Taking One Small Step Forward: Finding Low-Frequency Items in Data Streams

Michael Shell
School of Electrical and
Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332–0250
Email: http://www.michaelshell.org/contact.html

Homer Simpson
Twentieth Century Fox
Springfield, USA
Email: homer@thesimpsons.com

James Kirk
and Montgomery Scott
Starfleet Academy
San Francisco, California 96678-2391
Telephone: (800) 555–1212
Fax: (888) 555–1212

***Abstract*—Low-frequency items in data streams are becoming more and more important due to their large amount, and the ability to excavate the rich information contained in them is of great significance to many companies and organizations. We propose an one-pass algorithm, called *BFSS*, which can find items in a data stream with frequencies less than a user specified support. Our algorithm is simple and have small memory footprints. Although the output is approximate, we can guarantee the recall of *BFSS* is always 1 and the precision is high enough. Given a little modification, *BFSS* can be improved to *SBFSS*, which can handle data streams in limited and small space and gurantee a high precision together with a theoretically bounded recall. The experimental results tested on real data set show the recall of *BFSS* was 1 and the precision was above 0.95 using much less space compared to *nCount* and *rCount*, which are two naive methods finding low-frequency items in data streams. *SBFSS* outperformed *lCount*, a method finding low-frequency items in data streams using limited space with replacement strategy, in both precision and recall.**

I. INTRODUCTION

In many real-world applications, information such as web click data [1], stock ticker data [2], [3], sensor network data [4], phone call records [5], and network packet traces [6] appear in the form of data streams. Motivated by the above applications, researchers started working on novel algorithms for analyzing data streams. Problems studied in this context include approximate frequency moments [7], distinct values estimation [8], [9], bit counting [10], duplicate detection [11], [12], approximate quantiles [13], wavelet based aggregate queries [14], correlated aggregate queries [15], frequent elements [16]–[19] and top-k queries [20], [21].

However, to the best of our knowledge, there are no algorithms identifying low-frequency items over data streams. Low-frequency items, similar to the defination of frequent items, are items frequencies of which are less than a specified threshold S.

Zipf's law refers to the fact that many types of data studied in the physical and social sciences can be approximated with a Zipfian distribution, one of a family of related discrete power law distributions. Fig.1 describes relation between frequency rank and frequency of items that follow a power law distribution over data streams, and we can find that the distinct number of low-frequency items is much larger than the distinct number of frequent items, so under the restriction of limited memory it is much more difficult to identify low-frequency items than to identify frequent items.

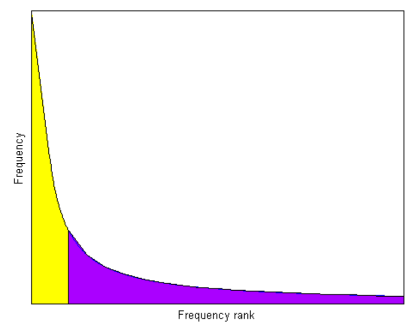


Fig. 1: Rank-frequency distribution

Nowadays, Low-frequency items over data streams are becoming more and more important because of the rich information they contain which can be easily understood through the observation of entropy of data streams [22]. We take one small step forward to maintain low-frequency items over data streams approximately and try to fill up the blank of low-frequency items mining over data streams in this paper.

A. Motivating Examples

1) Individual requirements mining: In such an era that information technology develops rapidly, it is much easier for us to get access to various of knowledge and information and we need through a few mouse clicks, for example, we can shop online with the help of e-commerce site, such as Amazon and Taobao etc. We can search whatever we are interested in through search engines, such as Google and Bing etc.

Our requirements are popular most times, for example, buying a regular water glass online or searching the information about a tourist attraction etc, and these popular requirements can be easily met because almost every e-commerce site sell all linds of water glasses and nearly every search engines provide links to popular tourist attractions worldwide.

However, we are no longer saiesfied with responses only to popular demands. For example, it is not so easy for us to buy embroidery stitches online or search information about a nameless small village in China, because these are individual requirements, and some e-commerce sites or search engines may neglect these requirements, for example, i can't find product information about embroidery stitches at Amazon while Taobao does.

Yusuf Mehdi, senior vice president of the Online Audience Group for Microsoft Bing, once said at Search Engine Strategies that major reason why Bing got begind with Google is neglecting "long tail" of search flow items of which appear a few times or even once, and it is of great importance for them to analyze them[1]. So it is individual requirements or in other words low-frequency demands, in some sense, that really make a difference rather than popular demands in areas like e-commerce and search engine market, and identifying them is the first step to analyze them.

2) Data distribution estimation: Data distribution is a basic and important property of a data stream. Sampling is a simple and fast method to estimate data distributions, however, sampling will lose much inforamtion of data streams and cause big errors sometimes. From Figure 1, we can observe that low-frequency elements have a great influence on data distributions of most data streams, in fact, low-frequency elements dominate the distinct elements in data streams of various distributions which can be seen in our experiments.

In fact, *BFSS* and *SBFSS* can find both frequent and low-frequency items in data streams efficiently, and in this way we can get much more accurate estimates of data distributions of data streams.

B. Our Contributions

We introduce and state the proplem of low-frequency items detection over data streams which is of great significance in areas such as individual requirements mining and data distribution estimation, and to the best of our knowledge there is no relative research up to now.

In this paper, we propose *BFSS*, which extends the classic frequent items detection algorithm *Space Saving* to maintain both frequent and low-frequency items in a data stream approximately. The basic idea of our solution is as follows: each item in a data stream is either a frequent one or a low-frequency one once the threshold $\phi[\in (0,1)]$ is set, so we can maintain low-frequency items by filtering the frequent items out. A major problem we have to deal with is to maintain an itemset items in which appear in the data stream, and this can be done approximately using a Bloom filter size of which is related to the size of the alphabet of a data stream. *BFSS* gurantees no false negatives and provably few false positives using small memory footprints.

However, the size of a Bloom filter must increase with the size of the alphabet of a data stream in order to keep a low false positive rate, and here comes a problem: In many embedded devices, such as sensors and routers etc., the storage space is limited and small. In this case, *BFSS* will produce many false positives. Inspired by the solution presented in [11], we propose *SBFSS* which extends *BFSS* to deal with data streams in limited and small space, and *SBFSS* gurantees few false positives and theoretically bounded false negatives.

[1] http://www.eweek.com/c/a/Search-Engines/Microsoft-Ignored-the-Long-Tail-in-Search-Bing-Boss-Says-396023

TABLE I: Major Notation Used in the Paper.

Notation	Meaning
BFSS	Our first algorithm
SBFSS	Our second algorithm
S	The input data stream
A	The domain of S
M	The size of A
M'	The number of distinct elements in S
ϕ	The user specified support threshold
D	The synopsis used in *BFSS* and *SBFSS*
$(e, f(e), \Delta(e))$	The form of each counter in D
E	The set of elements monitored in D
min	The minimum value of $f(e)$ in D
C	The maximum number of counters in D
m	The number of counters used in D
N	The number of elements in the input stream
H	The number of hash functions used in *BFSS* and *SBFSS*
$f_S(e)$	The frequency of element e in S
FPs	The elements in S with frequency no less than $\lfloor \phi N \rfloor$ wrongly output
FNs	The elements in S with frequency less than $\lfloor \phi N \rfloor$ wrongly neglected
BF	The Bloom filter used in *BFSS*
K	The size of *BF*
SBF	The Stable Bloom filter used in *SBFSS*
K'	The size of *SBF*
k	Number of cells used in *SBF*
d	Number of bits allocated per cell
max	The value a cell is set to
P	Number of cells we pick to decrement by 1 in each iteration
p	The probability that a cell is picked to be decremented by 1 in each iteration
p'	The probability that a cell is set in each iteration

C. Roadmap

In Section 2, we present the problem statement and some background on existing approches which deal with the problem of frequent items detection. Our solutions are presented and discussed in Section 3. In Section 4, we experimentally evaluate out methods. Conclusions are given in Section 5.

II. PRELIMINARIES

This section presents the problem statement and some representative algorithms solvimg *ϵ-Deficient Frequent Elements* [23] which will be formally defined below.Table I summarizes the major notation in this paper.

A. Problem Statement

Consider an input stream $S = e_1, e_2, ..., e_N$ of current length N, which arrives item by item. Let each item e_i belong to a universe set $A = \{a_1, a_2, ..., a_M\}$ of size M representing the input stream's domain. Let $f_S(a)$ denote the number of occurrences of a in S

Our problem *ϕ-Bounded Low-Frequency Elements* can be stated as follows: given a data stream S along with its domain A and a user specified threshold $\phi[\in (0,1)]$, output the subset $I(S,\phi) \subset A$ of symbols defined as $I(S,\phi) = \{a \in A : 0 < f_S(a) \leq \lfloor \phi N \rfloor\}$.

