Information Sciences 180 (2010) 4958–4974

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

Finding top-*k* elements in data streams

Nuno Homem *, Joao Paulo Carvalho

TULisbon – Instituto Superior Técnico, INESC-ID, R. Alves Redol 9, 1000-029 Lisboa, Portugal

ARTICLE INFO

Article history:
Received 10 November 2009
Received in revised form 11 August 2010
Accepted 15 August 2010

Keywords:
Approximate algorithms
Top-*k* algorithms
Most frequent
Estimation
Data stream frequencies

ABSTRACT

Identifying the most frequent elements in a data stream is a well known and difficult problem. Identifying the most frequent elements for each individual, especially in very large populations, is even harder. The use of fast and small memory footprint algorithms is paramount when the number of individuals is very large. In many situations such analysis needs to be performed and kept up to date in near real time. Fortunately, approximate answers are usually adequate when dealing with this problem. This paper presents a new and innovative algorithm that addresses this problem by merging the commonly used counter-based and sketch-based techniques for top-*k* identification. The algorithm provides the top-*k* list of elements, their frequency and an error estimate for each frequency value. It also provides strong guarantees on the error estimate, order of elements and inclusion of elements in the list depending on their real frequency. Additionally the algorithm provides stochastic bounds on the error and expected error estimates. Telecommunications customer's behavior and voice call data is used to present concrete results obtained with this algorithm and to illustrate improvements over previously existing algorithms.

1. Introduction

A data stream is a sequence of events or records that can be accessed in order of arrival, sometimes randomly, within a sub-set of last arrivals or a period of time. The data stream model applies to many fields where huge data sets, possibly infinite, are generated and storage of the entire set is not viable. Applications such as IP session logging, telecommunication records, financial transactions or real time sensor data generate so much information that the analysis has to be performed on the arriving data, in near real time or within a time window. In most cases, it is impossible to store all event details for future analysis due to the huge volume of data; usually only a set of summaries or aggregates is kept. Relevant information is extracted, transient or less significant data is dropped. In many cases those summaries have to be generated in a single pass over the arriving data. Collecting and maintaining summaries over data streams require fast and efficient methods. However approximate answers may, in many situations, be sufficient.

Identifying the most frequent elements in a data stream is a critical but challenging issue in several areas. The challenge is even bigger if this analysis is required at individual level, i.e., when a specific frequent elements list per individual is needed. This is essential when studying behavior and identifying trends. Some examples:

- Internet providers need to know the most frequent destinations in order to manage traffic and service quality;
- Finding the most frequent interactions is critical to any social networking analysis as it provides the basic information of connections and relations between individuals;

* Corresponding author.
E-mail addresses: nuno.homem@hotmail.com (N. Homem), joao.carvalho@inesc-id.pt (J.P. Carvalho).

0020-0255/$ - see front matter © 2010 Elsevier Inc. All rights reserved.
doi:10.1016/j.ins.2010.08.024

- Telecommunication operators need to identify the most common call destinations for each customer for a variety of reasons, like classifying the customer in terms of marketing segmentation, proposing better and attractive customer deals, giving specific discounts for more frequently called numbers, or profiling the customer and detecting fraud situations or fraudsters that have returned under new identities;
- Retail companies need to know the most common buys for each customer in order to better classify the customer and target him with the appropriate marketing campaigns, to propose better and more attractive customer deals, to give specific discounts for specific products, or to identify changes in spending profile or consumer habits.

Knowing the most frequent elements for each individual, or from their transactions, in a large population, is a quite distinct problem from finding the most frequent elements in a large set. For each individual the number of distinct elements may not be huge, but the number of individuals can be. Instead of a single large set of frequent elements, a large number of smaller sets of frequent elements are required.

In many situations this information has to be kept up to date and available at any time. Accurately identifying it in near real time without allocating a huge amount of resources to the task is the problem. A data stream with multiple transactions per day from several million individuals represents a huge challenge. Optimizing the process is therefore critical.

Classical exact top-k algorithms require checking the previous elements included in the list every time a new transaction is processed, either to insert new elements or to update the existing element counters. Exact top-k algorithms require large amounts of memory as they need to store the complete list of elements. Storing a 1000 list of elements per each individual if only the top-20 is required is a complete waste of resources. Approximate, small footprint algorithms are the solution when huge numbers of individual lists are required. For example, Telecommunication operators can range from less than 500,000 customers (a small operator) to more than 25,000,000 customers, and a list of the top-20 services or destinations might be needed for each one; Retail sellers have similar or an even larger number of customers, in some cases identified by the use of fidelity cards. The number of elements to be stored per customer is usually small as people tend to make frequent calls to a relatively small number of destinations and to frequently buy the same products.

This paper proposes a new algorithm for identifying the approximate top-k elements and their frequencies while providing an error estimate for each frequency. The algorithm gives strong guarantees on the error estimate, on the order of elements and on the inclusion of elements in the list depending on their real frequency. Stochastic error bounds that further improve its performance are also presented. The algorithm was designed for reduced memory consumption as it is intended to address the problem of solving a huge number of distinct frequent elements queries in parallel. It provides a fast update process for each element, avoiding full scans or batch processing. Since the algorithm provides an error estimate for each frequency, it also provides the possibility to control the quality of the estimate.

Although the focus of the algorithm is to produce a large number of good top-k lists, each based on the individual set of transactions, it can also be used to efficiently produce a single top-k list for a huge number of transactions.

The presented algorithm is innovative since it merges the two very distinct approaches commonly used to solve this sort of problems: counter-based techniques, and sketch-based techniques. It follows the principles presented by Metwally et al. [14] for Space-Saving algorithm but improves it quite significantly by narrowing down both the number of required counters, update operations and the error associated with the frequency estimate. This is achieved by filtering elements using a specially designed sketch. The merge of these two distinct approaches improves the existing properties of Space-Saving algorithm, providing lower stochastic bounds for estimate error and for the expected error.

In this work one also discusses how distinct implementation options can be used to minimize memory footprint and to better fit the algorithm and the corresponding data structures to the specific problem.

2. Typical behavior of mobile phone users

Although the algorithm is generic, it was originally developed to find top-k call destination numbers for each customer in telecommunications companies. Given the required precision in this situation, the use of approximate algorithms was considered more than adequate. The typical behavior of individual customers is presented to illustrate the relevance of this analysis. A set of real voice calls from mobile phone customers was used as a sample. As mobile phone users make more calls per day than landline phone users, this makes them ideal for use as a reference in this analysis.

The analysis is based on 962 blocks of calls from distinct customers, each with 500 mobile voice calls (made during a 2 or 3 months period in most cases). Note that since most customers make no more than 15 calls a day, 500 calls is a significant number of calls. On average, in 500 calls each customer would call 98.67 distinct numbers (with a standard deviation of 34.99). Fig. 1 presents the distinct destinations histogram. The highest value in this sample was 262. A 500 call block having 262 distinct destinations is not common in individual customers. In fact, this particular case is a SIM Box, an equipment used to deliver calls from an internal phone network to the mobile operator network, concentrating the traffic from several users.

Figs. 2 and 3 present the call frequency to distinct numbers (with distinct number ranked per received calls) for each block.

It can be seen that the 11th most called number receives in average less than 2% of the calls, and the 21st most called number receives less than 1% of the calls. The top-7 destinations receive 50% of the calls and the top-30 around 80%. Given these numbers, most telecommunication operators focus on the top-5 for offering discounts. For marketing segmentation purposes the top-10 are used and for fraudster identification purposes the top-20 might be relevant.

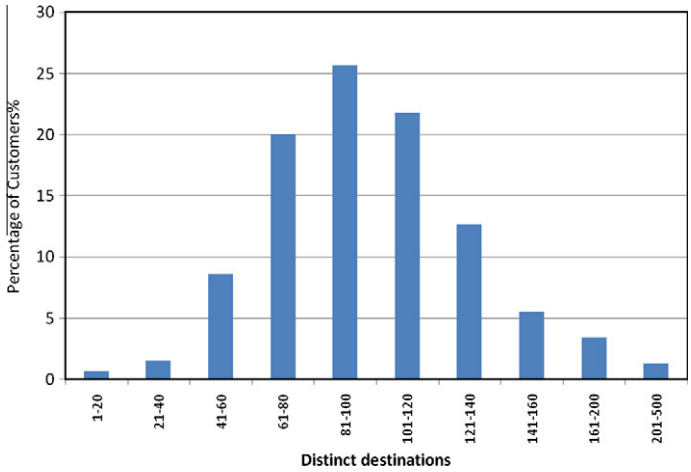

Fig. 1. Distinct destinations per customer call block histogram.

