SFBF: Scalable and Flexible Bloom Filter for Ever-increasing Big Data

Kun Xie[1], *Member, IEEE*, Xin Wang[1], *Member, IEEE*, Heng Tao[1], Yinghua Min[2], *Fellow, IEEE*, Gaogang Xie[2], *Member, IEEE*, and Keqin Li[3] *Fellow, IEEE*
[1] Department of Electrical and Computer Engineering, State University of New York at Stony Brook, USA
[2] Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100191, China
[3] Department of Computer Science, State University of New York, New Paltz, NY 12561, USA
cskxie@gmail.com, xwang@ece.sunysb.edu, franztaoheng@gmail.com, min@ict.ac.cn, xie@ict.ac.cn, lik@newpaltz.edu

***Abstract*—Bloom filters (BF) allow membership queries over sets with allowable error rate. It is widely used in databases, networks and distributed systems. Despite that BF can work well to represent a static data set by optimally setting the BF parameters to achieve a low false positive rate based on the data set size, when there are dynamic data arrivals thus constant data set changes, it would lead to high false positive errors in BF. To make BF scalable, we propose a Scalable and Flexible Bloom filter (SFBF) to control the false positive rate at a low level even when there are ever-increasing data arrivals and the size of the data set is unpredictable. Specifically, we propose a novel algorithm to adaptively generate hash functions and a light-weight SFBF query algorithm that takes advantage of the features of our proposed hash functions to provide quick BF query. Our performance results demonstrate that SFBF offers very good performance (low false positive rate and low query time) in the presence of dynamic data arrivals, even when the data set becomes very large.**

I. Introduction

Recently, various applications such as information retrieval, telecommunications, transaction records, network monitoring have generated a large amount of data, which makes the data intensive computing a major area of interest both in the industry and in the research community. Efficient storage, management, and processing of the massive amount of data are important and challenging.

As a space efficient probabilistic data structure, Bloom filter (BF) [1] can support the set membership query with a low probability of false positive (i.e., falsely identifying an element to be a member of the set while it is not), no false negative, as well as efficiency in query and storage. The simplicity, easiness of use, straightforward mapping to hardware, and excellent performance (low false positive rate and O(1) update complexity) make Bloom Filter attractive for many applications, especially for applications in distributed systems to share information about available data. [2], [3].

However, with unpredicted large scale and ever-increasing data, there is a big challenge to design an efficient and scalable Bloom filter. Despite the fact that BF can work well to represent a static data set where BF parameters are set to achieve a low false positive rate for a fixed size data set, there are many practical challenges to apply the conventional BF in the presence of "Big Data" for the following reasons:

First, the growth of data set will result in higher false positive rate if the conventional BF is applied for membership query. Dynamically expanding the BF size is an option to solve this problem, however, it needs a well-designed expanding rule to control the false positive rate at low level.

Second, with the size change of the BF, it introduces the need and challenge to find the hash functions that can obtain the hash values within the new BF range at low overhead.

Third, to quickly find the membership of an element in a big data set, it is critical to ensure light-weight BF matching to determine if the set contains the data item. The variations of the BF size and hash functions introduce additional challenge for quick data query.

Although several BF variants [4]–[14] have been proposed in the literature to suit various application needs to reduce the information processing and memory/bandwidth cost, among which, only a very few Bloom filters [13]–[16] have investigated the scalable problem. However, due to the use of static expanding rule or the lack of adaptive hash function design, existing solutions can be hardly applied in practical network environment.

To address these challenging issues for BF to function efficiently in the presence of large and varying data set, we propose a Scalable and Flexible Bloom filter (SFBF) infrastructure to achieve high matching accuracy with low false positive rate. SFBF can control the false positive rate at a low level and can achieve very low query time even when the total number of data items is unpredictable and data items arrive dynamically. Our contributions of this work can be summarized as follows:

- To represent large scale and ever-increasing data, instead of using a single Bloom filter, we consider the use of a BF set consisting of an array of SFBF vectors, with each BF vector added upon need and the length varied based on the flexible expanding speed.
- We propose a novel algorithm to adaptively generate hash functions according to various BF vector length in SFBF. Although multiple BF vectors in SFBF require finding multiple groups of hash values for each data item (one group for one BF vector), in our algorithm, only one group of hash values need to be calculated, from which other groups can be easily deduced though very simple and low cost bit shifting operations.

- We develop a light-weight SFBF query algorithm that takes advantage of the features of our proposed hash functions to quickly detect whether an element is in a data set or not. Although SFBF consists of multiple BF vectors, with our well designed hash functions, we can achieve significant computation complexity reduction for testing element in big data.

The rest of this paper is organized as follows. We review the related work in Section II. We present the basic BF techniques and our problem in Section III. We give an overview of SFBF design in Section IV, and then describe our algorithms for adaptive hash function generation and light-weight SFBF query in Sections VI and VII. In Section VIII, we analyze the performance of the proposed SFBF. We evaluate the performance of the proposed mechanisms through extensive simulations in Section IX, and conclude the work in Section X.

II. Related work

As a good summary data structure, Bloom filter (BF) can compactly represent a set and effectively filter out elements not belonging to the set. For various applications to reduce the information processing and memory/bandwidth cost, several BF variants are proposed for quick membership query. These include, key-value Bloom filter [4], counting Bloom filter [5], compressed Bloom filter [6], spacecode Bloom filter [7], spectral Bloom filter [17], stable Bloom filter [8], reservoir sampling based Bloom filter [9], streaming quotient filter [10], aging Bloom filter [11], and Bloom tree [12] among many.

However, the increase of data amount and unpredicted data arrival speed bring big challenges for designing and efficient and scalable bloom filter. Among current Bloom filter variants, to support dynamic data set, counting Bloom filter [5], stable Bloom filter [8], and aging Bloom filter [11] enable the deletion of old elements from Bloom filter. However, these studies can not be directly applied to solve the scalable problem which is the focus of this paper.

Despite the well-known challenge of applying BF to represent a dynamic set with constant increase of data, only a very limited number of attempts are made to deal with the scalability issue. The dynamic Bloom filter (DBF) in [13], [15] expands the Bloom filter by adding a fixed-size BF vector each time as the set size increases, but the false positive rate increases proportionally with the data set size. Authors in [16] propose to add new BF vector with the length s times that of the previous one added. Without modifying and controlling the hash functions to make hash values fall within the new BF vector range, this scheme may fail when the BF length changes. In our previous work [14], we propose a scheme to keep a low false positive rate by doubling the size of the newly added BF vector each time at the cost of BF space occupancy. Overall, existing studies expand the BF at the same expanding speed (e.g., 1, 2, and s times), which may fail when the data item arrivals are very dynamic.

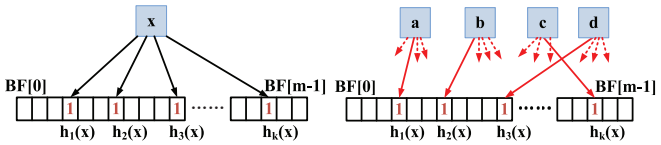
In contrast, we design a scalable and flexible Bloom filter (SFBF) with dynamic and flexible expanding speed. We also propose a novel algorithm to adaptively generate hash functions based on the vector length to expand in SFBF, and our specific hash design allows for fast BF matching.

III. Preliminaries, Motivation and Problem

In this section, we first introduce the fundamentals of basic Bloom filter, we then present the scalability requirement for ever-increasing big data and our design goal.

A. Basic concept of Bloom filter

A Bloom filter is used to represent a set $S = \{s_1, s_2, \cdots, s_n\}$ of n elements from a universe U. It can be represented as an m bit vector with elements $BF[0], BF[1], \cdots, BF[m-1]$, initially all set to 0. In many applications of Bloom filters, the summary message is captured with m-bit vector BF and transmitted in the system. The filter uses k independent hash functions $h_1, h_2, \cdots, h_k$, with each hash function independently mapping an element in the universe to a random number uniformly over the range $0, 1, \cdots, m-1$. For each element $x \in$ S, the bits $BF[h_i(x)]$ are set to 1 for $1 \leq i \leq k$, as shown in Fig.1(a). Given an item y, we check its membership by examining the BF whether the bits at positions $h_1(y), h_2(y), \cdots, h_k(y)$ are set to 1. If not, then y is definitely not a member of S. If $h_i(y)$ $(1 \leq i \leq k)$ are all set to 1, we assume that y is in S, although it can be wrong with some probability. As every element is hashed into the Bloom filter, the source element can not be observed from the Bloom filter. Therefore, the privacy of the source element is protected.



(a) Insertion operation (b) False positive

Fig. 1. Basic concept of Bloom filter.

As a probabilistic data structure, the Bloom-filter query may yield a false positive, i.e., the hashing results indicate that the element is in a data set while it actually does not. Suppose x in Fig.1(b) is not in the set, but elements a, b, c, d in the set have already set the corresponding bits to 1. Thus a false positive occurs. After n elements are hashed into Bloom filter, the probability that a specific bit is still 0 can be expressed as: $p = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{-kn/m}$. The false positive rate of a Basic Bloom filter can be computed as

$$f = (1-p)^k = \left(1 - e^{-kn/m}\right)^k. \tag{1}$$

B. Scalability requirement for big data

From (14), it is easy to observe that the false positive rate increases when n increases. For a given false positive rate required, f_0, the maximum number of elements that a Basic Bloom filter with length m can support is

$$n_0 = -\left(\ln\left(1 - e^{\ln f_0/k}\right) \cdot m\right)/k. \tag{2}$$

We call n_0 the capacity of the Basic Bloom filter with the length m and k hash functions.

Although a basic Bloom filter can work well to represent a static set whose size is known before the design and deployment, it may lead to a high false positive rate with dynamic data arrivals. When the number of data items represented by a BF

$n > n_0$, the false rate would be larger than f_0, and even approach 1. This would make every new coming data falsely reported to be in the set while it is not, which could render the BF useless. Simply setting m to a large value would help to accommodate more data, but it would lead to high storage and communication overhead as Bloom filters are usually applied as messages to exchange among distributed nodes in a network.

