceil \phi N \rceil))}.$$

In the context of the top-k problem, the guarantee of *Space-Saving* is the number of elements that are guaranteed to be in the top-k, that is, those whose guaranteed hits exceed $count_{k+1}$, as a percentage of the top-k. Formally, this is equal to

$$\frac{Count(e_i|(count_i - \varepsilon_i) > count_{k+1})}{k}.$$

It is worth noting that, throughout the set of experiments on synthetic data, the guarantee of *Space-Saving* was always constant at 1. That is, *Space-Saving* always guaranteed all its output to be correct.

Since it was not possible to estimate the α parameter of the real data set, we ran *Space-Saving* in a restricted space, and thus some of the experimental runs did not have a precision of 1. For this reason, we report both the guarantee and the precision of *Space-Saving* for both the frequent elements and the top-k problems in Tables I and II, respectively.

5.2.2.1 *The Frequent Elements Problem.* For the Frequent Elements problem, both the guarantee and the precision of *Space-Saving* were constant at 1.0, as is clear from Table I. That is, *Space-Saving outputs only the correct elements, nothing but the correct elements, and guarantees its output to be correct*.

5.2.2.2 *The Top-k Problem.* For the Top-k problem, the guarantee of *Space-Saving* ranged from 0.80 to 1.0, and the precision of *Space-Saving* ranged from 0.84 to 1.0, as is clear from Table II. In other words, *Space-Saving* was able to

guarantee 80% to 100% of its output to be correct. Throughout the experimental runs, the number of nonguaranteed elements was at most five.

6. ANSWERING CONTINUOUS QUERIES

After validating the theoretical analysis by experimental evaluation using both real and synthetic data, we extend the proposed algorithm to answer continuous queries about both frequent and top-k elements. Although incremental reporting of the answer is useful in many applications for monitoring interesting elements, we are not aware of any proposed solution for this problem. The main goal is to incrementally report any changes taking place in the answer set, without scanning all the monitored elements. Since these changes can take place after any stream observation, the *Increment-Counter* algorithm has to be modified to check for changes in the answer set, so that the cache is updated before it is used for the next advertisement rendering. The extensions to *Increment-Counter* are discussed below.

6.1 Continuous Queries for Frequent Elements

Incremental reporting of frequent elements can be classified into two types of reporting. The first type is reporting an infrequent element that has become frequent. This can happen when an element receives a hit that makes its frequency satisfy the minimum support $\lceil \phi N \rceil$. This can only happen for the observed element. The second type of updates is reporting that a group of frequent elements have become infrequent. This can happen because the minimum support $\lceil \phi N \rceil$ has increased as N gets incremented. Several elements may become infrequent after the last stream observation. Moreover, one stream observation can result in both types of updates.

Checking for updates of both types is more effective than running the *QueryFrequent* algorithms after every observation, that is, after the call to *Increment-Counter*. The subroutine *ContinuousQueryFrequent* that should be called at the end of each call to *Increment-Counter* and before the cleanup step is sketched in Figure 12.

ContinuousQueryFrequent should maintain a pointer, ptr_ϕ, to $Bucket_\phi$, the bucket of minimum value that satisfies the support. Initially, this pointer points to the initial bucket of the *Stream-Summary*. At the end of each call to the *Increment-Counter* algorithm and before deleting the empty bucket, it should invoke *ContinuousQueryFrequent*. *ContinuousQueryFrequent* should check if $Bucket_\phi$ still satisfies the required support after the stream size N has been incremented. If it does not satisfy the support any more, all the elements in the child list of $Bucket_\phi$ should be reported as frequent elements that have become infrequent, and ptr_ϕ should be moved to $Bucket_\phi^+$, the neighbor of $Bucket_\phi$ with larger value.

