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

Ahmed Metwally † Divyakant Agrawal Amr El Abbadi

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

Abstract

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 no 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.

1 Introduction

More than a decade ago, both the industry and the research communities realized the benefit of statistically analyzing vast amounts of historical data in discovering 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 evolved aggressively. For some applications, data is being generated at a rate high enough to make its long-term storage cost outweighs its benefits. Hence, such streams of data are stored temporarily, and should be mined fast 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 [4].

There is a growing need to develop new techniques to cope with high-speed streams, and answer online queries. Currently, data stream management systems are used for monitoring click streams [31], stock tickers [11, 46], sensor readings [7], telephone call records [15], network packet traces [17], auction bidding patterns [2], traffic management [3], network-aware clustering [12], and security against DoS [12]. [27] 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 [1], differences [20], distinct values estimation [22, 33, 45], bit counting [16], duplicate detection [42], approximate quantiles [28, 38, 40], histograms [30, 29], wavelet based aggregate queries [25, 41], correlated aggregate queries [23], elements classification [32], frequent elements [8, 13, 14, 17, 18, 19, 26, 35, 36, 39, 43], and top-k queries [5, 10, 17, 24, 43]. Earlier results on data streams were presented in [9, 21].

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, e.g., an impression (advertisement rendering) to a 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 script that refers the Browser to the commissioner's server that loads the advertisements, and logs the advertisement impression. Whenever a 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, who loads its own Web page on the surfer's Browser. A commissioner earns a commission on the advertisers' pay-

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

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

ments to the publishers. Therefore, a 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 surfers click them. On average, one click on a Pay-Per-Click advertisement generates as much revenue as rendering 500 Pay-Per-Impression advertisements. 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 which 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 could 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 Gigabytes 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. 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 microseconds, 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 to 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 paper, we propose an integrated online streaming algorithm for solving both problems of finding the top-k elements, and 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 solves 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 paper 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 in Section 7.

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 $0 \leq \phi \leq 1$; whereas 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 [10, 17], and are hence, impractical for applications with large alphabets. Thus, several relaxations of the original problems were proposed.

[1] The term "Heavy Hitters" was also used in [12].

2.1 Variations of the Problems

The FindCandidateTop(S, k, l) problem was proposed in [10] 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) [10] 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 k^{th} rank. That is, all the reported k elements have frequency very close to the k^{th} element. The Hot Items[2] problem is a special case of the frequent elements problem, proposed in [44], that asks for k elements, each of which has frequency more than $\frac{N}{k+1}$. This extends the early work done in [9, 21] for identifying a majority element. The most popular variation of the frequent elements problem, finding the ϵ-Deficient Frequent Elements [39], 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 [10, 14, 17, 18, 35, 36, 39] 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 [43]. 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 updated every time E_i is observed in the stream. If the observed ID is not monitored, i.e., there is no counter kept for this element, it is either disregarded, or some algorithm-dependent action is taken.

The *Sticky Sampling* algorithm [39] slices S into rounds of non-decreasing length. The probability an element is added to the list of counters, i.e. being monitored, decreases as the round length increases. At rounds' boundaries, for every monitored element, a coin is tossed 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 [39] breaks S up into equal rounds of length $\frac{1}{\epsilon}$. Throughout every round, non-monitored items 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 item is added in round r, it is given the benefit of doubt, its initial count is set to $r-1$, and the maximum possible over-estimation, $r-1$, is recorded for the new item. 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 $\lceil (\phi - \epsilon)N \rceil$.

[17] proposed the *Frequent* algorithm to solve the Hot Items problem. *Frequent*, a re-discovery of the algorithm proposed in [44], outputs a list of k elements with no guarantee on which elements, if any, have frequency more than $\frac{N}{k+1}$. The same algorithm was proposed independently in [36]. *Frequent* extends the early work done in [9, 21] for finding a majority item, using only one counter. The algorithm in [9, 21] monitors the first item in the stream. For each observation, the counter is incremented if the observed item 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 results. *Frequent* [17] keeps k counters to monitor k elements. If a monitored element is observed, its counter is incremented, else all counters are decremented. In case any counter reaches 0, it is assigned the next observed element. [17] also proposed a lightweight data structure that can decrement all counters in $O(1)$ operations. The sampling algorithm *Probabilistic-InPlace* [17] 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 $\frac{m}{2}$ 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 Ω(distinct values of the deleted counters), and thus, trades precision and constant per-item processing for counters' accuracy.

先填充一半，再减去一半

In general, counter-based techniques have fast per-item 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 bit-maps of counters. Usually, each element is hashed into the space of counters using a family of hash functions, and the hashed-to counters are updated for every hit of this element. The "representative" counters are then queried for the element frequency with expected loss of accuracy due to hashing collisions.

The probabilistic *CountSketch* algorithm, proposed in [10], solves the FindApproxTop(S, k, ϵ) problem. The space requirements of *CountSketch* decreases 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 $F_{k+1} \ldots F_{|A|}$. The family of hashing functions employed hashes every ID to its representative counters, such that, some counters are incremented, and the others are decremented, for every occurrence of this element in the stream. The approximate frequency of the element is estimated by finding the median from its representative counters. A heap of top-k elements is kept, and if the estimated frequency of the observed element exceeds the smallest estimated counter in the heap, the smallest element is replaced by the observed element.

The *GroupTest* algorithm, proposed in [14], answers queries about Hot Items, with a constant probability of

[2] The term "Hot Items" was coined later in [14].

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 $1 \dots |A|$. 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 universal hash functions to select the elements in each *FindMajority* system. When queried, the algorithm discards systems with more than one, or with no Hot Items. Also proposed is 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 in [18], which was also independently proposed in [35], is similar to *GroupTest*. Using the idea of Bloom's Filters [6], the *Multistage filters* algorithm hashes every element to a number of counters, that are updated 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 in [18] 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, [18] proposes some techniques to reduce the over-estimation errors in counters. Once an element is added to the Flow Memory, its counters are not monitored anymore by the *Multistage filters*. In addition, [18] proposed incrementing only the counter(s) of the minimum value.

The *hCount* algorithm [35], does not employ the error reduction techniques employed in [18]. 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 over-estimation 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 pre-process the stream for a top-k query, several frequent elements queries have to be issued to reach a lower bound on the frequency of the k^{th} element; and in order to use top-k queries to pre-process the stream for a frequent elements query, several top-k queries have to be issued to reach an upper bound on the number of frequent elements. To offer an integrated solution, we have generalized both problems to *accurately estimate the frequencies of significant*[3] *elements, and store these frequencies in an always-sorted structure*. We, then, devise a generalized algorithm for the generalized problem.

