SSS: An Accurate and Fast Algorithm for Finding Top-k Hot Items in Data Streams

Junzhi Gong*, Deyu Tian*, Dongsheng Yang*, Tong Yang*†, Tuo Dai*, Bin Cui*, Xiaoming Li*
*Department of Computer Science, Peking University, China
†Collaborative Innovation Center of High Performance Computing, NUDT, China

***Abstract*—Finding top-k hot items in a data stream is a critical problem in big data management. It can benefit various kinds of applications, such as data mining, databases, network traffic measurement, security, *etc.* However, as the speed of data streams become increasingly large, it becomes more and more challenging to design an accurate and fast algorithm for this problem. There are several existing algorithms, including *Space-Saving*, *Frequent*, *Lossy counting*, with *Space-Saving* being the most widely used among them. Unfortunately, all these existing algorithms cannot achieve high memory efficiency and high accuracy at the same time. In this paper, we propose an enhanced algorithm based on Space-Saving, named *Scoreboard Space-Saving* (SSS), which not only achieves much higher accuracy, but also works at fast and constant speed. The key idea of SSS is to predict whether each incoming item is a hot item or not by scoring. Experimental results show that SSS algorithm achieves up to 62.4 times higher accuracy than Space-Saving. The source code of SSS is available at GitHub [1].**

***Index Terms*—Data Structures, Finding Top-k Items, Space-Saving**

I. Introduction

A. Background and Motivation

Finding top-k hot items[2] is a critical problem in big data management. It can benefit various kinds of applications, such as data mining [2]–[4], databases [5], network traffic measurement [6]–[8], security [9], and more [10]–[15]. For example, in order to achieve load balance [6] in the datacenter network, the administrators need to find the largest flows (elephant flows) in the network traffic. For another example, in order to find the closest friends in the social network platform [13], the service providers need to find those people who have the most interactions with each user.

In many real data streams, the distribution of items' frequencies is highly skewed [16]–[20]. In other words, most items have a small frequency, while only a few items have a large frequency, and we call them *cold items* and *hot items*, respectively.

*Corresponding author: Tong Yang (Email: yang.tong@pku.edu.cn). This work was done by Junzhi Gong, Yang Zhou, and Dongsheng Yang under the guidance of their mentor: Tong Yang.

This work is partially supported by Primary Research & Development Plan of China (2016YFB1000304), National Basic Research Program of China (2014CB340400), NSFC (61472009, 61672061), the Open Project Funding of CAS Key Lab of Network Data Science and Technology, Institute of Computing Technology, Chinese Academy of Sciences.

[2]A hot item in a data stream refers to an item with a large frequency, while the frequency of an item refers to the number of times this item occurs in the data stream.

Finding top-k hot items is a challenging issue, although this problem has been studied for years [19]–[25]. Due to the vast size and high speed of data streams, it is very difficult to accurately record the information of each item. Therefore, approximate solutions gain wide acceptance. Sampling is one of the most widely used solution, but the accuracy is quite low. To enhance accuracy, the state-of-the-art algorithms process each item in data streams. Because of the high speed of data streams, these algorithms should achieve a fast and constant update speed. Due to the high latency of the slow memory such as DRAM (Dynamic RAM), the algorithms should not access the slow memory, but only access the fast memory such as SRAM (Static RAM) [26] whose size is tight. However, these algorithms cannot achieve high accuracy when memory is tight.

B. Limitation of Prior Art

Traditional solutions for finding top-k hot items can be divided into two categories: *sketch-based* and *counter-based*. *1) Sketch-based solutions* use sketches[3] (*e.g.*, the Count-min sketch [18], or the Count sketch [21]) to record frequencies of all items, and use a min-heap to report top-k hot items. The limitation of these algorithms is that *they record information of all cold items.* Such information is useless and harmful for finding top-k items, and requires additional memory usage. *2) Counter-based solutions* include *Space-Saving* [27], *Frequent* [28], [29], *Lossy counting* [30], *etc.* The key idea of these solutions is to treat each incoming item e_i as a hot item – assign e_i with a very large frequency, and then insert e_i with its large frequency into the top-k data structure (*e.g.*, Space-Saving uses the Stream-Summary as the top-k data structure). As time goes by, it gradually expels cold items out of the top-k data structure with a certain probability. However, in real data streams, most items are cold items, and processing so many cold items not only incurs extra overhead, but also causes significant inaccuracy for the ranking and frequency estimation of top-k items. In summary, both types of solutions do not handle cold items elegantly, and thus they cannot achieve high accuracy in real data streams. The design goal of this paper is to separate hot items from cold items in real data streams, so as to achieve high accuracy in finding top-k items.

[3]A sketch is a probabilistic data structure to store the frequency of items in a multiset.

C. Proposed Approach

In this paper, we propose a novel algorithm based on Space-Saving, named the *Scoreboard Space-Saving* (SSS), which achieves higher accuracy than existing algorithms. Besides using the Stream-Summary, the SSS algorithm adds a tiny queue and a **Scoreboard**. *The key idea of SSS is inserting only the hot items into Stream-Summary instead of inserting all items.* Using the queue and the Scoreboard, we can monitor recent incoming items, and compute each incoming item a score according to the information in the queue and the Scoreboard. The score indicates whether an item is a hot item or not. The queue can be quite short, which usually occupies less than 10% memory usage of the Stream-Summary. The Scoreboard can be any kinds of data structures, as long as it is capable of approximately recording frequencies of items in a multiset. We recommend using the Counting Bloom filter (CBF) [31] as the Scoreboard, because it is memory efficient and supports both insertion and deletion operations. More details are provided in Section II-A.

