A Two-List Framework for Accurate Detection of Frequent Items in Data Streams

David Vengerov

Oracle Labs, Belmont CA 94002, USA
david.vengerov@oracle.com

Abstract. The problem of detecting the most frequent items in large data sets and providing accurate frequency estimates for those items is becoming more and more important in a variety of domains. We propose a new two-list framework for addressing this problem, which extends the state-of-the-art Filtered Space-Saving (FSS) algorithm. An algorithm called FSSA giving an efficient array-based implementation of this framework is presented. An adaptive version of this algorithm is also presented, which adjusts the relative sizes of the two lists based on the estimated number of distinct keys in the data set. Analytical comparison with the FSS algorithm showed that FSSA has smaller expected frequency estimation errors, and experiments on both artificial and real workloads confirm this result. A theoretical analysis of space and time complexity for FSSA and its benchmark algorithms was performed. Finally, we showed that FSS2L framework can be naturally parallelized, leading to a linear decrease in the maximum frequency estimation error.

Keywords: Data Mining; Frequent Items; Skew Detection, Space-Saving.

1 Introduction

The problem of detecting the most frequent items in large data sets and providing accurate frequency estimates for those items arises in many areas:

- Internet providers need to know the most frequent destinations in order to manage traffic and service quality.
- Social network companies need to find the most frequent interactions among the users in order to extract information about connections and relations between individuals.
- Retail companies need to know the most common products purchased by each customer in order to better classify the customer and design appropriate marketing campaigns.
- An increasing number of businesses are finding it useful to monitor the most frequent topics in news streams or social media so as to adjust their business decisions based on these topics.
- Database engines can optimize query plans depending on the degree of skew in the underlying data and can give separate treatment to the most frequent items.

In order to be practical, any algorithm that tackles this problem has to perform its computations using limited memory, usually orders of magnitude less than the size of the data set. Another key practical requirement is performing all computations in one pass – without this feature, an algorithm would not be useful for very large data sets or data streams.

Several algorithms that meet these requirements have been proposed in the literature, the most well-known ones being Frequent [3], Lossy Counting (LC) [6] and Space-Saving (SS) [8]. The Frequent algorithm was shown to be much less accurate than either LC or SS (see, for example, [1,7]). The Space-Saving (SS) algorithm was shown (e.g., [1,7]) to be more accurate than Lossy Counting (LC). During our literature search, we found an improved version of SS called Filtered Space-Saving (FSS) [5], which was shown to be more accurate than SS, both in [5] and in our experiments in Section 5.

However, we discovered that the FSS algorithm suffers from the fact that infrequently observed items can still get onto the list of items tracked by FSS and displace from it many of the frequent items that we would like to keep. In order to avoid this problem, we extended the FSS algorithm by keeping information about the frequently observed items in a special protected list, which prevents that information from being displaced by the infrequent items. We refer to this two-list framework as FSS2L.

The rest of this paper is organized as follows. Section 2 describes the previously developed SS and FSS algorithms and presents a novel analysis of the expected behavior of the FSS algorithm. Section 3 presents the FSS2L framework and analyzes its expected behavior. It then describes an efficient implementation of this framework using a single array, which we called FSSA algorithm. Finally, it presents an algorithm for adapting the relative sizes of the two lists used in the FSS2L framework and discusses the impact of such adaptation. Section 4 presents analysis of the space and time complexity of FSSA and its benchmark algorithms. Section 5 evaluates these algorithms on both artificial and real distributions of item keys. Section 6 summarizes the results and explains how the FSSA algorithm can be parallelized.

2 Relevant Prior Algorithms

2.1 Space-Saving Algorithm

The Space-Saving (SS) algorithm [8] stores the estimated frequency of each monitored key in a hash map with a maximum size m. If the observed key i has a corresponding entry in the hash map, then its frequency estimate f_i is incremented. Otherwise, an entry is added to the hash map with the estimated frequency $f_{min} + 1$, where f_{min} is the minimum frequency among all the keys currently stored in the hash map. When the number of entries in the hash map exceeds m, the entry corresponding to the key with the smallest estimated frequency f_{min} is removed.

By assigning the frequency $f_{min} + 1$ to each newly inserted key, the SS algorithm never underestimates a key's frequency. This is so because a newly observed key could have been previously observed at most f_{min} times, then removed from the hash map (possibly when the previous key was inserted), and was just observed again. Since a key that was never observed before will be inserted with a frequency $f_{min} + 1$, the maximum possible error Δ_i in the frequency estimate for a key i satisfies $0 \leq \Delta_i \leq$

f_{min}. The value f_{min} will grow at the fastest rate if all observed keys are distinct, in which case f_{min} will be incremented by 1 after m keys are observed. Therefore, $f_{min} \leq N/m$, and the final worst-case accuracy guarantee for SS becomes $0 \leq \Delta_i \leq N/m$. A formal proof of this fact is given in [8].

2.2 Filtered Space-Saving Algorithm

The Filtered Space-Saving (FSS) algorithm [5] improves the accuracy of the SS algorithm by storing fewer keys in the hash map and allocating the "saved" space to an array H that holds hashed key values. Instead of always adding a new entry to the hash map when a key is observed for which no entry exists, the key is first hashed, uniformly, into the set $\{1, 2, \ldots, sizeH\}$, where $sizeH$ is the number of cells in H. If the key is hashed to k and $H[k]$ is smaller than $f_{min} - 1$, then $H[k]$ is increased by 1 and no further action is taken. Otherwise, a new entry is created in the hash map with this key, its estimated frequency f_i is set to $H[k]$+1 and its maximum error Δ_i is set to $H[k]$. When the number of keys in the hash map exceeds the preset upper bound, a key with frequency equal to f_{min} is removed, and if $H[j] < f_{min}$, where j is the value to which the removed key hashes, then $H[j]$ is set to f_{min} so that it would still represent the maximum overestimation error for any newly inserted key that hashes to cell j (in case the key with frequency f_{min} that was just removed is the next key to be observed again).

