A Shifting Bloom Filter Framework for Set Queries

Tong Yang*
Peking University, China
yangtongemail@gmail.com

Alex X. Liu
Michigan State University
alexliu@cse.msu.edu

Muhammad Shahzad
North Carolina State University
mshahza@ncsu.edu

Yuankun Zhong
Nanjing University, China
kun@smail.nju.edu.cn

Qiaobin Fu
Boston University, USA
qiaobinf@bu.edu

Zi Li
Nanjing University, China
zizi13572468@gmail.com

Gaogang Xie
ICT, CAS, China
xie@ict.ac.cn

Xiaoming Li
Peking University, China
lxm@pku.edu.cn

ABSTRACT

Set queries are fundamental operations in computer systems and applications. This paper addresses the fundamental problem of designing a probabilistic data structure that can quickly process set queries using a small amount of memory. We propose a Shifting Bloom Filter (ShBF) framework for representing and querying sets. We demonstrate the effectiveness of ShBF using three types of popular set queries: membership, association, and multiplicity queries. The key novelty of ShBF is on encoding the auxiliary information of a set element in a location offset. In contrast, prior BF based set data structures allocate additional memory to store auxiliary information. We conducted experiments using real-world network traces, and results show that ShBF significantly advances the state-of-the-art on all three types of set queries.

1. INTRODUCTION

1.1 Motivations

Set queries, such as *membership queries*, *association queries*, and *multiplicity queries*, are fundamental operations in computer systems and applications. *Membership queries check whether an element is a member of a given set.* Network applications, such as IP lookup, packet classification, and regular expression matching, often involve membership queries. *Association queries identify which set(s) among a pair of sets contain a given element.* Network architectures such as distributed servers often use association queries. For example, when data is stored distributively on two servers and the popular content is replicated over both servers to achieve load balancing, for any incoming query, the gateway needs to identify the server(s) that contain the data corresponding to that query. *Multiplicity queries check how many times an element appears in a multi-set.* A multi-set

allows elements to appear more than once. Network measurement applications, such as measuring flow sizes, often use multiplicity queries.

This paper addresses the fundamental problem of designing a probabilistic data structure that can quickly process set queries, such as the above-mentioned membership, association, and multiplicity queries, using a small amount of memory. Set query processing speed is critical for many systems and applications, especially for networking applications as packets need to be processed at wire speed. Memory consumption is also critical because small memory consumption may allow the data structure to be stored in SRAM, which is an order of magnitude faster than DRAM.

Widely used set data structures are the standard Bloom Filter (BF) [3] and the counting Bloom Filter (CBF) [11]. Let $h_1(.), \cdots, h_k(.)$ be k independent hash functions with uniformly distributed outputs. Given a set S, BF constructs an array B of m bits, where each bit is initialized to 0, and for each element $e \in S$, BF sets the k bits $B[h_1(e)\%m], \cdots, B[h_k(e)\%m]$ to 1. To process a membership query of whether element e is in S, BF returns true if all corresponding k bits are 1 (*i.e.*, returns $\wedge_{i=1}^{k} B[h_i(e)\%m]$). BF has no false negatives (FNs), *i.e.*, it never says that $e \notin S$ when actually $e \in S$. However, BF has false positives (FPs), *i.e.*, it may say that $e \in S$ when actually $e \notin S$ with a certain probability. Note that BF does not support element deletion. CBF overcomes this shortcoming by replacing each bit in BF by a counter. Given a set of elements, CBF first constructs an array C of m counters, where each counter is initialized to 0. For each element e in S, for each $1 \leqslant i \leqslant k$, CBF increments $C[h_i(e)\%m]$ by 1. To process a membership query of whether element e is in set S, CBF returns true if all corresponding k counters are at least 1 (*i.e.*, returns $\wedge_{i=1}^{k}(C[h_i(e)\%m] \geqslant 1)$). To delete an element e from S, for each $1 \leqslant i \leqslant k$, CBF decrements $C[h_i(e)\%m]$ by 1.

1.2 Proposed Approach

In this paper, we propose a *Shifting Bloom Filter* (ShBF) framework for representing and querying sets. Let $h_1(.), \cdots, h_k(.)$ be k independent hash functions with uniformly distributed outputs. In the construction phase, ShBF first constructs an array B of m bits, where each bit is initialized to 0. We observe that in general a set data structure needs to store two types of information for each element e: (1) *existence in-*

formation, *i.e.*, whether e is in a set, and (2) *auxiliary information*, *i.e.*, some additional information such as e's counter (*i.e.*, multiplicity) or which set that e is in. For each element e, we encode its existence information in k hash values $h_1(e)\%m, \cdots, h_k(e)\%m$, and its auxiliary information in an offset $o(e)$. Instead of, or in addition to, setting the k bits at locations $h_1(e)\%m, \cdots, h_k(e)\%m$ to 1, we set the bits at locations $(h_1(e)+o(e))\%m, \cdots, (h_k(e)+o(e))\%m$ to 1. For different set queries, the offset has different values. In the query phase, to query an element e, we first calculate the following k locations: $h_1(e)\%m, \cdots, h_k(e)\%m$. Let c be the maximum value of all offsets. For each $1 \leqslant i \leqslant k$, we first read the c bits $B[h_i(e)\%m], B[(h_i(e)+1)\%m], \cdots, B[(h_i(e)+c-1)\%m]$ and then calculate the existence and auxiliary information about e by analyzing where 1s appear in these c bits. To minimize the number of memory accesses, we extend the number of bits in ShBF to $m+c$; thus, we need $k\lceil \frac{c}{w} \rceil$ number of memory accesses in the worst case, where w is the word size. Figure 1 illustrates our ShBF framework.
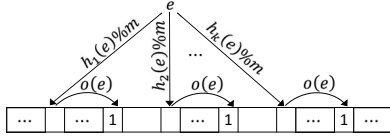

Figure 1: Shifting Bloom Filter framework.

We demonstrate the effectiveness of ShBF using three types of popular set queries: membership, association, and multiplicity queries.

1.2.1 Membership Queries

Such queries only deal with the existence information of each element, which is encoded in k random positions in array B. To leverage our ShBF framework, we treat $k/2$ positions as the existence information and the other $k/2$ positions as the auxiliary information, assuming k is an even number for simplicity. Specifically, the offset function $o(.) = h_{\frac{k}{2}+1}(.)\%(\overline{w}-1) + 1$, where $h_{\frac{k}{2}+1}(.)$ is another hash function with uniformly distributed outputs and $\overline{w}$ is a function of machine word size w. In the **construction phase**, for each element $e \in S$, we set both the $k/2$ bits $B[h_1(e)\%m], \cdots, B[h_{\frac{k}{2}}(e)\%m]$ and the $k/2$ bits $B[h_1(e)\%m + o(e)], \cdots, B[h_{\frac{k}{2}}(e)\%m + o(e)]$ to 1. In the **query phase**, for an element e, if all these k bits are 1, then we output $e \in S$; otherwise, we output $e \notin S$. In terms of false positive rate (FPR), our analysis shows that ShBF is very close to BF with k hash functions. In terms of performance, ShBF is about two times faster than BF because of two main reasons. First, ShBF reduces the computational cost by almost half because the number of hash functions that ShBF needs to compute is almost the half of what BF needs to compute. Second, ShBF reduces the number of memory accesses by half because although both ShBF and BF write k bits into the array B, when querying element e, by one memory access, ShBF obtains two bits about e whereas BF obtains only one bit about e.

1.2.2 Association Queries

For this type of queries with two sets S_1 and S_2, for elements in $S_1 \cup S_2$, there are three cases: (1) $e \in S_1 - S_2$, (2) $e \in S_1 \cap S_2$, and (3) $e \in S_2 - S_1$. For the first case, *i.e.*, $e \in S_1 - S_2$, the offset function $o(e) = 0$. For the second case, *i.e.*, $e \in S_1 \cap S_2$, the offset function $o(e) = o_1(e) = h_{k+1}(e)\%((\overline{w}-1)/2) + 1$, where $h_{k+1}(.)$

is another hash function with uniformly distributed outputs and $\overline{w}$ is a function of machine word size w. For the third case, *i.e.*, $e \in S_2 - S_1$, the offset function $o(e) = o_2(e) = o_1(e) + h_{k+2}(e)\%((\overline{w}-1)/2) + 1$, where $h_{k+2}(.)$ is yet another hash function with uniformly distributed outputs. In the **construction phase**, for each element $e \in S_1 \cup S_2$, we set the k bits $B[h_1(e)\%m + o(e)], \cdots, B[h_k(e)\%m + o(e)]$ to 1 using an appropriate value of $o(e)$ as just described for the three cases. In the **query phase**, given an element $e \in S_1 \cup S_2$, for each $1 \leqslant i \leqslant k$, we read the 3 bits $B[h_i(e)\%m]$, $B[h_i(e)\%m + o_1(e)]$, and $B[h_i(e)\%m + o_2(e)]$. If all the k bits $B[h_1(e)\%m], \cdots, B[h_k(e)\%m]$ are 1, then e may belong to $S_1 - S_2$. If all the k bits $B[h_1(e)\%m + o_1(e)], \cdots, B[h_k(e)\%m + o_1(e)]$ are 1, then e may belong to $S_1 \cap S_2$. If all the k bits $B[h_1(e)\%m + o_2(e)], \cdots, B[h_k(e)\%m + o_2(e)]$ are 1, then e may belong to $S_2 - S_1$. There are a few other possibilities that we will discuss later in Section 4.2, that ShBF takes into account when answering the association queries. In comparison, the standard BF based association query scheme, namely iBF, constructs a BF for each set. In terms of accuracy, iBF is prone to false positives whenever it declares an element $e \in S_1 \cup S_2$ in a query to be in $S_1 \cap S_2$, whereas ShBF achieves an FPR of zero. In terms of performance, ShBF is almost twice as fast as iBF because iBF needs $2k$ hash functions and $2k$ memory accesses per query, whereas ShBF needs only $k+2$ hash functions and k memory accesses per query.

