A Shifting Filter Framework for Dynamic Set Queries

Pengtao Fu, Lailong Luo, Deke Guo, *Senior Member, IEEE, Member, ACM*, Shangsen Li, and Yun Zhou, *Member, IEEE*

***Abstract*— Set query is a fundamental problem in computer systems. Plenty of applications rely on the query results of membership, association, and multiplicity. A traditional method that addresses such a fundamental problem is derived from Bloom filter. However, such methods may fail to support element deletion, require additional filters or apriori knowledge, making them unamenable to a high-performance implementation for dynamic set representation and query. In this paper, we envision a novel sketch framework that is multi-functional, non-parametric, space efficient, and deletable. As far as we know, none of the existing designs can guarantee such features simultaneously. To this end, we present a general shifting framework to represent auxiliary information (such as multiplicity, association) with the offset. Thereafter, we specify such design philosophy for a hash table horizontally at the slot level, as well as vertically at the bucket level. Theoretical and experimental results jointly demonstrate that our design works exceptionally well with three types of set queries under small memory.**

***Index Terms*— Dynamic set queries, element deletion, Cuckoo filters, Bloom filters, shifting framework.**

I. Introduction

SET queries upon membership, association, and multiplicity are fundamental requests in computer systems. They are involved in various applications, such as indexing in data centers, distributed file systems, database storage, data duplication, set reconciliations, network packets processing, and network traffic measurement.

Membership query decides whether an element exists in a given set or not. A typical scenario is regular expression matching. Many caches, routers and storage systems in networking and distributed systems [1], [2], [3], [4], [5], [6], [7] rely on membership query. For instance, Akamai's CDN (content delivery network) uses membership query to deal with HTTP request and object cache [8]. Moreover, various network applications, such as IP lookup and network packet classification [9], also often involve membership queries.

Association query fetches the affiliations of a given element. For instance, there are H sets $S_1, \cdots, S_H$ with intersections and an element e from those sets, association query is to query which set(s) e belongs to. Summary Cache [10] uses association query to achieve distributed caching. Association query is also involved in many other applications, including set reconciliations [11], indexing in data centers [12], distributed file system [13], database indexing [13] and data duplication [14].

Multiplicity query indicates how many times a given element appears in a multi-set, i.e., tells the multiplicity of this element. For example, Yahoo's Hadoop, PIG [15], and Microsoft's Cosmos [16] use multiplicity queries to assist the analysis of enormous data sets, such as web crawls, search logs, and click streams. Besides, many network measurement tasks [17], [18], including delay, burst detection, flow size estimation, flow distribution, and heavy hitter detection [19], [20], [21], [22], [23], [24], also often use multiplicity query.

As aforementioned, set queries address two types of information for each element e: 1)existence (membership) information, i.e., whether e is in a set; 2)auxiliary information, some additional information such as the frequency of e (i.e., multiplicity information) or which set that e is in (i.e., association information). Set queries rely on sketches to record and track the data information above, and have been widely used in computer networks, such as packet routing and forwarding, web caching, network monitoring, security enhancement, content delivery, etc [25]. Therefore, it is of great significance to improve set query performance with an elegant probabilistic data structure. We envision a design that properly concerns the following four design rationales for dynamic set queries.

- **Multi-functional (MF)**. The data structure supports various set queries, including the membership, association, and multiplicity queries as mentioned above.
- **Non-parametric (NP)**. The data structure works without any parametric restriction. In more detail, the data structure should be able to independently support set queries and maintain considerable performance irrespective of the parameter of sets.
- **Space efficient (SE)**. The data structure is space-friendly and uses as little space as possible. This is extremely important for space-scare scenarios, e.g., wireless sensor networks and commodity switches.

Manuscript received 22 February 2022; revised 13 November 2022 and 11 January 2023; accepted 18 February 2023; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor L. Huang. Date of publication 24 February 2023; date of current version 17 October 2023. This work was supported in part by the National Natural Science Foundation of China under Grant 62002378 and Grant U19B2024 and in part by the Research Funding of the National University of Defense Technology (NUDT) under Grant ZK20-30. *(Corresponding author: Lailong Luo.)*

Pengtao Fu, Deke Guo, Shangsen Li, and Yun Zhou are with the College of Systems Engineering, National University of Defense Technology, Changsha 410073, China.

Lailong Luo is with the College of Systems Engineering and College of Computer Science and Technology, National University of Defense Technology, Changsha 410073, China (e-mail: luolailong09@nudt.edu.cn).

Digital Object Identifier 10.1109/TNET.2023.3247628

TABLE I
OUR WORK V.S. OTHER SOLUTIONS

	BF [26]	CF [27]	CBF [28]	ABF [29]	MCF [11]	ShBF [7]	ES [17]	NA	SF
MF	×	×	×	×	×	✓	×	✓	✓
NP	—	—	✓	×	✓	×	✓	✓	✓
SE	✓	✓	×	✓	✓	✓	✓	×	✓
DE	×	✓	✓	×	✓	×	×	✓	✓

- **Deletable (DE)**. The data structure supports deleting a given element from a target set. Element deletion is essential for representing a dynamic set wherein the elements join and leave frequently.

These rationales, if realized, will bring unprecedented benefits for dynamic set queries. The existing data structures, however, fail to achieve the four rationales properly and simultaneously. As shown in Table I, many sketch data structures are proposed to support the above tasks in terms of membership query (Bloom filter (BF) [26], Cuckoo filter (CF) [27]), association query (Marked Cuckoo filter (MCF) [11]), and multiplicity query (Counting Bloom filter (CBF) [28], Adaptive Bloom filter (ABF) [29], Elastic Sketch (ES) [17]. In addition to suffering from the inability of deletion, poor space efficiency, or parametric performance (i.e., filter works with additional filters or apriori knowledge, or degrades with the range of auxiliary information expands), these efforts mainly work for one specific query, thus do not appear amenable for general set queries.

To this end, the state-of-the-art Shifting Bloom filter (ShBF) [7] addresses this problem by representing the auxiliary information (such as association or multiplicity) with the offset bits in the filter. However, ShBF is limited with two fatal drawbacks: 1) ShBF fails to represent dynamic sets, as it does not support element deletion; 2) The performance and design mechanism of ShBF are quite parametric. First, ShBF suffers from substantial performance degradation when the number of represented sets or the multiplicity of elements increases. Second, ShBF cannot work without the assistance of additional filters and apriori knowledge. Specifically, for association query, it has to build additional filters to identify the affiliation of elements; for multiplicity query, it must know the maximum multiplicity in advance during the query phase. One naive approach (NA) to address these issues is adding several bits in each slot of the deletion-supporting Cuckoo filter to store auxiliary information directly. However, each counter would use more bits to accommodate the maximum multiplicity of the most frequent element, which may be space-prohibitive, especially when the element frequencies are excessively skewed.

Consequently, a novel sketch that simultaneously covers the above four design rationales is required for dynamic set representation and query. To this end, this paper presents Shifting filter (SF), a new design of Cuckoo filter. The key idea of our method is designing a universal shifting framework and applying it to a partitioned CF to represent the auxiliary information of a set element. To be specific, Shifting filter uses the shifting framework in the vertical direction to store the element in a fixed position in the candidate buckets, instead of storing it in any available slot like CF, so as to obtain faster insertion, query, and deletion speed than CF. For association queries, Shifting filter further introduces a mark field in each slot to ease the affiliation representation of elements. For multiplicity query, Shifting filter encodes the multiplicity of a given element through our space-saving shifting framework assisted by the count field, avoiding performance degradation and enormous space overhead caused by excessive multiplicities. Moreover, we redesign CF by applying the shifting framework in the horizontal direction to improve space efficiency.

Table I shows the empirical results through a brief comparison of several related solutions. It indicates that Shifting filter achieves all the design rationales simultaneously and has the most remarkable advantages. As fundamental research, the proposed method can efficiently support set queries, including the membership, association and multiplicity query, instead of working like various sketch-based measurement designs, which rely on only a single query mechanism and help a single kind of task. Specifically, sketch in network measurement relies on multiplicity queries to support a specific measurement task, such as delay, burst detection, etc., as mentioned above. Therefore, our design can be widely used in many computer network fields, including network measurement, to provide efficient new technologies and theories support. The contributions of this paper can be summarized as follows:

- We present a universal shifting framework and use it to design Shifting filter, a novel redesign of the CF that can quickly process set queries and support set element deletion, using a small amount of memory.
- Our shift framework is tailored for the hash table. It works together with fingerprint and counting fields, which enables the Shifting filter easier to implement element deletion, multi-functional, non-parametric, and space efficient than traditional methods.
- We further improve the Shifting filter by applying the shifting framework in the vertical and horizontal directions for different application scenarios.
- We conduct comprehensive experiments to compare Shifting filter with its same-kinds. Theoretical analysis and numerical results demonstrate that Shifting filters realize comparable or better performance than competitors in terms of query throughput, insertion throughput, deletion throughput and accuracy.

The rest of this paper is organized as follows. Section II reviews the background and the related work. Section III describes the design principle of shifting framework and its two concrete specifications. Section IV, V and VI describes three variants of SF for membership, association and multiplicity query, respectively. Section VII presents a theoretical analysis of SF in different situations. Section VIII evaluates our design with comprehensive experiments upon BF, ShBF, ABF, CF, VCF, NA and ES. Finally, Section IX concludes this work.

II. RELATED WORK

A. Bloom Filter and Its Variants

Bloom filter. BF represents elements through a bit vector with m bits, which are initially set as 0. To insert any element e in a set S with n elements, k independent hash functions are employed to map the element to k positions in the bit vector. These mapped positions are all set to 1. To query e, BF just checks the k corresponding positions. If all the k bits are non-zero, BF indicates that $e \in S$; otherwise, BF judges that $e \notin S$.

However, BF may mistake an alien element as a set member when the k hashed positions in the bit vector are all 1 due to the unavoidable hash conflicts. The probability of such false positive error is $\epsilon_{BF} = (1 - e^{-kn/m})^k$. Intuitively, standard BF does not support element deletion unless it reconstructs the whole bit vector, as directly resetting the corresponding bits from 1 to 0 may cause false negative results for other elements. Many efforts have been made to further enhance BF in terms of deletion supporting (Counting BF [28]), capacity resizing (Dynamic BF [30]), reverse decoding (Invertible BF [31]), multiplicity representing (ABF [29] and ShBF [7]).

Adaptive Bloom filter. Adaptive BF [29] utilizes $k + c + 1$ hash functions, where c is the maximum multiplicity for elements in the set S. To insert an element e with multiplicity $c(e)$, ABF represents its membership information by setting the k bits to 1s and encodes its multiplicity information in the number of programmed 1s among the remaining $c(e)$ bits. ABF checks only the k corresponding bits to tell the existence information of the queried element. Moreover, ABF tackles the multiplicity query via counting how many bits are set to 1 by the latter hash functions. For instance, for an element e whose multiplicity is 10, ABF must check the following 11 bits of $h_1(e)\%m, \cdots, h_k(e)\%m$ until it matches the first 0 bit. Note that, under the framework of ABF, the membership information and multiplicity information in the bit vector may interfere with each other.

Shifting Bloom filter. Shifting BF [7] is an array of m bits which are all initialized to 0. For each element e, ShBF encodes its existence information in k independent 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, ShBF sets the bits at locations $(h_1(e) + o(e))\%m, \cdots, (h_k(e) + o(e))\%m$ to 1. ShBF can support membership query for sure, but also customize other types of queries by adjusting the offset function. For association query, if there are j different affiliations, ShBF uses offset function $o_i(e) = o_{i-1}(e) + h_{k+i}(e)\%((\bar{w}-1)/(j-1))+1$ to represent the element with the ith affiliation, where $\bar{w}$ is a function of machine word size and $o_0(e) = 0$. For instance, for this type of queries with two sets S_1 and S_2 and an element e in $S_1 \cup S_2$, there are three cases: 1) for $e \in S_1 - S_2$, the offset function $o_0(e) = 0$; 2) for $e \in S_2 - S_1$, the offset function $o_1(e) = h_{k+1}(e)\%((\bar{w}-1)/2) + 1$; 3) for $e \in S_1 \cap S_2$, the offset function $o_2(e) = o_1(e) + h_{k+2}(e)\%((\bar{w}-1)/2) + 1$. For multiplicity query, the offset function can be set as the multiplicity of the element. In the query phase, given the maximum value c of all offsets, ShBF will first check the c bits at positions $h_1(e)\%m, (h_1(e)+1)\%m, \cdots, (h_1(e)+c-1)\%m$. For the above j^{th} bit which is 1, if the k bits in the locations $(h_1(e) + j)\%m, \cdots, (h_k(e) + j)\%m$ are all non-zero, thus j may be a candidate answer for the multiplicity of element e. Then ShBF reports the one with the largest value among these candidates as the final answer to avoid underestimate.