In addition, without the prior knowledge of the size of the data set, it is also difficult to properly determine the value of m. Instead of designing BF with a static vector length, it would be more efficient to set a proper BF size initially and scale the Bloom filter when the number of data items increases beyond a preset capacity.

C. Design Goal

To control the false positive rate at low level in the presence of varying data set and data arriving speed, we aim to develop a scalable and flexible bloom filter (SFBF) to support the following three operations:

- insert (item a): insert item a into the SFBF
- extend: extend the SFBF when the size of data set expands beyond the capacity of the current SFBF
- query (item a): query whether item a is in the data set or not

Instead of providing a static expanding solution to handle the scalability problem, the BF design should take flexible and dynamic expanding rate into consideration in the presence of dynamic set changes in network and distributed systems.

IV. Design Overview

We design a Scalable and Flexible Bloom filter (SFBF) to control the false positive rate at a low level even when the data items arrive un-predictedly.

As shown in Fig.2, instead of using a single BF, we consider the use of a BF set consisting of a number of BF vectors to build the SFBF. Originally, SFBF has only one Bloom filter vector, which we call the baseline Bloom filter vector and is denoted as $SFBF_0$. When the data set becomes large with its size exceeding the capacity of the baseline bloom filter, a new bloom filter vector is added to represent the newly arrival elements. The process will continue and more Bloom filter vectors will be added upon need for dynamically representing the expanding data set.

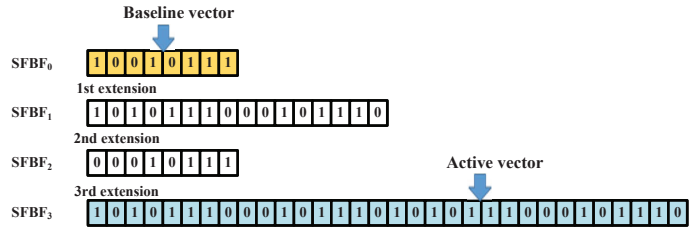


Fig. 2. Scalable and Flexible Bloom filter design for dynamic data set.

For a SFBF which has expanded i times thus consisting of $i+1$ BF vectors, according to the adding sequence of each BF vector, the vectors can be denoted as $SFBF_0$, $SFBF_1$, $\cdots$, $SFBF_i$. Among all the BF vectors, the last added one (i.e.,$SFBF_i$) is called the active Bloom filter which is applied to represent the newly arrival data items. We utilize n_l to record and track the number of items having been represented by the active Bloom filter vector $SFBF_i$.

The length of the baseline Bloom filter vector is called the baseline length of SFBF, denoted as m_0. According to (2), given the tolerable false positive rate f_0, baseline length m_0, and the number of hash functions k, the capacity of the baseline BF vector can be calculated as $n_0 = -\left(\ln\left(1-e^{\ln f_0/k}\right)\cdot m\right)/k$.

From (14), we observe that a BF vector's false positive rate $f = \left(1-e^{-kn/m}\right)^k$ depends on the number of hash functions k and $\frac{n}{m}$. In our construction process, to ensure the query performance of different vectors to have the same tolerable false positive rate f_0, we follow an ***expanding principle***. Specifically, we use the same number of hash functions k for each added BF vector, and our expanding will keep the $\frac{n_j}{m_j}$ of every BF vector $SFBF_j$ to be the same ($0 \le j \le i$). More especially, the capacity n_j expands proportionally to the length m_j of the added BF vector to keep $\frac{n_j}{m_j}$ the same, i.e., $\frac{n_j}{m_j} = \frac{n_0}{m_0}$.

Generally, a long BF vector can support more data items at the cost of larger space, while a short BF vector may result in frequent BF expansion. Different from DBF [13], [15] which adds a fixed-size BF vector each time as the set size increases, our SFBF adds BF vector with the vector length determined by the expanding speed. In order to better trade off between the BF extension cost and the space needed, the expanding speed in our paper varies according to practical application requirements.

Algorithm 1 Insertion and Extension Operations

Input: SFBF: An SFBF which extends i times with $i+1$ Bloom filter vectors $SFBF_0$, $SFBF_1$, $\cdots$, $SFBF_i$;
a: A newly arrival data item;
λ_{i+1}: The expanding speed for the $(i+1)$-th extension;
$SFBF_i$: the active Bloom filter vector with its capacity n_i;
n_l: The No. data items represented by the active vector $SFBF_i$

Output: The update SFBF.

1: **if** $n_l = n_i$ **then**
2: Add $SFBF_{i+1}$ with $m_{i+1} = 2^{\lambda_{i+1}-1}\cdot m_0$
3: $n_{i+1} = 2^{\lambda_{i+1}-1}\cdot n_0$
4: $i = i+1$
5: $n_l = 0$
6: **end if**
7: **for** $1 \le j \le k$ **do**
8: $SFBF_i[h_j(a)] = 1$
9: **end for**
10: $n_l = n_l + 1$

Algorithm 1 shows detailed data item insertion and extension operations for SFBF. For each data insertion, we should first identify whether the SFBF needs an extension or not. If the SFBF needs an extension, a new Bloom filter vector is added and set as the active Bloom filter. All newly arrival data will be inserted and represented by the active Bloom filter vector.

As shown in line 1, if the number of data items represented by the active Bloom filter vector (i.e., n_l) reaches its capacity (i.e.,n_i), the SFBF needs a extension. Given the expanding speed for the $i+1$-th extension (i.e.,λ_{i+1}), a new bloom filter vector

$SFBF_{i+1}$ with the length $m_{i+1} = m_0 2^{\lambda_{i+1}-1}$ is added. Note that the length of each newly added Bloom filter is $2^{(\lambda_{i+1}-1)}$ times of the baseline length. In Section VII, we will utilize this good properties of BF vector length to design the hash functions for SFBF and the quick SFBF query algorithm.

According to the expanding principle, to make $\frac{m_{i+1}}{n_{i+1}} = \frac{m_0}{n_0}$, as shown on line 3, the capacity of the newly added Bloom filter vector is $n_{i+1} = 2^{\lambda_{i+1}-1} \cdot n_0$. Then as shown on line 4, the newly added Bloom filter vector is set as active Bloom filter vector. A newly arriving data item is inserted to SFBF by setting the corresponding k locations of the active bloom filter vector to 1. Future data items will be hashed to the newest SFBF vector, until the number of new arriving data items exceeds its capacity and a new SFBF vector is added.

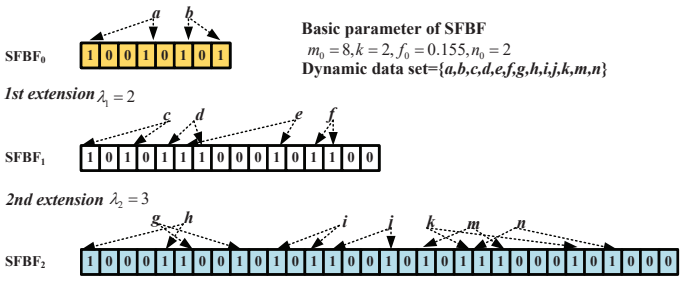


Fig. 3. Example of SFBF.

Fig.3 illustrates how an SFBF works. The SFBF is utilized to represent a dynamic data set with sequentially arriving data items $a, b, c, d, e, f, g, h, i, j, k, m, n$. The baseline Bloom filter vector is $SFBF_0$ with the length $m_0 = 8$ and the number of hash functions $k = 2$. For a better explanation, we assign a big false positive rate $f_0 = 0.155$ for this Bloom filter, and the Bloom filter can support $n_0 = 2$ data items according to (2).

When the data item c arrives, two data items a and b have been represented by $SFBF_0$, and $n_l = n_0 = 2$. According to the Step 1 of Algorithm 1, a new SFBF vector should be added. Given the expanding speed $\lambda_1 = 2$, according to the Step 2 of Algorithm 1, we can create a double-size new $SFBF_1$ of 16 bits to represent the new data item c.

When the data item g arrives, SFBF needs an extension again because $SFBF_1$ has already represented data items c, d, e, f, and the number of data items represented n_l is equal to $SFBF_1$'s capacity $n_1 = 4$. Given the expanding speed $\lambda_2 = 3$, a new $SFBF_2$ of 32 bits is created to represent the new data items. Therefore, all the data items $a, b, c, d, e, f, g, h, i, j, k, m, n$ can be represented in SFBF by an array of BF vectors $SFBF_0$, $SFBF_1$, and $SFBF_2$ with only two extensions.

V. Challenge of SFBF

To make the Bloom filter scalable with the size of the data set, the key technique taken by SFBF is to add a new BF vector upon need. The length of the BF vector is determined by the expanding speed instead of being fixed. As a result, SFBF usually consists of multiple BF vectors with various lengths.

Conventionally, a group of hash functions are preset to support a BF vector, and the value of each hash function will fall within the length range of the BF. As an array of BF vectors are used in SFBF with the length of each vector determined by the expanding speed, each BF vector needs its own group of hash functions according to its length.

In the example of Fig.3, three BF vectors $SFBF_0$, $SFBF_1$, and $SFBF_2$ are created with the vector lengths m_0, $2m_0$, and $4m_0$, respectively. Thus, three groups of hash functions need to be generated, with each having k hash functions. The hash range of each group should match the length of the corresponding BF vector, as shown in Fig.4.

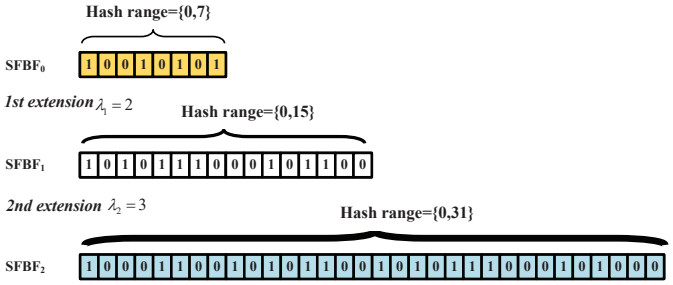


Fig. 4. Hash range requirement.