1.2.3 Multiplicity Queries

For multiplicity queries, for each element e in a multiset S, the offset function $o(.) = c(e) - 1$ where $c(e)$ is e's counter (*i.e.*, the number of occurrences of e in S). In the **construction phase**, for each element e, we set the k bits $B[h_1(e)\%m+c(e)-1], \cdots, B[h_k(e)\%m+c(e)-1]$ to 1. In the **query phase**, for an element e, for each $1 \leqslant i \leqslant k$, we read the c bits $B[h_i(e)\%m], B[h_i(e)\%m+1], \cdots, B[h_i(e)\%m+c-1]$, where c is the maximum number of occurrences that an element can appear in S. For these ck bits, for each $1 \leqslant j \leqslant c$, if all the k bits $B[h_1(e)\%m+j-1], \cdots, B[h_k(e)\%m+j-1]$ are 1, then we output j as one possible value of $c(e)$. Due to false positives, we may output multiple possible values.

1.3 Novelty and Advantages over Prior Art

The key novelty of ShBF is on encoding the auxiliary information of a set element in its location by the use of offsets. In contrast, prior BF based set data structures allocate additional memory to store such auxiliary information.

To evaluate our ShBF framework in comparison with prior art, we conducted experiments using real-world network traces. Our results show that ShBF significantly advances the state-of-the-art on all three types of set queries: membership, association, and multiplicity. For membership queries, in comparison with the standard BF, ShBF has about the same FPR but is about 2 times faster; in comparison with 1MemBF [17], which represents the state-of-the-art in membership query BFs, ShBF has $10\% \sim 19\%$ lower FPR and $1.27 \sim 1.44$ times faster query speed. For association queries, in comparison with iBF, ShBF has 1.47 times higher probability of a clear answer, and has 1.4 times faster query speed. For multiplicity queries, in comparison with Spectral BF [8], which represents the state-of-the-art in multiplicity query BFs, ShBF has $1.45 \sim 1.62$ times higher correctness rate and the query speeds are comparable.

2. RELATED WORK

We now review related work on the three types of set queries: membership, association, and multiplicity queries, which are mostly based on Bloom Filters. Elaborate surveys of the work on Bloom Filters can be found in [5, 14, 18, 19].

2.1 Membership Queries

Prior work on membership queries focuses on optimizing BF in terms of the number of hash operations and the number of memory accesses. Fan *et al.* proposed the Cuckoo filter and found that it is more efficient in terms of space and time compared to BF [10]. This improvement comes at the cost of non-negligible probability of failing when inserting an element. To reduce the number of hash computation, Kirsch *et al.* proposed to use two hash functions $h_1(.)$ and $h_2(.)$ to simulate k hash functions $(h_1(.) + i * h_2(.))\%m$, where $(1 \leqslant i \leqslant k)$; but the cost is increased FPR [13]. To reduce the number of memory accesses, Qiao *et al.* proposed to confine the output of the k hash functions within certain number of machine `word`s, which reduces the number of memory accesses during membership queries; but the cost again is increased FPR [17]. In contrast, ShBF reduces the number of hash operations and memory access by about half while keeping FPR about the same as BF.

2.2 Association Queries

Prior work on association queries focuses on identifying the set, among a group of pair-wise disjoint sets, to which an element belongs. A straightforward solution is iBF, which builds one BF for each set. To query an element, iBF generates a membership query for each set's BF and finds out which set(s) the unknown element is in. This solution is used in the Summary-Cache Enhanced ICP protocol [11]. Other notable solutions include kBF [20], Bloomtree [22], Bloomier [6], Coded BF [16], Combinatorial BF [12], and SVBF [15]. When some sets have intersections, there will be consecutive 1s in the filters, and the false positive rate will increase and formulas will change. In this paper, we focus on the query of two sets with intersections.

2.3 Multiplicity Queries

BF cannot process multiplicity queries because it only tells whether an element is in a set. Spectral BF, which was proposed by Cohen and Matias, represents the state-of-the-art scheme for multiplicity queries [8]. There are three versions of Spectral BF. The first version proposes some modifications to CBF to record the multiplicities of elements. The second version increases only the counter with the minimum value when inserting an element. This version reduces FPR at the cost of not supporting updates. The third version minimizes space for counters with a secondary spectral BF and auxiliary tables, which makes querying and updating procedures time consuming and more complex. Aguilar-Saborit *et al.* proposed Dynamic Count Filters (DCF), which combines the ideas of spectral BF and CBF, for multiplicity queries [2]. DCF uses two filters: the first filter uses fixed size counters and the second filter dynamically adjusts counter sizes. The use of two filters degrades query performance. Another well-known scheme for multiplicity queries is the Count-Min (CM) Sketch [9]. A CM Sketch is actually a partitioned Counting Bloom filter. We will show that our scheme is much more-memory efficient than CM sketches.

3. MEMBERSHIP QUERIES

In this section, we first present the construction and query phases of ShBF for membership queries. Membership queries are the "traditional" use of a BF. We use ShBF$_M$ to denote the ShBF scheme for membership queries. Second, we describe the updating method of ShBF$_M$. Third, we derive the FPR formula of ShBF$_M$. Fourth, we compare the performance of ShBF$_M$ with that of BF. Last, we present a generalization of ShBF$_M$. Table 1 summarizes the symbols and abbreviations used in this paper.

Table 1: Symbols & abbreviations used in the paper

Symbol	Description
m	size of a Bloom Filter
n	# of elements of a Bloom Filter
k	# of hash functions of a Bloom Filter
k_{opt}	the optimal value of k
S	a set
e	one element of a set
u	one element of a set
$h_i(s)$	the i-th hash function
FP	false positive
FPR	false positive rate
f	the FP rate of a Bloom Filter
p'	the probability that one bit is still 0 after inserting all elements into BF
BF	standard Bloom Filter
iBF	individual BF: the solution that builds one individual BF per set
ShBF	Shifting Bloom Filters
ShBF$_M$	Shifting Bloom Filters for membership qrs.
ShBF$_A$	Shifting Bloom Filters for association qrs.
ShBF$_\times$	Shifting Bloom Filters for multiplicities qrs.
Qps	queries per second
multi-set	a generalization of the notion of a set in which members can appear more than once
$o(.)$	offset(.), referring to the offset value for a given input
w	# of bits in a machine word
$\overline{w}$	the maximum value of offset(.) for membership query of a single set
c	the maximum number of times an element can occur in a multi-set

3.1 ShBF$_M$ – Construction Phase

The construction phase of ShBF$_M$ proceeds in three steps. Let $h_1(.), h_2(.), \cdots, h_{\frac{k}{2}+1}(.)$ be $\frac{k}{2} + 1$ independent hash functions with uniformly distributed outputs. First, we construct an array B of m bits, where each bit is initialized to 0. Second, to store the existence information of an element e of set S, we calculate $\frac{k}{2}$ hash values $h_1(e)\%m, h_2(e)\%m, \cdots, h_{\frac{k}{2}}(e)\%m$. To leverage our ShBF framework, we also calculate the offset values for the element e of set S as the auxiliary information for each element, namely $o(e) = h_{\frac{k}{2}+1}(e) \% (\overline{w} - 1) + 1$. We will later discuss how to choose an appropriate value for $\overline{w}$. Third, we set the $\frac{k}{2}$ bits $B[h_1(e)\%m], \cdots, B[h_{\frac{k}{2}}(e)\%m]$ to 1 and the other $\frac{k}{2}$ bits $B[h_1(e)\%m + o(e)], \cdots, B[h_{\frac{k}{2}}(e)\%m + o(e)]$ to 1. Note that $o(e) \neq 0$ because if $o(e) = 0$, the two bits $B[h_i(e)\%m]$

and $B[h_i(e)\%m+o(e)]$ are the same bits for any value of i in the range $1 \leqslant i \leqslant \frac{k}{2}$. For the construction phase, the maximum number of hash operations is $\frac{k}{2}+1$. Figure 2 illustrates the construction phase of ShBF$_M$.
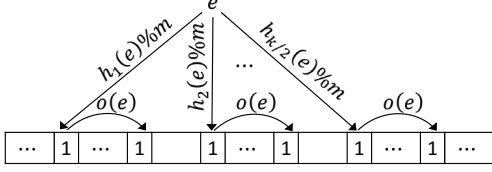

Figure 2: Illustration of ShBF$_\mathbf{M}$ construction phase.

We now discuss how to choose a proper value for $\overline{w}$ so that for any $1 \leqslant i \leqslant \frac{k}{2}$, we can access both bits $B[h_i(e)\%m]$ and $B[h_i(e)\%m+o(e)]$ in one memory access. Note that modern architecture like x86 platform CPU can access data starting at any byte, *i.e.*, can access data aligned on any boundary, not just on word boundaries. Let $B[h_i(e)\%m]$ be the j-th bits of a byte where $1 \leqslant j \leqslant 8$. To access bit $B[h_i(e)\%m]$, we always need to read the $j-1$ bits before it. To access both bits $B[h_i(e)\%m]$ and $B[h_i(e)\%m+o(e)]$ in one memory access, we need to access $j-1+\overline{w}$ bits in one memory access. Thus, $j-1+\overline{w} \leqslant w$, which means $\overline{w} \leqslant w+1-j$. When $j=8$, $w+1-j$ has the minimum value of $w-7$. Thus, we choose $\overline{w} \leqslant w-7$ as it guarantees that we can read both bits $B[h_i(e)\%m]$ and $B[h_i(e)\%m+o(e)]$ in one memory access.