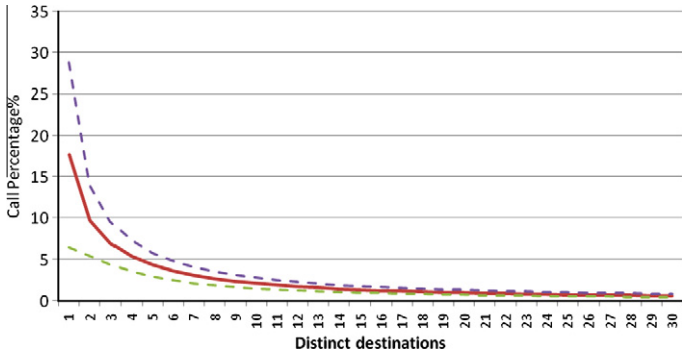

Fig. 2. Call% per distinct number.

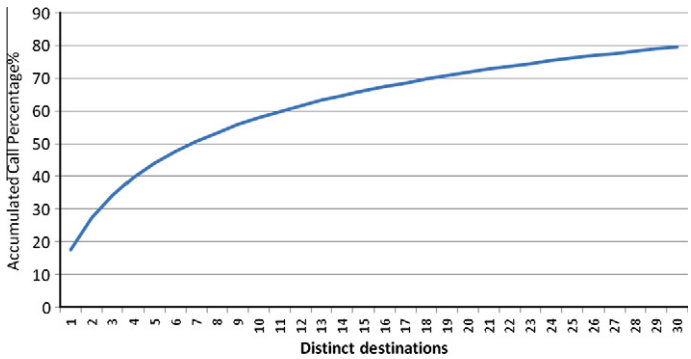

Fig. 3. Accumulated call% per distinct number.

The distribution of call% per distinct number (at least for the first 200 numbers) can be approximated with a Zipfian distribution (a distribution where the number of distinct elements are proportional to a function like $1/n\alpha$ with α a real value greater than 0) as $f(n) \approx k/n^{\alpha}$, where k is a constant, n is the rank in the ordered list of distinct destinations, and $\alpha \approx 1$.

3. Relation with previous work

Exact top-k algorithms require the complete list of distinct elements to be stored, resulting in the usage of large amounts of memory. Each new element must be checked against that list. If the element is already there, the counters are updated;

otherwise the new element is inserted. To allow a fast search of the elements, hash tables or binary trees may be used, resulting in an additional overhead in memory consumption.

Approximate algorithms can roughly be divided into two classes: Counter based techniques and Sketch-based techniques.

3.1. Counter based techniques

The most successful counter-based technique previously known, the Space-Saving algorithm, was proposed by Metwally et al. in [14]. Space-Saving underlying idea is to monitor only a pre-defined number of m elements and their associated counters. Counters on each element are updated to reflect the maximum possible number of times an element has been observed and the error that might be involved in that estimate. If an element that is already being monitored occurs again, the counter for the frequency estimate is incremented. If the element is not currently monitored it is always added to the list. If the maximum number of elements has been reached, the element with the lower estimate of possible occurences is dropped. The new element estimate error is set to the estimate of frequency of the dropped element. The new element frequency estimate equal to the error plus 1.

The Space-Saving algorithm will keep in the list all the elements that may have occurred at least the new estimate error value (or the last dropped element estimate) of times. This ensures that no false negatives are generated but allows for false positives. Elements with low frequencies that are observed in the end of the data stream have higher probabilities of being present in the list.

Since the Space-Saving algorithm maintains both the frequency estimate and the maximum error of this estimate, it also maintains implicitly the number of observed elements while the element was being monitored. Under some constraints, the order of the top-k elements is guaranteed to be the correct one. By allowing m, the number of elements in the monitored list, to be larger than k, the algorithm provides good results in most cases. The need to allow for m to be much larger than k motivated the algorithm proposed in this paper.

Metwally et al. provide in [14] a comparison between several algorithms used to address this problem. It is shown that the Space-Saving algorithm is already less demanding on counters than other comparable algorithms, like Lossy Counting [13] or Probabilistic Lossy Counting [5]. These algorithms process the data stream in batches and require counters to be zeroed at the end of each batch. This requires frequent scans of the counters and result in freeing space before it is really needed. Probabilistic Lossy Counting improves the rate of replacement of elements by using a probabilistic bound error instead of a deterministic value.

The Frequent algorithm, proposed in [4] and based on [15], outputs a list of k elements with no guarantee on which elements, if any, have frequency higher than $N/(k+1)$, where N is the total number of elements. The Frequent algorithm uses k counters to monitor k elements. If an element that is already being monitored occurs again, the counter for the frequency estimate is incremented; else all counters are decremented. If a counter is zeroed, it is made free for the observed element. All free counters can be filled with the next observed values. The results are poor as no estimate is made on the frequency of each element.

The sampling algorithm Probabilistic-InPlace proposed in [4], in line with Sticky Sampling [13], also addresses the search for the top-k elements, but is unable to provide estimates of the frequency or any guarantees on order. It also requires more space and processes elements in batches with counter deletions in the end of each batch.

A distinct approach, FDPM, is presented in [19]. FPDM focus in controlling the number of false negatives elements instead of false positives. This minimizes the probability of missing a frequent element to be missed while keeping memory usage under control. The main inconvenient is that more false positives may be included in the list.

3.2. Sketch-based techniques

Sketch-based algorithms use bitmap counters in order to provide a better estimate of frequencies for all elements. Each element is hashed into one or more values, those values are used to index the counters to update. Since the hash function can generate collisions, there is always a possible error. Keeping a large set of counters to minimize collision probability leads to an higher memory footprint when comparing with Space-Saving algorithm. Additionally, the entire bitmap counter needs to be scanned and elements sorted to answer to the top-k query.

GroupTest proposed in [2], is a probabilistic approach based on FindMajority [2]. However it does not provide information about frequencies or relative order. Multistage filters were proposed as an alternative in [7] but present the same problems. Sketch-based algorithms are however very useful when it is necessary to process huge number of elements and extract a single top-k list. The memory consumption of these algorithms usually depends on the number of distinct elements of the stream to ensure controlled collision rates. An experimental evaluation of some of the described algorithms is presented in [12].

3.3. Related data mining techniques

In the data mining area there has been a huge amount of work focused on obtaining frequent itemsets in high volume and high-speed transactional data streams. The use of FP-trees (Frequent Pattern trees) and variations of the FP-growth algorithms to extract frequent itemsets is proposed in [16,11]. Frequent itemsets extraction in a sliding window is proposed

in [17]. Although the problem of extracting itemsets may be slightly different, the same algorithms may be used to extract frequent elements.

4. The filtered Space-Saving algorithm (FSS)

In this work one proposes a novel algorithm, Filtered Space-Saving (FSS), that uses a filtering approach to improve on Space-Saving [14]. It also gets some inspiration on previous work around probabilistic counting and sliding windows statistics [3,6,8–10,18]. FSS uses a bitmap counter to filter and minimize updates on the monitored elements list and also to better estimate the error associated with each element. Instead of using a single error estimate value, it uses an error estimate dependent on the hash counter. This allows for better estimates by using the maximum possible error for that particular hash value instead of a global value. Although an additional bitmap counter has to be kept, it reduces the number of extra elements in the list needed to ensure high quality top-k elements. It will also reduce the number of list updates. The conjunction of the bitmap counter with the list of elements, minimizes the collision problem of most sketch-based algorithms. The bitmap counter size can depend on the number of k elements to be retrieved and not on the number of distinct elements of the stream, which is usually much higher.

FSS uses a bitmap counter with h cells, each containing two values, α_i and c_i, standing for the error and the number of monitored elements in cell i. The hash function needs to be able to transform the input values from stream S into a uniformly distributed integer range. The hashed value $h(x)$ is then used to access the corresponding counter. Initially all values of α_i and c_i are set to 0. The second storage element is a list of monitored elements with size m. The list is initially empty. Each element consists of 3 parts: the element itself v_j, the estimate count f_j and the associated error e_j. Fig. 4 presents a block diagram of the FSS algorithm.

