# A Shifting Framework for Set Queries

Tong Yang, Alex X. Liu, Muhammad Shahzad, Dongsheng Yang, Qiaobin Fu, Gaogang Xie, and Xiaoming Li

***Abstract*— Set queries are fundamental operations in computer networks. 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 further extend our shifting framework from BF-based data structures to sketch-based data structures, which are widely used to store multiplicities of items. 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.**

***Index Terms*— Set queries, Bloom filters, algorithms.**

## I. Introduction

### A. 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) [2] and the counting Bloom Filter (CBF) [3]. 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.

### B. 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

Manuscript received August 25, 2016; revised May 4, 2017; accepted June 20, 2017; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor M. Li. Date of publication August 17, 2017; date of current version October 13, 2017. This work was supported in part by the National Basic Research Program of China under Grant 2014CB340400, in part by the Primary Research & Development Plan of China under Grant 2016YFB1000304, in part by the NSFC under Grant 61472009 and Grant 61672061, in part by the Open Project Funding of CAS Key Lab of Network Data Science and Technology, Institute of Computing Technology, Chinese Academy of Sciences, in part by the Special Fund for Strategic Pilot Technology, Chinese Academy of Sciences, under Grant XDA06010302, in part by the National Science Foundation under Grant CNS-1318563, Grant CNS-1524698, Grant CNS-1421407, CNS-1616317, and Grant IIP-1632051, in part by the Shenzhen Research Project under Grant JCYJ20160330095313861, in part by the National Natural Science Foundation of China under Grant 61472184 and Grant 61321491, and in part by the Jiangsu Innovation and Entrepreneurship (Shuangchuang) Program. The preliminary version of this paper titled "A Shifting Bloom Filter Framework for Set Queries" was published in the Proceedings of the 42nd International Conference on Very Large Data Bases (VLDB) [1], New Delhi, India, September 2016. *(Corresponding author: Alex X. Liu.)*

T. Yang is with the Department of Computer and Science, Peking University, Beijing 100871, China, and also with the Collaborative Innovation Center of High Performance Computing, NUDT, Changsha 410073, China (e-mail: yangtongemail@gmail.com).

A. X. Liu is with the Department of Computer Science and Engineering, Michigan State University, East Lansing, MI 48824 USA, and also with the State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210093, China (e-mail: alexliu@cse.msu.edu).

M. Shahzad is with the Department of Computer Science, North Carolina State University, Raleigh, NC 27695 USA.

D. Yang and X. Li are with the Department of Computer and Science, Peking University, Beijing 100871, China.

Q. Fu is with the Department of Computer Science and Engineering, Boston University, Boston, MA 02215 USA.

G. Xie is with the Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100080, China.

Digital Object Identifier 10.1109/TNET.2017.2730227

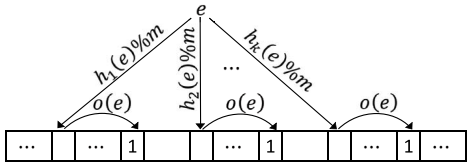



Fig. 1. Shifting bloom filter framework.

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 information*, *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.

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

*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 $e$.

*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 IV-B, 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.

*3) Multiplicity Queries:* For multiplicity queries, for each element $e$ in a multi-set $S$, 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. Another well known data structure to record and report multiplicities of elements is CM sketch [4]. CM sketches are more flexible and have more functions compared to Bloom filters. We apply our shifting framework to the CM sketches as well to achieve faster queries.

*4) Analysis of Computational Overhead:* Next, we briefly compare our shifting framework with BF. To insert an element, for $\text{ShBF}_\text{M}$ and $\text{ShBF}_\text{A}$, $o(e)$ is computed using only one hash function, and for $\text{ShBF}_\times$ no computation is required at all. Therefore, during an insertion, the number of hash functions

to be computed are less than or equal to $\frac{k}{2}+1$ . Similarly, during a query of ShBF, the number of hash functions to be computed are less than or equal to $\frac{k}{2}+2$. In comparison, Bloom filter needs $k$ computations of hash functions. Therefore, our shifting framework has a much smaller computational overhead. Furthermore, our framework needs fewer memory accesses because all bits to be read lie within a machine word length. After reading these bits, when checking them to restore the information, in the worst case, our framework goes through $k$ bits for $\text{ShBF}_\text{M}$, which are equal to BF, and $3k$ bits for $\text{ShBF}_\text{A}$, a little larger than $2k$ when using two individual BFs. For $\text{ShBF}_\times$, our framework checks $ck$ bits in the worst case, while BF does not support multiplicity query at all. Note that the average number of bits that our framework checks is significantly smaller than these worst case numbers of bits.

### C. 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 [5], 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 [6], 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. Furthermore, we have released the source code of our implementations of ShBF schemes and our implementation of existing state-of-the-art solutions [7].

## II. 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 on Bloom Filters and applications can be found in [**?**], [8]–[11].

### A. 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.* [12] proposed the Cuckoo filter and found that it is more efficient in terms of space and time compared to BF. This improvement comes at the cost of non-negligible probability of failing when inserting an element. To reduce the number of hash computation, Kirsch and Mitzenmacher 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.* [5] proposed to confine the output of the $k$ hash functions within certain number of machine `words`, which reduces the number of memory accesses during membership queries; but the cost again is increased FPR. In contrast, ShBF reduces the number of hash operations and memory access by about half while keeping FPR about the same as BF.

### B. 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 [3]. Other notable solutions include kBF [14], Bloomtree [15], Difference Bloom filter [16], Bloomier [17], Coded BF [18], Combinatorial BF [19]. 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.

### C. 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 [6]. 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.* [20] proposed Dynamic Count Filters (DCF), which combines the ideas of spectral BF and CBF, for multiplicity queries. 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 class of well-known data structure for multiplicity queries is the sketch, such as Count sketch [21], CM sketch [4], Pyramid sketch [22], and Slim-Fat sketch [23]. Note that our shifting framework can be applied to both Count sketches and CM sketches. As the CM sketch is much more accurate than the Count sketch, we focus on the application of our shifting framework on the CM sketch only in this paper.

## III. 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 $\text{ShBF}_\text{M}$ to denote the ShBF scheme for membership queries. Second, we describe the updating method of $\text{ShBF}_\text{M}$. Third, we derive the

TABLE I
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 |
| $\text{ShBF}_\text{M}$ | Shifting Bloom Filters for membership qrs. |
| $\text{ShBF}_\text{A}$ | Shifting Bloom Filters for association qrs. |
| $\text{ShBF}_\times$ | 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 |

FPR formula of $\text{ShBF}_\text{M}$. Fourth, we compare the performance of $\text{ShBF}_\text{M}$ with that of BF. Last, we present a generalization of $\text{ShBF}_\text{M}$. Table I summarizes the symbols and abbreviations used in this paper.

### A. ShBF$_M$ – Construction Phase

The construction phase of $\text{ShBF}_\text{M}$ 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 1 illustrates the construction phase of $\text{ShBF}_\text{M}$.

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.

### B. ShBF$_M$ – 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 $\frac{k}{2}$.

### C. ShBF$_M$ – Updating

Just like BF handles updates by replacing each bit by a counter, we can extend $\text{ShBF}_\text{M}$ to handle updates by replacing each bit by a counter. We use $\text{CShBF}_\text{M}$ to denote this counting version of $\text{ShBF}_\text{M}$. 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 $\text{CShBF}_\text{M}$. 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 $\text{CShBF}_\text{M}$ needs only $k/2$ memory accesses.

Due to the replacement of bits by counters, array $C$ in $\text{CShBF}_\text{M}$ uses much more memory than array $B$ in $\text{ShBF}_\text{M}$. To have the benefits of both fast query processing and small memory consumption, we can maintain both $\text{ShBF}_\text{M}$ and $\text{CShBF}_\text{M}$, 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.