At any point of time, *BFSS* output a list of items with the following guarantees:

1) All elements whose true frequency is no more than $\lfloor \phi N \rfloor$ are output. There are no *false negatives*.

2) No element whose true frequency exceeds $2\lfloor\phi N\rfloor$ is output.

As for data streams from large domains which means M is so large that *BFSS* can not handle in mempry, *SBFSS* output a list of items using proper space under the restriction of limited memory with acceptable false negatives and false positives, where a false positive is an item with frequency more than $\lfloor\phi N\rfloor$ wrongly output, and a false negative is an item with frequency no more than $\lfloor\phi N\rfloor$ wrongly neglected.

B. Related Work

As far as we know there are no related algorithms addressing this problem, however, the methods we propose are based on the algorithm solving *ϵ-Deficient Frequent Elements* which can be described as follows: given a input stream S of current length N and a support threshold $\phi \in (0, 1)$, return the items whose frequencies are guaranteed to be no smaller than $\lfloor(\phi - \epsilon)N\rfloor$ deterministically or with a probability of at least $1 - \delta$, where $\epsilon \in (0, 1)$ is a user-defined error and $\delta \in (0, 1)$ is a probability of failure, so we examine several algorithms solving *ϵ-Deficient Frequent Elements*.

research can be divided into two groups: *counter-based* techniques and *sketch-based* techniques.

Counter-Based Techniques keep an individual counter for each element in the monitored set, a subset of A. The counter of a monitored element, e_i, is updated when e_i occurs in the stream. If there is no counter kept for the observed ID, it is either disregarded, or some algorithm-dependent action is taken.

Two representative algorithms *Sticky Sampling* and *Lossy Counting* were proposed in [23]. The algorithms cut the stream into rounds, and they prune some potential low-requency items at the edge of each round. Though simple and intuitive, they suffer from zeroing too many counters at rounds boundaries, and thus, they free space before it is really needed. In addition, answering a frequent elements query entails scanning all counters.

The algorithm *Space Saving* , the one we are based at, was proposed in [20]. The algorithm maintains a synopsis which keeps all counters in an order according to the value of each counter's monitoring frequency plus maximum possible error. For a non-monitored item, the counter with the smallest counts, *min*, is assigned to monitor it, with the items monitoring frequency $f(e)$ set to 1 and its maximal possible error $\Delta(e)$ set to *min*. Since $min \leq \epsilon N$ (this follows because of the choice of the number of counters), the operation amounts to replacing an old, potentially infrequent item with a new, hopefully frequent item. This strategy keeps the item information until the very end when space is absolutely needed, and it leads to the high accuracy of *Space-Saving*. Experiments done in [24], [25] showed *Space-Saving* outperformed other Counter-Based techniques in recall and precision tests.

Sketch-Based Techniques do not monitor a subset of elements, rather provide, with less stringent guarantees, frequency estimation for all elements using bitmaps of counters. Usually, each element is hashed into the space of counters using a family of hash functions, and the hashed-to counters are updated for every hit of this element. Those representative counters are then queried for the element frequency with less accuracy, due to hashing collisions.

The *Count-Min Sketch* algorithm of Cormode and Muthukrishnan [26] maintains an array of $d \times w$ counters, and pairwise independent hash functions h_j map items onto $[w]$ for each row. Each update is mapped onto d entries in the array, each of which is incremented. The Markov inequality is used to show that the estimate for each j overestimates by less than n/w, and repeating d times reduces the probability of error exponentially.

The *hCount* algorithm was proposed in [27]. The data structure and algorithms used in *Count-Min Sketch* and *hCount* shared the similarity, but were simultaneously and independently investigated with different focuses.

III. OUR ALGORITHMS

In this section, we will discuss our approaches *BFSS* and *SBFSS* in detail.

A. The Challenges in ϕ-Bounded Low-Frequency Elements

ϕ-Bounded Low-Frequency Elements has two main challenges due to the different features between frequent items and low-frequency elements over data streams.

The long tail phenomenon. It can be easily proved that there are at most $\lceil 1/\phi\rceil$ frequent elements whose frequency is more than $\lfloor\phi N\rfloor$ in any data stream, however, there is no provable upper bound of the number of low-frequency elements whose frequency is less than $\lfloor\phi N\rfloor$. In fact, our experiments on both real and synthetic data show that low-frequency elements occupy most of the distinct elements appear in data streams, and it is almost impossible to maintain all low-frequency elements in memory. Another observation is that their frequencies are very low and close as well, and it may consume much space to separate low-frequency elements from frequent elements especially when ϕ is very small.

The unpredictability. A basic and common idea of *Counter-Based Techniques* is to discard potential infrequent elements dynamically, and it is based on the fact that potential infrequent elements will never become frequent elements if they don't appear afterwards, however, this fact no longer applies to low-frequency elements in data streams because potential elements will possibly become low-frequency elements if they don't appear afterwards. The unpredictability of low-frequency elements makes it difficult to maintain them directly like frequent elements.

B. The BFSS Algorithm

In consideration of the challenges in *ϕ-Bounded Low-Frequency Elements*, we tried to solve the problem indirectly by filtering frequent items out which is the underlying idea of *BFSS*.

Two algorithms are proposed for updating and outputting final result separately. Algorithm 1 maintains a Bloom filter BF of size K with H uniformly independent hash functions $\{h_1(x), ..., h_H(x)\}$ and a synopsis D with M counters. Each of these H hash function maps an element from A to $[0, ..., K-1]$. Initially each bit of BF is set to 0 and D has

M empty counters. Each newly arrived element in the stream is mapped to H bits in BF by the H hash functions and we set the H bits to 1. Then if we observe an element that is monitored in D, we just increment $f(e)$. If we observe an element, e_{new}, that is not monitored in D, handle it depending on whether there is an empty counter in D. If there is one, we just allocate it to e_{new} and set $f(e_{new})$ to 1 and set $\Delta(e_{new})$ to 0. If D is full, we just replace the element that currently has the least hits, min, with e_{new}. Assign $f(e_{new})$ the value $min+1$ and assign $\Delta(e_{new})$ the value min.

Algorithm 1 BFSS Update Algorithm

Input: Stream S, support threshold ϕ
$N = 0, m = 0, C = \lceil 1/\phi \rceil, K = \lambda', H = \mu'$; {$N$: length of stream; m: the number of counters used in D; C: the maximum number of counters in D; K: size of Bloom filter; H: number of hash functions;The value of λ' and μ' will be discussed in detail later.}
The form of the counters in D is $(e, f(e), \Delta(e))$
for $i = 0$ to $K-1$ **do**
  $BF[i] = 0$
end for
for each item e of stream S **do**
  for $i = 1$ to H **do**
    $BF[h_i(e)] = 1$
  end for
  if e is monitored in D **then**
    $f(e) = f(e) + 1$;
  else if $m < C$ **then**
    Assign a new counter $(e, 1, 0)$ to it
    $m = m + 1$
  else
    Let e_m be the element with least hits, min
    Replace e_m with e in D
    $f(e) = min + 1, \Delta(e) = min$
  end if
  $N = N + 1$;
end for

Algorithm 2 checks and outputs the element with frequency less than a user-specified threshold ϕ. For each element in A, we first check whether it is in BF. If the element is not in BF, it must not be a low-frequency item because it never appeared in the stream. If the element is in BF but not in D, it must be a low-frequency item and we output it, and we will give the reason later. If the element appears both in BF and D, we identify the element as a low-frequency element if $f(e) < \lfloor \phi N \rfloor + \Delta(e)$ with high probability. The time requirement of Algorithm 2 is linear to the range of universe. It is acceptable when the frequency of the requests is not high.

C. Analysis of BFSS

In this section, we present a theoretical analysis of *BFSS* described in Section III-B. We analyze *FNs*, *FPs*, space complexity, and time complexity. At last, we will identify the challenges to *BFSS*.

*1) **Analysis of FNs**:* In this section, we will prove that there are no *FNs* in the output of *BFSS*. The proof is based on Lemma 1 to Lemma 5, and the detailed proof of Lemma 1 to Lemma 3 can be found in [20].