B. Cuckoo Filter and Its Variants

Cuckoo filter. Cuckoo filter [27] is a lightweight probabilistic data structure that represents an element e by storing its fingerprint $f(e)$ directly. Structurally, CF maintains a cuckoo hash table consisting of m buckets, each of which contains b slots to accommodate at most b fingerprints. Standard CF provides two candidate buckets $h_1(e)$ and $h_2(e)$ for each element e through the partial cuckoo hashing technique [32]: $h_1(e) = hash(e), h_2(e) = h_1(e) \oplus hash(f(e))$. CF can support element deletions and constant-time membership queries. To insert e, CF first explores two alternate buckets, and $f(e)$ will be stored in either of them if there is an empty slot. Otherwise, CF randomly kicks out a fingerprint in one of the candidates and then reinserts $f(e)$ in the slot that has just been vacated. After that, CF calculates the other candidate bucket of the victim fingerprint and tries to reinsert it into that bucket. CF recursively performs this relocation scheme until a bucket has an available slot or the relocation reaches MAX times. To answer a membership query of e, CF just searches $f(e)$ in e's candidate buckets. If $f(e)$ is found, CF judges $e \in S$; otherwise, $e \notin S$. Due to hash conflicts, CF may tell a positive result when querying an alien element, such error occurs with a bounded probability $\epsilon_{CF} = 1-(1-\frac{1}{2^l})^{2b} \approx \frac{b}{2^{l-1}}$ where l is the length of the fingerprint. Typical CF variants include Adaptive CF [33], Consistent CF [34], Dynamic CF [35], Simplified CF [36], Vertical CF [37], Marked CF [11] and Vacuum filter [38]. They are investigated to enhance CF in terms of false positive rate, flexibility, key set extension, theoretical guarantee, space efficiency, set reconciliation and query throughput.

Vertical Cuckoo filter. As a new CF design, VCF improves the partial cuckoo hash strategy in CF by introducing two inverse bitmasks (e.g., bm_1 and bm_2). In this manner, VCF can provide two more candidate buckets for each element than CF, that is, $h_3(e) = hash(e) \oplus hash(f(e)) \wedge bm_1, h_4(e) = hash(e) \oplus hash(f(e)) \wedge bm_2$. The four candidate buckets of VCF can be indexed by each other without additional hash computation or access to element content. More candidates make VCF obtain a higher space utilization and a faster insertion throughput than CF, with a slight compromise of query speed and false positive rate.

Marked Cuckoo filter. Marked Cuckoo filter [11] is designed to represent the sets in each reconciliation participant. The MCF attaches a mark field in each slot to indicate which set(s) the stored fingerprint belongs to. MCF naturally inherits the functionalities from the standard CF, including element insertion, query, and deletion.

C. Sketch-Based Measurement Design

Elastic Sketch. Elastic Sketch (ES) [17] is a state-of-the-art design for network measurements. ES proposes a separation technique to separate elephant flows from mouse flows and keeps them in different parts. With such a framework, ES prefers an approximate rather than exact multiplicity query to achieve memory efficiency and speed up the querying at the cost of introducing small errors. The main drawbacks of ES are single-function, no support for deletion, and inaccurate query. Unfortunately, this limits the use of such fast and space-friendly ES in many practical applications.

III. Shifting Filters

This section proposes a universal shifting framework. On this basis, we implement our shifting framework on CF and VCF, and design a novel Shifting filter. We further extend it to two versions for different application scenarios.

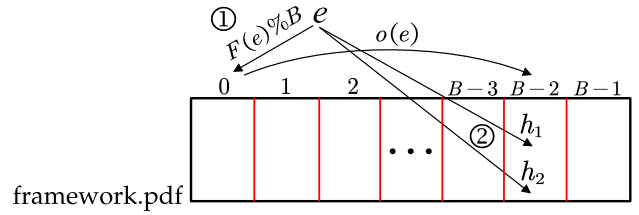


Fig. 1. An illustrative example of Shifting framework.

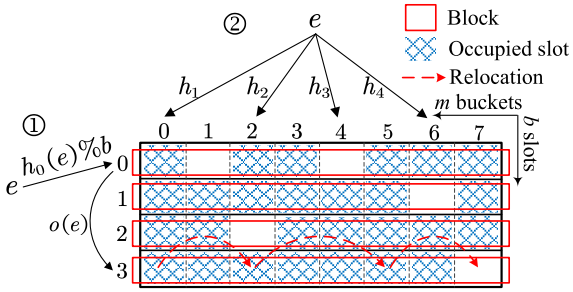


Fig. 2. An illustrative example of SFS. Note that SFS applies the shifting framework in vertical direction, thus $B = b$.

A. The Shifting Framework

We envision a design of probabilistic data structure that can adequately record both the membership and auxiliary information of a set element. We can use a fingerprint in the proper location for a given element to record its membership information, just like CF. However, the challenging issue with such a design is how to represent its auxiliary information with as little space as possible. As stated in ShBF, an intuitive insight is to exploit a shifting framework and encode auxiliary information in the offset. With this insight, below, we show how to realize our idea based on a hash table that uses a slot as the basic unit.

As shown in Fig. 1, we first partition a hash table into multiple identical blocks, e.g., B blocks, and each block consists of consecutive slots. To store an element e with its fingerprint $f(e)$, we select an initial block, say the i^{th} block ($i = 0$ in Fig. 1) through the function $F(e)\%B$, where $F(e)$ can be either a hash function or e's fingerprint $f(e)$. Instead of storing $f(e)$ in this initial location, we store $f(e)$ in the $(i + o(e))\%B^{th}$ block by appending the offset $o(e)$ which is associated with e's auxiliary information. Specifically, e is inserted in its candidate slots in this block by using other hash functions, such as h_1 and h_2. In the query phase, we first locate its initial block through $F(e)\%B$, then search $f(e)$ in its candidate slots in block $F(e)\%B$ and later blocks. If $f(e)$ is found in the j^{th} block, then we conclude that e is existent and decode its auxiliary information through the offset $j - F(e)\%B$. This offset is recorded by the interval between the initial block and the current block. Otherwise, if $f(e)$ is not found in any block, we say that e is nonexistent. To delete the inserted e, we simply remove $f(e)$ from the candidate slots in the block where it is found.

Note that we set the offset to a single direction. In other words, the shifting framework either always increases or decreases $o(e)$ based on the initial block position $F(e)\%B$. This paper uses the increment direction for convenience. Note that B has to be greater than the maximum value of $o(e)$. This limitation can be fixed by adding extra bits into each slot. Specifically, we can introduce counting bits into each slot to achieve smaller $o(e)$ by assisting in encoding the multiplicity information. For instance, we can use $o(e)+c\times B$ to represent the multiplicity $c(e)$ of an element e via the counting field c, thus we have $o(e) = c(e) - c \times B$. However, the offset $o(e)$ must be equal to $c(e)$ without c. In Section VI, we describe in detail how to use counting bits to assist in recording the multiplicity of elements.

Consider the following example: the data structure represents a multi-set S, and the maximum number of occurrences an element can appear in S is 1024. For the multiplicity query of an element, regardless of whether it is in S or not, ShBF musts read at least 1024 bits and further check k candidate bits for each returned 1 bit. However, in the case of adding 5 counting bits into each slot and setting B as 32, our design has to access only 32 blocks and has far fewer memory accesses than ShBF. The fewer memory accesses, the higher the query throughput. Moreover, the fewer the number of checks, the higher the query accuracy. Overall, Our method enables ultimate design flexibility with the shifting framework tailor-made for hash tables. Rather than directly encode excessive multiplicities in a long offset on the bit vector like ShBF, which may be time-prohibitive with severe performance degradation, our methods bound the offset range to guarantee limited access range, fast query response and high query accuracy. Moreover, our approach can easily implement element deletion via the fingerprint field in the hash table.

The above shifting framework is generic and can be applied to all filters which construct a hash table in a matrix form, such as CF and Count-Min Sketch [39]. Moreover, our method can obtain customized performances by partitioning the hash table differently.

B. Shifting Filter on Slot

Here we apply the shifting framework upon a hash table vertically. Specifically, we pack the slots that locate at the same position of different buckets as one block, as shown in Fig. 2. By doing so, the space efficiency of the filter will degrade, which will be discussed later. In order to migrate this deficiency, we design our method based on space-friendly VCF, abbreviating to SFS.

We start with an example of a naive SFS using Fig. 2. This SFS keeps 8 buckets, and each bucket has 4 slots. To insert a given element e, SFS first calculates its fingerprint $f(e)$ and four candidate buckets (i.e., bucket 0, 2, 4 and 6) like VCF. Then the final storage location of e is derived as the slots $(h_0(e)\%b + o(e))\%b$ in four candidate buckets, where h_0 is another hash function. After that, $f(e)$ is stored if one of them is empty. Otherwise, the SFS performs the eviction process. Like CF, SFS randomly chooses one of the candidate buckets, say bucket 0, and evicts the element in slot 3. Then SFS reinserts this victim element to its alternate location, i.e., slot 3 in bucket 2. However, another relocation will be triggered because the alternate location is occupied. This relocation procedure continues until an empty slot is found or the times of such relocations reach the predefined threshold. Once the eviction times reach the threshold, the SFS is considered too full to insert more elements.

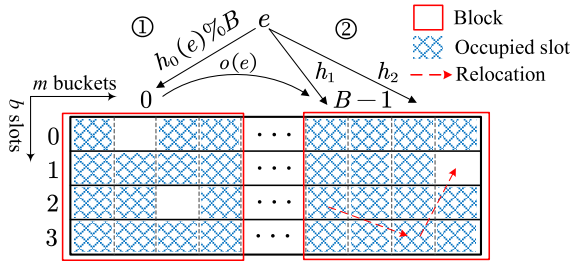


Fig. 3. An illustrative example of SFB. Note that SFB applies the shifting framework in horizontal direction.

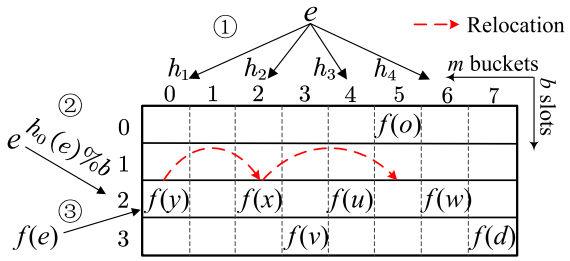


Fig. 4. A toy example of SFS_M, which is designed based on SFS.

To query element e, SFS checks whether $f(e)$ is stored in any of the four candidate buckets h_1, h_2, h_3 and h_4. Suppose that $f(e)$ is found, SFS returns a positive result for the existence information of e. Whereafter, SFS calculates the auxiliary information of e according to its initial block $h_0(e)\%b$ and the current block. Otherwise, SFS concludes that e is non-existent and returns no auxiliary information. The deletion phase of SFS is similar to that of CF. SFS first checks all slots in all candidate buckets. If $f(e)$ is found, that fingerprint would be removed from the corresponding slot. Otherwise, SFS declares that e is non-existent.

C. Shifting Filter on Buckets

Here we implement the shifting framework upon hash table horizontally, with the ambition of more space-saving. Such implementation is named as Shifting Filter on Bucket (SFB).

As shown in Fig. 3, SFB splits the entire hash table of the standard Cuckoo filter into B identical blocks, and each block contains multiple continuous buckets whose amount is a power of two. Like SFS, SFB first locates the target block to insert a given element e by $h_0(e)\%B + o(e)$, where B is the total number of blocks. Then SFS calculates two candidate buckets h_1 and h_2 (where $h_i \in [0, m/B)$, $i = 1$ *or* 2) in the target block through partial cuckoo hash. For convenience, we can set m/B as an integer. If there is any empty slot, SFB puts the fingerprint of e there; otherwise, it performs relocations in this block to find an appropriate slot.

In the query phase of e, for each $0 \le j \le B-1$, if there is a fingerprint matches $f(e)$ in bucket $h_1 + j \times m/B$ or $h_2 + j \times m/B$, SFB outputs the offset from $h_0(e)\%B$ to j as the auxiliary information value for e. To delete a given element e, SFB first query e, and then delete a copy of matched fingerprint if the query returns a positive result.