3.2 ShBF$_\mathbf{M}$ – Query Phase

Given a query e, we first read the two bits $B[h_1(e)\%m]$ and $B[h_1(e)\%m+o(e)]$ in one memory access. If both bits are 1, then we continue to read the next two bits $B[h_2(e)\%m]$ and $B[h_2(e)\%m+o(e)]$ in one memory access; otherwise we output that $e \notin S$ and the query process terminates. If for all $1 \leqslant i \leqslant \frac{k}{2}$, $B[h_i(e)\%m]$ and $B[h_i(e)\%m+o(e)]$ are 1, then we output $e \in S$. For the query phase, the maximum number of memory accesses is $\frac{k}{2}$.

3.3 ShBF$_\mathbf{M}$ – Updating

Just like BF handles updates by replacing each bit by a counter, we can extend ShBF$_\mathbf{M}$ to handle updates by replacing each bit by a counter. We use CShBF$_\mathbf{M}$ to denote this counting version of ShBF$_\mathbf{M}$. Let C denote the array of m counters. To insert an element e, instead of setting k bits to 1, we increment each of the corresponding k counters by 1; that is, we increment both $C[h_i(e)\%m]$ and $C[h_i(e)\%m+o(e)]$ by 1 for all $1 \leqslant i \leqslant \frac{k}{2}$. To delete an element $e \in S$, we decrement both $C[h_i(e)\%m]$ and $C[h_i(e)\%m+o(e)]$ by 1 for all $1 \leqslant i \leqslant \frac{k}{2}$. In most applications, 4 bits for a counter are enough. Therefore, we can further reduce the number of memory accesses for updating CShBF$_\mathbf{M}$. Similar to the analysis above, if we choose $\overline{w} \leqslant \lfloor \frac{w-7}{z} \rfloor$ where z is the number of bits for each counter, we can guarantee to access both $C[h_i(e)\%m]$ and $C[h_i(e)\%m+o(e)]$ in one memory access. Consequently, one update of CShBF$_\mathbf{M}$ needs only $k/2$ memory accesses.

Due to the replacement of bits by counters, array C in CShBF$_\mathbf{M}$ uses much more memory than array B in ShBF$_\mathbf{M}$. To have the benefits of both fast query processing and small memory consumption, we can maintain both ShBF$_\mathbf{M}$ and CShBF$_\mathbf{M}$, but store array B in fast SRAM and array C in DRAM. Note that SRAM is at least an order of magnitude faster than DRAM. Array B in fast SRAM is for processing queries and array C in slow DRAM is only for updating. After each update, we synchronize array C with array B. The synchronization is quite straightforward: when we insert an element, we insert it to both array C and B; when we delete an element, we first delete it from C, if there is at least one of the k counters becomes 0, we clear the corresponding bit in B to 0.

3.4 ShBF$_\mathbf{M}$ – Analysis

We now calculate the FPR of ShBF$_\mathbf{M}$, denoted as $f_{\text{ShBF}_\mathbf{M}}$. Then, we calculate the minimum value of $\overline{w}$ so that ShBF$_\mathbf{M}$ can achieve almost the same FPR as BF. Last, we calculate the optimum value of k that minimizes $f_{\text{ShBF}_\mathbf{M}}$.

3.4.1 False Positive Rate

We calculate the false positive rate of ShBF$_\mathbf{M}$ in the following theorem.

THEOREM 1. *The FPR of ShBF$_M$ for a set of n elements is calculated as follows:*

$$f_{\text{ShBF}_M} \approx (1-p)^{\frac{k}{2}}\left(1-p+\frac{1}{\overline{w}-1}p^2\right)^{\frac{k}{2}} \qquad (1)$$

where $p = e^{\frac{-nk}{m}}$.

PROOF. Let p' represent the probability that one bit (suppose it is at position i) in the filter B is still 0 after inserting information of all n elements. For an arbitrary element e, if $h_i(e)\%m$ does not point to i or $i-o(e)$, where $o(e) = h_{\frac{k}{2}}(e)\%(\overline{w}-1)+1$, then the bit at position i will still be 0, thus p' is given by the following equation.

$$p' = \left(\frac{m-2}{m}\right)^{\frac{kn}{2}} = \left(1-\frac{2}{m}\right)^{\frac{kn}{2}} \qquad (2)$$

When m is large, we can use the identity $\sum\limits_{x}^{\infty}\left(1-\frac{1}{x}\right)^{-x} = e$, to get the following equation for p'.

$$p' = \left(1-\frac{2}{m}\right)^{\frac{kn}{2}} = \left(\left(1-\frac{2}{m}\right)^{\frac{m}{2}}\right)^{\frac{kn}{m}} \approx e^{-\frac{nk}{m}} \qquad (3)$$

Let X and Y be the random variables for the event that the bit at position $h_i(.)$ and the bit at position $h_i(.)+h_{\frac{k}{2}+1}(.)$ is 1, respectively. Thus, $P\{X\} = 1-p'$. Suppose we look at a hash pair $\langle h_i, h_{\frac{k}{2}}+1\rangle$, we want to calculate $P\{XY\}$. As $P\{XY\} = P\{X\} \times P\{Y|X\}$, next we calculate $P\{Y|X\}$. There are $\overline{w}-1$ bits on the left side of position h_i. The $1s$ in these $\overline{w}-1$ bits could be due to the first hash function in a pair and/or due to the second hash function in the pair. In other words, event X happens because a hash pair $\langle h_j, h_{\frac{k}{2}+1}\rangle$ sets the position h_i to 1 during the construction phase. When event X happens, there are two cases:

1. The event $X1$ happens, *i.e.*, the position h_i is set to 1 by $h_{\frac{k}{2}+1}$, *i.e.*, the left $\overline{w}-1$ bits cause h_i to be 1, making X and Y independent. Thus, in this case $P\{Y\} = 1-p'$.

2. The event $X2$ happens, *i.e.*, the position h_i is set to 1 by h_j. In this case, As $P\{X1\} + P\{X2\} =1$, thus, $P\{Y|X\} = P\{Y|X, X1\} \times P\{X1\} + P\{Y|X, X2\} \times P\{X2\}$.

Next, we compute $P\{X1\}$ and $P\{X2\}$.

As there are $\overline{w}-1$ bits on the left side of position h_i, there are $\overline{w}-1$ combinations, *i.e.*, $\binom{\overline{w}-1}{1} = \overline{w}-1$. Probability that any bit of the $\overline{w}-1$ bits is 1 is $1-p'$. When one bit in the $\overline{w}-1$ bits is 1, probability that this bit sets the bit at location h_i using the hash function $h_{\frac{k}{2}}+1$ to 1 is $\frac{1}{\overline{w}-1}$. Therefore, $P\{X1\} = \binom{\overline{w}-1}{1} \times (1-p') \times \frac{1}{\overline{w}-1} = 1-p'$. Consequently, $P\{X2\} = 1 - P\{X1\} = p'$. Again there are two cases:

1. If the bit which $h_i(x)$ points to is set to 1 by the left $1s$, X and Y are independent, and thus $P\{Y\} = \binom{\overline{w}-1}{1} \times (1-p') \times \frac{1}{\overline{w}-1} = 1-p'$.

2. If the bit which $h_i(x)$ points to is not set to 1 by the left $1s$, then it must set one bit of the latter $\overline{w}-1$ bits to be 1. This case will cause one bit of the latter $\overline{w}-1$ bits after position h_i to be 1. In this case, there are following two situations for the second hashing $h_i + h_{\frac{k}{2}+1}$:

   (a) when the second hash points to this bit, the probability is $\frac{1}{\overline{w}-1} \times 1$;

   (b) otherwise, the probability is $(1 - \frac{1}{\overline{w}-1}) \times (1-p')$.

When the second case above happens, $P\{Y|X, X2\}$ is given by the following equation.

$$P\{Y|X, X2\} = \frac{(1-p')(\overline{w}-2)}{\overline{w}-1} + \frac{1}{\overline{w}-1} = \left(1 - \frac{\overline{w}-2}{\overline{w}-1}p'\right) \quad (4)$$

Integrating the two cases, we can compute $P\{Y|X\}$ as follows.

$$P\{Y|X\} = (1-p')(1-p') + \left(1 - (1-p')\right)\left(1 - \frac{\overline{w}-2}{\overline{w}-1}p'\right) \quad (5)$$

The probability that all the first hashes point to bits that are 1 is $(1-p')^{\frac{k}{2}}$. The probability that the second hash points to a bit that is 1 is the $\frac{k}{2}$-th power of Equation (5). Thus, the overall FPR of ShBF_M is given by the following equation.

$$f_{\text{ShBF}_M} = (1-p')^{\frac{k}{2}} \left((1-p')(1-p') + p'\left(1 - \frac{\overline{w}-2}{\overline{w}-1}p'\right)\right)^{\frac{k}{2}}$$
$$= (1-p')^{\frac{k}{2}} \left(1 - p' + \frac{1}{\overline{w}-1}p'^2\right)^{\frac{k}{2}} \quad (6)$$

Note that when $\overline{w} \to \infty$, this formula becomes the formula of the FPR of BF.