The value of α_i gives an indication on "how many of the hits occurring in that cell were not considered for the monitored list". In order for an element to be included in the monitored list, it has to have at least as many possible hits α_i as the minimum of the estimates in the monitored list, $\mu = min\{f_j\}$. While the list has free elements, α_i and μ are set to 0.

The algorithm is quite simple. When a new value is received, the hash is calculated and the bitmap counter is checked; If there are already monitored elements with that same hash ($c_i > 0$), the monitored list is searched to see if this particular element is already there; If the element is in the list then the estimate count f_j is incremented.

A new element will only be inserted into the list if $\alpha_i + 1 \geqslant \mu$. If the element is not to be included in the monitored list then α_i is incremented. In fact α_i stands for the maximum number of times an element that has $hash(x) = i$ and that is not in the monitored list value could have been observed.

If the monitored list has already reached its maximum allowed size and a new element has to be included, the element with the lower f_j is selected for deletion. If there are several elements with the same value, then one of those with the larger value of e_j is selected. If the list is kept ordered by decreasing f_j and increasing e_j then the last element is always removed.

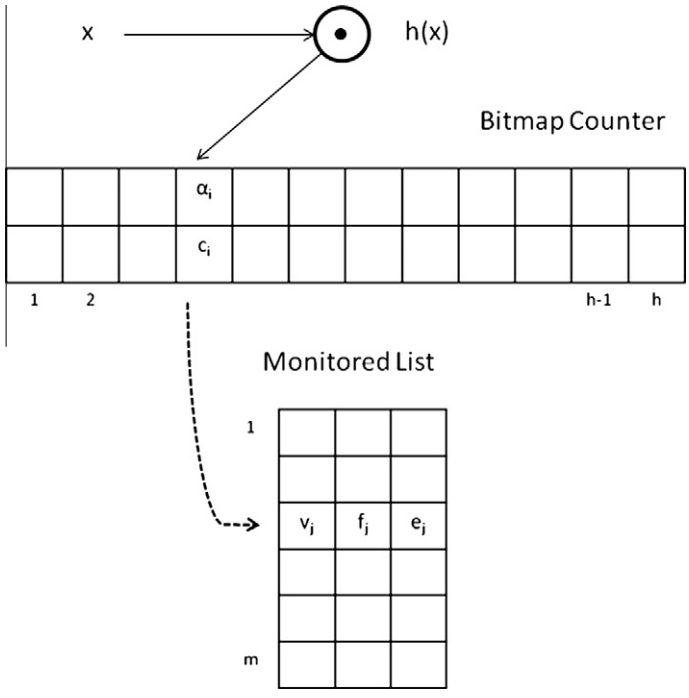

Fig. 4. FSS algorithm diagram.

When the selected element is removed from the list, the corresponding bitmap counter cell is updated, c_j is decreased and α_i is set with the maximum error incurred for that position in a single element, which corresponds to the estimate for the removed element, $\alpha_i = f_j$.

The new element is included in the monitored list ordered by decreasing f_j and increasing e_j, c_i is incremented, $f_j = \alpha_i + 1$ and $e_j = \alpha_i$.

It is easy to see that the Space-Saving algorithm is a particular case of FSS when $h = 1$.

Obvious optimizations to this algorithm, such as the use of better structures to hold the list of elements, to keep up to date μ, or to speed up access to each element will not be covered at this stage. Fig. 5 presents the proposed FSS algorithm.

5. Properties of filtered Space-Saving

FSS builds on the properties of Space-Saving algorithm and adds some very interesting stochastic guarantees for lower bounds of error μ and for the expected value of μ.

Theorems 1–3 present strong guarantees inherited from Space-Saving algorithm. Theorems 4–7 present bounds on μ and the expected value of μ, $E(\mu)$, that allow error to be stochastically limited depending on the total number of elements N and the number of cells h in the bitmap counter.

Lemma 1. *The value of μ and of each α_i increases monotonically over time. For every t, t_0 after the algorithm has been started:*

$$t \geqslant t_0 \Rightarrow \alpha_i(t) \geqslant \alpha_i(t_0) \tag{1}$$

$$t \geqslant t_0 \Rightarrow \mu(t) \geqslant \mu(t_0) \tag{2}$$

Proof. As α_i and μ are integers:

$$\alpha_i + 1 > \mu \Longleftrightarrow \alpha_i \geqslant \mu$$

By construction whenever $\alpha_i \geqslant \mu$ there is a replacement in the monitored list and there will be a new element inserted where $f_j = \alpha_i + 1$. At that instant μ will be updated to $min\{f_j\}$, so:

$$\alpha_i(t) \leqslant max\{\alpha_i(t)\} \leqslant \mu(t) = min\{f_j(t)\} \leqslant f_j(t) \tag{3}$$

```
Algorithm: Filtered Space-Saving(h cells, m counters, S stream)
begin
        for each element, x, in S {
                set μ to min {f_j}
                let i be the hash(x) mod h
                if c_i is not 0 {
                        if x is monitored {
                                let j be the index of x in the list
                                increment counter f_j
                                continue for next x
                        }
                } // the following will only be executed if x is not monitored

                if  α_i +1 >= μ {
                        if list size equals m {
                                let n be the index of one element with lower f_j
                                        and for same f_j with higher e_j
                                let k be the hash(x) mod h
                                decrement c_k
                                set α_k = f_i
                                remove v_n
                        }
                        include x in the list in index j
                        set v_j to x
                        set e_j to α_i and f_j to α_i+1
                        increment counter c_i

                } else {
                        increment counter α_i
                }
        }// end for
end
```

Fig. 5. FSS algorithm.

Consider t_1 as the moment before the replacement and $t_2$the moment after the replacement, $t_2 > t_1$. If the replaced element in the list was the only element to have $f_j(t_1) = \mu(t_1)$ then:

$$\mu(t_2) = \alpha_i(t_1)_{+1} > \mu(t_1)$$

If more than one element in the list had $f_j(t_1) = \mu(t_1)$ then

$$\mu(t_2) = \mu(t_1)$$

Therefore in every situation:

$$\mu(t_2) \geqslant \mu(t_1)$$

In the replacement process, the cell in the bitmap corresponding to the hash of the replaced element will be updated with the value of that element:

$$\alpha_i(t_2) = f_k(t_1) = \mu(t_1) \geqslant max\{\alpha_i(t_1)\} \geqslant \alpha_i(t_1)$$

Whenever a new hit does not require a replacement in the list, one counter among the m counters for f_j plus the h counters for α_i will be incremented. Therefore at all times:

$$t \geqslant t_0 => \mu(t) \geqslant \mu(t_0)$$
$$t \geqslant t_0 => \alpha_i(t) \geqslant \alpha_i(t_0) \qquad \square$$

Lemma 2. *The total number of elements N in the stream S is greater than or equal to estimated frequencies in the monitored list. Let N_i be the number of elements with hash equal to i from the stream S:*

$$N = \sum_i N_i \geqslant \sum_j f_j \tag{4}$$

Proof. When the element is being monitored, every hit in S increments at most one counter among the m counters for f_j when the element is being monitored. This is true even when a new element replaces an older one, as the new element f_i enters with exactly the older value plus the observed hit. $\square$

Theorem 1. *Among all counters, the minimum counter value, μ, is no greater than N/m.*

Proof. Since the total number of elements is always higher than the total of the m counters:

$$N = \sum_i N_i \geqslant \sum_j f_j = \sum_j (f_j - \mu) + \sum_j \mu = \sum_j (f_j - \mu) + m\mu \tag{5}$$

and $f_j - \mu \geqslant 0$, then:

$$\mu \leqslant \left[N - \sum_j (f_j - \mu) \right] / m \leqslant N/m \qquad \square \tag{6}$$

Theorem 2. *An element x_i with $F_i > \mu$, where is F_i the real number of times x_i appears in the stream S, must exist in the monitored list.*

Proof. The proof is by contradiction. Assume x_i is not in the monitored list. Then, all hits should have been counted in some α_i. Note that not even replacements of other elements in the list can lower α_i so it would not be lower than F_i.