D. Analysis of Errors

Like CF and ShBF, SF (including both SFS and SFB) may cause false positive errors during the above query or deletion phase of an alien element e_1 which has not been recorded in the filter before. The reason is that one element, e for instance, stored in e_1's candidate buckets may have the same fingerprint with e_1 due to the potential hash collisions. In this case, SF will return the false existence and multiplicity result for querying e_1. The existence and auxiliary information of represented e will also be damaged after the deletion of e_1. Therefore, an element must have been previously inserted before deleting in SF. This requirement also holds for all other deletion-allowed filters.

Moreover, SF may find a fingerprint matched $f(e)$ which does not belong to e during the above query or deletion phase of e. This scene occurs when more than one element shares two candidate buckets and has the same fingerprint. For example, another element e_2 resides in one of e's candidate buckets and collides on fingerprint with e. Such *multiple positives* problem also exists in ShBF. When querying e, SF will output accurate existence information while multiple possible auxiliary information values. Moreover, SF may accidentally delete the fingerprint of e when performing the deletion on e_2. Such a *false deletion* problem does not affect elements' existence information but impacts auxiliary information. For instance, after the *false deletion* of e, the query of e will still obtain a positive result about existence information, while returning the auxiliary information of e_2 in mistake for that of e. *Multiple positives* problem is the expected false-positive behavior of an approximate set membership data structure, and its probability remains bounded by a tunable false positive rate.

IV. Membership Queries

This section utilizes a shifting framework on slots, i.e., SFS, to design an appropriate filter for membership queries of a dynamic set. We use SFS_M to denote this scheme.

A. SFS_M-Construction Phase

Since membership represents only dealing with the existence information of each element, we set the offset in SFS as 0 to design SFS_M. Fig. 4 demonstrates the construction phase of SFS_M with an example of inserting an element e of set S. First, SFS_M calculates the fingerprint $f(e)$ and locates four candidate buckets h_1, h_2, h_3 and h_4 of e, i.e., bucket 0, 2, 4, and 6 in Fig. 4. Second, since the offset is discarded here, SFS_M performs $p = h_0(e)\%b$ to determine which block that e should be stored in, i.e., slot 2. Third, if any slot p is empty in the candidate buckets, SFS_M stores $f(e)$ there. Otherwise, SFS_M randomly chooses one of the candidate slots, say slot 2 in bucket 0, and evicts the fingerprint $f(y)$ in it. Then SFS_M reinserts this victim to its other alternate slots. In this example, the victim $f(y)$ will trigger another relocation of fingerprint $f(x)$ in bucket 2. Then $f(x)$ will be relocated to its alternative candidate slot, i.e., slot 2 in bucket 5. This allocation procedure continues until an empty slot is available or the times of such relocations reach the predefined threshold. Finally, SFS_M inserts $f(e)$ in slot 2 of bucket 0, which has just been vacated. Note that each insertion in SFS_M only performs fingerprint relocation in the same block. The details are shown in Algorithm 1.

Algorithm 1 Insert (e) in SFS$_{\mathrm{M}}$

$f = f(e) =$ fingerprint(e), $p = h_0(e)\%b$;
$h_1 =$ hash(e), $h_2 = h_1 \oplus$ hash(f) $\wedge\ bm_1$, $h_3 = h_1 \oplus$ hash(f) $\wedge\ bm_2$, $h_4 = h_1 \oplus$ hash(f);
if *slot p in bucket h_1, h_2, h_3 or h_4 is empty* **then**
  add f to slot p of that bucket, **return** Done;
$H_1 =$ randomly pick h_1, h_2, h_3 or h_4;
for *t=0; $t < MAX$; $t++$* **do**
  swap f and the fingerprint in slot p of bucket H_1;
  $H_2 = H_1 \oplus$ hash(f) $\wedge\ bm_1$, $H_3 = H_1 \oplus$ hash(f) $\wedge\ bm_2$, $H_4 = H_1 \oplus$ hash(f);
  if *slot p in bucket H_2, H_3 or H_4 is empty* **then**
    add f to slot p of that bucket, **return** Done;
  $H_1 =$ randomly pick $[H_2]$, $[H_3]$ or $[H_4]$;
Hashtable is considered full, **return** Failure

Algorithm 2 Query (e) in SFS$_{\mathrm{M}}$

$f = f(e) =$ fingerprint(e), $p = h_0(e)\%b$;
$h_1 =$ hash(e), $h_2 = h_1 \oplus$ hash(f) $\wedge\ bm_1$, $h_3 = h_1 \oplus$ hash(f) $\wedge\ bm_2$, $h_4 = h_1 \oplus$ hash(f);
if *slot p in bucket h_1, h_2, h_3 or h_4 has f* **then**
  return True;
return False

B. SFS$_{\mathrm{M}}$-Query Phase

The element query phase of SFS$_{\mathrm{M}}$ is detailed in Algorithm 2. To query an element e, SFS$_{\mathrm{M}}$ first computes its fingerprint, four candidate buckets, and initial block. Then SFS$_{\mathrm{M}}$ reads all candidate slots. If any existing fingerprint matches, SFS$_{\mathrm{M}}$ returns *True*. Otherwise, the filter returns a negative result about the existence information for e. Fig. 4 shows a membership query of whether e is in S. In this case, SFS$_{\mathrm{M}}$ will find $f(e)$ in one of e' candidate slots, i.e., slot 2 in bucket 0, then SFS$_{\mathrm{M}}$ judges $e \in S$. Instead of simply searching the target element in candidate buckets like CF, SFS$_{\mathrm{M}}$ only focuses on four candidate slots during each query phase, thereby obtaining a faster query response than CF.

C. SFS$_{\mathrm{M}}$-Delete Phase

Inheriting from CF, SF supports element deletion without rebuilding the entire filter or aiding by other auxiliary data structures. The deletion process of SFS$_{\mathrm{M}}$ is much simpler as no auxiliary information is represented. Similar to the query phase, SFS$_{\mathrm{M}}$ calculates the fingerprint, candidate buckets and initial block for a given element e at first. Then SFS$_{\mathrm{M}}$ checks all candidate slots. If any slot matches, the fingerprint in that slot will be removed. Taking Fig. 4 as an example, to delete e, SFS$_{\mathrm{M}}$ just needs to clear the slot 2 in bucket 0 where $f(e)$ is located. Otherwise, SFS$_{\mathrm{M}}$ reports that e does not exist.

V. Association Queries

This section proposes a Shifting filter called SFS$_{\mathrm{A}}$ for association queries of dynamic sets and presents the construction, query, and delete phases of SFS$_{\mathrm{A}}$. This section further intuitively compares its performance with ShBF.

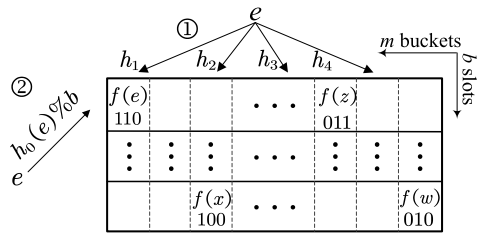


Fig. 5. A toy example of SFS$_{\mathrm{A}}$, where the mark field has three bits to explicitly indicate whether the stored element is a member of S_1, S_2, and S_3 (from right to left), respectively.

A. SFS$_{\mathrm{A}}$-Construction Phase

In order to support association query of dynamic sets more efficiently and accurately, SFS$_{\mathrm{A}}$ extends the similar design concepts of Marked Cuckoo filter to SFS$_{\mathrm{M}}$, as shown in Fig 5. Specifically, SFS$_{\mathrm{A}}$ constructs a cuckoo hash table, where each slot has two fields, including the fingerprint field to represent the element's existence information and the mark field to record the affiliation information of the element. Notice that the number of bits in the mark field equals the number of represented sets, and each bit is initialized to 0. Assuming that there are n elements in total H sets, then SFS$_{\mathrm{A}}$ adds H additional bits in each slot to explicitly label the affiliation(s) of the accommodated fingerprint.

To insert an arbitrary element e in set S_i ($i = 1, \cdots, H$), SFS$_{\mathrm{A}}$ puts its fingerprint into the fingerprint field in one of its candidate slots, as similar as SFS$_{\mathrm{M}}$. After that, SFS$_{\mathrm{A}}$ converts only the i^{th} bit in the mark field to 1. Moreover, if e is also in other sets, the corresponding bit(s) in the mark field will also be set to 1. In the case of Fig 5, to insert an element e which is in set $S_2 \cap S_3$, SFS$_{\mathrm{A}}$ first locates one of e's empty candidates, such as the first slot in bucket h_1. Then SFS$_{\mathrm{A}}$ stores the $f(e)$ in the fingerprint field of this slot and sets the second and third (from right to left) bits in the mark field therein to 1s. Notice that when a victim is kicked out, the 1s in the mark field of that slot will be reset as 0s. Pairwisely, when the victim is reinserted, the corresponding mark field bit(s) in the target slot will be set to 1s. That is, SFS$_{\mathrm{A}}$ bundles the fingerprint and the mark bits together during the fingerprint relocation phase. This guarantees the correctness of the recorded affiliation information.

B. SFS$_{\mathrm{A}}$-Query Phase

The query phase of SFS$_{\mathrm{A}}$ is similar to that of SFS$_{\mathrm{M}}$ in Algorithm 2, to query e with fingerprint $f(e)$, SFS$_{\mathrm{A}}$ just checks its four candidate slots. If any existing fingerprint matches $f(e)$, SFS$_{\mathrm{A}}$ then reads the mark field bits and returns the association information of e. Taking Fig 5 (where $H = 3$) as an example, the mark field of the slot that stores $f(e)$ is 110, explicitly demonstrating that e belongs to both set S_2 and S_3. If $f(e)$ is not found, SFS$_{\mathrm{A}}$ returns *False* to indicate that e is not a member of any set.

C. SFS$_{\mathrm{A}}$-Delete Phase

In addition to eliminating element e from all its affiliates with only one execution, SFS$_{\mathrm{A}}$ also supports deleting e from

a specific set S_i. As an element may belong to multiple sets, there are three cases in the delete phase of SFS_A as follows:

Case 1: the deleted element's fingerprint cannot be found in its candidate slots, or the i^{th} bit in the mark field is 0 when deleting an element from S_i. It means this element is nonexistence or it is not in the target set. In this case, SFS_A returns *False*. For instance, if deleting e from S_1 in Fig 5, SFS_A will recognize the 0 on the first mark field bit and output *False* to declare that e is not in S_1.

Case 2: deleting a stored element e from S_i and the i^{th} bit in the mark field is 1. In this case, SFS_A resets the i^{th} bit to 0 to indicate that e does not belong to S_i anymore. Moreover, if all bits in the mark field are 0s after this deletion operation, SFS_A further removes the fingerprint in this slot. For example, to delete the element x from S_3 in Fig 5, SFS_A sets the third bit to 0 and then clears the fingerprint field.

Case 3: deleting a stored element e from all of its affiliates. In this case, SFS_A simply tries to remove its fingerprint $f(e)$ and reset all 1s in the mark field to eliminate e from all affiliates. In Fig 5, to eliminate e, SFS_A first locates the candidate slot that $f(e)$ is in, i.e., the first slot in bucket h_1. Then SFS_A deletes $f(e)$ from this slot and sets the second and third 1s in the mark field therein to 0s.

D. Comparing SFS_A With ShBF

Compared with ShBF, SFS_A performs better in the following three aspects: First, as ShBF utilizes different offsets to identify the affiliation of elements, ShBF must obtain the attribution information of each element before storing it. So ShBF has to build one sketch (eg, Bloom filter or Cuckoo filter) for each set to obtain this apriori knowledge, incurring substantial space overhead. However, SFS_A can represent the affiliation of elements directly through the mark field without additional data structure.

Second, ShBF does not support the deletion of elements unless it reconstructs the whole data structure or replaces each bit with a counter. These two strategies, however, are substantially more complex than SFS_A and do not appear amenable to a high performance implementation for dynamic sets test. Fortunately, SFS_A not only supports deletion but also can easily update the affiliation of one stored element through changing the mark field bits.