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 does not always have a panoramic understanding of the application data to issue meaningful queries. Many times, 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 propose our counter-based *Space-Saving* algorithm and its associated *Stream-Summary* data structure. The underlying idea is to maintain partial information of interest; i.e., 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.

In an ideal situation, any significant element, E_i, with rank i, that has received F_i hits, should be accommodated in the i^{th} counter. However, due to errors in estimating the frequencies of the elements, the order of the elements in the data structure might not reflect their exact ranks. For this reason, we will denote the counter at the i^{th} position in the data structure as $count_i$. The counter $count_i$ estimates the frequency f_i, of some element e_i. If the i^{th} position in the data structure has the right element, i.e., the element with the i^{th} rank, E_i, then $e_i = E_i$, and $count_i$ is an estimation of F_i.

The algorithm is straightforward. If 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. The new element, e, could have actually occurred between 1 and $min + 1$ times. We assign $count_m$ the value

[3]The significant elements are interesting elements that can be output in meaningful 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;

Figure 1: The *Space-Saving* Algorithm

$min + 1$, since we designed the algorithm to err only on the positive side, i.e., to never miss a frequent element.

For each monitored element e_i, we keep track of its maximum over-estimation, ε_i, resulting from the initialization of its counter when it was inserted into the list. That is, when starting to monitor e_i, set ε_i to the counter value that was evicted. Keeping track of the over-estimation error for each elements is only 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.

In general, the top elements among non-skewed 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. Frequent elements will reside in the counters of bigger values, and will not be distorted by the ineffective hits of the infrequent elements, and thus, will never be replaced out of the monitored counters. Meanwhile, the numerous infrequent elements will be striving to reside in the smaller counters, whose values grow slower than those of the larger counters.

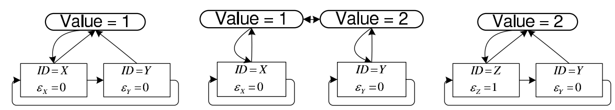
In addition, if the skew remains, but the popular elements change over time, the algorithm adapts automatically. The elements that are growing more popular will gradually be pushed to the top of the list as they receive more hits. If one of the previously popular elements loses its popularity, it will receive less hits. Thus, its relative position will decline, as other counters get incremented, and it might eventually get dropped from the list.

Even if the data is not skewed, the errors in the counters 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 over-estimation errors in counters' values.

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

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;

Figure 2: The *Increment-Counter* Algorithm



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

Figure 3: Example of updates to *Stream-Summary* with $m = 2$

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; and 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 1 *Assuming $m = 2$, and $A = \{X, Y, Z\}$. The stream $S = X, Y$ will yield the* Stream-Summary *in Figure 3(a), after the two counters accommodate the observed* *elements. When another Y arrives, a new bucket is created with value 2, and Y gets attached to it, as shown in Figure 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 in [17]. However, to look up a value of a counter using the data structure proposed in [17], it takes $O(m)$, while *Stream-Summary* look-ups 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 1 *The length, N, of the stream is equal to the sum of all the counters in the* Stream-Summary *data structure. That is, $N = \sum_{i \le m}(count_i)$*

Proof. Every hit in S increments only one counter among the m counters. This is true even when a replacement happens, i.e., the observed hit e was not previously monitored, and it replaces another counter e_m. This is because $count_m$ was 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 $N = 4$. $S = X, Z, Y, Y$ 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.

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 1 we deduce the following.

Lemma 2 *Among all counters, the minimum counter value, min, is no greater than $\lfloor \frac{N}{m} \rfloor$.*

Proof. Lemma 1 can be rewritten as:

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

All the terms in the summation of Equation 1 are non-negative, i.e., all counters are no smaller than min, hence $min \le \lfloor \frac{N}{m} \rfloor$. □

We are interested in min since it represents an upper bound on the over-estimation in any counter in *Stream-Summary*. This relation is established in Lemma 3.

Lemma 3 *For any element e_i in the* Stream-Summary*, $0 \le \varepsilon_i \le min$, i.e., $f_i \le (f_i + \varepsilon_i) = count_i \le f_i + min$.*

Proof. From the algorithm, the over-estimation of e_i, ε_i, is non-negative, because any observed element is always given the benefit of doubt. The over-estimation ε_i is always assigned the value of the minimum counter at the time e_i started being observed. Since the value of the minimum counter monotonically increases over time until it reaches the current min, then for all monitored elements $\varepsilon_i \le min$. □

Moreover, any element E_i, with frequency $F_i > min$, is guaranteed to be monitored, as shown next.

Theorem 1 *An element E_i with $F_i > min$, must exist in* 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, 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 1 and Lemma 3, we can infer an interesting general rule about the over-estimation of elements' counters. For any element E_i, with rank $i \le m$. The frequency of E_i, F_i, is no more than $count_i$, the counter occupying the i^{th} 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 2 *Whether or not E_i occupies the i^{th} position in the* Stream-Summary*, $count_i$, the counter at position i, is no smaller than F_i, the frequency of the element with rank i, E_i.*

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

- The element E_i is not monitored. Thus, from Theorem 1, $F_i \le min$. 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, 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, $count_i \ge F_i$.

- The element E_i is at position i. From Lemma 3, $count_i \ge f_i = F_i$.

- 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, and $x < i$, then the estimated frequency of E_x is no smaller than F_i. Since $y \geq i$, then the $count_i \geq count_y$, 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 2 is significant, since it enables estimating an upper bound 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, $rank(e_i) > Count(e_j|(count_j - \varepsilon_j) > count_i)$. Thus, Theorem 2 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, we sequentially traverse *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.

Next, we determine the value of m that guarantees a user specified error rate, ϵ.

4.1.1 The General Case

We will analyze the space requirements for the general case of any data distribution.

Theorem 3 *Assuming no specific data distribution,* Space-Saving *uses a number of counters of* $\min(|A|, \frac{1}{\epsilon})$ *to find all frequent elements with error* ϵ*. Any element,* e_i*, with frequency* $f_i > \phi N$ *is guaranteed to be reported.*

```
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;
```

Figure 4: Reporting Frequent Elements

Proof. From Theorem 1, any element e_i whose $f_i > min$ is guaranteed to be in the *Stream-Summary*; and since the upper bound of ε_i is min, from Lemma 2, it follows that $\varepsilon_i \leq min \leq \lfloor \frac{N}{m} \rfloor$. If we set $min = \epsilon N$, then $m \geq \frac{1}{\epsilon}$ guarantees an error rate of ϵ. Since $\phi \geq \epsilon$, from Theorem 1, any element with frequency greater than ϕN is monitored in the *Stream-Summary*, and hence is guaranteed to be reported. □

The bound of Theorem 3 is tight. For instance, this can happen if all the IDs in the stream are distinct. In addition, Theorem 3 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.