Since $F_i > \mu$, this would mean that $\alpha_i \geqslant F_i > \mu$ which breaks the hypothesis, so x_i should be in the list. $\square$

Lemma 3. *The value $f_i - e_i$ is always the number of times an element was observed since it was included in the list. Denoting F_i as the number of times that element x_i was effectively in the stream S, the real frequency:*

$$f_i - e_i \leqslant F_i \leqslant f_i \tag{7}$$

Proof. The first part is trivial to prove as F_i may be greater or equal to the number the element has effectively registered while it was monitored. To prove the second part, let one assume the opposite, i.e., $F_i > f_i$. In this case, $F_i > f_i > f_i - e_i + e_i = F_i > f_i - e_i + \mu(t_0)$, where t_0 is the moment immediately before x_i was included in the list. But this would mean that at that moment t_0, $F_i(t_0) > \mu(t_0)$, which breaks the previous hypothesis. So $F_i \leqslant f_i$. $\square$

In fact FSS maintains all the properties of Space-Saving, inclusively the guarantee of a maximum error of the estimate.

Theorem 3. *Whether or not x_i occupies the ith position in the monitored list, f_i, the counter at position i, is no smaller than F_i, the frequency of the element with rank i, x_i.*

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

1. The element x_i is not monitored. Thus, from previous property, $Fi \leqslant \mu$. Thus any counter in the monitored list is no smaller than *Fi*.
2. The element x_i is at position j, such that $j > i$. Knowing that the estimate of x_i is always bigger or equal to F_i, $f_j \geqslant F_i$. Since j is greater than i, then f_i is no smaller than f_j, the estimated frequency of x_i. Thus, $f_i \geqslant F_i$.
3. The element x_i is at position i, so $f_i \geqslant F_i$.
4. The element x_i is at position j, such that $j < i$. Thus, at least one element x_u with rank $u < i$ is located in some position y, such that $y \geqslant i$. Since the estimated frequency of x_u is no smaller than its frequency, F_u, and $u < i$, then the estimated frequency of x_u is no smaller than F_i. Since $y \geqslant i$, then the $f_i \geqslant f_y$, which is equal to the estimated frequency of x_u. Therefore, $f_i \geqslant F_i$.

Therefore, in all cases, $f_i \geqslant F_i$. □

As for Space-Saving, this is significant, since it enables estimating an upper bound on the rank of an element. The rank of an element x_i has to be less than j if the guaranteed hits of x_i are less than the counter at position j. That is $f_j < (f_i - e_i) \Rightarrow rank(x_i) < j$. Conversely, the rank of an element x_i is greater than the number of elements having guaranteed a number of hits larger than f_i. That is, $rank(x_i) > count(x_j | (f_j - e_j) > f_i)$. This helps establishing the order-preservation property among the top-k, as discussed later.

For FSS it can additionally be proven that μ is limited by the distribution of hits in the bitmap counter:

Theorem 4. *In* FSS, for any cell that has monitored elements:

$$\mu \leqslant max\{N_i\} = N_{max}, \quad \text{and} \tag{8}$$

$$E(\mu) \leqslant E(N_{max}) \tag{9}$$

Proof. The use of a bitmap counter-based on a hash function that maps values into a pseudo-random uniformly distributed range, gives the algorithm a probabilistic behavior. Let A be the monitored list, A_i the set of all elements in A such that $hash(v_j) = i$ and t_{j0} the moment element v_j was inserted in the monitored list:

$$hash(v_j) = i \Rightarrow e_j = a_i(t_{j0}) \leqslant a_i(t), \quad t \geqslant t_{j0}$$

The total number of elements N in stream S can be rewritten as the sum of N_i.

$$N = \sum_i N_i$$

This means that, in fact, a few hits may be discarded when a replacement is done, and that N_i is greater than (if at least a replacement was done) or equal (no replacements) to α_i plus the number of effective hits in monitored elements. Let A be the monitored list and A_i the set of all elements in A such that $hash(v_j) = i$:

$$N_i \geqslant \alpha_i + \sum_{j \in Ai} (f_j - e_j)$$

$$N_i \geqslant \alpha_i + \sum_{j \in Ai} (f_j - e_j) \geqslant \alpha_i + \sum_{j \in Ai} (f_j - \alpha_i)) \geqslant \alpha_i - c_i\alpha_i + \sum_{j \in Ai} f_j$$

$$N_i \geqslant \alpha_i - c_i\alpha_i(t_0) + \sum_{j \in Ai} f_j \geqslant \alpha_i - c_i\alpha_i + c_i\mu$$

$$c_i\mu \leqslant N_i + (c_i - 1)\alpha_i \leqslant c_iN_i$$

For all $c_i > 0$:

$$\mu \leqslant max\{N_i\}$$

This means that the maximum estimate error μ is lower than the maximum number of hits in any cell that has monitored elements. It also means that for any i:

$$\mu \leqslant max\{N_i\} = N_{max}$$

It is therefore trivial to conclude:

$$E(\mu) \leqslant E(N_{max}) \qquad \square$$

FSS has the very interesting property that error depends on the ratio between h and m and the number of samples N, and it can be guaranteed that this error is lower than in Space-Saving with a given probability. In fact for high values of N it gives a high probability of being much lower than Space-Saving. The following analysis considers a uniform distribution of elements, this is one of the worst and possibly less interesting cases to identify top-k elements as the data is not skewed but shows how the error bounds are improved by the bitmap counter filter.

Theorem 5. *In* FSS*, assuming a uniform distribution of the elements x and a hash function with a pseudo-random uniform distribution, the expected value of* μ*,* $E(\mu)$*, is a function of N, the number of elements in stream S and the number of elements h in bitmap counter.*

$$E(\mu) \leqslant E(N_{max}) = N(\Gamma(N+1, N/h)/\Gamma(N+1))^h - \sum_{N \geqslant i \geqslant 1} (\Gamma(i, N/h)/\Gamma(i))^h] \tag{10}$$

where $\Gamma(x)$ *is the complete gamma function and* $\Gamma(x,y)$ *is the incomplete gamma function.*

Proof. To estimate the N_i values in each cell, consider the Poisson approximation to the binomial distribution of the variables as in [1]. One will assume that the events counted are received during the period of time T with a Poisson distribution. So:

$$\lambda = N/T$$

Or considering $T = 1$, the period of measure:

$$\lambda = N$$

This replaces the exact knowledge of N by an approximation as $E(x) = N$, with $x = P(\lambda)$. Although the expected value of this random variable is the same as the initial value, the variance is much higher: it is equal to the expected value. This translates in introducing a higher variance in the analysis of the algorithm. However, since the objective is to obtain a maximum bound, this additional variance can be considered.

Consider that the hash function $h(x)$ distributes x uniformly over the h counters. The average rate of events falling in counter i can then be calculated as $\lambda_i = \lambda/h$.

The probability of counter i receiving k events in $[0,1]$ is given by:

$$P(N_i = k|t = 1) = P_{ik}(1) = \lambda_i^k e^{-\lambda i T}/k! = (N/h)^k e^{-N/h}/k!$$

And the cumulative distribution function is:

$$P(N_i \leqslant k|t = 1) = \Gamma(k+1, N/h)/\Gamma(k+1)$$

To estimate $E(N_{max})$ lets first consider the probability of having at least one N_i larger than k:

$$P(N_{max} \leqslant k|t = 1) = P(N_i \leqslant k \text{ for all } i|t = 1)$$

As the Poisson distributions are infinitely divisible, the probability distributions and each of the resulting variables are independent:

$$P(N_{max} \leqslant k|t = 1) = \prod P(N_i \leqslant k|t = T) = \prod \Gamma(k+1, N/h)/\Gamma(k+1) \Longleftrightarrow$$
$$P(N_{max} \leqslant k|t = 1) = [\Gamma(k+1, N/h)/\Gamma(k+1)]^h = g(k+1)$$

and since k is integer:

$$\begin{aligned}
P(N_{max} = k|t = 1) &= P(N_{max} \leqslant k|t = 1) - P(N_{max} \leqslant k-1|t = 1) \Longleftrightarrow P(N_{max} = k|t = 1) \\
&= (\Gamma(k+1, N/h)/\Gamma(k+1))^h - (\Gamma(k, N/h)/\Gamma(k))^h \Longleftrightarrow P(N_{max} = k|t = 1) \\
&= [(\Gamma(k+1, N/h)/k)^h - (\Gamma(k, N/h))^h]/\Gamma(k)^h \Longleftrightarrow E(\mu) \leqslant E(N_{max}) \\
&= \sum_i i[(\Gamma(i+1, N/h)/\Gamma(i+1))^h - (\Gamma(i, N/h)/\Gamma(i))^h] \Longleftrightarrow E(N_{max}) = \sum_i i[g(i+1) - g(i)] \\
&= [g(2) - g(1)] + 2[g(3) - g(2)] + \cdots + N[g(N+1) - g(N)] \Longleftrightarrow E(N_{max}) = Ng(N+1) - \sum_{N \geqslant i \geqslant 1} g(i) \\
&= N(\Gamma(N+1, N/h)/\Gamma(N+1))^h - \sum_{N \geqslant i \geqslant 1} (\Gamma(i, N/h)/\Gamma(i))^h] \qquad \square
\end{aligned}$$

Theorem 5 allows h and N to be chosen so that the expected value for the maximum estimated error is as low as needed. It also allows h and N to be chosen so that the maximum estimated error is as low as needed with a given probability φ.