When the observed element e_i has its counter $count_i$ incremented, *ContinuousQueryFrequent* should check the new bucket of $count_i$, $Bucket_i'$. If $count_i$ has moved from an infrequent bucket to another infrequent bucket, or from a frequent bucket to another frequent bucket, then there is no need to update the set of frequent elements. Only if the new bucket of $count_i$ satisfies $\lceil \phi N \rceil$ and the

```
Algorithm: ContinuousQueryFrequent(counter count_i)
begin
  //Incrementing the stream size
  N ++;
  //Reporting elements that are becoming infrequent
  let Bucket_φ be the bucket ptr_φ points to
  let Bucket_φ^+ be Bucket_φ's neighbor of larger value
  if (Bucket_φ < ⌈φN⌉){
    Report Bucket_φ's child-list as infrequent;
    Move ptr_φ to Bucket_φ^+;
  }
  //Reporting e_i if it becomes frequent
  let Bucket'_i be the new bucket of count_i
  let Bucket'_φ be the new bucket ptr_φ points to
  if (Bucket'_i > ⌈φN⌉ AND Bucket'_i ≤ Bucket'_φ){
    let e_i be the element of count_i
    Report e_i as frequent;
    Move ptr_φ to Bucket'_i;
  }
end;
```

Fig. 12. Incremental reporting of frequent elements.

old bucket did not, e_i should be reported as an infrequent element that is now frequent. In this case, ptr_ϕ should be moved to point to $Bucket'_i$, the new bucket of $count_i$, since we are sure then that this is the bucket of minimum value that satisfies the support. The algorithm *ContinuousQueryFrequent* checks for this condition by making sure that $Bucket'_i$ has a value which satisfies the support, and its value is no greater than the value of the bucket pointed to by ptr_ϕ.

Reporting an element that is becoming frequent is O (1); and reporting a group of elements that are becoming infrequent is O (|elements becoming infrequent|). Thus, *ContinuousQueryFrequent* takes O (|updated elements|) to update the cache.

In the *Increment-Counter* algorithm, the old bucket of $count_i$ is deleted if its child list is empty. The *ContinuousQueryFrequent* algorithm should be called before deleting the old bucket of $count_i$. Otherwise, ptr_ϕ could be pointing to a deleted bucket, and there would be no efficient way to know which bucket is $Bucket_\phi^+$, except by scanning all the buckets in the *Stream-Summary* data structure, which is not a constant time operation.

6.2 Continuous Queries for Top-k Elements

Answering continuous queries about top-k is similar to answering continuous queries about frequent elements. *ContinuousQueryTop-k* should maintain a pointer, ptr_k, to $Bucket_k$, the bucket to which $count_k$ belongs, where $count_k$ is the counter at the kth position in the *Stream-Summary* data structure. Hence, the top-k elements should be elements that belong to all the buckets with values no less than the value of $Bucket_k$. However, there might be more than k elements that belong to buckets with values no less than that of $count_k$. For instance, if $k = 100$, and the buckets with values more than $count_k$ have 95 elements, and $Bucket_k$ has more than five elements, then some elements that belong to $Bucket_k$ will not be reported among the top-k. In case $Bucket_k$ has more elements than needed to report the top-k, *ContinuousQueryTop-k* should report a subset

of the elements of $Bucket_k$ as being among the top-k. The rest of the $Bucket_k$ elements, even though they have the same value as $count_k$, are not reported as being among the top-k. Thus, *ContinuousQueryTop-k* should maintain a set Set_k of elements that belong to $Bucket_k$, and have been reported as Top-k. Initially, Set_k is set empty, and ptr_k points to the initial bucket of the *Stream-Summary*.

The underlying idea is to keep track of *boundary* elements that lie on the boundary between the top-k and the non-top-k elements. Such elements can move from outside the top-k to inside the top-k, if their frequency increases. Only an element that belong to $Bucket_k$ that is not a member of Set_k can be reported as an element which is entering the top-k set of elements, if it receives a hit. Elements which belong to Set_k will not change the top-k if they receive hits. Other elements that belong to buckets other than $Bucket_k$ will not effect the top-k if they receive hits.