Algorithm 2 BFSS Query Algorithm

Input: *BF*, D, ϕ, A, N, M
Output: low-frequency elements with threshold ϕ
$flag = true$; { *flag*: indicate whether an element is in *BF* or not}
for $i = 0$ to $M-1$ **do**
  $flag = true$
  for $j = 1$ to H **do**
    if $BF[h_j(A[i])] == 0$ **then**
      $flag = false$
      break;
    end if
  end for
  if $flag == true$ **then**
    if $A[i]$ is monitored in D **then**
      if $f(A[i]) \le \lfloor \phi N \rfloor + \Delta(A[i])$ **then**
        output $A[i]$ as a low-frequency element
      end if
    else
      output $A[i]$ as a low-frequency element
    end if
  end if
end for

Lemma 1: $N = \sum_{\forall i | e_i \in E}(f(e_i))$

Proof: Every hit in S increments only one counter by 1 among the M counters which can be easily proved when D has empty counters. It is true even when a replacement happens, i.e., the observed element e was not previously monitored and D has no empty counters, and it replaces another element e_m. This is because we add $f(e_m)$ to $f(e)$ and increment $f(e)$ by 1. Therefore, at any time, the sum of all counters is equal to the length of the stream observed so far. ∎

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

Proof: Lemma 1 can be written as:

$$min = \frac{N - \sum_{\forall i | e_i \in E}(f(e_i) - min)}{m} \quad (1)$$

All the items in the summation of Equation 1 are non negative because all counters are no smaller than min, hence $min \le \lfloor \frac{N}{m} \rfloor$. ∎

Lemma 3: For any element $e \in E$, $0 \le \Delta(e) \le \lfloor \phi N \rfloor$.

Proof: From Algorithm 1, $\Delta(e)$ is non-negative because any observed element is always given the benefit of doubt. $\Delta(e)$ is always assigned the value of the minimum counter at the time e started being observed. Since the value of the minimum counter monotonically increases over time until it reaches the current min, then for all monitored elements $\Delta(e) \le min$.

Consider thses two cases: i) If $m = \lceil \frac{1}{\phi} \rceil$, $min \le \lfloor \phi N \rfloor$; ii) if $m < \lceil \frac{1}{\phi} \rceil$, $\Delta(e) = 0$. For any case, we have $0 \le \Delta(e) \le \lfloor \phi N \rfloor$. ∎

Lemma 4: An element e with $f_S(e) > min$, must be monitored in D.

Proof: The proof is given in [20], however, we think the proof is not very rigorous because it is based on the fact that the true frequency of e must be no more than its estimated frequency, i.e. $f_S(e) \le f(e)$, but this fact should be based on Lemma 4.

My proof is also by contradiction. Assume $e \notin E$. Then, it was evicted previously, and we assume that it had been monitored in $i(>0)$ time slots, and e appeared $n_j(0 < j \le i)$ times in the jth time slot. n_j satisfies:

$$\sum_{j=1}^{i} n_j = f_S(e) \tag{2}$$

We assume that $\Delta(e_j)(>0)$ denotes the error estimation assigned to e at the start of the jth time slot, and we have the following inequality because the minimum counter value increases monotonically:

$$\begin{aligned} \Delta(e_1)+n_1 &\le \Delta(e_2) \\ \Delta(e_2)+n_2 &\le \Delta(e_3) \\ &... \\ \Delta(e_{i-1})+n_{i-1} &\le \Delta(e_i) \\ \Delta(e_i)+n_i &\le min \end{aligned} \tag{3}$$

After adding up the left and right sides of inequality group 3, we can get:

$$\Delta(e_1) + \sum_{j=1}^{i} n_j \le min \tag{4}$$

We can get $f_S(e) \le min$ from equation 2 and inequation 4. This contradicts the condition $f_S(e) > min$.

In fact, Lemma 4 can also be stated as: For any element $e \notin E$ in S, $f_S(e) \le min \le \lfloor \phi N \rfloor$. ■

Lemma 5: For any element $e \in E$, $f(e) - \Delta(e) \le f_S(e) \le f(e) \le f(e) - \Delta(e) + min$

Proof: From Lemma 3, we can easily get $f(e) \le f(e) - \Delta(e) + min$. $f(e) - \Delta(e)$ is the true frequency of e since it was lastly observed, so $f(e) - \Delta(e) \le f_S(e)$. From Lemma 4, we can find that $\Delta(e)$ over estimated the frequency of e before it was observed ,and it clearly indicates $f_S(e) \le f(e)$. From the above, we can prove Lemma 5. ■

Theorem 1: There are no *FNs* in the output of *BFSS*.

Proof: We only hava to prove that the elements we don't output contain no low-frequency elements. From Algorithm 1, only two kinds of elements are not ouput: i) elements filtered out by the *BF*. ii) elements monitored in D with $f(e) - \Delta(e) > \lfloor \phi N \rfloor$. The first kind of elements are obviously not low-frequency elements because they never appeared in S. From Lemma 5, we know that the elements monitored in D with $f(e) - \Delta(e) > \lfloor \phi N \rfloor$, must satisfy $f_S(e) > \lfloor \phi N \rfloor$. ■

2) ***Analysis of FPs:*** In this section, we will give a theoretically strict bound of expectation of number of *FPs* in output of *BFSS* regardless of data distribution of S, and a tighter bound can be derived for data stream under Pareto distribution [28].

Lemma 6: Any element e with $f_S(e) > 2\lfloor \phi N \rfloor$, must not be output.

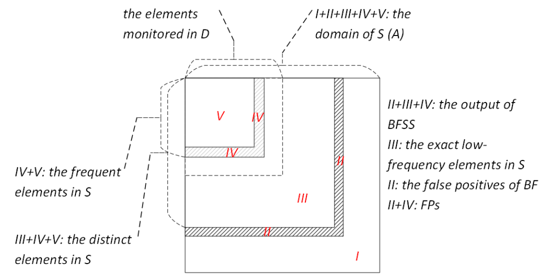


Fig. 2: The schematic diagram of the analysis of *BFSS*

Proof: From Lemma 4, we know that any element e with $f_S(e) > 2\lfloor \phi N \rfloor$ must be monitored in D, i.e. $e \in E$. Then from Lemma 5, we can get that elements monitored in D must satisfy $f_S(e) \le f(e)$, and that is to say any element e monitored in D with $f_S(e) > 2\lfloor \phi N \rfloor$, must satisfy $f(e) > 2\lfloor \phi N \rfloor$. At last from Lemma 3, we can prove that any element e with $f(e) > 2\lfloor \phi N \rfloor$, must satisfy $f(e) > \lfloor \phi N \rfloor + \Delta(e)$, which means e will not be output. ■

Lemma 7: The probability of a false positive of *BF* is no more than:

$$(1 - (1 - \frac{1}{K})^{HM})^H \tag{5}$$

Proof: First, a false positive of *BF* means an element in A not appearing in S but not filtered out by *BF*. Observe that after inserting M keys into a table of size K, the probability that a particular bit is still 0 is exactly

$$(1 - \frac{1}{K})^{HM} \tag{6}$$

Hence the probability of a false positive in this situation is $(1 - (1 - \frac{1}{K})^{HM})^H$. However, we know that M denotes the size of A which is the domain of S, so the number of distinct elements in S must be no more than M, i.e. $M' \le M$, and further we have $(1 - (1 - \frac{1}{K})^{HM'})^H \le (1 - (1 - \frac{1}{K})^{HM})^H$. ■

Theorem 2: Assuming no specific data distribution, the expectation of the number of *FPs* in the output of *BFSS*, denoted as $E(\#FPs)$, satisfies:

$$E(\#FPs) < M(1 - (1 - \frac{1}{K})^{HM})^H + \lceil \frac{1}{\phi} \rceil \tag{7}$$

Proof: From Algorithm 1, we can observe that two kinds of elements contribute to *FPs*: i) the false positives of *BF*; ii) the elements with $f_S(e) > \lfloor \phi N \rfloor$ wrongly output. The two cases correspond to the gray areas in Fig. 2 which is abstracted out from the analysis of *BFSS*. Concretely speaking, the ligth gray area represents the elements with $f_S(e) > \lfloor \phi N \rfloor$ wrongly output, i.e. the second case, and the dark gray area represents the elements not filtered out by *BF*, i.e. the first case. Furthermore, the output of *BFSS* is represented by $II + III + IV$, and the exact low-frequency elements in S is represented by III.

For the first case, we define the independent 0-1 random variables $x_i (1 \le i \le M)$ for each element in A, and the value of x_i depends on whether a_i appeared in S or not. If a_i appeared in S, then $x_i = 0$; If not, then with a probability of

$(1-(1-\frac{1}{K})^{HM'})^H$, $x_i = 1$; From the defination of x_i, we can find that the expectation of the number of the false positives of *BF* equals $E(\sum_{i=1}^{M} x_i)$, i.e. $(M-M')(1-(1-\frac{1}{K})^{HM'})^H$.

For the second case, we know from Lemma 6 that elements with $f_S(e) > 2\lfloor \phi N \rfloor$ must not be not output, so only elements with $\lfloor \phi N \rfloor < f_S(e) \leq 2\lfloor \phi N \rfloor$ are likely to be output, and these elements must be monitored in D which can be directly derived from Lemma 4, so the maximum number of these elements is $\lceil \frac{1}{\phi} \rceil$. From above, due to the linear properties of expectation, we can get:

$$E(\#FPs) \leq (M-M')(1-(1-\frac{1}{K})^{HM'})^H + \lceil \frac{1}{\phi} \rceil \quad (8)$$

In addition, $M' \leq M$, so inequation 7 can be easily derived from inequation 8. ■

However, from the derivation above, we can see that $\frac{1}{\phi}$ is a loose bound of the second case because elements with $f_S(e) > 2\lfloor \phi N \rfloor$ can not be output, and we count them in. So if we make an assumption of the distribution of S, we will get a relatively tighter bound of the second case.