Figs. 6 and 7 illustrate the behavior of N_i and N_{max} probability mass function using the N/h as the x axis. For $N = 12{,}000$ and $h = 240$:

For $N = 120{,}000$ and $h = 240$ the N_i function is relatively narrower and therefore N_{max} approaches $N/h = 1$:

Table 1 shows how the cumulative distribution function and the expected value for N_{max} evolve with N.

This gives an intuitive justification for using the error filter: the estimate error can be made lower than a certain value (per example 120%) of N/h for large enough values of N by using more cells in the filter than in the monitored list (since each entry in the monitored list requires space equivalent to at least 3 counters).

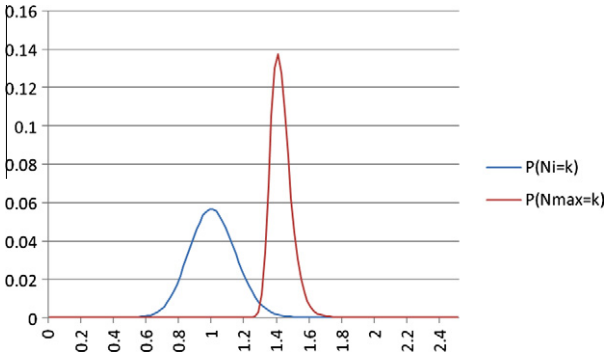

Fig. 6. Probability mass function for N_i and N_{max}, $N = 12{,}000$ and $h = 240$.

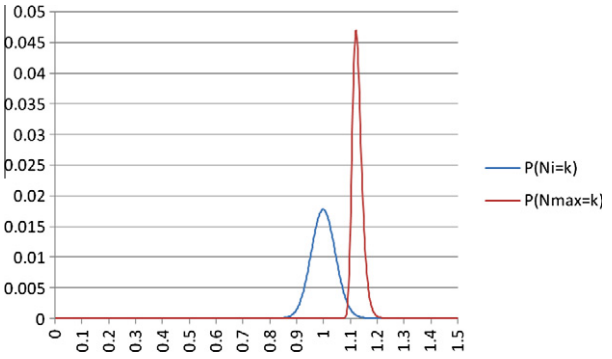

Fig. 7. Probability mass function for N_i and N_{max}, $N = 120{,}000$ and $h = 240$.

Table 1
$E(N_{max})$ as a function of k, N and $P(N_{max} \leqslant k)$.

		$P(N_{max} \leqslant k) \geqslant \varphi$			$E(N_{max})$
		$\varphi = 0.9$	$\varphi = 0.99$	$\varphi = 0.999$	
$h = 240$, $N = 12{,}000$	k	75	80	84	70.96
	$k/(N/h)$	1.5	1.6	1.68	1.419
$h = 240$, $N = 120{,}000$	k	576	590	603	563.89
	$k/(N/h)$	1.152	1.118	1.206	1.128

Theorem 6. *Assuming a uniform distribution of the elements x and a hash function with a pseudo-random uniform distribution, the probability of the estimate error μ being lower than an value τ can be made as low as required, given N the number of elements in stream S, by setting the number of elements in bitmap counter h.*

$$P(\mu \leqslant \tau) \leqslant [1/2(1 + erf((k - N/h)/(2N/h)^{1/2}))]^h, \tau \geqslant 1 \tag{11}$$

where erf is the Gauss error function.

Proof. Consider that a Poisson distribution with parameter λ is approximately normal for large λ. The approximating normal distribution has parameters $\mu = \sigma^2 = \lambda$. This can be seen as a different approximation to the real binomial distribution. For large N/h it is as good as the Poisson distribution.

Consider k as the highest integer lower than τ. Assuming every N_i as independent variables with a normal distribution with $\mu = \sigma^2 = N/h$, then the probability of N_i being less or equal to k is:

$$P(N_i \leqslant k | t = 1) = 1/2[1 + erf((k - N/h)/(2N/h)^{1/2})]$$

So,

$$P(N_{max} \leqslant k | t = 1) = \prod P(N_i \leqslant k | t = 1) = [1/2(1 + erf((k - N/h)/(2N/h)^{1/2}))]^h \qquad \square$$

Theorem 7. *In* FSS, *assuming a uniform distribution of the elements x and a hash function with a pseudo-random uniform distribution, the estimate error is bound by $\mu < \varepsilon N/h$ with a given probability φ if:*

$$N > 2h/(\varepsilon - 1)^2[erf^{-1}(2\varphi^{1/h} - 1)]^2, \quad \text{and} \quad \varepsilon > 1 \tag{12}$$

ε is in fact the margin to consider when dimensioning h to achieve a given error with φ probability when N elements are expected.

Proof. Consider φ such that:

$$P(N_{max} \leqslant k|t=1) \geqslant \varphi \Longleftrightarrow P(N_{max} \leqslant k|t=1) = [1/2(1+erf((k-N/h)/(2N/h)^{1/2}))]^h$$
$$\geqslant \varphi \Longleftrightarrow 1/2(1+erf((k-N/h)/(2N/h)^{1/2})) \geqslant \varphi^{1/h} \Longleftrightarrow erf((k-N/h)/(2N/h^{1/2}))$$
$$\geqslant 2\varphi^{1/h}-1 \Longleftrightarrow (k-N/h)/(2N/h)^{1/2}) \geqslant erf^{-1}(2\varphi^{1/h}-1)$$

With $k = \varepsilon N/h$:

$$(\varepsilon N/h - N/h)/(2N/h)^{1/2}) = (N/h)^{1/2}(\varepsilon-1)/(2)^{1/2} \geqslant erf^{-1}(2\varphi^{1/h}-1)N > 2h/(\varepsilon-1)^2[erf^{-1}(2\varphi^{1/h}-1)]^2 \qquad \square$$

6. Frequent elements

This section presents the properties of FSS regarding answering frequent elements queries. It can be shown that in fact FSS retains the properties of Space-Saving regarding given guarantees.

As in Space-Saving, in order to answer queries about the frequent elements, the monitored list is sequentially traversed until an element with frequency less than defined minimum frequency or support is found. This is proven by Metwally et al. in [14]. Frequent elements are reported in an ordered list of elements. An element, x_i, is guaranteed to be a frequent element if its guaranteed number of hits, $f_i - e_i$, exceeds $ceil(\varphi N)$, the minimum support. If for each reported element x_i, $f_i - e_i > ceil(\varphi N)$, then the algorithm guarantees that all, and only the frequent elements are reported.

Answering frequent elements queries follow the Space-Saving algorithm. As for Space-Saving, assuming no specific data distribution, FSS uses a number of counters m of:

$$min(|A|, 1/\varepsilon) \tag{13}$$

where $|A|$ represents the number of elements in the monitored list, to find all frequent elements with error ε. Any element, x_i, with frequency $F_i > \varphi N$ is guaranteed to be reported. The proof is omitted as it follows Space-Saving and it has been proved in Metwally et al. [14]

For the more interesting case of noiseless Zipfian data with parameter α, to calculate the frequent elements with error rate ε, FFS follows Space-Saving and uses only:

$$min(|A|, (1/\varepsilon)^{1/\alpha}, 1/\varepsilon) \text{ counters} \tag{14}$$

This is regardless of the stream permutation. The proof is omitted as it follows Space-Saving.