We briefly introduce how SSS works as follows. We divide all items into three types: cold items, potential hot items, and hot items. We also set two score thresholds ($\mathcal{B}$ and $\mathcal{E}$) to determine which type each item belongs to. For each item, we first compute its score $\mathcal{S}$ from the Scoreboard. If $\mathcal{S} \geqslant \mathcal{E}$, the item is a hot item, and we insert it into the Stream-Summary. If $\mathcal{B} \leqslant \mathcal{S} < \mathcal{E}$, the item is a potential hot item, and we increase its score in the Scoreboard. If $\mathcal{S} < \mathcal{B}$, then the item is a cold item, and we not only increase its score in the Scoreboard, but also insert it into the tiny queue to give it a chance to accumulate. Once it is expelled from the queue, if its score is still smaller than $\mathcal{B}$, then it fails to accumulate, and its recorded information in the queue and the Scoreboard will be removed. Otherwise, it becomes a potential hot item, and we keep its score unchanged. In this way, we keep the Stream-Summary unaffected by the cold items, and also we do not need much memory to record the cold items.

Key technique challenge: The key technique challenge of SSS is how to set appropriate thresholds. With more and more items arriving, the frequency, rank of top-k hot items are changing. Therefore, thresholds should be dynamically adjusted to achieve accurate classifications. To address this issue, we assume that the data stream obeys Zipfian distribution [32], [33], use history data to learn the parameters of the distribution by a *machine learning* program, and set thresholds based on the distribution function.

D. Key Contributions

1) First, we propose an enhanced algorithm based on Space-Saving, named Scoreboard Space-Saving (SSS), which achieves higher accuracy compared to the state-of-the-art and works at fast insertion speed.
2) Second, we make mathematical analysis and conduct a series of experiments. The experimental results show that SSS outperforms the state-of-the-art algorithm (Space-Saving).

II. Background and Related Work

A. Counting Bloom Filter

A Counting Bloom filter (CBF) consists of an array B with w counters, and it is associated with t hash functions $g_1(.), g_2(.) \ldots g_t(.)$. When inserting an item e, it first computes the t hash functions and maps e to k corresponding counters $B[g_1(e)\%w], B[g_2(e)\%w] \ldots B[g_t(e)\%w]$. Then it increments these t counters by 1. When querying an item e, it just reports the minimum value of the t mapped counters. Note that the CBF does not suffer from *under-estimation* errors. The structure of the CBF is shown in Figure 1. The CBF is usually very memory efficient so that it can record the frequencies for many items with small memory usage.

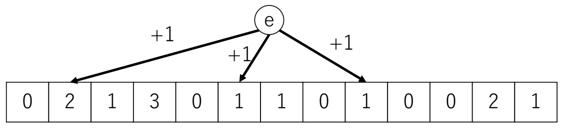

Fig. 1. The structure of the Counting Bloom filter.

B. Existing algorithms

In this section, we present the details of existing algorithms. Existing algorithms for finding top-k hot items can be divided into two categories: sketch-based algorithms and counter-based algorithms. Sketch-based algorithms use a sketch (such as the Count sketch [21] or the Count-min sketch [18]) and a min-heap. Given an incoming item e_i, if it has already been recorded in the min-heap, they increment the corresponding frequency by 1. If not, e_i will be inserted into the sketch, and at the same time the sketch reports an estimated value $\hat{s}_i$ for e_i. If $\hat{s}_i$ is larger than the frequency of the item at the root node of the min-heap, they make a swap: expelling the item at the root node of the min-heap into the sketch, and insert $< e_i, \hat{s}_i >$ into the root node; otherwise, the insertion ends.

Traditional counter-based algorithms include *Frequent* [28], [29], *Lossy counting* [30], *Space-Saving* [27], *etc.*, with Space-Saving being the most well-known of them. The key data structure of Space-Saving is called *Stream-Summary*, a combined data structure of a hash table and a min-heap. To improve accuracy, the Stream-Summary structure in Space-Saving stores m ($m \gg k$) hot items and their frequencies, and only reports the largest k items. The main benefit of Stream-Summary is that it can search and update an item in $O(1)$ time, and can also find the item with the smallest frequency in Stream-Summary in $O(1)$ time. The specific implementation of Stream-Summary is an ordered linked list with m buckets (Due to space limitations, details are omitted). The insertion process of Space-Saving is as follows: 1) If the incoming item e_i is in the Stream-Summary, it increments the corresponding frequency; 2) Otherwise, it locates and deletes the item with the smallest frequency $\hat{s}_{min}$. Then, it inserts $< e_i,\ \hat{s}_{min} + 1 >$. Therefore, it considers that the frequency of each new item is higher than the current smallest frequency in Stream-Summary.

In summary, both two kinds of algorithms cannot handle cold items elegantly, and those cold items in data streams can significantly reduce the accuracy.

III. The Scoreboard Space-Saving Algorithm

A. Rationale

To overcome the shortcomings of the Space-Saving algorithm, we propose an enhanced algorithm based on Space-Saving, named Scoreboard Space-Saving (SSS). *The key idea of SSS is inserting only the hot items into Stream-Summary instead of inserting all items.* SSS uses a queue and a Scoreboard as a classifier to identify whether an incoming item is likely to be a hot item. The Scoreboard can be any kinds of data structure as long as it can approximately store the frequencies of items in a multiset. In this paper, we use the Counting Bloom filter (CBF) as the Scoreboard, because it achieves high accuracy and high memory efficiency.

Here we briefly introduce how the queue and the Scoreboard help to identify hot items. We divide all the items into three types: cold items, potential hot items and hot items. Moreover, we use the Scoreboard to maintain a score for each item, and also set two threshold values to identify which type an item belongs to. For each incoming item, we first get its score from the Scoreboard and judge which kind of item it belongs to. We insert hot items into Stream-Summary, and insert other items into the Scoreboard. For cold items, we also insert them into the queue to give them a chance to accumulate. When an item is expelled from the queue after a period of time, if it is still a cold item, *i.e.*, the item fails to accumulate, then its score is deleted from the Scoreboard. Otherwise, its score is unchanged. As a result, cold items are only passers-by in the Scoreboard. Therefore, the Scoreboard can be very memory efficient because it only holds hot items and potential hot items, which account for a very small part of all items in real data streams.