Theorem 4 *Any deterministic counter-based algorithm uses a number of counters of at least* $\min(|A|, \frac{1}{2\epsilon})$ *to find all frequent elements with error* ϵ*.*

Proof. The proof is similar to that given in [8]. 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 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 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 $\frac{1}{2\epsilon}$. □

4.1.2 Zipf Distribution Analysis

A Zipfian [47] data set, with parameter α, has the frequency, F_i, of an element, E_i, with the i^{th} rank, such that $F_i = \frac{N}{i^\alpha \zeta(\alpha)}$, where $\zeta(\alpha) = \sum_{i=1}^{|A|} \frac{1}{i^\alpha}$ converges to a small constant inversely proportional to α, except for $\alpha \leq 1$.

For instance, $\zeta(1) \approx \ln(1.78|A|)$. 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, 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 1 that:

Lemma 4 *An element E_i, has $F_i > min_{max}$, and regardless of the ordering of S, is guaranteed to be monitored, if and only if $i \leq r$.*

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

Theorem 5 *Assuming noiseless Zipfian data with parameter α, to calculate the frequent elements with error rate ϵ,* Space-Saving *uses only* $\min(|A|, (\frac{1}{\epsilon})^{\frac{1}{\alpha}}, \frac{1}{\epsilon})$ *counters. This is regardless of the stream permutation.*

Proof. From Equation 1, and Lemma 3, $min_{max} \geq \frac{N - \sum_{i \leq m} f_i}{m}$, from which it can be ascertained that $min_{max} \geq \frac{N - \sum_{i \leq m} F_i}{m}$. From Lemma 4, 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 $\left(\sum_{i=2}^{|A|/m} \frac{1}{i^\alpha}\right)^{\frac{1}{\alpha}}$ does not have a closed form, m is set to satisfy a stronger constraint, which is $m > r(\zeta(\alpha) - 1)^{\frac{1}{\alpha}}$.

Since $F_{r+1} = \frac{N}{r^\alpha \zeta(\alpha)} < min_{max} < \epsilon N$, 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 $r = \left(\frac{1}{\epsilon \zeta(\alpha)}\right)^{\frac{1}{\alpha}}$. Combining this result with the relation between m and r established above implies $m > \left(\frac{\zeta(\alpha) - 1}{\epsilon \zeta(\alpha)}\right)^{\frac{1}{\alpha}}$ will guarantee an error which is bound by ϵ. If $\alpha > 1$, the upper bound on ϵ will be enforced by satisfying $m = \left(\frac{1}{\epsilon}\right)^{\frac{1}{\alpha}}$. Otherwise, the bound of $m \geq \frac{1}{\epsilon}$ will apply from the general case discussed previously in Theorem 3. $\square$

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, that can not be directly inferred from the theoretical bounds.

4.1.3 Comparison with Similar Work

The bound of Theorem 3 is tighter than those guaranteed by the algorithms in [18, 35, 39]. *Sticky Sampling* [39] has a space bound of $\frac{2}{\epsilon} \ln(\frac{1}{\phi\delta})$, where δ is the failure probability. *Lossy Counting* [39] has a bound of $\frac{1}{\epsilon} \ln(\epsilon N)$. Both the *hCount* algorithm [35], and the *Multistage filters* [18] require a number of counters bounded by $\frac{e}{\epsilon} * \ln\left(\frac{-|A|}{\ln \delta}\right)$. Furthermore, *Space-Saving* has a tighter bound than *GroupTest* [14], whose bound is $O(\frac{1}{\phi} \ln(\frac{1}{\delta\phi}) \ln(|A|))$, which is less scalable than *Space-Saving*. 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 [17] 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 under-estimation error bound, as proved in [8]. However, there is no straightforward feasible extension of the algorithm to track the under-estimation 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 non-monitored element increases the errors for all the monitored elements, since their counters get decremented. Therefore, elements of higher frequency are more error prone, and thus, it is still difficult to guess the frequent elements, which is not the case for *Space-Saving*. Even more, the structure proposed in [17] is built and queried in a way that does not allow the user to specify an error threshold, ϵ. Thus, the algorithm has only one parameter, the support ϕ, which increases the number of false positives dramatically, as will be clear from the experimental results in Section 5.

The number of counters used in *GroupTest* [14] 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 $1 \ldots |A|$ can only be enforced by building an indexed lookup table that maps every ID to a unique number in the range $1 \ldots |A|$. 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 IDs were observed in the stream. This is in contrast to *Space-Saving*, which only requires the m 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}.

```
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;
```

Figure 5: Reporting Top-k

4.2 Top-k Elements

For the top-k elements, the algorithm can output the first k elements. An element, e_i, is **guaranteed to be among the top-k** if its guaranteed number of hits, $count_i - \varepsilon_i$, exceeds $count_{k+1}$, the over-estimated number of hits for the element in position $k+1$. Since, from Theorem 2, $count_{k+1}$ is an upper bound on F_{k+1}, the hits of the element of rank $k+1$, E_{k+1}, then e_i is in the top-k elements.

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 $1 \dots m-1$. 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 over-estimated 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 paper, we assume that the algorithm outputs only the first k elements, i.e., 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 $\Theta(|A|)$ [10, 17] 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, k, ϵ) [10], 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 6 *Regardless of the data distribution, to solve the FindApproxTop(S, k, ϵ) problem,* Space-Saving *uses* $\min(|A|, \frac{N}{\epsilon F_k})$ *counters. Any element with frequency more than* $(1-\epsilon)F_k$ *is guaranteed to be monitored.*

Proof. This is another form of Theorem 3, but $min = \epsilon F_k$, instead of ϵN. By the same token, we set $m = \frac{1}{\epsilon} * \frac{N}{F_k}$ 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 7 *Assuming the data is noiseless Zipfian with parameter* $\alpha > 1$*, to calculate the exact top-k,* Space-Saving *uses* $\min(|A|, O((\frac{k}{\alpha})^{\frac{1}{\alpha}} k))$ *counters. When* $\alpha = 1$*, the space complexity is* $\min(|A|, O(k^2 \ln(|A|)))$*. This is regardless of the stream permutation. Also, the order among the top-k elements is preserved.*

Proof. From Equation 1, Lemma 3, and Lemma 4, 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, it follows that, $F_{r+1} \leq \frac{N - \sum_{i \leq r} F_i}{m-r}$. Rewriting frequencies in their Zipfian form yields: $m - r \leq (r+1)^\alpha \sum_{i=r+1}^{|A|} \frac{1}{i^\alpha}$. This relation 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, $0 \leq \varepsilon_i \leq min_{max}$, 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 essential 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: $\left[(\zeta(\alpha)+1)\left(\frac{k}{\alpha}\right)^{\frac{1}{\alpha}}(k+1)\right] + \zeta(\alpha) < m$. 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 $\min(|A|, O(\left(\frac{k}{\alpha}\right)^{\frac{1}{\alpha}} k))$.