### D. ShBF$_M$ – Analysis

We now calculate the FPR of $\text{ShBF}_\text{M}$, denoted as $f_{\text{ShBF}_\text{M}}$. Then, we calculate the minimum value of $\overline{w}$ so that $\text{ShBF}_\text{M}$

can achieve almost the same FPR as BF. Last, we calculate the optimum value of $k$ that minimizes $f_{\text{ShBF}_\text{M}}$.

*1) False Positive Rate:* We calculate the false positive rate of $\text{ShBF}_\text{M}$ in the following theorem.

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

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

*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.

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

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

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

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.

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

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

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

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 $\text{ShBF}_\text{M}$ is given by the following equation.

$$\begin{aligned} f_{\text{ShBF}_\text{M}} &= (1-p')^{\frac{k}{2}}\left((1-p')(1-p') + p'\left(1 - \frac{\overline{w}-2}{\overline{w}-1}p'\right)\right)^{\frac{k}{2}} \\ &= (1-p')^{\frac{k}{2}}\left(1 - p' + \frac{1}{\overline{w}-1}p'^2\right)^{\frac{k}{2}} \end{aligned} \tag{6}$$

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 Equ. 3, $p' \approx p$. Consequently, we get:

$$f_{\text{ShBF}_\text{M}} \approx (1-p)^{\frac{k}{2}}\left(1 - p + \frac{1}{\overline{w}-1}p^2\right)^{\frac{k}{2}} \tag{7}$$

which is the equation in the theorem statement. □

Note that the above calculation of FPRs is based on the original Bloom's FPR formula [2]. In 2008, Bose *et al.* [24] pointed out that Bloom's formula [2] is slightly flawed and gave a new FPR formula. 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.* [25] further pointed out that Bose's formula is also slightly flawed and gave another FPR formula. 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.

*2) Optimizing System Parameters:* Next, we describe how to calculate the optimum values for $\overline{w}$ and $k$.

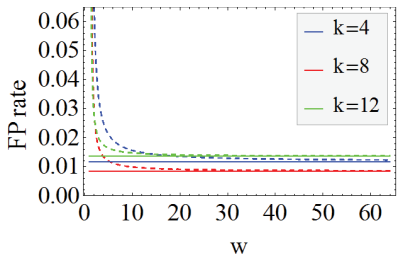



Fig. 2. FPR vs. $\overline{w}$. $m = 100000$, $n = 10000$. The dashed lines represent $\text{ShBF}_\text{M}$, while the solid lines represent BF.

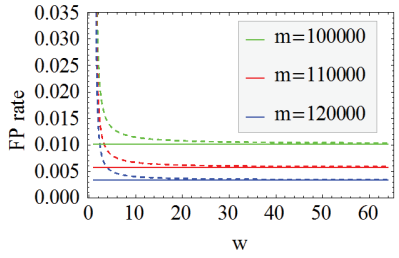



Fig. 3. FPR vs. $\overline{w}$. $k = 10$, $n = 10000$. The dashed lines represent $\text{ShBF}_\text{M}$, while the solid lines represent BF.

*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 $\text{ShBF}_\text{M}$ to achieve the same FPR with BF. We plot $f_{\text{ShBF}_\text{M}}$ of $\text{ShBF}_\text{M}$ as a function of $\overline{w}$ in Figures 2 and 3. The dashed lines in Figure 2 show $f_{\text{ShBF}_\text{M}}$ vs. $\overline{w}$ for $n = 10000$, $m = 100000$, and $k$ =4, 8, and 12 and the dashed lines in Figure 3 show $f_{\text{ShBF}_\text{M}}$ 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 $\text{ShBF}_\text{M}$ 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, $\text{ShBF}_\text{M}$ will achieve almost the same FPR as BF.

*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.*, $\frac{\partial}{\partial k} f_{\text{ShBF}_\text{M}} = 0$, 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 $\frac{\partial}{\partial k} f_{\text{ShBF}_\text{M}} = 0$ 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.

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

Substituting the value of $k_{\text{opt}}$ from the equation above into Equation (1), the minimum value of $f_{\text{ShBF}_\text{M}}$ is given by the following equation.

$$f^{\text{min}}_{\text{ShBF}_\text{M}} = 0.6204^{\frac{m}{n}} \tag{8}$$

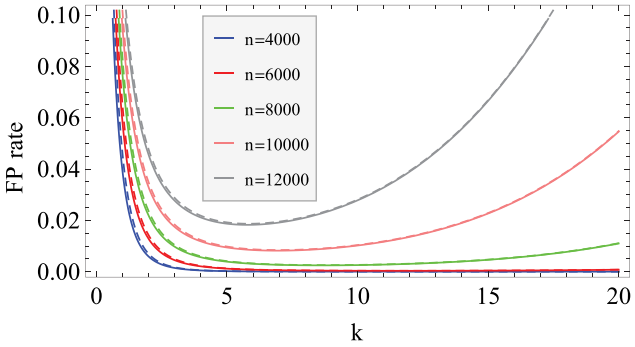



Fig. 4. $\text{ShBF}_\text{M}$ FPR vs. BF FPR.

### *E. Comparison of ShBF$_M$ FPR With BF FPR*

Our theoretical comparison of $\text{ShBF}_\text{M}$ and BF shows that the FPR of $\text{ShBF}_\text{M}$ is almost the same as that of BF. Figure 4 plots FPRs of $\text{ShBF}_\text{M}$ and BF using Equations (1) and (9), respectively for $m = 100000$ and $n = 4000$, 6000, 8000, 10000, 12000. The dashed lines in the figure correspond to $\text{ShBF}_\text{M}$ whereas the solid lines correspond to BF. We observe from this figure that the sacrificed FPR of $\text{ShBF}_\text{M}$ in comparison with the FPR of BF is negligible, while the number of memory accesses and hash computations of $\text{ShBF}_\text{M}$ are half in comparison with BF.

Next, we formally arrive at this result. We calculate the minimum FPR of BF as we calculated for $\text{ShBF}_\text{M}$ in Equation (8) 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 $\text{ShBF}_\text{M}$, BF can also report true with a small probability, which is the FPR of BF and has been well studied in literature [2]. It is given by the following equation.

$$f_{\text{BF}} = \left(1 - \left(1 - \frac{1}{m}\right)^{nk}\right)^k \approx \left(1 - e^{-\frac{nk}{m}}\right)^k \tag{9}$$

For given values of $m$ and $n$, the value of $k$ that minimizes $f_{\text{BF}}$ is $= \frac{m}{n}\ln 2 = 0.6931\frac{m}{n}$. Substituting this value of $k$ into Equation (9), the minimum value of $f_{\text{BF}}$ is given by the following equation.

$$f^{\text{min}}_{\text{BF}} = \left(\frac{1}{2}\right)^{\left(\frac{m}{n}\ln 2\right)} \approx 0.6185^{\frac{m}{n}} \tag{10}$$

By comparing Equations (8) and (10), we observe that the FPRs of $\text{ShBF}_\text{M}$ and BF are almost the same. Thus, $\text{ShBF}_\text{M}$ achieves almost the same FPR as BF while reducing the number of hash computations and memory accesses by half.