FSS has the following worst-case accuracy guarantee: $0 \leq \Delta_i \leq H[k] \leq f_{min} \leq N/M$, where M is the number of elements it stores in the hash map. These guarantees can be easily inferred from the operations of FSS. Since Δ_i for each newly inserted key i that hashes to cell k is set to $H[k]$, and $H[k]$ can still grow over time, we get $0 \leq \Delta_i \leq H[k]$. The values stored in H are incremented only when $H[k] + 1 < f_{min}$ for the observed key or when a key is evicted and its corresponding $H[j] < f_{min}$, implying that $H[k]$ is always $\leq f_{min}$ for every k. Finally, let S_0 be the sum of the frequencies of all keys monitored by FSS and note that every observed key either increments S_0 by 1 or leaves it unchanged. Since N keys are observed, $N \geq S_0 \geq M f_{min}$, which implies $f_{min} \leq N/M$.

Since FSS stores fewer keys in its hash map than SS, the worst-case error for FSS derived above is larger than that for SS. However, such an error can only be observed if all keys happen to hash to the same cell in the hash array, which has a negligible probability for any real hashing scheme whose goal is to spread the keys uniformly over all cells in the hash array.

2.3 Efficiently Finding the Smallest Frequency Key

In order to implement efficiently all frequency estimation algorithms built on top of SS, one needs to be able to quickly find the key with the smallest frequency f_{min} when it needs to be evicted from the hash map. In order to perform this eviction in $O(1)$ time, the authors of the SS algorithm came up with the "Stream Summary" data structure [8] for storing information about the monitored keys.

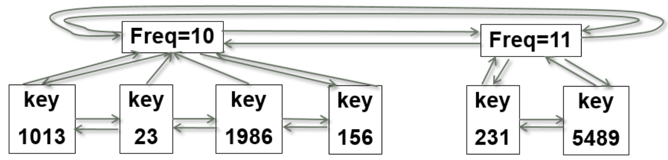

Fig. 1. The "modified" Stream Summary linked list data structure used for storing keys in our implementation of SS and FSS.

In this data structure, each element in the top-level double linked list is a "bucket" with a specified frequency F_b (either 10 or 11 in Fig. 1), which contains its own linked list of nodes that stores keys that have estimated frequency F_b. The nodes are always added to the end of the list inside each bucket. If the total number of keys tracked by Stream Summary has exceeded a prespecified threshold, then the first node in the first bucket is removed. Each "bucket" maintains pointers to the first node and to the last node on its list, which makes insertion and removal operations take $O(1)$ time in the linked list. When a key that is present in the hash map with an estimated frequency f_i is observed in the data set, it is moved to the end of the linked list in the bucket with $F_b = f_i + 1$, and if such a bucket does not exist, a new bucket is created (and is inserted into a proper location in the list of buckets so as to still keep it sorted by frequency) and the node with the observed key is added to the list of nodes in that bucket.

In order to perform the removal of a node from its current location in $O(1)$ time, each node can also maintain a pointer to the previous node in the list, making the list of nodes doubly linked (otherwise, a linear search will be needed in order to find the node before the one being removed, so that it could then be linked with the node after the one being removed). This modified version of the Stream Summary was used in the experiments described in Section 5 and is shown in Fig. 1.

We also used a suggestion that was made in [8] (and explained in more detail in [2]) for the hash map to hold pointers into the Stream Summary, so that no linear search would be needed in order to find specific keys in it. This implementation requires storing 2 numbers in the hash map for each monitored key (the key and the pointer to the Stream Summary node) and storing 4 numbers in the corresponding node: the key and the pointers to the parent bucket, to the previous node and to the next node.

3 Proposed Algorithms

3.1 FSS2L Framework

The general idea of the proposed FSS2L framework is to split the hash map entries into two conceptual lists: List0 where new keys are first placed and List1 to which keys that are considered to be frequent are promoted from List0. Keys in List0 are sorted by their estimated frequency, while those in List1 are sorted by their "hits" value – the number of times there were observed since they were first added to List0. The hits value of each monitored key i can be computed dynamically from its estimated frequency and maximum error, as will be explained below.

The operations of this FSS-two-list (FSS2L) framework are shown in Fig. 2. When a new key i is observed and is found in List1, its estimated frequency is incremented and it is moved closer to the top of List1 in order to still keep that list sorted by hits. If the observed key i is found in List0, then it is upgraded to List1 if $h_i \geq h_{min} - 1$ (where h_{min} is the minimum number of hits for any element in List1) and its estimated frequency is incremented. Otherwise, the estimated frequency f_i for this key is incremented and, if needed, the key is moved closer to the top of List0. If the key is upgraded and if the number of elements m_1 in List1 became greater than a fraction q of the maximum number of keys that the algorithm can track (which we will denote by M to be consisted with the FSS algorithm), then the last element from List1 is inserted into List0.

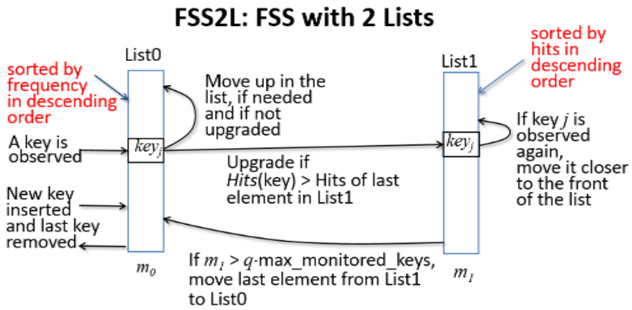

Fig. 2. The keys in List1 are protected in the FSS2L framework from being washed out by the inflow of new keys that are unlikely to be observed again.