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 7, 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 1 *Assuming Zipfian data with parameter $\alpha > 1$, to calculate the exact frequent elements,* Space-Saving *uses only* $\min(|A|, O(\left(\frac{1}{\phi}\right)^{\frac{\alpha+1}{\alpha^2}}))$ *counters. When $\alpha = 1$, the space complexity is* $\min(|A|, O(\frac{1}{\phi^2 \ln(|A|)} + \ln(|A|)))$. *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 $i = \lfloor \frac{1}{\zeta(\alpha)\phi}^{\frac{1}{\alpha}} \rfloor$.

From Theorem 7, the number of counters needed to calculate the exact top-i elements is $\left[(\zeta(\alpha)+1)\left(\frac{i}{\alpha}\right)^{\frac{1}{\alpha}}(i+1)\right] + \zeta(\alpha)$. Substituting $i = \lfloor \frac{1}{\zeta(\alpha)\phi}^{\frac{1}{\alpha}} \rfloor$, yields $\left[(\zeta(\alpha)+1)\left(\frac{\lfloor \frac{1}{\zeta(\alpha)\phi}^{\frac{1}{\alpha}} \rfloor}{\alpha}\right)^{\frac{1}{\alpha}}(\lfloor \frac{1}{\zeta(\alpha)\phi}^{\frac{1}{\alpha}} \rfloor + 1)\right] + \zeta(\alpha)$.

If $\alpha = 1$, then $\zeta(\alpha) = \zeta(1) \approx \ln(1.78|A|)$, and the space complexity reduces to $\min(|A|, O(\frac{1}{\phi^2 \ln(|A|)} + \ln(|A|)))$.

If $\alpha > 1$, then $\zeta(\alpha)$ converges to a small constant inversely proportional to α, and the space complexity reduces to $\min(|A|, O(\left(\frac{1}{\phi}\right)^{\frac{\alpha+1}{\alpha^2}}))$. □

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* [10], for a large range of practical values of the parameters $|A|$, ϵ, and k. *CountSketch* solves the relaxed version of the problem, FindApproxTop(S, k, ϵ), with failure probability δ, using space of $O(\log(\frac{N}{\delta})(k + \frac{1}{(\epsilon F_k)^2} \sum_{i=k+1}^{|A|} F_i{}^2))$, with a large constant hidden in the big-O notation [10, 14]. The bound of *Space-Saving* for the relaxed problem is $\frac{N}{\epsilon F_k}$, 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 non-skewed 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 [10] can be rewritten as $O(\log(\frac{N}{\delta}) * (k + \frac{k^2}{\epsilon^2}\left(\frac{1}{k+1} - \frac{1}{|A|}\right))) \approx O(\frac{k}{\epsilon^2}\log(\frac{N}{\delta}))$. Even more, depending on the data distribution, *Space-Saving* can guarantee the reported top-k, or a subset of them, to be correct, with weak data skew; while *CountSketch* does not offer any guarantees.

In the case of Zipf Distribution, the bound of [10] is $O(k \log(\frac{N}{\delta}))$. For $\alpha > 1$, the bound of *Space-Saving* is $O(\left(\frac{k}{\alpha}\right)^{\frac{1}{\alpha}} k)$. Only when $\alpha = 1$, the space complexity is $O(k^2 \ln(|A|))$, 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 against the best algorithms known so far for both problems. 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 [14]. It is worth noting that an algorithm will have a recall, and a precision of 1 if it outputs all and exactly 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.

For the frequent elements problem, we compared *Space-Saving* to *GroupTest* [14], and *Frequent* [17]. For *GroupTest*, and *Frequent*, we used the C code available on the web-site of the first author of [14]. For the top-k problem, we implemented *Probabilistic-InPlace* [17], and *CountSketch* [10]. For *CountSketch* [10], we implemented the median algorithm by Hoare [34] with Median-of-three partition, which has a linear run time, in the average case [37]. Instead of maintaining a heap as suggested in [10], 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 space used by *CountSketch*. For the hidden constant of the space bounds given in [10], 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 2.66GHz PC, with 1.0GB RAM, and 80GB 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 uniform, 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 algorithms' performances 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 algorithms' results as we varied the queries' parameters.

5.1.1 Varying the Data Skew

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

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 1. When the Zipf parameter is 0.5, we assign the same number of counters as in the case when the Zipf parameter is 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 has 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. In this case, 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 Figure 6(b), and Figure 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 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

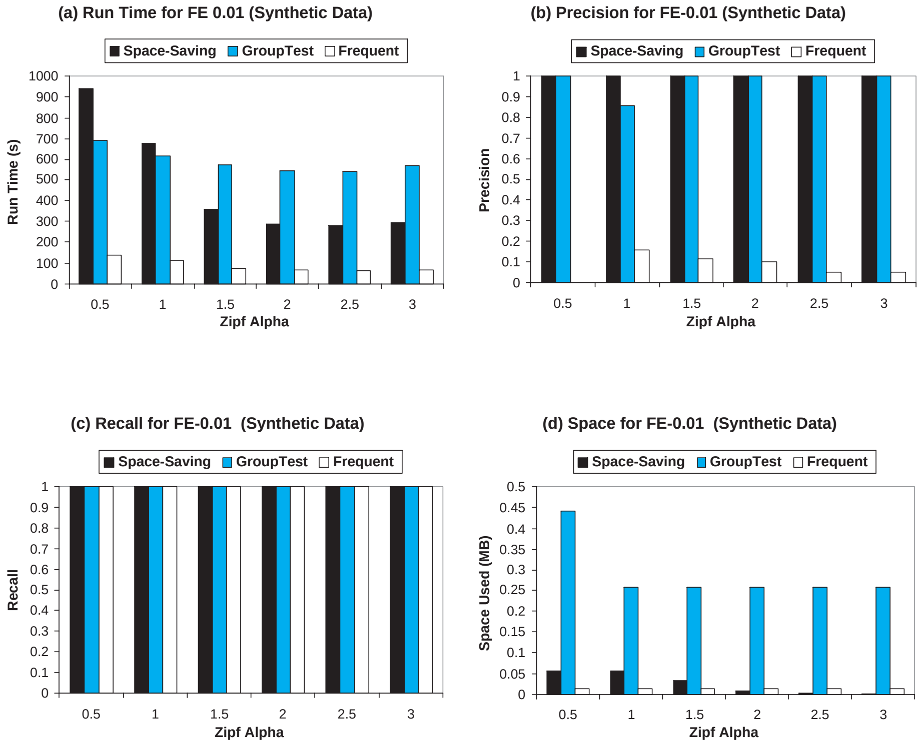


Figure 6: Performance Comparison for the Frequent Elements Problem Using Synthetic Zipfian Data - Varying Data Skew