### *F. Generalization of ShBF$_M$*

As mentioned earlier, $\text{ShBF}_\text{M}$ 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.

$$p' = \left(1-\frac{t+1}{m}\right)^{\frac{kn}{t+1}} \approx e^{-\frac{kn}{m}} \tag{11}$$

Note that this probability formula is essentially $k$ times product of $e^{-\frac{n}{m}}$. Thus, we can treat our ShBF$_{\mathrm{M}}$ 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.

$$f = (1-p')^{\frac{k}{t+1}} \times (f_{group})^{\frac{k}{t+1}} \tag{12}$$

where

$$f_{group} = \frac{1}{t}\times(1-p')^2 \times \frac{(1-p')^t - \left(1-\frac{w-1-t}{w-1}\times p'\right)^t}{(1-p') - \left(1-\frac{w-1-t}{w-1}\times p'\right)} + p'\times\left(1-\frac{w-1-t}{w-1}\times p'\right)^t \tag{13}$$

*False Positive Rate of Generalized ShBF$_M$:* The generalization above is more complicated than ShBF$_{\mathrm{M}}$. Its false positive rate can be derived using Equations (12) and (13) as described next. When querying a non-existent element, a false positive occurs when a hash function group returns all $1s$, where the hash function group is $\langle h_i(x), o_1(x), \ldots, o_t(x)\rangle$ $(1 \leqslant i \leqslant \frac{k}{t+1})$. There are two cases to consider here.

1) If the corresponding bit of $h_i(x)$ is not set by the left $w-1$ bits, then $h_i(x)$ must cause $t$ bits, in the following $w-1$ bits, to be set to 1. There are $t+1$ situations in total. Note that the $r^{\text{th}}$ $(r \in [0,t])$ situation represents that the corresponding bits of $r$ hash functions out of $t$ are set by $h_i(x)$, and another $t-r$ bits are not set by $h_i(x)$. Considering the corresponding bits of the hash function subset $\langle o_1(x), \ldots, o_t(x)\rangle$, the probability that each of these bits is set by $h_i(x)$ is

$$\lambda_1 = \frac{t}{w-1} \tag{14}$$

and the probability that each of these bits is not set by $h_i(x)$ is

$$\lambda_2 = \left(1-\frac{t}{w-1}\right)\times(1-p') \tag{15}$$

Therefore, the probability that the $r^{\text{th}}$ situation occurs is $C_t^r \times \lambda_1^r \times \lambda_2^{t-r}$. The total probability that the $t+1$ situations occur is given by the following equation.

$$f_I = \Sigma_{r=0}^{t} C_t^r \times \lambda_1^r \times \lambda_2^{t-r} = (\lambda_1+\lambda_2)^t \tag{16}$$

2) If the corresponding bit of $h_i(x)$ is set by the left $w-1$ bits, then the problem can be divided into $t$ situations, where each situation has a probability of $1/t$. Note that the $l'-$th $(l' \in [0, t-1])$ situation represents that the maximum number of bits in the current subgroup $\langle o_1(x), \ldots, o_t(x)\rangle$, which are set to 1 by the previous hash function group causing the corresponding bit of $h_i(x)$ to be set to 1. Since our hash function group adopts partitioned Bloom filter, each hash function in the previous hash function group can cause at most 1 bit to be set to 1 in current hash function group. When the corresponding bit of $h_i(x)$ in the previous hash function group lies within the first $\frac{w-1}{t}$ bits, then the previous hash function group will cause 0 bit to be set to 1 in current subgroup $\langle o_1(x), \ldots, o_t(x)\rangle$, because there is only one possibility that the last hash function in the previous group causes the bit of current $h_i(x)$ to be set to 1. This is the $0^{\text{th}}$ situation. Similarly, for the $l'-$th situation, if the corresponding bit of $h_i(x)$ in the previous hash function group lies within the range $\left[w-1+l'\times\frac{w-1}{t}, w-1+(l'+1)\times\frac{w-1}{t}\right)$, there are at most $l'$ bits in the current hash function group set by the previous group, and at least $t-l'$ bits in the current hash function group not set by the previous group. Therefore, for the $l'-$th situation, the probability that all the bits in the current hash function group are 1 is given by the following equation.

$$\begin{aligned} f_{l'} &= \frac{1}{t}\times\left(\Sigma_{r'=0}^{l'} C_{l'}^{r'} \times \lambda_1^{r'} \times \lambda_2^{l'-r'}\right)\times(1-p')^{t-l'} \\ &= \frac{1}{t}(\lambda_1+\lambda_2)^{l'}\times(1-p')^{t-l'} \end{aligned} \tag{17}$$

The total probability that all the $t$ situations happen is

$$\begin{aligned} f_{II} &= \Sigma_{l'=0}^{t-1} f_{l'} = \Sigma_{l'=0}^{t-1}\frac{1}{t}\times\left((\lambda_1+\lambda_2)^{l'}\times(1-p')^{t-l'}\right) \\ &= \frac{1}{t}\times\left((\lambda_1+\lambda_2)^0\times(1-p')^{t-0} + \ldots \right. \\ &\quad \left. + \frac{1}{t}\times(\lambda_1+\lambda_2)^{t-1}\times(1-p')^{t-(t-1)}\right) \end{aligned} \tag{18}$$

Assuming $x = \Sigma_{l'=0}^{t}\left((\lambda_1+\lambda_2)^{l'}\times(1-p')^{t-l'}\right)$, we get the following equation

$$x\times\frac{\lambda_1+\lambda_2}{1-p'} = x-(1-p')^t+(\lambda_1+\lambda_2)^t \tag{19}$$

By solving Equ. 19, we obtain the solution

$$x = \frac{(1-p')^t-(\lambda_1+\lambda_2)^t}{(1-p')-(\lambda_1+\lambda_2)}\times(1-p') \tag{20}$$

Probability that case (1) happens is $C_{w-1}^1\times\left(\frac{1}{w-1}\right)\times(1-p') = (1-p')$. Therefore, the probability that case (2)

happens is $(1-(1-p'))=p'$. Combining 1) and 2), we know that when $h_i(x)$ is 1, the probability that all the bits of the current group $\langle o_1(x),\ldots,o_t(x)\rangle$ are 1 is $f_{group}$ given by Equation (13). Probability that the corresponding bits of the first $k/(t+1)$ hash functions are 1 is $(1-p')^{k/(t+1)}$. Therefore, the false positive of the generalized ShBF$_{\rm M}$ is

$$f=(1-p')^{\frac{k}{t+1}}\times(f_{group})^{\frac{k}{t+1}} \tag{21}$$

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

## IV. 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$_{\rm A}$ to denote the ShBF scheme for association queries. Second, we describe the updating methods of ShBF$_{\rm A}$. Third, we derive the FPR of ShBF$_{\rm A}$. Last, we analytically compare the performance of ShBF$_{\rm A}$ with that of iBF.

### A. ShBF$_A$ – Construction Phase

The construction phase of ShBF$_{\rm A}$ 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$_{\rm A}$ 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$_{\rm A}$ 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 smaller than the size of a machine word minus 7. 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$_{\rm A}$ 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$_{\rm A}$ 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 III-A. 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.

If due to some reason, we do not know which sets an element belongs to, $S_1-S_2$, $S_1\cup S_2$, or $S_2-S_1$, we should at least know whether the element belongs to $S_1$ or $S_2$. In this case, we can simply set the offset of elements in $S_1$ to 0, and set the offset of elements in $S_2$ to $o_2(e)$. If both o($e$) and $o_2(e)$ are 1, we consider that $e$ is in the intersection of A and B.

If set $S_1$ and $S_2$ are distributed in the network, we build two Bloom filters. One with $o(e)=0$ for $S_1$, and the other one with $o(e)=1$ for $S_2$. Next, we send the former Bloom filter to the server that holds set $S_2$, and apply the logical OR operation on these two Bloom filters. When querying an item, we can just check the $k$ hashed bits with offset 0 as well as the $k$ hashed bits with offset 1 to tell which sets the incoming item belongs to.

### B. 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$_{\rm A}$ 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$_{\rm A}$ records (but does not yet declare) $e\in S_1-S_2$. 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$_{\rm A}$ records $e\in S_1\cap S_2$. 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$_{\rm A}$ records $e\in S_2-S_1$.