Third, a fatal shortcoming of ShBF is its performance. Its accuracy, false positive rates and query throughput will enormously deteriorate when there are excessive sets to represent. This is because the number of possible results of the element attribution relationship increases exponentially with the number of sets to represent. Assuming that there are in total H sets, the affiliation of an element has 2^H possible outcomes. In this case, ShBF has to calculate $k+2^H$ hash functions and check corresponding bits to respond to an association query. Thus its query time is $O(2^H)$, with a sharp increment in FPR and a sudden decline in accuracy. SFS_A declines the query overhead to $O(1)$ by checking only H mark field bits. As the affiliation of elements is explicitly represented by mark field bits, the accuracy of SFS_A is nearly 100%. That is, SFS_A can always correctly identify which set(s) contain a given element.

VI. Multiplicity Queries

This section proposes two variants of Shifting filters for multiplicity query based on SFS and SFB. We denote them as SFS_X and SFB_X. This section further details the construction, query and delete phases of SFS_X and SFB_X, and analytically compare their performance with ShBF.

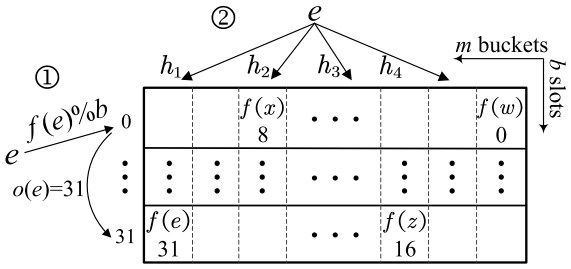


Fig. 6. A toy example of SFS_X, which is designed based on SFS.

A. SFS_X

In order to alleviate the rapid expansion of the search range caused by the excessive element frequency, SFS_X introduces extra bits into each slot to assist in recording the multiplicity information of the element, as shown in Fig. 6. Specifically, SFS_X introduces a count field into each slot based on SFS in Fig. 2, and each bit in the count field is initialized to 0.

Since SFS_X is evolved from SFS, the construction, query and delete phases of them are the same, except for the following two aspects. First, SFS_X uses $f(e)\%b$ to obtain the initial block, instead of $h_0(e)\%b$ mentioned in Section III. This helps SFS_X to reduce a hash calculation operation, thereby reducing the computational complexity. Second, SFS_X utilizes the offset and the count field together to represent the multiplicity information of the element.

SFS_X-Construction phase. Algorithm 3 details the construction phase of SFS_X to insert $[e, c(e)]$, where e is a given element and $c(e)$ is the multiplicity information of e. Like SFS, to insert an element e that appears $c(e)$ times, SFS_X performs $(f(e)\%b + (c(e)-1)\%b)\%b$ to determine the final block that contains candidate slots, where $f(e)\%b$ is the initial position and $(c(e)-1)\%b$ is the offset. To store the existence information of e, SFS_X records $f(e)$ with the fingerprint field of one empty or cleared candidate slot. To store the auxiliary information of e, which in this case is $c(e)$ of element e in S, SFS_X increases the count field in slot where $f(e)$ located by $\lfloor (c(e)-1)/b \rfloor$. A toy example is depicted in Fig. 6, wherein SFS_X sets $b = 32$ and inserts the element e with $c(e) = 1024$. In this case, SFS_X first calculates that the initial position of e is the block 0 where slot 0 is located. Then SFS_X calculates the offset $o(e) = (c(e)-1)\%b = 31$ and further locates the final position, i.e., block 31. After calculating e's four candidate buckets, SFS_X can lock e's four candidate slots, i.e., slots 31 in bucket h_1, h_2, h_3, and h_4. SFS_X then randomly picks one of these empty slots, such as slot 31 in bucket h_1, stores e's fingerprint $f(e)$ in its fingerprint field and sets its counting field as $\lfloor (c(e)-1)/b \rfloor = 31$. In our designs, the auxiliary information of e is encoded in both the offset and the count field. Like SFS_A, SFS_X also bundles the fingerprint and the count field together when relocating the overflowed fingerprint during the relocation phase.

SFS_X-Query phase. The query phase of SFS_X is illustrated in Algorithm 4. To query e, SFS_X first explores all e's candidate buckets, i.e., bucket h_1, h_2, h_3, and h_4 in Fig 6. If the fingerprint field of any slot matches with $f(e)$, SFS_X

Algorithm 3 Insert $(e, c(e))$ in SFS$_X$

f = $f(e)$ = fingerprint(e), c = $\lfloor (c(e)-1)/b \rfloor$;
p' = $(f\%b + (c(e)-1)\%b)\%b$;
h_1 = hash(e), h_2 = $h_1 \oplus$ hash(f) $\wedge$ bm_1, h_3 = $h_1 \oplus$ hash(f) $\wedge$ bm_2, h_4 = $h_1 \oplus$ hash(f);
if *slot p' in bucket h_1, h_2, h_3 or h_4 is empty* **then**
  add $[f, c]$ to slot p' of that bucket, **return** Done;
H_1 = randomly pick h_1, h_2, h_3 or h_4;
for *t=0; $t < MAX$; $t++$* **do**
  swap $[f, c]$ and the fingerprint in slot p' of H_1;
  H_2 = $H_1 \oplus$ hash(f) $\wedge$ bm_1, H_3 = $H_1 \oplus$ hash(f) $\wedge$ bm_2, H_4 = $H_1 \oplus$ hash(f);
  if *slot p' in bucket H_2, H_3 or H_4 is empty* **then**
    add $[f, c]$ to slot p' of that bucket;
    return Done;
  H_1 = randomly pick $[H_2]$, $[H_3]$ or $[H_4]$;
// Hashtable is considered full, **return** Failure

Algorithm 4 Query (e) in SFS$_X$

f = $f(e)$ = fingerprint(e), p = $f\%b$;
h_1 = hash(e), h_2 = $h_1 \oplus$ hash(f) $\wedge$ bm_1, h_3 = $h_1 \oplus$ hash(f) $\wedge$ bm_2, h_4 = $h_1 \oplus$ hash(f);
for *t=0; $t < b$; $t++$* **do**
  if *slot t in bucket h_1, h_2, h_3 or h_4 has f* **then**
    if $t \geq p$ **then**
      $\Delta = t - p + 1$
    else if then
      $\Delta = t - p + b + 1$
    $c(e) = \Delta + c \times b$, where c is the value of the count field in that slot;
    return $c(e)$;
return False

indicates that e is in set S. Further, SFS$_X$ reads the count field c of that slot and then calculates $\Delta + c \times b$ as the multiplicity of an element e, where Δ is the offset from the p^{th} block to the p'^{th} block. In the case of Fig 6, for the multiplicity query of e, SFS$_X$ will calculate Δ as $31 - 0 + 1 = 32$ and tell the multiplicity of e as $32 + 31 \times 32 = 1024$.

SFS$_X$-Delete phase. The delete phase of SFS$_X$ is much simpler than the query phase. To delete e, SFS$_X$ searches its fingerprint $f(e)$ in all candidate buckets. If any slot matches, SFS$_X$ clears both the fingerprint field and the count field of this slot. Otherwise, SFS$_X$ returns *False* to indicate that e is not in set S. In Fig 6, to delete e, SFS$_X$ checks its four candidate buckets and locates the slot that $f(e)$ is in, i.e., slot 31 in bucket h_1. Then SFS$_X$ removes $f(e)$ from this slot and sets the count field therein to 0.

B. SFB$_X$

Like SFS$_X$, SFB$_X$ also extends each slot by adding a count field. Together with the offset on buckets represents multiplicity information. Besides, SFB$_X$ also replaces the independent hash function $h_0(e)$ of SFB in Fig. 3 with $f(e)$.

SFB$_X$-Construction phase. To insert a given element e with multiplicity $c(e)$, SFB$_X$ first computes $(f(e)\%B+(c(e)-1)\%B)\%B$ to locate the candidate block, then inserts the fingerprint $f(e)$ to e's two candidate buckets and set the corresponding count bits to $\lfloor (c(e)-1)/B \rfloor$. The counter in SFB$_X$ evicts along with the fingerprint.

SFB$_X$-Query phase. In the query phase of e, SFB$_X$ explores each slot from the candidate buckets in the first block to last one, until there is a fingerprint matches $f(e)$ or all blocks are traversed. Assuming the i^{th} $(0 \leq i \leq B-1)$ block has fingerprint $f(e)$, SFB$_X$ calculates $\Delta + c \times B$ as e's multiplicity, where c is the counter's value of corresponding slot and Δ is the offset from block $f(e)\%B$ to block i. If there is no $f(e)$ in all candidate buckets, SFB$_X$ returns *False*.

SFB$_X$-Delete phase. The deletion process of SFB$_X$ is the same as that of SFB, except with one more step to clear the count field in the matched slot.

C. Comparison With ShBF

Our Shifting filters for multiplicity query, including SFS$_X$ and SFB$_X$, are superior to ShBF in the following four ways. 1) as mentioned in Section V, the element deletion in SFS$_X$ and SFB$_X$ are much simpler than ShBF. 2) ShBF needs a priori knowledge of the most frequent element. However, this is unnecessary for our solutions. 3) ShBF suffers from severe degradation in terms of accuracy, false positive rates and query throughput when representing elements with skewed frequencies. However, SFS$_X$ and SFB$_X$ can remain a high performance consistently. 4) To alleviate the problem of returning multiple possible results due to hash conflicts, ShBF must traverse all bits in the entire search range and return the maximum value of all matching results. Such design is highly time-consuming, especially when for an extensive search range. By contrast, SFS$_X$ and SFB$_X$ rely on fingerprints to mitigate hash conflicts. The probability of false matching caused by fingerprint conflicts is low. Therefore, in the query phase, SFS$_X$ and SFB$_X$ can immediately end the search and return the corresponding result as long as there is a fingerprint that matches $f(e)$.

VII. Theoretical Analysis

This section launches a theoretical analysis of all Shifting filter variants in terms of insertion failure probability, false positive rate, and space efficiency.

A. Insertion Failure Probability

SFS-analysis. Since the sequence index of slots has been used to represent elements' auxiliary information, SFS relocates the overflow fingerprints only in the location of one fixed slot in each insertion phase. Such design incurs a degree of space efficiency degradation on the data structure. In order to explore the impact of these restrictions on the space utilization of filters, we derive a lower bound of the insertion failure probability below.

Let us first derive the probability that two distinct elements collide in the same four buckets. This situation may arise

when they: 1) have the same fingerprint, which occurs with probability $1/2^l$; 2) have the first candidate bucket as h_1, h_2, h_3 or h_4, which occurs with probability $4/m$. Thus the probability of a given set of q items sharing the same four candidate buckets is $(4/m{\cdot}1/2^l)^{q-1}$. When inserting n random elements to an empty table of $m = \gamma n$ buckets for a constant γ, as long as there are $q = 5$ elements mapped into the same four candidate slots, the insertion will fail. This probability provides a lower bound for insertion failure of SFS. Since there are in total $\binom{n}{5}$ different possible sets of 5 elements out of n elements, the expected number of 5 elements colliding during the construction phase of SFS is

$$\binom{n}{5}\left(\frac{4}{2^l{\cdot}m}\right)^4 = \binom{n}{5}\left(\frac{4}{2^l{\cdot}\gamma n}\right)^4 = \Omega\left(\frac{n}{\gamma^4 \cdot 2^{4l-1}}\right) \quad (1)$$

It is obvious that $\gamma^4 \cdot 2^{4l-1}$ must be $\Omega(n)$ to avoid a non-trivial probability of insertion failure, as otherwise this expectation is $\Omega(1)$. Accordingly, the minimum number of bits required for the fingerprint in SFS must be

$$l_{\mathrm{SFS}} = \lceil \log_2 2n/4 - \log_2 \gamma \rceil \quad (2)$$

This result seems somewhat disadvantageous, as the lower bound for fingerprint size is $\Omega(\log n)$ in SFS. At the same time, such indicator in the standard Cuckoo filter is $\Omega(\log n/b)$, which can be adjusted by the b factor in the denominator. This is because SFS leverages slot position to represent more information, thereby cannot store elements at will, incurring an inferior space efficiency. However, in actual use, the fingerprint length is jointly decided by both the space budget and the target false positive rate. Therefore, this trade-off should be handled by users according to their performance requirements.