was 3.0. This shows that *Space-Saving* adapts well to the data skew.

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

From Figure 7(b), the output of *Probabilistic-InPlace* was not comparable to the other two algorithms 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 is 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 keep, while the proved bound on the space used by *CountSketch* is fixed for $\alpha > \frac{1}{2}$ [10]. 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 not skewed, and 33 times smaller when the data was skewed. The run time of *CountSketch* decreased as α increased, since the number of times *CountSketch* has to estimate 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 share the same bucket. This reduced the number of operations 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

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

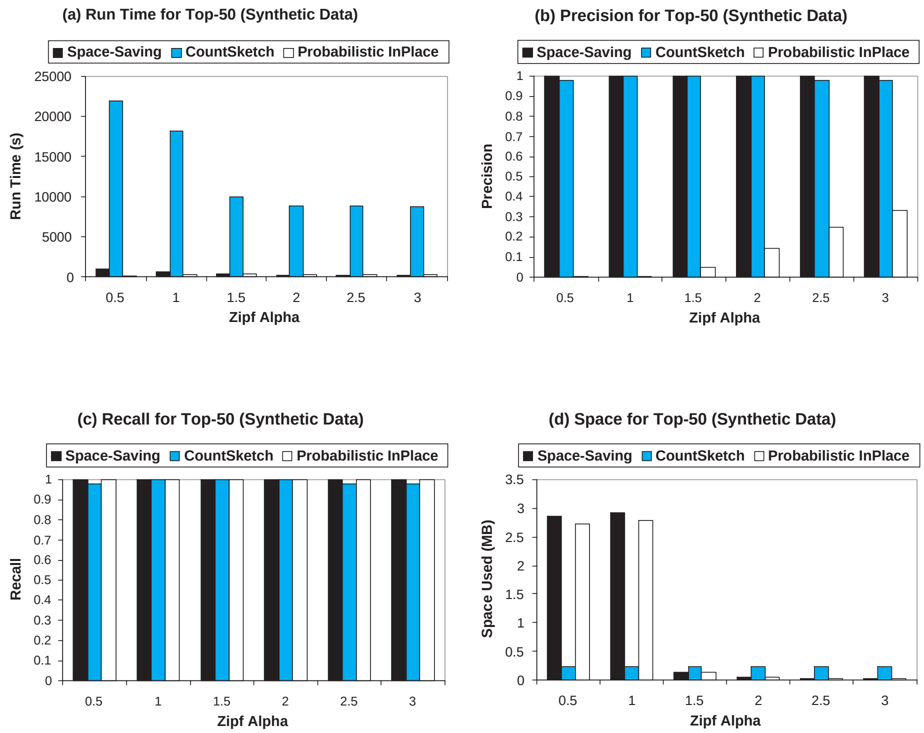


Figure 7: Performance Comparison for the Top-k Problem Using Synthetic Zipfian Data - Varying Data Skew

higher than 5 hits per ms, since when the data was almost uniform, *CountSketch* took 0.219 ms, on average, to process each observation in the stream.

5.1.2 Varying the Query Parameters

This set of experiments measure 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.

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 5 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 its 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*.

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 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 5 times smaller.

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

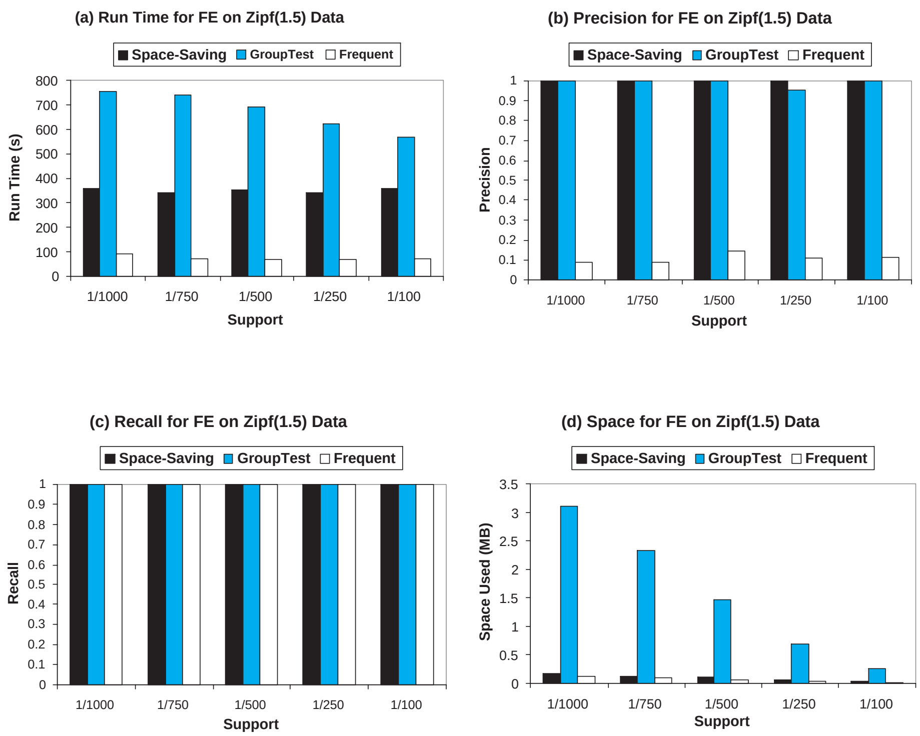


Figure 8: Performance Comparison for the Frequent Elements Problem Using Synthetic Zipf(1.5) Data - Varying the Support

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 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 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. Therefore, we made use of this set of experiments to provide a simulation for the behavior of *Space-Saving* when running in limited space. However, we did not fix the space available for all experiments at the same size, but made it a function of the query parameters, and examined how *Space-Saving* behaves under restricted conditions. 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 algorithms' behavior 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 measure how the algorithms perform under different realistic query parameters.