B. Definitions

Here we introduce some useful definitions which help to illustrate our algorithm in detail.

Score $\mathcal{S}$: For convenience, we use $\mathcal{S}_e$ to denote the score of item e. When we use the CBF as the Scoreboard, $\mathcal{S}_e$ represents the estimate frequency of item e in the CBF.

Elephant value $\mathcal{E}$: The elephant value is used to identify whether an item is a hot item. When $\mathcal{S}_e \geqslant \mathcal{E}$, we treat e as a hot item.

Base value $\mathcal{B}$: The base value is used to identify whether an item is a potential hot item (items with a huge probability of being a hot item in the future). When $\mathcal{B} \leqslant \mathcal{S}_e < \mathcal{E}$, we treat e as a potential hot item.

C. Insertion and Query Process

In this part, we present the insertion and query processes of SSS (see pseudo-code in Algorithm 1). First, we introduce the insertion process of SSS in detail. Assume that there is an incoming item e. We first check whether e is already in Stream-Summary. If so, we directly increment the frequency of e in Stream-Summary. Otherwise, we then get the score of e ($\mathcal{S}_e$) from the Scoreboard, and we perform different operations depending on $\mathcal{S}_e$ (See Figure 2).

Case 1: If $\mathcal{S}_e \geqslant \mathcal{E}$, then e is thought to be a hot item. Therefore, we replaces e_{min} with e in Stream-Summary, and then increments its size by 1. Here e_{min} represents the item with minimum frequency in Stream-Summary.

Case 2: If $\mathcal{B} \leqslant \mathcal{S}_e < \mathcal{E}$, we treat e as a potential hot item. Then we insert e into the Scoreboard to increase its score, and we do not push e into the queue.

Case 3: If $\mathcal{S}_e < \mathcal{B}$, then e is thought to be a cold item. Then we not only insert it into the Scoreboard to increase its score, but also push e into the queue.

When inserting an item into the queue, we also expel the item in the head of the queue (assume that the item is e'). Moreover, we get the score of e' ($\mathcal{S}_{e'}$), and also perform different operations on it depending on its score.

Case 1: If $\mathcal{S}_{e'} \geqslant \mathcal{B}$, then we regard e' as a hot item or a potential hot item. We do not delete it from the Scoreboard in this case.

Case 2: If $\mathcal{S}_{e'} < \mathcal{B}$, then e' is thought as a cold item. We then delete it from the Scoreboard to decrease its score to remove the influence of cold items.

Next, we introduce the query process of SSS. The query process can be divided into two steps: flushing and reporting.

Flushing: Before querying the top-k hot items, we first expel all items out of the queue, and use the same algorithm mentioned above to handle them.

Reporting: We simply report the first k items with the largest frequencies in the Stream-Summary structure. Moreover, we add $\mathcal{E}$ to the frequency of each item, because these items are not inserted into Stream-Summary when its score is smaller than $\mathcal{E}$.

Algorithm 1: The insertion process of SSS.

```
Input: An incoming item $e$, the Stream-Summary
       structure $SS$, the queue $q$
1  if $e \in SS$ then
2      $SS[e]++$;
3  else
4      if $\mathcal{S}_e \geqslant \mathcal{E}$ then
5          $SS.replace(e_{min}, e)$;
6          $SS[e]++$;
7      if $\mathcal{S}_e < \mathcal{E}$ && $\mathcal{S}_e \geqslant \mathcal{B}$ then
8          $\mathcal{S}_e++$;
9      if $\mathcal{S}_e < \mathcal{B}$ then
10         $\mathcal{S}_e++$;
11         $q.push(e)$;
12         $e' \leftarrow q.pop()$;
13         if $\mathcal{S}_{e'} < \mathcal{B}$ then
14             $\mathcal{S}_{e'}--$;
```

D. Constant Base Value and Dynamic Elephant Value

Selecting an appropriate elephant value $\mathcal{E}$ and a base value $\mathcal{B}$ is important. The base value $\mathcal{B}$ is used to filter cold items

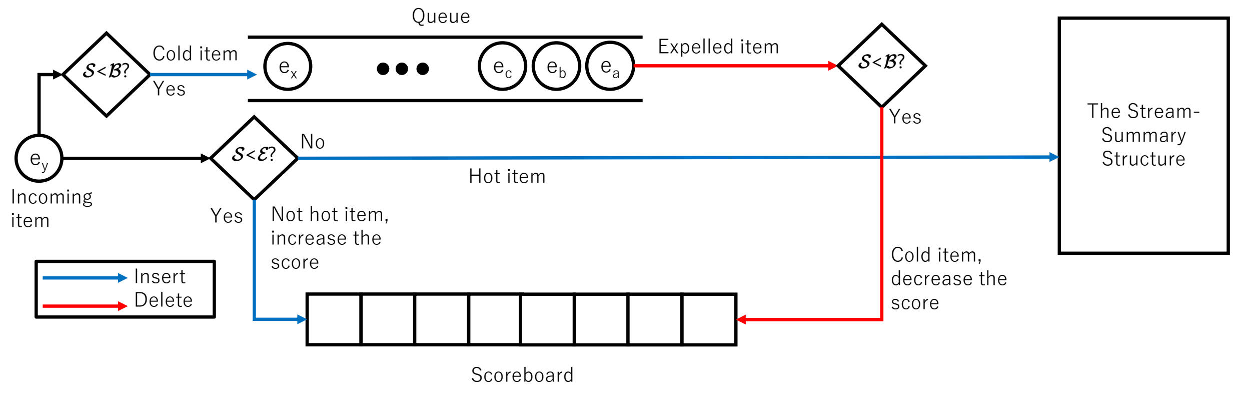
Queue
Cold item
Yes
$\mathcal{S}<\mathcal{B}$?
e_x
e_c
e_b
e_a
Expelled item
$\mathcal{S}<\mathcal{B}$?
Yes
The Stream-Summary Structure
e_y
Incoming item
$\mathcal{S}<\mathcal{E}$?
No
Hot item
Yes
Not hot item, increase the score
Cold item, decrease the score
Insert
Delete
Scoreboard

Fig. 2. The structure and algorithm of SSS.

and identify potential hot items in the data stream. The base value does not need to be too large, since in real data streams, there are usually about half of items that have a frequency of 1. But the base value should also not be too small, because a small base value has a poor performance of filtering cold items. In our implementation, we simply set the base value $\mathcal{B} = 10$, a number that we discover performing well in most cases.