7. Top-*k* elements and order

For the top-k elements, the FSS algorithm can output the first k elements. An element, x_i, is guaranteed to be among the top-k if it's guaranteed number of hits, $f_i - e_i$, exceeds f_{k+1}, the over-estimated number of hits for the element in position $k+1$. Since, f_{k+1} is an upper bound on F_{k+1}, the hits of the element of rank $k+1$, E_{k+1}, then x_i is in the top-k elements.

The results have guaranteed top-k if by simply inspecting the results, the algorithm can determine that the reported top-k elements are correct. FSS and Space-Saving report a guaranteed top-k if for all i:

$$(f_i - e_i) \geqslant f_{k+1} \tag{15}$$

where $i \leqslant k$.

That is, all the reported k elements are guaranteed to be among the top-k elements.

In addition to having guaranteed top-k, the order of elements among the top-i 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 $i \leqslant k$.

Regardless of the data distribution, to solve the query of the approximate top-k in the stream S within an error ε, FSS and Space-Saving use:

$$min(|A|, N/\varepsilon F_k) \text{ counters} \tag{16}$$

Any element with frequency higher than $(1-\varepsilon)F_k$ is guaranteed to be monitored. The proof is omitted as it follows Space-Saving and it has been proved in Metwally et al. [14].

As for Space-Saving, assuming the data is noiseless Zipfian with parameter $\alpha > 1$, to calculate the exact top-k, FSS uses:

$$min(|A|, O((k/\alpha)^{1/\alpha}k)) \text{ counters} \tag{17}$$

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

$$min(|A|, O(k^2 ln(|A|))) \tag{18}$$

This is regardless of the stream permutation. Also, the order among the top-k elements is preserved. The proof is omitted as it follows Space-Saving and it has been proved in Metwally et al. [14].

8. Implementation

The practical implementation of FSS may include the Stream-Summary data structure presented in [14]. An index by hash code may be used to speed up searches (a binary search may be used), but depending on the number of elements in the monitored list, other options may as well be used. For small number of m (a few tens of elements), a simple array of m elements can be a viable and fast option. To speed up insertions and replacements, an additional array with the indexes of each element ordered by f_i may be used. This trades some performance for a very compact implementation. This may be the preferred option for telecommunication related implementations where the total number of elements in the list to be kept is 50 or even less.

Any implementation of FSS has to use a bitmap counter to hold the α_i counters. The c_i counters are however optional as they may well be replaced by a single bit indicating the existence or not of elements in the monitored list with that hash code. When an element is removed the bit is kept set. If the following search of element with hash i do not find an element then the bit is cleared. This reduces memory at the expense of a potential additional search in the monitored list (one that Space-Saving would always do anyway). Eventually c_i counters may even be dropped at the expense of a search in the monitored list for every element in the stream S.

A further improvement in the memory usage of FSS can be made by using filter counters smaller than the counters needed for the monitored list. In the above example, with $N = 120{,}000$, 17 bits would be needed for the monitored list counters. For the filter, only counters that map values between $min\{N_i\}$ and $max\{N_i\}$ are required, since all counters can be decremented by $min\{N_i\}$, and the common factor $min\{N_i\}$ stored in a single variable.

Using a similar approach to the $max\{N_i\}$, it can be seen that $N_min = max\{N_i\}$ has the following behavior:

$$P(N_{min} > k|t = 1) = \prod(1 - P(N_i \leqslant k|t = 1)) = (1 - P(N_i \leqslant k|t = T))^h \Longleftrightarrow P(N_{min} > k|t = 1) = (1 - \Gamma(k+1, N/h)/\Gamma(k+1))^h \quad (19)$$

Since $E(N_i) - N_{min} \leqslant N_{max} - E(N_i)$ due to Poisson distribution properties:

$$max\{N_i\} - min\{N_i\} \leqslant 2[N_{max} - E(N_i)] = 2N_{max} - 2N/h \quad (20)$$

For the example with $N = 120{,}000$, the value of N_{min}will be greater than 403 with probability 0.999. This means that a counter with a range of 256 will be more than enough to hold the $max\{N_i\} - min\{N_i\}$ value with a very high probability. It also means that a counter could have only 8 bits instead of 17 and therefore a single entry in the monitored list could be transformed in 6 cells for filtering.

In this case the estimate error for $N > 120{,}000$ and $h = 240$ with probability 0.999 will be lower than:

$$\mu \leqslant N_{max} \approx 1.206\ N/h = 603 \quad (21)$$

To ensure a similar error with Space-Saving, $m = 120{,}000/603 \approx 199$ would be needed. This would require $3 * 199 = 597$ counters of 17 bits each. FSS requires 240 counters of 8 bits each (equivalent to roughly 40 entries in Space-Saving), plus enough entries in the monitored list to hold at least the intended top-k values. This is in fact much less space than the required for Space-Saving at the expense of a very small probability of error.

In fact experimental results show that the μ will usually be significantly lower than N_{max}. Another interesting finding from experiments is that the filter cells can in fact misrepresent N_{min} and allow for cell underflow (by discarding very low and improbable values of N_i due to insufficient counter range as N_{max} increases) without significant impacts in the error estimate.

When the element identifier has the same size as the counters, Space-Saving, requires 3 variables for each entry in the list. Memory usage is still increased by the need of pointers for the Stream-Summary structure (as described in [14]). Using indexes instead of pointers will further reduce this. When using c bits per counter, Space-Saving will need 3 counters for each entry plus m counters with $log_2 m$ bits (the indexes only need to store values from 0 to $m - 1$):

$$S_{SS}(m) \approx 3m\ c + m\ log2\ m\ bits \quad (22)$$

FSS requires the same $3m$ counters for element storage plus an index and h counters plus a single bit. The counters in the filter can however be smaller. For the filter only counters that map values between $min\{N_i\}$ and $max\{N_i\}$ are required as all counters can always be decremented by $min\{N_i\}$. For most situations counters can have half the size of the counters in the monitored list, $b = c/2$. This can be resumed in:

$$S_{FSS}(m) \approx 3m\ c + m\ log2\ m + h(c/2 + 1) \quad (23)$$

The practical dimensioning rule used in the following experiments was to trade 1 element in the Space-Saving monitored list for 6 counters in the filter table. This would keep half the list and change the rest for filter. So with $h = 6m$ and with $c >> 1$:

$$S_{SS}(mSS)/S_{FSS}(m_{FSS}) \approx 1 \Rightarrow m_{FSS} \approx 1/2 m_{SS} \quad (24)$$

9. Experimental results

The first set of tests with Space-Saving and FSS try to show the performance of both algorithms in a typical telecommunications scenario where the top-10 or top-20 destinations are needed per customer. This does not require a huge number of

elements in the monitored list since the number of destinations on a typical customer is relatively small: as previously shown, in 500 calls an average 98 distinct destinations is expected. As most of the traffic is made to the top-10 or top-20 destinations, these lists are enough for most practical purposes (either fraud or marketing related). As also previously said, 500 is a large enough sample of calls, since most users will take more than a month to make 500 voice calls.

The results for the test of the two algorithms are shown in Table 2. SS_m is the m parameter for Space-Saving, and FSS_m and FSS_h are respectively the m and h parameters for FSS.

Update performance indicators are also given. *Inserts* are the average number of elements inserted in the monitored list, *Replacements* the number of elements removed from the monitored list, and *Hits* the number of hits in the monitored list. Note that FSS has one operation more than Space-Saving, the bitmap counter inspection and possible update – but these are simple and fast operations.

Full List Recall is the ratio between the correct top-k elements included in the full monitored list and i, *Full List Precision* is the ratio between the correct top-k elements included in the full monitored list and its size m.

Top-k Recall (and *Top-k Precision*) is the ratio between correct top-k elements included in the returned top-k elements and k. This is the main quality indicator of the algorithms as this reflects the number of correct values obtained when fetching a top-k list.

Estimate error is the square root of minimum squared error (MSE) of the frequency estimate for the top-k elements returned (out of k elements). It does not include incorrect elements or the error of estimate of top-k elements not returned.