The *Stream-Summary* data structure needs to be modified slightly, so that it can tell if k distinct elements have been observed in the stream. This modification helps at the transient start, when all distinct elements observed are among the top-k.

Telling whether or not k distinct elements have been observed in the stream is an easy problem. It is enough to keep a counter that is incremented every time an element is deleted from the initial bucket in the *Stream-Summary*, and is inserted into a bucket of value 1. However, for simplicity, we will delete these details from the algorithm, and assume an oracle exists, which will answer the question for us.

After receiving more than k distinct elements, a new element reported as being among the top-k implies that another element is no longer in the top-k. The algorithm *ContinuousQueryTop-k* is responsible for this task, and should be called at the end of each call to *Increment-Counter* and before the clean up step, as sketched in Figure 13.

The first two cases in *ContinuousQueryTop-k* handle the special cases when the distinct elements in the stream are no more than k. In Case 1, the algorithm checks if the number of distinct elements observed is strictly less than k. If this is true, then e_i, the observed element, should be reported among the top-k if this is the first occurrence of e_i. In Case 2, if e_i is the kth distinct element reported, then the number of distinct elements has changed from $k-1$ to k because of the last observation, e_i. Thus, in addition to reporting e_i as being among the top-k, ptr_k has to be moved to $Bucket_1$, the bucket of value 1. Since e_i is the kth distinct element, the top-k are all the elements in all the buckets with values no less than 1. Thus Set_k should include all the elements that belong to $Bucket_1$.

Case 3 is the general case. This case is executed only if e_i moves from $Bucket_k$ to $Bucket_k^+$, the neighbor of $Bucket_k$ with larger value. If e_i was already among the top-k, that is, it did belong to Set_k, then the top-k elements did not change, and it needs to be deleted from Set_k, since it does not belong to $Bucket_k$ any more. However, if e_i is a boundary element, that is, it did not belong to Set_k, then e_i is moving from outside top-k to inside top-k. Thus e_i has to be reported as being among the top-k. In addition, an element has to be picked from Set_k, deleted from Set_k, and reported as a non-top-k element.

```
Algorithm: ContinuousQueryTop-k(counter count_i)
begin
  let e_i be the element of count_i
  //Case 1: not all top-k have been reported
  if less than k distinct elements are recieved
    if (count_i = 1)
      Report e_i as among the top-k;
  //Case 2: e_i is the k^th element reported
  if e_i is the k^th distinct element recieved{
    Report e_i as among the top-k;
    let Bucket_1 be the bucket of value 1
    Move ptr_k to Bucket_1;
    Insert Bucket_1's child-list into Set_k;
  }
  //Case 3: The general case
  //k elements have been already reported as top-k
  let Bucket_k be the bucket ptr_k points to
  let Bucket_k^+ be Bucket_k's neighbor of larger value
  let Bucket'_i be the new bucket of count_i
  if (Bucket'_i = Bucket_k^+){
    if (e_i ∈ Set_k){
      Delete e_i from Set_k;
    }else{
      Select any element e from Set_k;
      Delete e from Set_k;
      Report e as not among top-k;
      Report e_i as being among top-k;
    }
    if Set_k is empty{
      Move ptr_k to Bucket_k^+;
      Insert Bucket_k^+'s child-list into Set_k;
    }
  }
end;
```

Fig. 13. Incremental reporting of top-k.

Whether e_i belongs to Set_k or not, the deletion of an element from Set_k might leave Set_k empty. In this case, we are sure that there are exactly k elements in the buckets with values more than that of $Bucket_k$. Those are the top-k elements. Hence ptr_k should be moved to point to $Bucket_k^+$, the neighbor of $Bucket_k$ with larger value, and Set_k should be initialized to contain all the elements in the child list of $Bucket_k^+$.