Based on what ShBF$_{\rm A}$ recorded after analyzing the $3k$ bits, there are following 7 outcomes. If ShBF$_{\rm A}$ records that:

1) only $e\in S_1-S_2$, it declares that $e$ belongs to $S_1-S_2$.
2) only $e\in S_1\cap S_2$, it declares that $e$ belongs to $S_1\cap S_2$.
3) only $e\in S_2-S_1$, it declares that $e$ belongs to $S_2-S_1$.
4) both $e\in S_1-S_2$ and $e\in S_1\cap S_2$, it declares that $e$ belongs to $S_1$ but is unsure whether or not it belongs to $S_2$.
5) both $e\in S_2-S_1$ and $e\in S_1\cap S_2$, it declares that $e$ belongs to $S_2$ but is unsure whether it belongs to $S_1$.
6) both $e\in S_1-S_2$ and $e\in S_2-S_1$, it declares that $e$ belongs to $S_1-S_2\cup S_2-S_1$.
7) all $e\in S_1-S_2$, $e\in S_1\cap S_2$, and $e\in S_2-S_1$, it declares that $e$ belongs $S_1\cup S_2$.

Note that for all these seven outcomes, the decisions of ShBF$_{\rm A}$ 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$_{\rm A}$ 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*.

### C. ShBF$_A$ – Updating

Just like BF handles updates by replacing each bit by a counter, we can also extend ShBF$_{\rm A}$ to handle updates by

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

TABLE II
COMPARISON BETWEEN $\text{SHBF}_\text{A}$ 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 |
| $\text{ShBF}_\text{A}$ | m=(n1+n2-n3)k/ln2 | k+2 | k | $(1-0.5^k)^2$ | NO |

replacing each bit by a counter. We use $\text{CShBF}_\text{A}$ to denote this counting version of $\text{ShBF}_\text{A}$. 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 $\text{ShBF}_\text{A}$ and $\text{CShBF}_\text{A}$, but store array $B$ in fast SRAM and array $C$ in slow DRAM. After each update, we synchronize array $C$ with array $B$.

### D. $\text{ShBF}_\text{A}$ – Analysis

Recall from Section IV-B that $\text{ShBF}_\text{A}$ may report seven different outcomes. Next, we calculate the probability of each outcome. Let $P_i$ denote the probability of the $i^{\text{th}}$ 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

$$P(X \leqslant x, Y \leqslant y) = P(X \leqslant x) * P(Y \leqslant y) \tag{22}$$

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

$$\begin{aligned} &P(X + z \leqslant x + z, Y + z \leqslant y + z) \\ &\quad = P(X \leqslant x, Y \leqslant y) \\ &\quad = P(X \leqslant x) * P(Y \leqslant y) \\ &\quad = P(X + z \leqslant x + z) * P(Y = z \leqslant y + z) \end{aligned} \tag{23}$$

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 $\text{ShBF}_\text{A}$, the probability $p'$ that any given bit is still 0 is given by the following equation.

$$p' = (1 - 1/m)^{kn'} \tag{24}$$

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

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.

$$\begin{aligned} P_1 &= P_2 = P_3 = (1 - 0.5^k)^2 \\ P_4 &= P_5 = P_6 = 0.5^k * (1 - 0.5^k) \\ P_7 &= (0.5^k)^2 \end{aligned} \tag{25}$$

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 25. 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, $\text{ShBF}_\text{A}$ gives a *clear* answer, and with probability of only $9.756 * 10^{-4}$, $\text{ShBF}_\text{A}$ 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.

### E. Comparison Between $\text{ShBF}_A$ 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 II presents a comparison between $\text{ShBF}_\text{A}$ and iBF. We observe from the table that $\text{ShBF}_\text{A}$ 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 $\text{ShBF}_\text{A}$, when $m = (n_1 + n_2 - n_3)k/\ln 2$, the probability of answering a clear answer is$(1 - 0.5^k)^2$.

### F. Using $\text{ShBF}_A$ for Membership Queries

We can answer membership queries with the constructed bitmap for association queries. Specifically, Given two sets A and B, and a constructed bitmap for association query, now we are only curious about whether an element $e$ is in A or not. We just check the $2k$ bits, B$[h_i(e)\%m]$ and B$[h_i(e)\%m + o_1(e)]$ $(1 \leqslant i \leqslant k)$. If either all the former $k$ bits or all the latter $k$ bits are 1, $e$ is reported as in A.

## V. 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 show how our shifting model applies to CM sketches.

### A. 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 multi-set $S$, 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.

### B. 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 $k\lceil \frac{c}{w} \rceil$.

### C. ShBF$_\times$ – Updating

*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 multi-set $S$, ShBF$_\times$ first obtains its multiplicity $z$ from $B$ as explained in Section V-B. 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 V-B. 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.

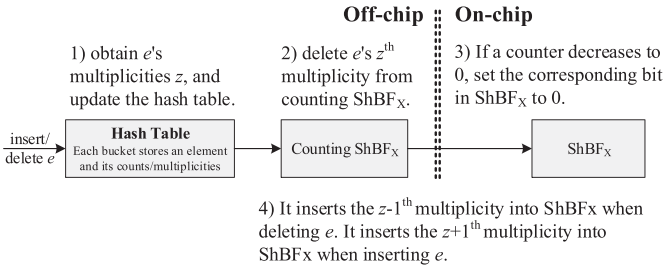



Fig. 5. The update process of ShBF$_\times$.

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.

*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, 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$_\times$.

*3) ShBF$_\times$ – Updating Without Accessory Data Structures:* At times, if we are not allowed to use the DRAM memory, or if we do not know the frequency in advance, we should get rid of both CShBF$_\times$ and the hash table, and our framework can be adjusted as follows: When querying an element $e$, we check the number of continuous 1s starting from the $k$ hashed bits. If the minimum number of continuous 1s is $m$, the estimated frequency of $e$ is reported as $m + 1$. When inserting an element $e$, first we query it in the ShBF$_\times$ to get an estimated frequency $freq(e)$. Next, we set the bits of offsets $o(e) = freq(e) + 1$ from every hashed bits to 1. When deleting an element $e$, first we query it in the ShBF$_\times$ to get an estimated frequency $freq(e)$. After that, we set the bits of offsets $o(e) = freq(e) - 1$ from every hashed bits to 1.

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.

### D. 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 multi-set $S_m$, 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.

$$f_0 \approx \left(1 - e^{-\frac{kn}{m}}\right)^k \tag{26}$$

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.

$$CR = (1 - f_0)^c \tag{27}$$

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.

$$CR' = (1 - f_0)^{j-1} \tag{28}$$

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.

### E. A Glimpse at a Unified Bloom Filter

Our shifting framework is a general data structure. It can support not only the prevalent types of queries, but also other kinds of queries that possibly emerge. First we show two new kinds of queries that our framework can handle: 1) given $z$ $(z \geqslant 2)$ sets, each pair of which have no intersection, for any incoming item $e$, which set does $e$ belong to? In this example, we can simply set $o(e)$ as the set ID. 2) Given $2z$ sets and an incoming item $e$, does $e$ belong to the first $z$ set or the other $z$ sets? In this example, we can define $o(e) = 0$ for the first $z$ sets and define $o(e) = 1$ for the remaining $z$ sets.

Furthermore, the shifting framework can even work when all three kinds of queries co-exist in the system. More specifically, there are $z$ multi-sets and three kinds of queries in the system: 1) whether an incoming item belongs to the $z$ sets? 2) which sets does $e$ belong to? and 3) if $e$ is in one of the $z$ sets, what is the frequency of $e$? Shifting CBF is a unified data structure that can answer the above three kinds of queries at the same time. The values in counters reveal the membership and the frequency, while the offset $o(e)$ of the non-zero counters represents the set ID of the element $e$. More specifically, when inserting $e$, we first compute $k$ hash functions to locate $k$ base counters. Then we do a right shift of $o(e) =$ (set id) to get another $k$ counters, and we only increase the latter $k$ counters by 1. When querying an element $e$, first we compute $k$ hash functions to locate the base counters. Then we check the $zk$ counters in the right of the $k$ base counters. If all the $k$ counters of offset $j$ is non-zero, then we report that $e$ belongs to the $j^{th}$ set of the $z$ sets, and its frequency is the smallest value of the $k$ counters of offset $j$.