The elephant value $\mathcal{E}$ is used to identify hot items, so it is directly related to the performance of SSS. It is obvious that the elephant value should be adjusted dynamically to the total number of items N and the distribution of the data stream. For the typical data streams in the real world, most items have a small frequency while only a few items have a huge frequency. The distribution of the data stream is similar to the Zipfian distribution [32], [33], and thus we can set the elephant value based on the Zipfian distribution. We set the elephant value $\mathcal{E}$ based on a function of the total number of items N and the number of buckets in Stream-Summary m. The function $\mathcal{E}(N, m)$ is:

$$\mathcal{E}(N, m) = \frac{CN}{m^a}$$

where C and a are two constant numbers, and we set the elephant value $\mathcal{E} = \lceil \mathcal{E}(N, m) \rceil$. Note that the distribution of the data stream can usually keep consistent, thus we can set C and a based on the history data stream. Therefore, we can use a *machine learning* program to learn the two parameters C and a using a certain machine learning algorithm. Note that the total number of items must be larger than the elephant value, *i.e.*, $\mathcal{E}(N, m) \leqslant N$ in any cases, and therefore, we have $m^a \geqslant C$. Here we present how we learn C and a based on the history data stream (see pseudo-code in Algorithm 2).

Sampling: The first step is to get items in the history data stream. Here we can only sample the history data stream instead of collecting all items because of the high speed of data streams. We use σ (*e.g.*, $1\% \sim 10\%$) to represent the sample rate.

Exact Counting: In order to get the frequency of each sampled item, each sampled item is inserted into a small hash table. To be specific, in the hash table, the key is the item ID and the value is its frequency.

Test Points: A key element of the machine learning program is the test set. Based on the function $\mathcal{E}(N, m)$, the test set is a set of three-tuple $< N_i, m_i, \mathcal{E}_i >$. To provide the test set for the machine learning program, we need to set some test points. Assume that the size of the history data stream is N_h. We split the history data stream into b blocks, and the sizes of those b blocks are equal to each other. At the end of the i^{th} ($1 \leqslant i \leqslant b$) block, for each sampled item, we insert the three-tuple $< \frac{N_h i}{b}, m_j, s_j >$ into the test set, where s_j is the frequency of the item and m_j is its rank.

Model Selection and Learning: We use the *linear regression* model [34] in the machine learning program. Then the program learns the value C and a, which can be used in setting the elephant value $\mathcal{E}$ for the latter data stream.

Algorithm 2: The machine learning program.

```
Input: Sampling rate $\sigma$, the number of blocks $b$, the size
       of the history data stream $N_h$, the hash table $T$
1  $N \leftarrow 0$;
2  $\Omega \leftarrow \Phi$;
3  for $i \leftarrow 1$ to $N_h$ do
4      if $i \% \frac{1}{\sigma} = 0$ then
5          $e_i \leftarrow Sample()$;
6          $T[e_i]++$;
7          $N++$;
8      if $i \% \frac{N_h}{b} = 0$ then
9          for $e_j \in T$ do
10             $m_j \leftarrow Rank(e_j)$;
11             $\Omega.insert(< N, m_j, T[e_j] >)$;
12 $TargetFunction \leftarrow Function(\frac{CN}{m^a})$;
13 $< C, a > \leftarrow LinearRegression(TargetFunction, \Omega)$;
14 return $< C, a >$;
```

IV. Mathematical Analysis

In this part, we make mathematical analysis on SSS algorithm. The analysis shows that SSS inherits some strong guarantees from Space-Saving.

Lemma IV.1. *For any item e, if the final score of e is $\mathcal{S}_e$, then there are totally at least $\mathcal{S}_e$ items that are inserted into the Scoreboard but are not inserted into Stream-Summary.*

Proof. When an incoming item is inserted into Stream-Summary, it will not be inserted into the Scoreboard. But when the item is not inserted into Stream-Summary, it will be inserted into the Scoreboard to increase its score. Therefore, when the item e has a score of $\mathcal{S}_e$, there are at least $\mathcal{S}_e$ items that are not inserted into Stream-Summary, but are inserted into the Scoreboard.

□

Theorem 1. *The total number of items N in the data stream is larger than or equal to the sum of estimate frequencies in Stream-Summary plus $\mathcal{E}^*$:*

$$N = \sum_i N_i \geqslant \sum_j \hat{s}_j + \mathcal{E}^*$$

where N_i represents the frequency of i^{th} item, $\hat{s}_j$ represents the estimate frequency stored in the j^{th} bucket in Stream-Summary, and $\mathcal{E}^*$ represents the final elephant value.

Proof. An item is inserted into Stream-Summary if and only if its score $\mathcal{S}$ in the Scoreboard is larger than or equal to $\mathcal{E}$. For convenience, we use η to represent the number of items that are not inserted into Stream-Summary, and we call the highest score among all items $\mathcal{S}_h$. We have $\eta \geqslant \mathcal{S}_h$ based on Lemma IV.1. Next we prove Theorem 1 by discussing the relationship of $\mathcal{S}_h$ and $\mathcal{E}^*$.