The first set of trials, Trials 1 and 2, uses relatively small blocks of calls per customer and identifies the top-10 or top-20 destinations. Note that for Space-Saving $m = 3k$ is used, so 30 elements are used in the list when finding top-10 elements and

Table 2
Parameters and results for FSS ans Space-Saving algorithms in Trials 1 and 2.

		Trial 1.0	Trial 1.1	Trial 2.0	Trial 2.1
Block size		500	800	500	800
Average Distinct		98	130	98	130
Number of Samples		962	715	962	715
Top-k		10	10	20	20
SS m		30	30	60	60
FSS m		15	15	30	30
FSS h		90	90	180	180
FSS	Inserts	54.89	66.89	71.38	84.23
	Replacements	39.92	51.89	41.61	54.35
	Hits	296.86	477.73	346.69	561.91
	M	9.73	14.8	4.52	6.73
	Full List Recall	0.9758	0.9768	0.9892	0.9893
	Full List Precision	0.6916	0.6748	0.737	0.71
	Top-k Recall	0.9433	0.9409	0.9604	0.9633
	Top-k Precision	0.9433	0.9409	0.9604	0.9633
	Estimate Error	1.27	1.52	0.81	0.95
SS	Inserts	142.8	222.64	109.94	161.3
	Replacements	112.95	192.71	51.39	101.88
	Hits	356.16	577.35	386.41	636.46
	M	8.96	15.39	2.52	4.7
	Full List Recall	0.9705	0.9631	0.9958	0.9931
	Full List Precision	0.3475	0.3342	0.3856	0.3629
	Top-k Recall	0.9002	0.8858	0.967	0.956
	Top-k Precision	0.9002	0.8858	0.967	0.956
	Estimate Error	1.35	2.18	0.3715	0.64

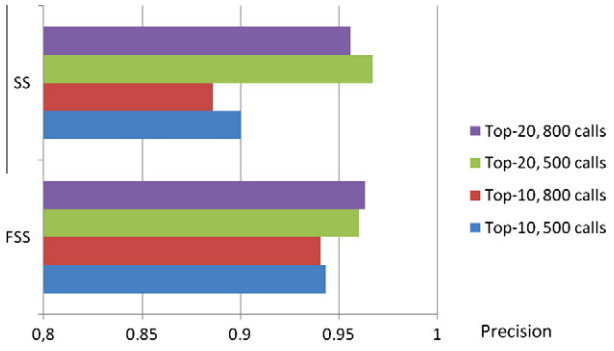

Fig. 8. Top-k Precision for SS and FSS algorithms in Trials 1 and 2.

60 when finding top-20 elements. Note that this is a significant size for the list, as an average of 98 distinct numbers in a 500 call block is expected (130 in an 800 call block).

In these trials it can be observed that both Top-k Recall and Top-k Precision of FSS are marginally better than that of Space-Saving for top-10. It is interesting to see that in the top-20 trial the results are much closer. In fact, Space-Saving with $m = 60$ is able to store a large part of the distinct values and therefore the improvement of FSS is not relevant.

Table 3
Parameters and execution results for Trial 3.

		Trial 3.0	Trial 3.1	Trial 3.2
Block size		5000	10,000	20,000
Average distinct		3130	5322	8774
Number of samples		166	83	41
Top-k		20	20	20
SS m		200	200	200
FSS m		100	100	100
FSS h		600	600	600
FSS	Inserts	1103.32	1921.33	3519.17
	Replacements	1003.32	1821.33	3419.17
	Hits	393.01	780.09	1522.68
	μ	13.79	25.4	48.41
	Full List Recall	0.6287	0.6153	0.5775
	Full List Precision	0.1552	0.1386	0.1217
	Top-k Recall	0.4996	0.4686	0.4439
	Top-k Precision	0.4996	0.4686	0.4439
	Estimate error	5.57	9.97	18.47
SS	Inserts	4234.83	8457.69	16941.8
	Replacements	4034.83	8257.69	16741.8
	Hits	765.16	1542.3	3058.19
	μ	23.92	48.86	98.8
	Full List Recall	0.4278	0.3701	0.3376
	Full List Precision	0.0528	0.0419	0.0356
	Top-k Recall	0.2539	0.1987	0.1585
	Top-k Precision	0.2539	0.1987	0.1585
	Estimate error	14.87	31.09	63.82

Table 4
Parameters and execution results for Trial 4.

			Trial 4.0	Trial 4.1	Trial 4.2
Block size			5000	10,000	20,000
Average distinct			3130	5322	8774
Number of samples			166	83	41
Top-k			20	20	20
FSS	$m = 100, h = 600$	μ	13.79	25.4	48.41
		Top-k Recall and Precision	0.4996	0.4686	0.4439
		Estimate error	5.57	9.97	18.47
	$m = 200, h = 1200$	μ	8.04	14.48	27.07
		Top-k Recall and Precision	0.7894	0.7734	0.7597
		Estimate error	2.36	3.37	5.34
	$m = 300, h = 1800$	μ	5.99	10.51	19.36
		Top-k Recall and Precision	0.8858	0.8951	0.889
		Estimate error	1.44	2.179	2.6
	$m = 400, h = 2400$	μ	5	8.92	15.97
		Top-k Recall and Precision	0.9246	0.9265	0.939
		Estimate error	1.25	1.57	2.019
SS	$m = 200$	μ	23.92	48.86	98.8
		Top-k Recall and Precision	0.2539	0.1987	0.1585
		Estimate error	14.87	31.09	63.82
	$m = 400$	μ	11.84	23.91	48.87
		Top-k Recall and Precision	0.4987	0.4138	0.3402
		Estimate error	4.71	9.5	18.12
	$m = 600$	μ	7.46	15.79	32.07
		Top-k Recall and Precision	0.7322	0.6192	0.5583
		Estimate error	1.82	4.4055	7.87
	$m = 800$	μ	5	11.42	23.82
		Top-k Recall and Precision	0.853	0.7903	0.7329
		Estimate error	1.25	2.12	4.08

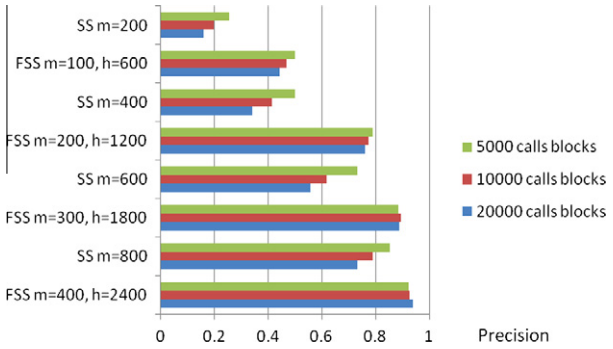

Fig. 9. Top-20 Precision with increasing number of calls in Trials 4.

Table 5
Parameters and execution results for Trial 5.

			Trial 5.0	Trial 5.1	Trial 5.2
SS m			2000	4000	8000
FSS m			1000	2000	4000
FSS h			6000	12,000	24,000
FSS	Top-10	μ	207	114	60
		Top-k Precision and Recall	0.8	0.9	0.9
		Estimate error	59.42	36.45	13.06
	Top-20	μ	207	114	60
		Top-k Precision and Recall	0.85	0.9	0.95
		Estimate error	59.67	31.27	14.3
	Top-50	μ	207	114	60
		Top-k Precision and Recall	0.86	0.94	0.96
		Estimate error	61.69	26.29	11.52
	Top-100	μ	207	114	60
		Top-k Precision and Recall	0.76	0.89	0.97
		Estimate error	60.38	25.95	11.39
SS	Top-10	μ	388	183	80
		Top-k Precision and Recall	0.8	0.8	0.9
		Estimate error	145.68	66.65	25.5
	Top-20	μ	388	183	80
		Top-k Precision and Recall	0.7	0.85	0.9
		Estimate error	170.97	69.87	27.05
	Top-50	μ	388	183	80
		Top-k Precision and Recall	0.68	0.84	0.92
		Estimate error	171.95	69.26	26.24
	Top-100	μ	388	183	80
		Top-k Precision and Recall	0.55	0.74	0.87
		Estimate error	175.66	67.56	24.85

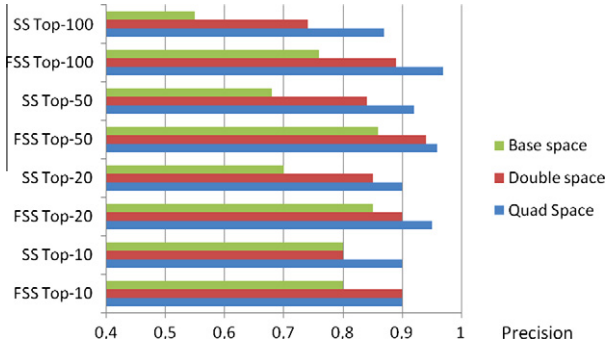

Fig. 10. Top-k Precision with increasing space in Trials 5.

