Ark Filter: A General and Space-Efficient Sketch for Network Flow Analysis

Lailong Luo, Pengtao Fu, Shangsen Li, Deke Guo, *Senior Member, IEEE, Member, ACM*, Qianzhen Zhang, and Huaimin Wang

***Abstract*— Sketches are widely deployed to represent network flows to support complex flow analysis. Typical sketches usually employ hash functions to map elements into a hash table or bit array. Such sketches still suffer from potential weaknesses upon throughput, flexibility, and functionality. To this end, we propose Ark filter, a novel sketch that stores the element information with either of two candidate buckets indexed by the quotient or remainder between the fingerprint and filter length. In this way, no further hash calculations are required for future queries or reallocations. We further extend the Ark filter to enable capacity elasticity and more functionalities (such as frequency estimation and top-k query). Comprehensive experiments demonstrate that, compared with Cuckoo filter, Ark filter has $2.08\times$, $1.34\times$, and $1.68\times$ throughput of deletion, insertion, and hybrid query, respectively; compared with Quotient filter, Ark filter has $4.55\times$, $1.74\times$, and $22.12\times$ throughput of deletion, insertion, and hybrid query, respectively; compared with Bloom filter, Ark filter has $2.55\times$ and $2.11\times$ throughput of insertion and hybrid query, respectively.**

***Index Terms*— Ark filter, network flow analysis, data sketch.**

I. Introduction

NETWORK flow analysis is essential for network monitoring, measuring, planning, billing, and security [1]. Nowadays, data sketches are implemented to represent the collected flow information for instant queries such as membership, frequency, cardinality, etc. Several surveys summarize the usage and design of sketches in the network community [2], [3], [4]. Basically, many sketches are initially proposed to represent and analyze network flows. Typical sketches include Bloom filter [5], Cuckoo filter [6], count-min [7], Quotient filter [8] and their variants [9], [10], [11], [12]. We use the word "element" rather than "flow" to ease the description of general sketches. Such sketches usually employ hash functions to map the element content into one or multiple locations in a hash table or bit array. Then the interested information of the mapped elements is therefore recorded by the bits, fingerprints, or counters in these locations.

Sketches may adopt different strategies to enable varied functionalities. Bloom filter [5] relies on h independent hash functions to map each element into h locations of a bit array initialized as 0s. Then these bits are set to 1 to record the membership information explicitly. Queries are simply responded to by checking these bits. By contrast, Cuckoo filter [6] stores the fingerprint of every element directly with a hash table that has m buckets (columns) and b slots (rows). By providing two candidate buckets to each element, Cuckoo filter guarantees a high space utilization. Besides membership query, more fields (such as counter [13], [14], checkSum [11], [15], flag bits [16], [17]) are added into Bloom and Cuckoo for better performance and more functionalities (e.g., deletion, counting).

To support network flow analysis better, we envision the sketches with the following three features: 1) **high-throughput**, the throughput for flow insertion, deletion, and query should be constant-time and fast enough; 2) **space-lineal**, the space used by the sketch is proportional to the number of flows to record and the length of the sketch can be set at will; 3) **multi-functional**, the sketch is easy to be tuned to support diverse functionalities beyond membership query. Such features, if realized simultaneously, will bring great benefits to the sketch-based network flow analysis, in terms of throughput, space, and usability. However, the mainstream sketches today fail to achieve these features in one stroke. Next, we take the Cuckoo filter as a representative example to convey this point.

Cuckoo [6] is declared to be practically better than Bloom and many state-of-the-art sketches are designed based on it. However, Cuckoo filter still has its intrinsic shortage upon throughput. For an element x, when both of the two candidates are full, Cuckoo filter kicks out a stored fingerprint randomly to accommodate x's fingerprint η_x. The kicked-out victim will thereafter be redirected to its alternative candidate bucket. Such a reallocation process is allowed for at most max times to improve space utilization. The two candidate buckets are indexed by: $h_1(x) = hash(x)$ and $h_2(x) = h_1(x) \oplus hash(\eta_x)$. Each reallocation in Cuckoo filter has to

Manuscript received 22 June 2022; revised 14 January 2023; accepted 26 March 2023; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor G. Iosifidis. Date of publication 13 April 2023; date of current version 19 December 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 under Grant ZK20-30. *(Corresponding author: Lailong Luo.)*

Lailong Luo is with the Science and Technology on Information Systems Engineering Laboratory and the National Laboratory for Parallel and Distributed Processing, National University of Defense Technology, Changsha, Hunan 410073, China (e-mail: luolailong09@nudt.edu.cn).

Pengtao Fu, Shangsen Li, Deke Guo, and Qianzhen Zhang are with the Science and Technology on Information Systems Engineering Laboratory, National University of Defense Technology, Changsha, Hunan 410073, China (e-mail: fupengtao@nudt.edu.cn; lishangsen@nudt.edu.cn; dekeguo@nudt.edu.cn; zhangqianzhen18@nudt.edu.cn).

