Evolving Systems
DOI 10.1007/s12530-010-9020-z

ORIGINAL PAPER

Finding top-*k* elements in a time-sliding window

Nuno Homem · Joao Paulo Carvalho

Received: 31 August 2010 / Accepted: 27 October 2010

Abstract Identifying the top-*k* most frequent elements is one of the many problems associated with data streams analysis. It is a well-known and difficult problem, especially if the analysis is to be performed and maintained up to date in near real time. Analyzing data streams in time sliding window model is of particular interest as only the most recent, more relevant events are considered. Approximate answers are usually adequate when dealing with this problem. This paper presents a new and innovative algorithm, the Filtered Space-Saving with Sliding Window Algorithm (FSW) that addresses this problem by introducing in the Filtered Space Saving (FSS) algorithm an approximated time sliding window counter. The algorithm provides the top-*k* list of elements, their frequency and an error estimate for each frequency value within the sliding window. It provides strong guarantees on the results, depending on the elements real frequencies. Experimental results detail performance on real life cases.

Keywords Approximate algorithms · Top-*k* algorithms · Most frequent · Estimation · Data stream frequencies · Sliding window

1 Introduction

Identifying the top-*k* most frequent elements in a data stream is a very relevant problem in several domains, since top-*k* elements are necessary for a large number of different applications. Examples include; traffic shaping systems that control IP quality of service (which require information on the higher-traffic sources and destinations), one-to-one marketing applications (finding the most frequently requested products) or fraud detection systems (finding the most heavily used services). Finding the most frequent elements in a given period can be used to model a system, and tracking the changes in most frequent elements can model the system evolution. In an evolving system, the most frequent events produced by an individual or system may describe, or help to describe, its behavior in a dynamically changing and evolving environment.

Applications such as IP session logging, telecommunication records, financial transactions or real time sensor data, generate so much information that analysis needs to be done on the arriving data in near-real time. The data stream model has been widely used as a computational model for such applications. In a data stream, a huge number of events, possibly infinite, are observed, and storage of the entire set is not viable. As a result, only a set of summaries or aggregates may be kept; relevant information is extracted and transient or less significant data is discarded. Since summaries or synopsis must be created in a single pass, optimizing the process is therefore critical. Fortunately, approximate answers may be sufficient in many situations.

In many domains, individual or system behavior does not remain static but evolves over time; the sliding window model will be considered by making the assumption that behavior is stable during the time window. In the sliding window model only the last N elements or the elements observed during the last period of time T are considered. Usually the most recent data is the most interesting. In

N. Homem (✉) · J. P. Carvalho
TULisbon, Instituto Superior Técnico, INESC-ID,
R. Alves Redol 9, 1000-029 Lisbon, Portugal
e-mail: nuno.homem@hotmail.com

J. P. Carvalho
e-mail: joao.carvalho@inesc-id.pt

Published online: 18 November 2010

many cases, recent data can be used to predict future behavior or trends. In this paper, one will consider the more general case of the last elements observed during a period of time T, as the N last elements problem can be easily addressed by considering that each element arrives at a fixed rate N/T.

Classical exact top-k algorithm requires the full list of distinct elements to be kept. The list is always searched to insert new elements or to update existing element counters. Exact top-k algorithm may require huge amounts of memory. The problem is even worse when a sliding window view is required, since events out of the window must be removed from the list. Exact algorithm requires the full list of events to be stored.

This work proposes a new algorithm for identifying the approximate top-k elements and their frequencies while providing an error estimate for each frequency. The Filtered Space-Saving with Sliding Window (FSW) algorithm is a novel approach that introduces the sliding model constraints into the top-k problem. It uses an approximation to a T time sliding window by considering p basic sub-windows of fixed duration. Each of the basic sub-windows will have a fixed start and end time. When a basic sub-window expires, a new one will replace it. The events observed in the expired window will no longer be active or contribute to the results. Although this introduces a "*jumping*" sliding window, the approximation is good for most purposes.

The algorithm gives strong guarantees on the inclusion of elements in the list depending on their real frequency. Since the algorithm provides an error estimate for each frequency, it also provides the possibility to control the quality of the estimate.

The algorithm is innovative as it builds on Filtered Space Saving (FSS) algorithm (Homem and Carvalho 2010b) and introduces the time dimension into the results. FSS presents the best known performance in approximate top-k algorithms; it provides strong error guarantees on the error estimate, order of elements and inclusion of elements in the list depending on their real frequency. FSS also provides stochastic bounds on the error and expected error estimates.

2 Relation with previous work

To solve the unrestricted top-k problem with reasonable resources, approximate algorithms have been proposed. Those algorithms can roughly be divided into two classes: Counter based techniques and Sketch based techniques. Books such as (Aggarwal 2007) and (Muthukrishnan 2005) present many of the existing algorithms and data structures to handle data streams.

2.1 Top-k counter based techniques

Some considerable work has been done in the unrestricted top-k problem. Metwally et al. (2005) proposed the Space-Saving algorithm. 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 occurrences 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.

Demaine et al. (2002) presented a deterministic algorithm to answer the ε-approximate frequent problem without making any assumption on the distribution of the item frequencies. Although related, frequent and top-k problems are distinct. In the frequent elements problem the threshold is given a priori and in many cases the relative frequencies of each element are not relevant. The algorithm is simple and elegant: it needs $1/\varepsilon$ simple counters to count the items in the stream. A counter is used for each possible item and initialized to 0. When a new item is read, the counter is incremented, and if after the increment there are more than $1/\varepsilon$ counters with value greater than 0, each of these counters will be decremented once. When all the N elements in the data stream are processed, the set of items whose counters have value at least $(\theta - \varepsilon)N$ are returned, with θ the given threshold.

Metwally et al. (2005) provide in a comparison between several algorithms such as Lossy Counting (Manku and Motwani 2002), Probabilistic Lossy Counting (Dimitropoulos et al. 2008) and Frequent (Demaine et al. 2002).

2.2 Top-k 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 that 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 a higher memory footprint when comparing with Space-Saving algorithm. Additionally, the entire bitmap counter needs to be scanned and elements sorted to answer the query. Some algorithms like GroupTest (Cormode and Muthukrishnan 2003) do not provide information about frequencies or relative order. Multistage filters were proposed as an alternative in (Estan and Varghese 2003) but present the same problems. Other interesting algorithms that can be used on update streams, not supporting insertion and delete operations, were presented in (Ganguly 2003), allowing top-k element identification with their respective frequencies with a specified probability.

Overviews, comparative summaries and experimental evaluation of some of the described algorithms were presented in (Manerikar and Palpanas 2009) and (Cormode and Hadjieleftheriou 2010).

2.3 Top-k mixed techniques

The Filtered Space-Saving (FSS) algorithm was initially presented by the authors in (Homem and Carvalho 2010a) and extended in (Homem and Carvalho 2010b), and merges the two distinct approaches for top-k algorithms. It improves Space-Saving (Metwally et al. 2005) quite significantly by narrowing down the number of required counters, update operations and the error associated with the frequency estimate.

FSS uses a hashed bitmap counter with h cells to filter and minimize updates on the monitored elements list and 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 elements in the list needed to ensure high quality top-k answers. 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 depends 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. The bitmap counter cells hold α_i, the maximum error associated with elements with hash i. Figure 1 presents a block diagram of the FSS algorithm with the two storage elements, bitmap counter with h cells and the monitored list with m elements.

When a new value is received, the hash is calculated and the bitmap counter is verified. If there are already monitored elements with that same hash, the 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; otherwise, the element is checked to see if it should be added.

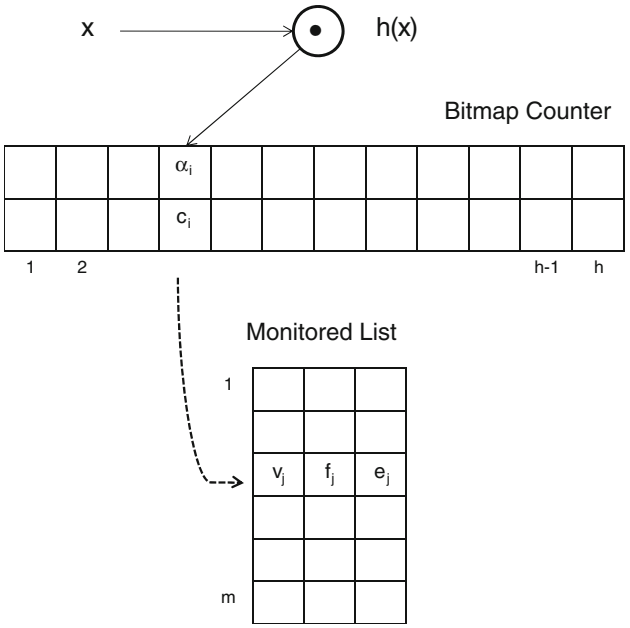

Fig. 1 FSS algorithm diagram

The maximum estimated error of the algorithm, μ, is the minimum of the estimate counts in the list. While there are free elements in the list, μ is set to 0. A new element will be inserted into the list if $\alpha_i + 1 \geq \mu$. The new element is included in the monitored with an estimate of $\alpha_i + 1$.

If the observed element is not to be included in the list, then α_i is incremented. In fact the value α_i stands for the number of elements with hash value i that have not been counted in the monitored list. It is essentially the maximum number of times that an element with this hash value could have been observed while not in the list.

If the list has exceeded its maximum allowed size, then the element with the lower estimate is removed. When an element is removed, the corresponding bitmap counter cell is updated and the error for the elements with hash i becomes equal to the estimate of the removed value, $\alpha_i = f_j$. When $h = 1$ FSS is exactly the Space-Saving algorithm.

2.4 Sliding windows algorithms

Solving the top-k problem over a sliding window is, however, more complex and has been less studied. Unrestricted top-k algorithms do not have obvious extensions to the sliding window model. The critical problem is that expired elements must be discounted, meaning that some

information has to be stored about what has been received during the entire window.