If the observed key i is not found in List0 or List1, its key is hashed into the array $H[k]$ and the same procedure as in FSS is followed in order to decide whether or not this key should be inserted into List0 (ahead of all items with the same frequency, so as to stay longer in the list). This procedure implies that if a new key i is inserted into List0, it is assigned estimated frequency $f_i = H[k] + 1$ and a maximum error $\Delta_i = H[k]$. Therefore, its "hits" value (the number of times it was observed after the algorithm started monitoring it) can always be computed dynamically as $h_i = f_i - \Delta_i - 1$.

3.2 FSSA Algorithm

A straight-forward implementation of FSS2L would use two Stream Summary lists of the type described in Section 2.3 (for storing information about keys in List0 and List1 and efficiently moving the keys inside these lists or between them) and two hash maps (HashMap0 and HashMap1 with entries indexed by keys from the corresponding lists) for efficiently finding the key of interest in each Stream Summary list (as was suggested at the end of Section 2.3).

However, there are cases when it is undesirable to use a linked list implementation. For example, if the data structures need to be written to disk periodically and then read from disk, then the pointer information will become inaccurate. Also, the number of nodes representing keys in List0 can initially grow to M, and then the unused nodes will need to be deallocated as List1 grows to its maximum size of qM while List0 shrinks down to $(1 - q)M$. Deallocation of memory can be hard to implement in some contexts, and the deallocated memory may not be reusable until the algorithm finishes its data processing. These practical concerns motivated us to develop a single array implementation of the FSS2L framework, which we called FSSA algorithm, shown in Fig. 3.

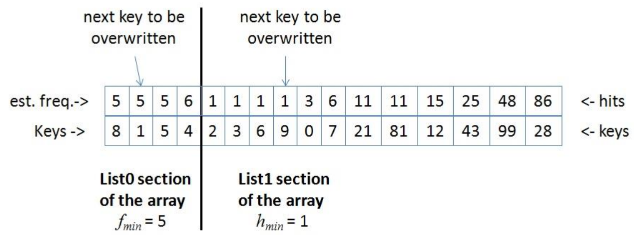

Fig. 3. The 2D array data structure used in FSSA. Keys in the List0 section are sorted by estimated frequency, while those in the List1 section are sorted by observed hits.

This algorithm allocates a 2D array of size M-by-2 to store information about the monitored keys. If the key belongs (conceptually) to List0, then the array stores the key and its estimated frequency. If the key belongs (conceptually) to List1, then the array stores the key and its number of hits. The keys from List0 are kept together at the front of the array and are sorted by frequency, while the keys from List1 are kept together at the end of the array and are sorted by hits.

In order to make insertions of new keys into the List0 section of the array more efficient, instead of always inserting new keys ahead of all other keys with frequency equal to f_{min}, FSSA cycles through those keys and overwrites them one after another. In order to accomplish this, FSSA stores the index of the next key to be overwritten and the largest index for which a key still has frequency equal to f_{min}. Similarly, in order to make "upgrades" of keys from the List0 section to the List1 section more efficient, FSSA stores the index of the next key to be overwritten in List1 and the largest index for which a key still has hits equal to h_{min}. Also, in order to minimize movement of memory within this array, when FSSA increments the estimated frequency for key i and decides to move it closer toward the end of the List0 section, it simply finds, using binary search, the last key that has frequency equal to f_i and swaps it with key i (which will now have frequency $f_i + 1$). A similar strategy is used when incrementing the hits value for a key in the List1 section.

When a new key i is observed and is found in HashMap1, the value of h_i is first computed as $h_i = f_i - \Delta_i - 1$ using f_i and f_i stored in the hash map. Then, the position of key i in the List1 section of the array is determined using binary search for h_i (followed by a short linear search over all keys that have the same hits value). Similarly, if the observed key i is found in HashMap0, then its position in the List0 section of the array is determined using binary search for keys that have a frequency f_i followed by a linear search for the exact key. Therefore, FSSA can perform all of its operations by storing only 5 numbers for each monitored key: the key, its estimated frequency and its maximum error in the hash map and two more numbers in the sorted array. This makes FSSA the most space efficient algorithm out of the ones we considered, as will be discussed in Section 4.

3.3 Optimizations that can be added to FSS and FSSA

In order to speed up the operations with a large hash map, the FSS and FSSA can substitute it by an array A of the same size as H, with $A[k]$ storing a hash map for those keys that hash to k. In this way, every operation with the hash map will become faster as it will be done on a much smaller hash map (which is implemented in C++ as a red-black tree with update time being logarithmic in the hash map size). As a result, our experiments showed that the overall execution time for the algorithms was reduced by 25-50% (a larger improvement was observed for lower levels of skew in the frequency distribution of keys).

Another optimization can be added to FSS and FSSA to reduce their space consumption. It is based on the observation made in [5] that with a good uniform hashing of keys into H, the difference between $\max_k H[k]$ and $\min_k H[k]$ is small. Therefore, after observing every n keys (where n can be $10 \cdot sizeH$), $\min_k H[k]$ is computed and is added to a variable H_c that holds the common amount subtracted so far from all cells in H, while all cells in H are decremented by $\min_k H[k]$. As a result, it is sufficient to restrict cells in the hash array to store nonnegative integers that are less than 2^{16} (i.e., take up only 2 bytes each). Then, the criterion for adding to the hash map a new key that hashes to cell k would be $H[k] + H_c \geq f_{min} - 1$. Also, when a key that hashes to cell j is removed from H, we set $H[j] = \max(f_{min} - H_c, H[j])$, where the max() operator is needed because if the removed key was recently moved from List1 to List0, then $f_{min} - H_c$ can be smaller than $H[j]$.

3.4 Adapting the Ratio of List0 Size to List1 Size in FSS2L

During our experiments with different frequency distributions of keys, we discovered that the maximum allowed size of List1, which we have previously denoted by $q \cdot M$ with q being a tunable parameter, has an important effect on the results obtained by the FSS2L framework.