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 5 times more space than *Frequent*, as 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 $1 \ldots 4{,}235{,}870$ so as to be able to compare it with *Space-Saving*, though we did not account 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 5 times less than *GroupTest*, and ran faster most of the time, as shown in Figure 10(a).

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 $100 * k$. The results are shown in Figure 11.

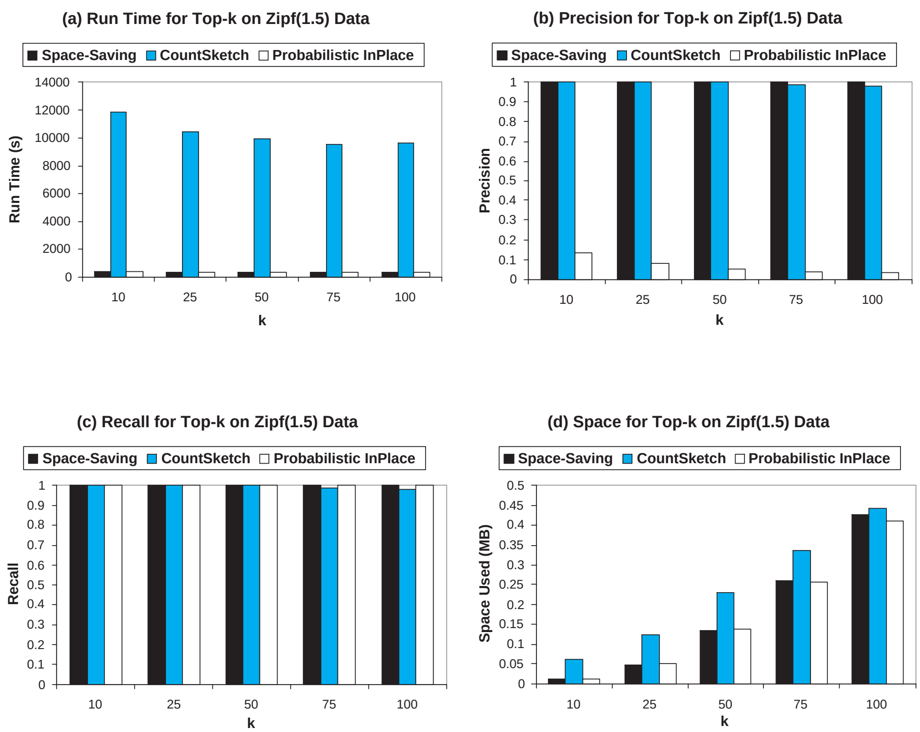


Figure 9: Performance Comparison for the Top-k Problem Using Synthetic Zipf(1.5) Data - Varying the k Parameter

Although *Probabilistic-InPlace* had good recall, as shown in Figures 11(b), its precision, as clear from Figure 11(c), was not comparable to the two other algorithms, since its highest precision was 0.020. The run time of *Probabilistic-InPlace* was 4 to 5 times less tha that of *Space-Saving*, and their space usages were very comparable.

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 8 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.

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. i.e., 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 1, and 2.

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 1. That is, *Space-Saving outputs only the correct elements, nothing but the correct elements, and guarantees its output to be correct.*

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 2. In other words, *Space-Saving* was

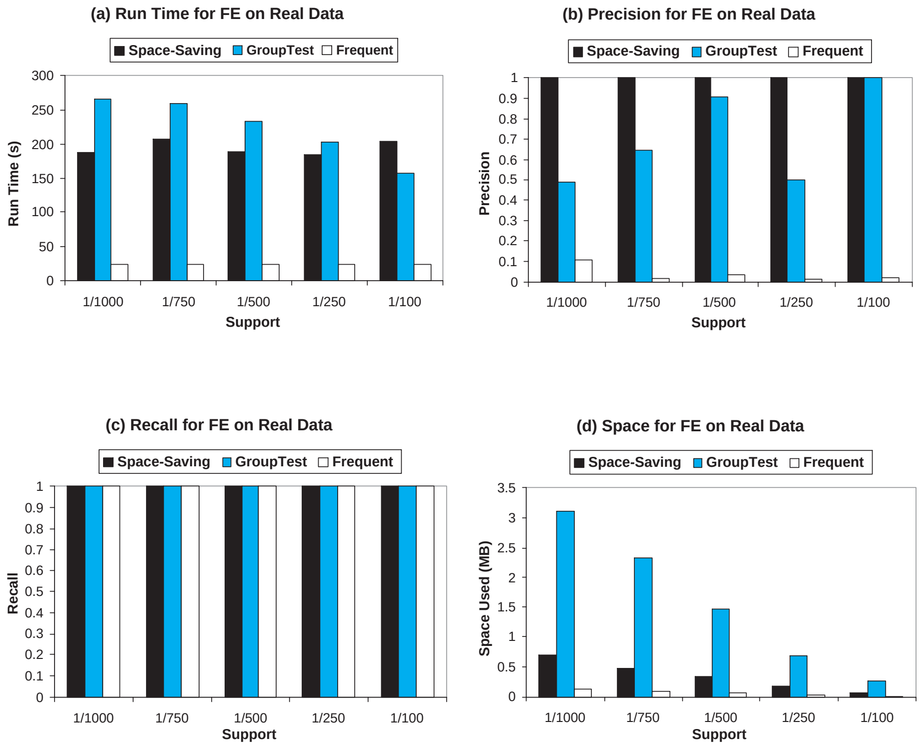


Figure 10: Performance Comparison for the Frequent Elements Problem Using a Real Click Stream

Support	Number of Frequent Elements	Size of Output	Number of Guaranteed Frequent Elements	Accuracy	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 1: *Space-Saving* Guarantee for the Frequent Elements Problem Using a Real Click Stream

able to guarantee 80% to 100% of its output to be correct. Throughout the experimental runs, the number of non-guaranteed elements was at most 5.