The problem of maintaining statistics on data streams over a sliding window has been presented by Datar et al. (2002). This seminal paper presented memory optimal algorithms for maintaining statistics such as count, sum of positive integers, average, etc. Babcock et al. (2003) extend their work in with algorithms for variance and k-medians. Algorithms that provide approximate histograms are a basic building block for many of the top-k or frequent elements algorithms for sliding windows.

Datar et al. (2002) presented the exponential histogram (EH). Their approach toward solving the counting problem in a sliding window is to maintain a histogram that records the timestamp of selected 1's that are active, i.e., within the sliding window. Timestamps are maintained in buckets each representing a given number of 1's. If the timestamp of the last bucket expires, the bucket is deleted. When a new data element arrives with value is 0, it is ignored; otherwise, a new bucket with size 1 and the current timestamp is created. The list of buckets is then traversed in order of increasing sizes. If there are $k/2 + 2$ buckets of the same size, the oldest two of these buckets are merged into a single one of double the size. The EH algorithm maintains a data structure which can give an estimate for the counting in sliding window problem with relative error at most $1/k$ using $(k/2 + 1)(\log(2N/k + 1) + 1)$ buckets.

A distinct approach using time decaying aggregates is proposed by (Cohen and Strauss 2006). The problem of maintaining time decayed aggregates is addressed, for general decay functions, including exponential decay, polynomial decay and sliding window. An algorithm for maintaining polynomial decay approximately with storage of $O(\log(N) \log \log(N))$ bits is given. It shows that polynomial decay can be tracked nearly as efficiently as exponential decay.

Exponential histograms (EH) have been expanded by Qiao et al. (2003) to provide approximate answers to sliding window statistics in multivariate streams. A two-dimensional histogram is proposed to support sliding window queries for multiple elements using exponential histograms as the basic building block. To further compress the exponential histograms, a condensed exponential histogram is proposed, maintaining the error bound. The algorithm partitions both time and data values and guarantees a bound on the estimation error.

In (Zhu and Shasha 2002) the concept of Basic Windows were introduced to handle windowed aggregates. The sliding window is divided into Basic Windows of equal duration. Each Basic Window stores a synopsis and the corresponding timestamp. The timestamp is used to expire the Basic Window and replace it by a new one. Results are refreshed after the current Basic Window is processed. The approach to sliding windows used in this work is similar. In (Zhu and Shasha 2002) the synopsis is created using Discrete Fourier Transform and allows multiple stream statistics such as average and standard deviation to be monitored.

Several algorithms have been proposed for the frequent elements problem in sliding windows, the problem of finding elements whose frequency within the sliding window exceeds a given threshold. Golab et al. (2003) proposed a deterministic algorithm for identifying frequent items in sliding windows. It follows the approach of (Zhu and Shasha 2002), to divide the sliding window into sub-windows, store only a synopsis of each sub-window, and re-evaluate the query when the most recent sub-window is full. It identifies items occurring with a frequency that exceeds a given threshold. The main issue with this algorithm is that it requires that item frequencies can be counted exactly within a single Basic Window. Once the current Basic Window is closed, the top-k elements and respective frequencies are stored in a synopsis. The sum of the frequencies of the last item of all the synopsis is the upper limit on the frequency of an element that does not appear on any of the top-k lists. The algorithm reports all items whose sum of frequencies exceeds the upper limit. Another possible issue is that if k is small, then the upper limit may be very large and the algorithm will not report any frequent items.

Lee and Ting (2006) proposed and algorithm based on Frequent (Demaine et al. 2002) but extending it to the sliding windows model. The algorithm is obtained by replacing all the counters by sliding window counters. These sliding window counters are based on a λ-snapshot, for estimating the number of 1-bits in a bit stream over the sliding window. The λ-snapshot just samples every other λ bits in the stream. The λ-snapshot is implemented as a dequeue and the sliding window is supported by a simple shift operation. The decrement operation used in Frequent is replaced by clearing the last 1 bit in the λ-snapshot. The scheme is extended to variable length windows in order to support time sliding windows. As in Frequent (Demaine et al. 2002), this algorithm does not provide frequencies estimates for each element and can not be used to answer the top-k problem.

The concept of recent frequent elements is presented in (Tantono et al. 2008). The algorithm considers the N last observed elements and does not include any extension to the time sliding window. The algorithm maintains a $M \times H$ sketch, where M and H are defined based on the specific problem, and uses a structure in each cell to store an approximate summary of recent elements count. H hash functions are used to update and retrieve a set of structures for each element. To estimate the frequency of an element, the algorithm uses the minimum value of those obtained

from the retrieved structures. Although the authors present results for the top-k problem, it seems very limited (values of k are lower than 10) and it may require exhaustive search as no identifiers are kept stored for the elements.

2.5 Related data mining techniques

There has been a huge amount of work in the data mining area 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 (Tanbeer et al. 2009a) and (Hu et al. 2008). Frequent itemsets extraction in a sliding window is proposed in (Tanbeer et al. 2009b) and (Hua-Fu Li and Suh-Yin Lee 2009). The problem of extracting itemsets is slightly different, and the proposed solutions do not solve the top-k problem in sliding windows.

3 The filtered space-saving with partitions (FSP)

An initial and obvious approach to extend FSS to handle sliding windows, with the sliding window approximated by p basic time sub-windows of fixed duration, is to use FSS to identify the most frequent elements in each sub-window, and then merge the p result sets to have the answer to the full sliding window. This approach, with a few improvements, will be designated as Filtered Space-Saving with Partitions (FSP).

Updates will only occur in the last (and only active) sub-window in the monitored list. Only that sub-window will require the use of the FSS bitmap counter filter. All other $p - 1$ sub-windows or partitions need only to store the list of elements, which will be referred as a partition list.

To provide an error estimate, each element in the partition list entries should have 3 values: the element itself v_j; the estimate count f_j; the associated error e_j.

Results can be obtained by merging the partition lists with the active monitored list. This merge operation can be avoided if a full list is kept at all times. Whenever an update occurs in the monitored list, the full list should also be updated.

Elements that expire at the end of validity of each partition will be removed from the full list. Figure 2 presents the pseudo-code for the FSP algorithm.

FSP requires additional space and increases the number of update operations. It requires p partition lists with m elements each, one full list with up to mp elements and a bitmap counter with h cells. Whenever an element in the monitored list of the active sub-window is observed, two lists have to be updated. FSP also requires a batch update at each partition expiry to update the full list with the values of the expired list.

FSP does not maintain the properties of FSS (Homem and Carvalho 2010b) for the full sliding window. These are only maintained within each sub-window.

4 The filtered space-saving with sliding window algorithm (FSW)

The filtered space saving with sliding windows (FSW) is a further improvement over FSP. In fact it is a novel algorithm that builds on the authors proposed Filtered Space Saving (FSS) algorithm (Homem and Carvalho 2010b) and adds a time sliding window filter capability to it.

FSW extends FSS by adding a temporal histogram to each of the list elements and to the bitmap counter cells. The sliding window will be approximated by considering p basic time sub-windows of equal duration. The temporal histogram will consist of p counters or buckets, one for each sub-window, containing the count of the observed elements during that sub-window. Observations will always be accounted in the last and active bucket. When a sub-window expires the corresponding bucket is removed.

Whenever a sub-window expires, all element counts have to be updated. The events observed in the expired window will no longer contribute to the results. The bitmap counter will also be updated in the same manner. This avoids the duplicate list update and all the partition lists of FSP at the expense of some histogram updates.

When an element is inserted in the monitored list, the histogram from the corresponding bitmap counter is copied; when an element in the monitored list is observed, its counter and histogram are updated. If the observed element is not in the monitored list, then the corresponding bitmap counter and histogram are updated.

When an element is removed, the histogram associated with the corresponding bitmap counter has to be merged with the element histogram in order to minimize the loss of information. This is performed by setting each bucket i to the maximum of the two histograms buckets. The bitmap counter is then updated to the sum of the buckets of the histograms. Unlike in FSS, the resulting value of the bitmap counter can be higher than the count of the removed element. By keeping the timestamp of element insertion, the number of buckets in the histogram that needs to be merged can be determined. Buckets referring to periods before insertion do not need to be merged.

The histogram representation must have some properties:

- Fast update of the active bucket (the bucket that will be updated with arriving elements);

Fig. 2 FSP algorithm

```
Algorithm: FSP (h cells, m counters, p periods, S stream)

// update function
begin
        for each element, x, in S {
                update FSS for the active sub-window s

                if the monitored list in the FSS is updated {
                        if an element was inserted {
                                if v_j is not in the full list F {
                                        insert v_j in the full list F as v_f
                                        set e_f to α_i
                                        set f_f to α_i+1
                                } else {
                                        add α_i to e_f
                                        add α_i+1 to f_f
                                }
                        } else {
                                update v_f in the full list F {
                                        increment f_f
                                }
                        }

                        if an element was removed from the list {
                                subtract e_j to e_f
                                subtract f_j to f_f
                                if f_f = 0 {
                                        remove v_f from full list F
                                }
                        }
                }
        }// end for
end


// expire function
begin
        for each element, v_j, in expired partition list L_p-1 {
                 let v_f be the element v_j in the full list F {
                                subtract e_j to e_f
                                subtract f_j to f_f
                                if f_f = 0 {
                                        remove v_f from full list F
                                }
                }
        end for

        drop L_p-1
        rotate lists L
        store the monitored list of FSS as L_1
        reinitialize FSS for the active sub-window s
end
```

- Fast expiry, i.e. removal of the expired bucket and total value update;
- Fast merger of histograms, ensuring that the maximum value for each bucket of the two initial histograms is kept in the resulting histogram;
- Allow incremental merging of the histogram, as in many cases, element removal occurs soon after insertion and only a few buckets need to be updated.