Theorem 3: Assuming noiseless Pareto data with parameter α and $f_S(e_m)$, where $\alpha(\alpha > 0)$ and e_m denotes the element with the lowest freqeuncy, we have:

$$E(\#FPs) < M(1-(1-\frac{1}{K})^{HM})^H + T \quad (9)$$

$$T = (\frac{f_S(e_m)}{\lfloor \phi N \rfloor})^{\alpha}(1-\frac{1}{2^{\alpha}})M \quad (10)$$

Proof: The Pareto distribution[2] has the following property: If X is a random variable with a Pareto distribution, then the probability that X is greater than some number x, i.e. the survival function, is given by:

$$Pr(X > x) = \begin{cases} (\frac{x_m}{x})^{\alpha} & x \geq x_m \\ 1 & x < x_m \end{cases}$$

where x_m is the minimum possible value of X, and α is a positive parameter. So In such a case, the expected value of the number of the elements with $\lfloor \phi N \rfloor < f_S(e) \leq 2\lfloor \phi N \rfloor$ is $(\frac{f_S(e_m)}{\lfloor \phi N \rfloor})^{\alpha}(1-\frac{1}{2^{\alpha}})M'$. ■

We may not calculate the exact value of T because we can't get the exact value of $f_S(e_m)$, however, once a query comes, the parameters are confirmed and so is the value of T, furthermore, the upper bound of the value of $E(\#FPs)$ is confirmed as well. In this sense, it is meaningful to give the expression of T.

From Inequation 7, we observe that the upper bound of $E(\#FPs)$ is related with the values of H and K, and our goal is to minimize the value of $E(\#FPs)$ by choosing the appropriate values. In addtion, the value of K directly influence the space complexity of *BFSS*, so we will make some analysis of the values of K and H then.

[2]https://en.wikipedia.org/wiki/Pareto_distribution

TABLE II: The value of F under various $\frac{K}{M}$ and H combinations.

K/M	H	$H=8$	$H=9$	$H=10$	$H=11$	$H=12$
13	9.01	0.00199	0.00194	0.00198	0.0021	0.0023
14	9.7	0.00129	0.00121	0.0012	0.00124	0.00132
15	10.4	0.000852	0.000775	0.000744	0.000747	0.000778
16	11.1	0.000574	0.000505	0.00047	0.000459	0.000466
17	11.8	0.000394	0.000335	0.000302	0.000287	0.000284

*3) **Space complexity**:* In this section, we will give some analysis of the space complexity of *BFSS* including how to choose the appropriate values of H and K.

Choosing the appropriate values of H and K. Our goal is to minimize $E(\#FPs)$, and from Algorithm 1, we see that the values of M and ϕ are confirmed at the very beginning before our algorithm. In this way, our task is to minimize the value of $(1-(1-\frac{1}{K})^{HM})^H$, denoted as F, by calculating the appropriate values of H and K, and we have:

$$F \approx (1-e^{-\frac{HM}{K}})^H = e^{H\ln(1-e^{-\frac{HM}{K}})} \quad (11)$$

Let $P = e^{-\frac{HM}{K}}$ and $G = H\ln(1-e^{-\frac{HM}{K}})$, and in order to minimize F, we only have to minimize G:

$$G = (-\frac{K}{M})\ln(P)\ln(1-P) \quad (12)$$

The first derivative of G with respect to P, denoted as G', is:

$$G' = \frac{K}{M}(\frac{1}{1-P}\ln(P) - \frac{1}{P}\ln(1-P)) \quad (13)$$

From Equation 13, we can easily observe that when $P = \frac{1}{2}$, G reaches its minimum value. In this case, we have:

$$H = \frac{K}{M} \times \ln 2 \quad (14)$$

Combined with Equation 11 and Equation 14, we have:

$$F \approx (1-e^{-\frac{HM}{K}})^H = (\frac{1}{2})^H \approx (0.6185)^{\frac{K}{M}} \quad (15)$$

From the above derivation, we know that once the value of $\frac{K}{M}$ is confirmed, we can get the appropriate value of H to minimize the value of F. For example, if the value of $\frac{K}{M}$ is set to 13, the appropriate value of H is $\frac{K}{M} \times \ln 2 \approx 9.1$, because the value of H is an integer, we set it to 9. Considering the length limit, we give a small fraction of the value of F under various $\frac{K}{M}$ and H combinations in Table II.

From Equation 15, we notice that the larger the value of H is, the smaller the value of F is. However, the value of K increases monotonically with the increasing value of H once the value of M is confirmed which can be observed from Equation 14, in other words, in order to decrease the value of $E(\#FPs)$, we have to increase the size of *BF*, i.e. K.

For example, if the value of M is 1M and the value of ϕ is 0.001. From Table II, we see that if the value of K is 16M, the appropriate value of H should be 11, therefore the value of F is 0.000459, and the upper bound of the value of $E(\#FPs)$,calculated by Inequation 7, is $1M \times 0.000459 + 1/0.001 = 1459$. Meanwhile, if the value of K is 17M, the corresponding values of H and F are 12 and 0.000284, therefore the upper bound of the value of $E(\#FPs)$ is 1284. From the proof of Theorem 2, we know that in order to caculate the exact upper bound of the value of $E(\#FPs)$, the bound given in Inequation 7 is not so tight, in fact, the experimental results indicate a much better performence.

Theorem 4: The space complexity of *BFSS* is $O(\lambda M + min(\lceil \frac{1}{\phi} \rceil, M))$, where $\lambda \in N^*$, the value of which is discussed detaily above.

Proof: The space complexity of *BFSS* includes two part: i) the size of *BF*, i.e. K. ii) the number of counters used in D, i.e. m. In order to get the minimum value of $E(\#FPs)$, the value of K should be multiple times of the value of M, i.e. $K = \lambda M (\lambda \in N^*)$. From Algorithm 2, we have $m \leq min(\lceil \frac{1}{\phi} \rceil, M)$. So the space complexity of *BFSS* is $O(\lambda M + min(\lceil \frac{1}{\phi} \rceil, M))$. ■

In conclusion, there exists a trade off between the number of *FPs* and the space consumption of *BFSS*. Theoretically, once the value of M is confirmed, the larger the value of λ, the smaller the number of *FPs* should be, and the larger the value of K should be.

Furthermore, consider a naive method to solve ϕ*-Bounded Low-Frequency Elements*: we simply allocate each element in A a counter, and update the corresponding counter for each element in S, when a query comes, we just output the elements with $f_S(e) \leq \lfloor \phi N \rfloor$. Obviously the method can maintain the low-frequency elements precisely, and the space complexity of the method is $O(M)$. *BFSS* has no advantage over the naive method in space complexity considering big O though, *BFSS* needs not to store the exact value or the fingerprint of each element which may consume much space in total especially when the value is long string, URL etc. The experimental results indicate *BFSS* is much more space efficient too.

*4) **Time complexity**:* In this section, we will discuss the time complexity of *BFSS*.

Theorem 5: Processing each data stream element needs $O(1)$ time, independent of the size of the stream.

Proof: From Algorithm 1, we know the processing time for each element consists two parts: i) the time spent hasing each element into *BF*. ii) the time spent updating the corresponding counter in D. Due to H is constant, time spent hashing each element into *BF* is constant too. Consider two cases in updating the corresponding counter in D: i) element e is monitored in D, and we only have to update the corresponding counter. ii) element e is not monitored in D, and we need to first locate the counter with the minimum $f(e)$, then update it. Obviously, the time spent for any case is constant because $\lceil \frac{1}{\phi} \rceil$ is constant. Therefore, processing each element needs only $O(1)$ time in total. ■

*5) **The challenges to BFSS**:* In this section, we will identify the existing problems in *BFSS*.

From the analysis in section III-C3, we observe that K should increase monotonically with M in order to keep a low rate of *FPs*, however, what if the space available is limited and small? The precision of *BFSS* will be very low in this case.

Inspired by the algorithm proposed in [11], we present the *SBFSS* algorithm, which can find low-frequency elements over data streams in limited and small space with few *FPs* together with theoretically bounded *FNs*.

D. The SBFSS Algorithm

In this section, we will introduce *SBFSS*, which modifies *BFSS* and can find low-frequency items in data streams using limited and small space.