Since Set_k can have at most k elements at a time, we assume it can be stored in an associative memory, and thus, all the operation on Set_k is $O(1)$. Otherwise, it can be stored in a hash table, and the amortized cost of any operation will still be $O(1)$. It is easy to see that the amortized cost of *ContinuousQueryTop-k* is constant. Although the step of inserting all the elements of one bucket into Set_k is not $O(1)$, this cost will be amortized since Set_k will have exactly one element deleted every time an element moves from $Bucket_k$ to $Bucket_k^+$. Thus, on average, one element will be inserted and another will be deleted from Set_k for every element moving from $Bucket_k$ to $Bucket_k^+$, which is $O(1)$ per observation.

Like *ContinuousQueryFrequent*, *ContinuousQueryTop-k* should be called before deleting the old bucket of $count_i$. Otherwise, ptr_k could be pointing to a deleted bucket, and there would be no constant time method to know which bucket is $Bucket_k^+$.

7. DISCUSSION

This article 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 bounds on the error rate for approximate counts of elements, using very limited space. We showed that *Space-Saving* can handle both the frequent elements and top-k queries because it efficiently estimates the elements' frequencies. The memory requirements were analyzed with special attention to the case of skewed data. Moreover, this article introduced and motivated the problem of answering continuous queries about top-k, and frequent elements, through incremental reporting of changes to the answer sets. Minor extensions were applied to use the same set of algorithms to answer continuous queries. We conducted extensive experiments using both synthetic and real data sets to validate the benefits of the proposed algorithm.

This is the first algorithm, to the best of our knowledge, which guarantees the correctness of the frequent elements as well as the correctness and the order of the top-k elements, when the data is skewed.

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 which have occurred 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 no other algorithm has better space bounds than those of *Space-Saving*.

The main practical strengths of *Space-Saving* is that it can use whatever space is available 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 tuning. It is interesting that running the algorithm on the available space ensures that more important elements are less susceptible to noise.

ACKNOWLEDGMENT

The authors thank the associate editor, Prof. H. Mannila, and the anonymous referees for insightful comments on the article.

REFERENCES

ALON, N., MATIAS, Y., AND SZEGEDY, M. 1996. The space complexity of approximating the frequency moments. In *Proceedings of the 28th ACM STOC Symposium on the Theory of Computing*. 20–29.

ARASU, A., BABU, S., AND WIDOM, J. 2003a. CQL: A language for continuous queries over streams and relations. In *Proceedings of the 9th DBPL International Conference on Data Base and Programming Languages*. 1–11.

ARASU, A., BABU, S., AND WIDOM, J. 2003b. The CQL Continuous Query Language: Semantic foundations and query execution. Tech. rep. 2002-67. Stanford University, Stanford, CA.

BABCOCK, B., BABU, S., DATAR, M., MOTWANI, R., AND WIDOM, J. 2002. Models and issues in data stream systems. In *Proceedings of the 21st ACM PODS Symposium on Principles of Database Systems*. 1–16.

BABCOCK, B. AND OLSTON, C. 2003. Distributed top-k monitoring. In *Proceedings of the 22nd ACM SIGMOD International Conference on Management of Data*. 28–39.

BLOOM, B. H. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM 13*, 7, 422–426.

BONNET, P., GEHRKE, J., AND SESHADRI, P. 2001. Towards sensor database systems. In *Proceedings of the 2nd IEEE MDM International Conference on Mobile Data Management*. 3–14.

BOSE, P., KRANAKIS, E., MORIN, P., AND TANG, Y. 2003. Bounds for frequency estimation of packet streams. In *Proceedings of the 10th SIROCCO International Colloquium on Structural Information and Communication Complexity*. 33–42.

BOYER, R. AND MOORE, J. 1981. A fast majority vote algorithm. Tech. rep. 1981-32. Institute for Computing Science, University of Texas, Austin, Austin, TX.

CHARIKAR, M., CHEN, K., AND FARACH-COLTON, M. 2002. Finding frequent items in data streams. In *Proceedings of the 29th ICALP International Colloquium on Automata, Languages and Programming*. 693–703.