Furthermore, to ensure the strong guarantees presented in Section 5, the following conditions are required:

- Sub-windows boundaries, i.e. bucket boundaries, must be the same for all elements;
- Exact counts for each sub-window are required. Histograms can not be approximated.

An obvious choice for the histogram representation is the equal-width histogram, where each bucket represents a sub-window. In fact, sub-window duration does not need to be the same as long as all histograms share the same boundaries. In such situation the properties of the algorithm remain unchanged. This representation is not optimal in

terms of memory usage but allows for a very simple and fast implementation, especially if circular arrays are used.

Figure 3 presents a block diagram of the FSW algorithm. FSW uses two storage elements. The first is 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 and an histogram E_i. The histogram E_i has p buckets (E_{it} stands for bucket t of bitmap counter i), being E_{i1} the oldest bucket and E_{ip} the newest and active bucket. 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, c_i and E_{it} 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 4 values and a histogram: the element itself v_j, the estimate count f_j, the associated error e_j, the insertion timestamp t_j and the histogram C_j. The histogram C_i has also p buckets (C_{jt} stands for bucket t of monitored list element j) with C_{i1} as the oldest bucket and C_{ip} the newest and active bucket.

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 μ is set to 0.

When a new element 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 and C_{jp} are incremented.

A new element will only be inserted into the list if $\alpha_i + 1 \geq \mu$. If the element is not to be included in the monitored list, then α_i and E_{ip} are 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 in the sliding window.

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 removal. 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.

When the selected element is removed from the list, the corresponding bitmap counter cell is updated, c_j is decreased, the histograms merged by setting $E_{jt} = \max\{C_{jt}, E_{jt}\}$, for $t = 1 \ldots p$, $\alpha_i = \sum_t E_{jt}$ with the maximum possible error incurred for that position.

The new element is included in the monitored list ordered by decreasing f_j and increasing e_j, c_i is incremented, $f_j = \alpha_i + 1$, $e_j = \alpha_i$, $t_j = t_0$, and $C_j = E_i$ (t_0 stands for the moment of insertion).

FSW requires sub-window expiry to be processed in a batch manner. At the moment of expiry, both the monitored list and the bitmap counter need to be scanned and updated.

For every element in the monitored list, f_j and e_j have to be decremented by C_{j1} (e_j minimum is 0), C_j rotated (assuming a circular array implementation) and C_{jp} set to *0*. If all observations of an element expire, $f_j = 0$, the element is removed from the monitored list. After all elements are updated, the monitored list must be ordered by decreasing f_j and increasing e_j.

Every element, E_i, in the bitmap counter has to be rotated (assuming a circular array implementation) and E_{ip} set to 0. The maximum possible error for cell i, is set to $\alpha_i = \sum_t E_{jt}$.

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. Figure 4 presents the pseudo-code for the FSW algorithm.

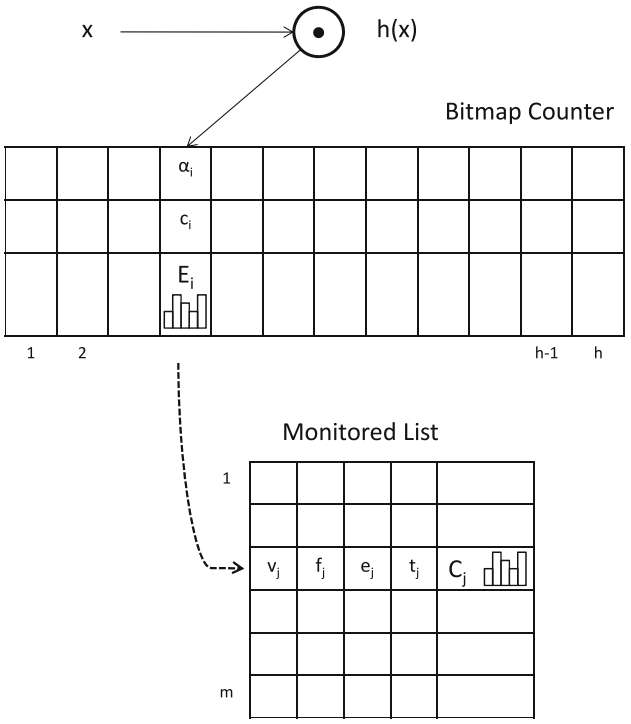

Fig. 3 FSW algorithm diagram

5 Properties of FSW

Unfortunately, FSW does not maintain all the properties of FSS regarding μ or the guarantee of a maximum error of the estimate as presented in (Homem and Carvalho 2010b).

Fig. 4 FSW algorithm

```
Algorithm: FSS (h cells, m counters, p periods, S stream)

// update function
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
                                increment counter C_jp
                                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(v_n) mod h
                                decrement c_k
                                remove v_n
                                for all t = 1..p
                                        set E_kt to max {E_kt, C_nt}
                                end for
                                set α_k  to ∑_t E_kt
                        }
                        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,
                        set t_j to T_0 and C_j to E_i
                        increment counter c_i

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

// expire function, assuming a circular array representation for histograms C and E
begin
        for each element, v_j, in list {
                set f_j to f_j-C_j1
                set e_j to max{e_j-C_j1, 0}
                rotate C_j
                set C_jt to 0
                if f_j = 0 {
                        remove v_j
                        let k be the hash(v_j) mod h
                        decrement c_k
                }
        end for

        order list by descending f_j and for same f_j with ascending e_j

        for each counter in bitmap {
                rotate C_j
                set E_jt to 0
                set α_k  to ∑_t E_kt
        end for
end
```

It is easy to construct an example where old elements with the same expiry sub-window completely fill the monitored list and avoid the inclusion of new elements during the full window. The list will not contain any elements when the old elements expire, therefore the error is not bound by a fraction of N (the total number of events in the data stream).

Some of the interesting stochastic properties of FSS are also kept in FSW. It can be proven that μ is limited by the distribution of hits in the bitmap counter. FSW keeps the property of FSS that the maximum error depends on the ratio between h and the number of distinct values D in the data stream (Homem and Carvalho 2010b).

The use of $hash(x)$ as a proxy to the element x for filtering and error estimation purposes, has the obvious disadvantage of generating collisions. These collisions affect the frequency estimation and introduce error; in order to minimize error, the hash function must map the keys to the hash values as evenly as possible. The hash values should be uniformly distributed. A good randomizing or a

cryptographic function would be a good choice for the hash function (although performance should also be taken into consideration). One will consider that the hash function $hash(x)$ distributes x uniformly over the h counters.

Theorems 1 to 4 present strong guarantees similar to FSS algorithm. Theorem 5 presents stochastic bounds that limit the error depending on the total number of distinct elements D and the number of cells h in the bitmap counter.

Please note that, throughout the text, the use of the i and j subscripts when referring to single element assumes that an element with hash i may be included in the monitored list in the position j.

Lemma 1. *The value of μ and of each α_i and E_{iw} increases monotonically over time during the sub-window w. For every t, $t_0 \in]t_{w-1}, t_w]$, with t_{w-1} being the time the sub-window has started, and t_w the time it expires*:

$$t \geq t_0 \Rightarrow \alpha_i(t) \geq \alpha_i(t_0) \quad (1)$$

$$t \geq t_0 \Rightarrow \mu(t) \geq \mu(t_0) \quad (2)$$

$$t \geq t_0 \Rightarrow E_{iw}(t) \geq E_{iw}(t_0) \quad (3)$$

Proof. As α_i and μ are integers:

By construction, α_i and E_{iw} are incremented when a non-monitored list element is observed between expiries. α_i and E_{iw} maintain the values if the element is monitored or inserted in the list. α_i and E_{iw} get updated when an element is removed from the list.

In case of a removal, considering t_- and t_+ as the instants before and after the removal:

$$E_{it}(t_+) = \max\{E_{it}(t_-), C_{jt}(t_-)\} \geq E_{it}(t_-)$$

So:

$$\alpha_i(t_+) = \sum_t E_{jt}(t_+) \geq \sum_t E_{jt}(t_-) = \alpha_i(t_-)$$

Therefore, at all times between expiries:

$$t \geq t_0 \Rightarrow \mu(t) \geq \mu(t_0)$$

$$t \geq t_0 \Rightarrow \alpha_i(t) \geq \alpha_i(t_0)$$

$$t \geq t_0 \Rightarrow E_{iw}(t) \geq E_{iw}(t_0)$$

End of Proof.

Lemma 2. *The value of α_i is always greater than or equal to the maximum number of observations F_x of any element x with hash i that is not in the monitored list.*

Proof. An element that is not in the monitored list has either never been there during the entire sliding window or has been removed.

By construction any observation of an element with hash i is always counted either in the monitored list or in the E_i histogram.

Consider an element that is never inserted in the list. In this case, all observations counted in E_i histogram, in each sub-window E_{iw}, will be at least equal to the observations of element x in that sub-window, F_{xt}. Therefore:

$$\alpha_i = \sum_w E_{jw} \geq \sum_t F_{xw} = F_x$$

By construction, an element in the monitored list maintains the exact count of observations since it was inserted in the list, $f_j - e_j$. The value of e_j is the equal to the value of α_i at the instant of insertion minus all expired observations.

At the time of removal from the list:

$$\alpha_i = \sum_w \max\{E_{iw}, C_{jw}\}$$

For every sub-window before insertion in the monitored list, $t_0 > t_w$:

$$C_{jw} = E_{iw} \geq F_{xw}$$

For the sub-window where insertion in the monitored list occurred, $t_{w-1} > t_0 \geq t_w$:

$$C_{jw} = C_{jw}(t_j) + F_x(t_w) - Fx(t_j) \geq F_x(t_w) = F_{xw}$$

For every sub-window where the element was already in monitored list, $t_w > t_0$:

$$C_{jw} = F_{xw};$$

Therefore:

$$\alpha_i = \sum_w \max\{E_{iw}, C_{jw}\} \geq \sum_w \max\{E_{iw}, F_{xw}\} \geq \sum_w F_{xw} = F_x$$

So at all times:

$$\alpha_i \geq F_x$$

End of Proof.