The major difference between *BFSS* and *SBFSS* is the Bloom filter we use. Concretely speaking, *BF* is a regular Bloom filter, while *SBF* change bits in *BF* into cells, each consisting of one or more bits. In addition, from the explaination above, the size of *BF*, i.e. K, should increase monotonically with M, however, there can be no linear relationship between the size of *SBF*, i.e. K', and M, and K' can be any integer predefined by users. Defination 1 is the detailed defination of *SBF*.

Definition 1: *SBF* is defined as an array of integer $SBF[1], SBF[2], ..., SBF[k]$ whose minimum value is 0 and maximum value is max. Each element of the array is allocated d bits. The relation between max and d is then $max = 2^d - 1$. Compared to bits in a regular Bloom lter, each element of the *SBF* is called a cell.

Like *BFSS*, *SBFSS* consists of two phases: updating and quering. Algorithm 3 and Algorithm 4 give the detailed descriptions of them. Due to the length limit, no more tautology here.

Algorithm 3 SBFSS Update Algorithm

Input: Stream S, support threshold ϕ

```
The form of the counters in D is (e, f(e), Δ(e))
for i = 0 to k − 1 do
    SBF[i] = 0
end for
for each item e of stream S do
    Select P different cells uniformly at random
        SBF[j_1]...SBF[j_P], P ∈ {1, ..., k}
    for each cell SBF[j] ∈ {SBF[j_1]...SBF[j_P]} do
        if SBF[j] ≥ 1 then
            SBF[j] = SBF[j] − 1
        end if
    end for
    for i = 1 to H do
        SBF[h_i(e)] = max
    end for
    if e is monitored in D then
        f(e) = f(e) + 1;
    else if m < C then
        Assign a new counter (e, 1, 0) to it
        m = m + 1
    else
        Let e_m be the element with least hits, min
        Replace e_m with e in D
        f(e) = min + 1, Δ(e) = min
    end if
    N = N + 1;
end for
```

From Algorithm 3, we observe that there are two kinds of operation on *SBF*. First randomly decrement P cells by 1 so as to make room for fresh elements ; then set the same H cells as in the update process to max.

Algorithm 4 SBFSS Query Algorithm

Input: *SBF*, D, ϕ, A, N, M
Output: low-frequency elements with threshold ϕ

```
flag = true; { flag: indicate whether an element is in SBF or not}
for i = 0 to M − 1 do
   flag = true
   for j = 1 to H do
      if SBF[h_j(A[i])] == 0 then
         flag = false
         break;
      end if
   end for
   if flag == true then
      if A[i] is monitored in D then
         if f(A[i]) ≤ ⌊φN⌋ + Δ(A[i]) then
            output A[i] as a low-frequency element
         end if
      else
         output A[i] as a low-frequency element
      end if
   end if
end for
```

Intuitively, due to the random deletion operation, *SBF* does not exceed its capacity in a data stream scenario, however, false negatives may come up as well.

E. Analysis of SBFSS

Like *BFSS*, we will give theoretical analysis of *SBFSS* in this section, including the equivalence of *SBF'* (the *SBF* used in [11]) and *SBF*, *FNs*, *FPs*, parameters setting, and time complexity.

*1) **Analysis of the equivalence of SBF' and SBF**:* *SBF'* is used to detect duplicates over data streams, in fact, from Algorithm 3 and Algorithm 4, we can observe that *SBF* is used to filter out the elements that never appeared in S, and we check whether the element in A is duplicated one by one, if the element is a duplicate, we just claim it appeared in S, in this sense, *SBF'* and *SBF* share the same function. So the properties of *SBF'*, including the probability of a false positive and a false negative of *SBF'*, are fully applicable to *SBF*.

*2) **Analysis of FPs**:* In this section, we will give a theoretically upper bound of the expectation of the number of *FPs* in the ouput of *SBFSS*.

The detailed proof of Lemma 8 can be found in [11]. Because of the limitation of length, no more tautology here.

Lemma 8: The probability of a false positive (FP) of *SBF* is no greater than :

$$(1-(\frac{1}{1+\frac{1}{P(1/H-1/m)}})^{max})^H \tag{16}$$

Theorem 6: Assuming no specific data distribution, the expectation of the number of *FPs* in the output of *SBFSS*,

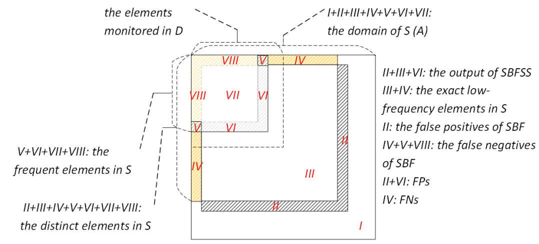


Fig. 3: The shematic diagram of the analysis of *SBFSS*

denoted as $E'(\#FPs)$, satisfies:

$$E'(\#FPs) < M\min\{(1-(\frac{1}{1+\frac{1}{P(1/H-1/m)}})^{max})^H, \quad (1-(1-\frac{1}{k})^{HM})^H\} + \lceil\frac{1}{\phi}\rceil \tag{17}$$

Proof: From Fig. 3, we notice that the *FPs* of *SBFSS* has two parts: i) the false positives of *SBF*, i.e. II; ii) the elements with $f_S(e) > \lfloor\phi N\rfloor$ wrongly output, i.e. VI. In fact, V is the only difference between the *FPs* of *SBFSS* and the *FPs* of *BFSS*, for *BFSS*, the elements in V will be output, however, due to the false negatives of *SBF* in *SBFSS*, i.e. $IV+V+VIII$, the elements in V will be neglected by *SBFSS*, in this sense, the false negatives of *SBF* inversely reduce the number of *FPs* of *SBFSS*.

In fact, the posibility of a false positive of *SBF* must be smaller than that of *BF* under the same conditions because *SBF* brings in decrement operation. For the first case, we can get the upper bound of the expectation of the number of the false positives of *SBF* through Lemma 7 and Lemma 8. For the second case, it is clearly to see from Fig. 3 that the maximum of the area of VI is $\lceil\frac{1}{\phi}\rceil$, which is the area of the square surrounded by dotted lines, and obviously it is a loose bound. The residual derivation is obvious and exactly the same as Theorem 2, and no more tautology here. ∎

Like *BFSS*, a tighter bound of the area of VI can be obtained if we assume the distribution of S, and the bound is $(\frac{f_S(e_m)}{\lfloor\phi N\rfloor})^{\alpha}(1-\frac{1}{2^{\alpha}})M$, the derivation of which is exactly the same as E.q. 10, if the distribution is Pareto. In addition, due to the exsitence of V, the number of the elemens with $f_S(e) > \lfloor\phi N\rfloor$ wrongly output is smaller than that of *BFSS* regardless of the data distribution of S.

*3) **Analysis of FNs**:* In this section, unlike *BFSS*, we will give a theoretically upper bound of the expectation of the number of *FNs* in the ouput of *SBFSS*.

From Fig. 3, we can see that the *FNs* of *SBFSS*, IV, is a portion of the false negatives of *SBF*, $IV+V+VIII$. Therefore the upper bound of the number of *FNs* in the output of *SBFSS* is the number of the false negatives of *SBF*.

First, let us consider the probability of a false negative of *SBF* which is an error when an element in S wrongly filtered out by *SBF*. Therefore only the elements in S can generate false negatives, obviously the number of false negatives is related to the input data distribution.

Suppose an element $A[i]$ appearing in S whose last appearence in S before a query comes is $e_{i-\delta_i}$, where δ_i represents the number of elements in S during the time between the last appearance of $A[i]$ and a query, is hashed into H cells, $SBF[C_{i1}], ..., SBF[C_{iH}]$. A false negative happens if any of those H cells is decremented to 0 during the δ_i iterations when a query comes. Let $PR_0(\delta_i, p'_{ij})$ be the probability that cell $C_{ij}(j = 1...H)$ is decremented to 0 within the δ_i iterations, and A_l denotes the event that within the N iterations the most recent setting operation applied to the cell occurs at iteration $N - l$, and A'_N denotes the event that cell has never been set within the whole N iterations. $PR_0(\delta_i, p'_{ij})$ can be computed as:

$$PR_0(\delta_i, p'_{ij}) = \sum_{l=max}^{\delta_i - 1} [Pr(SBF_{\delta_i} = 0|A_l)Pr(A_l)] + Pr(SBF_{\delta_i} = 0|A_{\delta_i})Pr(A'_{\delta_i}) \tag{18}$$

where

$$Pr(SBF_{\delta_i} = 0|A_l) = \sum_{j=max}^{l} \binom{l}{j} p^j (1-p)^{l-j} \tag{19}$$

$$Pr(A_l) = (1 - p'_{ij})^l p'_{ij} \tag{20}$$

$$Pr(SBF_{\delta_i} = 0|A'_{\delta_i}) = \sum_{j=max}^{\delta_i} \binom{\delta_i}{j} p^j (1-p)^{\delta_i - j} \tag{21}$$

$$Pr(A'_{\delta_i}) = (1 - p'_{ij})^{\delta_i} \tag{22}$$

Based on E.q 18, the expected value of the false negatives of *SBF*, $E(\#FN)$, can be easily derived, and the tailed derivation of E.q 18 and Lemma 9 can be found in [11], therefore no more tautology here.