If SFBF is applied to determine whether an element is a member of the data set, all the BF vectors should be tested in the worst case which would involve $(i + 1) * k$ hash computation. Although one group of hash functions can be applied to the same-length Bloom filter vectors, as SFBF consists of multiple BF vectors with various lengths, the query algorithm will involve a lot of hash computations. The high computational cost is the main challenge in realizing SFBF.

To simplify the hash function generation procedure and more importantly to ensure light-weight query and low-cost maintenance of SFBF, we propose an algorithm to adaptively generate hash functions according to the length of BF vector in Section VI, based on which, we further propose a quick BF query algorithm in Section VII.

VI. Adaptive Hash function Generation

In this section, we first introduce the basic H_3 hash function, and then propose a new adaptive hash function generation algorithm.

A. Review of H_3 hash function

H_3 hash function is a Universal class of hash function introduced by L.Carter and M.Wegman [18]. Requiring only Boolean AND and XOR operations to calculate the key's hash value, H_3 hash functions are easy to implement using either software or hardware [19], [20].

Each hash function in H_3 class is a linear transformation $B^T = Q_{r\times w}A^T$ that maps a w-bit binary string $A = a_1 a_2 \cdots a_w$ to an r-bit binary string $B = b_1 b_2 \cdots b_r$ as follows:

$$\begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_r \end{pmatrix} = \begin{pmatrix} q_{11} & q_{12} & \cdots & q_{1w} \\ q_{21} & q_{22} & \cdots & q_{2w} \\ \cdots & \cdots & \cdots & \cdots \\ q_{r1} & q_{r2} & \cdots & q_{rw} \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_w \end{pmatrix} \quad (3)$$

where A and B are the input key (index) and its hash value, and the hash generation matrix $Q_{r\times w}$ is an $r \times w$ matrix defined

over $GF(2) = \{0, 1\}$ with each H_3 hash function uniquely corresponding to such a $Q_{r\times w}$. The hash function of $Q_{r\times w}$ can map the key ranged in $\{0, 2^w - 1\}$ to a hash value ranged in $\{0, 2^r - 1\}$.

The multiplication and addition in $GF(2)$ is Boolean AND($\bullet$) and XOR($\oplus$), respectively. According to (3), each bit of B is calculated as follows:

$$b_i = (a_1 \bullet q_{i1}) \oplus (a_2 \bullet q_{i2}) \oplus \cdots \oplus (a_w \bullet q_{iw}) \ (i = 1, 2, \cdots, r) \quad (4)$$

We take two examples to illustrate the H_3 class hash function. In the first example, the hash generation matrix is

$$Q_{2\times 8} = \begin{pmatrix} 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}, \quad (5)$$

where $w = 8$, $r = 2$, and the hash function is used to map the input key (index) to its hash value: $\{0, \ldots, 2^8 - 1 = 255\} \rightarrow \{0, \ldots, 2^2 - 1 = 3\}$. Under this hash function, the hash value for index 69 can be calculated by Eq(3), expressed as follows

$$\begin{aligned} &h_1(69) = h_1(01000101) \\ &= \begin{pmatrix} 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \end{aligned} \quad (6)$$

where $\begin{pmatrix} 1 \\ 0 \end{pmatrix}^T = \begin{pmatrix} 1 & 0 \end{pmatrix} = 2\,(decimal)$, so the hash value of 69 under the hash function h_1 is $h_1(69) = 2$.

In the second example, the hash generation matrix is

$$Q_{3\times 8} = \begin{pmatrix} 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{pmatrix}, \quad (7)$$

where $w = 8$ and $r = 3$. Similarly, the hash value for index 69 can be calculated as follows

$$\begin{aligned} &h_2(69) = h_2(01000101) \\ &= \begin{pmatrix} 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} \end{aligned} \quad (8)$$

and $\begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}^T = \begin{pmatrix} 1 & 0 & 1 \end{pmatrix} = 5\,(decimal)$, then the hash value of 69 under the hash function h_2 is $h_2(69) = 5$.

In these two examples, the same input index 69 is hashed to different values using two different hash functions. When comparing the matrices in Eq.(7) and Eq.(5), we observe that the matrix in Eq.(7) has one more row and their first two rows are the same. In Section VII, we will investigate the relationship of these similar matrices and exploit the hash matrix features to design a light weight query algorithm for SFBF.

B. Hash function generation

From the basic structure of a H_3 hash function, the hash value range is determined by the number of rows of its hash generation matrix. Therefore, we can generate different value-range hash functions using one base matrix, with the number of rows to select from the base matrix adapted according to the length of the BF vector. In order to generate k hash functions for each SFBF vector, we only need one group of k base generation matrices $Q_{R\times w}[1], Q_{R\times w}[2], \cdots, Q_{R\times w}[k]$.

In order to reduce the overhead for hash function generation and speed up BF query, we propose to generate hash functions for different SFBF vectors from the same k base matrices, $Q_{R\times w}[1], Q_{R\times w}[2], \cdots, Q_{R\times w}[k]$. The generation process is presented in Algorithm 2. For a given $SFBF_i$ whose length is m_i, the hash functions are generated by selecting the first l_i rows of $Q_{R\times w}$ of the base generation matrices, where l_i is determined by $l_i = \log_2 m_i$.

Algorithm 2 Adaptive hash function generation

Input: k base hash generation matrices $Q_{R\times w}[1]$,$Q_{R\times w}[2]$, $\cdots$,$Q_{R\times w}[k]$.
A vector $SFBF_i$ with its length m_i.
Output: k hash functions for $SFBF_i$.

1: Calculate the number of rows l_i needed to form the hash generation matrix for $SFBF_i$ according to its length: $l_i = \log_2 m_i$.
2: Select the first l_i rows of the base hash generation matrices $Q_{R\times w}[u], 1 \le u \le k$ to form the hash matrices $Q_{l_i\times w}[u], 1 \le u \le k$ for $SFBF_i$.
3: Return $Q_{l_i\times w}[1], Q_{l_i\times w}[2], \cdots, Q_{l_i\times w}[k]$.

Fig. 5 presents an example to illustrate Algorithm 2. For $SFBF_0$, as $m_0 = 8$, the first $\log_2(m_0) = 3$ rows in the base matrix are extracted to build the hash function. Similar processes are adopted to generate the hash functions for $SFBF_1$ and $SFBF_2$. As the lengths of $SFBF_1$ and $SFBF_2$ are 16 and 32, the hash functions for $SFBF_1$ and $SFBF_2$ are 4 rows and 5 rows, respectively. As the hash functions for different BF vectors are generated from the same base matrix, it creates strong relationship among the hash values calculated for different BF vectors. This feature can be exploited to largely reduce the computation cost for BF query, as shown in Section VII.

With Algorithm 2, the SFBF insertion operation in Algorithm 1 can be described more clearly as follows. During the ith extension, a new $SFBF_i$ is added to SFBF with the vector length $m_i = 2^{\lambda_i - 1} \cdot m_0$, where λ_i is the expanding speed. According to Algorithm 2, the corresponding k hash functions are generated by selecting the first $l_i = log_2 m_i = \log_2\left(2^{\lambda_i - 1} \cdot m_0\right) = \lambda_i - 1 + l_0$ rows of the base matrices where $l_0 = \log_2(m_0)$, denoted as $Q_{l_i\times w}[u], 1 \le u \le k$. All data items to insert into $SFBF_i$ should use these hash functions to find the corresponding bits in $SFBF_i$ and set them to 1.

It is worth pointing out that, although our SFBF consists of multiple SFBF vectors, we only need to store one group of k base matrices $Q_{R\times w}[1], Q_{R\times w}[2], \cdots, Q_{R\times w}[k]$ at the very beginning. To support unpredictably high expanding speed and thus the large length of SFBF vector, the number of rows R in these bash matrices are initially set to a large value. Our generation algorithm is very simple and can support fast BF query as we will see in Section VII.

VII. Low cost SFBF query and maintenance

In this section, we first analyze the special requirement for SFBF query, and then investigate the relationship of hash

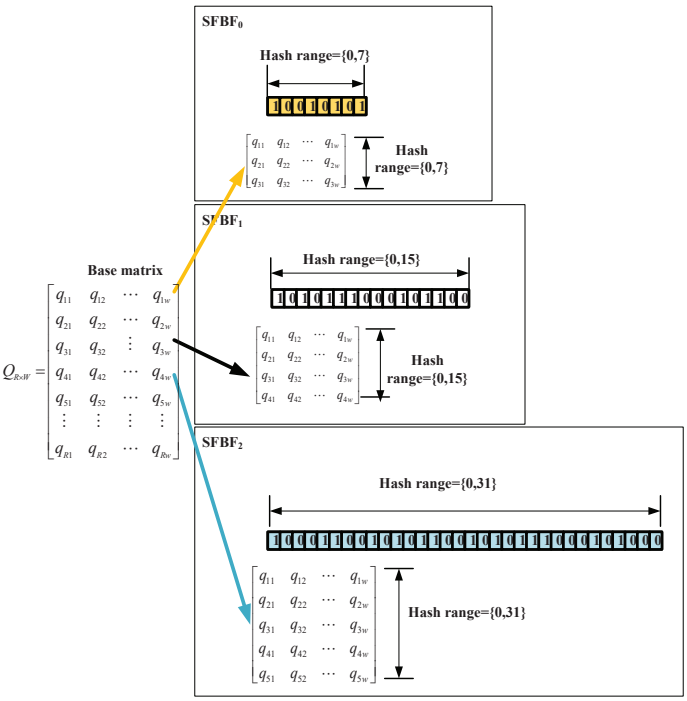


Fig. 5. Different hash functions for different BF vectors are generated from the same base matrix.

functions generated by Algorithm 2. Based on the analysis, we propose a light-weight query algorithm, and present our low-cost SFBF filter maintenance schemes.

A. SFBF query requirement

For an SFBF vector $SFBF_j$, to determine if an data item q has already been inserted into the vector, it will check whether $SFBF_j[h_1(q)], SFBF_j[h_2(q)], \cdots, SFBF_j[h_k(q)]$ with positions calculated by corresponding k hash functions all set to 1. For an SFBF with the initial BF vector plus i extensions, in the worst case, it needs to search through the whole array of SFBF vectors, and the query procedure would require $k(i+1)$ times of hash computations. This not only introduces high computation cost but also large searching time.