Let we represent $e^{-\frac{nk}{m}}$ by p. Thus, according to equation 3, $p' \approx p$. Consequently, we get:

$$f_{\text{ShBF}_M} \approx (1-p)^{\frac{k}{2}} \left(1 - p + \frac{1}{\overline{w}-1}p^2\right)^{\frac{k}{2}} \quad \square$$

Note that the above calculation of FPRs is based on the original Bloom's FPR formula [3]. In 2008, Bose *et al.* pointed out that Bloom's formula [3] is slightly flawed and gave

a new FPR formula [4]. Specifically, Bose *et al.* explained that the second independence assumption needed to derive f_{Bloom} is too strong and does not hold in general, resulting in an underestimation of the FPR. In 2010, Christensen *et al.* further pointed out that Bose's formula is also slightly flawed and gave another FPR formula [7]. Although Christensen's formula is final, it cannot be used to compute the optimal value of k, which makes the FPR formula practically not much useful. Although Bloom's formula underestimates the FPR, both studies pointed out that the error of Bloom's formula is negligible. Therefore, our calculation of FPRs is still based on Bloom's formula.

3.4.2 Optimizing System Parameters

Minimum Value of $\overline{w}$: Recall that we proposed to use $\overline{w} \leqslant w - 7$. According to this inequation, $\overline{w} \leqslant 25$ for 32-bit architectures and $\overline{w} \leqslant 57$ for 64-bit architectures. Next, we investigate the minimum value of $\overline{w}$ for ShBF_M to achieve the same FPR with BF. We plot f_{ShBF_M} of ShBF_M as a function of $\overline{w}$ in Figures 3(a) and 3(b). Figure 3(a) plots f_{ShBF_M} vs. $\overline{w}$ for $n = 10000$, $m = 100000$, and $k = 4$, 8, and 12 and Figure 3(b) plots f_{ShBF_M} vs. $\overline{w}$ for $n = 10000$, $k = 10$, and $m = 100000, 110000$, and 120000. The horizontal solid lines in these two figures plot the FPR of BF. From these two figures, we observe that when $\overline{w} > 20$, the FPR of ShBF_M becomes almost equal to the FPR of BF. Therefore, to achieve similar FPR as of BF, $\overline{w}$ needs to be larger than 20. Thus, by using $\overline{w} = 25$ for 32−bit and $\overline{w} = 57$ for 64−bit architecture, ShBF_M will achieve almost the same FPR as BF.

(a) $m = 100000$, $n = 10000$ (b) $k = 10$, $n = 10000$

Figure 3: FPR vs. $\overline{w}$.
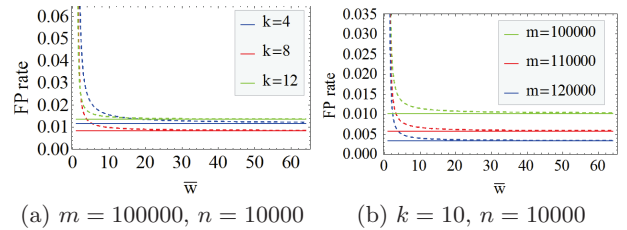
Optimum Value of k: Now we calculate the value of k that minimizes the FPR calculated in Equation (1). The standard method to obtain the optimal value of k is to differentiate Equation (1) with respect to k, equate it to 0, *i.e.*, $\frac{\partial}{\partial k} f_{\text{ShBF}_M} = 0$, and solve this equation for k. Unfortunately, this method does not yield a closed form solution for k. Thus, we use standard numerical methods to solve the equation $\frac{\partial}{\partial k} f_{\text{ShBF}_M} = 0$ to get the optimal value of k for given values of m, n, and $\overline{w}$. For $\overline{w} = 57$, differentiating Equation (1) with respect to k and solving for k results in the following optimum value of k.

$$k_{\text{opt}} = 0.7009 \frac{m}{n}$$

Substituting the value of k_{opt} from the equation above into Equation (1), the minimum value of f_{ShBF_M} is given by the following equation.

$$f_{\text{ShBF}_M}^{\min} = 0.6204^{\frac{m}{n}} \quad (7)$$

3.5 Comparison of ShBF$_\text{M}$ FPR with BF FPR

Our theoretical comparison of ShBF$_\text{M}$ and BF shows that the FPR of ShBF$_\text{M}$ is almost the same as that of BF. Figure 4 plots FPRs of ShBF$_\text{M}$ and BF using Equations (1) and (8), respectively for $m = 100000$ and $n = 4000, 6000, 8000, 10000, 12000$. The dashed lines in the figure correspond to ShBF$_\text{M}$ whereas the solid lines correspond to BF. We observe from this figure that the sacrificed FPR of ShBF$_\text{M}$ in comparison with the FPR of BF is negligible, while the number of memory accesses and hash computations of ShBF$_\text{M}$ are half in comparison with BF.
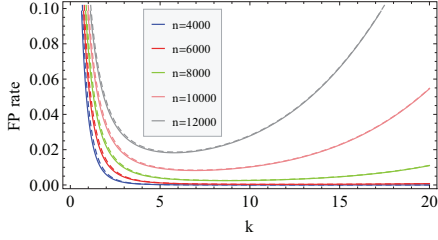

Figure 4: ShBF$_\text{M}$ FPR vs. BF FPR.

Next, we formally arrive at this result. We calculate the minimum FPR of BF as we calculated for ShBF$_\text{M}$ in Equation (7) and show that the two FPRs are practically equal. For a membership query of an element u that does not belong to set S, just like ShBF$_\text{M}$, BF can also report true with a small probability, which is the FPR of BF and has been well studied in literature [3]. It is given by the following equation.

$$f_{\text{BF}} = \left(1 - \left(1 - \frac{1}{m}\right)^{nk}\right)^k \approx \left(1 - e^{-\frac{nk}{m}}\right)^k \qquad (8)$$

For given values of m and n, the value of k that minimizes f_{BF} is $= \frac{m}{n}\ln 2 = 0.6931\frac{m}{n}$. Substituting this value of k into Equation (8), the minimum value of f_{BF} is given by the following equation.

$$f_{\text{BF}}^{\min} = \left(\frac{1}{2}\right)^{\left(\frac{m}{n}\ln 2\right)} \approx 0.6185^{\frac{m}{n}} \qquad (9)$$

By comparing Equations (7) and (9), we observe that the FPRs of ShBF$_\text{M}$ and BF are almost the same. Thus, ShBF$_\text{M}$ achieves almost the same FPR as BF while reducing the number of hash computations and memory accesses by half.

3.6 Generalization of ShBF$_\text{M}$

As mentioned earlier, ShBF$_\text{M}$ reduces k independent hash functions to $k/2 + 1$ independent hash functions. Consequently, it calculates $k/2$ locations independently and remaining $k/2$ locations are correlated through the equation $h_i(e) + o_1(e)$ $(1 \leqslant i \leqslant k/2)$. Carrying this construction strategy one step further, one could replace the first $k/2$ hash functions with $k/4$ independent hash functions and an offset $o_2(e)$, i.e., $h_j(e) + o_2(e)$ $(1 \leqslant j \leqslant k/4)$. Continuing in this manner, one could eventually arrive at $log(k) + 1$ hash functions. Unfortunately, it is not trivial to calculate the FPR for this case because $log(k)$ is seldom an integer. In this subsection, we simplify this log method into a $linear$ method by first using a group of $\frac{k}{t+1}$ $(1 \leqslant t \leqslant k-1)$ hash functions

to calculate $\frac{k}{t+1}$ hash locations and then applying shifting operation t times on these hash locations.

Consider a group of hash function comprising of $t + 1$ elements, i.e., $\langle h_1(x), h_2(x), \ldots, h_{t+1}(x)\rangle$. After completing the construction phase using this group of hash functions, the probability that any given bit is 0 is $\frac{m-w}{m} + \frac{w-1}{m}\frac{w-2}{w-1}\cdots\frac{w-t-1}{w-t} = 1 - \frac{t+1}{m}$. To insert n elements, we need $\frac{nk}{t+1}$ such group insertion operations. After completing the insertion, the probability p' that one bit is still 0 is given by the following equation.

$$p' = \left(1 - \frac{t+1}{m}\right)^{\frac{kn}{t+1}} \approx e^{-\frac{kn}{m}} \qquad (10)$$

Note that this probability formula is essentially k times product of $e^{-\frac{n}{m}}$. Thus, we can treat our ShBF$_\text{M}$ as a *partitioned Bloom filter*, where the output of each hash function covers a distinct set of consecutive $\frac{w-1}{t}$ bits. Setting $w = m$ makes this scheme partitioned Bloom filter. The equations below calculate the FPR f for this scheme.

$$f = \left(1 - p'\right)^{\frac{k}{t+1}} \times (f_{group})^{\frac{k}{t+1}} \qquad (11)$$

where

$$f_{group} = \frac{1}{t} \times \left(1 - p'\right)^2 \times \frac{(1-p')^t - \left(1 - \frac{w-1-t}{w-1} \times p'\right)^t}{(1-p') - \left(1 - \frac{w-1-t}{w-1} \times p'\right)}$$
$$+ p' \times \left(1 - \frac{w-1-t}{w-1} \times p'\right)^t \qquad (12)$$

Due to space limitations, we have moved the derivation of this equation to our extended version at arxiv.org [21].

When $t = 1$, its false positive rate can be simplified as $f = (1 - p')^{\frac{k}{2}} * \left(1 - p' + \frac{1}{w-1} \times p'^2\right)^{\frac{k}{2}}$. Similarly, when w goes to infinity, FPR simplifies to $f = (1-p')^k$, which is the formula for FPR of a *standard Bloom filter*.