First of all, note that if we let $q = 0$, then FSSA behaves very much like FSS, differing from it only in the ordering of keys within groups that have the same frequency (our experimental results confirm this claim). As was pointed out in Section 2.3, the implementation of SS (and by extension of FSS) currently assumed in the literature requires storing 2 numbers in the hash map and 4 numbers in the Stream Summary data structure for each monitored key. However, as was described in Section 3.2, FSSA can perform all its operations by storing 3 numbers in the hash map and 2 numbers in the array for each monitored key. Thus, by setting $q = 0$ in the FSSA algorithm, we obtain FSSA(0), which is a more space efficient implementation of FSS than is currently known in the literature.

As q is increased, more and more of the highest frequency keys can enjoy protection on List1 from being displaced by an inflow of infrequently observed keys. However, it also becomes more difficult for keys to get onto List1, since they will spend less time in List0 (whose size is reduced as q is increased). Furthermore, a smaller List0 implies a faster turnover of its keys, which in turn implies a faster growth rate for f_{min}.

When the number of distinct keys in the dataset has the same order of magnitude as the number of entries in the hash array, even a perfectly random hash function will not be able to spread out the keys uniformly over the hash array and some of its cells will have one or zero keys hashed to it. If a medium-frequency key i happens to be the only key hashed to a certain cell $H[k]$ in the array, then we often found that f_{min} grows faster than $H[k]$ if q is set to 0.75 or larger, and key i does not make it into List0. As our experiments in Section 5 show, when the number of distinct keys in the data set is small, FSSA(0.75) will miss some of the top n most frequent items, while FSSA(0.25) will discover many of them but will have a larger frequency estimation error than FSSA(0.75). Table 5 shows that the effect described above becomes more noticeable when the degree of skew in the frequency distribution of keys decreases, because it becomes harder for the nth most frequent key to be observed sufficiently many times to catch up with the growing f_{min}.

The above observations suggest that if one is most concerned with minimizing the frequency estimation error, then one should use FSS2L with $q = 0.75$. However, if one is most concerned with not missing any of the top n most frequent items for a value of n that is as large as possible, then one should start with $q = 0.25$, then estimate the number of distinct keys in the dataset, and if this number is found to be much larger than the size of the hash array, then one should increase q to 0.75.

In order to not use too much extra space by running a separate algorithm for estimating the number of distinct keys (such as HyperLogLog described in [4]), it is possible to add a small data structure to the FSS2L framework, which can then be used to determine whether or not the number of keys in the data set is much larger than $sizeH$. This data structure consists of a boolean array B of size $sizeH$-by-9 which is initialized with FALSE values. Whenever a key is observed, hashed to cell k in H and is then not inserted into HashMap0, then if $B[k,8]$ is FALSE, this key is hashed uniformly into the set $\{0,1,\ldots,7\}$, say to a value j, and then the cell $B[k,j]$ is set to TRUE. At that point, if $B[k,j]$=TRUE for $j \leq 7$, then $B[k,8]$ is set to TRUE. Then, when the number of observed keys becomes 32$\cdot$ $sizeH$, the cells $B[k,8]$ are examined for all k, and if the fraction of them with $B[k,8]$=TRUE is greater than 0.01, then we can conclude that the number of distinct keys in the dataset is much larger than $sizeH$, and hence the value of q should be increased from 0.25 to 0.75 (if one is most concerned with not missing any of the top n most frequent items). In experiments presented in Section 5, this algorithm is called Adaptive FSSA, abbreviated as AFSSA.

4 Space and Time Complexity of Considered Algorithms

All the SS-based algorithms discussed in this paper start with checking whether or not an entry for the observed key exists in the appropriate hash map, which on average takes $O(1)$ time. If the hash map does contain such an entry, then this entry needs to be retrieved, which takes $O(\log(m))$ time. When an entry needs to be removed from the hash map, the $O(\log(m))$ cost is incurred once again. If the modified Stream Summary data structure described in Section 2.3 is used to store information about each key and a hash map is used to store pointers into Stream Summary, then all Stream Summary operations take constant time.

Such an implementation requires storing pairs (key, pointer) in the hash structure as well as the key, the pointer to the next node, the pointer to the previous node, and the pointer to the parent bucket in the Stream Summary. Each bucket in the Stream Summary needs to store its frequency as well as the pointers to the first and to the last node in its linked list. In modern database systems each number takes 8 bytes and each pointer uses 8 bytes in order to handle 64-bit memory spaces. Thus, in order to track m keys, SS requires $48m + 3b$ bytes, where b is the number of buckets in the Stream Summary. If the frequency distribution of keys has a low skew, then b is likely to be small (usually 2 or 3 in our experiments), but if the skew is large, then b can be close to m.

The authors of the FSS algorithm suggest in [5] that FSS can track only half as many keys as SS and still be more accurate than SS if the same amount of memory is given to both algorithms. They also suggest that the FSS hash array H should have 3 times as many cells as the number of monitored keys (so if FSS monitors $\frac{m}{2}$ keys, then $sizeH = \frac{3m}{2}$). Using these settings and the same data structures as the ones used for SS, all the data structures in FSS besides the hash array will use $24m + 3b$ bytes (assuming, for simplicity, that the same number of buckets b is present in the Stream Summary for both algorithms). If the hash array is implemented using 2 bytes for each cell as was described at the end of Section 3.3, then the total space taken by FSS would be equal to $24m + 3b + 2\frac{3m}{2} = 27m + 3b$ bytes in order to track $m/2$ keys. The FSS operations are essentially the same as those of SS, but because it monitors $\frac{m}{2}$ keys, the hash map operations take a little less time than in the SS algorithm, even though they still take on the order of $O(\log(m))$ time. More importantly, however, is that FSS needs to do fewer such operations, since an observed key is not always added to the hash map. Thus, we would expect the FSS algorithm to have a noticeably smaller computation time than SS, which is confirmed by the experiments presented in Section 5.