Huaimin Wang is with the National Laboratory for Parallel and Distributed Processing, National University of Defense Technology, Changsha, Hunan 410073, China (e-mail: whm_w@163.com).

Digital Object Identifier 10.1109/TNET.2023.3263839

execute one hash calculation and one XOR operation. Such calculations may burden the insertion throughput, especially when the filter is already highly utilized. *Therefore, a more lightweight reallocation strategy is necessary to further speed up the throughput of Cuckoo filter.*

Besides, Cuckoo filter is not space-lineal since its length (m) has to be precisely in the form of $m = 2^c$, where $c > 1$ is an integer; otherwise, the XOR operation may run out of range. The lack of design flexibility makes Cuckoo filter either underutilized or incapable of representing a given number of elements. For instance, given $b = 4$ and 520 elements to represent, the filter length should be $m = 256$, leading to a severe waste of space; yet letting $m = 128$ will not provide enough slots for these elements. *Consequently, a flexible structure whose capacity is proportional to the number of elements to represent is required for better space efficiency.*

Lastly, complex network flow analysis requires the sketch to support multiple functionalities besides membership queries yet the original cuckoo filter and Bloom filter fail to tackle this issue. Representative functionalities include dynamic representation (capacity elasticity), cardinality estimation, frequency estimation, and top-k query. These functionalities are helpful for flow accounting, heavy hitter detection, super spreader detection, entropy estimation, and beyond. However, the existing sketches fail to support such functionalities or discuss the generalization for such functionalities. For example, Bloom filter [5], counting Bloom filter [13], Dynamic Bloom filter [9] realize membership query, element deletion, and dynamic representation, however, in a separated manner. Yet none of them consider these functionalities jointly. This problem also fits Cuckoo filter [6] and dynamic cuckoo filter [12]. *In other words, a more general sketch with joint consideration of more functionalities is demanded to support network flow analysis.*

To meet the above needs of network flow analysis, this paper presents Ark filter, a novel sketch that simultaneously achieves the above design rationales. Ark filter consists of a hash table with m buckets, each with exactly b slots. In each slot, there are two fields, i.e., the $Carry$ field and the $Flag$ bit. Ark filter also provides two candidate buckets for each element, and reallocations are also allowed for at most max time to achieve a higher space utilization. Ark filter bounds any fingerprint in the range $[0, m(m-1)]$. Unlike Cuckoo filter, which derives out the candidate buckets with hash functions, Ark filter multiplexes the fingerprint information and the candidate index information. Specifically, the Ark filter divides the element fingerprint into two parts (quotient and remainder upon the filter length) to index its two candidate buckets. After that, the bucket indexed by the quotient only needs to store the remainder part with the $Carry$ field, and vise versa. The $Flag$ bit will be correspondingly set as 1 or 0 to explicitly indicate the $Carry$ field stores quotient or remainder. With this design, to ease the reallocation process, the alternative candidate bucket can be indexed directly by the $Carry$, $Flag$, and the index of the current candidate bucket. Consequently, the hash calculation and XOR operation during each Cuckoo filter reallocation are no longer needed.

Moreover, we redesign the Ark filter to support more functionalities. To be specific, the Dynamic Ark filter (DAF) scales up(down) by adding(removing) untapped(underutilized) rows (i.e., changing the value of b) from the filter to enable capacity elasticity. For frequency estimation, the Counting Ark filter (CAF) adds a counter field into each slot to track the frequency consistently. As for top-k query, the sorted CAF proposes to rank the elements in every bucket according to their frequencies in either an ascending or descending order. We summarize the contribution of this paper as follows:

- We present Ark filter, a novel sketch that multiplexes the fingerprint information and the candidate index information such that the candidate buckets for each element can be indexed with the fingerprint directly. In this way, only one hash calculation is needed for each element, thus saving time for element insertion, deletion, and query.
- We further propose several Ark filter variants to support capacity elasticity, frequency estimation, and top-k query. DAF changes the value of b dynamically to fit the real set volume; CAF tracks the element frequency with the counter field in each slot; and the sorted CAF ranks the elements in each bucket so that the top-k queries can get instant response.
- We conduct comprehensive experiments to compare the Ark filter with its same-kinds. Theoretical analysis and numerical results demonstrate that Ark filter achieves the wanted design rationales simultaneously. It outperforms Cuckoo, Bloom, and Quotient in terms of the query, insertion, and deletion throughput. Its variants also realize comparable or better performance than their competitors.

The remainder of this paper is organized as follows. Section II introduces the background and related work. Section III details the design of Ark filter. Section IV presents the three variants of Ark to support capacity elasticity, frequency estimation, and top-k query. Section V reports the evaluation results. Section VI discusses several additional issues of our Ark filter and at last, Section VI concludes the whole paper.