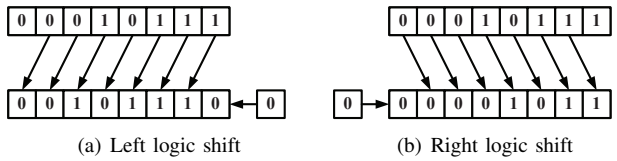
In this section, we design a light weight query algorithm which can largely reduce the total times of hash computations from $k(i+1)$ to k. Before presenting our query algorithm, we first analyze the relationship of the generated hash functions.

B. Relationship of generated hash functions

Among SFBF vectors, we can find the longest vector $SFBF_{max}$ with the length m_{max}, and denote $l_{\max} = \log_2 m_{\max}$. From our proposed hash generation algorithm, the k hash functions for $SFBF_{max}$ can be obtained by taking the first l_{max} rows of k base matrices respectively, denoted as $Q_{l_{\max}\times w}[1], Q_{l_{\max}\times w}[2], \cdots, Q_{l_{\max}\times w}[k]$.

Theorem 1 will show that for a given data item q, if we have the hash values for $SFBF_{max}$, the corresponding hash values for other SFBF vectors under their corresponding hash functions can be easily deduced by simply right-shifting the hash values calculated for $SFBF_{max}$.

Before presenting the Theorem, we first introduce the bit logic shifting operation using two examples in Fig.6. The bit logic shifting is a bitwise operation performing on the binary representation of an integer, where zeros are shifted in to replace the discarded bits. In this paper, we use right logic shift operation ($>>$) to deduce hash values.



(a) Left logic shift (b) Right logic shift

Fig. 6. Logic shift operation.

***Theorem** 1:* Suppose the data item q's hash values under hash functions of $Q_{l_{\max}\times w}[u]$ where $(1 \le u \le k)$ for BF vector $SFBF_{max}$ are $Addr_{max}[u](q)$, then

$$Addr_j[u](q) = Addr_{\max}[u](q) >> (\log_2 m_{\max} - \log_2 m_j) \quad (9)$$

where $Addr_j[u](q)(1 \le u \le k)$ are the hash values of the data item q under the hash functions for vector $SFBF_j$ whose length is m_j.

Proof: The base generation matrix $Q_{R\times w}[u](1 \le u \le k)$ is

$$Q_{R\times w}[u] = \begin{bmatrix} q_{11} & q_{12} & \cdots & q_{1w} \\ q_{21} & q_{22} & \cdots & q_{2w} \\ \vdots & \vdots & \vdots & \vdots \\ q_{l_j 1} & q_{l_j 2} & \cdots & q_{l_j w} \\ \vdots & \vdots & \vdots & \vdots \\ q_{l_{\max}1} & q_{l_{\max}2} & \cdots & q_{l_{\max}w} \\ \vdots & \vdots & \vdots & \vdots \\ q_{R1} & q_{R2} & \cdots & q_{Rw} \end{bmatrix} \quad (10)$$

Let $l_j = \log_2 m_j$. According to Algorithm 2, the hash functions for $SFBF_j$ are hash functions with their hash matrices consisting of the first l_j rows of the base generation matrices, that is $Q_{l_j\times w}[u](1 \le u \le k)$.

Let the bit string of data item q's index be $x = a_1 a_2 \cdots a_w$. The bit strings of the two hash values $Addr_{max}[u](q)$ and $Addr_j[u](q)$ are $s_1 s_2 \cdots s_{l_{\max}}$ and $t_1 t_2 \cdots t_{l_j}$, respectively.

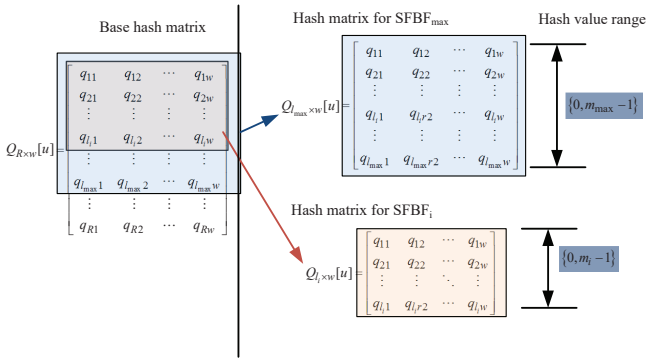


Fig. 7. Hash functions for $SFBF_{max}$ and $SFBF_j$.

According to (4), we can obtain

$$s_i = (a_1 \bullet q_{i1}) \oplus (a_2 \bullet q_{i2}) \oplus \cdots \oplus (a_w \bullet q_{iw}) \ (i = 1, 2, \cdots, l_{\max}) \quad (11)$$

and

$$t_i = (a_1 \bullet q_{i1}) \oplus (a_2 \bullet q_{i2}) \oplus \cdots \oplus (a_w \bullet q_{iw}) \ (i = 1, 2, \cdots, l_j) \quad (12)$$

It is easy to conclude that $s_i = t_i$ for $(i = 1, 2, \cdots, l_j)$ because the hash matrices $Q_{l_j \times w}[u]$ for $SFBF_j$ and $Q_{l_{\max} \times w}[u]$ for $SFBF_{max}$ are generated from the same base matrix, and hash matrices $Q_{l_j \times w}[u]$ and $Q_{l_{\max} \times w}[u]$ have the same first l_j rows, as shown in Fig.7.

Therefore, $Addr_j[u](q)$ can be formed by taking the first l_j bits of the bit-string $Addr_{\max}[u](q)$. This nice feature indicates that $Addr_j[u](q)$ can be easily obtained by simply right-shifting $Addr_{\max}[u](q)$ for $\log_2 m_{\max} - \log_2 m_j$ times without need of recalculating k hash values, i.e.,$Addr_j[u](q) = Addr_{\max}[u](q) >> (\log_2 m_{\max} - \log_2 m_j)(1 \leq u \leq k)$. The proof completes. ∎

We illustrate the hash value deduction method through an example. In the first example of Section VI-A, $h_1(69) = (10)$. In the second example in Section VI-A, $h_2(69) = (101)$. The corresponding hash generation matrices of h_1 and h_2 are in Eq.(5) and Eq.(7), respectively. Obviously, h_1 is the same as the first two rows of h_2. According to Theorem 1, we can obtain $h_1(69) = h_2(69) >> 1 = (101) >> 1 = (10)$, which is exactly the hash value of directly applying hash function h_1 to 69.

Given an data item and its hash values for $SFBF_{max}$, the Theorem 1 indicates that a simple bit shifting operation is good enough for obtaining hash values for other SFBF vectors. In the next section, we will take advantage of this nice feature to reduce the total number of hash computations for SFBF query from $k(i+1)$ to k. This not only significantly reduces the complexity but more importantly ensures the SFBF to be scalable, with its computation complexity kept the same regardless of the number of vectors in SFBF.

C. Query algorithm

Taking advantage of the features of our proposed hash functions, we propose our light weight SFBF query algorithm, as shown in Algorithm 3

When answering a query of the form "Has the data item q been stored and represented by an SFBF?", we first find the longest vector $SFBF_{max}$ and then hash the data item q using matrices $Q_{l_{max} \times w}[u](1 \leq u \leq k)$ to get the hash values $Addr_{max}[u](q)(1 \leq u \leq k)$.

When querying against $SFBF_j$, instead of directly computing the k hash values $Addr_j[u](q)(1 \leq u \leq k)$ by using $Q_{l_j \times w}[u]$, we can exploit the shifting operation to obtain the hash locations of the data item for $SFBF_j$ through $Addr_j[u](q) = Addr_{\max}[u](q) >> (l_{\max} - \log_2 m_j)(1 \leq u \leq k)$. Then by checking whether all these locations in $SFBF_j[Addr_j[u](q)], (1 \leq u \leq k)$ are set to 1, we can complete the data item query against $SFBF_j$.

Although completing a membership query needs to look up all $(i+1)$ SFBF vectors in the worst case, taking advantage of the shifting operations introduced in Theorem 1, it only needs k times of hash computations to obtain the hash locations of a data item in $SFBF_{max}$ and deduce the locations for other SFBF vectors easily. Therefore, our proposed query algorithm can achieve very good performance at low computation cost.

As Bloom filter is usually utilized as an in-memory structure to represent the data set to facilitate duplication detection. According to that, when the data set expands and makes the SFBF reach a memory threshold, we may want to delete the data items that have not been tested for duplication for a long time. Fortunately, it is very easy to support such maintenance operations in SFBF. To achieve this, we can maintain a time-tag for each SFBF-vector, and the time-tap is updated when an data item in this SFBF is matched. To release the storage, we can delete the SFBF vectors with the old time-tag if needed.

Algorithm 3 Light weight SFBF query algorithm

Input: k base hash generation matrices $Q_{R \times w}[1]$,$Q_{R \times w}[2]$, $\cdots$,$Q_{R \times w}[k]$.
Data item q.
SFBF extends i times with $i+1$ SFBF vectors.

Output: whether the data item q has been stored and is duplicated.

1: Among SFBF vectors, find the longest vector $SFBF_{max}$ with the length m_{max}. According to Algorithm 2, the k hash functions for $SFBF_{max}$ can be obtained by taking the first l_{max} rows of k base matrices respectively, denoted as $Q_{l_{\max} \times w}[1], Q_{l_{\max} \times w}[2], \cdots, Q_{l_{\max} \times w}[k]$, where $l_{\max} = \log_2 m_{\max}$.
2: Apply hash functions of $Q_{l_{\max} \times w}[u]$ to data item q to obtain the k hash values of the data item, denoted as $Addr_{\max}[u](q)(1 \leq u \leq k)$.
3: **for** $j = i; j >= 0; j--$ **do**
4: According to Theorem 1, the k hash values of data item q for $SFBF_j$ can be obtained by

$$Addr_j[u](q) = Addr_{\max}[u](q) >> (l_{\max} - \log_2 m_j) \quad (13)$$

for $(1 \leq u \leq k)$, where m_j is the length of $SFBF_j$.
5: Check all the k locations $SFBF_j[Addr_j[u](q)]$ for $0 \leq u \leq k$ are all set to 1, if Yes, return true, otherwise, continue to check other SFBF vectors.
6: **end for**

VIII. SFBF Analysis

In this section, we theoretically analyze the performance of the proposed SFBF from different perspectives.