4. ASSOCIATION QUERIES

In this section, we first describe the construction and query phases of ShBF for association queries, which are also called membership test. We use ShBF$_\text{A}$ to denote the ShBF scheme for association queries. Second, we describe the updating methods of ShBF$_\text{A}$. Third, we derive the FPR of ShBF$_\text{A}$. Last, we analytically compare the performance of ShBF$_\text{A}$ with that of iBF.

4.1 ShBF$_\text{A}$ – Construction Phase

The construction phase of ShBF$_\text{A}$ proceeds in three steps. Let $h_1(.), \cdots, h_k(.)$ be k independent hash functions with uniformly distributed outputs. Let S_1 and S_2 be the two given sets. First, ShBF$_\text{A}$ constructs a hash table T_1 for set S_1 and a hash table T_2 for set S_2. Second, it constructs an array B

of m bits, where each bit is initialized to 0. Third, for each element $e \in S_1$, to store its existence information, ShBF$_A$ calculates k hash functions $h_1(e)\%m, \cdots, h_k(e)\%m$ and searches e in T_2. If it does not find e in T_2, to store its auxiliary information, it sets the offset $o(e) = 0$. However, if it does find e in T_2, to store its auxiliary information, it calculates the offset $o(e)$ as $o(e) = o_1(e) = h_{k+1}(e)\%((\overline{w} - 1)/2) + 1$, where $h_{k+1}(.)$ is a hash function with uniformly distributed output and $\overline{w}$ is a function of machine word size w, which we will discuss shortly. Fourth, it sets the k bits $B[h_1(e)\%m + o(e)], \cdots, B[h_k(e)\%m + o(e)]$ to 1. Fifth, for each element $e \in S_2$, to store its existence information, ShBF$_A$ calculates the k hash functions and searches it in T_1. If it finds e in T_1, it does not do anything because its existence as its auxiliary information have already been stored in the array B. However, if it does not find e in T_1, to store its auxiliary information, it calculates the offset $o(e)$ as $o(e) = o_2(e) = o_1(e) + h_{k+2}(e)\%((\overline{w}-1)/2) + 1$, where $h_{k+2}(.)$ is also a hash function with uniformly distributed output. Last, it sets the k bits $B[h_1(e)\%m + o(e)], \cdots, B[h_k(e)\%m + o(e)]$ to 1. To ensure that ShBF$_A$ can read $B[h_i(e)\%m]$, $B[h_i(e)\%m + o_1(e)]$, and $B[h_i(e)\%m + o_2(e)]$ in a single memory access when querying, we let $\overline{w} \leqslant w - 7$. We derived this condition $\overline{w} \leqslant w - 7$ earlier at the end of Section 3.1. As the maximum value of $h_i(e)\%m + o_2(e)$ can be equal to $m + \overline{w} - 2$, we append the m-bit array B with $\overline{w} - 2$ bits.

4.2 ShBF$_A$ – Query Phase

We assume that the incoming elements always belong to $S_1 \cup S_2$ in the load balance application[1] for convenience. To query an element $e \in S_1 \cup S_2$, ShBF$_A$ finds out which sets the element e belongs to in the following three steps. First, it computes $o_1(e)$, $o_2(e)$, and the k hash functions $h_i(e)\%m$ ($1 \leqslant i \leqslant k$). Second, for each $1 \leqslant i \leqslant k$, it reads the 3 bits $B[h_i(e)\%m]$, $B[h_i(e)\%m + o_1(e)]$, and $B[h_i(e)\%m + o_2(e)]$. Third, for these $3k$ bits, if all the k bits $B[h_1(e)\%m], \cdots, B[h_k(e)\%m]$ are 1, e may belong to $S_1 - S_2$. In this case, ShBF$_A$ records (but does not yet declare) $e \overline{\in} S_1 - S_2$. Similarly, if all the k bits $B[h_1(e)\%m + o_1(e)], \cdots, B[h_k(e)\%m + o_1(e)]$ are 1, e may belong to $S_1 \cap S_2$ and ShBF$_A$ records $e \overline{\in} S_1 \cap S_2$. Finally, if all the k bits $B[h_1(e)\%m + o_2(e)], \cdots, B[h_k(e)\%m + o_2(e)]$ are 1, e may belong to $S_2 - S_1$ and ShBF$_A$ records $e \overline{\in} S_2 - S_1$.

Based on what ShBF$_A$ recorded after analyzing the $3k$ bits, there are following 7 outcomes. If ShBF$_A$ records that:

1. only $e \overline{\in} S_1 - S_2$, it declares that e belongs to $S_1 - S_2$.

2. only $e \overline{\in} S_1 \cap S_2$, it declares that e belongs to $S_1 \cap S_2$.

3. only $e \overline{\in} S_2 - S_1$, it declares that e belongs to $S_2 - S_1$.

4. both $e \overline{\in} S_1 - S_2$ and $e \overline{\in} S_1 \cap S_2$, it declares that e belongs to S_1 but is unsure whether or not it belongs to S_2.

5. both $e \overline{\in} S_2 - S_1$ and $e \overline{\in} S_1 \cap S_2$, it declares that e belongs to S_2 but is unsure whether or not it belongs to S_1.

6. both $e \overline{\in} S_1 - S_2$ and $e \overline{\in} S_2 - S_1$, it declares that e belongs to $S_1 - S_2 \cup S_2 - S_1$.

7. all $e \overline{\in} S_1 - S_2$, $e \overline{\in} S_1 \cap S_2$, and $e \overline{\in} S_2 - S_1$, it declares that e belongs $S_1 \cup S_2$.

Note that for all these seven outcomes, the decisions of ShBF$_A$ do not suffer from false positives or false negatives. However, decisions 4 through 6 provide slightly incomplete information and the decision 7 does not provide any information because it is already given that e belongs to $S_1 \cup S_2$. We will shortly show that the probability that decision of ShBF$_A$ is one of the decisions 4 through 7 is very small, which means that with very high probability, it gives a decision with *clear* meaning, and we call it a *clear answer*.

4.3 ShBF$_A$ – Updating

Just like BF handles updates by replacing each bit by a counter, we can also extend ShBF$_A$ to handle updates by replacing each bit by a counter. We use CShBF$_A$ to denote this counting version of ShBF$_A$. Let C denote the array of m counters. To insert an element e, after querying T_1 and T_2 and determining whether $o(e) = 0$, $o_1(e)$, or $o_2(e)$, instead of setting k bits to 1, we increment each of the corresponding k counters by 1; that is, we increment the k counters $C[h_1(e)\%m + o(e)], \cdots, C[h_k(e)\%m + o(e)]$ by 1. To delete an element e, after querying T_1 and T_2 and determining whether $o(e) = 0$, $o_1(e)$, or $o_2(e)$, we decrement $C[h_i(e)\%m + o(e)]$ by 1 for all $1 \leqslant i \leqslant k$. To have the benefits of both fast query processing and small memory consumption, we maintain both ShBF$_A$ and CShBF$_A$, but store array B in fast SRAM and array C in slow DRAM. After each update, we synchronize array C with array B.

4.4 ShBF$_A$ – Analysis

Recall from Section 4.2 that ShBF$_A$ may report seven different outcomes. Next, we calculate the probability of each outcome. Let P_i denote the probability of the ith outcome. Before proceeding, we show that $h_i(.) + o(.)$ and $h_j(.) + o(.)$, when $i \neq j$, are independent of each other. For this we show that given two random variables X and Y and a number $z \in R^+$, where R^+ is the set of positive real numbers, if X and Y are independent, then X+z and Y+z are independent. As X and Y are independent, for any $x \in R$ and $y \in R$, we have

$$P(X \leqslant x, Y \leqslant y) = P(X \leqslant x) * P(Y \leqslant y) \qquad (13)$$

Adding z to both sides of all inequality signs in $P(X \leqslant x, Y \leqslant y)$, we get

$$
\begin{aligned}
P(X + z &\leqslant x + z, Y + z \leqslant y + z) \\
&= P(X \leqslant x, Y \leqslant y) \\
&= P(X \leqslant x) * P(Y \leqslant y) \\
&= P(X + z \leqslant x + z) * P(Y = z \leqslant y + z)
\end{aligned}
\qquad (14)
$$

Therefore, $X + z$ and $Y + z$ are independent.

Let n' be the number of distinct elements in $S_1 \cup S_2$, and let k be the number of hash functions. After inserting all n' elements into ShBF$_A$, the probability p' that any given bit is still 0 is given by the following equation.

$$p' = \left(1 - \frac{1}{m}\right)^{kn'} \qquad (15)$$

This is similar to one minus the false positive probability of a standard BF. When $k = \ln 2 \frac{m}{n'}$, $p' \approx 0.5$.

---

[1]The application is mentioned in the first paragraph of Introduction Section.

414

Note that the probabilities for outcomes 1, 2, and 3 are the same. Similarly, the probabilities for outcomes 4, 5, and 6 are also the same. Following equations state the expressions for these probabilities.

$$P_1 = P_2 = P_3 = (1 - 0.5^k)^2$$
$$P_4 = P_5 = P_6 = 0.5^k * (1 - 0.5^k) \qquad (16)$$
$$P_7 = (0.5^k)^2$$

When the incoming element e actually belongs to one of the three sets: $S_1 - S_2$, $S_1 \cap S_2$, and $S_2 - S_1$, there is one combination each for $S_1 - S_2$ and $S_2 - S_1$ and two combinations for $S_1 \cap S_2$. Consequently, the total probability is $P_1 + P_4 * 2 + P_7$, which equals 1. This validates our derivation of the expressions in Equation 16. As an example, let $k = \frac{m}{n'} \ln 2 = 10$. Thus, $P_1 = P_2 = P_3 = (1 - 0.5^{10})^2 \approx 0.998$, $P_4 = P_5 = P_6 = 0.5^{10} * (1 - 0.5^{10}) = 9.756 * 10^{-4}$, and $P_7 = (1 - 0.5^{10})^2 \approx 9.54 * 10^{-7}$. This example shows that with probability of 0.998, ShBF$_A$ gives a *clear* answer, and with probability of only $9.756 * 10^{-4}$, ShBF$_A$ gives an answer with incomplete information. The probability with which it gives an answer with no information is just $9.54 * 10^{-7}$, which is negligibly small.