II. Related Work

A. The State-of-the-Art Sketches

Bloom filter and its variants. Bloom filter [5] maintains a bit vector with m bits, which are all initialized as 0. For any element x in a set S with n members, h independent hash functions are employed to map it into the vector. The h corresponding bits in the vector are set to 1. To query an element x, Bloom filter just checks the h corresponding bits: if all the h bits are non-zero, Bloom filter returns *True* to indicate that x is a member of S; by contrast, if any of the bits is zero, Bloom filter judges that $x \notin S$ and returns *False*. Intuitively, the Bloom filter may lead to one-sided false positive error yet never false negative. For a query, the potential false positive rate is $\xi_{BF} \approx (1 - e^{-hn/m})^h$. When $h = h_{opt} = \frac{m}{n} \ln 2$, $\xi_{BF} = 0.5^h \approx 0.6185^{m/n}$. Typical Bloom filter variants include Counting Bloom filter [13] (CBF), Dynamic Bloom filter [9] (DBF), Invertible Bloom filter [15] (IBF) and Invertible Bloom lookup table [11] (IBLT). They are

investigated to support element deletion, capacity resizing, and reverse decoding. More Bloom filter variants are proposed to improve the Bloom filter from a performance or generalization perspective in diverse circumstances [2], [3], [4], [10].

Cuckoo filter and its variants. Cuckoo filter [6] represents an element x by storing its fingerprint η_x directly. Cuckoo filter consists of m bucket, each of which has b slots to accommodate at most b fingerprints. Cuckoo filter provides two candidate buckets for x so that η_x can be stored in either of them. The two candidates are indexed as: $h_1(x) = hash(x)$, and $h_2(x) = h_1(x) \oplus hash(\eta_x)$.

To insert an element x, Cuckoo filter first checks the two candidate buckets, and if either of them has available slot(s), the fingerprint η_x will be stored there. Otherwise, Cuckoo filter has to randomly kick out an existing fingerprint from these two candidates and put η_x in that slot. The kicked-out fingerprint is marked as a victim and directed to its alternative bucket. If the alternate has an empty slot to reside the victim, the insertion is terminated successfully; otherwise, the above *kick-out-and-reallocate* scheme continues exploring more buckets until such a reallocation reaches max times. To answer the membership query, the Cuckoo filter just needs to check the two candidate buckets with the following bounded false positive rate:

$$\xi_{CF} = 1 - (1 - \frac{1}{2^l})^{2b} \approx \frac{2b}{2^l}, \tag{1}$$

where l is the length of a fingerprint.

The simplified Cuckoo filter [18] calculates the indices of buckets for an element x as $h_1(x)$ and $h_1(x) \oplus \eta_x$ such that a theoretical guarantee is generated based on graph theory. The adaptive Cuckoo filter [16] tries to remove false positive errors from the vector by resetting the collided fingerprints with optional hash functions. Dynamic Cuckoo filter [12] (DCF) dynamically maintains multiple homogeneous CFs to enable elastic capacity. The consistent Cuckoo filter [19] maps the buckets and elements onto a consistent hash ring to enable more fine-grained capacity alterations. Morton filter [20] introduces virtual buckets and divides logical buckets into memory-aligned blocks to realize higher throughput. Vacuum filter [21] confines the element insertion, deletion, and query within a chunk to achieve significant throughput improvements and memory access decrements.

The Quotient filters. Quotient filter [8], [22] is a hash table of slots to store the fingerprints of elements with the quotient technique. The fingerprint of an element is divided into two parts, i.e., the u most significant bits as the quotient and the v least significant bits as the remainder. A remainder is stored in the slot suggested by the quotient. For each slot, there are two or three bits to handle the hash collisions. However, they have to set their lengths as a power of two. That will somehow hurt the generalization of quotient filters.

B. Sketches With Additional Functionalities

Frequency estimation. Frequency is a basic feature of any element in a dataset to analyze. Maintaining a sorted table or hash table to store the frequencies is not advisable for massive space overhead. To this end, a bunch of sketches is designed for frequency estimation. Count-Min [7] maintains a matrix of counters (initialized as 0s) with d rows and w columns. A coming element is mapped into the matrix with d independent hash functions. Thereafter, the corresponding d counters will be added up by 1. With such a framework, the frequency of an arbitrary element will be returned as the minimum value among the d corresponding counters. Several variants of Count-Min are presented to fit the frequency distribution better, e.g., Count-Mean-Min [23].

Multiple Bloom filter variants are also tuned to respond to frequency queries, e.g., Adaptive Bloom filter (ABF) [24], Shifting Bloom filter (SBF) [25]. To represent an element whose frequency is f, ABF needs $q + f + 1$ independent hash functions in total. Among them, q hash functions act for membership just as the traditional Bloom filter does; while f hash functions set f bits to 1s to explicitly record the frequency. ABF has to check $q + f + 1$ bits to respond to the query until the last bit is 0. As for SBF, it records the membership information of each element just like the traditional Bloom filter does. Additionally, auxiliary information (e.g., multiplicity) is further recorded by calculating an offset function $o(x)$. To query the membership information, SBF just checks the corresponding k membership bits. However, to respond to a multiplicity query, SBF has to check c bits after every membership bit, where c is the maximum multiplicity in the dataset. Therefore, its query throughput can be an essential drawback. Moreover, both ABF and SBF need to know the exact multiplicity before inserting the elements, making them inapplicable in dynamic scenarios wherein multiplicity changes timely.