A. False positive rate

In following theorem, we will show that the false positive rate of our SFBF depends on the SFBF's total extension rounds.

Theorem 2: To represent a data set with n elements, the SFBF has extended i times, includes $i+1$ SFBF vectors where the last $SFBF_i$ represents n_l elements. Then the false positive rate of this SFBF is

$$f^{SFBF} = 1 - \left(1 - \left(1 - e^{-kn_0/m_0}\right)^k\right)^i \left(1 - \left(1 - e^{-kn_l/m_i}\right)^k\right). \quad (14)$$

Proof: For $0 \leq j \leq i-1$, the false positive rate of $SFBF_j$ is $f(m_j, k, n_j) = \left(1 - e^{-kn_j/m_j}\right)^k$. According to the expanding principle adopted in Section IV, as $\frac{n_j}{m_j} = \frac{n_0}{m_0}$, we have $\left(1 - e^{-kn_j/m_j}\right)^k = \left(1 - e^{-kn_0/m_0}\right)^k$ for $0 \leq j \leq i-1$. The false positive rate of the last $SFBF_i$ is $f(m_i, k, n_l) = \left(1 - e^{-kn_l/m_i}\right)^k$. Thus, the false positive rate of the SFBF can

be expressed as :

$$\begin{aligned} f^{SFBF} &= 1 - \prod_{j=0}^{j=i-1} \left(1 - f\left(m_j, k, n_j\right)\right)\left(1 - f\left(m_i, k, n_l\right)\right) \\ &= 1 - \left(1 - \left(1 - e^{-kn_0/m_0}\right)^k\right)^i \left(1 - \left(1 - e^{-kn_l/m_i}\right)^k\right), \end{aligned} \tag{15}$$

which completes the proof. ■

B. CPU time for SFBF querying

CPU time cost for SFBF querying is important in busy network environments. The query time consists of hash function computing time, comparing time, and shifting time.

More specially, given an SFBF which has extended i times (including $i+1$ SFBF vectors) with k hash functions, the hash function computing time is $O(k)$. To check whether an element is in the data set using this SFBF, it only needs k comparison operations and k shifting operations when the element to be queried is represented by the last $SFBF_i$ in an ideally case, while it needs $k(i+1)$ comparison operations and ki shifting operations in the worst case to check all the $(i+1)$ SFBF vectors. Hence the average time complexity of the comparison time and shifting time are expressed as $O((k+k(i+1))/2) = O(k(i+2)/2)$ and $O(k(i+1)/2)$, respectively.

Therefore, the time complexity to query an element against an SFBF is $O(k) + O(k(i+2)/2) + O(k(i+1)/2)$. Obviously, the CPU query time of an SFBF depends on the SFBF's total extension rounds.

C. Space size and load factor

The load factor of an SFBF (denoted as L_{SFBF}) is defined as the ratio of the number of bits in the SFBF to the number of elements represented by the SFBF. Obviously, load factor reflects the space efficiency to represent a data set.

Assume that the SFBF has extended i times and includes $i+1$ SFBF vectors, we can calculate the space size of the SFBF in a straightforward way by summing up the sizes of all SFBF vectors $SFBF_j$ $(0 \le j \le i)$, expressed as

$$M_{SFBF} = \sum_{j=0}^{i} m_j, \tag{16}$$

where $m_j = 2^{\lambda_j - 1} \cdot m_0$ and λ_j is the expanding speed for the j-th extension.

***Theorem** 3:* Given that an SFBF represents a set of n elements with $i+1$ SFBF vectors, the load factor L_{SFBF} of the SFBF satisfies that

$$\frac{m_0}{n_0} \le L_{SFBF} \le \frac{m_0}{n_0} + \frac{m_i}{n_l} \tag{17}$$

where m_0 and n_0 are the size and capacity of the first SFBF vector $SFBF_0$, m_i and n_l are the size of the last SFBF vector $SFBF_i$ and the total number of elements represented by $SFBF_i$.

Proof: From the definition of load factor, we can obtain that

$$L_{SFBF} = \frac{M_{SFBF}}{n} = \frac{\sum_{j=0}^{i} m_j}{\sum_{j=0}^{i-1} n_j + n_l} \tag{18}$$

where m_j and n_j are the size and the capacity of $SFBF_j$. According to the expanding principle introduced in Section IV, we have

$$\frac{m_j}{n_j} = \frac{m_0}{n_0} \qquad for \qquad 0 \le j \le i-1. \tag{19}$$

Furthermore, according to Eq(19), we have

$$L_{SFBF} = \frac{\sum_{j=0}^{i-1} m_j + m_i}{\sum_{j=0}^{i-1} n_j + n_l} \ge \frac{\sum_{j=0}^{i-1} m_j + m_i}{\sum_{j=0}^{i-1} n_j + n_i} = \frac{m_0}{n_0} \tag{20}$$

$$L_{SFBF} = \frac{\sum_{j=0}^{i-1} m_j + m_i}{\sum_{j=0}^{i-1} n_j + n_l} \le \frac{\sum_{j=0}^{i-1} m_j}{\sum_{j=0}^{i-1} n_j} + \frac{m_i}{n_l} = \frac{m_0}{n_0} + \frac{m_i}{n_l} \tag{21}$$

Combining (20) and (21), the proof completes. ■

The last SFBF vector $SFBF_i$ represents n_l elements, which may be much smaller than its capacity n_i. Therefore, we can conclude that the large the load factor of the last SFBF vector $\frac{m_i}{n_l}$, the large the SFBF's load factor.

D. Impact of expanding speed

In this paper, at each extension, the capacity i.e., (n_i) and the size of the newly added $SFBF_i$ (i.e., m_i) are $2^{\lambda_i - 1} \cdot n_0$ and $2^{\lambda_i - 1} \cdot m_0$, respectively, where λ_i is the expanding speed for ith extension.

The expanding speed has an impact on the performance of an SFBF. On the one hand, a large expanding speed may result in large size and large capacity of the newly added SFBF vector, which further makes the size and load factor of SFBF large. On the other hand, the larger the expanding speed, the fewer the extension rounds needed for SFBF to represent a dynamic data set, which further makes the SFBF achieve lower false positive rate and lower query CPU time.

SFBF provides a scalable and flexible design of Bloom filter. The proposed SFBF may have extensive applications, especially the applications that need to deal with continuous increase of data set. To implement SFBF in different network scenarios, we should set a proper expanding speed according to the network scenario by taking into account both the space and performance requirement.

IX. Performance Evaluation

We first describe the experiment setup, then present the experiment results.

A. Experiment Setup

We apply both real-world and synthetic data traces to comprehensively evaluate our proposed SFBF structure. Four data traces are used in the experiments:

- MAWI [21]: This traffic data repository is maintained by the MAWI Working Group of the WIDE Project. The MAWI trace we use contains 15 minutes of daily traffic captured from a trans-Pacific link between Japan and the United States. The data are publicly available, and have the packet payloads omitted and IP addresses anonymous. Originated from January 2001, MAWI project currently maintains more than 14 years of traffic.

- Synthetic trace: All the data items in this trace are randomly generated within the range from 0 to $2^{32}-1$.
- Calgary-HTTP [22]: This trace contains the HTTP requests to the University of Calgary's Department of Computer Science WWW server located at Calgary, Alberta, Canada.
- NASA-HTTP [23]: This trace contains two month's HTTP requests to the NASA Kennedy Space Center WWW server in Florida.

For the first trace MAWI, we extract the Source IP address (32bit) from the MAWI trace as the key to insert into and query against the SFBF. For the synthetic trace, the generated data item is directly used as the key to operate against the SFBF. For the last two traces, we apply MurMurhash's hash to the URL string to generate the 32 bit URL fingerprint, which is used as the key to operate against the SFBF. The detailed information of traces used in this paper are shown in Table I .

TABLE I
REAL TRAFFIC TRACES

data set	data time	No. Items
MAWI 1	2015-01-01 [14:00:01-14:00:15]	58066726
MAWI 2	2015-2-21 [13:59:59-14:14:59]	58523738
Calgary-HTTP	[1995-08-28-00:00:00] [1995-09-10-23:59:59]	3328587
NASA-HTTP	[1995-07-01-00:00:00] [1995-08-31-23:59:59]	3461612

Utilizing above traces, we perform a set of experiments to evaluate the performance of SFBF on a LENOVO workstation equipped with Intel(R) Core(TM)i5-5200U CPU(2.20GHz) and 8.00GB RAM. Each experiment dedicates to a specific SFBF, which is associated with randomly selected k base hash generation matrices of dimension 32×32, i.e., the number of columns is $w = 32$ and the number of rows is $R = 32$. We set the number of columns in the base hash generation matrices to be 32, as the keys we use for all the trace cases are all 32 bits. That is, the IP address in MAWI trace is 32 bits, the URL fingerprint in traces Calgary-HTTP and NASA-HTTP is 32 bits, and the data in the Synthetic trace are generated within the range from 0 to $2^{32}-1$. The length and the capacity of the baseline BF vector in SFBF are set to $m_0 = 1024$ and $n_0 = 64$, respectively. The number of hash functions is set to $k = 6$.

Each experiment includes two steps:

1) Following the insertion operation in Algorithm 1, we use an SFBF consisting of multiple BF vectors to represent the first 3×10^4 data items in the data trace, with data items arriving sequentially. k hash functions are applied to each data item to set k locations of the active BF vector to 1. Starting with the smallest baseline BF vector, when the data expands, SFBF expands and a new BF vector is created and added into SFBF if necessary with the filter length determined by the expanding speed λ. After all data items are represented by SFBF, the space usage of SFBF is known.
2) In order to evaluate the performance, after all data items in a data set have been represented by SFBF, we use another 150,000 data items in the traces to determine the false positive rates of SFBFs. We run the querying Algorithm 3 using the same k hash generation matrices in the insertion step. If the querying process returns true for a data item not in the set, it is counted as a filter false positive. The false positive rate for SFBF is the faction of 1.5×10^5 total queries that are false positive. We also insert a timer to measure the querying time.