Theorem 1. *An element with a total number of observations F_{xw} observed in sub-window w must be in the list if F_{xw} is greater than μ_w, the value of μ at t_w.*

Proof. From Lemma 1 is known that within sub-window w, μ increases monotonically; therefore it is at its maximum at the end of the sub-window period t_w. During the entire sub-window, μ is less than or equal to μ_w.

From Lemma 2 is known that if an element is not in the monitored list, $\alpha_i \geq F_x$.

Therefore, if the F_{xw}-th observation occurs for an element in sub-window w, that element is either already in the monitored list or is inserted, as by construction any element with $\alpha_i(t_-) + 1 > \mu$ is inserted in the monitored list. Considering t_- and t_+ as the instant before and the instant after:

$$\alpha_i(t_-) \geq F_x(t_+) - 1 \Rightarrow \alpha_i(t_-) + 1 \geq F_x(t_+) = F_{xw} > \mu_w \geq \mu$$

End of Proof.

Lemma 3. *The estimated count f_j of an element in the monitored list is at least equal to its total number of observations F_x.*

Proof. By construction the algorithm keeps an exact count of the observations while the element is in the monitored list. From Lemma 2 is known that before insertion, at moment t_-;

$$\alpha_i(t_-) \geq F_x(t_-)$$

$$f_j(t_+) = e_j(t_+) + 1 = \alpha_i(t_-) + 1 \geq F_x(t_-) + 1 = F_x(t_+)$$

Therefore, for a monitored element one has at all times:

$$f_j \geq F_x$$

End of Proof.

Theorem 2. *An element is kept in the monitored list as long as its total number of observations $F_w > \mu_w$ for each sub-window w since it was observed (inclusive).*

Proof. From Theorem 1 it is known that the element was in the monitored list when it was last observed with $F_w > \mu_w$.

By construction removals occur only for elements with $f_j = \mu$.

From Lemma 2 is known that if an element is not in the monitored list, then $\alpha_i \geq F_x$.

Therefore if the F_w observation occurs for an element in sub-window w, that element is either already in the monitored list or is inserted, as by construction, any element with $\alpha_i(t_-) + 1 > \mu$ is inserted in the monitored list:

$$\alpha_i(t_-) \geq F_x(t_+) - 1 \Rightarrow \alpha_i(t_-) + 1 \geq F_x(t_+) > \mu_w \geq \mu$$

End of Proof.

Lemma 4. *The value of e_j in an element x with hash i included in the monitored list is always lower or equal to α_i.*

Proof. By construction e_i is set at the instant of insertion of the element in the list to the value of α_i. At that instant E_j and C_j are equal.

$$e_i(t_j) = \alpha_i(t_j) = \sum_w C_{jw}(t_j)$$

Whenever an expiry happens, the value C_{jk} of the expired sub-window k is decreased from both e_i and α_i, keeping the difference between the two variables and $\alpha_i \geq e_i$. If $e_i - C_{jk} < 0$ then e_i is set to 0, keeping $\alpha_i \geq e_i$.

End of Proof.

Lemma 5. *Let N_i be the total number of observations for hash i. Let A be the monitored list, A_i the set of all elements in A such that hash(v_j) = i, with j being the position of element in the monitored list. For all elements in A_i:*

$$N_i \geq F_j,\ v_j \text{ in } Ai \tag{4}$$

$$N_i \geq \alpha_i + \sum_{vj \text{ in } Ai} (f_j - e_j) \tag{5}$$

Proof. By definition N_i is the sum of the observation of all elements with hash i so it is always greater or equal to one of the parts.

$$N_i = \sum_{vj \text{ in } Ai} F_j$$

Whenever an element with hash i is observed, it is either in the monitored list and is counted in $f_j - e_j$, or it is not in the list and is counted in α_i. When an element with hash i is inserted in the list, $f_j - e_j = 1$, so that is the only accounted observation. When an element with hash i is removed from the list, considering t_1 as the moment before the replacement, t_2 the moment after the replacement, and t_0 the moment when the removed element was inserted in the list:

$$\alpha_i(t_2) = f_j(t_1) = f_j(t_1) - e_j + e_j = f_j - e_j + \alpha_i(t_0) \leq f_j - e_j + \alpha_i(t_1)$$

The resulting value for α_i is lower than the sum of the two previous values. Therefore,

$N_i \geq \alpha_i + \sum_{vj \text{ in } Ai} (f_j - e_j)$ in all situations.

End of Proof.

Theorem 3. *In FSW, for any cell i in the bitmap counter that has monitored elements:*

$$\mu \leq N_i$$

$$\mu \leq \min\{N_i\} = N_{\min} \tag{6}$$

$$E(\mu) \leq E(N_{\min}). \tag{7}$$

where N is the total number of elements observed during the full sliding window.

Proof. The total number of elements N in stream S can be rewritten as the sum of N_i, the total of elements with hash i.

$$N = \sum_i N_i$$

Each element contributes only to one counter: either it is counted in α_i or it is counted in a monitored element observations $f_j - e_j$. 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 $hash(v_j) = i$:

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

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

$$N_i \geq \alpha_i - c_i\alpha_i + \sum_{j\,in\,Ai} f_j \geq \alpha_i - c_i\alpha_i + c_i\mu$$

$$c_i\mu \leq N_i + (c_i - 1)\alpha_i \leq c_i N_i$$

For all $c_i > 0$:

$$\mu \leq N_i$$

This means that our 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:

$$\mu \leq \min\{N_i\} = N_{\min}$$

It is therefore trivial to conclude:

$$E(\mu) \leq E(N_{\min})$$

End of Proof.

Lemma 6 *In FSS and FSW, for any cell i in the bitmap counter that has monitored elements, let r be the rank of N_i when ordering all N_i in descending order, N^r the r-th element in that ordering and k the number distinct values of i with elements in the monitored list:*

$$N_i \leq N^r \tag{8}$$

$$\mu \leq \min\{N_i\} \leq N^k \tag{9}$$

$$E(\mu) \leq E(N^k) \tag{10}$$

where N^k is k-th ranked element in the full set of bitmap counters in descending order.

Proof. The set of bitmap counters that has at least one monitored element is a subset of the full list of bitmap counters. If the two sets are ordered in descending order, then it is trivial to see that the r-th element in the larger set must be equal or greater than the r-th element in the contained set.

By construction, c_i is number of elements x in the monitored list with $hash(x) = i$. Let k be the number of distinct values of i with elements in the monitored list:

$$m = \sum_{i:\,Ai\,in\,A} c_i = k + \sum_{i:\,Ai\,in\,A} (c_i - 1)$$

The min $\{N_i\}$ is the last element in the set of bitmap counters that has at least one monitored element. This list has exactly k elements, therefore the min $\{N_i\}$ must be equal or lower than the k-th element in the full set N^k.

End of Proof.

Theorem 4. *In FSS and FSW, μ is bound by the number of observations in N^τ, the τ-th element in the ordering of the full set of N_i, with τ being the minimum value that allows for:*

$$\#(U_{j=1..r} Aj) \geq m \tag{11}$$

where #(.) is the number of distinct elements in the set, A^j is the set of elements associated with N^j and m the size of the monitored list.

In that situation:

$$\mu \leq N^\tau \tag{12}$$

$$E(\mu) \leq E(N^\tau) \tag{13}$$

Proof. Let k be the number distinct values of i with elements in the monitored list. From Lemma 6:

$$\mu \leq N^k$$

Assuming the monitored list has m values (otherwise $\mu = 0$), with A^j being the set of elements associated with N^j then for k its trivial to have:

$$\#(U_{j=1..k} Aj) \geq m$$

The conditions are met for $\tau = k$. As τ is chosen to be the minimum value, τ is at most k.

$$\tau \leq k \Rightarrow N^\tau \geq N^k \geq \mu$$

End of Proof.

Theorem 4 justifies the use of a larger bitmap counter to minimize collisions, as less collisions lead to a higher value of τ and therefore to a lower bound for μ.

Theorem 5. *In FSS and FSW, assuming a hash function with a pseudo-random uniform distribution, the expected value of the number of observations in N^τ, the τ-th element in the ordering of the full set of N_i, depends on the number of distinct elements D in stream S and the number of elements h in bitmap counter.*

$$E(\tau) \leq m/\left[D(\Gamma(D+1,\, D/h)/\Gamma(D+1))^h. - \sum_{D \geq i \geq 1} (\Gamma(i,\, D/h)/\Gamma(i))^h\right] \tag{14}$$

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

Proof. Consider D_i as the number of distinct elements in each bitmap counter i. To estimate the D_i values in each cell, consider the Poisson approximation to the binomial distribution of the variables as in (Bertsekas 1995). One will assume that the events counted are received during the period of time T with a Poisson distribution. So:

$$\lambda = D/T$$

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

$$\lambda = D$$

This replaces the exact knowledge of D by an approximation as $E(x) = D$, 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 elements 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:

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

And the cumulative distribution function is:

$$P(D_i \leq k|t = 1) = \Gamma(k + 1, D/h)/\Gamma(k + 1)$$

To estimate $E(D_{\max})$ let us first consider the probability of having at least one D_i larger than k:

$$P(D_{\max} \leq k|t = 1) = P(D_i \leq k \text{ for all } i\,|t = 1)$$

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

$$\begin{aligned} P(D_{\max} \leq k|t = 1) &= \prod P(D_i \leq k|t = T) \\ &= \prod \Gamma(k + 1, D/h)/\Gamma(k + 1) \\ &\Leftrightarrow \\ P(D_{\max} \leq k|t = 1) &= [\,\Gamma(k + 1, D/h)/\Gamma(k + 1)]^h \\ &= g(k + 1), \end{aligned}$$

and since k is integer:

$$\begin{aligned} P(D_{max} = k|t = 1) &= P(D_{max} \leq k|t = 1) \\ &\quad - P(D_{\max} \leq k - 1|t = 1) \\ &\Leftrightarrow \\ P(D_{\max} = k|t = 1) &= (\Gamma(k + 1, D/h)/\Gamma(k + 1))^h \\ &\quad - (\Gamma(k, D/h)/\Gamma(k))^h \\ &\Leftrightarrow \\ P(D_{\max} = k|t = 1) &= \left[(\Gamma(k+1, D/h)/k)^h - (\Gamma(k, D/h))^h\right] \\ &\quad /\Gamma(k)^h \\ &\Leftrightarrow \end{aligned}$$

$$\begin{aligned} E(D_{max}) &= \sum_i i \left[(\Gamma(i + 1, D/h)/\Gamma(i + 1))^h \right. \\ &\quad \left. -(\Gamma(i, D/h)/\Gamma(i))^h\right] \\ &\Leftrightarrow \\ E(D_{max}) &= \sum_i i[g(i + 1) - g(i)] \\ &= [g(2) - g(1)] + 2[g(3) - g(2)] + \ldots \\ &\quad + D[g(D + 1) - g(D)] \\ &\Leftrightarrow \\ E(D_{max}) &= Dg(D + 1) - \sum_{D \geq i \geq 1} g(i) \\ &= D(\Gamma(D + 1, D/h)/\Gamma(D + 1))^h \\ &\quad - \sum_{N \geq i \geq 1} \Gamma(i, D/h)/\Gamma(i)^h \end{aligned}$$

As τ will greater than m/D_{max}:

$$E(\tau) \leq m/E(D_{max})$$

End of Proof.

Figure 5 illustrates the behavior of D_i and $D_{\max}$ probability mass function using the D_i as the x axis. For $D = 100{,}000$ and $h = 10{,}000$:

The expected value of $E(D_{\max}) = 24.28$. The intuitive justification for using the error filter is that by using more cells in the filter than in the monitored list one can control and minimize the collisions (since each entry in the monitored list requires space equivalent to at least 3 counters). This in turn will increase τ, lower the N^r value and further bound the maximum error μ.

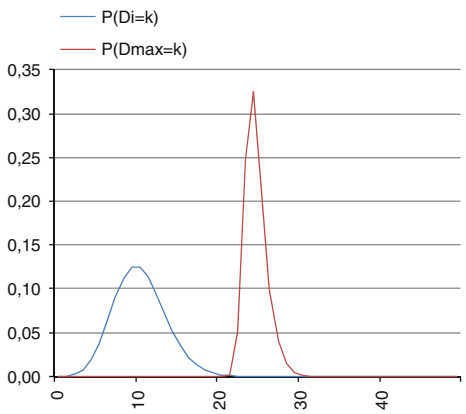

Fig. 5 Probability mass function for D_i and $D_{\max}$, $D = 100{,}000$ and $h = 10{,}000$

6 The improved filtered space-saving with sliding window algorithm (FSWr)

FSW uses a bitmap counter to hold the α_i counters and E_i histograms. The same number of α_i counters and E_i histograms are used. However, this is not required, and more counters than histograms could be used. This will allow more detail to be kept during the active sub-window without all the space that would be required if the same number of histograms were used. In fact this provides a good trade-off between space and performance without breaking any of the FSW properties presented in Section 5.

The idea is to set a ratio r between α_k counters and E_i histograms. The $hash(x)\ mod\ hr = k$ of an element x is now used to access the more detailed bitmap counter α_k, while the histogram E_j is accessed by dividing k by the ratio r (integer division div). All the properties of FSW hold as long as the active bucket E_{jp} is kept with the maximum of all α_k such that $j = i\ div\ r$.

For the remaining of this paper, FSW will designate both FSW and FSWr. In fact FSW can be seen as FSWr with $r = 1$.

7 Non-unique filtering in FSW algorithm (FSWf)

When used in data streams with huge number of distinct values, the inclusion of a non-unique filter is an interesting improvement to FSWr. The idea is to always count any element that is already in the monitored list, but to filter elements before updating the bitmap counter. Elements that one knows that were not previously seen during the sub-window should not update the bitmap counter. This can easily be implemented using a bit array of f bits and $hash(x)\ mod\ f$ as the index to that array. If the bit is clear, then the element was not observed during that sub-window.

The bit is set when receiving a new element. If a bit was previously set, then the element might have previously been observed and should update the bitmap counter. As this might be the second observation, if E_{jw} is still zero, then the bitmap counter should be increased by 2. This ensures that if an element occurs twice during the sub-window it is properly counted. If it is observed only once (and no collisions occur) it is ignored. At the start of a new sub-window the bit array is cleared.

The maximum error this filtering introduces is of 1 per each valid sub-window before the element was inserted in the monitored list. No error is added while the element is being monitored. An additional parameter f is required and should be set high enough to keep a low collision rate during a sub-window period.

8 Implementation

The practical implementation of FSW follows the one of FSS (Homem and Carvalho 2010b) with the addition of histograms to each element in the monitored list and to the bitmap counter. It may include the Stream Summary data structure presented in (Metwally et al. 2005).

FSW needs a bitmap counter to hold the α_i counters and E_i histograms. 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 does not find an element, then the bit is cleared. This reduces memory at the expense of a potential additional search in the monitored list. 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.

Histograms can be implemented using a circular array of counters. A global index is kept to the active bucket. During merge operations, t_j can be used to determine if a full merge as to be performed, or if only a partial merge is required as the element may have been very recently inserted in the monitored list. Only the active bucket and the buckets that may have been updated since insertion need to be merged. Most of times the removals occur for recently inserted elements, therefore avoiding unnecessary merge operations can improve significantly the performance.

One obvious improvement on the basic implementation of the histograms is the use of compression algorithms for histogram representation. In fact, the values contained in each of the histogram bucket do not require a very large counter in most cases; the use of lower size counters, distinct counters sizes or any lossless data compression techniques such as Huffman encoding can reduce the space required per histogram. Lazy histogram creation for the histograms in the monitored lists can also be used, as in most cases the elements are inserted and deleted from the list in the same sub-window or within a few sub-windows. The space required might only be allocated after the first sub-window expiry or after a few sub-windows, with values being kept in a transient structure.

Another possible improvement is the use of non-equal width buckets for the histograms. If the distribution of elements over time is known a priori, then the sub-window durations can be adapted to distribute weights through the buckets. More balanced buckets may lower the error introduced by sub-window expiry updates. Changing the duration of the sub-windows does not change any of the properties of FSW (duration is not relevant in the proofs).

The implementation of FSW using approximate histograms is also a possibility, even if many of the properties of

FSW do not hold, as they require an exact representation of sub-window counts. However, preliminary results on real life data show that performance may still be adequate for many situations, and that the trade-off between space saved by the approximate representation of the histograms and the increase in monitored list or bitmap counter size may be a good option.

Approximate histograms over a sliding window has generated many interesting developments: wavelets-based histograms were presented in (Matias et al. 1998); exponential histograms were presented in (Datar et al. 2002); smooth histograms that extends the class of functions that can be approximated on a sliding window were presented in (Braverman and Ostrovsky 2007); waves, a further improvement to exponential histograms with sum and count, was presented in (Gibbons and Tirthapura 2002) with time and memory optimal algorithms. A fast incremental algorithm for maintaining approximate algorithms was presented in (Gilbert et al. 2002) and an algorithm for updating wavelets-based histograms in (Matias et al. 2000).

The main issue when using approximate histograms to implement FSW is that some specific update operations need to be supported.

Exponential histograms, smooth histograms or waves are not an option as they do not provide a merge operation that returns the max count between two histograms. Wavelets-based histograms allow for such a merge to be performed although the values to be merged have to be decoded, merged and then encoded. Wavelets-based histograms can also be used as circular arrays, the update operation requiring only the difference between values to be propagated as shown in (Matias et al. 2000). The use of a small cache to hold the most recent sub-windows might solve the performance problems as most merge operations require the merge of only a few buckets.

The final issue to be considered when implementing these algorithms is the setting of parameters. A few guidelines will be discussed, although at this stage no strong rules are available to ensure optimal performance. Clearly, performance has to be tuned for a specific application by testing distinct parameterizations using training data. One will consider k will be a given value that depends only on the specific problem being solved using these algorithms.

The number of sub-windows p within the required sub-window requires weighting two factors; increase in p will give a smother sliding window approximation, but will also require additional space and operations. If the problem does not suggests a natural value for p then the required time precision should drive the setting of the parameter.

Setting the length of the monitored list m depends mainly on k and the number of distinct elements D in the data stream. Clearly m should always be at least twice the value of k to ensure reasonable results, but when applied to a data stream with very high dispersion, m has to be increased. One can expect FSW to behave in a similar manner to FSS; in (Homem and Carvalho 2010b) good results were achieved in finding the top-100 elements in a data stream with more than 100,000 distinct elements using $m = 2{,}000$.

The parameter h will determine the number of histograms to maintain in the bitmap counter. Good results were obtained when h is set between 3 and 6 times m. With h less than 3 times m the filtering effect by the bitmap counter is less significant. Setting a value of h too high may not increase the results significantly and will have a high impact in terms of space use.

Parameter r controls the ratio between entries in the hash counters used in the active sub-window and the number of histograms in the bitmap counter. In FSWr the parameter r can be used to increase the filtering effect of the bitmap counter without all the increase in space allocation that would be required by setting an equivalent value for h. Values between 4 and 20 were tested and, again, higher r values will generate better results in higher dispersion data streams.

The parameter f, used in FSWf, controls the number of bits available in the non-unique filter. The value should be set depending on the number of distinct elements expected during a single sub-window. A value within the same magnitude of D would be appropriate.