Lemma 9: There must be an "average" $\widehat{\delta}$ and an "average" $\widehat{p'}$ such that:

$$E(\#FN) = M'[1 - (1 - PR_0(\widehat{\delta}, \widehat{p'}))^H] \tag{23}$$

Theorem 7: Assuming no specific data distribution, the expectation of the number of *FNs* in the output of *SBFSS*, denoted as $E'(\#FNs)$, satisfies:

$$E'(\#FNs) \leq M[1 - (1 - PR_0(\widehat{\delta}, \widehat{p'}))^H] \tag{24}$$

Proof: Like *FPs*, two kinds of elements contribute to *FNs*: i) the elemtents with $0 < f_S(e) \leq \lfloor \phi N \rfloor$ wrongly filtered out by *SBF*; ii) the elemtents with $f_S(e) \leq \lfloor \phi N \rfloor$ monitored in D wrongly neglected. In fact, elements with $f_S(e) \leq \lfloor \phi N \rfloor$ must satisfy $f(e) \leq \lfloor \phi N \rfloor + \Delta(e)$, which means no elements with $f_S(e) \leq \lfloor \phi N \rfloor$ monitored in D wrongly neglected. For the first case, we assume that the false negatives are all elements with $0 < f_S(e) \leq \lfloor \phi N \rfloor$, i.e. neglecting $V + VIII$ in Fig. 3, which is the worst case, and in this way we have $E(\#FN) = E(\#FNs) = M'[1 - (1 - PR_0(\widehat{\delta}, \widehat{p'}))^H] \leq M[1 - (1 - PR_0(\widehat{\delta}, \widehat{p'}))^H]$. ■

From E.q 17, we observe that the upper bound of $E'(\#FPs)$ is completely related to *SBF* once the parameter ϕ is confirmed, and the upper bound of $E'(\#FNs)$ is totally related to *SBF* which can be observed from E.q 24. In this way, we will give some analysis of how to choose appropriate parameters of *SBF* next.

*4) **Parameters setting**:* In this section, we will discuss how to choose appropriate parameters of *SBF* in *SBFSS*.

For *SBFSS*, users will specify ϕ, $\#FPs$ and memory limit L. The memory consumption of *SBFSS* mainly contains two parts: *SBF* and D. From Algorithm 3, we know that there are at most $\lceil \frac{1}{\phi} \rceil$ counters in D, and in this way we can estimate the memory available for *SBF*, and through E.q 17, we can estimate the FP rate, in fact, the FP rate can be a little larger than the caculated one beacause only a part of elements cause false positives. Therefore our goal is to choose a combination of max, H and P that minimizes the number of FNs under the condition that the FP rate is within a confirmed threshold and a fix amount of space.

In fact, a detailed discussion of how to choose these parameters has been given in [11], which can concluded as: A formula can be obtained to compute the value of P from other parameters provided that max and H have been chosen already; then find the optimal values of H for each case of $max(1, 3, 7)$ by trying limited number(≤ 10) of values of H on the FN rate formulas; Last, max can be set empirically, and specially in the case that no prior knowledge of the input data is available, max was suggested to be 1;

*5) **Time complexity**:* In this section, we will analyze the time complexity of *SBFSS*.

For *SBFSS*, the amount of space has been confirmed, so we just focus on time complexity.

Theorem 8: For *SBFSS*, Processing each data stream element needs $O(1)$ time, independent of the size of space and the stream.

Proof: Like *BFSS*, the processing time for each element also consists two parts: i) the time spent updating *SBF*; ii) the time spent updating D. In fact, from Algorithm 1 and Algorithm 3, we can see that *BFSS* and *SBFSS* share the same operations on D, so for *SBFSS*, the time spent updating D is constant once ϕ is confirmed, which means the size of space has no impact on it. For the first part, a detailed explaination has been given in [11] to prove that the time spent updating *SBF* is O(1), independent of the size of space and the stream. Therefore the time complexity of *SBFSS* is O(1). ■

IV. Experiments

In this section, we first describe our data sets and the implementation details of *BFSS* and *SBFSS*, then we tested the performance of *BFSS* under different parameter settings on both real and synthetic data sets, then we compared the results against the method *nCount* and the method *rCount*. We were insterested in the *recall*, the number of correct elements found as a percentage of the number of correct elements, and the *precision*, the number of correct elements found as a percentage of the entire output, then we calculated F_1. We also measured the space used by *BFSS*, *nCount* and *rCount*, which is essential to handle data streams, however, due to the page limit, we can not show the results of synthetic data here. For *SBFSS*, we compared the recall and precision against the method *lCount* on real data in different size of limited space. The detailed implementations of *nCount*, *rCount* and *lCount* will be given later. Last, we give a summarization of the experimental results. All the algorithms were compiled using the same compiler, and were run on a AMD Opteron(tm) 2.20GHz PC, with 64GB RAM, and 1.8TB Hard disk.

A. Data Sets

Real Data. For real data experiments, we used the dataset[3] which consists of all the requests made to the 1998 World Cup Web site between April 30, 1998 and July 26, 1998. We extracted 7×10^7 URLs in total.

Synthetic Data. We generated several synthetic Zipan data sets with the Zipf parameter varying from 0.5, which is very slightly uniform, to 3.0, which is highly skewed, with a fixed increment of 0.5. The size of each data set, N, is 10^7, and the alphabet was of size 10^6.

B. Implementation Issues

***BFSS* Implementation.** It is simple and straight forward to implement *BFSS*: 1) hash each incoming stream element into H numbers, and set the corresponding H bits to 1. 2) we maintain a min heap to index each counter in D, if there exists an empty counter, we just insert a new counter into the heap, otherwise we replace the top counter of the heap, at last we readjust the heap. 3) when a query comes, we just do as Algorithm 2 indicates.

***SBFSS* Implementation.** The only difference between *BFSS* implementation and *SBFSS* implementation is the implementation of *BF* and *SBF*. For *SBF*, we generate a random number in each iteration, decrement the corresponding cell and $(P - 1)$ cells adjacent to it by 1; this process is faster than generating P random numbers for each element; although the processes of picking the P cells are not independent, each cell has a probability of P/k for being picked at each iteration. Our analysis still holds. Then we hash each incoming stream element to H numbers, and set the corresponding H cells to max. Taking the number of *FNs*, the number of *FPs*, updating time and limited space we can use into consideration, we set $max = 3, H = 4$ and $P = 1$.

***nCount* Implementation.** The basic idea of *nCount* is to allocate each distinct element in S a counter, we use a hash table to index these counters to speedup updating, and it is time consuming if we use a linked list or an array. When an element comes, we increment the corresponding counter using hash table. When a query comes, we traverse the hash table and output the low-frequency elements.

***rCount* Implementation.** *rCount* shares a similar idea with *nCount*, however, unlike *nCount*, *rCount* uses Rabin's method [] to hash each element in S to a 64-bit fingerprint, or it will consume too much space especially when the elements are URLs. With this fingerprinting technique, there is a small chance that two dierent URLs are mapped to the same fingerprint. When an element comes, we first calculate its fingerprint and then increment the corresponding counter using hash table. When a query comes, we traverse the alphabet A and calculate the fingerprint of each element in A, then we check the corresponding counter and output it if it is a low-frequency element.

[3] http://ita.ee.lbl.gov/html/contrib/WorldCup.html

***lCount* Implementation.** *lCount* maintains a limited and fixed space, like *rCount*, *lCount* uses Rabin's method in order to adjust the space consumption through the maximum number of counters *lCount* can maintain. In addition, we use a max heap to index the counters in *lCount*. When an element comes, if there exists an empty counter, we insert a new counter into the heap, otherwise we replace the top counter of the heap, i.e. the counter with the maximum value, and set the value of the new counter to 1, at last we readjust the heap. When a query comes, we traverse the alphabet A and calculate the fingerprint of each element in A, then we check if the corresponding counter exists in the heap and output it if it is a low-frequency element.