1) If $\mathcal{S}_h = \mathcal{E}^*$, we have:

$$\sum_j \hat{s}_j \leqslant N - \eta$$

with equality if and only if no item is deleted from the Scoreboard. Therefore, we have:

$$N = \sum_i N_i \geqslant \sum_j \hat{s}_j + \eta \geqslant \sum_j \hat{s}_j + \mathcal{E}^*$$

2) If $\mathcal{S}_h < \mathcal{E}^*$, then no item is inserted into Stream-Summary after $\mathcal{E}$ became larger than $\mathcal{S}_h$, because the score of all items is less than the elephant value. We use δ to represent the total number of items when $\mathcal{E} > \mathcal{S}_h$, and use N' to represent the total number of items when $\mathcal{E} \leqslant \mathcal{S}_h$. Based on the function $\mathcal{E}(N, m)$, we have:

$$N' = \frac{m^a(\mathcal{S}_h + 1)}{C}$$

and we also have:

$$N \geqslant \frac{m^a \mathcal{E}^*}{C}$$

Therefore, we have:

$$\delta = N - N' + 1 \geqslant \frac{m^a(\mathcal{E}^* - \mathcal{S}_h - 1)}{C} + 1$$

Based on Lemma IV.1, we have:

$$\sum_j \hat{s}_j \leqslant N - \eta - \delta$$

and we also have $m^a \geqslant C$, then:

$$\begin{aligned} N = \sum_i N_i &\geqslant \sum_j \hat{s}_j + \eta + \delta \\ &\geqslant \sum_j \hat{s}_j + \mathcal{S}_h + N - N' + 1 \\ &\geqslant \sum_j \hat{s}_j + \mathcal{S}_h + \frac{m^a(\mathcal{E}^* - \mathcal{S}_h - 1)}{C} + 1 \\ &\geqslant \sum_j \hat{s}_j + \mathcal{S}_h + (\mathcal{E}^* - \mathcal{S}_h - 1) + 1 \\ &= \sum_j \hat{s}_j + \mathcal{E}^* \end{aligned}$$

□

Theorem 2. *The minimum estimate frequency μ among buckets in the Stream-Summary structure is less than or equal to $\lfloor \frac{N-\mathcal{E}^*}{m} \rfloor$, where N represents the total number of items in the data stream, and m represents the number of buckets in the Stream-Summary structure.*

Proof. It is obvious that the frequency stored in any bucket of the Stream-Summary structure is larger than or equal to μ ($\hat{s}_j \geqslant \mu$). According to Theorem 1, we have:

$$\begin{aligned} N = \sum_i N_i &\geqslant \sum_j \hat{s}_j + \mathcal{E}^* \\ &= \sum_j (\hat{s}_j - \mu) + \sum_j \mu + \mathcal{E}^* \\ &= \sum_j (\hat{s}_j - \mu) + m\mu + \mathcal{E}^* \end{aligned}$$

and $\hat{s}_j \geqslant \mu$, then:

$$\mu = \frac{N - \sum_j (\hat{s}_j - \mu) - \mathcal{E}^*}{m} \leqslant \frac{N - \mathcal{E}^*}{m}$$

and μ is an integer, so we have:

$$\mu \leqslant \lfloor \frac{N - \mathcal{E}^*}{m} \rfloor$$

□

Definition IV.1. *Assume that the item e has been deleted from the Scoreboard for κ_e times. We define that κ_e is the* ***frequency loss*** *of item e.*

Theorem 3. *For an item e with a frequency of s_e, if*

$$s_e - \kappa_e - \mathcal{E}^* > \mu$$

then e must exist in the Stream-Summary.

Proof. When item e arrives for the last time, there are the following two cases:

Case 1: e is in by the Stream-Summary. Then the estimated frequency of e in the Stream-Summary is $s_e - \kappa_e - \mathcal{E}^*$. Because $s_e - \kappa_e - \mathcal{E}^* > \mu$, item e will never be expelled out of the Stream-Summary.

Case 2: e is not in the Stream-Summary. Because $s_e - \kappa_e - \mathcal{E}^* > \mu$, e is treated as a hot item. Therefore, e is inserted into the Stream-Summary, and will never be expelled out of the Stream-Summary.

□

Therefore, based on Theorem 2 and Theorem 3, if $s_e - \kappa_e - \mathcal{E}^* > \lfloor \frac{N-\mathcal{E}^*}{m} \rfloor$, then e must exist in the Stream-Summary.

V. Experimental Results

In this section, we present experimental results of SSS and Space-Saving. Space-Saving is proved to achieve higher accuracy than *Frequent* and *Lossy counting* [27], so we do not compare SSS with these two algorithms. The source code of CSS was provided by the author [35], and is written in Java, which is much slower than that written in C++. Therefore, we also do not compare SSS with CSS.

A. Experimental Setup

Our experimental programs are running on a machine with dual 6-core CPUs (24 threads, Intel Xeon CPU E5-2620 @2GHz) and a 62GB DRAM memory. The datasets used in our experiments are real IP packets captured from our campus, and we identify each packet with its five-tuple: source IP address, destination IP address, source port, destination port, and protocol type. The total number of items N is 10M, and there are 1.1M distinct items. To evaluate the performance of SSS and Space-Saving, we write a C++ program. *To achieve a head-to-head comparison, we guarantee that the memory usage of SSS is no larger than that of Space-Saving.*

B. Evaluate Metrics

Precision: It indicates how many estimate answers are correct. The PRecision (PR) is defined as:

$$PR = \frac{|\overline{D}_k \cap \hat{D}_k|}{|\hat{D}_k|}$$

where $\overline{D}_k$ represents the set of real top-k hot items, and $\hat{D}_k$ represents the set of estimate answers.

Average Ranking Error: The Average ranKing Error (AKE) is defined as:

$$AKE = \frac{1}{|\overline{D}_k|} \sum_{i=1}^{|\overline{D}_k|} |\hat{r}_{e_i} - r_{e_i}| \ (\hat{r}_{e_i} = |\overline{D}_k| + 1 \text{ if } e_i \notin \hat{D}_k)$$

where e_i represents the i^{th} item, $\hat{r}_{e_i}$ represents the rank of e_i in $\hat{D}_k$, and r_{e_i} represents the rank of e_i in $\overline{D}_k$.