Moreover, considering the skewness of flow size (number of packets contained by a flow), efforts have been made to distinguish the large flows from the small ones and record them respectively with different strategies or unequal-sized counters. Specifically, TowerSketch [26] has different-sized counters for different arrays so that the large flows will be kept by long counters and vice versa. The Stingy Sketch [27] also alters the fixed-size counters into multiple smaller counters yet organizes them as a tree structure. The Cold filter [28] uses a two-layer sketch with small counters to accurately record the small flows. If all of the corresponding counters for a flow overflows, that flow will be marked as a large flow and represented by other sketches. Especially, Cocosketch [29] tries to answer *partial key queries* by recording the meta information in the data plane while building the query results in the control plane. We believe such methodologies are orthogonal to our design and can be applied to optimize the counter field of the CAF and Sorted CAF.

Top-k query. Finding the largest k elements, also referred to as the top-k elements, is a fundamental problem for various data analysis jobs. To this end, Lossy Counting [30] and Space-Saving [31] maintain a data structure that counts only part of the promising elements. More precisely, Space-Saving inserts a new element in the data structure by replacing an existing element with the lowest frequency; while Lossy Counting expels the elements at each time slot to make room for new elements. Such designs will cause significant errors when the space is quite limited. HeavyKeeper [32], by contrast, records the information of top elements while

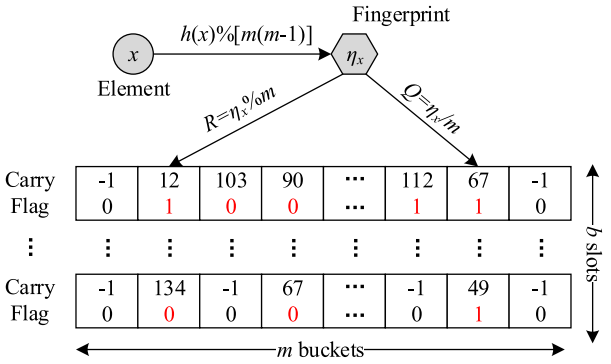


Fig. 1. The framework of Ark filter. For an arbitrary element x, its two candidate buckets are indexed by the quotient and remainder generated by the fingerprint η_x and the filter length m.

ignoring insignificant ones. However, it may kill those elements which may grow as top members.

III. The Ark Filter Methodology

A. Framework of Ark Filter

As shown in Fig. 1, the Ark filter consists of m buckets, each of which has b slots. Each slot has two fields, i.e., the *Carry* field and the *Flag* field. The *Carry* field records the part information of the fingerprint (either the remainder or the quotient as specified later). The *Flag* field is a single bit to explicitly mark that the stored information is the remainder part (by setting *Flag* as 1) or the quotient part (by remaining *Flag* as 0) of the fingerprint. Initially, the *Carry* field is set as -1 and the *Flag* bit is set as 0 for an empty slot. For any element x, a hash function $h(x)$ is employed to generate a fingerprint η_x. The fingerprint η_x ranges from 0 to $m(m-1)$. Ark filter differentiates from Cuckoo filter [6] and its variants mainly from two aspects. First, Ark filter stores partial information of the fingerprint, either the quotient part or the remainder part. Second, Ark filter provides two candidate buckets with the indexes of quotient and remainder, rather than calculated with the partial cuckoo hashing strategy. To be specific, the quotient and remainder (denoted as Q and R, respectively) are calculated as follows:

$$Q_x = \eta_x / m, \tag{2}$$

$$R_x = \eta_x \% m. \tag{3}$$

In Ark filter, an element is successfully represented if the corresponding remainder R_x is stored by the bucket indexed by the quotient, or the quotient Q_x is resided by the bucket with the index of R_x. To resolve the hash collision and improve the space utilization, Ark filter also follows the "kick-out-and-reallocate" strategy. To be specific, if either of the two candidate buckets has an empty slot, the quotient or remainder information will be stored there. By contrast, if both candidates are saturated, a random stored quotient (when the *Flag* bit is 0) or remainder (if the *Flag* bit is 1) will be kicked out as a victim to offer space for the coming element x. Thereafter, the victim will be redirected to its alternative candidate bucket. Such reallocations succeed if there is no further victim or fail when reallocations reach the pre-defined upper bound max.