4.5 Comparison between ShBF$_A$ with iBF

For association queries, a straightfoward solution is to build one individual BF (iBF) for each set. Let n_1, n_2, and n_3 be the number of elements in S_1, S_2, and $S_1 \cap S_2$, respectively. For iBF, let m_1 and m_2 be the size of the Bloom filter for S_1 and S_2, respectively. Table 2 presents a comparison between ShBF$_A$ and iBF. We observe from the table that ShBF$_A$ needs less memory, less hash computations, and less memory accesses, and has no false positives. For the iBF, as we use the traffic trace that hits the two sets with the same probability, iBF is optimal when the two BFs use identical values for the optimal system parameters and have the same number of hash functions. Specifically, for iBF, when $m_1 + m_2 = (n_1 + n_2)k/\ln 2$, the probability of answering a *clear answer* is $\frac{2}{3}(1 - 0.5^k)$. For ShBF$_A$, when $m = (n_1 + n_2 - n_3)k/\ln 2$, the probability of answering a clear answer is $(1 - 0.5^k)^2$.

5. MULTIPLICITY QUERIES

In this section, we first present the construction and query phases of ShBF for multiplicity queries. Multiplicity queries check how many times an element appears in a multi-set. We use ShBF$_\times$ to denote the ShBF scheme for multiplicity queries. Second, we describe the updating methods of ShBF$_\times$. Last, we derive the FPR and correctness rate of ShBF$_\times$.

5.1 ShBF$_\times$ – Construction Phase

The construction phase of ShBF$_\times$ proceeds in three steps. Let $h_1(.), \cdots, h_k(.)$ be k independent hash functions with uniformly distributed outputs. First, we construct an array B of m bits, where each bit is initialized to 0. Second, to store the existence information of an element e of multiset S, we calculate k hash values $h_1(e)\%m, \cdots, h_k(e)\%m$. To calculate the auxiliary information of e, which in this case is the count $c(e)$ of element e in S, we calculate offset $o(e)$ as $o(e) = c(e) - 1$. Third, we set the k bits

$B[h_1(e)\%m + o(e)], \cdots, B[h_k(e)\%m + o(e)]$ to 1. To determine the value of $c(e)$ for any element $e \in S$, we store the count of each element in a hash table and use the simplest collision handling method called collision chain.

5.2 ShBF$_\times$ – Query Phase

Given a query e, for each $1 \leqslant i \leqslant k$, we first read c consecutive bits $B[h_i(e)\%m]$, $B[h_i(e)\%m + 1]$, $\cdots$, $B[h_i(e)\%m + c - 1]$ in $\lceil \frac{c}{w} \rceil$ memory accesses, where c is the maximum value of $c(e)$ for any $e \in S$. In these k arrays of c consecutive bits, for each $1 \leqslant j \leqslant c$, if all the k bits $B[h_1(e)\%m + j - 1], \cdots, B[h_k(e)\%m + j - 1]$ are 1, we list j as a possible candidate of $c(e)$. As the largest candidate of $c(e)$ is always greater than or equal to the actual value of $c(e)$, we report the largest candidate as the multiplicity of e to avoid false negatives. For the query phase, the number of memory accesses is $k \lceil \frac{c}{w} \rceil$.

5.3 ShBF$_\times$ – Updating

5.3.1 ShBF$_\times$ – Updating with False Negatives

To handle element insertion and deletion, ShBF$_\times$ maintains its counting version denoted by CShBF$_\times$, which is an array C that consists of m counters, in addition to an array B of m bits. During the construction phase, ShBF$_\times$ increments the counter $C[h_i(e)\%m + o(e)]$ $(1 \leqslant i \leqslant k)$ by one every time it sets $B[h_i(e)\%m + o(e)]$ to 1. During the update, we need to guarantee that one element with multiple multiplicities is always inserted into the filter one time. Specifically, for every new element e to insert into the multiset S, ShBF$_\times$ first obtains its multiplicity z from B as explained in Section 5.2. Second, it deletes the z−th multiplicity ($o(e) = z - 1$) and inserts the $(z + 1)$−th multiplicity ($o(e) = z$). For this, it calculates the k hash functions $h_i(e)\%m$ and decrements the k counters $C[h_i(e)\%m + z - 1]$ by 1 when the counters are $\geqslant 1$. Third, if any of the decremented counters becomes 0, it sets the corresponding bit in B to 0. Note that maintaining the array C of counters allows us to reset the right bits in B to 0. Fourth, it increments the k counters $C[h_i(e)\%m + z]$ by 1 and sets the bits $B[h_i(e)\%m + z]$ to 1.

For deleting element e, ShBF$_\times$ first obtains its multiplicity z from B as explained in Section 5.2. Second, it calculates the k hash functions and decrements the counters $C[h_i(e)\%m + z - 1]$ by 1. Third, if any of the decremented counters becomes 0, it sets the corresponding bit in B as 0. Fourth, it increments the counters $C[h_i(e)\%m + z - 2]$ by 1 and sets the bits $B[h_i(e)\%m + z - 2]$ to 1.

Note that ShBF$_\times$ may introduce false negatives because before updating the multiplicity of an element, we first query its current multiplicity from B. If the answer to that query is a false positive, *i.e.*, the actual multiplicity of the element is less than the answer, ShBF$_\times$ will perform the second step and decrement some counters, which may cause a counter to decrement to 0. Thus, in the third step, it will set the corresponding bit in B to 0, which will cause false negatives.

5.3.2 ShBF$_\times$ – Updating without False Negatives

To *eliminate false negatives*, in addition to arrays B and C, ShBF$_\times$ maintains a hash table to store counts of each element. In the hash table, each entry has two fields: element and its counts/multiplicities. When inserting or deleting element e, ShBF$_\times$ follows four steps shown in Figure 5. First,

Table 2: Comparison Between $ShBF_A$ and iBF.

	Optimal Memory	#hash computations	#memory accesses	Probability of a clear answer	false positives
iBF	m1+m2=(n1+n2)k/ln2	2k	2k	$\frac{2}{3}(1-0.5^k)$	YES
$ShBF_A$	m=(n1+n2-n3)k/ln2	k+2	k	$(1-0.5^k)^2$	NO

we obtain e's counts/multiplicities from the hash table instead of $ShBF_\times$. Second, we delete e's z-th multiplicity from $CShBF_\times$. Third, if a counter in $CShBF_\times$ decreases to 0, we set the corresponding bit in $ShBF_\times$ to 0. Fourth, when inserting/deleting e, we insert the $(z-1)-$th/$(z+1)-$th multiplicity into $ShBF_\times$.
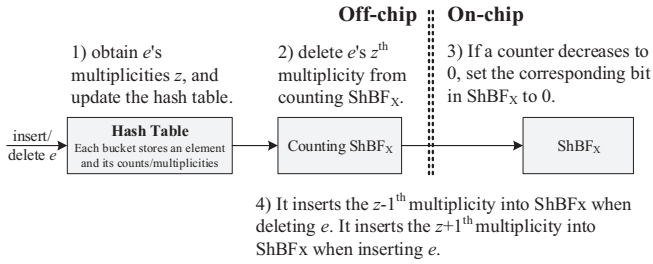
Figure 5: The update process of $ShBF_\times$.

Note that although the counter array C and the hash table are much larger than the bit array B, we store B in SRAM for processing multiplicity queries and store C and the hash table in DRAM for handling updates.

5.4 $ShBF_\times$ – Analysis

For multiplicity queries a false positive is defined as reporting the multiplicity of an element that is larger than its actual multiplicity. For any element e belonging to multiset S_m, $ShBF_\times$ only sets k bits in B to 1 regardless of how many times it appears in S_m. This is because every time information about e is updated, $ShBF_\times$ removes the existing multiplicity information of the element before adding the new information. Let the total number of distinct elements in set S_m be n. The probability that an element is reported to be present j times is given by the following equation.

$$f_0 \approx \left(1 - e^{-\frac{kn}{m}}\right)^k \qquad (17)$$

We define a metric called *correctness rate*, which is the probability that an element that is present j times in a multi-set is correctly reported to be present j times. When querying an element not belonging to the set, the correctness rate CR is given by the following equation.

$$CR = (1 - f_0)^c \qquad (18)$$

When querying an element with multiplicity j $(1 \leqslant j \leqslant c)$ in the set, the correctness rate CR' is given by the following equation.

$$CR' = (1 - f_0)^{j-1} \qquad (19)$$

Note the right hand side of the expression for CR' is not multiplied with f_0 because when e has j multiplicities, all positions $h_i(e) + j$, where $1 \leqslant i \leqslant k$, must be 1.

6. PERFORMANCE EVALUATION

In this section, we conduct experiments to evaluate our ShBF schemes and side-by-side comparison with state-of-the-art solutions for the three types of set queries.

6.1 Experimental Setup

We give a brief overview of the data we have used for evaluation and describe our experimental setup.