SFB-analysis. As SFB splits the hash table into multiple blocks and restricts the two candidate buckets in one block, the probability of two distinct elements have their first candidate bucket as h_1 or h_2 is $2B/m$. Therefore, the expected number of groups of $2b+1$ elements colliding during the construction phase of SFB is

$$\binom{n}{2b+1}\left(\frac{2B}{2^l \cdot m}\right)^{2b} = \Omega\left(\frac{n}{(\gamma \cdot 2^l/B)^{2b}}\right) \quad (3)$$

In this case, the minimum number of bits required for the fingerprint in SFB must be

$$l_{\mathrm{SFB}} = \left\lceil \log_2 n/2b + \log_2 \frac{B}{\gamma} \right\rceil \quad (4)$$

Equ. 4 indicates that the fingerprint size in SFB can be saved by the b factor in the denominator of the lower bound. In other words, as long as SFB uses reasonably sized buckets, its fingerprint size can remain small, which is helpful for implementation with small memory.

B. False Positive Rate

For membership query, false positive error can be intuitively considered as a filter returning a positive result when querying a non-existent element. Here we extend the definition of false positive error to the association and multiplicity query. Specifically, false positive error may arise when a filter regards an alien element as a member of any represented set in the association query phase, or the output multiplicity of this element is larger than 0 in the multiplicity query phase. The probability of such error is the false positive rate (FPR).

SFS-analysis. When looking up a non-existent element e_1 in a slot, if this slot is occupied, the probability that e_1 is matched against the stored fingerprint is $1/2^l$, otherwise 0. In the worst case to query e_1, SFS must probe all four candidate buckets each with b filled slots, thus the probability of this query returning a false positive successful match is

$$\mathrm{FPR}_{\mathrm{SFS}} = 1 - (1 - 1/2^l)^{4b} \approx 4b/2^l \quad (5)$$

Equ. 5 indicates that FPR is inversely proportional to the fingerprint length l and is positively associated with bucket size b, and also reflects an upper bound of the total probability of a false-positive fingerprint hit. To obtain the target false positive rate ϵ, SFS must guarantee $4b/2^l \leq \epsilon$, thus the minimum fingerprint length can be calculated as

$$l_{\mathrm{SFS}} \geq \lceil \log_2 (1/\epsilon) + \log_2 (4b) \rceil \quad (6)$$

The query phase in $\mathrm{SFS_M}$ and $\mathrm{SFS_A}$ works differently from that in $\mathrm{SFS_X}$. $\mathrm{SFS_M}$ and $\mathrm{SFS_A}$ only check the fingerprints at one slot rather than all slots in candidate buckets, and thus have a lower chance to incur fingerprint collisions. Hence the FPR of $\mathrm{SFS_M}$ and $\mathrm{SFS_A}$ is

$$\mathrm{FPR}_{\mathrm{SFS'}} = 1 - (1 - 1/2^l)^4 \approx 1/2^{l-2} \quad (7)$$

Thus the minimum fingerprint size of $\mathrm{SFS_M}$ and $\mathrm{SFS_A}$ is

$$l_{\mathrm{SFS'}} \geq \lceil \log_2 (1/\epsilon) + 2 \rceil \quad (8)$$

SFB-analysis. As SFB has to check two candidate buckets in all blocks for any given element, its search scope is $2bB$ slots, and thus we have

$$\mathrm{FPR}_{\mathrm{SFB}} = 1 - (1 - 1/2^l)^{2bB} \approx 2bB/2^l \quad (9)$$

The above result indicates that the FPR of SFB is also positively associated with block number B besides bucket size b. Thus the minimum fingerprint length of SFB is

$$l_{\mathrm{SFB}} \geq \lceil \log_2 (1/\epsilon) + \log_2 (2bB) \rceil \quad (10)$$

C. Space Efficiency

After obtaining the minimum fingerprint size that appears amenable to both insertion failure probability and FPR requirements, we can measure the space efficiency of these filters, through the average bits per element. Assume a filter with m buckets each of which has b slots. Each slot records the fingerprint with l bits. The total bits of the filter is thus mbl. After inserting elements to this empty table, let α be the load factor, which means the ratio between the maximum size of the filter and its capacity when the first insertion failure occurs. Therefore, there are $mb\alpha$ elements represented by this table/filter. Thus the bits per element (also the amortized space cost) of this filter is

$$\mathrm{BPE} = \frac{mbl}{mb\alpha} = l/\alpha \quad (11)$$

where the value of l should not be smaller than the lower bound determined by Equ. 2 and 6 for SFS, Equ. 2 and 8 for $\mathrm{SFS_M}$, Equ. 4 and 10 for SFB. Moreover, when it comes to

filters with auxiliary fields, such as mark fields in SFS_A and count fields in SFS_X, Equ. 11 should be adjusted as

$$\text{BPE} = \frac{mb(l+l')}{mb\alpha} = (l+l')/\alpha \tag{12}$$

where l' is the amount of bits used in each auxiliary field.

D. Precision

Due to hash conflicts, filters' association and multiplicity query results may be inconsistent with the actual values. Specifically, such an error may arise when filters find a fingerprint that conflicts with the target fingerprint before reading the correct auxiliary information. In this case, filters will mistake another distinct element with different auxiliary information as the queried one. Therefore, the probability of returning an accurate result is paramount for evaluating the results they output. Next we deduce the lower bound of the precision of SFS_A, SFS_X and SFB_X.

SFS_A-analysis. SFS_A uses the mark field to record the affiliation information of elements explicitly. In other words, the probability of returning an accurate result is 100% as long as there is no false positive error. Therefore, the lower bound of the precision of SFS_A is

$$P_{\text{SFS}_\text{A}} = 1 - \text{FPR}_{\text{SFS}'} = 1 - 1/2^{l-2} \tag{13}$$

SFS_X-analysis. As shown in Algorithm 4, to query an element e for its multiplicity information, SFS_X searches $f(e)$ from the first four slots to the last ones in e's four candidate buckets until there is one fingerprint matching or all candidate slots are checked. Assuming that $f(e)$ is stored in the i^{th} ($i \in \{1, 2, \cdots, b\}$) four candidate slots, the probability of a false fingerprint match before reading $f(e)$ is at most $1 - \left(1 - 1/2^l\right)^{4i}$ in SFS_X. Since an arbitrary element is assigned to each candidate slot with equal probability, the average precision of SFS_X is

$$P_{\text{SFS}_\text{X}} = \sum_{i=1}^{b} \left(1 - 1/2^l\right)^{4i} / b \tag{14}$$

SFB_X-analysis. Like SFS_X, SFB_X also successively probes two candidate buckets from the first block to the last one to query a given element. As a consequence, the probability of SFB_X falsely matching a fingerprint before the correct slot is at most $1 - \left(1 - 1/2^l\right)^{2bi}$, where i represents the sequence number of the block storing the target element. The probability of an element being inserted into each block is also equal in SFB_X, so the average precision of SFB_X is

$$P_{\text{SFB}_\text{X}} = \sum_{i=1}^{B} \left(1 - 1/2^l\right)^{2bi} / B \tag{15}$$

E. Comparison With ShBF

ShBF-analysis. Recall that for a set of n elements, the FPR of ShBF for membership query is

$$\text{FPR}_{\text{ShBF}_\text{M}} \approx (1-p)^{\frac{k}{2}} \left(1 - p + \frac{1}{\bar{w}-1} p^2\right)^{\frac{k}{2}} \tag{16}$$

where $p = e^{\frac{-nk}{m}}$, k is the hash function number for each element, m is the bit vector length of ShBF and $\bar{w}$ is a function of machine word size. ShBF further optimizes system parameters to minimize $\text{FPR}_{\text{ShBF}_\text{M}}$ [7], it indicates that the optimum value for $\bar{w}$ is $\bar{w}_{\text{opt}} = 57$ for 64−bit architecture, and the optimum value for k is $k_{\text{opt}} = 0.7009\frac{m}{n}$. Substituting the value of $\bar{w}_{\text{opt}}$ and k_{opt} into Equ. 16, the minimum value of $\text{FPR}_{\text{ShBF}_\text{M}}$ is given by the following equation.

$$\text{FPR}^{\min}_{\text{ShBF}_\text{M}} = 0.6204^{\frac{m}{n}} \tag{17}$$

Recall that the base of BF in Equ. 17 is 0.6185 [7], which indicates that the FPRs of ShBF_M and BF are almost the same. For the association query of an element with I possible affiliations, or the multiplicity query of an element in a multi-set S_m that the maximum number of times an element can occur is L, the FPR of ShBF is adjusted as

$$\text{FPR}^{\min}_{\text{ShBF}_{\text{A or X}}} = 1 - (1 - 0.6185^{\frac{m}{n}})^Z \tag{18}$$

where $Z = I - 1$ ($Z = L$) for association (multiplicity) query.

When querying an element with multiplicity J ($1 \le J \le L$) in S_m, the precision of ShBF_X, i.e., the probability of this element is correctly reported to be present J times is

$$P_{\text{ShBF}_\text{X}} = (1 - 0.6185^{\frac{m}{n}})^{L-J} \tag{19}$$

FPR Comparison. Let us compare the FPR of SFS_M and ShBF_M at the same space overhead. For membership query, assume that to represent a set with $n = 0.95 \times 2^{20}$ elements, SFS_M maintains 2^{18} buckets with $b = 4$ and $l = 18$. Thus the space overhead of SFS_M is 18×2^{20} bits, which is also the value of m in ShBF_M. Substituting these parameters value above into Equ. 17 and Equ. 7, the FPRs of ShBF_M and SFS_M are 1.32×10^{-4} and 1.5×10^{-5}, respectively. For association query, we further assume that the affiliation number is $I = 8$ and each mark field in SFS_A uses 3 bits. Then the space overhead is 21×2^{20} bits, we have $\text{FPR}_{\text{ShBF}_\text{A}} = 1.71 \times 10^{-4}$ by substituting $m = 21 \times 2^{20}$ into Equ. 18, while $\text{FPR}_{\text{SFS}_\text{A}}$ is still equal to 1.5×10^{-5}. For multiplicity query, we further assume that $L = 1024$ and each count field in SFS_X uses 5 bits, so we have $\text{FPR}_{\text{ShBF}_\text{A}} = 9.05 \times 10^{-3}$ and $\text{FPR}_{\text{SFS}_\text{X}} = 4.88 \times 10^{-4}$ through Equ. 18 and Equ. 5, respectively. The above results indicate that compared to ShBF, SFS reduces the FPR by up to one order of magnitude for various queries.

Precision Comparison. To compare the worst precision of SFS_X and ShBF_X at the same space overhead, we further set the $l = 16$ in the case of the FPR analysis for multiplicity query above. According to Equ. 19, we find that ShBF_X achieves the worst precision when J has the minimum value. Therefore, substituting $m = (16+5) \times 2^{20}$ and $J = 1$ into Equ. 19, we have $P^{\min}_{\text{ShBF}_\text{X}} = 0.9753$. Since SFS_X searches target fingerprint step by step, it has the worst precision when target locates at the last slot. With this insight, we can calculate that $P^{\min}_{\text{SFS}_\text{X}} = (1 - 1/2^l)^{4b-1} = 0.9981$. Apparently SFS performs better than ShBF in terms of precision.

VIII. Evaluation

A. Experiment Setup

Platform: All experiments are conducted in a machine with an Intel Core i7 processor and 16GB DRAM. All codes are available at Github (https://github.com/fptjy/Shifting-filter-framework).

Dataset: We use synthetic and real-world datasets to make the experimental results more statistically significant and convincing. Specifically, for membership queries, we randomly generate numerous strings of sufficient length, and such strings constitute a synthetic dataset for testing. For association queries, we generate synthetic datasets with different affiliations under the circumstances of two, three, and four subsets. For multiplicity queries, we first generate synthetic datasets wherein the data multiplicity follows a normal distribution with different expectations u and standard deviations δ ($u = 2^i, \delta = i$ where i varies from 5 to 10), because the normal distribution is one of the most common distributions in statistics. Furthermore, we use five public traffic traces from MAWI [40] with the 5-tuple as the flow ID. These real-world datasets contain 2.3 to 4.2 million network flows, where the distinct flows' number varies from 19k to 31k, and the multiplicity of the top-1 flow, i.e., c, varies from 49376 to 86069. Unlike the synthetic datasets above, the multiplicity of network flows in traffic traces is highly skewed and has a heavy-tailed distribution.