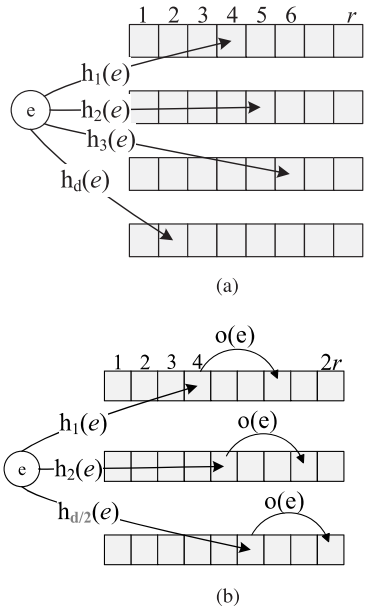



Fig. 6. FPR vs. $\overline{w}$. (a) CM sketch. (b) ShCM sketch.

### F. Shifting Count-Min Sketch

Besides Spectral BF, count-min sketch (CM sketch) can also be used to record and report multiplicities of elements in a multiset [4]. As shown in Figure 6(a), a CM sketch consists of $d$ vectors, and each vector has $r$ counters. Each vector $v_i$ $(1 \leqslant i \leqslant d)$ has a hash function $h_i(.)$ associated with it. In the construction phase, for each element $e$, we first calculate the $d$ hash functions associated with the $d$ vectors, and then increment the $d$ counters $v_1[h_1(e)], v_2[h_2(e)], \ldots, v_d[h_d(e)]$ by 1. In the query phase, given a query for the multiplicity of element $e$, the CM sketch reports the minimum value among the $d$ values $v_1[h_1(e)], v_2[h_2(e)], \ldots, v_d[h_d(e)]$ as the multiplicity of $e$.

Although CM sketch is simple and easy to implement, it is not as fast as BFs. One query on CM sketch needs $d$ hash computations and $d$ memory accesses if the length of all counters is smaller than a machine word. In this case, we can use our shifting framework to halve the number of memory accesses and hash functions. Figure 6(b) shows our shifting version of the CM sketch, called shifting count-min (ShCM) sketch. ShCM sketch consists of $d/2$ vectors and each vector has $2r$ counters. We represent the $i^{\text{th}}$ vector of ShCM sketch with $v_i$, where $1 \leqslant i \leqslant d/2$. Each vector $v_i$ has a hash function $h_i(.)$ associated with it. In the construction phase, for each element $e$, ShCM sketch first calculates the $d/2$ hash functions associated with the $d/2$ vectors, and then increment the $d/2$ counters $v_1[h_1(e)], v_2[h_2(e)], \ldots, v_{d/2}[h_{d/2}(e)]$ by 1. After that, it increments the counters $v_1[h_1(e) + o(e)], v_2[h_2(e) + o(e)], \ldots, v_{d/2}[h_{d/2}(e) + o(e)]$ by 1. In the query phase, given a query for the multiplicity of element $e$, the ShCM sketch reports the minimum value among the $d$ values $v_1[h_1(e)], v_2[h_2(e)], \ldots, v_{d/2}[h_{d/2}(e)]$ and $v_1[h_1(e) + o(e)], v_2[h_2(e) + o(e)], \ldots, v_{d/2}[h_{d/2}(e) + o(e)]$ as the multiplicity of $e$ To access the values of the pair of counters $v_i[h_i(e)]$ and $v_i[h_i(e)+o(e)]$ in a single memory access, we set

$o(e) = h_{d/2+1}\%(\overline{w}-1)+1$, where $\overline{w} \leqslant (w-7)/r$ and $w$ is the number of bits in a machine word.

*ShCM Analysis:* In this section, we derive the expression for the correctness rate $C_{ShCM}$ of ShCM sketch. The correctness rate of a sketch is defined as the expected value of the percentage of times the value reported by a sketch for the multiplicity of an element is exactly equal to the true multiplicity of that item. Before deriving an expression for $C_{ShCM}$, we first derive an expression for the correctness rate $C_{CM}$ of the CM sketch. A CM sketch with $d$ vectors and $w$ counters per vector is a special type of counting Bloom filter, called partitioned counting Bloom filter. This partitioned counting Bloom filter has $d \times w$ counters and $d$ hash functions, where the output of the hash function $h_i(.)$ lies in the range $[((i-1)\times w)+1, i\times w]$, where $1 \leq i \leq d$. Consequently, the correctness rate of the CM sketch can be computed by extending our results for the Bloom filters.

Consider a set $S$ that has $n$ distinct elements. When querying an item $e \notin S$, the query result of a partitioned counting Bloom filter is incorrect when the $d$ counters to which the $d$ hash functions point are all non-zero. Probability of this happening in the partitioned counting Bloom filters is the same as the correctness rate of the CM sketch, because as stated earlier, a CM sketch is essentially the same as partitioned counting Bloom filter. Formally, when there are $d\times w$ counters, and $n$ distinct items in the CM sketch, the correctness rate of CM sketch is given by the following equation.

$$C_{\text{CM}} = 1-\left(1-\left(1-\frac{d}{d\times\overline{w}}\right)^{n}\right)^{d} = 1-\left(1-\frac{1}{\overline{w}^{n}}\right)^{d} \tag{29}$$

As the ShCM sketch is similar to ShBF$_{\text{M}}$, we get a similar expression for the correctness rate $C_{ShCM}$ of ShCM sketch as Equation (1).

$$C_{ShCM} \approx (1-p)^{\frac{d}{2}}\left(1-p+\frac{1}{\overline{w}-1}p^{2}\right)^{\frac{d}{2}} \tag{30}$$

here $p = (1-\frac{d}{d\times\overline{w}})^{n} = (1-1/\overline{w})^{n}$.

Although Equ. 29 and 30 are quite different, the ultimate value is almost the same. This shows that the Shifting CM sketch achieves half the memory accesses while keeping the accuracy almost unchanged compared to the conventional CM sketch.

*Shifting Count-Min-Min Sketches:* To achieve even better accuracy, we propose an enhancement of ShCM sketch called Shifting Count-min-min Sketch, represented by ShCM$_{min}$. It differs from the ShCM sketch only in the construction phase. When inserting an element $e$, the ShCM$_{min}$ sketch first finds the minimum counter(s) among the $d$ counters $v_1[h_1(e)]\ldots v_{d/2}[h_{d/2}(e)]$ and $v_1[h_1(e)+o(e)]\ldots v_{d/2}[h_{d/2}(e)+o(e)]$, and then increases only the minimum counters(s) by 1. In this way, ShCM$_{min}$ increases fewer counters during each insertion compared to ShCM. Consequently, each counter in the $ShCM_{min}$ sketch is always less than or equal to the corresponding counter in the ShCM sketch. This implies that in querying elements, the multiplicity returned by ShCM$_{min}$ is no larger than ShCM sketch, thu, s reducing the error. The only shortcoming of ShCM$_{min}$ sketch is that it does not support deletions, and thus fits only the applications that do not need deletions.

## VI. 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.

### A. 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 evaluate the accuracy of our proposed schemes further, we also generated and used synthetic data sets. To generate the synthetic data sets, we simply used the rand() function in C and randomly produced items. We observe during our evaluations that the accuracies and speeds are similar as on the real-world network traces. Therefore, we only show the results from our experiments on real-world network traces.

*Hash Functions:* We collected several hash functions from open source web site [26] 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.