Algorithm 1 Element insertion(x)

Input: The element to insert x
Calculate the fingerprint as $\eta_x = h(x)\%[m(m-1)]$;
Decide the candidate buckets as $Q_x = \eta_x / m$ and $R_x = \eta_x \% m$;
if *Either* $AF[Q_x]$ *or* $AF[R_x]$ *has empty slots* **then**
  if $AF[Q_x]$ *is the selected bucket* **then**
    Locate the first empty slot $AF[Q_x][i]$;
    $AF[Q_x][i].Carry = R_x$;
    $AF[Q_x][i].Flag = 1$;
    return True;
  else
    Locate the first empty slot $AF[R_x][i]$;
    $AF[R_x][i].Carry = Q_x$;
    $AF[R_x][i].Flag = 0$;
    return True;
else
  $count = 0$;
  Let T be a random integer selected from $\{R_x, Q_x\}$;
  Let C and ϕ be the *Carry* and *Flag* fields of a random slot in $AF[T]$;
  Empty the random slot from $AF[T]$;
  Store Q_x or R_x with the empty slot and set/reset the *Flag* bit;
  while $count \leq max$ **do**
    if $AF[C]$ *has empty slot* **then**
      Store T in $AF[C]$;
      Set/reset $Flag$ in the empty slot based on ϕ;
      return True;
    else
      $r = random(0, b)$;
      $temp_c = AF[C][r].Carry$;
      $temp_\phi = AF[C][r].Flag$;
      $AF[C][r].Carry = T$;
      $AF[C][r].Flag = \neg\phi$;
      $T = C$;
      $C = temp_c$;
      $\phi = temp_\phi$;
    $count \mathrel{+}= 1$;
  return False;

Ark filter naturally multiplexes the index information of candidate buckets and fingerprint information with such a framework. This methodology only needs one hash calculation for each insertion. Besides, the fingerprint ranges in $[0, m(m-1)]$ such that both the quotient and remainder part will never overflow from the filter.

B. Operations of Ark Filter

Ark filter supports general element-oriented operations such as element insertion, query, and deletion. Besides, for two sets of elements, denoted as A and B respectively, typical set operations such as union, subtraction, and intersection are also required for big data analysis. Therefore, filter-

level functionalities (e.g., compact, subtraction, intersection) between two Ark filters should be additionally enabled.

Element insertion. Intuitively, there are two basic principles for inserting an arbitrary element x: 1) the information of x is stored in either of its candidate buckets; 2) Ark filter represents x by storing Q_x in the bucket $AF[R_x]$ or reserving R_x with the bucket $AF[Q_x]$. As specified in Algorithm 1, Ark filter first derives out the fingerprint, the remainder, and the quotient of x. If either $AF[R_x]$ or $AF[Q_x]$ has an empty slot, the Q_x or R_x will be stored there by leveraging the *Carry* and *Flag* fields of that slot (line 3 to 13). Otherwise, Ark filter has to kick out an existing quotient (if the *Flag* is 0) or remainder (if the *Flag* is 1) in $AF[R_x]$ or $AF[Q_x]$ to accommodate x. The victim, on the other hand, will be redirected to its alternative candidate bucket. If that bucket can successfully represent the victim, the algorithm returns true; otherwise, an existing element from that bucket will be kicked out as a new victim. Such a kick-out-and-reallocate strategy keeps exploring the filter until no further victim or such reallocations reach a given upper bound max (line 16 to 34). During the whole process, the *Flag* bit is updated as 1(0) if the slot stores a remainder(quotient). Note that when the filter returns false, the filter is marked as full, and no more elements should be inserted. The time complexity of inserting an element is $O(b \times max)$, since at most max reallocations will be conducted, and each reallocation needs to check at most b slots in a bucket.

Algorithm 2 Membership query(x)

Input: The element to query x
1 Calculate the fingerprint as $\eta_x = h(x)\%[m(m-1)]$;
2 Decide the candidate buckets as $Q_x = \eta_x/m$ and $R_x = \eta_x\%m$;
3 **for** $i = 0$ *to* $b-1$ **do**
4 **if** $AF[Q_x][i].Carry == R_x$ *and* $AF[Q_x][i].Flag == 1$ **then**
5 **return** True;
6 **for** $i = 0$ *to* $b-1$ **do**
7 **if** $AF[R_x][i].Carry == Q_x$ *and* $AF[R_x][i].Flag == 0$ **then**
8 **return** True;
9 **return** False;

Membership query. As specified in Algorithm 2, to query the membership of any element x, Ark filter first derives out the fingerprint η_x, as well as the two candidate buckets Q_x and R_x. After that, Ark filter just checks the two buckets. In the $AF[Q_x]$ bucket, if R_x can be found in any slot, and the *Flag* bit in that slot is precisely 1, Ark filter commits the membership of x and returns true. Alternatively, if Q_x is searched out as the *Carry* field in a slot and the *Flag* bit in that slot is right 0, Ark filter also returns true. Unlike Cuckoo filters which verify the fingerprints directly, Ark filter checks whether the quotient part is stored by the bucket, which is indexed by the remainder with the correct *Flag* bit, or vice versa. The time complexity of membership query is still constant since only two buckets will be accessed and examined.