Besides SFBF, for performance comparison, we implement DBF proposed in [13] which expands the Bloom filter by adding a fixed-size BF vector each time as the data set becomes larger. For each parameter combination, we repeat our experiments 100 times and take their average.

Moreover, as SFBF is designed to represent a dynamic data set, to evaluate the performance of SFBF under different expanding speed, different expanding speed λ are simulated to control the SFBF's expanding process.

- λ_1: following the sequence: 1,2,3,4,5,7,9,...
- λ_2: following the sequence: 1,3,5,7,9,11,13,...
- λ_3: following the sequence: 1,1,2,2,3,3,4,4,...
- λ_4: randomly within [1,3]
- λ_5: randomly within [1,5]

Under λ_1, when the data set expands, the lengths of each vector added are $n_0, 2n_0, 2^2n_0, 2^3n_0, \cdots$, so the vector length follows the exponential growth. Under λ_2, the lengths of each vector added are $n_0, 2^2n_0, 2^4n_0, 2^6n_0, \cdots$, where the vector length also follows the exponential growth but at a speed faster than that under λ_1. Under λ_3, the lengths of each vector added are $n_0, n_0, 2n_0, 2n_0, 2^2n_0, 2^2n_0, 2^3n_0, 2^3n_0, \cdots$, which expands slower than that under λ_1. Under λ_4, the expanding speed is randomly generated within [1,3]. As a result, there are three different length vectors with $n_0, 2n_0, 2^2n_0$. Similarly, under λ_5, there are five different length vectors with $n_0, 2n_0, 2^2n_0, 2^3n_0, 2^4n_0$.

B. Evaluation Results

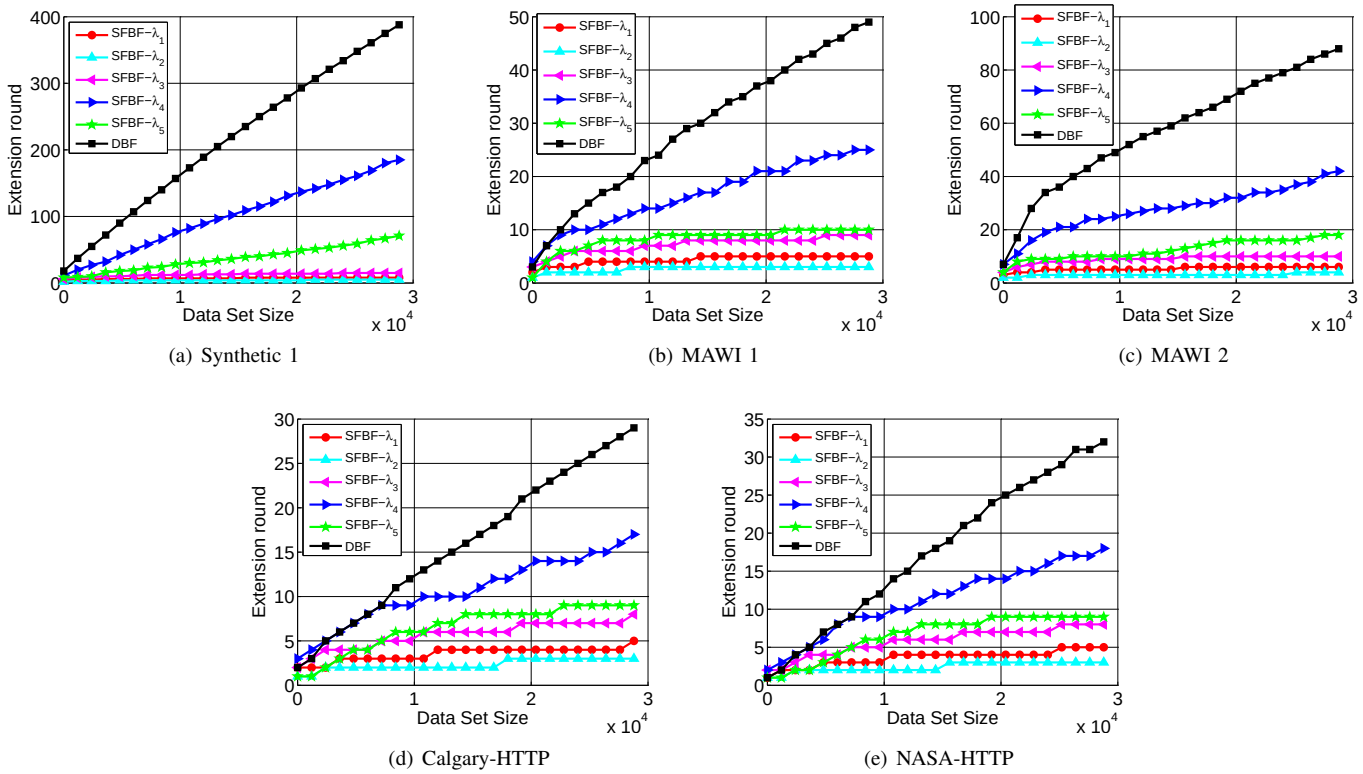
To evaluate the performance of SFBF, four performance metrics are applied to evaluate the proposed SFBF:

- Extension round: the total number of extension rounds needed to represent the dynamic increasing data set.
- False positive rate: defined as the ratio between the number of data items indicated by a SFBF that they are inserted into the SFBF even though they are not and the total number of the queried items.
- Querying CPU time: the total CPU time consumed to query an item against an SFBF.
- Size of filter: the space usage to represent the dynamic increasing data set.

1) Extension round

As discussed in the theoretical analysis, the expanding speed has an impact on the performance of an SFBF. The larger the expanding speed λ, the fewer the number of extension rounds needed for SFBF to represent a dynamic data set, as shown in Fig.8. This also indicates that the number of BF vectors in SFBF is also smaller, which allows for better performance with lower false positive rate and query CPU time, as shown in the next a few sections.

For the five implemented traces, at the same BF's parameter setting, the extension round under Synthetic trace is largest among all other traces. All data items in the Synthetic trace



(a) Synthetic 1 (b) MAWI 1 (c) MAWI 2

(d) Calgary-HTTP (e) NASA-HTTP

Fig. 8. Extension round

are randomly generated which have a high probability to be unique, but many data items in other traces are duplicated (duplicated source IP address in MAWI trace, and duplicated URL in Calgary-HTTP and NASA-HTTP traces). Therefore, even though the total number of data items to represent are the same, the total unique data items in the Synthetic trace are larger than those in other traces, which results in a larger extension round and higher false positive rate (in Fig.9).

As shown in Fig.8, different from our SFBF, the extension procedures under DBF depends on the size of the data set, and the number of extension rounds increases about linearly with the size of data set.

2) False positive rate

To illustrate the false positive rates under different Bloom Filter schemes, besides DBF and SFBF, Fig.9 also shows the curve of basic Bloom filter (denoted as BasicBF). As the data set expands to 3×10^4, the false positive rate increases under all BF schemes. With the continuous increase of the set size n, the false positive rate of BasicBF approaches 1 rapidly, while our SFBF with all various expanding speed λ can well control the false positive rate to remain at very low level.

In the small figure of Fig.9, the curve of BasicBF is removed so DBF and SFBF can be more clearly represented. According to Eq(14), the false positive ratio is the monotone increasing function of extension round. Consequently, we can observe that the larger the expanding speed λ, the smaller the false positive ratio of our SFBF as a smaller number of extension rounds is needed.

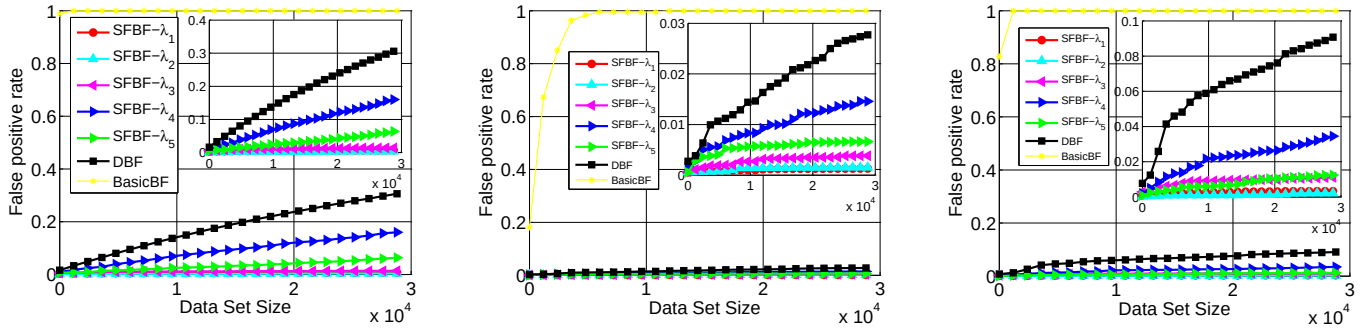
When the data set expands to $n = 3 \times 10^4$, which is 388 times that of $n_0 = 64$, the false positive rates of DBF reach 30.59% (for the Synthetic trace in Fig.9(a)), which makes it not usable. The results indicate that DBF is not really scalable.

3) Querying CPU time

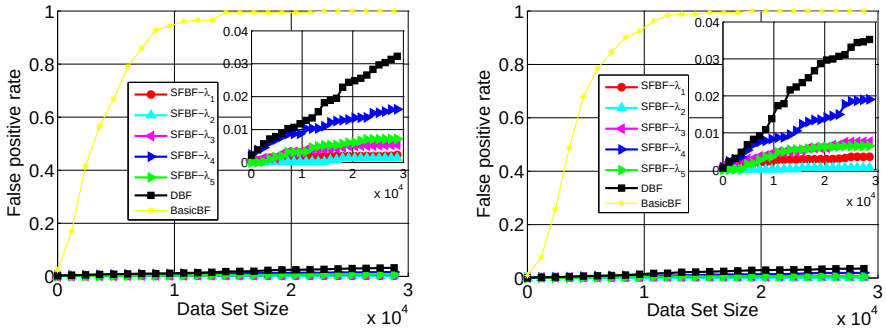
Fig.10 and Fig.11 compare the query time among different BF designs. In order to find the longest query time, besides the first 3×10^4 data items, additional 1.5×10^5 data items in the traces are queried against the bloom filters. We run 100 times to get the average results. DBF consists of BF vectors with the same length, while our SFBF consists of BF vectors with varying filter length.