*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 running Windows 7. It has a 64KB L1 code cache, a 64KB L1 data cache, a 512KB L2 cache, and a 4MB L3 cache. The DRAM size of our computer is 8GB.

### B. ShBF$_M$ – Evaluation

In this section, we first validate the false positive rate of ShBF$_{\text{M}}$ calculated in Equation (1) using our experimental results. Then we compare ShBF$_{\text{M}}$ with BF and 1MemBF [13], which represents the prior scheme for answering membership queries, in terms of FPR, the number of memory accesses, and query processing speed.

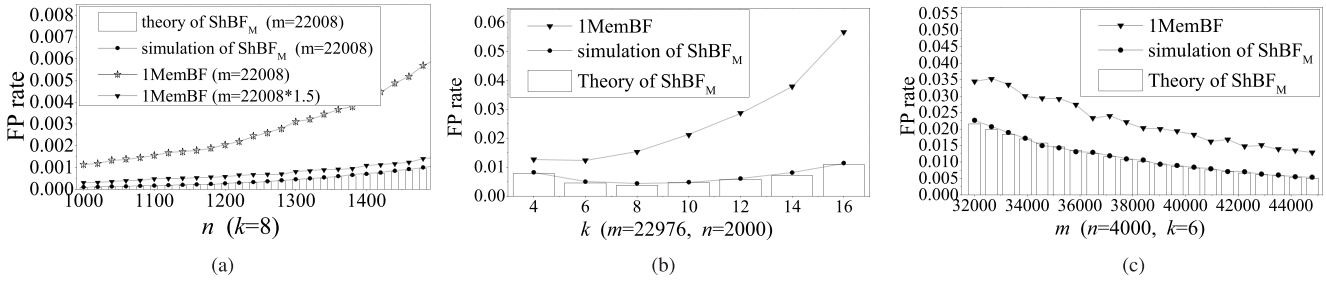



Fig. 7. Comparison false positive rates of $\text{ShBF}_\text{M}$ and 1MemBF. (a) Changing $n$. (b) Changing $k$. (c) Changing $m$.

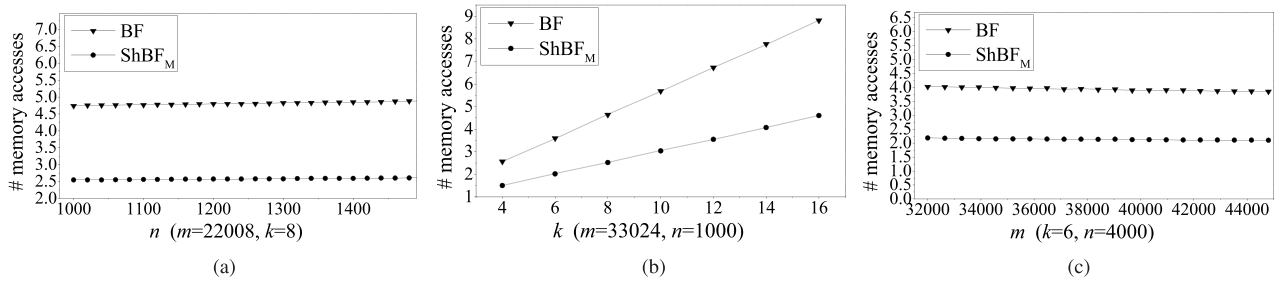



Fig. 8. Comparison of number of memory accesses per query of $\text{ShBF}_\text{M}$ and BF. (a) Changing $n$. (b) Changing $k$. (c) Changing $m$.

*1) $\text{ShBF}_M$ – False Positive Rate: Our experimental results show that the FPR of $\text{ShBF}_M$ 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 $\text{ShBF}_M$ and then repeatedly insert 20 elements until the total number of elements inserted into $\text{ShBF}_M$ became 1500. On inserting each set of 20 elements, we generated membership queries for $7,000,000$ elements whose information was not inserted into $\text{ShBF}_M$ and calculated the false positive rate. Figure 7(a) shows the false positive rate of $\text{ShBF}_\text{M}$ calculated through these simulations as well as through Equation (1). The bars in Figure 7(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 $\text{ShBF}_M$ 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 $\text{ShBF}_\text{M}$ is the same as relative error for BF calculated using simulation and the theory developed by Bloom [2]. Using same parameters, the FPR of 1MemBF is over $5 \sim 10$ times that of $\text{ShBF}_\text{M}$. If we increase the space allocated to 1MemBF for storage to 1.5 times of the space used by $\text{ShBF}_\text{M}$, the FPR of 1MemBF is still a little more than that of $\text{ShBF}_\text{M}$ 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 $\text{ShBF}_M$ is much smaller than that of 1MemBF when changing $k$ and $m$.* Figure 7(b) and Figure 7(c) show the FPRs of $\text{ShBF}_\text{M}$ and 1MemBF for different values of $k$ and $m$, respectively.

*2) $\text{ShBF}_M$ – Memory Accesses: Our results show that $\text{ShBF}_M$ 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 8(a), 8(b), and 8(c) show the number of memory accesses for $\text{ShBF}_M$ 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 $\text{ShBF}_M$ is also about half of that of standard BF.

*3) $\text{ShBF}_M$ – Query Processing Speed: Our results show that $\text{ShBF}_M$ 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 $\text{ShBF}_\text{M}$ reduces both hash computation and memory accesses. In our experiments, using those hashes which passed our randomness test, $\text{ShBF}_\text{M}$ exhibits faster query processing speed than that of 1MemBF. It is possible that 1MemBF is faster than $\text{ShBF}_\text{M}$ 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 9(a), 9(b), and 9(c) show the query processing speed for $\text{ShBF}_M$, standard BF, and 1MemBF for different values of $n$, $k$, and $m$, respectively.

### C. $\text{ShBF}_A$ – Evaluation

In this section, we first validate the probability of a clear answer of $\text{ShBF}_\text{M}$ calculated in Table II using our

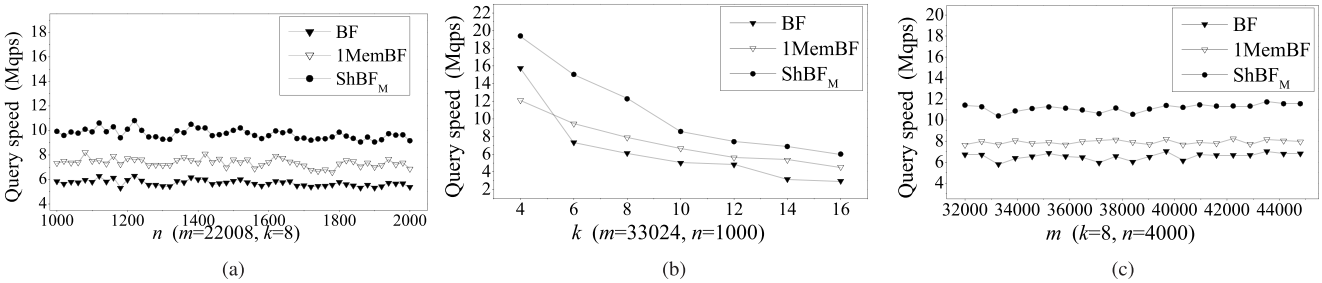



Fig. 9. Comparison of query processing speeds of $\text{ShBF}_\text{M}$, BF, and 1MemBF. (a) Changing $n$. (b) Changing $k$. (c) Changing $m$.