Note also that the number of update operations in the monitored list is significantly lower for FSS.

Fig. 8 presents Top-k Precision for both SS and FSS algorithms in Trials 1 and 2.

The following block of trials, Trials 3, compares SS with FSS for a larger number of calls. In this case the calls are not from a single user but calls made in a consecutive period of time. Increasing the time period gives us more calls. In this case only the top-20 elements are found. The number of distinct elements per block of calls is now much higher so m and h increased proportionally for both algorithms. Table 3 shows the results for this second block of trials:

The absolute results are not reasonable for practical purposes, but the behavior of each algorithm is now more evident. Anyway only a very small list of elements is being used when compared to the number of distinct values in the sample (the list is less than 6.4% the number of distinct elements). Space-Saving with $m = 200$ is only able to find 25.39% of the top-20 elements when returning the first 20 elements in the list for 5000 calls and degrades significantly the performance when N increases. On the other hand, FSS is able to find nearly half of the top-20 elements and its performance degrades less when N increases.

Table 4 and Fig. 9 present the results for Trials 4, with an increasing value of m.

Overall, the performance of FSS is consistently better than that of Space-Saving for similar memory usage. A very relevant aspect is that FSS performance does not degrades with the increasing number of calls as in the case of Space-Saving.

FSS with $m = 200$ and $h = 1200$, for $N = 20{,}000$, behaves better than Space-Saving with $m = 800$ that uses the double of the space.

Also important, FSS performance for larger blocks of calls improves more than for smaller blocks as the overall space allocation increases (for $m = 100$, $h = 600$ performance is better for 5000 calls blocks than for 2000, for $m = 400$, $h = 2400$, it is the inverse). This is due to the behavior of the stochastic filtering of error, the ratio between N_i and $N_{max}$is reduced as the number of calls increases (as shown in Figs. 6 and 7).

Table 5 and Fig. 10 show the results for Trials 5 when using the whole set of calls, with $N = 800{,}000$, totaling 112,462 distinct values. It can be seen that FSS is consistently better than SS for similar memory usage.

10. Conclusions

This paper presents a new algorithm that builds on top of the best existing algorithm for answering the top-k problem. It effectively merges the best properties of two distinct approaches to this problem, the counter-based techniques and sketch-based ones. The new algorithm keeps the already excellent guarantees provided by Space-Saving algorithm regarding inclusion of elements in the answer, ordering of elements, maximum estimation error and provides probabilistic guarantees of increased precision.

The FSS algorithm filters and splits the error of Space-Saving algorithm through the use of a bitmap counter. It is also shown that this approach minimizes the operations of update of the monitored elements list by avoiding elements with insufficient hits being inserted in the list. This helps to minimize the overall error and therefore the error associated with each element. It eliminates the excess of trail elements from the data stream that Space-Saving usually includes in the monitored list and that have a very high estimation error.

Although FSS requires an additional bitmap counter, smaller counters are used overall allowing a trade-off. By replacing some of the entries in the Space-Saving monitored list by additional cells in this bitmap counter it is possible to keep the same space but have better expected performance out of the algorithm. In this paper a single entry in the monitored list was exchanged for 6 additional cells in the bitmap counter. This seems to be a reasonable exchange rate specially if one considers the usual counter size (Space-Saving implementations usually use either 16 or 32 bits counters, FSS may use 8 bit or 16 bit counters). This exchange rate coupled to the probabilistic guarantees of FSS (especially for large values of N, the stream size) achieves better results. In fact FSS exchanges deterministic guarantees for tighter probabilistic ones.

This paper presents experimental results that detail improvements over Space-Saving algorithm, both in precision and in performance when using similar memory space. Memory consumption was key in the analysis, since the practical problem being solved is memory bound. In this regard, the use of FSS is envisioned with even less memory than Space-Saving and with better precision. Although execution time will depend on specific aspects of the implementation, the number of operations required by each algorithm was also detailed and points to reductions in execution time.

FSS is a low memory footprint algorithm that can answer not only the top-k problem for large number of transactions but also the problem of answering huge number of top-k problems for a relatively small number of transactions. As such, its applicability goes much beyond telecommunications or retail applications. It can be used in any other domain as long as the appropriate implementation choices and dimensioning are made.

Acknowledgement

This work was in part supported by FCT (INESC-ID multi annual funding) through the PIDDAC Program funds.

References

[1] D. Bertsekas, Dynamic Programming and Optimal Control, Vol. 1, Athena Scientific, 1995.
[2] G. Cormode, S. Muthukrishnan. What's Hot and What's Not: TrackingMost Frequent Items Dynamically, in: Proceedings of the 22nd ACM PODS Symposium on Principles of Database Systems, Pages 296–306, 2003.

[3] M. Datar, A. Gionis, P. Indyk, R. Motwani, Maintaining Stream Statistics Over Sliding Windows, SIAM Journal on Computing 31 (6) (2002).
[4] E. Demaine, A. López-Ortiz, 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.
[5] X. Dimitropoulos, P. Hurley, A. Kind, Probabilistic lossy counting: an efficient algorithm for finding heavy hitters, ACM SIGCOMM Computer Communication Review 38 (1) (2008).
[6] C. Estan, G. Varghese. New directions in traffic measurement and accounting, in: Proceedings of SIGCOMM 2002 (2002), ACM Press. (Also: UCSD technical report CS2002-0699, February, 2002; available electronically).
[7] C. Estan, G. Varghese, New directions in traffic measurement and accounting: focusing on the elephants, ignoring the mice, ACM Transactions on Computer Systems 21 (3) (2003) 270–313.
[8] C. Estan, G. Varghese, M. Fisk, Bitmap algorithms for counting active flows on high speed links. Technical Report CS2003-0738, UCSD, 2003.
[9] L. Fan, P. Cao, J. Almeida, A. Broder, Summary cache: a scalable wide-area web cache sharing protocol, IEEE/ACM Transactions on Networking 8 (3) (2000) 281–293, doi:10.1109/90.851975.
[10] P. Flajolet, N. Martin, Probabilistic counting algorithms for data base applications, Journal of Computer and System Sciences 31 (2) (1985).
[11] T. Hu, S. Sung, H. Xiong, Q. Fu, Discovery of maximum length frequent itemsets, Information Sciences 178 (2008) 69–87.
[12] N. Manerikar, T. Palpanas, Frequent items in streaming data: an experimental evaluation of the state-of-the-art, Data & Knowledge Engineering 68 (4) (2009) 415–430.
[13] G. Manku, 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.
[14] A. Metwally, D. Agrawal, A. Abbadi, Efficient Computation of Frequent and Top-*k* Elements in Data Streams, Technical Report 2005-23, University of California, Santa Barbara, 2005.
[15] J. Misra, D. Gries, Finding repeated elements, Science of Computer Programming 2 (1982) 143–152.
[16] S. Tanbeer, C. Ahmed, B. Jeong, Y. Lee, Efficient single-pass frequent pattern mining using a prefix-tree, Information Sciences 179 (5) (2009) 559–583.
[17] S. Tanbeer, C. Ahmed, B. Jeong, Y. Lee, Sliding window-based frequent pattern mining over data streams, Information Sciences 179 (22) (2009) 3843–3865.
[18] K. Whang, B. Vander-Zanden, H. Taylor, A linear-time probabilistic counting algorithm for database applications, ACM Transactions on Database Systems 15 (2) (1990).
[19] J. Yu, Z. Chong, H. Lu, Z. Zhang, A. Zhou, A false negative approach to mining frequent itemsets from high speed transactional data streams, Information Sciences 176 (2006) 1986–2015.