CHEN, J., DEWITT, D., TIAN, F., AND WANG, Y. 2000. NiagaraCQ: A scalable continuous query system for Internet databases. In *Proceedings of the 19th ACM SIGMOD International Conference on Management of Data*. 379–390.

CORMODE, G., KORN, F., MUTHUKRISHNAN, S., AND SRIVASTAVA, D. 2003. Finding hierarchical heavy hitters in data streams. In *Proceedings of the 29th VLDB International Conference on Very Large Data Bases*. 464–475.

CORMODE, G., KORN, F., MUTHUKRISHNAN, S., AND SRIVASTAVA, D. 2004. Diamond in the rough: Finding hierarchical heavy hitters in multi-dimensional data. In *Proceedings of the 23rd ACM SIGMOD International Conference on Management of Data*. 155–166.

CORMODE, G. AND MUTHUKRISHNAN, S. 2003. What's hot and what's not: Tracking most frequent items dynamically. In *Proceedings of the 22nd ACM PODS Symposium on Principles of Database Systems*. 296–306.

CORTES, C., FISHER, K., PREGIBON, D., ROGERS, A., AND SMITH, F. 2000. Hancock: A language for extracting signatures from data streams. In *Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 9–17.

DATAR, M., GIONIS, A., INDYK, P., AND MOTWANI, R. 2002. Maintaining stream statistics over sliding windows. In *Proceedings of the 13th ACM SIAM Symposium on Discrete Algorithms*. 635–644.

DEMAINE, E., LÓPEZ-ORTIZ, A., AND MUNRO, J. 2002. Frequency estimation of internet packet streams with limited space. In *Proceedings of the 10th ESA Annual European Symposium on Algorithms*. 348–360.

ESTAN, C. AND VARGHESE, G. 2003. New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice. *ACM Trans. Comput. Syst. 21*, 3, 270–313.

FANG, M., SHIVAKUMAR, S., GARCIA-MOLINA, H., MOTWANI, R., AND ULLMAN, J. 1998. Computing iceberg queries efficiently. In *Proceedings of the 24th VLDB International Conference on Very Large Data Bases*. 299–310.

FEIGENBAUM, J., KANNAN, S., STRAUSS, M., AND VISWANATHAN, M. 1999. An approximate L1-difference algorithm for massive data streams. In *Proceedings of 40th FOCS Annual Symposium on Foundations of Computer Science*. 501–511.

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

FLAJOLET, P. AND MARTIN, G. 1985. Probabilistic counting algorithms. *J. Comput. Syst. Sci. 31*, 182–209.

GEHRKE, J., KORN, F., AND SRIVASTAVA, D. 2001. On computing correlated aggregates over continual data streams. In *Proceedings of the 20th ACM SIGMOD International Conference on Management of Data*. 13–24.

GIBBONS, P. AND MATIAS, Y. 1998. New sampling-based susmary statistics for improving approximate query answers. In *Proceedings of the 17th ACM SIGMOD International Conference on Management of Data*. 331–342.

GILBERT, A., KOTIDIS, Y., MUTHUKRISHNAN, S., AND STRAUSS, M. 2001. Surfing wavelets on streams: One-pass summaries for approximate aggregate queries. In *Proceedings of the 27th VLDB International Conference on Very Large Data Bases*. 79–88.

GOLAB, L., DEHAAN, D., DEMAINE, E., LÓPEZ-ORTIZ, A., AND MUNRO, J. 2003. Identifying frequent items in sliding windows over online packet streams. In *Proceedings of the 1st ACM SIGCOMM Internet Measurement Conference*. 173–178.

GOLAB, L. AND OZSU, M. 2003. Issues in data stream management. *ACM SIGMOD Rec. 32*, 2, 5–14.

GREENWALD, M. AND KHANNA, S. 2001. Space-efficient online computation of quantile summaries. In *Proceedings of the 19th ACM SIGMOD International Conference on Management of Data*. 58–66.