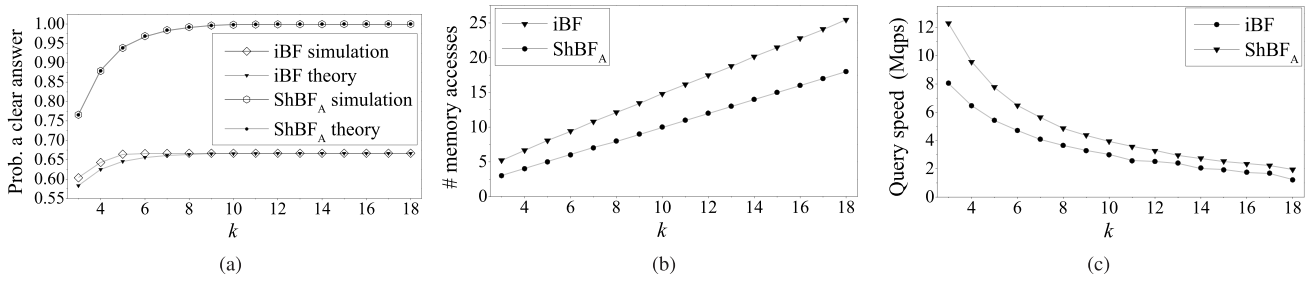



Fig. 10. Comparison of $\text{ShBF}_\text{A}$ and iBF. (a) Prob. a clear answer. (b) # memory accesses per query. (c) Query processing speed.

experimental results. Then we compare $\text{ShBF}_\text{A}$ with iBF in terms of FPR, memory accesses, and query processing speed.

*1) $\text{ShBF}_A$ – Probability of Clear Answer: Our results show that probability of clear answer for $\text{ShBF}_A$ calculated in Table II 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 10(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.

*2) $\text{ShBF}_A$ – Memory Accesses: Our results show that the average number of memory accesses per query of $\text{ShBF}_A$ is 0.66 times of that of iBF.* Figure 10(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.

*3) $\text{ShBF}_A$ – Query Processing Speed: Our results show that the average query processing speed of $\text{ShBF}_A$ is 1.4 times faster than that of iBF.* Figure 10(c) plots the the query processing speed of $\text{ShBF}_\text{A}$ and iBF for different values of $m$.

### D. $\text{ShBF}_\times$ – Evaluation

In this section, we first validate the correctness rate (CR) of $\text{ShBF}_\times$ calculated in Equation (27). Then we compare $\text{ShBF}_\times$ with spectral BF [6] and CM sketches [4] 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.

*1) $\text{ShBF}_\times$ – Correctness Rate: Our results show that the CR of $\text{ShBF}_\times$ calculated in Equation* (27) *matches with the CR calculated experimentally. Our results also show that on average, the CR of $\text{ShBF}_\times$ 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 $\text{ShBF}_\times$ is more memory efficient, we use 1.5 times the optimal memory (*i.e.*, $1.5 * nk/ln2$) for all the three filters. Figure 11(a) shows the results from our experiments for CR. Experimental results show that the CR calculated through experiments matches with the CR calculated theoretically.

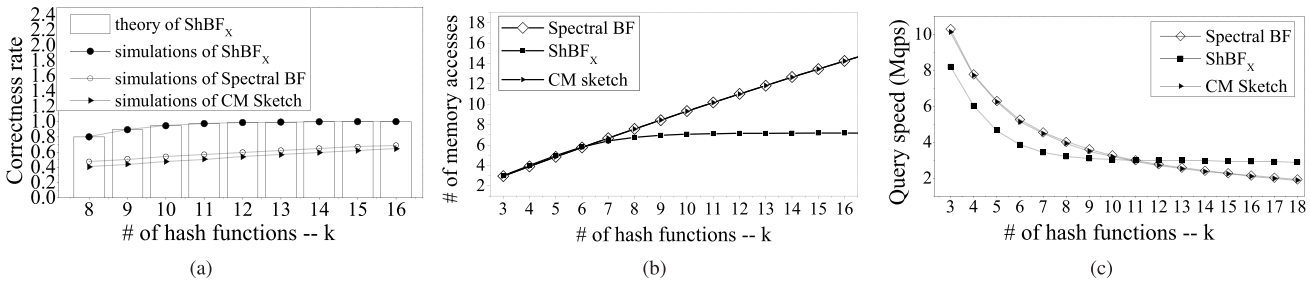
*2) $\text{ShBF}_\times$ – Memory Accesses: Our results show that the number of memory accesses of $\text{ShBF}_\times$ is smaller than that of spectral BF and CM sketches for $k \geqslant 7$, and almost equal for $k < 7$.* Figure 11(b) plots the number of memory accesses of $\text{ShBF}_\times$, CM sketch, and spectral BF, calculated from the same experiments that we used to plot Figure 11(a) except that $k$ ranges from 3 to 18.

*3) $\text{ShBF}_\times$ – Query Processing Speed: Our results show that $\text{ShBF}_\times$ is faster than spectral BF and CM sketches when $k \geqslant 11$.* We evaluate the query processing speed of $\text{ShBF}_\times$, CM sketch, and spectral BF using the same parameters as for Figure 11(b). Figure 11(c) plots the query processing speeds of $\text{ShBF}_\times$ and spectral BF. We observe from this figure that when $k > 11$, the average query processing speed of $\text{ShBF}_\times$ is over 3 Mqps.

### E. ShCM – Evaluation

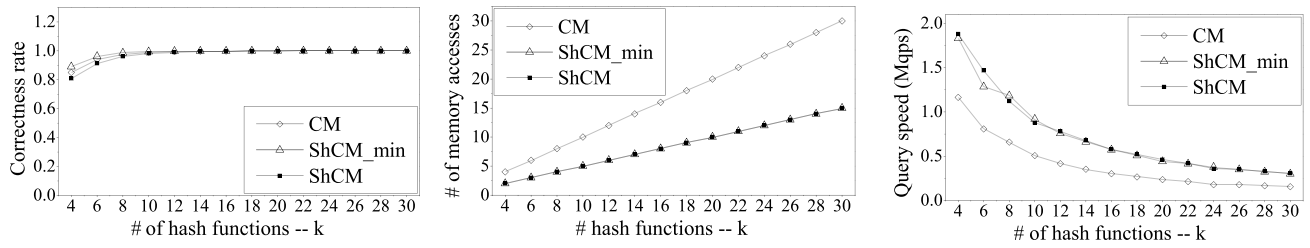
In this section, we compare the correctness rate, the number of memory accesses, and the query processing speed of ShCM sketch, $\text{ShCM}_{min}$ sketch, and CM sketch.

*1) ShCM Sketch - Correctness Rate: Our results show that the $\text{ShCM}_{min}$ sketch has higher correctness rate compared to*




(a) (b) (c)

Fig. 11. Comparison of ShBF$_\times$, Spectral BF, and CM sketches. (a) Correctness rate (CR). (b) Memory accesses. (c) Query processing speed.




(a) (b) (c)

Fig. 12. Comparison of CM sketches, ShCM sketches, and ShCM$_{min}$ sketches. (a) Correctness rate (CR). (b) Memory accesses. (c) Query processing speed.

*the CM sketch.* For the experiments reported in this section, we set $c$ = 15, $n$ = 1,000,000, and vary $k$ in the range $2 \leqslant k \leqslant 30$. We use 4 bits for each counter. For each value of $k$, we use the same memory for the the three sketches, which means that the ShCM sketch and the ShCM$_{min}$ sketch have twice the number of vectors, and half the number of counters per vector compared with the CM sketch. The CM sketch has 1,000,011 vectors, while the ShCM sketch and the ShCM$_{min}$ sketch have vectors. The CM sketch has $k$ counters per vector, while the ShCM sketch and the ShCM$_{min}$ sketch have $k$/2 counters per vector. We perform 200,000 queries for each sketch. Figure 12(a) shows the results from our experiments for the correctness rate. Experimental results show that the correctness rate of the ShCM sketch is almost the same as that of the CM sketch, and the ShCM$_{min}$ sketch outperforms the CM sketch notably, especially when $k$ is small.