Algorithm 3 Element deletion(x)

Input: The element to query x
1 Calculate the fingerprint as $\eta_x = h(x)\%[m(m-1)]$;
2 Decide the candidate buckets as $Q_x = \eta_x/m$ and $R_x = \eta_x\%m$;
3 **for** $i = 0$ *to* $b-1$ **do**
4 **if** $AF[Q_x][i].Carry == R_x$ *and* $AF[Q_x][i].Flag == 1$ **then**
5 $AF[Q_x][i].Carry == -1$;
6 $AF[Q_x][i].Flag == 0$;
7 **return** True;
8 **for** $i = 0$ *to* $b-1$ **do**
9 **if** $AF[R_x][i].Carry == Q_x$ *and* $AF[R_x][i].Flag == 0$ **then**
10 $AF[R_x][i].Carry == -1$;
11 **return** True;
12 **return** False;

Algorithm 4 Filter Compact(AF_A and AF_B)

Input: Two homogeneous Ark filters AF_A and AF_B
1 %% Suppose AF_A stores more elements than AF_B;
2 **for** $i = 0$ *to* $m-1$ **do**
3 **for** $j = 0$ *to* $b-1$ **do**
4 **if** $AF_B[i][j].Carry \neq -1$ **then**
5 **if** $AF_A[i]$ *or* $AF_A[AF_B[i][j].Carry]$ *has an empty slot* **then**
6 Store $AF_B[i][j].Carry$ with $AF_A[i]$ or store i with $AF_A[AF_B[i][j].Carry]$;
7 Set the $Flag$ bit of this empty slot;
8 **if** *This element cannot be stored by* AF_A *by reallocations* **then**
9 **return** False;
10 **return** True;

Element deletion. To delete an element x, Ark filter has to figure out whether x has been represented previously. As illustrated in Algorithm 3, Ark filter first calculates the fingerprint, the quotient, and the remainder for x. Then, Ark filter tries to delete the quotient information Q_x from the bucket whose index is exactly R_x and vise versa. Note that, before deletion, the $Flag$ bit has to be checked to make sure that the right element is removed from the filter. If it succeeds, Ark filter returns true; otherwise, it returns false to demonstrate that x is not stored before. The time complexity of element deletion also remains constant.

Filter compact/union. Given two homogeneous filters (denoted as AF_A and AF_B) with the same parameters m and b, filter compact means to merge the two filters as one which contains all the fingerprints. As shown in Algorithm

4, suppose that AF_A stores more elements than AF_B, the compact operation tries to represent the fingerprints in AF_B with AF_A. If all the fingerprints in AF_B can be inserted into AF_A successfully, the algorithm returns true, else false. By doing so, the updated AF_A records all the elements in the union set $A \cup B$. Such operation can be accomplished iff $|A| + |B|$ is less than the capacity of AF_A. Besides, when multiple filters are maintained to represent dynamic sets wherein elements arrive or perish dynamically, the compact algorithm can be employed to recycle the sparse filters. This functionality is quite essential for space-scarce situations. The time complexity of this operation is $O(m \times b)$.

Filter subtraction. Given two homogeneous Ark filters AF_A and AF_B, subtracting AF_B from AF_A means to remove the common elements from AF_A. Algorithm 5 specifies how the common elements are removed from AF_A. For any slot $AF_A[i][j]$ in AF_A, if its $Carry$ field is not -1, this slot stores an element. Then we try to search this element from AF_B. If $AF_A[i][j].Carry$ can be found in $AF_B[i]$ and these two slots have equal $Flag$ value, or i can be found in the bucket $AF_B[AF_A[i][j].Carry]$ and the $Flag$ of that slot is $\neg AF_A[i][j].Flag$, this element is a common element and will be removed from AF_A. After traversing the whole AF_A filter and emptying necessary slots, AF_A only contains the elements in $A - B$. Pair-wisely, by subtracting AF_A from AF_B, AF_B will only store the elements in $B - A$. The time complexity of such an operation is also $O(m \times b)$.

Algorithm 5 Subtracting AF_B From AF_A

Input: Two homogeneous Ark filters AF_A and AF_B

```
for i = 0 to m − 1 do
    for j = 0 to b − 1 do
        if AF_A[i][j].Carry ≠ −1 then
            if AF_A[i][j].Carry can be found in AF_B[i]
            and the Flag bit fits or i can be found in
            AF_B[AF_A[i][j].Carry] and the Flag bit fits
            then
                Delete the information of this element
                from the filter AF_A;
return AF_A;
```

Filter intersection. Intersection is also a fundamental operation between two given sets. For Ark filters AF_A and AF_B, intersection means to determine the common elements/fingerprints recorded. To this end, this algorithm just goes over a filter w.l.o.g, AF_A, which has fewer stored fingerprints. If a fingerprint η_t in AF_A cannot be found in AF_B, then η_t is removed from AF_A. After that, the algorithm returns the adjusted AF_A as the intersection, which only stores elements in $A \cap B$. The time complexity of this operation is $O(m \times b)$.