9 Experimental results

Given the approximation to the sliding window used in FSP and FSW, it is natural to compare the results, memory and operations required by those algorithms with the alternative of using one top-k algorithm starting at each sub-window. Instead of using FSP or FSW for collecting daily top-k results with an hourly sub-window, one could use 24 instances of Space-Saving or FSS starting at each hour.

Space-Saving requires only one dimensioning parameter, m_{SS}, the number of elements in the monitored list. FSS requires two parameters, m_{FSS}, the number of elements in the monitored list and h_{FSS}, the number of cells in the bitmap counter. In terms of memory use, Space-Saving requires at least 3 m_{SS} counters. Comparable or better results can be obtained with FSS using the same 3 m_{SS} counters by setting $m_{FSS} = m_{SS}/2$ and $h_{FSS} = 3\ m_{FSS}$ as shown in (Homem and Carvalho 2010b). The use of p sub-windows would require $3.\ p.\ m_{SS}$ counters.

Considering that the same dimensioning is used for FSP, increasing its bitmap counter by r (as it only requires one bitmap counter), then $m_{FSP} = m_{FSS}$ and $h_{FSP} = 3\ r.\ m_{FSS}$,

with the number of counters used (as in (Homem and Carvalho 2010b) c_j will be ignored):

$$n_{FSP} = 2.3.p.m_{FSP} + h_{FSP} = 6\,p\,m_{FSS} + 3\,r.m_{FSS} = 6\,m_{FSS}(p + r/2) \quad (15)$$

If the same dimensioning is used for FSW, $m_{FSW} = m_{FSS}$ and $h_{FSW} = 3\ m_{FSS}$, p sub-windows and histograms using p counters, then the number of counters used is:

$$n_{FSW} = m_{FSW}(4 + p) + h_{FSW}(p + r) = m_{FSS}(4 + p) + 3m_{FSS}(p + r) = m_{FSS}(4 + 4p + 3r) \quad (16)$$

with these dimensioning rules, the number of counters in FSW will always be lower than FSS or FSP for any number of sub-windows $p > 2 + r$.

If memory usage is comparable between a single instance of FSW and p instances of Space-Saving or FSS, the difference is huge regarding operations. An element update in FSW requires at most one search/update in the monitored list, one update in the bitmap counter and one update in a histogram. Both bitmap counters and histograms have direct access through an index so the operations are very fast. Each of the instances of Space-Saving will require a search/update in the monitored list for each element. FSS requires the same search/update in the monitored list and one update in the bitmap counter. Having p instances of Space-Saving or FSS increases linearly the number of operations during an element update.

FSW will also require full scans on the monitored list and bitmap counter at sub-window expiry, in a batch process. This will account at most for m_{FSW} updates on the monitored list, h_{FSW} update operations on the bitmap counter, and $m_{FSW} + h_{FSW}$ updates on the histograms. However, if one considers the number of updates in a single sub-window to be greater than $m_{FSW} + h_{FSW}$, the batch process will require less operations than for update during the same window. Under those assumptions, the total operations required for FSW during any sub-window will be in the order of those required by 2 instances of FSS.

The initial set of tests will identify the most frequent words in a large set of newspaper articles. This set comprises 5,208 articles with a total of 1,489,947 words. Texts were written in Portuguese by 87 distinct authors, published during a period of 180 days in a newspaper considered a reference in Portuguese daily newspapers. One can expect this data to present a zipfian distribution.

Articles were processed in chronological order, split into words and punctuation, and simultaneously processed using Space-Saving, FSS, FSP, FSW and FSWr. In this case the sub-windows duration was set to the minimal period distinguishing articles, as these were dated on a daily basis.

To establish a baseline for precision evaluation, exhaustive computation of the exact top-k elements was performed. For each time window in the tests, the top-k elements were determined by exact, brute force method. The precision value of each algorithm was obtained comparing the algorithm results with the exact top-k elements.

Please note that each run for a single algorithm covers a much longer period than the defined sliding window. Each execution generates results at every sub-window period end. Instead of several distinct runs of the algorithm, one has considered the more interesting and more realistic execution of the algorithm in a sustained manner, generating consecutive results as it would be expected in a real life situation. Repeating the execution of the algorithm using the same data would result in exactly the same results, since the pseudo-random hash generates exactly the same value for the same key. For each test scenario, the average and standard deviation of precision results in each sub-window can provide a good indicator of the precision estimate and dispersion achievable with the algorithm.

The first set of tests identifies the weekly top-500 most frequent words. Space-Saving and FSS tests were performed by initiating an instance of the algorithm at each day and keeping it active for a single week. FSP, FSW and FSWr instances were initialized only once and kept active during the full 180 days period of data.

For the experimental tests, the following indicators were considered:

- *Top-k Precision* (and *Top-k Recall*) 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.
- The maximum error μ for each algorithm. The values are not comparable between algorithms; for FSW and FSWr the value shown is the last value of μ, obtained at the end of the execution, for Space-Saving and FSS the value shown is the average of the ending values of μ. FSP does not provide a maximum error μ for the full sliding window.
- The estimated number of *Counters* required by the algorithm. In FSP, as the number will depend on the full list size two values are presented; the observed number of counters and the maximum possible value.
- Operations performance indicators are also given. *Inserts* are the total number of elements inserted in the monitored list, *Removals* the number of elements removed from the monitored list including those that expired, and *Hits and Updates* the number of hits in the monitored list plus the number of elements updates

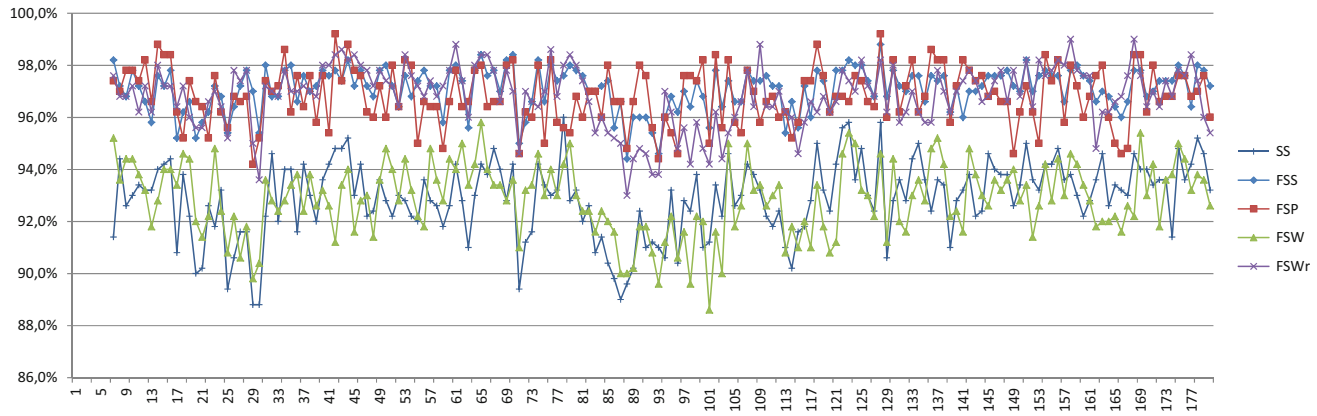

Fig. 6 Weekly top-500 words precision for each algorithm

Table 1 Performance results for weekly top-500 words tests

Algorithm	7 × SS	7 × FSS	FSP	FSW	FSWr
Precision average	92.8%	97.1%	96.8%	92.8%	96.8%
Precision std. deviation	1.5%	0.8%	1.1%	1.4%	1.2%
μ	15.81	13.55	–	15	13
Counters	52,500	52,500	56,000–67,500	43,750	55,000
Inserts	3,809,897	2,228,267	1,104,979	194,470	158,377
Removals	3,374,897	2,010,767	874,294	193,220	157,127
Hits and updates	6,245,284	5,140,202	2,065,535	991,980	1,000,951
Bucket updates	–	–	–	333,159	280,504

during the sub-window expiries. *Bucket Updates* show the number of histograms buckets merged during deletions and expiries.

Parameters for each of the algorithms were set to ensure the memory used in each type remained comparable as previously enunciated:

- For Space-Saving; $m_{SS} = 2{,}500$
- For FSS; $m_{FSS} = 1{,}250$ and $h_{FSS} = 3{,}750$
- For FSP; $m_{FSP} = 1{,}250$ and $h_{FSP} = 15{,}000$ and $p = 7$
- For FSW; $m_{FSW} = 1{,}250$, $h_{FSW} = 3{,}750$ and $p = 7$
- For FSWr; $m_{FSWr} = 1{,}250$, $h_{FSWr} = 3{,}750$, $p = 7$ and $r = 4$

Figure 6 shows the precision results for each of the algorithms at the end of each day. Periods with less than p full days were not considered. Table 1 details the performance of each algorithm.

The second set of tests repeat the experiment for the period of 1 month, maintaining the daily sub-window. Algorithm parameterization was kept with the exception of FSP and FSWr, were r was changed to 16 to take advantage of the larger period p. Figure 7 and Table 2 show the results of this set of sets.

Clearly and as expected, FSS achieves the best results, being consistently better than any other in every situation. Space-Saving performs worse than FSS. The number of operations required by the use of multiple instances of FSS is huge when compared to those required by FSW. This is even worse for Space-Saving. FSW performance degrades significantly from weekly to monthly period, being the less stable of all. FSWr and FSP performance follows very near FSS. The number of operations required by FSP is still very high. FSWr provides an excellent compromise between results and required operations.

One of the most interesting aspects regarding FSW and FSWr behavior is that, most of the times, insertion and subsequent removal of elements from the monitored list occur in a very short time. In fact most of the elements are removed from the monitored list in the same sub-window they were inserted in. Table 3 presents the percentage of removals as a function of the duration of stay in the monitored list.

This motivates the use the wider bitmap counter for the active sub-window, as it allows more detail to be kept during that critical stage.