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, i.e., 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 clean up 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

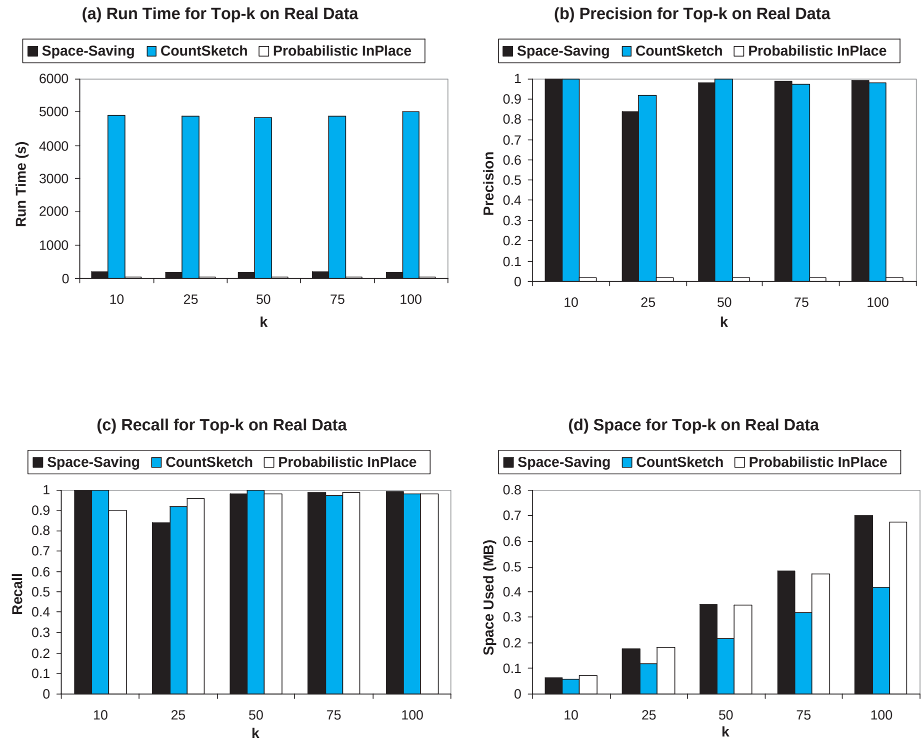


Figure 11: Performance Comparison for the Top-k Problem Using a Real Click Stream

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

Table 2: *Space-Saving* Guarantee for the Top-k Problem Using a Real Click Stream

Algorithm: ContinuousQueryFrequent(counter $count_i$)
begin
  //Incrementing the stream size
  N ++;
  //Reporting elements that are becoming infrequent
  let $Bucket_\phi$ be the bucket ptr_ϕ points to
  let $Bucket_\phi^+$ be $Bucket_\phi$'s neighbor of larger value
  if ($Bucket_\phi < \lceil \phi N \rceil$){
    Report $Bucket_\phi$'s child-list as infrequent;
    Move ptr_ϕ to $Bucket_\phi^+$;
  }
  //Reporting e_i if it becomes frequent
  let $Bucket_i'$ be the new bucket of $count_i$
  let $Bucket_\phi'$ be the new bucket ptr_ϕ points to
  if ($Bucket_i' > \lceil \phi N \rceil$ **AND** $Bucket_i' \leq Bucket_\phi'$){
    let e_i be the element of $count_i$
    Report e_i as frequent;
    Move ptr_ϕ to $Bucket_i'$;
  }
end;

Figure 12: Incremental Reporting of Frequent Elements

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 *Continuous-QueryFrequent*. *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 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 $O(|\text{elements becoming infrequent}|)$. Thus, *ContinuousQueryFrequent* takes $O(|\text{updated elements}|)$ 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 k^{th} 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 5 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, 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 received 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

```
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;
```

Figure 13: Incremental Reporting of Top-k

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 that 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 k^{th} 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 k^{th} 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, i.e., 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, i.e., 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.

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 paper has devised an integrated approach for solving an interesting family of problems in data streams. The *Stream-Summary* data structure was proposed, and utilized by the *Space-Saving* algorithm to guarantee strict 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 paper 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 benefit of the proposed algorithm.

This is the first algorithm, to the best of our knowledge, that 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, our space bounds are still tighter than those of other algorithms.

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. The underlying reason is that it can be easily shown that the expected value of the over-estimation, ε_i, increases monotonically with the sum of the length of the stream sections when e_i was not monitored, which is inversely related to f_i.

References

[1] N. Alon, Y. Matias, and M. Szegedy. The Space Complexity of Approximating the Frequency Moments. In *Proceedings of the 28th ACM STOC Symposium on the Theory of Computing*, pages 20–29, 1996.

[2] A. Arasu, S. Babu, and J. Widom. CQL: A Language for Continuous Queries over Streams and Relations. In *Proceedings of the 9th DBPL International Conference on Data Base and Programming Languages*, pages 1–11, 2003.

[3] A. Arasu, S. Babu, and J. Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. Technical Report 2002-67, Stanford University, 2003.

[4] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and Issues in Data Stream Systems. In *Proceedings of the 21st ACM PODS Symposium on Principles of Database Systems*, pages 1–16, 2002.

[5] B. Babcock and C. Olston. Distributed Top-k Monitoring. In *Proceedings of the 22nd ACM SIGMOD International Conference on Management of Data*, pages 28–39, 2003.

[6] Burton H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, 1970.

[7] P. Bonnet, J. Gehrke, and P. Seshadri. Towards Sensor Database Systems. In *Proceedings of the 2nd IEEE MDM International Conference on Mobile Data Management*, pages 3–14, 2001.

[8] P. Bose, E. Kranakis, P. Morin, and Y. Tang. Bounds for Frequency Estimation of Packet Streams. In *Proceedings of the 10th SIROCCO International Colloquium on Structural Information and Communication Complexity*, pages 33–42, 2003.

[9] R. Boyer and J. Moore. A Fast Majority Vote Algorithm. Technical Report 1981-32, Institute for Computing Science, University of Texas, Austin, 1981.

[10] M. Charikar, K. Chen, and M. Farach-Colton. Find-

ing Frequent Items in Data Streams. In *Proceedings of the 29th ICALP International Colloquium on Automata, Languages and Programming*, pages 693–703, 2002.

[11] J. Chen, D. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proceedings of the 19th ACM SIGMOD International Conference on Management of Data*, pages 379–390, 2000.

[12] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava. Finding Hierarchical Heavy Hitters in Data Streams. In *Proceedings of the 29th ACM VLDB International Conference on Very Large Data Bases*, pages 464–475, 2003.

[13] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava. 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*, pages 155–166, 2004.

[14] G. Cormode and S. Muthukrishnan. 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*, pages 296–306, 2003.

[15] C. Cortes, K. Fisher, D. Pregibon, A. Rogers, and F. Smith. Hancock: A Language for Extracting Signatures from Data Streams. In *Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 9–17, 2000.