Average Relative Error: The Average Relative Error (ARE) is defined as:

$$ARE = \frac{1}{|\hat{D}_k|} \sum_{i=1}^{|\hat{D}_k|} \frac{|\hat{s}_{e_i} - s_{e_i}|}{s_{e_i}}$$

where e_i $(1 \leqslant i \leqslant |\hat{D}_k|)$ represents the i^{th} item in $\hat{D}_k$, $\hat{s}_{e_i}$ represents the estimate size of e_i and s_{e_i} represents the real size of e_i.

Average Absolute Error: The Average Absolute Error (AAE) is defined as:

$$AAE = \frac{1}{|\hat{D}_k|} \sum_{i=1}^{|\hat{D}_k|} |\hat{s}_{e_i} - s_{e_i}|$$

C. Precision

In this part, we focus on the precision of SSS and the Space-Saving (SS) algorithm. We measure the precision of both algorithms with different k and different m (the number of buckets in Stream-Summary). In the experiment of varying k, we set $m = k$ for SSS and set $m = 2k$ for Space-Saving, and k ranges from 200 to 1000. In the experiment of varying m, we set $k = 400$, and we make m range from k to $2k$ for SSS, and from $2k$ to $3k$ for Space-Saving. For both experiments, we set the queue length of SSS to k, and set the width w of the CBF to $30k$.

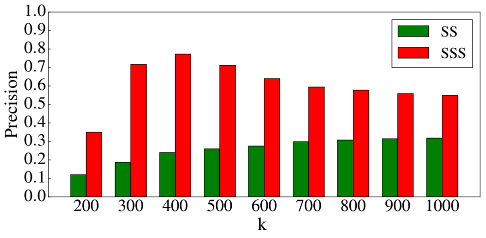

Fig. 3. Precision of top-k hot items when varying k.

Precision vs. k: *Our experimental results show that SSS is [1.73, 3.83] times more accurate than Space-Saving. To be specific, when $k = 400$, the precision of SSS reaches 77.25% while that of Space-Saving is only 24.00%.* As shown in Figure 3, the precision of SSS is always obviously higher than that of Space-Saving. When varying k from 200 to 1000, the precision of SSS is always higher than 35%, and is higher than 55% for most cases. On the contrary, the precision of Space-Saving is always lower than 32%. In a word, the precision of SSS is much higher than that of Space-Saving, especially when k is not large.

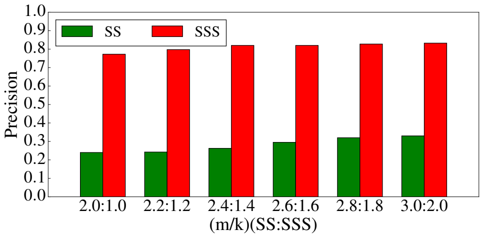

Fig. 4. Precision of top-k hot items when varying m.

Precision vs. m: *Our experimental results show that SSS is [2.52, 3.29] times more accurate than Space-Saving. To be specific, as m increases, the precision of SSS exceeds 80%, while that of the Space-Saving is only 33% when $m = 3k$.* The experimental results are shown in Figure 4. When increasing m, the precision of both algorithms increases slightly, but SSS always has a great advantage over Space-Saving.

D. Accuracy of Ranking

In this part, we focus on the average ranking error of both two algorithms. The ranking error reveals the accuracy of the estimate ranks of the top-k hot items. We have conducted experiments varying k and m. We also present how the ranks distributed by plotting a scatter diagram.

Average ranking error vs. k: *Our experimental results show that the average ranking error of SSS is [1.23, 2.81] times lower than that of Space-Saving. The difference between the average ranking error of both algorithms ranges from 33.79 to 99.38.* The experimental results are shown in Figure 5. To be specific, when $k = 300$, the average ranking error of SSS is only 46.81, while that of Space-Saving is more than 130.

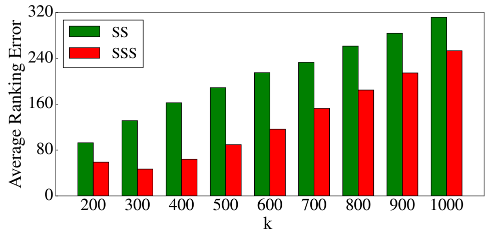

Fig. 5. Average ranking error of top-k hot items when varying k.

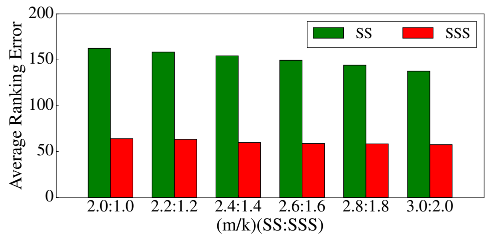

Fig. 6. Average ranking error of top-k hot items when varying m.

Average ranking error vs. m: *Our experimental results show that the average ranking error of SSS is [2.39, 2.58] times lower than that of Space-Saving. The difference between the average ranking error of both algorithms ranges from 80.1 to 98.5.* As shown in Figure 6, the average ranking error of both algorithms decreases as m increases, because more buckets in the Stream-Summary structure lead to higher accuracy. Nevertheless, SSS achieves higher accuracy of ranking than Space-Saving.

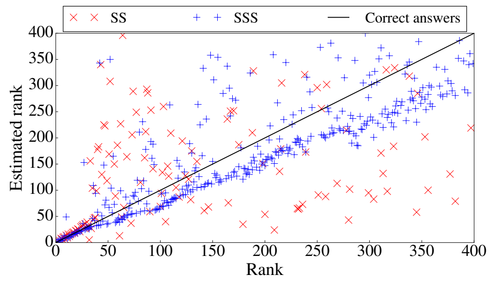

Fig. 7. Distribution of items' ranks of both algorithms.