The FSSA algorithm stores 5 numbers for each monitored key, and if its hash array is implemented using 2 bytes for each cell, then its total space consumption would be $20m + 3m = 23m$ bytes if it were to monitor $\frac{m}{2}$ keys. The maximum operational time for FSSA is $O(\log(m))$ because keys in the array are found using binary search.

If one's primary objective is not to miss any of the top n most frequent keys and one decides to add an additional data structure to FSS2L as described in Section 3.4, then the space consumption will increase by $\frac{3}{2}m \cdot \frac{9}{8} = \frac{27}{16}m$ bytes. We called the resulting algorithm Adaptive FSSA (AFSSA). Table 1 summarizes the space requirements for the algorithms evaluated in Section 5.

5 Experimental Results

5.1 Experiments on Artificial Data

We first evaluated SS, FSS, FSSA and AFSSA algorithms on artificial data sets with keys following Zipf and Exponential distributions. The Zipf(α) distribution was defined over 1 million keys, and then 1 million keys were randomly drawn from this distribution and processed by each algorithm (the nth most frequent key was drawn with probability $P(n) = K/n^{\alpha}$, where K is a constant that makes all probabilities add up to 1). The values of the skew parameter α were chosen to lie between 0.4 and 1.2, which covers the region of interest where detection of frequent items is possible (some skew is present) but is not super easy (the skew is not very large). The number of distinct keys observed in a data set of 10^6sampled keys ranged, correspondingly, between 600000 and 60000, approximately.

For the Exponential distribution, the probability of observing the nth most frequent key was given by $P(n) = a \cdot e^{-c \cdot n}$, where c is the skew parameter and a is a suitably chosen constant that makes all probabilities add up to 1 on the support [0, NumKeys]. For consistency with experiments performed with the Zipf distribution, NumKeys was chosen to be 1 million. The values of the skew parameter c were chosen to cover the range where detection of frequent items is possible but is not super easy, which corresponded to the range between 0.0003 and 0.0009. The number of distinct keys observed in a data set of 10^6 sampled keys ranged correspondingly, between 21000 and 8000, approximately. Therefore, when evaluating the frequent item detection algorithms for the Exponential distribution with c between 0.0003 and 0.0009, the SS list size m should be much smaller than 8000 in order to make the problem interesting. We chose $m = 1000$ for the experiments described below. In order to make a fair comparison between the algorithms, we set their space consumption (by varying the number of keys each of them is tracking) equal to that of SS with $m = 1000$. The hash array size for FSS and FSSA was set to be 3 times the number of keys each of them monitors, following the suggestion made by the authors of FSS in [5].

The accuracy of the algorithms was measured in several ways. One of them was the number of sequential true top n most frequent keys detected by the algorithm, which we called "topN." For example, if key 1 is the most frequent, key 2 is the next most frequent, and so on, and if the algorithm detects keys 1, 2, 3, 4, 6, … (i.e., it misses key 5), then topN equals 4. However, even if an algorithm detects the top 4 most frequent keys, it can still have very large frequency estimation errors for them. Therefore, in order to record this aspect of accuracy, we also computed the Mean Absolute Error for each algorithm for the top 750 keys (sorted by estimated frequency) on its list.

Tables 1-3 show the performance metrics described above, averaged over 50 trials (each cell shows estimated mean and standard error), for the algorithms discussed in this paper under the Zipf distribution of key frequencies. The value of the q parameter for the FSSA algorithm is given in parenthesis in the top row of each table. The last column corresponds to the AFSSA algorithm, where the q parameter was adjusted using the procedure described at the end of Section 3.4.

These tables show that FSS is more accurate than SS despite monitoring fewer keys. FSS also runs faster, confirming the theoretical analysis of its run time performed in Section 4. As expected, FSSA(0) is more accurate than FSS because both of them have a very similar logic (as was noted at the beginning of Section 3.4) but FSSA(0) is more space-efficient than FSS and hence it monitors more keys than FSS.

Tables 1-3 also show that for small levels of skew, when many distinct keys are observed, FSSA with q=0 detects a much shorter topN sequence than with $q > 0$. However, for $\alpha = 1.2$ the number of distinct keys observed becomes small enough for the dynamics described in Section 3.4 to manifest itself, and consequently FSSA(q) detects the longest topN sequence for small values of q. Despite this fact, FSSA(0.75) has the smallest MAE because whichever keys get into its List1 early on with small estimation errors tend to stay in that list (and hence their errors do not increase). The AFSSA algorithm increases q from 0.25 to 0.75 for $\alpha \leq 1$, but for $\alpha = 1.2$ it detects that the number of observed distinct keys is small enough and does not increase q from its initial value of 0.25. Despite adapting correctly to the shape of the distribution of keys, the performance of AFSSA is hampered by the fact that it uses more space than FSSA and hence is able to track fewer keys. As a result, it is not able to match the best-performing algorithm for each level of skew, but it still detects a longer topN sequence than FSSA(0.25) for the most interesting case of intermediate skew levels.

It is also interesting to observe that FSSA(0.75) and AFSSA (which are implemented using an efficient array implementation) have a smaller run time than FSS (which is implemented using Stream Summary) for $\alpha \leq 0.8$. The reason for this improvement is that whenever a new key needs to be inserted into the array of hash maps (described at the beginning of Section 3.3), FSS performs this operation on larger hash maps than FSSA(0.75) or AFSSA, which allocate only 1/4 of the monitored keys to List0.

For large degrees of skew, when $\alpha > 0.8$, insertions of new keys into the hash maps become more and more rare, and the run time of all algorithms decreases. SS algorithm runs much slower than the other algorithms for low degrees of skew because it adds a new key to the hash map whenever it is observed and is not in the hash map, while the other FSS-based algorithms add only those keys that get hashed to a cell in *H* with a high enough value.