[16] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining Stream Statistics over Sliding Windows. In *Proceedings of the 13th ACM SIAM Symposium on Discrete Algorithms*, pages 635–644, 2002.

[17] E. Demaine, A. López-Ortiz, and J. Munro. Frequency Estimation of Internet Packet Streams with Limited Space. In *Proceedings of the 10th ESA Annual European Symposium on Algorithms*, pages 348–360, 2002.

[18] C. Estan and G. Varghese. New Directions in Traffic Measurement and Accounting: Focusing on the Elephants, Ignoring the Mice. *ACM Transactions on Computer Systems*, 21(3):270–313, 2003.

[19] M. Fang, S. Shivakumar, H. Garcia-Molina, R. Motwani, and J. Ullman. Computing Iceberg Queries Efficiently. In *Proceedings of the 24th ACM VLDB International Conference on Very Large Data Bases*, pages 299–310, 1998.

[20] J. Feigenbaum, S. Kannan, M. Strauss, and M. Viswanathan. An Approximate L1-Difference Algorithm for Massive Data Streams. In *Proceedings of 40th FOCS Annual Symposium on Foundations of Computer Science*, pages 501–511, 1999.

[21] M. Fischer and S. Salzberg. Finding a Majority Among N Votes: Solution to Problem 81-5. *Journal of Algorithms*, 3:376–379, 1982.

[22] P. Flajolet and G. Martin. Probabilistic Counting Algorithms. *Journal of Computer and System Sciences*, 31:182–209, 1985.

[23] J. Gehrke, F. Korn, and D. Srivastava. On Computing Correlated Aggregates Over Continual Data Streams. In *Proceedings of the 20th ACM SIGMOD International Conference on Management of Data*, pages 13–24, 2001.

[24] P. Gibbons and Y. Matias. New Sampling-Based Summary Statistics for Improving Approximate Query Answers. In *Proceedings of the 17th ACM SIGMOD International Conference on Management of Data*, pages 331–342, 1998.

[25] A. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Surfing Wavelets on Streams: One-Pass Summaries for Approximate Aggregate Queries. In *Proceedings of the 27th ACM VLDB International Conference on Very Large Data Bases*, pages 79–88, 2001.

[26] L. Golab, D. DeHaan, E. Demaine, A. López-Ortiz, and J. Munro. Identifying Frequent Items in Sliding Windows over OnLine Packet Streams. In *Proceedings of the 1st ACM SIGCOMM Internet Measurement Conference*, pages 173–178, 2003.

[27] L. Golab and M. Ozsu. Issues in Data Stream Management. *ACM SIGMOD Record*, 32(2):5–14, 2003.

[28] M. Greenwald and S. Khanna. Space-Efficient Online Computation of Quantile Summaries. In *Proceedings of the 19th ACM SIGMOD International Conference on Management of Data*, pages 58–66, 2001.

[29] S. Guha, P. Indyk, M. Muthukrishnan, and M. Strauss. Histogramming Data Streams with Fast Per-Item Processing. In *Proceedings of the 29th ICALP International Colloquium on Automata, Languages and Programming*, pages 681–692, 2002.

[30] S. Guha, N. Koudas, and K. Shim. Data-Streams and Histograms. In *Proceedings of the 33rd ACM STOC Symposium on the Theory of Computing*, pages 471–475, 2001.

[31] S. Gunduz and M. Ozsu. 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*, pages 535–540, 2003.

[32] P. Gupta and N. McKeown. Packet Classification on Multiple Fields. In *Proceedings of the ACM SIGCOMM International Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 147–160, 1999.

[33] P. Haas, J. Naughton, S. Sehadri, and L. Stokes. Sampling-Based Estimation of the Number of Distinct Values of an Attribute. In *Proceedings of the 21st ACM VLDB International Conference on Very Large Data Bases*, pages 311–322, 1995.

[34] C. Hoare. Algorithm 65: Find. *Communications of the ACM*, 4(7):321–322, 1961.

[35] C. Jin, W. Qian, C. Sha, J. Yu, and A. Zhou. Dynamically Maintaining Frequent Items over a Data Stream. In *Proceedings of the 12th ACM CIKM International Conference on Information and Knowledge Management*, pages 287–294, 2003.

[36] R. Karp, S. Shenker, and C. Papadimitriou. A Simple Algorithm for Finding Frequent Elements in Streams and Bags. *ACM Transactions on Database Systems*, 28(1):51–55, 2003.

[37] P. Kirschenhofer, H. Prodinger, and C. Martinez. Analysis of Hoare's FIND Algorithm With Median-

Of-Three Partition. *Random Structures Algorithms*, 10(1-2):143–156, 1997.

[38] X. Lin, H. Lu, J. Xu, and J. Yu. 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*, pages 362–374, 2004.

[39] G. Manku and R. Motwani. Approximate Frequency Counts over Data Streams. In *Proceedings of the 28th ACM VLDB International Conference on Very Large Data Bases*, pages 346–357, 2002.

[40] G. Manku, S. Rajagopalan, and B. Lindsay. 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*, pages 251–262, 1999.

[41] Y. Matias, J. Vitter, and M. Wang. Dynamic Maintenance of Wavelet-Based Histograms. In *Proceedings of the 26th ACM VLDB International Conference on Very Large Data Bases*, pages 101–110, 2000.

[42] A. Metwally, D. Agrawal, and A. El Abbadi. Duplicate Detection in Click Streams. In *Proceedings of the 14th WWW International World Wide Web Conference*, pages 12–21, 2005. An extended version appears as a University of California, Santa Barbara, Department of Computer Sciemce, Technical Report 2004-23.

[43] A. Metwally, D. Agrawal, and A. El Abbadi. Efficient Computation of Frequent and Top-k Elements in Data Streams. In *Proceedings of the 10th ICDT International Conference on Database Theory*, pages 398–412, 2005. An extended version appears as a University of California, Santa Barbara, Department of Computer Sciemce, Technical Report 2005-23.

[44] J. Misra and D. Gries. Finding Repeated Elements. *Science of Computer Programming*, 2:143–152, 1982.

[45] K. Whang, B. Vander-Zanden, and H. Taylor. A Linear-Time Probabilistic Counting Algorithm for Database Applications. *ACM Transactions on Database Systems*, 15:208–229, 1990.

[46] Y. Zhu and D. Shasha. StatStream: Statistical Monitoring of Thousands of Data Streams in Real Time. In *Proceedings of the 28th ACM VLDB International Conference on Very Large Data Bases*, pages 358–369, 2002.

[47] G. Zipf. *Human Behavior and The Principle of Least Effort.* Addison-Wesley, 1949.