C. Performance of BFSS

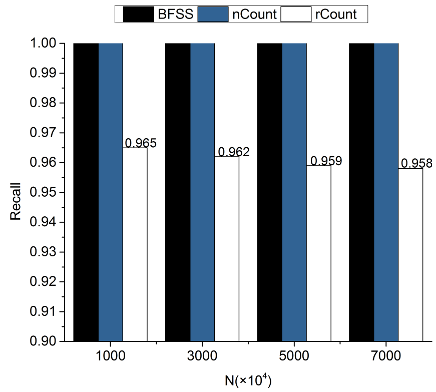
The query issued for *BFSS*, *nCount* and *rCount* was to find all elements, with frequency no more than $\frac{N}{10^3}$. The results comparing the recall, precision, F_1 and space used by the algorithms are summarized in Fig. 4. N was varied from 10^7 to 7×10^7, and $\phi = 0.001$.

The recalls achieved by *BFSS* and *nCount* were constant at 1 for all values of N, as is clear from Fig. 4a, however, due to the existence of hash collisions, *rCount* may neglect a few low-frequency items, and the recall of *rCount* remained above 0.95. From Fig. 4b, with the increasing of N, the precision of *BFSS* decreased ranging from 0.969 to 0.943 since the false positive rate of *BF* increased with the increasing of N, and the precision of *rCount* increased ranging from 0.931 to 0.978 because only the elements not appearing in S might cause false positives which can be observed from the detailed implementation of *rCount*, and the number of these elements decreased with the increasing of N, in this way the precision of *rCount* would increase with the increasing of N, in addition, the precision of *nCount* remained unchanged at 1 because *nCount* just kept a counter for each element in S. Being concerned with both precision and recall, we calculated the values of F_1 of these algorithms based on the precisions and recalls as Fig. 4c shows, and we can find that *BFSS* outperformed *rCount* for all values of N. The advantage of *BFSS* is evident in Fig. 4d, which shows that *BFSS* achieved a reduction in the space used by a factor ranging from 6 when N was 10^7 up to 16 when N was 7×10^7, and it is interesting to note that with the increasing of N, the space used by *nCount* and *rCount* increased rapidly since more counters were needed with the increasing of N, however, the space used by *BFSS* was stable and even decreased a little with the increasing of N because the space complexity is not related to N which can be seen from the analysis of the space complexity of *BFSS*, and the minor difference between the space used by *BFSS* of different N existed in the space used by D.

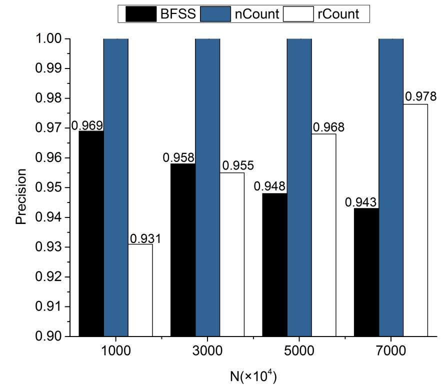
Table III shows the performance of *BFSS* with different parameter settings. We can see that the precision increased with the increasing of K, and once K and N were confirmed the precision was almost confirmed too, which means ϕ had little affection on the precison of *BFSS* and the most *FPs* were caused by *BF*.

D. Performance of SBFSS

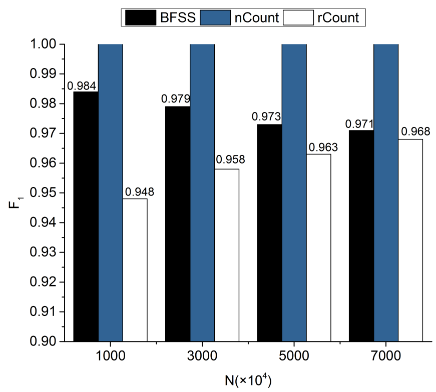
As mentioned before, when the total space is limited, the space left for the Bloom filter is limited as well, and it may



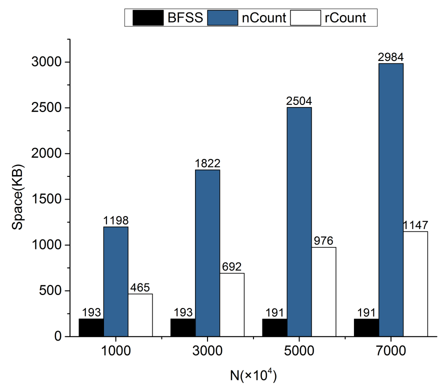
(a) Recall for 0.001-BLFE



(b) Precision for 0.001-BLFE



(c) F_1 for 0.001-BLFE



(d) Space for 0.001-BLFE

Fig. 4: Performance Comparison between *BFSS*, *nCount* and *rCount* Using Real Data - Varying N

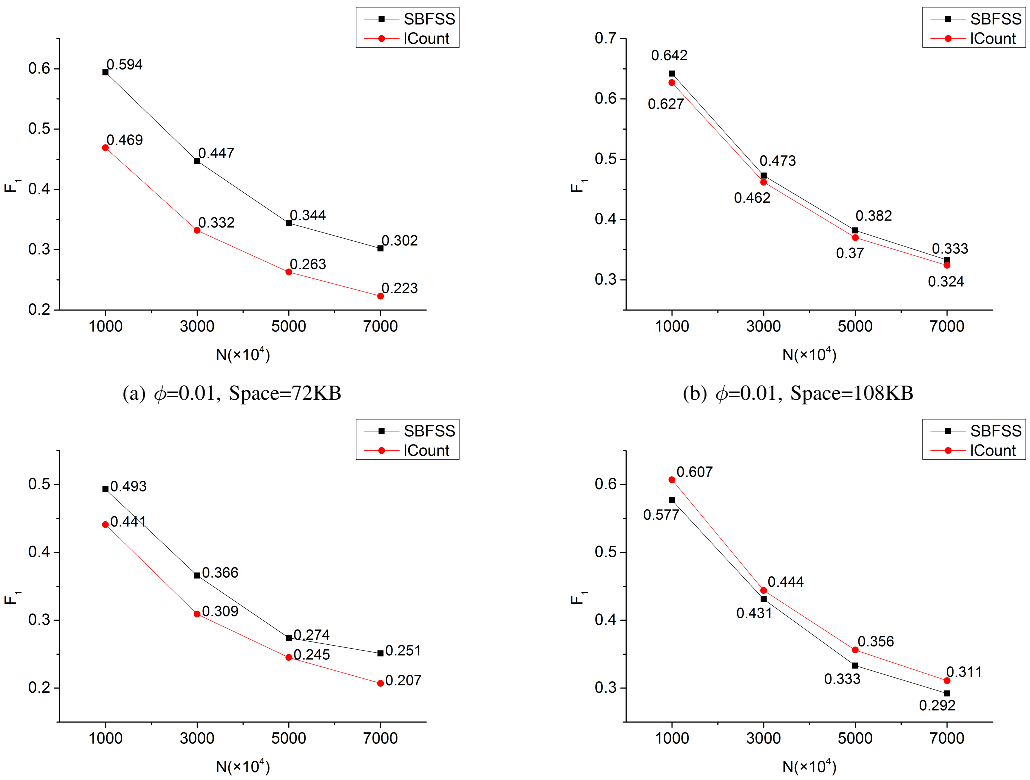
TABLE III: Performance of *BFSS* Using Real Data - Varying ϕ, K and N

ϕ	K	Space	Precision				Recall			
			$N = 10^7$	$N = 3 \times 10^7$	$N = 5 \times 10^7$	$N = 7 \times 10^7$	$N = 10^7$	$N = 3 \times 10^7$	$N = 5 \times 10^7$	$N = 7 \times 10^7$
0.01	270000	$49KB$	0.970	0.959	0.949	0.944	1	1	1	1
	400000	$66KB$	0.984	0.979	0.974	0.972	1	1	1	1
	800000	$116KB$	0.996	0.994	0.992	0.992	1	1	1	1
0.001	270000	$193KB$	0.969	0.958	0.948	0.943	1	1	1	1
	400000	$210KB$	0.984	0.978	0.974	0.972	1	1	1	1
	800000	$260KB$	0.996	0.994	0.993	0.992	1	1	1	1

TABLE IV: Performance Comparison between *SBFSS* and *lCount* Using Real Data in Limited Space - Varying ϕ, N