Ark filter can also support filter-level operations for heterogeneous Ark filters, if these filters share the same range of fingerprints and the same hash function. In this situation, Ark filters can deduce the original fingerprint of a represented element by jointly considering the *Carry*, *Flag*, and the *index* of the bucket:

$$\eta_t = \begin{cases} Carry \times m + index, & if\ Flag = 0 \\ index \times m + Carry, & if\ Flag = 1 \end{cases} \tag{4}$$

Then, when querying η_t against another Ark filter, the candidate buckets can be located with η_t directly. Therefore, the above compact, subtraction, and intersection can also be conditionally applied to heterogeneous Ark filters.

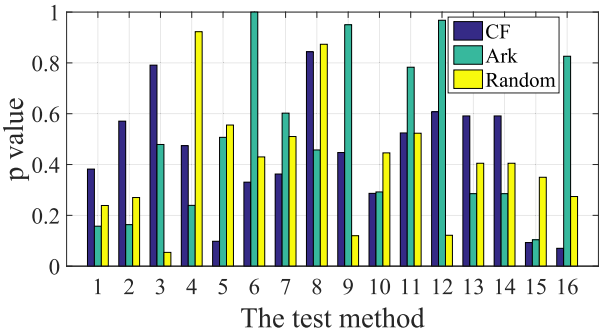


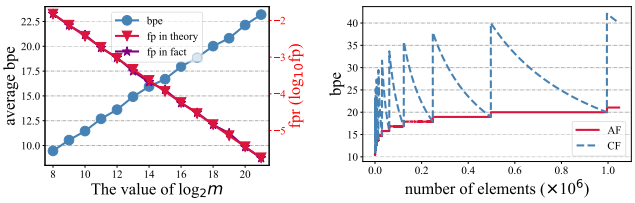
Fig. 2. The p values of random tests presented by NIST [34] at the level of 1%, when $m = 2^{15}$, $b = 2$, and $N = 50,432$. It is obvious that the Ark filter passes all the randomness tests simultaneously.

C. Performance Analysis

Randomness of candidate buckets. In effect, the randomness of candidate buckets is the core of exploring empty slots in the filters. Unlike Cuckoo filter, which selects the two candidate buckets with the *partial-key cuckoo hashing* strategy [33], Ark filter calculates the quotient or remainder as the indexes of its candidate buckets. Therefore, the Cuckoo filter requires two hash functions, one for fingerprint generation and another for candidate bucket selection. Ark filter, on the contrary, only relies on one single hash function to deduce the fingerprint and derive out the candidate buckets with the fingerprint. The randomness of fingerprint guarantees the randomness of the candidate buckets. We test the randomness of candidate buckets with the 16 methods[1] suggested by the NIST (National Institute of Standards and Technology) [34]. As shown in Fig. 2, we consider three different ways to calculate the candidate buckets, i.e., the *partial-key cuckoo hashing* in the cuckoo filter, the fingerprint-enabled method in Ark filter and two independent hash functions. According to the p values of these tests, we conclude that these methods pass all 16 random tests and generate random candidate buckets for elements.

Besides, we also note a special case of Ark filter wherein $Q_x = R_x$ for an element x. In this case, x can only be stored by this bucket. For a bucket with index i, this special case occurs iff $\eta_x = i \times m + i$. Therefore, for the whole filter, the probability that all of the N elements has exactly 2 diverse

[1]Frequency Test, frequency test within a block, runs test, test for the longest of Ones in a block, binary matrix rank test, discrete Fourier transform test, non-overlapping template matching test, overlapping template matching test, Maurer's universal statistical test, linear complexity test, serial test, approximate entropy test, cumulative sums (forward) test, cumulative sums (reverse) test, random excursions test, and random excursions variant test.



(a) Fpr when m varies.. (b) Bpe when $\overline{n}$ varies.

Fig. 3. The false positive rate (fpr) and bits per element (bpe) of Ark filter.

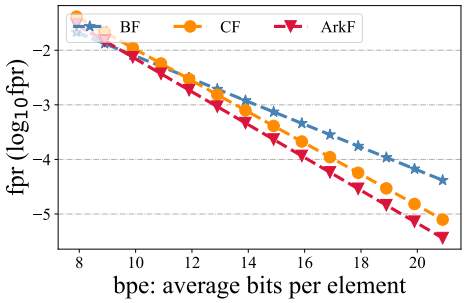


Fig. 4. The trade-off between bpe and fpr, where $b_{CF} = b_{AF} = 4$ and $\alpha_{CF} = \alpha_{AF} = 0.95$, respectively .

candidate buckets is:

$$\frac{\binom{m(m-1)-m}{N}}{\binom{m(m-1)}{N}} = \frac{\binom{m(m-2)}{N}}{\binom{m(m-1)}{N}}. \tag{5}$$

The same cases may also happen when the Cuckoo filter employs two independent hash functions. However, the *partial-key cuckoo hashing* strategy will not incur this trouble due to the *XOR* operation.