Data set: We evaluate the performance of ShBF and state-of-the-art solutions using real-world network traces. Specifically, we deployed our traffic capturing system on a 10Gbps link of a backbone router. To reduce the processing load, our traffic capturing system consists of two parallel sub-systems each of which is equipped with a 10G network card and uses *netmap* to capture packets. Due to high link speed, capturing entire traffic was infeasible because our device could not access/write to memory at such high speed. Thus, we only captured 5-tuple flow ID of each packet, which consists of source IP, source port, destination IP, destination port, and protocol type. We stored each 5-tuple flow ID as a 13-byte string, which is used as an element of a set during evaluation. We collected a total of 10 million 5-tuple flow IDs, out of which 8 million flow IDs are distinct. To further evaluate the accuracy of our proposed schemes, we also generated and used synthetic data sets.

Hash functions: We collected several hash functions from open source web site [1] and tested them for randomness. Our criteria for testing randomness is that the probability of seeing 1 at any bit location in the hashed value should be 0.5. To test the randomness of each hash function, we first used that hash function to compute the hash value of the 8 million unique elements in our data set. Then, for each bit location, we calculated the fraction of times 1 appeared in the hash values to empirically calculate the probability of seeing 1 at that bit location. Out of all hash functions, 18 hash functions passed our randomness test, which we used for evaluation of ShBF and state-of-the-art solutions.

Implementation: We implemented our query processing schemes in C++ using Visual C++ 2012 platform. To compute average query processing speeds, we repeat our experiments 1000 times and take the average. Furthermore, we conducted all our experiments for 20 different sets of parameters. As the results across different parameter sets follow same trends, we will report results for one parameter set only for each of the three types of queries.

Computing platform: We did all our experiments on a standard off the shelf desktop computer equipped with an Intel(R) Core i7-3520 CPU @2.90GHz and 8GB RAM running Windows 7.

6.2 $ShBF_M$ – Evaluation

In this section, we first validate the false positive rate of $ShBF_M$ calculated in Equation (1) using our experimental results. Then we compare $ShBF_M$ with BF and 1Mem-BF [13], which represents the prior scheme for answering
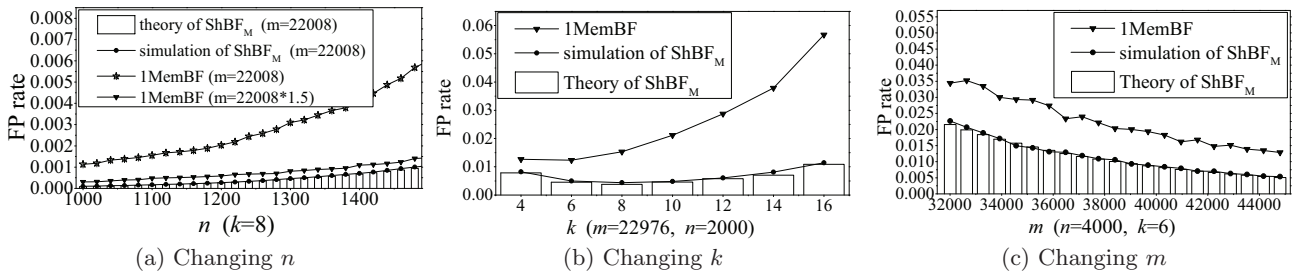
(a) Changing n

(b) Changing k

(c) Changing m

Figure 6: Comparison false positive rates of $ShBF_M$ and $1MemBF$.

(a) Changing n

(b) Changing k

(c) Changing m
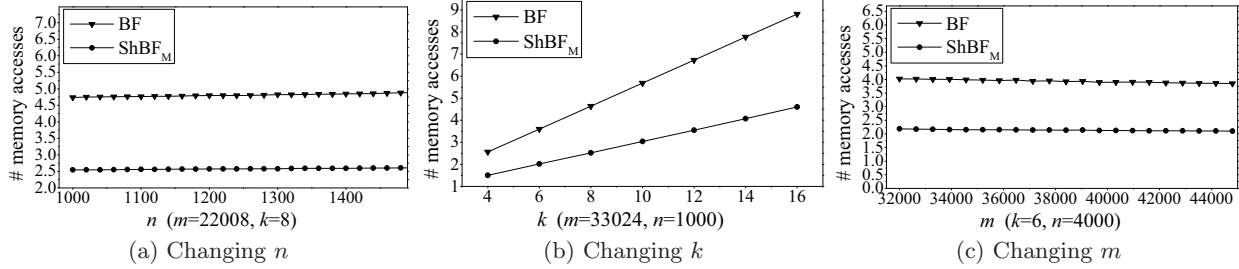
Figure 7: Comparison of number of memory accesses per query of $ShBF_M$ and BF.

(a) Changing n

(b) Changing k

(c) Changing m
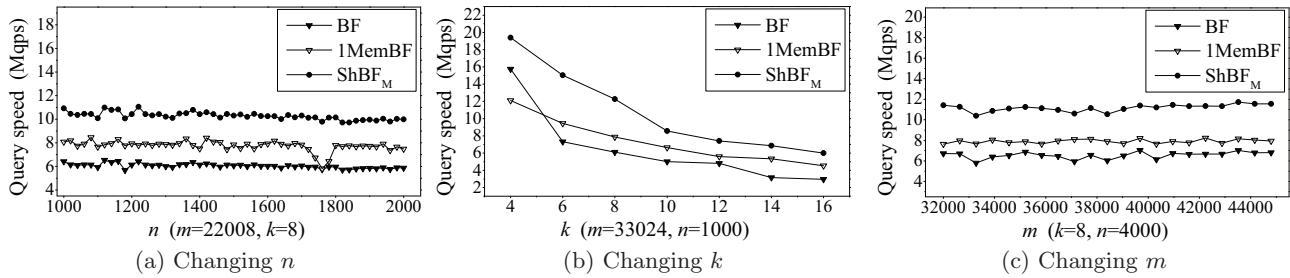
Figure 8: Comparison of query processing speeds of $ShBF_M$, BF, and $1MemBF$.

(a) Prob. a clear answer

(b) # memory accesses per query

(c) Query processing speed
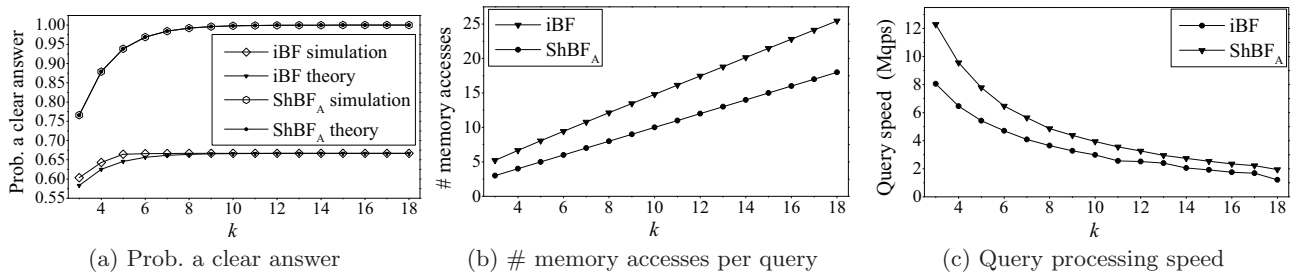
Figure 9: Comparison of $ShBF_A$ and iBF.

(a) Correctness rate (CR)

(b) Memory accesses

(c) Query processing speed
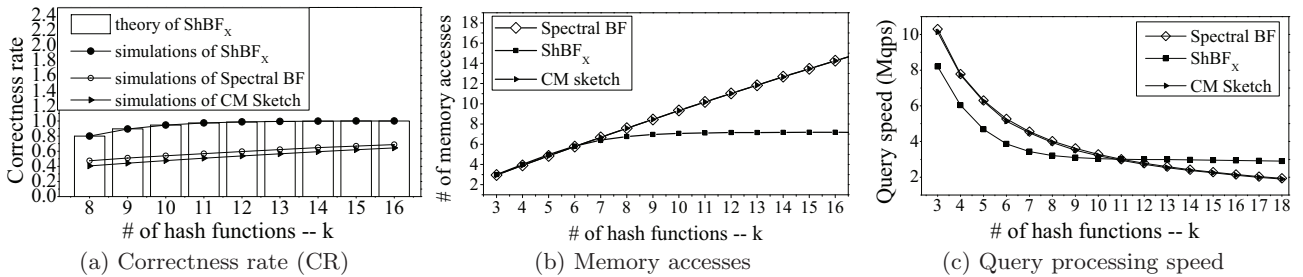
Figure 10: Comparison of $ShBF_X$, Spectral BF, and CM Sketch.

membership queries, in terms of FPR, the number of memory accesses, and query processing speed.

6.2.1 ShBF$_M$ – False Positive Rate

Our experimental results show that the FPR of ShBF$_M$ calculated in Equation (1) *matches with the FPR calculated experimentally.* For the experiments reported in this section, we set $k = 8$, $m = 22008$, $\overline{w} = 57$, and vary n from 1000 to 1500. We first insert 1000 elements into ShBF$_M$ and then repeatedly insert 20 elements until the total number of elements inserted into ShBF$_M$ became 1500. On inserting each set of 20 elements, we generated membership queries for $7,000,000$ elements whose information was not inserted into ShBF$_M$ and calculated the false positive rate. Figure 6(a) shows the false positive rate of ShBF$_M$ calculated through these simulations as well as through Equation (1). The bars in Figure 6(a) represent the theoretically calculated FPR, whereas the lines represent the FPR observed in our experiments.