Table 1. Length of the correctly detected topN sequence when keys are drawn from a Zipf(α) distribution

α	SS	FSS	FSSA(0)	FSSA(0.25)	FSSA(0.5)	FSSA(0.75)	AFSSA
0.4	**μ=0.1** s.e.=0.0	**μ=0.3** s.e.=0.1	**μ=0.5** s.e.=0.1	**μ=19** s.e.=0.9	**μ=19** s.e.=0.8	**μ=16** s.e.=0.8	**μ=16** s.e.=0.8
0.6	**μ=3.7** s.e.=0.2	**μ=20** s.e.=0.3	**μ=26** s.e.=0.5	**μ=80** s.e.=2	**μ=118** s.e.=2.7	**μ=145** s.e.=3.1	**μ=133** s.e.=2.6
0.8	**μ=33** s.e.=0.5	**μ=139** s.e.=0.8	**μ=168** s.e.=1	**μ=197** s.e.=2	**μ=250** s.e.=3	**μ=299** s.e.=4	**μ=277** s.e.=4
1	**μ=109** s.e.=0.9	**μ=359** s.e.=1	**μ=424** s.e.=2	**μ=424** s.e.=1	**μ=451** s.e.=2	**μ=471** s.e.=3	**μ=458** s.e.=3
1.2	**μ=209** s.e.=1	**μ=554** s.e.=1	**μ=636** s.e.=1	**μ=636** s.e.=1	**μ=636** s.e.=1	**μ=606** s.e.=2	**μ=602** s.e.=1

Table 2. Mean Absolute Error (MAE) when keys are drawn from a Zipf(α) distribution

α	SS	FSS	FSSA(0)	FSSA(0.25)	FSSA(0.5)	FSSA(0.75)	AFSSA
0.4	**μ=997** s.e.=0.0	**μ=291** s.e.=0.0	**μ=250** s.e.=0.0	**μ=199** s.e.=0.2	**μ=160** s.e.=0.3	**μ=101** s.e.=0.3	**μ=109** s.e.=0.3
0.6	**μ=984** s.e.=0.2	**μ=275** s.e.=0.2	**μ=233** s.e.=0.1	**μ=171** s.e.=0.1	**μ=101** s.e.=0.1	**μ=44** s.e.=0.2	**μ=52** s.e.=0.3
0.8	**μ=870** s.e.=0.6	**μ=190** s.e.=0.4	**μ=151** s.e.=0.3	**μ=135** s.e.=0.4	**μ=72** s.e.=0.2	**μ=10** s.e.=0.	**μ=17** s.e.=0.1
1	**μ=535** s.e.=0.9	**μ=70** s.e.=0.3	**μ=45** s.e.=0.4	**μ=44** s.e.=0.4	**μ=34** s.e.=0.3	**μ=2.8** s.e.=0.1	**μ=6.0** s.e.=0.1
1.2	**μ=192** s.e.=0.5	**μ=12** s.e.=0.1	**μ=3.8** s.e.=0.0	**μ=3.8** s.e.=0.0	**μ=3.8** s.e.=0.0	**μ=1.0** s.e.=0.0	**μ=6.7** s.e.=0.1

Table 3. Run time in ms when keys are drawn from a Zipf(α) distribution

α	SS	FSS	FSSA(0)	FSSA(0.25)	FSSA(0.5)	FSSA(0.75)	AFSSA
0.4	**μ=1313** s.e.=6	**μ=474** s.e.=2	**μ=535** s.e.=6	**μ=488** s.e.=2	**μ=440** s.e.=4	**μ=432** s.e.=3	**μ=420** s.e.=2
0.6	**μ=1283** s.e.=6	**μ=446** s.e.=1	**μ=503** s.e.=2	**μ=468** s.e.=2	**μ=418** s.e.=1	**μ=343** s.e.=1	**μ=328** s.e.=1
0.8	**μ=1141** s.e.=4	**μ=386** s.e.=3	**μ=436** s.e.=2	**μ=436** s.e.=2	**μ=390** s.e.=2	**μ=334** s.e.=2	**μ=330** s.e.=4
1	**μ=777** s.e.=3	**μ=244** s.e.=1	**μ=356** s.e.=2	**μ=343** s.e.=2	**μ=337** s.e.=2	**μ=315** s.e.=2	**μ=304** s.e.=2
1.2	**μ=365** s.e.=1	**μ=133** s.e.=1	**μ=257** s.e.=1	**μ=250** s.e.=2	**μ=246** s.e.=2	**μ=236** s.e.=2	**μ=238** s.e.=1

Tables 4-6 show how the considered algorithms perform when the frequency of each key is drawn from the Exponential distribution described at the beginning of this section. Table 4 shows that for a small degree of skew, FSSA(0.25) detects the longest topN sequence, which once again matches the analysis performed in Section 3.4.

Table 4. Length of the correctly detected topN sequence when keys are drawn from the Exponential distribution

c	SS	FSS	FSSA(0)	FSSA(0.25)	FSSA(0.5)	FSSA(0.75)	AFSSA
0.3	**μ=0.5** s.e.=0.1	**μ=2.2** s.e.=0.3	**μ=6.7** s.e.=0.8	**μ=11.5** s.e.=1.3	**μ=9.2** s.e.=1	**μ=4.8** s.e.=0.6	**μ=5.2** s.e.=0.4
0.5	**μ=0.7** s.e.=0.2	**μ=270** s.e.=2	**μ=501** s.e.=2	**μ=501** s.e.=2	**μ=493** s.e.=2	**μ=209** s.e.=8	**μ=395** s.e.=2
0.7	**μ=1.8** s.e.=0.3	**μ=512** s.e.=1	**μ=715** s.e.=1	**μ=716** s.e.=1	**μ=716** s.e.=1	**μ=549** s.e.=5	**μ=660** s.e.=1
0.9	**μ=10** s.e.=1	**μ=676** s.e.=1	**μ=744** s.e.=1	**μ=744** s.e.=1	**μ=744** s.e.=1	**μ=669** s.e.=3	**μ=736** s.e.=1