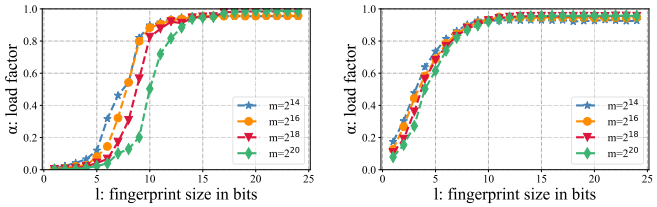
Parameter setting: All Shifting filters maintain $m = 2^{18}$ and $b = 4$, except for SFS_X ($m = 2^{15}$, $b = 2^5$). For traffic traces tests, the value of m in SFS_X (SFB_X) is tuned as 2^{10} (2^{13}). The l is 18 in SFS_M and 16 in other Shifting filters. All Shifting filters set $MAX = 500$. ShBF configures the value of k through $k_{\text{opt}} = 0.7009\frac{m}{n}$. For the membership query experiments, we maintain the overall false positive rate of the filters as 1.5×10^{-5}, and the space occupancy of filters varies from 5% to 95%. For the association and multiplicity query experiments, the competitors maintain the same space overhead with Shifting filters, except for NA ($m = 2^{13}$, $b = 2^2$, $l = 16$), which consumes 18.5% more space due to its bigger counters. The experiments for association and multiplicity queries of synthetic datasets are conducted under 95% space occupancy. In other words, the data set fed to a filter has a size equal to 0.95 times the capacity of this filter, and this numeric is close to the load factor of both Cuckoo and Shifting filters. For multiplicity queries of traffic traces, as flows with different sizes are inserted into filters whose capacity is fixed, the space occupancy of these filters varies from 59.7% to 95.5%. Elastic Sketch sets the predefined threshold of the rate between negative votes and positive votes as eight and utilizes three arrays in its light part. All Shifting filters utilize Python built-in hash functions and DJB hash functions.

B. Metrics

Throughput: We perform insertion, query and deletion operations of all elements, and record the total time used. Then the throughput is defined as $\frac{N}{T}$, where N is the total number of operations, and T is the total measured time.

Precision: Precision is defined as $\frac{\varphi}{N}$, where N is the number of queries among which φ queries are correctly responded.

Average Relative Error (ARE): ARE is defined as $\frac{1}{|\Psi|}\sum_{e_i \in \Psi}\frac{|\hat{x}_i - x_i|}{x_i}$, where Ψ is the set consists of queried elements, $\hat{x}_i$ is the estimated multiplicity of element e_i, whose real multiplicity is x_i. ARE can evaluate the error rate of the element multiplicity estimated by the filter.



(a) SFS. (b) SFB.

Fig. 7. Load factor α achieved of SFS and SFB.

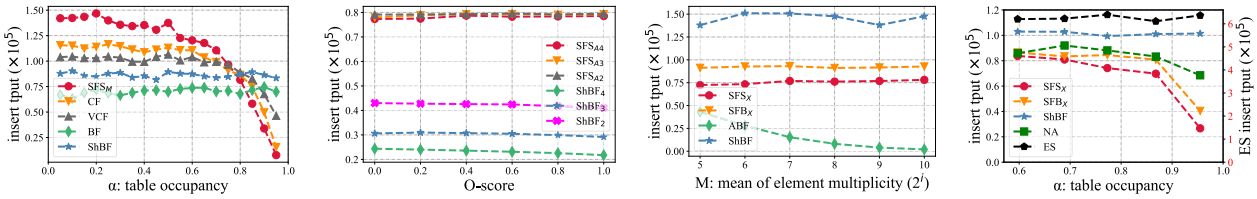
Overlap score (O-score): Overlap score is an indicator defined by us to reflect the overlap degree of several subsets in an universe. If there are total Z target subsets, the proportion of elements belonging to i target subsets at the same time is y_i. Then O-score of this universe can be defined as $\sum_{i=0}^{Z} i \times y_i / Z$. For example, there are two target subsets S_1, S_2 and an universe U. The proportion of element which belongs to $U - S_1 \cup S_2$, $S_1 \cup S_2 - S_1 \cap S_2$ and $S_1 \cap S_2$ is 70%, 20% and 10%, respectively. So the O-score of U is 0.2. Note that the above formula is adjusted as $\sum_{i=1}^{Z}(i-1) \times y_i / Z$ when it comes to constructing insertion datasets with different O-score. Aiming at association query, O-score is utilized to explore the performance of filters under different distributions of element affiliation.

Consider the following example: ShBF performs an association query of an element e after constructing through two data sets consisting of S_1 and S_2, one with an O-score of 0.5 and the other with an O-score of 0.9. Recall that ShBF would first check if e falls into the first two cases, i.e., $e \in S_1 - S_2$ or $e \in S_2 - S_1$, and ShBF utilizes the offset function $o_2(e) = o_1(e) + h_{k+2}(e)\%((\bar{w} - 1)/2) + 1$ for $e \in S_1 \cap S_2$. Consequently, ShBF has to execute more hash computation and read more bits for those elements in $S_1 \cap S_2$. It follows that ShBF will perform worse when facing the data set whose O-score is 0.9. That is, the more off-center the O-score, the more the overhead time for the association query of those parametric data structures. With this insight, designing an indicator to analyze how data structures perform when element affiliation distributions vary is interesting and essential.

C. Numerical Results

1) Space Efficiency: Either SFS or SFB supports a high performance query of dynamic sets, with a compromise of space efficiency. We measure the load factor α of SFS and SFB with different fingerprint lengths l, as shown in Fig. 7. We vary l from 1 to 24 for the experiments. Random synthetic elements are inserted into an empty filter until an insertion failure, and then we measure the achieved α. We fix b to 4 for each bucket and set B as 32 for SFB. We run this experiment thirty times for filters with $m = 2^{14}, 2^{16}, 2^{18}$ and 2^{20} buckets, then record their minimum load factors.

As shown in Fig. 7, with sufficiently long fingerprints, SFS and SFB with $b = 4$ realize 98% and 95% occupancy, respectively. SFB is more space-friendly than SFS because it needs shorter fingerprints for high space utilization. As suggested by the theory, the minimum l required by SFS to obtain a low FPR (0.001) is 15 bits, which is also the minimum l required to achieve close-to-optimal occupancy. In effect, SFS has a



(a) Membership. (b) Association. (c) Multiplicity (synthetic). (d) Multiplicity (trace-driven).

Fig. 8. The insertion throughput of different types of query.

better space efficiency for the practical application of high accuracy than its same kinds. When filters remain the same FPR of 1.5×10^{-5}, the BPE of BF, ShBF, CF, VCF, and SFS_M would be configured as 23.11, 24.11, 20, 21.05 and 18.95, respectively. SFS uses the fewest bits for each element and saves that by 21.4%, 18%, 9.98%, and 5.25% than ShBF, BF, VCF, and CF, respectively.

2) Insertion Throughput: Fig. 8(a) indicates the instantaneous element insertion throughput of those filters for membership query when α grows. As depicted in Fig. 8(a), the insertion throughput of BF and ShBF remain constant while the other three filters show dramatic declines when α grows. The reason is that they have to execute the kick-out-and-reallocate process for overflowed buckets or slots when the filters get crowded. Compared to CF and VCF, SFS_M has a better insertion performance when the α is lower than 0.75. Because SFS_M checks fewer candidate slots to find an empty slot for the coming fingerprint. However, as the table gets crowded, the relocation strategy begins to take a major role in diminishing the construction speed, especially for SFS_M with fewer candidate slots. In general, SFS_M's overall construction speed is 1.14×, 1.18×, 1.30× and 1.60× faster compared with CF, VCF, ShBF and BF, respectively.

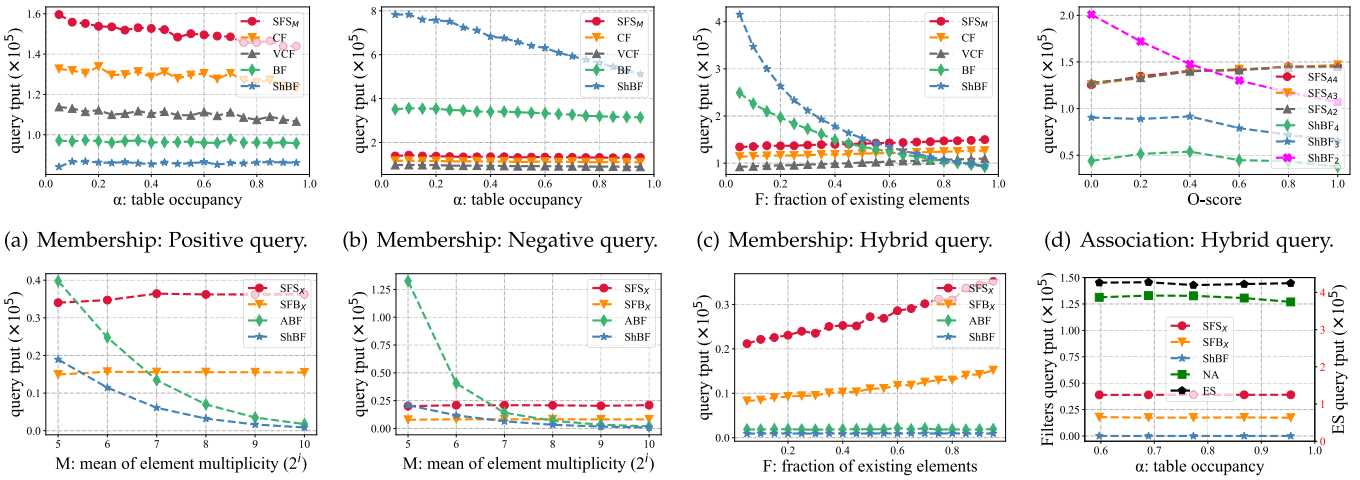
For association query, we vary the O-score and test the overall insert throughput of ShBF and SFS_A as shown in Fig. 8 (b), where the integer subscript in the legend means the number of subsets that the insertion dataset contains. SFS_A ensures a high and almost constant insert throughput, while the construction speed of ShBF is much lower and declines with the increments of O-score and the number of affiliations. Because with a higher O-score, ShBF needs more hash calculations in its offset function, consuming more time. Besides, ShBF must explore assistant filters to fetch the affiliations of a given element before inserting it. On the whole, compared with ShBF, SFS_A accelerates the insertion throughput by 3.36×, 2.61×, and 1.88× in the case of 4, 3 and 2 subsets, respectively.

Fig. 8(c) further compares the overall insert throughput of different filters for multiplicity query when another comparison method ABF is adopted. In this experiment, we feed these filters with synthetic datasets, then quantify their overall construct speed with the average multiplicity (i.e., M) increasing from 2^5 to 2^{10}. Like SFS_A, SFS_X and SFB_X also remain a high and almost constant construct speed. Since the element multiplicities are used directly as offsets instead of other hash values, the insertion throughput of ShBF is significantly improved. However, ABF performs the worst and gets worse as multiplicity increases because ABF exploits the extra hash functions whose number is the same as the recorded elements' multiplicity.

For multiplicity query experiments with traffic traces, we have added two new competitors to replace unqualified ABF, i.e., the NA method and Elastic Sketch (ES) [17]. As shown in Fig. 8(d), as the input traffic traces size increases, the data structures get crowded, and the α also grows, the insertion throughput of ShBF and ES remain constant, while CF variants, including SFS_X, SFB_X and NA show a degree of decline. Although ES is fast to insert, it records the network flows arriving one by one, substantially different from the working mechanism of those data structures that can directly record auxiliary information. In general, compared with these filters of the same type, SFS_X has a slightly inferior insertion throughput when tracing network flows.

3) Query Throughput: **Membership query.** First, we vary α of these filters from 0 to 95%, and measure their instantaneous query throughput when the queried elements occupy 100% (positive) and 0% (negative) of the represented elements, as shown in Fig. 9(a) and (b). The throughput of CF, VCF and SFS_M is stable across different α on both negative and positive queries. Because the total number of slots to read and compare remains constant irrespective of the filter occupancy. In contrast, BF and ShBF behave differently when serving negative and positive queries. For the positive query, they must always check in total k bits irrespective of filter occupancy, thus providing constant query throughput. SFS_M outperforms others when serving positive queries while it is inferior to ShBF and BF when it comes to negative queries. BF can immediately terminate the next series of hash function calculations and candidate bits access once a zero bit is found. CF and its variants use the fingerprints in the hash table rather than the bit vector to encode elements information. Therefore, SFS_M can only end the query once a fingerprint match or all candidate buckets have been traversed. That explains why ShBF performs better than SFS_M when facing a negative query. Despite compromising negative query throughput due to the working mechanism, SFS_M also gains many advantages, such as element deletion support, higher accuracy, better space efficiency, and good scalability.