*2) ShCM Sketch - Memory Access: Our results show that the memory accesses of the ShCM sketch and the ShCM$_{min}$ sketch are about half of that of the CM sketch.* Figure 12(b) shows the results from our experiments for memory accesses. As all of the three sketches access the memory once for each column and the ShCM sketch and the ShCM$_{min}$ sketch have only half of the columns, the result is obvious.

*3) ShCM Sketch - Query Processing Speed: Our results show that the ShCM sketch and the ShCM$_{min}$ sketch are much faster compared to the CM sketch.* We evaluate the query processing speed of the ShCM sketch and CM sketch using the same parameters as for Figure 12(a). Figure 12(c) plots the query processing speeds of the three sketches. We observe from this figure that the average query processing speed of the ShCM sketch and the ShCM$_{min}$ sketch are always higher than the CM sketch. Although all of the three architectures need to calculate $k$ hash functions, the ShCM sketch and the ShCM$_{min}$ sketch only need to access the memory $k/2$ times instead of $k$ times, which saves nearly half the amount of time.

## VII. Conclusion

The key contribution of this paper is in proposing Shifting 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.

## Acknowledgements

The authors would like to thank the anonymous reviewers for their thoughtful suggestions.

## References

[1] T. Yang *et al.*, "A shifting bloom filter framework for set queries," *Proc. VLDB Endowment*, vol. 9, no. 5, pp. 408–419, 2016.
[2] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970.
[3] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: A scalable wide-area Web cache sharing protocol," *IEEE/ACM Trans. Netw.*, vol. 8, no. 3, pp. 281–293, Jun. 2000.
[4] G. Cormode and S. Muthukrishnan, "An improved data stream summary: The count-min sketch and its applications," *J. Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.
[5] Y. Qiao, T. Li, and S. Chen, "One memory access bloom filters and their generalization," in *Proc. IEEE INFOCOM*, Apr. 2011, pp. 1745–1753.
[6] S. Cohen and Y. Matias, "Spectral bloom filters," in *Proc. ACM SIGMOD*, 2003, pp. 241–252.
[7] *A Shifting Bloom Filter Framework for Set Queries*. Accessed on Jul. 2016. [Online]. Available: http://net.pku.edu.cn/~yangtong/

[8] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz, "Theory and practice of bloom filters for distributed systems," *IEEE Commun. Surveys Tuts.*, vol. 14, no. 1, pp. 131–155, Feb. 2012.
[9] A. Kirsch, M. Mitzenmacher, and G. Varghese, "Hash-based techniques for high-speed packet processing," in *Algorithms for Next Generation Networks*. London, U.K.: Springer, 2010, pp. 181–218.
[10] A. Broder and M. Mitzenmacher, "Network applications of bloom filters: A survey," *Internet Math.*, vol. 1, no. 4, pp. 485–509, 2004.
[11] T. Yang *et al.*, "Guarantee ip lookup performance with fib explosion," in *Proc. ACM SIGCOMM*, 2014, pp. 39–50.
[12] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo filter: Practically better than bloom," in *Proc. 10th ACM Int. Conf. Emerg. Netw. Experim. Technol.*, 2009, pp. 75–88.
[13] A. Kirsch and M. Mitzenmacher, "Less hashing, same performance: Building a better bloom filter," in *Algorithms–ESA*. Berlin, Germany: Springer, 2006, pp. 456–467.
[14] 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*, Apr. 2014, pp. 1150–1158.
[15] M. Kyoon, J. Son, and S.-H. Shin, "Bloom tree: A search tree based on bloom filters for multiple-set membership testing," in *Proc. IEEE INFOCOM*, Apr. 2014, pp. 1429–1437.
[16] D. Yang, D. Tian, J. Gong, S. Gao, T. Yang, and X. Li, "Difference Bloom filter: A probabilistic structure for multi-set membership query," in *Proc. IEEE ICC*, 2017, doi: 10.1109/ICC.2017.7996678.
[17] 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, pp. 30–39.
[18] Y. Lu, B. Prabhakar, and F. Bonomi, "Bloom filters: Design innovations and novel applications," in *Proc. 43rd Annu. Allerton Conf.*, 2005, pp. 1006–1015.
[19] F. Hao, M. Kodialam, T. Lakshman, and H. Song, "Fast multiset membership testing using combinatorial bloom filters," in *Proc. IEEE INFOCOM*, Apr. 2009, pp. 513–521.
[20] J. Aguilar-Saborit, P. Trancoso, V. Muntes-Mulero, and J.-L. Larriba-Pey, "Dynamic count filters," *ACM SIGMOD Rec.*, vol. 35, no. 1, pp. 26–32, 2006.
[21] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," in *Automata, Languages and Programming*. Berlin, Germany: Springer, 2002.
[22] T. Yang, Y. Zhou, H. Jin, S. Chen, and X. Li, "Pyramid sketch: A sketch framework for frequency estimation of data streams," in *Proc. VLDB Endowment*, 2017, pp. 1–2.
[23] T. Yang *et al.*, "SF-sketch: A fast, accurate, and memory efficient data structure to store frequencies of data items," in *Proc. IEEE ICDE*, Apr. 2017, pp. 103–106.
[24] P. Bose *et al.*, "On the false-positive rate of bloom filters," *Inf. Process. Lett.*, vol. 108, no. 4, pp. 210–213, 2008.
[25] K. Christensen, A. Roginsky, and M. Jimeno, "A new analysis of the false positive rate of a bloom filter," *Inf. Process. Lett.*, vol. 110, no. 21, pp. 944–949, 2010.
[26] *A Shifting Bloom Filter Framework for Set Queries*. Accessed on Jan. 2015. [Online]. Available: http://burtleburtle.net/bob/hash/evahash.html

**Tong Yang** received the Ph.D. degree in computer science from Tsinghua University in 2013. He visited the Institute of Computing Technology, Chinese Academy of Sciences, China, from 2013 to 2014. He is currently a Research Assistant with the Computer Science Department, Peking University. His research interests include IP lookups, bloom filters, sketches, and KV stores.

**Alex X. Liu** received the Ph.D. degree from The University of Texas at Austin in 2006. He is currently an Associate Professor with the Department of Computer Science and Engineering, Michigan State University. His research interests include networking, security, and dependable systems. He received the IEEE and IFIP William C. Carter Award in 2004 and the U.S. National Science Foundation CAREER Award in 2009. He also received the Withrow Distinguished Scholar Award in 2011 at Michigan State University.

**Muhammad Shahzad** received the Ph.D. degree in computer science from Michigan State University in 2015. He is currently an Assistant Professor with the Department of Computer Science, North Carolina State University, USA. His research interests include design, analysis, measurement, and modeling of networking and security systems. He received the 2015 Outstanding Graduate Student Award, the 2015 Fitch Beach Award, and the 2012 Outstanding Student Leader Award at Michigan State University.

**Dongsheng Yang** is currently pursuing the bachelor's degree with Peking University, guided by T. Yang. He has published a few papers about networking and big data. His research interests include social network and data mining.

**Qiaobin Fu** is currently pursuing the Ph.D. degree with the Department of Computer Science, Boston University. He is a member of the Networks Research Group. His advisor is Prof. J. W. Byers. He is also involved in designing algorithms and building systems in networking.

**Gaogang Xie** received the Ph.D. degree in computer science from Hunan University, Changsha, China, in 2002. He is currently a Professor and the Director of the Network Technology Research Center, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China. His research interests include programmable virtual routers, future Internet architecture, and Internet measurement.

**Xiaoming Li** is currently a Professor in computer science and technology and the Director of the Institute of Network Computing and Information Systems, Peking University, China. He has been leading the effort of developing a Chinese search engine (Tianwang) since 1999. He is also the Founder of the Chinese Web Archive (Web InfoMall). His current research interest is in search engine and Web mining.