For SFBF, we implement two query schemes to handle the varying filter length, a query algorithm with the shifting operation (denoted as Shift) as shown in Fig.10, and a query algorithm with the direct hash computation (denoted as Direct) as shown in Fig.11. In the first scheme, when querying an element against a BF array with multiple BF vectors, we execute one set of hash computations for the k hash functions over the largest BF vector, and then take the bit shifting operation to obtain the corresponding addresses of element to query in other BF vectors. In the second scheme, when querying a data item against a BF array, every same-length-BF-vector needs one set of hash computations for k hash functions. Because DBF consists of multiple BF vectors with the same filter length, its hash addresses for the element queried in all BF vectors are the same and only one set of hash computations with k hash functions is needed. For performance comparison, we draw the curve of DBF in both Fig.10 and Fig.11.

Comparing the Shifting scheme in Fig.10 with the Direct scheme in Fig.11, obviously, our Shift scheme proposed is very efficient and effective in largely reducing the query CPU time. Take the trace MAWI1 for example, when the data set expands to $n = 3 \times 10^4$, the average querying CPU time under SFBF-

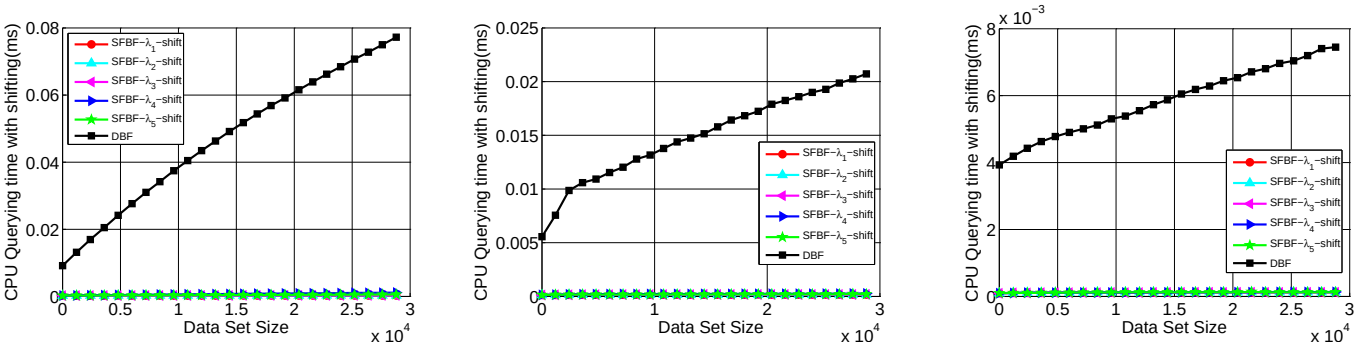


(a) Synthetic 1 (b) MAWI 1 (c) MAWI 2

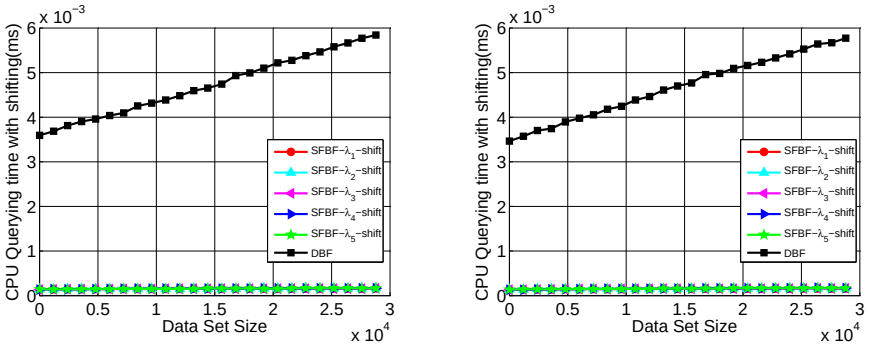


(d) Calgary-HTTP (e) NASA-HTTP

Fig. 9. False positive rate



(a) Synthetic 1 (b) MAWI 1 (c) MAWI 2

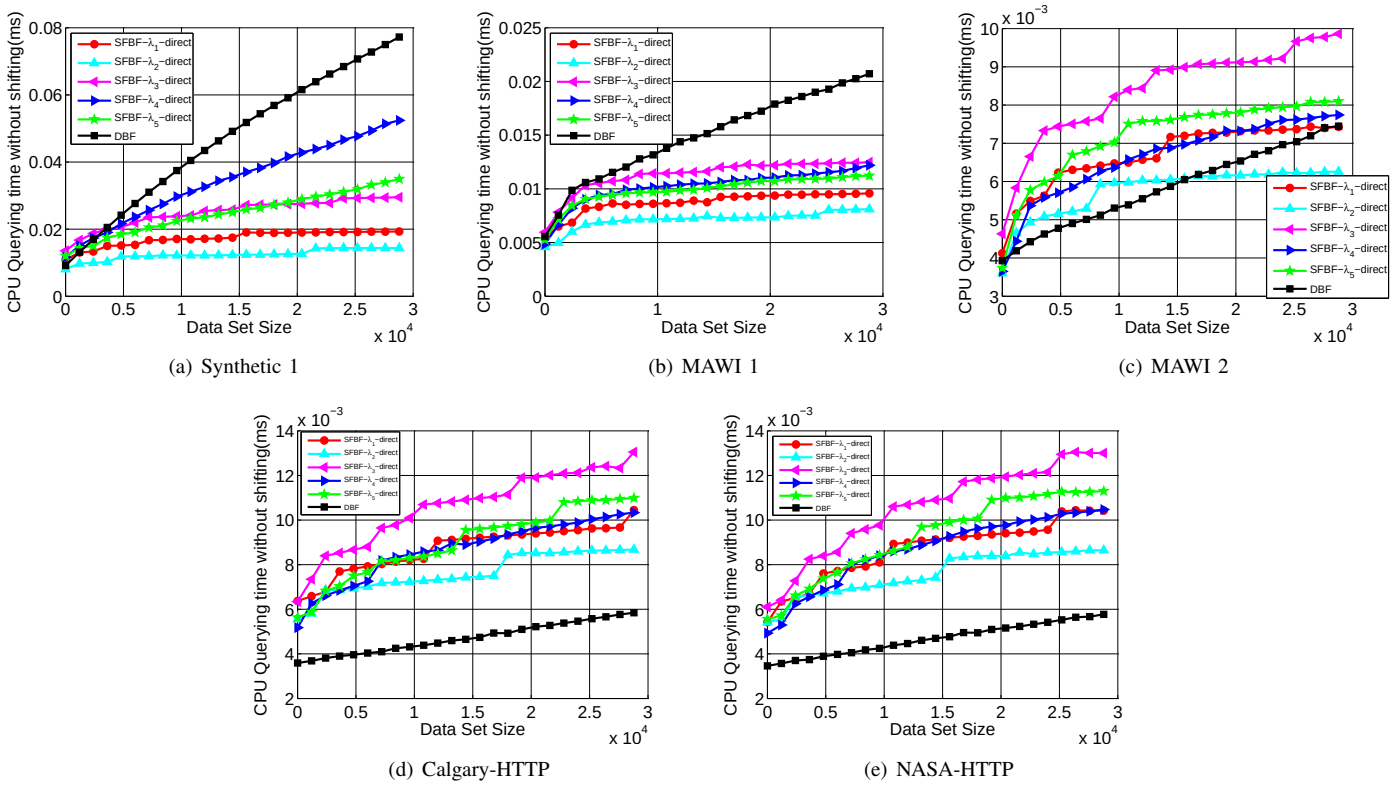


(d) Calgary-HTTP (e) NASA-HTTP

Fig. 10. Querying CPU time with shifting

λ_1-Shift, SFBF-λ_2-Shift, SFBF-λ_3-Shift, SFBF-λ_4-Shift, SFBF-λ_5-Shift, and are only 1.8%, 2.0%, 1.4%, 2.2% and 1.7% of those under SFBF-λ_1-Direct, SFBF-λ_2-Direct, SFBF-λ_3-Direct, SFBF-λ_4-Direct and SFBF-λ_5-Direct.

The query time of DBF consists of hash function computation time and comparison time while the later depends on the total number of extension rounds. The query CPU time of SFBF consists of the hash function computation time, comparison



(a) Synthetic 1 (b) MAWI 1 (c) MAWI 2

(d) Calgary-HTTP (e) NASA-HTTP

Fig. 11. Querying CPU time without shifting

time, and shifting time, among which the later two depend on the total number of extension rounds.

Compared to DBF, our SFBF can well control the extension round even when the data set is expanded to a very large size, as shown in Fig.8, therefore the increase of querying time in DBF is much faster than SFBF when the set expands, as shown in Fig.10. When the data set expands to $n = 3 \times 10^4$, DBF needs much more CPU time than those of SFBF, as shown in Fig.10. Take the trace MAWI1 as an example, the average query time of DBF is 0.027 ms, while the time of SFBF-λ_1-Shift, SFBF-λ_2-Shift, SFBF-λ_3-Shift, SFBF-λ_4-Shift, and SFBF-λ_5-Shift are $0.1733 * 10^{-3}$ ms, $0.1743 * 10^{-3}$ ms, $0.1856 * 10^{-3}$ ms, $0.2689 * 10^{-3}$ and $0.1989 * 10^{-3}$ ms, respectively. This also demonstrates that our shifting operation is very efficient and effective in largely reducing the query CPU time.

It is noted that querying a data item only takes less than $0.2 * 10^{-3}$ ms for a query in a 2.20GHz computer, which is quite acceptable.