Second, we fix α to 95% and vary the fraction of positive elements in the query set, i.e., F from 0.05 to 0.95 for each filter, and then record the overall query throughput with Fig. 9(c). The results indicate that the query throughput of SFS_M increases when F grows, while it decreases substantially for ShBF and BF. Our method outperforms other filters when F is larger than 0.5. The reason is that CF and its variants search fewer slots before finding the target element as F increases. However, BF and ShBF have to check more bits. The throughput of ShBF is nearly double that of BF, irrespective of the types of queries. The reason is that ShBF only used half number of hash functions compared with BF.



(a) Membership: Positive query. (b) Membership: Negative query. (c) Membership: Hybrid query. (d) Association: Hybrid query.

(e) Multiplicity: Positive query. (f) Multiplicity: Negative query. (g) Multiplicity: Hybrid query. (h) Multiplicity:Trace-driven query.

Fig. 9. The query throughput of different types of query.

Association query. Given $\alpha = 0.95$, we vary O-score from 0 to 1 and then measure the overall query throughput of filters. As shown in Fig. 9(d), no matter how many sets the queried element is, SFS_A remains a fast query speed which gets even faster with the growth of O-score. Because SFS_A explicitly represents element affiliations by mark field bits, it can respond faster when more queries are positive. However, the performance of ShBF degrades when the elements may belong to more sets or O-score gets larger. This is because ShBF must compute additional hash functions and read more bits. In brief, when serving queries with 2, 3 and 4 subsets, SFS_A has 0.95×, 1.71× and 3.03× overall query throughput compared with ShBF, respectively.

Multiplicity query. Fig. 9(e) and (f) depict filters' query throughput of coping with synthetic datasets with varying M when all queries are negative and positive. SFS_X and SFB_X always ensure a high query throughput irrespective of the increase of M or the type of queries. SFS_X performs better than SFB_X as it fetches fewer slots. However, ShBF and ABF suffer substantial performance degradation when M grows. We also note that ShBF works worse than ABF. In fact, SFS_X and SFB_X always fetch a fixed number of candidate slots or buckets. However, ShBF must check almost $c \times k$ bits to avoid underestimation where c is the maximum multiplicity of all inserted elements. For a positive query with multiplicity i, ABF has to compute additional $i + 1$ hash functions and read $k + i + 1$ bits at least. ABF can return immediately after fetching the first 0 for a negative query. Unfortunately, ABF has to compute more hash functions and check more bits if this negative query incurs a false positive error, which is common when M grows. In short, the negative (positive) query throughput of SFS_X is 2.31, 2.37 and 5.07 (2.54, 0.6 and 2.75) times higher than SFB_X, ABF and ShBF, respectively.

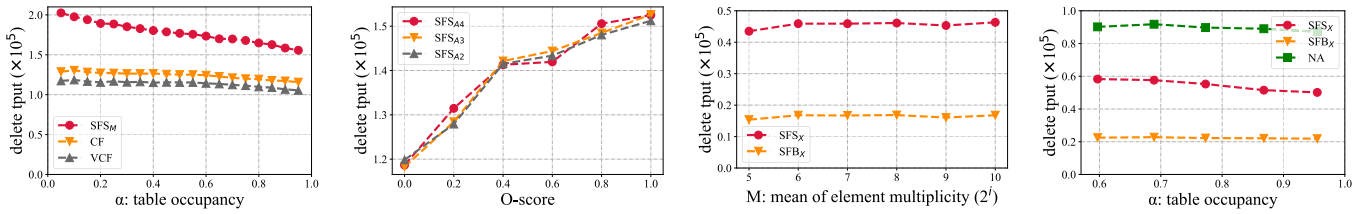
Fig. 9(g) shows the hybrid query throughput of filters with fixed $M = 2^{10}$ when checking synthetic datasets. SFS_X always performs best, followed by SFB_X. They respond even faster as F increases. This is because if there are fewer negative queries, the query may return earlier before all candidate slots or buckets are checked. In contrast, ShBF and BF remain a much lower query throughput. SFS_X has 2.45×, 14.41× and 29.04× overall hybrid query throughput, compared with SFB_X, ABF and ShBF, respectively.

In Fig. 9(h), we fix the F above as 0.5 and further measure the hybrid query throughput of data structures through real-world trace-driven experiments. The performance of CF variants slightly degrades with the α growing due to more occupied slots having to check. Besides, SFS_X performs worse than NA because NA checks much fewer slots at the cost of 18.5% more space overhead. The query speed of ShBF remains extremely low and gets even lower with the increment of the top-1 flow's multiplicity, i.e., c. By contrast, ES maintains a high and stable query throughput irrespective of the increase of either α or c. Compared with ES, NA, SFB_X and ShBF, SFS_X has 0.09×, 0.3×, 2.23×, and 2623× trace-driven hybrid query throughput.

4) Deletion Throughput: Since BF, ShBF, and ES fail to enable element deletion, we quantify the deletion throughput of our methods compared with CF, VCF, and NA for three types of set queries in Fig. 10. For membership query, SFS_M has 1.56× and 1.43× deletion throughput compared with VCF and CF, respectively. Besides, all filters experience instant deletion throughput drop when α grows. That is a natural result of searching for more slots before deleting the target fingerprint. For association query, the deletion throughputs of SFS_A with different numbers of affiliations are about the same, maintaining a high level and increasing with the increment of O-score. For multiplicity queries of synthetic datasets, the deletion curves of both SFS_X and SFB_X are similar to curves of positive query in Fig. 9(e), and the deletion throughput of SFS_X is 2.77× compared with SFB_X in Fig. 10(c). For multiplicity queries of traffic traces, as shown in Fig. 10(d), SFS_X has 0.61× and 2.45× deletion throughput compared with the space-prohibitive NA method and SFB_X.

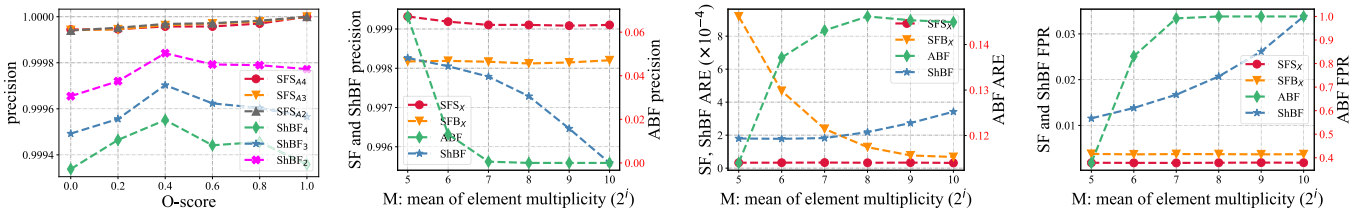
5) Error Rate: We further evaluate the error rate of affiliation and frequency estimation as shown in Fig. 11 and Fig. 12. The quantified metrics include precision, ARE and FPR.

Precision. As depicted by Fig. 11(a), the precision of SFS_A is always higher than the lower bound 99.994% derived by Equ. 13, which is better than ShBF. Furthermore, we find that ShBF suffers from a precision degradation when the number



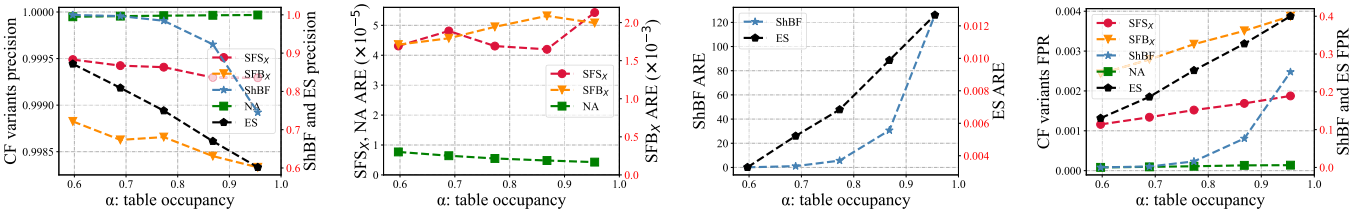
(a) Membership. (b) Association. (c) Multiplicity (synthetic). (d) Multiplicity (trace-driven).

Fig. 10. The deletion throughput of different types of query.



(a) Association: precision. (b) Multiplicity: precision. (c) Multiplicity: ARE (d) Multiplicity: FPR

Fig. 11. The error rate of different queries for synthetic datasets.



(a) Multiplicity: precision. (b) Multiplicity: ARE. (c) Multiplicity: ARE (d) Multiplicity: FPR

Fig. 12. The error rate of multiplicity queries for real-word network traffic traces.

of affiliation increases, as also predicted by the analysis in Subsection V-D. Because ShBF has to check more bits with a higher probability of false matches. For multiplicity queries of synthetic datasets in Fig. 11(b), when M increases from 2^5 to 2^{10}, both ShBF and ABF show significant accuracy drops, and ABF has terrible performance. By contrast, our SFS_X and SFB_X always guarantee above 99.9% and 99.8% multiplicity estimation precision, as Equ. 14 and Equ. 15 also suggests. The fundamental reason is that our method records the exact multiplicity directly. However, ShBF and ABF have to check the non-zero bits to estimate the multiplicity. Besides, ABF realizes worse accuracy than ShBF since it sets much more bits to 1 when recording an element with a large multiplicity. Consequently, the non-zero bits set by other elements may overestimate the actual multiplicity more easily in ABF. Moreover, the membership and multiplicity information in ABF may interfere with each other, leading to even worse precision. For multiplicity queries of traffic traces in Fig. 12(a), SFS_X and SFB_X still guarantee above 99.9% and 99.8% multiplicity estimation precision, far ahead of ShBF and ES, especially at rated 95% workload. NA always maintains nearly 100% precision because it records the multiplicity exactly and fetches fewer slots. With the input of five traffic traces of different sizes, the precision of all data structures degrades more or less for different reasons. For CF variants, the reason is that filters have to check more fingerprints as α grows. For ShBF and ES, it is because the number of vacant bits or count fields decreases when α grows, so they are more likely to fetch incorrect multiplicity information. The increase in c also exacerbates the degradation of ShBF precision. Overall, the trace-driven precision of SFS_X at 95% workload is 1.34× and 1.66× compared with ShBF and ES.

ARE. For multiplicity queries of synthetic datasets in Fig. 11(c), SFS_X has near 3.0×10^{-5} ARE, irrespective of the increment of M. In contrast, the ARE of SFB_X declines from 9.0×10^{-4} to 6.7×10^{-5}, while increases from 1.8×10^{-4} to 3.4×10^{-4} (0.114 to 0.145) in ShBF (ABF). The above results happen for two main reasons: 1) the larger M is, the more bits ShBF has to search, and the more likely it mistakes an element with an incorrect multiplicity. 2)the larger M is, the more bits in ABF are set to 1, so the mechanism of representing information by 0 or 1 in ABF is more likely to fail in a limited space. For trace-driven multiplicity queries in Fig. 12(b) and (c), all data structures suffer ARE increment but NA, especially for ShBF and ES. The reasons for that are the same as that of precision degradation described above. In short, the ARE with synthetic datasets of SFS_X is 0.10×, 0.14×, and 0.0002× compared with SFB_X, ShBF, and ABF, and the trace-driven ARE of SFS_X is 8.04×, 0.02×, 0.006×, and 0.000001× compared with NA, SFB_X, ES, and ShBF.

FPR. Fig. 11(d) shows the multiplicity query FPR of filters checking synthetic datasets. The FPR of SFS_X and SFB_X remain fairly low, which is always lower than the upper bounds 0.002 and 0.004 derived in Equ. 5 and Equ. 9, respectively. However, as the average multiplicity grows, this metric degrades for ShBF and ABF. ABF performs the worst with

significant growth from 0.37 to 1. The reason is that substantial extra bits have been set to 1 in ABF to record element multiplicities. For trace-driven multiplicity queries, Fig. 12(d) shows more clearly that FPR for SFS_X and SFB_X is always below 0.002 and 0.004, which further confirms our theoretical analysis. Similar to the ARE numeric results in Fig. 12 (b) and (c), all data structures suffer performance degradation with varying degrees of FPR increments, including NA, and the reasons are also same to that of precision degradation. In brief, the overall FPR tested with synthetic datasets of SFS_X is 0.494×, 0.089×, and 0.002× compared with SFB_X, ShBF, and ABF, and the trace-driven FPR of SFS_X is 13.6×, 0.48×, 0.02×, and 0.006× compared with NA, SFB_X, ShBF, and ES.