Our results show that the relative error between the FPRs of ShBF$_M$ calculated using simulation and theory is less than 3%, which is practically acceptable. Relative error is defined as $|FPR_s - FPR_t|/FPR_t$, where FPR_s is the false positive rate calculated using simulation and FPR_t is the false positive rates calculated using theory. The relative error of 3% for ShBF$_M$ is the same as relative error for BF calculated using simulation and the theory developed by Bloom *et al.* [3]. Using same parameters, the FPR of 1MemBF is over $5 \sim 10$ times that of ShBF$_M$. If we increase the space allocated to 1MemBF for storage to 1.5 times of the space used by ShBF$_M$, the FPR of 1MemBF is still a little more than that of ShBF$_M$ because hashing k values into one or more words incurs serious unbalance in distributions of 1s and 0s in the memory, which in turn results in higher FPR.

Our results also show that the FPR of ShBF$_M$ is much smaller than that of 1MemBF when changing k and m. Figure 6(b) and Figure 6(c) show the FPRs of ShBF$_M$ and 1MemBF for different values of k and m, respectively.

6.2.2 ShBF$_M$ – Memory Accesses

Our results show that ShBF$_M$ answers a membership query using only about half the memory accesses and hash computations and twice as fast compared to BF. Our experiments for evaluating the number of memory accesses per query are similar to that for false positive rate, except that, now we query $2 * n$ elements, in which n elements belong to the set. Figures 7(a), 7(b), and 7(c) show the number of memory accesses for ShBF$_M$ and standard BF for different values of n, k, and m, respectively. We also observed from our experiments that standard deviation in the results for ShBF$_M$ is also about half of that of standard BF.

6.2.3 ShBF$_M$ – Query Processing Speed

Our results show that ShBF$_M$ has 1.8 and 1.4 times faster query processing speed compared to BF and 1MemBF, respectively. Although 1MemBF only needs one memory access per query, it needs $k + 1$ hash functions. BFs are usually small enough to be stored in on-chip memory (such as caches, FPGA block RAM), thus the speed of hash computation will be slower than memory accesses. In contrast, our ShBF$_M$ reduces both hash computation and memory accesses. In our experiments, using those hashes which passed our randomness test, ShBF$_M$ exhibits faster query processing

speed than that of 1MemBF. It is possible that 1MemBF is faster than ShBF$_M$ when using simple hash functions, but this probably incurs larger FPR. Our experiments for evaluating the query processing speed are similar to that for memory accesses, except that, here we also compare with 1MemBF. Figures 8(a), 8(b), and 8(c) show the query processing speed for ShBF$_M$, standard BF, and 1MemBF for different values of n, k, and m, respectively.

6.3 ShBF$_A$ – Evaluation

In this section, we first validate the probability of a clear answer of ShBF$_M$ calculated in Table 2 using our experimental results. Then we compare ShBF$_A$ with iBF in terms of FPR, memory accesses, and query processing speed.

6.3.1 ShBF$_A$ – Probability of Clear Answer

Our results show that probability of clear answer for ShBF$_A$ calculated in Table 2 matches with the probability calculated experimentally. We performed experiments for both iBF and ShBFA using two sets with 1 million elements such that their intersection had 0.25 million elements. The querying elements hit the three parts with the same probability. While varying the value of k, we also varied the value of m to keep the filter at its optimal. Note that in this case, iBF uses 1/7 times more memory than ShBFA. We observe from Figure 9(a) that the simulation results match the theoretical results, and the average relative error is 0.7% and 0.004% for iBF and ShBFA, respectively, which is negligible. When the value of k reaches 8, the probability of a clear answer reaches 66% and 99% for iBF and ShBFA, respectively.

6.3.2 ShBF$_A$ – Memory Accesses

Our results show that the average number of memory accesses per query of ShBF$_A$ is 0.66 times of that of iBF. Figure 9(b) shows the number of memory accesses for different values of k. We observed similar trends for different values of m and n, but have not including the corresponding figures due to space limitation.

6.3.3 ShBF$_A$ – Query Processing Speed

Our results show that the average query processing speed of ShBF$_A$ is 1.4 times faster than that of iBF. Figure 9(c) plots the the query processing speed of ShBF$_A$ and iBF for different values of m.

6.4 ShBF$_\times$ – Evaluation

In this section, we first validate the correctness rate (CR) of ShBF$_\times$ calculated in Equation (18). Then we compare ShBF$_\times$ with spectral BF [8] and CM Sketches [9] in terms of CR, number of memory accesses, and query processing speed. The results for CM Sketches and Spectral BF are similar because their methods of recording the counts is similar.

6.4.1 ShBF$_\times$ – Correctness Rate

Our results show that the CR of ShBF$_\times$ calculated in Equation (18) *matches with the CR calculated experimentally. Our results also show that on average, the CR of ShBF$_\times$ is 1.6 times and 1.79 times of that of Spectral BF and CM Sketches, respectively.* For the experiments reported in this section, we set $c = 57$, $n = 100,000$, and vary k in the range $8 \leqslant k \leqslant 16$. For spectral BF and CM sketches, we set use 6 bits for each counter. For each value of k, as ShBF$_\times$ is more memory efficient, we use 1.5 times the optimal memory (*i.e.*,

$1.5 * nk/ln2)$ for all the three filters. Figure 10(a) shows the results from our experiments for CR. Experimental results show that the CR calculated through experiments matches with the CR calculated theoretically.

6.4.2   *ShBF$_\times$ – Memory Accesses*

Our results show that the number of memory accesses of ShBF$_\times$ is smaller than that of spectral BF and CM Sketches for $k \geqslant 7$, and almost equal for $k < 7$. Figure 10(b) plots the number of memory accesses of ShBF$_\times$, CM Sketch, and spectral BF, calculated from the same experiments that we used to plot Figure 10(a) except that k ranges from 3 to 18.

6.4.3   *ShBF$_\times$ – Query Processing Speed*

Our results show that ShBF$_\times$ is faster than spectral BF and CM Sketches when $k \geqslant 11$. We evaluate the query processing speed of ShBF$_\times$, CM Sketch, and spectral BF using the same parameters as for Figure 10(b). Figure 10(c) plots the query processing speeds of ShBF$_\times$ and spectral BF. We observe from this figure that when $k > 11$, the average query processing speed of ShBF$_\times$ is over 3 Mqps.

7.   CONCLUSION

The key contribution of this paper is in proposing Shifting Bloom Filter, a general framework to answer a variety of set queries. We present how to use ShBF to answer three important set queries, *i.e.*, membership, association, and multiplicity queries. The key technical depth of this paper is in the analytical modeling of ShBF for each of the three types queries, calculating optimal system parameters, and finding the minimum FPRs. We validated our analytical models through simulations using real world network traces. Our theoretical analysis and experimental results show that ShBF significantly advances state-of-the-art solutions on all three types of set queries.

8.   ACKNOWLEDGEMENTS

9.   REFERENCES

[1] Hash website [on line]. Available: http://burtleburtle.net/bob/hash/evahash.html.

[2] J. Aguilar-Saborit, P. Trancoso, V. Muntes-Mulero, and J.-L. Larriba-Pey. Dynamic count filters. *ACM SIGMOD Record*, 35(1):26–32, 2006.

[3] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[4] P. Bose, H. Guo, E. Kranakis, A. Maheshwari, P. Morin, J. Morrison, M. Smid, and Y. Tang. On the false-positive rate of bloom filters. *Information Processing Letters*, 108(4):210–213, 2008.

[5] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. *Internet mathematics*, 1(4):485–509, 2004.

[6] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal. The bloomier filter: an efficient data structure for static support lookup tables. In *Proc. ACM-SIAM*, 2004.

[7] K. Christensen, A. Roginsky, and M. Jimeno. A new analysis of the false positive rate of a bloom filter. *Information Processing Letters*, 110(21):944–949, 2010.

[8] S. Cohen and Y. Matias. Spectral bloom filters. In *Proc. ACM SIGMOD*, pages 241–252, 2003.

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

[10] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher. Cuckoo filter: Practically better than bloom. 2009.

[11] 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)*, 8(3):281–293, 2000.

[12] F. Hao, M. Kodialam, T. Lakshman, and H. Song. Fast multiset membership testing using combinatorial bloom filters. In *Proc. IEEE INFOCOM*, 2009.

[13] A. Kirsch and M. Mitzenmacher. Less hashing, same performance: building a better bloom filter. In *Algorithms–ESA 2006*, pages 456–467. Springer, 2006.

[14] A. Kirsch, M. Mitzenmacher, and G. Varghese. Hash-based techniques for high-speed packet processing. In *Algorithms for Next Generation Networks*, pages 181–218. Springer, 2010.

[15] M. Lee, N. Duffield, and R. R. Kompella. Maple: A scalable architecture for maintaining packet latency measurements. In *Proc. ACM IMC 2012*.

[16] Y. Lu, B. Prabhakar, and F. Bonomi. Bloom filters: Design innovations and novel applications. In *43rd Annual Allerton Conference*, 2005.

[17] Y. Qiao, T. Li, and S. Chen. One memory access bloom filters and their generalization. In *Proc. IEEE INFOCOM*, pages 1745–1753, 2011.

[18] J. Roozenburg. A literature survey on bloom filters. *Research Assignment, November*, 2005.

[19] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz. Theory and practice of bloom filters for distributed systems. *Communications Surveys and Tutorials, IEEE*, 14(1):131–155, 2012.

[20] 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 *Proc. IEEE INFOCOM*, 2014.

[21] T. Yang, A. X. Liu, and et al. A shifting bloom filter framework for set queries. *arXiv:1510.03019*, 2015.

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