Table 5. Mean Absolute Error (MAE) when keys are drawn from the Exponential distribution

c	SS	FSS	FSSA(0)	FSSA(0.25)	FSSA(0.5)	FSSA(0.75)	AFSSA
0.3	**μ=822** s.e.=0.4	**μ=190** s.e.=0.5	**μ=129** s.e.=0.9	**μ=97** s.e.=0.5	**μ=52** s.e.=0.4	**μ=2.9** s.e.=0.1	**μ=119** s.e.=0.3
0.5	**μ=716** s.e.=0.7	**μ=79** s.e.=0.4	**μ=23** s.e.=0.2	**μ=23** s.e.=0.2	**μ=18** s.e.=0.1	**μ=2.0** s.e.=0.1	**μ=43** s.e.=0.3
0.7	**μ=626** s.e.=0.9	**μ=24** s.e.=0.2	**μ=2.15** s.e.=0.0	**μ=2.15** s.e.=0.0	**μ=2.15** s.e.=0.0	**μ=1** s.e.=0.1	**μ=6.55** s.e.=0.1
0.9	**μ=481** s.e.=2	**μ=6.5** s.e.=0.1	**μ=0.54** s.e.=0.0	**μ=0.54** s.e.=0.0	**μ=0.54** s.e.=0.0	**μ=0.48** s.e.=0.0	**μ=1.0** s.e.=0.0

Table 6. Run time in ms when keys are drawn from the Exponential distribution

c	SS	FSS	FSSA(0)	FSSA(0.25)	FSSA(0.5)	FSSA(0.75)	AFSSA
0.3	**μ=1061** s.e.=7	**μ=328** s.e.=2	**μ=402** s.e.=2	**μ=437** s.e.=2	**μ=409** s.e.=2	**μ=371** s.e.=3	**μ=413** s.e.=6
0.5	**μ=952** s.e.=3	**μ=260** s.e.=2	**μ=360** s.e.=2	**μ=407** s.e.=2	**μ=394** s.e.=2	**μ=387** s.e.=2	**μ=401** s.e.=3
0.7	**μ=826** s.e.=3	**μ=225** s.e.=1	**μ=346** s.e.=2	**μ=398** s.e.=2	**μ=402** s.e.=3	**μ=389** s.e.=2	**μ=378** s.e.=2
0.9	**μ=748** s.e.=4	**μ=209** s.e.=1	**μ=347** s.e.=2	**μ=405** s.e.=3	**μ=397** s.e.=2	**μ=393** s.e.=2	**μ=372** s.e.=2

When keys are sampled from the Exponential distribution, the number of distinct keys observed is not much greater than the size of the hash array H, which makes it difficult for medium-frequency keys to get into List0. Furthermore, when the degree of skew in the distribution of key frequencies decreases, it becomes more difficult for the nth most frequent key to differentiate itself from the less frequent keys, which exacerbates the dynamics described in Section 3.4 and causes FSSA(0.25) to detect longer topN sequences than FSSA(q) for $q > 0.25$ when $c = 0.0003$.

The AFSSA algorithm does not increase the q value from its initial setting of 0.25 because it detects that the data set does not have too many distinct keys and. As a result, AFSSA detects a longer topN sequence than FSSA(0.75). Comparing this with the previous experiments on the Zipf distribution, we see that AFSSA provides a robust mid-level performance and is able to avoid, through adaptation, the poor performance sometimes suffered by each fixed FSSA algorithm.

Another interesting observation is that the run time of AFSSA is smaller than that of FSS for a small degree of skew in the Zipf distribution, while for the Exponential distribution it is always larger than that of FSS. This happens because the Exponential distribution has a much smaller number of distinct keys and hence new keys are inserted into the hash map relatively less frequently, thus reducing the disadvantage of FSS due to its larger hash maps for List0. On the other hand, updates of existing keys happen relatively more frequently, and such updates require a binary search for the key in List1 of AFSSA, while FSS contains pointers in the List1 hash maps to each key in its Stream Summary.

5.2 Experiments on Real Data

We now present experimental results on the data sets chosen from the “Open-Source Data Mining Library” [9]. We selected four largest data sets with the largest number of distinct keys. The first data set was “Kosarak”, which contains anonymized click-stream data from a Hungarian news portal (8019015 keys, 41270 of which are distinct). The second data set was “Retail”, which contains anonymized retail market basket data from an anonymous Belgian retail store (908576 keys, 16470 of which are distinct). The third data set was “BMS2”, which contains anonymized click-stream and purchase data from Gazelle.com, a legwear and legcare web retailer that closed their online store on 8/18/2000. This data set was used in the KDD-Cup 2000 competition and it contains 358278 keys, 3340 of which are distinct. The fourth data set was “Bible”, where each key corresponds to a distinct word in the Bible (787066 keys, 13905 distinct).

Performance of the considered algorithms on these data sets is shown in Tables 7 - 9. Since the number of distinct keys in these data sets covered the range similar to that seen in the Exponential distribution, the observed results follow the same pattern as the one observed in Tables 4-6.