ϕ	Space	Precision (SBFSS / lCount)				Recall (SBFSS / lCount)			
		$N = 10^7$	$N = 3 \times 10^7$	$N = 5 \times 10^7$	$N = 7 \times 10^7$	$N = 10^7$	$N = 3 \times 10^7$	$N = 5 \times 10^7$	$N = 7 \times 10^7$
0.01	$72KB$	$1/0.920$	$1/0.930$	$0.999/0.940$	$1/0.959$	$0.432/0.315$	$0.288/0.202$	$0.208/0.153$	$0.178/0.126$
	$108KB$	$1/0.919$	$1/0.932$	$1/0.942$	$1/0.960$	$0.473/0.476$	$0.310/0.307$	$0.236/0.230$	$0.200/0.195$
0.001	$72KB$	$0.995/0.886$	$0.980/0.906$	$0.982/0.922$	$0.990/0.907$	$0.328/0.294$	$0.225/0.186$	$0.159/0.141$	$0.144/0.117$
	$108KB$	$0.997/0.897$	$0.987/0.904$	$0.987/0.926$	$0.994/0.952$	$0.406/0.459$	$0.276/0.294$	$0.200/0.220$	$0.171/0.186$

cause many *FPs* when the space is small. Table IV shows the performance comparison between *SBFSS* and *lCount* when the space was limited. When ϕ was 0.01, *SBFSS* outperformed *lCount* in both precision and recall, and we can see that the precision of *SBFSS* was stable and constant at 1 for most values of N, and the precision of *lCount* increased with the increasing of N the reason of which is much like *rCount*. In addition, the recalls of *SBFSS* and *lCount* decreased with N since the space was limited and the information we could maintain was limited too, which directly caused the decreasing of the recalls. When ϕ was 0.001, the rules of how the precisons and recalls changed was similar to the case when ϕ was 0.01, and we



(a) ϕ=0.01, Space=72KB

(b) ϕ=0.01, Space=108KB

(c) ϕ=0.001, Space=72KB

(d) ϕ=0.001, Space=108KB

Fig. 5: Performance Comparison of F_1 between *SBFSS* and *lCount* - Varying ϕ and Space

can find that *SBFSS* still maintained high accuracy, however, the general performance of *SBFSS* was not as good as the case when ϕ was 0.01 because much more space could be allocated to *SBF* when ϕ was smaller and the total space was limited. Another interesting observation is that *SBFSS* performed better when the limited space is small, which can be clearly seen from Fig. 5, which shows the comparison of F_1 between *SBFSS* and *lCount*, especially when ϕ was 0.001 and space was 108KB, the performance of *lCount* was even better than *BFSS*, as is clear from Fig. 5d, however, *SBFSS* mainly applies to the case when the space is limited and small or *BFSS* will perform better.

V. Conclusion

Low-frequency items in data streams contain much useful information which can be widely used in commercial decision-making, data analysis, etc., and locating them is the first step. We present *BFSS*, an effective and space efficient algorithm that aims to solve the problem *ϕ-Bounded Low-Frequency Elements* approximately. *BFSS* can output all low-frequency items, i.e. no *FNs*, and only a few frequent items, the number of which can be theoretically bounded. The experimental results show the recall of *BFSS* is 1 and the precision can be above 0.9 using a small space. When the memory is limited and *BFSS* can not guarantee high precision, we propose *SBFSS* which can guarantee high precision and theoretically bounded recall.

The future directions of our work can be, but are not limited to: 1) output low-frequency items along with their approximate frequencies. 2) handle sliding window queries. 3) support both insertion and deletion of elements. 4) find low-frequency item sets. In fact, like frequent elements, there are many work can be done towards low-frequency elements, however, due to the special features of them, we still have a long and tough way to go and we just take a small step forward.

Acknowledgment

The authors would like to thank...

References

[1] S. Gunduz and M. Ozsu, “A web page prediction model based on click-stream tree representation of user behavior,” in *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2003, pp. 535–540.

[2] J. Chen, D. DeWitt, F. Tian, and Y. Wang, “Niagaracq: A scalable continuous query system for internet databases,” in *Proceedings of the 19th ACM SIGMOD International Conference on Management of Data*, 2000, pp. 379–390.

[3] Y. Zhu and D. Shasha, “Statstream: Statistical monitoring of thousands of data streams in real time,” in *Proceedings of the 28th ACM VLDB International Conference on Very Large Data Bases*, 2002, pp. 358–369.

[4] P. Bonnet, J. Gehrke, and P. Seshadri, “Towards sensor database systems,” in *Proceedings of the 2nd IEEE MDM International Conference on Mobile Data Management*, 2001, pp. 3–14.

[5] C. Cortes, K. Fisher, D. Pregibon, A. Rogers, and F. Smith, “Hancock: A language for extracting signatures fromdata streams,” in *Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2000, pp. 9–17.

[6] E. Demaine, A. Lopez-Ortiz, and J. Munro, "Frequency estimation of internet packet streams with limited space," in *Proceedings of the 10th ESA Annual European Symposium on Algorithms*, 2002, pp. 348–360.

[7] N. Alon, Y. Matias, and M. Szegedy, "The space complexity of approximating the frequency moments," in *Proceedings of the 28th ACM STOC Symposium on the Theory of Computing*, 1996, pp. 20–29.

[8] P. Haas, J. Naughton, S. Sehadri, and L. Stokes, "Sampling-based estimation of the number of distinct values of an attribute," in *Proceedings of the 21st ACM VLDB International Conference on Very Large Data Bases*, 1995, pp. 311–322.

[9] D. M. Kane, J. Nelson, and D. P. Woodruff, "An optimal algorithm for the distinct elements problem," in *Proceedings of the 29th ACM PODS Symposium on Principles of Database Systems*, 2010, pp. 41–52.

[10] M. Datar, A. Gionis, P. Indyk, and R. Motwani, "Maintaining stream statistics over sliding windows," in *Proceedings of the 13th ACM SIAM Symposium on Discrete Algorithms*, 2002, pp. 635–644.

[11] F. Deng and D. Rafiei, "Approximately detecting duplicates for streaming data using stable bloom filters," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, 2006, pp. 25–36.

[12] A. Metwally, D. Agrawal, and A. E. Abbadi, "Duplicate detection in click streams," in *Proceedings of the 14th WWW International World Wide Web Conference*, 2005, pp. 12–21.

[13] X. Lin, H. Lu, J. Xu, and J. Yu, "Continuously maintaining quantile summaries of the most recent n elements over a data stream," in *Proceedings of the 20th IEEE ICDE International Conference on Data Engineering*, 2004, pp. 362–374.

[14] A. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss, "Surng wavelets on streams: One-pass summaries for approximate aggregate queries," in *Proceedings of the 27th ACM VLDB International Conference on Very Large Data Bases*, 2001, pp. 79–88.

[15] J. Gehrke, F. Korn, and D. Srivastava, "On computing correlated aggregates over continual data streams," in *Proceedings of the 20th ACM SIGMOD International Conference on Management of Data*, 2001, pp. 13–24.

[16] P.Bose, E.Kranakis, P.Morin, and Y.Tang, "Bounds for frequency estimation of packet streams," in *Proceedings of the 10th SIROCCO International Colloquium on Structural Information and Communication Complexity*, 2003, pp. 33–42.

[17] G.Cormode, F.Korn, S.Muthukrishnan, and D.Srivastava, "Diamond in the rough: Finding hierarchical heavy hitters in multi-dimensional data," in *Proceedings of the 23rd ACM SIGMOD International Conference on Management of Data*, 2004, pp. 155–166.

[18] S. G.Cormode, F.Korn, "Whats hot and whats not: Trackingmostfrequentitems dynamically," in *Proceedings of the 22nd ACM PODS Symposium on Principles of Database Systems*, 2003, pp. 296–306.

[19] Q. Huang and P. P. C. Lee, "Ld-sketch: A distributed sketching design for accurate and scalable anomaly detection in network data streams," in *Proceedings of the 33rd IEEE INFOCOM International Conference on Computer Communications*, 2014, pp. 1420–1428.

[20] A. Metwally, D. Agrawal, and A. E. Abbadi, "Efficient computation of frequent and top-k elements in data streams," in *Proceedings of the 10th International Conference on Database Theory (ICDT'05)*, Jan 2005, pp. 398–412.

[21] Q. Huang and P. P. C. Lee, "Distributed top-k monitoring," in *Proceedings of the 22nd ACM SIGMOD International Conference on Management of Data*, 2003, pp. 28–39.

[22] A. Lall, V. Sekara, M. Ogihara, J. Xu, and H. Zhang, "Data streaming algorithms for estimating entropy of network traffic," in *ACM SIGMETRICS*, Jun 2006.

[23] G. S. Manku and R. Motwani, "Approximate frequency counts over data streams," in *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB'02)*, Aug 2002.

[24] G. Cormode and M. Hadjieleftheriou, "Methods for finding frequent items in data streams," *The VLDB Journal*, vol. 19, no. 1, pp. 3–20, 2010.

[25] H. Liu, Y. Lin, and J. Han, "Methods for mining frequent items in data streams: an overview," *Knowledge and Information Systems*, vol. 26, no. 1, pp. 1–30, 2011.

[26] G. Cormode and S.Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[27] C. Jin, W. Qian, C. Sha, J. X. Yu, and A. Zhou, "Dynamically maintaining frequent items over a data stream," in *Proceedings of the 12th International Conference on Information and Knowledge Management*, 2003, pp. 287–294.

[28] L. Adamic. Zipf, power-laws, and pareto - a ranking tutorial. [Online]. Available: http://www.hpl.hp.com/research/idl/papers/ranking/ranking.html