In summary, SF saves space by 21.4%, 18%, 9.98%, and 5.25% than ShBF, BF, VCF, and CF, when they remain the same FPR of 1.5×10^{-5}. SF also saves space by 18.5% than NA. For membership query, SF has 1.14×, 1.18×, 1.30× and 1.60× insertion throughput compared with CF, VCF, ShBF, and BF. SF also has 1.56× and 1.43× deletion throughput compared with VCF and CF, respectively. For association query, in the case of four, three, and two subsets, SF has 3.36×, 2.61×, and 1.88× insertion throughput, and 0.95×, 1.71× and 3.03× query throughput compared with ShBF, respectively. The greater the number of subsets, the lower the insertion throughput, query throughput, and query accuracy of ShBF, while SF remains stable performance irrespective of the number of subsets varies. For multiplicity queries of synthetic datasets, SFS_X has 2.45×, 29.04× and 14.41× hybrid query throughput, 0.10×, 0.14× and 0.0002× ARE, and 0.494×, 0.089× and 0.002× FPR compared with SFB_X, ShBF and ABF, respectively. For trace-driven multiplicity queries, SFS_X has 2.23×, 0.3×, 2623× and 0.09× hybrid query throughput, 1.0×, 1.0×, 1.34× and 1.66× precision, 0.02×, 8.04×, 0.000001× and 0.006× ARE, and 0.48×, 13.6×, 0.02× and 0.006× FPR compared with SFB_X, NA, ShBF and ES, respectively. Given the strength of query throughput, deletion throughput, precision, ARE, and FPR, we prefer to recommend SFS_X for multiplicity queries. However, if space is the primary resource for optimizing, then SFB_X is a better choice as it is more space-efficient than SFS_X.

IX. Conclusion

This paper presents the Shifting filter design for dynamic set representation and query with the ambition of higher throughput, lower error rate, fewer constraints, and more functionalities. We exhibit how to use Shifting filter to answer three crucial dynamic set queries, i.e., membership, association, and multiplicity query. At its core, Shifting filter encodes the auxiliary information of the set element in the offset on slot or bucket level, with mark field or count field as assistance. Such a design avoids significant performance degradation while ensuring compact space overhead. Theoretical analysis and comprehensive experiments commit that Shifting filter and its variants support elegantly all three types of set queries with comparable or better performance than its competitors. The major limitation of our methods is that the various throughputs are not high enough, especially the query throughput. In future work, we will try to design a lightweight indexing scheme to replace the partial cuckoo hashing technique in the Cuckoo filter so that the data structure can respond faster to various data operations.

Acknowledgment

The authors would like to thank all the anonymous reviewers for their insightful feedback.

References

[1] 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, 1st Quart., 2012.

[2] X. Wang, Y. Liu, Z. Yang, K. Lu, and J. Luo, "Robust component-based localizationin sparse networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 5, pp. 1317–1327, May 2014.

[3] Y. Zhai, H. Xu, H. Wang, Z. Meng, and H. Huang, "Joint routing and sketch configuration in software-defined networking," *IEEE/ACM Trans. Netw.*, vol. 28, no. 5, pp. 2092–2105, Oct. 2020.

[4] P. Reviriego and O. Rottenstreich, "The tandem counting Bloom filter—It takes two counters to tango," *IEEE/ACM Trans. Netw.*, vol. 27, no. 6, pp. 2252–2265, Dec. 2019.

[5] A. Sánchez-Macián, P. Reviriego, J. A. Maestro, and S. Liu, "Single event transient tolerant Bloom filter implementations," *IEEE Trans. Comput.*, vol. 66, no. 10, pp. 1831–1836, Oct. 2017.

[6] T. Buddhika, S. L. Pallickara, and S. Pallickara, "Pebbles: Leveraging sketches for processing voluminous, high velocity data streams," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 8, pp. 2005–2020, Aug. 2021.

[7] T. Yang et al., "A shifting Bloom filter framework for set queries," *ACM J. VLDB Endowment*, vol. 9, no. 5, pp. 408–419, Jan. 2016.

[8] B. M. Maggs and R. K. Sitaraman, "Algorithmic nuggets in content delivery," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 3, pp. 52–66, Jul. 2015.

[9] J. H. Mun and H. Lim, "New approach for efficient IP address lookup using a Bloom filter in trie-based algorithms," *IEEE Trans. Comput.*, vol. 65, no. 5, pp. 1558–1565, May 2016.

[10] 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.

[11] L. Luo, D. Guo, Y. Zhao, O. Rottenstreich, R. T. B. Ma, and X. Luo, "MCFsyn: A multi-party set reconciliation protocol with the marked cuckoo filter," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 11, pp. 2705–2718, Nov. 2021.

[12] S. Wu, D. Jiang, B. C. Ooi, and K.-L. Wu, "Efficient B-tree based indexing for cloud data processing," *Proc. VLDB Endowment*, vol. 3, nos. 1–2, pp. 1207–1218, 2010.

[13] M. K. Aguilera, W. Golab, and M. A. Shah, "A practical scalable distributed B-tree," *Proc. VLDB Endowment*, vol. 1, no. 1, pp. 598–609, Aug. 2008.

[14] G. Lu, Y. J. Nam, and D. H. C. Du, "BloomStore: Bloom-filter based memory-efficient key-value store for indexing of data deduplication on flash," in *Proc. IEEE 28th Symp. Mass Storage Syst. Technol. (MSST)*, Pacific Grove, CA, USA, Apr. 2012, pp. 1–11.

[15] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig latin: A not-so-foreign language for data processing," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Vancouver, BC, Canada, Jun. 2008, pp. 1099–1110.

[16] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *Proc. 2nd ACM SIGOPS/EuroSys Eur. Conf. Comput. Syst.*, Lisbon, Portugal, Mar. 2007, pp. 59–72.

[17] T. Yang et al., "Elastic sketch: Adaptive and fast network-wide measurements," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Budapest, Hungary, Aug. 2018, pp. 561–575.

[18] Y. Zhang et al., "CocoSketch: High-performance sketch-based measurement over arbitrary partial key query," in *Proc. ACM SIGCOMM Conf.*, Aug. 2021, pp. 207–222.

[19] L. Liu et al., "SF-sketch: A two-stage sketch for data streams," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 10, pp. 2263–2276, Oct. 2020.

[20] L. Tang, Q. Huang, and P. P. C. Lee, "A fast and compact invertible sketch for network-wide heavy flow detection," *IEEE/ACM Trans. Netw.*, vol. 28, no. 5, pp. 2350–2363, Oct. 2020.

[21] H. Wang, H. Xu, L. Huang, and Y. Zhai, "Fast and accurate traffic measurement with hierarchical filtering," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 10, pp. 2360–2374, Oct. 2020.
[22] D. Tong and V. K. Prasanna, "Sketch acceleration on FPGA and its applications in network anomaly detection," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 4, pp. 929–942, Apr. 2018.
[23] Z. Zhong, S. Yan, Z. Li, D. Tan, T. Yang, and B. Cui, "BurstSketch: Finding bursts in data streams," in *Proc. Int. Conf. Manage. Data*, Jun. 2021, pp. 2375–2383.
[24] D. Paul, Y. Peng, and F. Li, "Bursty event detection throughout histories," in *Proc. IEEE 35th Int. Conf. Data Eng. (ICDE)*, Macao, China, Apr. 2019, pp. 1370–1381.
[25] L. Luo, D. Guo, R. T. B. Ma, O. Rottenstreich, and X. Luo, "Optimizing Bloom filter: Challenges, solutions, and comparisons," *IEEE Commun. Surveys Tuts.*, vol. 21, no. 2, pp. 1912–1949, 2nd Quart., 2018.
[26] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970.
[27] 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. Exp. Technol.*, Sydney, NSW, Australia, Dec. 2014, pp. 75–88.
[28] L. Fan, P. Cao, J. M. Almeida, and A. Z. Broder, "Summary cache: A scalable wide-area web cache sharing protocol," in *Proc. ACM SIGCOMM*, Vancouver, BC, Canada, Aug. 1998, pp. 254–265.
[29] Y. Matsumoto, H. Hazeyama, and Y. Kadobayashi, "Adaptive Bloom filter: A space-efficient counting algorithm for unpredictable network traffic," *IEICE Trans. Inf. Syst.*, vol. 91, no. 5, pp. 1292–1299, 2008.
[30] D. Guo, J. Wu, H. Chen, Y. Yuan, and X. Luo, "The dynamic Bloom filters," *IEEE Trans. Knowl. Data Eng.*, vol. 22, no. 1, pp. 120–133, Jan. 2010.
[31] D. Eppstein, M. T. Goodrich, F. Uyeda, and G. Varghese, "What's the difference? Efficient set reconciliation without prior context," in *Proc. ACM SIGCOMM Conf. (SIGCOMM)*, Toronto, ON, Canada, 2011, pp. 218–229.
[32] B. Fan, D. G. Andersen, and M. Kaminsky, "MemC3: Compact and concurrent memcache with dumber caching and smarter hashing," in *Proc. USENIX NSDI*, Lombard, IL, USA, Apr. 2013, pp. 371–384.
[33] M. Mitzenmacher, S. Pontarelli, and P. Reviriego, "Adaptive cuckoo filters," in *Proc. ALENEX*, New Orleans, LA, USA, Jan. 2018, pp. 36–47.
[34] L. Luo, D. Guo, O. Rottenstreich, R. T. B. Ma, X. Luo, and B. Ren, "The consistent cuckoo filter," in *Proc. IEEE INFOCOM Conf. Comput. Commun.*, Apr. 2019, pp. 712–720.
[35] H. Chen, L. Liao, H. Jin, and J. Wu, "The dynamic cuckoo filter," in *Proc. IEEE 25th Int. Conf. Netw. Protocols (ICNP)*, Toronto, ON, Canada, Oct. 2017, pp. 1–10.
[36] D. Eppstein, "Cuckoo filter: Simplification and analysis," in *Proc. SWAT*, vol. 53, Reykjavik, Iceland, Jun. 2016, p. 8.
[37] P. Fu, L. Luo, S. Li, D. Guo, G. Cheng, and Y. Zhou, "The vertical cuckoo filters: A family of insertion-friendly sketches for online applications," in *Proc. IEEE 41st Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Washington, DC, USA, Jul. 2021, pp. 57–67.
[38] M. Wang, M. Zhou, S. Shi, and C. Qian, "Vacuum filters: More space-efficient and faster replacement for Bloom and cuckoo filters," *VLDB Endow.*, vol. 13, no. 2, pp. 197–210, 2019.
[39] P. Reviriego, J. Martinez, and M. Ottavi, "Soft error tolerant count min sketches," *IEEE Trans. Comput.*, vol. 70, no. 2, pp. 284–290, Feb. 2021.
[40] (2016). *The Wide Internet Traces*. [Online]. Available: http://mawi.wide.ad.jp/mawi

Pengtao Fu received the B.S. and M.S. degrees from the College of Systems Engineering, National University of Defense Technology, Changsha, China, in 2020 and 2022, respectively. His research interests include data structure and network measurement.

Lailong Luo received the B.S., M.S., and Ph.D. degrees from the College of Systems Engineering, National University of Defense Technology, Changsha, China, in 2013, 2015, and 2019, respectively. He is currently an Associate Professor with the College of Systems Engineering, National University of Defense Technology. His research interests include data structure and distributed networking systems.

Deke Guo (Senior Member, IEEE) received the B.S. degree in industry engineering from the Beijing University of Aeronautics and Astronautics, Beijing, China, in 2001, and the Ph.D. degree in management science and engineering from the National University of Defense Technology, Changsha, China, in 2008. He is currently a Professor with the College of Systems Engineering, National University of Defense Technology. His research interests include distributed systems, software-defined networking, data center networking, wireless and mobile systems, and interconnection networks. He is a member of ACM.

Shangsen Li received the B.S. degree in automation from Northeastern University, Shenyang, China, in 2019, and the M.S. degree from the College of Systems Engineering, National University of Defense Technology, Changsha, China, in 2021, where he is currently pursuing the Ph.D. degree. His research interests include network measurement, SDN, and sketch data structure.

Yun Zhou (Member, IEEE) received the Ph.D. degree in computer science from the Queen Mary University of London, U.K., in 2015. He is currently an Associate Professor with the College of Systems Engineering, National University of Defense Technology, Changsha, China. His research interests include machine learning and probabilistic graphical models. He has published several papers in reputed journals and conferences in this area, including INFOCOM, IJCAI, *International Journal of Advanced Research* (IJAR), UAI, and PGM.