Table 7. Length of the correctly detected topN sequence for real data sets

	SS	FSS	FSSA(0)	FSSA(0.25)	FSSA(0.5)	FSSA(0.75)	AFSSA
Kosarak	178	567	743	743	743	722	711
Retail	114	495	618	618	618	357	581
BMS2	337	700	728	729	728	528	722
Bible	359	682	703	703	703	633	694

Table 8. Mean Absolute Error (MAE) for real data sets

	SS	FSS	FSSA(0)	FSSA(0.25)	FSSA(0.5)	FSSA(0.75)	AFSSA
Kosarak	2675	67	1.8	1.8	1.8	1.0	8.9
Retail	435	40	19	19	19	10	24
BMS2	82	6	3.3	3.2	3.2	2.5	3.9
Bible	94	5.3	3.0	3.0	3.0	2.0	3.6

Table 9. Run time in ms for real data sets

	SS	FSS	FSSA(0)	FSSA(0.25)	FSSA(0.5)	FSSA(0.75)	AFSSA
Kosarak	4424	1198	2153	2333	2356	2312	2316
Retail	589	167	287	330	322	299	306
BMS2	164	63	139	130	159	144	136
Bible	195	80	224	215	201	192	198

6 Conclusion

This paper presented a new 2-list framework (called FSS2L) for detecting the most frequent keys in a data set and for accurately estimating their frequencies. We also presented a space-efficient implementation of this framework using a single array FSSA(q) algorithm, where the parameter q controls the relative sizes of the two lists. We showed that FSSA(0) performs the same operations as the previously published FSS algorithm [5] but using less space than the "standard" implementation assumed in the literature (e.g., [1,2]) based on the Stream-Summary data structure. As a result, FSSA(0) gives more accurate results than FSS when their space consumption is equalized.

Dependence of FSSA on the parameter q was studied analytically and experimentally. Our results showed that if one is primarily interested in detecting most accurately the top n most frequent items, then one should use a small value of q (0.25 was shown to work well in our experiments) if the number of distinct keys in the data set is not much larger than the size of the hash array H used by FSS2L; otherwise, one should use a large value of q (0.75 was shown to work well in our experiments).

Accurate detection of the top n most frequent keys is the important accuracy measure when a distributed (or parallel) hash join needs to be performed by a database engine. In this case, in order to minimize the overall execution time, no single node (or process) should be overwhelmed by needing to process a join with a very frequent key. In order to achieve this goal, the most frequent items are detected by analyzing a sample of the data and then information about these items is broadcast to all nodes (or processes). The execution time of this scheme depends on the frequency of the first true most frequent item that is not detected by the algorithm, since processing of this item by a single node (or process) to which this item is hashed will be the bottleneck for the overall distributed join scheme. Therefore, the long sequence of the true top n consecutive most frequent items detected by the FSSA algorithm makes it an especially attractive algorithm for use with distributed hash joins.

We have also described an adaptive version of FSSA (called AFSSA), which starts with $q = 0.25$, then estimates the number of distinct keys in the data set, and then increases q to 0.75 if the number of distinct keys was found to be much larger than the size of H. Our experiments on both artificial and real data sets showed that AFSSA is able to adapt to the shape of the distribution of keys and avoid the poor performance sometimes suffered by fixed FSSA(0.25) and FSSA(0.75) algorithms. However, the smallest mean absolute error (MAE) in frequency estimates was consistently obtained by FSSA(0.75), and so if one is primarily interested in minimizing MAE while detecting keys that are reasonably frequent, then one should use FSSA with $q = 0.75$.

FSS2L algorithms can be naturally parallelized for use in a multi-process environment. This can be achieved by breaking up a large data set into P equal parts and then assigning each part i to be scanned by process p_{1i}. While scanning, each process p_{1i} hashes scanned keys into the range [0, $P \cdot R$], and then the keys that hash into [0, R] get sent to process p_{21}, those that hash into $[R, 2R]$ get sent to process p_{22}, etc. Each process p_{2j} then uses the same hash function to hash the incoming keys into its hash array of size R. As a result, the processes p_{2j} would work with disjoint sets of keys, and assuming that each of them uses as much memory as in the serial case, the accuracy guarantee $0 \leq \Delta_i \leq H[k] \leq f_{min} \leq \lfloor 2N/(m \cdot P) \rfloor$ would apply for the estimated frequency of each key i, with the factor P appearing in the denominator because each process p_{2j} is expected to observe only N/P keys. Then, at the end of the data set, a "*top n*" query would take the top *n* most frequent elements from each process, merge together the resulting P top *n* lists, and then return top *n* elements from the merged list. This parallelized version of FSSA has been integrated into the Oracle database.

References

1. Cormode G. and Hadjieleftheriou M.: Finding frequent items in data streams. VLDB Endowment 1(2), pp. 1530–1541 (2008).
2. Das, S., Antony S., Agrawal, D., El Abbadi, A.: Thread Cooperation in Multicore Architectures for Frequency Counting over Multiple Data Streams. VLDB Endowment 2(1), pp. 217-228 (2009).
3. Demaine E., L´opez-Ortiz A., and Munro J. I.: Frequency estimation of internet packet streams with limited space. In: Proceedings of the European Symposium on Algorithms (ESA), pp. 348-360 (2002).
4. P. Flajolet, E. Fusy, O. Gandouet, and F. Meunier.: Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. In: Proceedings of the 13th Conference on Analysis of Algorithm, pp. 127–146 (2007).
5. Homem N., Carvalho J.: Finding top-k elements in data streams. Information Sciences 180(24), pp. 4958–4974 (2010).
6. Manku G. and Motwani R.: Approximate Frequency Counts over Data Streams. In: Proceedings of 28th International Conference on Very Large Data Bases (VLDB), pp. 346-357, Morgan Kaufmann, Hong Kong, China (2002).
7. Manerikar N. and Palpanas T.: Frequent items in streaming data: An experimental evaluation of the state-of-the-art, Data & Knowledge Engineering 68(4), pp. 415-430 (2009)
8. Metwally A., Agrawal D., and Abbadi A. E.: Efficient computation of frequent and top-k elements in data streams. In: Proceedings of International Conference on Database Theory, pp. 398-412, Springer-Verlag, Edinburgh, Scotland (2005).
9. Open-Source Data Mining Library, http://www.philippe-fournier-viger.com/spmf/index.php?link=datasets.php