An additional set of tests present an even harder scenario: finding the most frequent destinations in a large set of mobile calls. The data set includes 6,270,022 real call

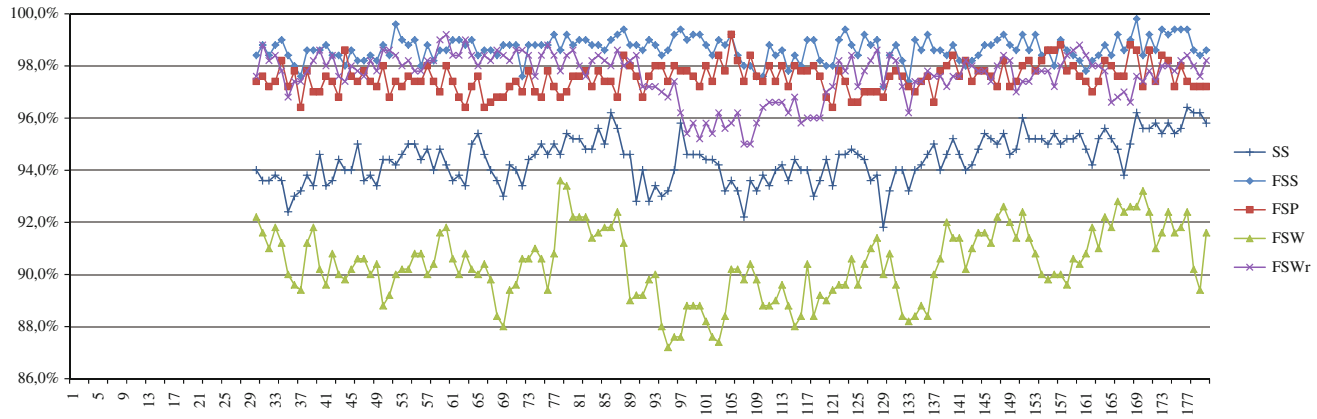

Fig. 7 Monthly top-500 words precision for each algorithm

Table 2 Performance results for monthly top-500 words tests

Algorithm	30 × SS	30 × FSS	FSP	FSW	FSWr
Precision average	94.4%	98.7%	97.5%	90.4%	97.6%
Precision std. deviation	0.9%	0.5%	0.5%	1.4%	0.9%
μ	73.41	55.01	–	71	57
Counters	225,000	225,000	198,000–285,500	158,750	215,000
Inserts	13,619,455	4,598,213	990,202	102,120	62,576
Removals	13,241,955	4,409,463	754,888	100,870	61,326
Hits and updates	23,934,302	19,771,780	2,323,366	982,196	994,295
Bucket updates	–	–	–	237,388	172,715

Table 3 Monitored list removal statistics

Period	7 days		30 days	
Duration in the list	FSW (%)	FSWr (%)	FSW (%)	FSWr (%)
Less than 1 sub-window	65.5	66.8	58.7	58.3
Exactly 1 sub-window	18.9	16.3	19.8	16.1
Exactly 2 sub-windows	6.8	6.2	7.5	7.2
More than 2 sub-windows	8.8	10.6	14.0	18.4

records, from a total of 3,170 distinct accounts (many with multiple users), made during a period of 35 days. The calls records are ordered by starting time. Within this set, a total of 913,325 distinct destinations were identified. Of those, 385,082 received just a single call. The calls present a very characteristic daily and weekly cyclical distribution.

The set of tests will identify the daily top-500 most frequent destinations within a hourly sub-window. Again, Space-Saving and FSS tests were performed by initiating an instance of the algorithm at each hour and keeping it active for a single day. FSP, FSWr and FSWf instances were initialized only once and kept active during the full 35 days period of data.

Due to the very high number of distinct values, the overall parameters had to be set higher than in previous tests. FSWr and FSWf instances were also given extra space in the monitored list to ensure comparable performance:

- For Space-Saving; $m_{SS} = 8{,}000$
- For FSS; $m_{FSS} = 4{,}000$ and $h_{FSS} = 12{,}000$
- For FSP; $m_{FSP} = 4{,}000$ and $h_{FSP} = 96{,}000$ and $p = 24$
- For FSWr; $m_{FSWr} = 8{,}000$, $h_{FSWr} = 12{,}000$ $p = 24$ and $r = 8$
- For FSWf; $m_{FSWf} = 8{,}000$, $h_{FSWf} = 12{,}000$, $p = 24$, $r = 8$ and $f = 200{,}000$

Note that the unique element filter of FSWf requires a bit array with 200,000 bits, equivalent to 12,500 counters with 16 bits each.

Figure 8 shows the precision results for each of the algorithms at the end of each hour. Periods with less than 1 day were not considered. Table 4 details the performance of each algorithm.

There is a clear weekly cycle, and precision increases significantly during weekends as the number of calls decreases. Space-Saving and FSS performance are not good as there is a huge number of single call destinations, and this introduces a very high number of insertions and deletions in

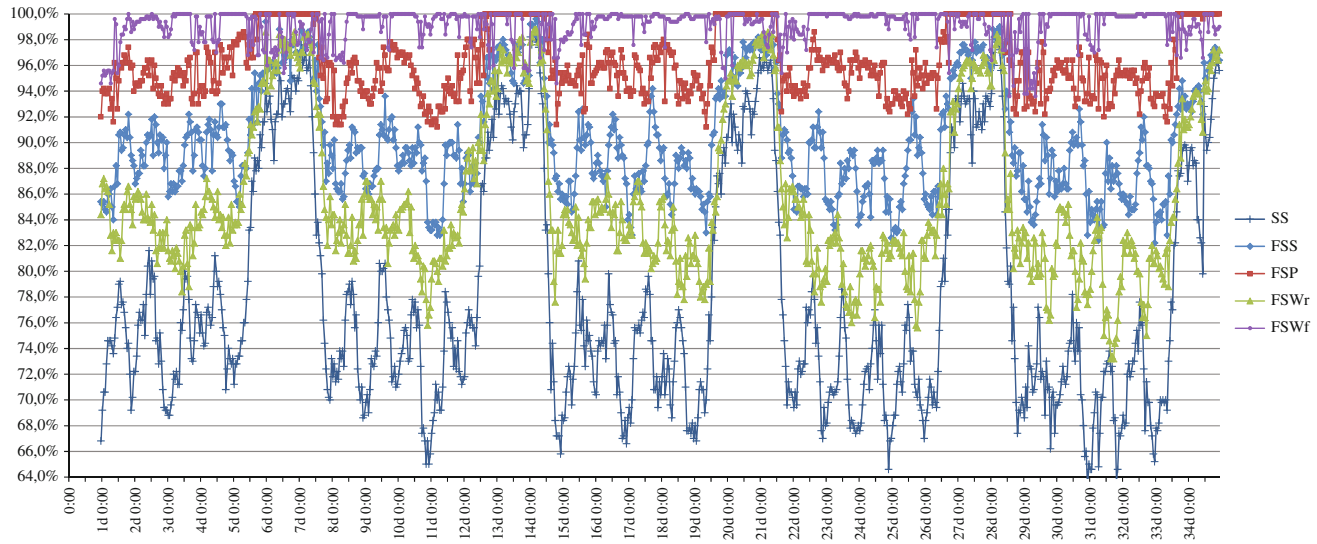

Fig. 8 Daily top-500 destinations precision for each algorithm

Table 4 Performance results for daily top-500 destinations tests

Algorithm	24 × SS	24 × FSS	FSP	FSWr	FSWf
Precision average	78.0%	90.2%	96.3%	85.8%	99.1%
Precision std. deviation	9.2%	4.3%	2.6%	6.3%	1.3%
μ	1.74	1.17	–	4	1
Counters	576,000	576,000	460,000–672,000	412,000	496,000
Inserts	96,050,191	59,779,547	7,928,158	2,208,715	462,695
Removals	89,522,191	56,515,547	5,331,534	2,120,715	455,778
Hits and updates	51,205,610	39,967,243	6,006,718	8,656,562	8,365,987
Bucket updates	–	–	–	8,436,011	4,842,797

the bottom of the monitored list. This huge substitution rate degrades the error estimate. The effect is greater in Space-Saving than in FSS as the bitmap counter filters some of the replacements. The same effect is also present in FSW. FSP handles this very well as it "averages" each hour and computes the final answer. The non-unique filtering option is clearly appropriate for this scenario as it removes most of the single call destinations and focuses the algorithm on the most significant destinations.

The second set of tests repeats the experiment for a 3-day period, with a 4-hour sub-window. Algorithm parameterization was kept. Figure 9 and Table 5 show the results of these tests.

The 3-day period breaks the weekly cycle except for Space-Saving. Space-Saving, FSS and FSW performance remain affected by the huge volume of single call destinations. In fact, this large number of unique elements introduces a sort of "noise", increasing the error μ and lowering significantly the precision. Without the filtering effect of the bitmap counter Space-Saving performance is heavily degraded and becomes very unstable. The averaging of FSP delivers excellent results. The non-unique filtering option of FSWf is still critical for exceptional and stable results.