Distribution of items' ranks: *Our experimental results show that the ranks provided by SSS are more accurate than that of Space-Saving.* As shown in Figure 7, there are more points of SSS than Space-Saving, which means that the precision of SSS is higher. Furthermore, the points of SSS are closer to the correct answer than Space-Saving, which shows that the ranks given by SSS are more accurate.

E. Accuracy of Estimate Sizes

In this part, we focus on the accuracy of item frequencies in the Stream-Summary structure. We mainly focus on the following four metrics to evaluate the accuracy: 1) Absolute error (AE) of each item e, which is defined as $|\hat{s}_e - s_e|$; 2) Average absolute error (AAE); 3) Relative error (RE) of each item, which is defined as $|\hat{s}_e - s_e|/s_e$; 4) Average relative error (ARE). Because the number of buckets m is different between the two algorithms (let $m = k$ for SSS and $m = 2k$ for Space-Saving), we only focus the accuracy of the first k items.

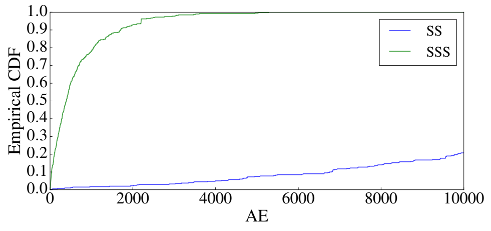

Fig. 8. Empirical CDF of absolute error.

Empirical CDF of AE: *Our experimental results show that more than 90% items have an AE less than 1700 for SSS, while for Space-Saving, there are only 2%.* To be specific, we set $k = 400$ in this experiment. The results are shown in Figure 8. As AE increases, the empirical CDF of SSS increases significantly, while it keeps at a very low value for Space-Saving. Furthermore, for SSS, the CDF reaches 100% when AE reaches 5285, but for the Space-Saving, the CDF reaches 100% when AE reaches 12453. The results indicate that for Space-Saving, most items suffer from great error, while for SSS, the estimated frequencies of most items are accurate.

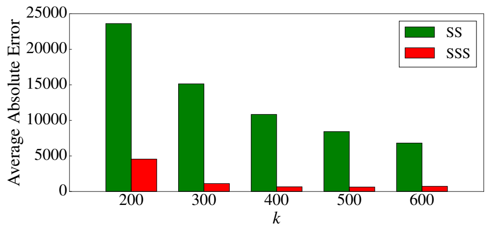

Fig. 9. Average absolute error of top-k hot items when varying k.

Average Absolute Error: *Our experimental results show that the AAE of SSS is [5.18, 16.52] times lower than that of Space-Saving, with different k ranging from 200 to 600.* The results of this part of the experiment are shown in Figure 9. To be specific, the AAE of SSS is only 624.5 when $k = 500$, while for Space-Saving, the AAE is always higher than 8400.

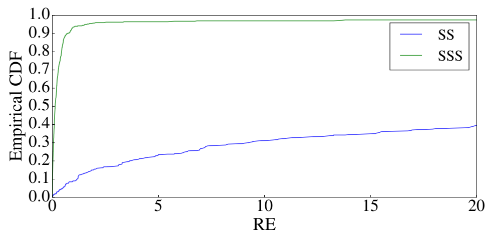

Fig. 10. Empirical CDF of relative error.

Empirical CDF of RE: *Our experimental results show that more than 90% items have an RE less than 0.69 for SSS, while for Space-Saving, there are only 7.75%. More than 95% items have an RE less than 1.53 for SSS, while for the Space-Saving the percentage is only 13.25%.* We also set $k = 400$ in this experiment. As shown in Figure 10, as RE increases, the empirical CDF increases significantly for SSS, and reaches nearly 100% when RE is still smaller than 10. While for Space-Saving, the CDF increases slowly as RE increases from 0 to 100, with a value of only 39.5% when RE reaches 20.

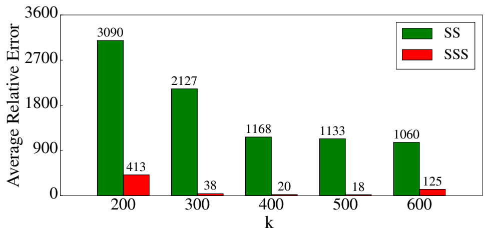

Fig. 11. Average relative error of top-k hot items when varying k.

Average Relative Error: *Our experimental results show that the ARE of SSS is [7.5, 62.4] times lower than that of Space-Saving, with different k varying from 200 to 600.* As shown in Figure 11, the ARE of SSS is only about 18 when $k = 500$, while for Space-Saving, the ARE is always higher than 1100.

VI. Conclusion

Finding top-k hot items in data streams is a critical problem for big data management. However, as the sizes of data streams become increasingly large, it becomes more and more difficult to design an accurate and fast algorithm for this problem. There are many existing algorithms for finding top-k hot items, and the *Space-Saving* algorithm is the most well-known algorithm. However, existing algorithms including Space-Saving, cannot achieve high accuracy and high memory efficiency at the same time. In this paper, we propose an enhanced algorithm based on Space-Saving, named Scoreboard Space-Saving (SSS). By using a queue and a Scoreboard, SSS achieve higher accuracy and high memory efficiency at the same time, and also achieves fast and constant speed. We also make mathematical analysis and conduct a series of experiments to compare the performance of SSS and Space-Saving. The experimental results show that SSS achieves up to 62.4 times higher accuracy than Space-Saving. We believe that SSS can be applied to improve the performance of finding top-k hot items.

References

[1] Source code. https://github.com/AltF4Top/Scoreboard-Space-Saving.

[2] Katsiaryna Mirylenka, Graham Cormode, Themis Palpanas, and Divesh Srivastava. Conditional heavy hitters: detecting interesting correlations in data streams. *very large data bases*, 24(3):395–414, 2015.

[3] Joong Hyuk Chang and Won Suk Lee. Finding recent frequent itemsets adaptively over online data streams. pages 487–492, 2003.