For the different traces, the query time under the Synthetic trace are much larger than other traces. To query a data item not presented by the SFBF, all the BF vectors should be checked unless a false positive happens. Thus, a data set requiring a larger number of BF vectors would lead to a large query time. As data items in the synthetic trace have a high probability of being unique, it requires more BF vectors to represent the corresponding data set. In addition, data in other traces have more duplicated items, so their query hit ratios are larger than that in the Synthetic trace. When there is a query hit, it does not need to check all the BF vectors, which would also result in a smaller query time.

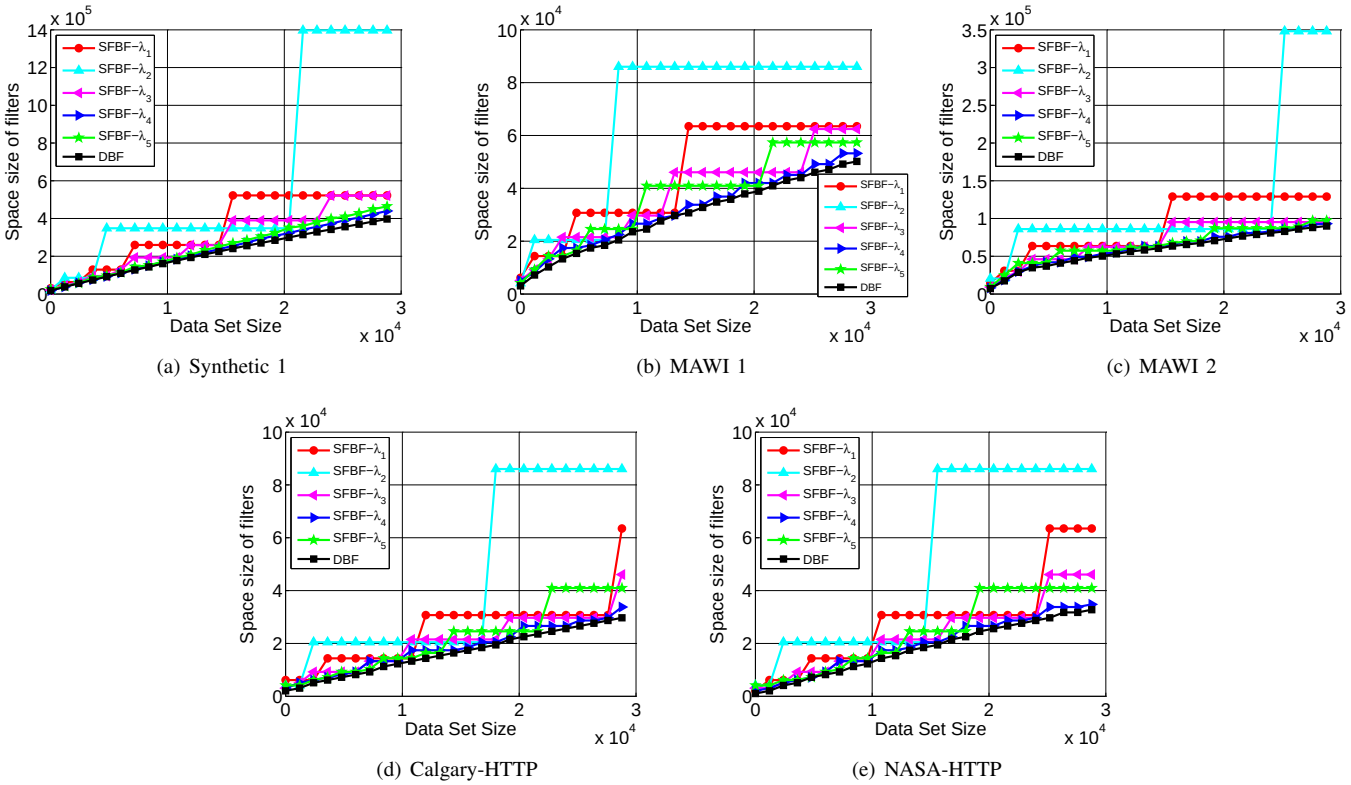
4) Size of filter

Fig.12 compares different Bloom filter designs in term of space usage. Take the trace Synthetic 1 as an example (in Fig.12(b)) , as the data set expands, the curves of DBF, SFBF-λ_1, SFBF-λ_2, SFBF-λ_3, SFBF-λ_4, SFBF-λ_5 increase in the form of ladders with different width, while the DBF ladder has the same width for each level. The width in a ladder level denotes the maximum number of data items that can be represented by the corresponding BF vector, which depends on the expanding speed in our SFBF design.

Take the trace MAWI1 as an example, when the data set expands to 30,000, even though the space usage under DBF and SFBF-λ_1, SFBF-λ_3, SFBF-λ_4 and SFBF-λ_5 is close, the false positive rates of SFBF-λ_1, SFBF-λ_3, SFBF-λ_4 and SFBF-λ_5 are only 4.67%, 13.82%, 52.58% and 23.87% of that of DBF, and the query time of SFBF-λ_1-Shift, SFBF-λ_3-Shift, SFBF-λ_4-Shift, SFBF-λ_5-Shift, and SFBF-λ_6-Shift are 2.24%, 2.21%,2.34% and 2.25% of that of DBF, as can be seen from Fig.9(b) and Fig.10(b). In summary, the performance results demonstrate that SFBF can well control the false positive rate to be at very low level and low query CPU time even when the data set expands to very large size.

X. CONCLUSION

A Bloom Filter (BF) is a space-efficient probabilistic data structure allowing membership queries over sets with certain allowable errors. It is widely used in many applications which take advantage of its ability to compactly represent a set, and filter out effectively any element that does not belong to the set. However, efficiency and scalability becomes big challenge in applying BF to a big data environment in which large scale and



(a) Synthetic 1 (b) MAWI 1 (c) MAWI 2

(d) Calgary-HTTP (e) NASA-HTTP

Fig. 12. Space size of filters

ever-increasing data arrive unpredictably. This paper proposes a Scalable and Flexible Bloom filter (SFBF) to address the scalability issue of Bloom filters. Specifically, we propose a novel algorithm to adaptively generate hash functions and a light-weight SFBF query algorithm that takes advantage of the features of our proposed hash functions to ensure light-weight member matching in a dynamic big data environment. Bloom filters and related variants have found many applications. It is expected that SFBF is applicable in many practical systems with constant changes of the number of members in a set.

REFERENCES

[1] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[2] M. M. A Broder, "Network applications of bloom filters: A survey," *Internet Mathematics*, vol. 1, no. 4, pp. 485–509, 2004.

[3] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz, "Theory and practice of bloom filters for distributed systems," *Communications Surveys & Tutorials, IEEE*, vol. 14, no. 1, pp. 131–155, 2012.

[4] S. Xiong, Y. Yao, Q. Cao, and T. He, "kbf: A bloom filter for key-value storage with an application on approximate state machines," in *IEEE INFOCOM 2014*.

[5] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: a scalable wide-area web cache sharing protocol," *IEEE/ACM Transactions on Networking (TON)*, vol. 8, no. 3, pp. 281–293, 2000.

[6] M. Mitzenmacher, "Compressed bloom filters," *IEEE/ACM Transactions on Networking (TON)*, vol. 10, no. 5, pp. 604–612, 2002.

[7] A. Kumar, J. Xu, and J. Wang, "Space-code bloom filter for efficient per-flow traffic measurement," *IEEE Journal on Selected Areas in Communications*, vol. 24, no. 12, pp. 2327–2339, 2006.

[8] F. Deng and D. Rafiei, "Approximately detecting duplicates for streaming data using stable bloom filters," in *ACM SIGMOD 2006*.

[9] S. Dutta, S. Bhattacherjee, and A. Narang, "Towards intelligent compression in streams: a biased reservoir sampling based bloom filter approach," in *EDBT 2012*.

[10] S. Dutta, A. Narang, and S. K. Bera, "Streaming quotient filter: a near optimal approximate duplicate detection approach for data streams," *Proceedings of the VLDB Endowment*, vol. 6, no. 8, pp. 589–600, 2013.

[11] M. Yoon, "Aging bloom filter with two active buffers for dynamic sets," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 22, no. 1, pp. 134–138, 2010.

[12] M. K. Yoon, J. Son, and S.-H. Shin, "Bloom tree: A search tree based on bloom filters for multiple-set membership testing," in *IEEE INFOCOM 2014*.

[13] D. Guo, J. Wu, H. Chen, X. Luo, *et al.*, "Theory and network applications of dynamic bloom filters.," in *INFOCOM 2006*.

[14] K. Xie, Y. Min, D. Zhang, J. Wen, and G. Xie, "A scalable bloom filter for membership queries," in *Global Telecommunications Conference, 2007. GLOBECOM'07. IEEE*, pp. 543–547, IEEE, 2007.

[15] J. Wei, H. Jiang, K. Zhou, and D. Feng, "Mad2: A scalable high-throughput exact deduplication approach for network backup services," in *IEEE MSST 2010*.

[16] P. S. Almeida, C. Baquero, N. Preguiça, and D. Hutchison, "Scalable bloom filters," *Information Processing Letters*, vol. 101, no. 6, pp. 255–261, 2007.

[17] S. Cohen and Y. Matias, "Spectral bloom filters," in *ACM SIGMOD 2003*.

[18] J. L. Carter and M. N. Wegman, "Universal classes of hash functions," *Journal of computer and system sciences*, vol. 18, no. 2, pp. 143–154, 1979.

[19] M. Ramakrishna, E. Fu, and E. Bahcekapili, "Efficient hardware hashing functions for high performance computers," *IEEE Transactions on Computers*, vol. 46, no. 12, pp. 1378–1381, 1997.

[20] M. Ramakrishna and G. Portice, "Perfect hashing functions for hardware applications," in *Data Engineering, 1991. Proceedings. Seventh International Conference on*, pp. 464–470, IEEE, 1991.

[21] "Mawi working group traffic archive." http://mawi.nezu.wide.ad.jp/mawi/.

[22] "Calgary-http." http://ita.ee.lbl.gov/html/contrib/Calgary-HTTP.html.

[23] "Nasa-http." http://ita.ee.lbl.gov/html/contrib/NASA-HTTP.html.