GUHA, S., INDYK, P., MUTHUKRISHNAN, M., AND STRAUSS, M. 2002. Histogramming data streams with fast per-item processing. In *Proceedings of the 29th ICALP International Colloquium on Automata, Languages and Programming*. 681–692.

GUHA, S., KOUDAS, N., AND SHIM, K. 2001. Data-streams and histograms. In *Proceedings of the 33rd ACM STOC Symposium on the Theory of Computing*. 471–475.

GUNDUZ, S. AND OZSU, M. 2003. A Web page prediction model based on click-stream tree representation of user behavior. In *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 535–540.

GUPTA, P. AND MCKEOWN, N. 1999. Packet classification on multiple fields. In *Proceedings of the ACM SIGCOMM International Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. 147–160.

HAAS, P., NAUGHTON, J., SEHADRI, S., AND STOKES, L. 1995. Sampling-based estimation of the number of distinct values of an attribute. In *Proceedings of the 21st VLDB International Conference on Very Large Data Bases*. 311–322.

HOARE, C. 1961. Algorithm 65: Find. *Commun. ACM 4*, 7, 321–322.

JIN, C., QIAN, W., SHA, C., YU, J., AND ZHOU, A. 2003. Dynamically maintaining frequent items over a data stream. In *Proceedings of the 12th ACM CIKM International Conference on Information and Knowledge Management*. 287–294.

KARP, R., SHENKER, S., AND PAPADIMITRIOU, C. 2003. A simple algorithm for finding frequent elements in streams and bags. *ACM Trans. Database Syst. 28*, 1, 51–55.

KIRSCHENHOFER, P., PRODINGER, H., AND MARTINEZ, C. 1997. Analysis of Hoare's FIND algorithm with median-of-three partition. *Random Struct. Algorith. 10*, 1–2, 143–156.

LIN, X., LU, H., XU, J., AND YU, J. 2004. Continuously maintaining quantile summaries of the most recent N elements over a data stream. In *Proceedings of the 20th IEEE ICDE International Conference on Data Engineering*. 362–374.

MANKU, G. AND MOTWANI, R. 2002. Approximate frequency counts over data streams. In *Proceedings of the 28th VLDB International Conference on Very Large Data Bases*. 346–357.

MANKU, G., RAJAGOPALAN, S., AND LINDSAY, B. 1999. random sampling techniques for space efficient online computation of order statistics of large datasets. In *Proceedings of the 18th ACM SIGMOD International Conference on Management of Data*. 251–262.

MATIAS, Y., VITTER, J., AND WANG, M. 2000. Dynamic maintenance of wavelet-based histograms. In *Proceedings of the 26th VLDB International Conference on Very Large Data Bases*. 101–110.

METWALLY, A., AGRAWAL, D., AND EL ABBADI, A. 2005a. Duplicate detection in click streams. In *Proceedings of the 14th WWW International World Wide Web Conference*. 12–21. An extended version appears as a University of California, Santa Barbara, Department of Computer Science Technical Report 2004-23.

METWALLY, A., AGRAWAL, D., AND EL ABBADI, A. 2005b. Efficient computation of frequent and top-k elements in data streams. In *Proceedings of the 10th ICDT International Conference on Database Theory*. 398–412. An extended version appears as a University of California, Santa Barbara, Department of Computer Science, Technical Report 2005-23.

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

WHANG, K., VANDER-ZANDEN, B., AND TAYLOR, H. 1990. A linear-time probabilistic counting algorithm for database applications. *ACM Trans. Database Syst. 15*, 208–229.

ZHU, Y. AND SHASHA, D. 2002. StatStream: Statistical monitoring of thousands of data streams in real time. In *Proceedings of the 28th VLDB International Conference on Very Large Data Bases*. 358–369.

ZIPF, G. 1949. *Human Behavior and The Principle of Least Effort*. Addison-Wesley, Reading, MA.

Received January 2005; revised January 2006, May 2006; accepted June 2006