[4] Y K Cheung and Ada Waichee Fu. Mining frequent itemsets without support threshold: with and without item constraints. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1052–1069, 2004.

[5] Mohamed A Soliman, Ihab F Ilyas, and K Chenchuan Chang. Top-k query processing in uncertain databases. pages 896–905, 2007.

[6] Gero Dittmann and Andreas Herkersdorf. Network processor load balancing for high-speed links. *Proc. of the 2002 Int. Symp. on Performance Evaluation of Computer and Telecommunication Systems*, 735, 2002.

[7] S L Johnsson and Chingtien Ho. Optimum broadcasting and personalized communication in hypercubes. *IEEE Transactions on Computers*, 38(9):1249–1268, 1989.

[8] Robert T Schweller, Zhichun Li, Yan Chen, Yan Gao, Ashish Gupta, Yin Zhang, Peter A Dinda, Mingyang Kao, and Gokhan Memik. Reversible sketches: enabling monitoring and analysis over high-speed data streams. *IEEEACM Transactions on Networking*, 15(5):1059–1072, 2007.

[9] Yu Zhang, Binxing Fang, and Yongzheng Zhang. Identifying heavy hitters in high-speed network monitoring. *Science in China Series F: Information Sciences*, 53(3):659–676, 2010.

[10] Abdul Kabbani, Mohammad Alizadeh, Masato Yasuda, Rong Pan, and Balaji Prabhakar. Af-qcn: Approximate fairness with quantized congestion notification for multi-tenanted data centers. pages 58–65, 2010.

[11] Joong Hyuk Chang and Won Suk Lee. A sliding window method for finding recently frequent itemsets over online data streams. *Journal of Information Science and Engineering*, 20(4):753–762, 2004.

[12] Chihhsiang Lin, Dingying Chiu, Yihung Wu, and Arbee L P Chen. Mining frequent itemsets from data streams with a time-sensitive sliding window. pages 68–79, 2005.

[13] Nuno Homem and Joao Paulo Carvalho. Finding top-k elements in data streams. *Information Sciences*, 180(24):4958–4974, 2010.

[14] Tong Yang, Alex X Liu, Muhammad Shahzad, Dongsheng Yang, Qiaobin Fu, Gaogang Xie, and Xiaoming Li. A shifting framework for set queries. *IEEE/ACM Transactions on Networking*, 25(5):3116–3131, 2017.

[15] Tong Yang, Alex X Liu, Muhammad Shahzad, Yuankun Zhong, Qiaobin Fu, Zi Li, Gaogang Xie, and Xiaoming Li. A shifting bloom filter framework for set queries. *Proceedings of the VLDB Endowment*, 9(5):408–419, 2016.

[16] Pratanu Roy, Arijit Khan, and Gustavo Alonso. Augmented sketch: Faster and more accurate stream processing. In *Proc. SIGMOD 2016*.

[17] Graham Cormode. Sketch techniques for approximate query processing. *Foundations and Trends in Databases. NOW publishers*, 2011.

[18] Graham Cormode and S Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[19] Tong Yang, Yang Zhou, Hao Jin, Shigang Chen, and Xiaoming Li. Pyramid sketch: A sketch framework for frequency estimation of data streams. *Proceedings of the VLDB Endowment*, 10(11):1442–1453, 2017.

[20] Yang Zhou, Tong Yang, Jie Jiang, Bin Cui, Minlan Yu, Xiaoming Li, and Steve Uhlig. Cold filter: A meta-framework for faster and more accurate stream processing. *Proc. SIGMOD 2018*.

[21] Moses Charikar, Kevin Chen, and Martin Farachcolton. Finding frequent items in data streams. *international colloquium on automata, languages and programming*, 312(1):693–703, 2002.

[22] Graham Cormode and S Muthukrishnan. What's hot and what's not: tracking most frequent items dynamically. *symposium on principles of database systems*, 30(1):249–278, 2003.

[23] Richard M Karp, Scott Shenker, and Christos H Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM Transactions on Database Systems*, 28(1):51–55, 2003.

[24] Yin Zhang, Sumeet Singh, Subhabrata Sen, Nick Duffield, and Carsten Lund. Online identification of hierarchical heavy hitters: algorithms, evaluation, and applications. pages 101–114, 2004.

[25] Chao Wang, Qing Zhao, and Chennee Chuah. Group testing under sum observations for heavy hitter detection. *information theory and applications*, pages 149–153, 2015.

[26] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Flowradar: a better netflow for data centers. pages 311–324, 2016.

[27] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient computation of frequent and top-k elements in data streams. *international conference on database theory*, pages 398–412, 2005.

[28] Erik Demaine, Alejandro López-Ortiz, and J Munro. Frequency estimation of internet packet streams with limited space. *AlgorithmsESA 2002*, pages 11–20, 2002.

[29] Richard M Karp, Scott Shenker, and Christos H Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM TODS*, 28(1):51–55, 2003.

[30] Gurmeet Singh Manku and Rajeev Motwani. Approximate frequency counts over data streams. In *Proc. VLDB*, pages 346–357, 2002.

[31] Li Fan, Pei Cao, Jussara M Almeida, and Andrei Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEEACM Transactions on Networking*, 8(3):281–293, 2000.

[32] David MW Powers. Applications and explanations of Zipf's law. In *Proc. EMNLP-CoNLL*. Association for Computational Linguistics, 1998.

[33] Graham Cormode and S Muthukrishnan. Summarizing and mining skewed data streams. In *Proceedings of the 2005 SIAM International Conference on Data Mining*, pages 44–55. SIAM, 2005.

[34] Douglas C Montgomery, Elizabeth A Peck, and G Geoffrey Vining. *Introduction to linear regression analysis*. John Wiley & Sons, 2015.

[35] Ran Ben-Basat, Gil Einziger, Roy Friedman, and Yaron Kassner. Heavy hitters in streams and sliding windows. In *Proc. IEEE INFOCOM*, 2016.