10 Conclusions

This paper presents new algorithms that build on top of the best existing algorithm for answering the unrestricted top-k problem and extends it to the sliding window scenario. The FSS algorithm was proposed by the authors (Homem and Carvalho 2010b) to solve the unrestricted top-k problem. It improves Space-Saving algorithm proposed by Metwally et al. (2005). It merges the best properties of two distinct approaches to this problem, the counter-based techniques and sketch-based ones and provides strong guarantees regarding inclusion of elements in the answer, ordering of

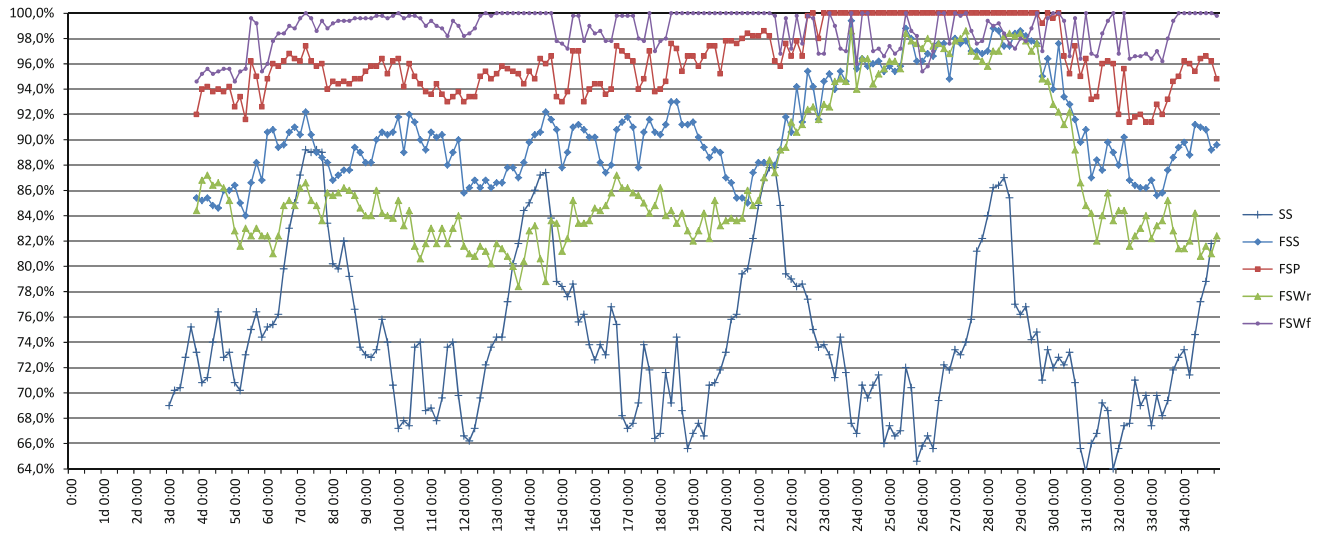

Fig. 9 3-Day top-500 destinations precision for each algorithm

Table 5 Performance results for 3-day top-500 destinations tests

Algorithm	18 × SS	18 × FSS	FSP	FSWr	FSWf
Precision average	74.1%	90.8%	96.6%	93.4%	99.1%
Precision std. deviation	6.1%	2.0%	1.1%	1.5%	0.6%
μ	1.29	0.78	–	18	7
Counters	432,000	432,000	396,000 to 528,000	400,000	412,500
Inserts	67,916,042	34,278,985	5,800,014	1,282,823	481,935
Removals	66,380,042	33,510,985	4,914,817	1,274,735	473,935
Hits and updates	36,654,727	28,110,337	5,481,997	3,524,475	3,588,049
Bucket updates	–	–	–	2,798,116	1,734,535

elements, maximum estimation error and probabilistic guarantees of increased precision. The use of a bitmap counter in this algorithm minimizes the operations of update of the monitored elements list by avoiding elements with insufficient hits being inserted in the list. The two algorithms were used as a reference for comparing results.

A less optimized algorithm, both in memory and operations, FSP, is presented as an intermediate step when solving the top-k problem in a sliding window. It basically reuses the FSS algorithm within a sub-window and generates a full period answer by merging the sub-windows results. It provides excellent precision by averaging several periods. It, however, requires more than the double of update operations of FSW and its derivatives. It may also give worse results in situations where the entire distribution changes very significantly during a single period.

The FSW algorithm extends FSS by including histograms to allow for time expiration of elements. It uses an approximation to a sliding window by considering p basic time sub-windows. Sub-windows can have equal or non-equal duration. Although this is in fact a "*jumping*" sliding window the approximation is good for most purposes.

Unfortunately, FSW does not maintain all the properties of FSS, but it still provides strong guarantees on the inclusion of elements in the list depending on their real frequency. It also provides stochastic error bounds that further improve the performance. FSW provides an error estimate for each frequency, giving the possibility to control the quality of the estimate.

This paper presents experimental results that illustrate the precision and performance of the algorithms when using a similar memory space. The two sets of data used represent two real life situations where determining the top-k elements is important. Memory consumption was key in the analysis, since this problem arises in many situations where memory is limited. In this regard, the use of FSW is envisioned with even less memory than FSP and with

comparable 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.

Two important improvements over the basic FSW are also presented: the use of a wider bitmap counter for the active sub-window while keeping a smaller set of histograms for sub-window expiry; and the use of a non-unique filter for situations were large numbers of unique and low frequency elements introduce "noise" in the top-k detection. With these two improvements FSW delivers exceptional results, in some cases better than those of FSS for comparable periods. Adequate dimensioning of the parameters for each specific situation can therefore generate good results with minimal memory consumption.

FSW and its derivatives are low memory footprint algorithms that can answer not only the top-k problem within a sliding window for large number of transactions, but also the problem of answering huge number of top-k problems for a single data stream (as in the case of individual top-k lists). It can be used in any domain as long as the appropriate implementation choices and dimensioning are made.

References

Aggarwal C (ed) (2007) Data streams: models and algorithms, Springer, Berlin

Babcock B, Datar M, Motwani R, O'Callaghan L (2003) Maintaining variance and k-medians over data stream windows. In: Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, pp 234–243

Bertsekas D (1995) Dynamic programming and optimal control. vol 1. Athena Scientific, Nashua

Braverman V, Ostrovsky R (2007) Smooth histograms for sliding window, in FOCS 2007, pp 283–293

Cohen E, Strauss M (2006) Maintaining time-decaying stream aggregates. J Algorithms 59(1):19–36

Cormode G, Hadjieleftheriou M (2010) Methods for finding frequent items in data streams. VLDB J, vol 19

Cormode G, Muthukrishnan S (2003) What's hot and what's not: tracking most frequent items dynamically. In: Proceedings of the 22nd ACM PODS symposium on principles of database systems, pp 296–306

Datar M, Gionis A, Indyk P, Motwani R (2002) Maintaining stream statistics over sliding windows. SIAM J Comput 31(6):1794–1813

Demaine E, López-Ortiz A, Munro J (2002) Frequency estimation of internet packet streams with limited space. In: Proceedings of the 10th ESA Annual European Symposium on Algorithms, pp 348–360

Dimitropoulos X, Hurley P, Kind A (2008) Probabilistic Lossy Counting: an efficient algorithm for finding heavy hitters, ACM SIGCOMM Comput Commun Rev, vol 38, no. 1

Estan C, Varghese G (2003) New directions in traffic measurement and accounting: focusing on the elephants, ignoring the mice. ACM Trans Comput Syst 21(3):270–313

Ganguly S (2003) Counting distinct items over update streams. Theoret Comput Sci 378(3):211–222

Gibbons P, Tirthapura S (2002) Distributed streams algorithms for sliding windows. In: Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures, pp 63–72

Gilbert A, Guha S, Indyk P, Kotidis Y, Muthukrishnan S, Strauss M (2002) Fast, small-space algorithms for approximate histogram maintenance In: Proceedings of the thiry-fourth annual ACM symposium on Theory of computing, pp 389–398

Golab L, DeHaan D, Demaine E, Lopez-Ortiz A, Munro J, (2003), Identifying frequent items in sliding windows over on-line packet streams. In: Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement table of contents, pp 173–178

Homem N, Carvalho J (2010) Estimating Top-k destinations in data streams, computational intelligence for knowledge-based systems design. Springer, Berlin/Heidelberg, pp 290–299

Homem N, Carvalho J (2010) Finding top-k elements in data streams. Inform Sci 180(24):4958–4974

Hu T, Sung S, Xiong H, Fu Q (2008) Discovery of maximum length frequent itemsets. Inf Sci 178:69–87

Hua-Fu Li, Suh-Yin Lee (2009) Mining frequent itemsets over data streams using efficient window sliding techniques. Expert Syst Appl 36(2):1466–1477

Lee L, Ting H (2006) A simpler and more efficient deterministic scheme for finding frequent items over sliding windows. In: Proceedings of the 25th ACM SIGMOD-SIGACT-SIGART symposium on principles of satabase systems, pp 290–297

Manerikar N, Palpanas T (2009) Frequent items in streaming data: an experimental evaluation of the state-of-the-art. Data Knowl Eng 68(4):415–430

Manku G, Motwani R (2002) Approximate frequency counts over data streams. In: Proceedings of the 28th ACM VLDB international conference on very large data bases, pp 346–357

Matias Y, Vitter J, Wang M (1998) Wavelet-based histograms for selectivity estimation. In: Proceedings of the 1998 ACM SIGMOD international conference on management of data, pp 448–459

Matias Y, Vitter J, Wang M (2000) Dynamic maintenance of wavelet-based histograms. VLDB, pp 101–110

Metwally A, Agrawal D, Abbadi A (2005) Efficient computation of frequent and Top-k elements in data streams. Technical Report 2005-23. University of California, Santa Barbara

Muthukrishnan S (2005) Data streams: algorithms and applications, foundations and trends. Theoret Comput Sci 1(2):117–236

Qiao L, Agrawal D, Abbadi A (2003) Supporting sliding window queries for continuous data streams. In: Proceedings of the 15th international conference on scientific and statistical database management, pp 85–94

Tanbeer S, Ahmed C, Jeong B, Lee Y (2009a) Efficient single-pass frequent pattern mining using a prefix-tree. Inf Sci 179(5): 559–583

Tanbeer S, Ahmed C, Jeong B, Lee Y (2009b) Sliding window-based frequent pattern mining over data streams. Inf Sci 179(22): 3843–3865

Tantono F, Manerikar N, Palpanas T (2008) Efficiently discovering recent frequent items in data streams In: Scientific and statistical database management, Lecture Notes in Computer Science, pp 222–239

Zhu Y, Shasha D (2002) Statistical monitoring of thousands of data streams in real time. In: Proceedings of the 28th international conference on very large data bases, pp 358–369