Considering the randomness of candidate buckets, the following conclusion introduced in [35] and [36] for Cuckoo filter still holds in the framework of Ark filter: given the number of elements to represent n and the number of buckets in the filter m, there is a threshold T such that when $\frac{n}{m} \leq T$, all the n elements can be successfully represented by the filter with probability $1-o(1)$; otherwise, the filter fails to represent all the elements with probability $1-o(1)$. Basically, a larger b results in a larger T, which implies a higher space utilization in the filter.

False positive rate (fpr). Ark filter employs a hash function to generate a fingerprint for an element x. Such collisions lead to potential false positive errors of membership query, i.e., regarding a non-member element as an element of the set. For instance, given two elements $x \in A$ and $y \notin A$ and x has been recorded by the filter, then the query of y against the filter may result in a false positive error. Considering that the candidate buckets of x are only decided by its fingerprint η_x, we can calculate the false positive rate of querying an element as:

$$\xi = \frac{\overline{n}}{m(m-1)}, \tag{6}$$

where m is the length of Ark filter and $\overline{n} \leq n$ denotes the number of fingerprints represented by the filter. Fig. 3(a) presents the theoretical and practical false positive rates of Ark filter when m increases from 2^8 to 2^{21} while $b = 4$. We remain the space utilization as 0.95 in this experiment. The false positive decreases consistently when m increases. Besides, the theoretical and practical fpr values match with each other well.

Bits per element (bpe). According to Fig. 3(a), the bpe of Ark filter increases when m grows since more bits are required to represent the stored quotient/remainder in each slot.

Theorem 1: Given the same b and $\overline{n}$, with the same false positive rate guarantee, the bpe of Ark filter is no more than that of Cuckoo filter.

Proof: This theorem can be derived out by letting Equ. 6 equal to Equ. 1 then calculating the bpe of Ark filter and Cuckoo filter. We omit the details here for space reasons. ■

As shown in Fig. 3(b), when the number of elements to represent increases constantly, the bpe of Ark filter is no more than that of Cuckoo filter. The initial reason is that the Cuckoo filter's filter length (i.e., m) can only be the form of 2^c, where c is a non-negative integer. As a result, the space of Cuckoo filter is always under-utilized. By contrast, in Ark filter, the overall space overhead can be more flexible and proportional to the number of elements to represent.

Parameter setting in Ark filter. Equ. 6 suggests that more buckets in the filter guarantee a lower false positive rate; by contrast, as stated in [35] and [36], better space utilization requires larger b which lowers the value of m in return. Therefore, there is a natural contradiction between space utilization and false positive rate in Ark filter. As declared in [35] and [36], the increase of b leads to marginal upgrade of space utilization. We note that when $b = 2$, the space utilization may reach about 90% ($T \approx 1.80$) with a high probability. Consequently, when the space is given, we remain $b = 2$ in Ark filter to ensure a large value of m, such that a lower false positive rate is generated; pair-wisely, when n is known previously, we calculate the length of Ark filter as $m = \frac{b \times n}{T}$, where $b = 2$ and $T \approx 1.80$, to save space and ensure the probability of successful representation. Note that the length of Ark filter can be set at will, while the length of a Cuckoo filter is only allowed to be 2^c ($c > 1$) to avoid overflow of *XOR* operations. This flexibility makes Ark filters more practical than Cuckoo filters.

Trade-off between bpe and fpr. For probabilistic data structures, there is a trade-off between bpe and fpr. Lower fpr can be generated with higher bpe. Here we derive out the relationship between bpe and fpr in Bloom filter, Cuckoo filter, and Ark filter, respectively. For Bloom filter, $\xi_{BF} \approx (1 - e^{-hn/m_{BF}})^h = (1 - e^{-h/bpe_{BF}})^h$, given the optimal number of hash functions $h_{opt} = \frac{m_{BF}}{n} \times \ln 2 = bpe_{BF} \times \ln 2$, we have:

$$\xi_{BF} = (1 - e^{-\ln 2})^{\ln 2 \times bpe_{BF}} \tag{7}$$

For Cuckoo filter, let α_{CF} be the allowed space utilization. We have $bpe_{CF} = \frac{l \times b_{CF} \times m_{CF}}{b_{CF} \times m_{CF} \times \alpha_{CF}} = \frac{l}{\alpha_{CF}}$; thus $l = bpe_{CF} \times \alpha_{CF}$. Then,

$$\xi_{CF} = \frac{2b}{2^l} = \frac{2b}{2^{bpe_{CF} \times \alpha_{CF}}}, \tag{8}$$

where l is the length of fingerprint and b_{CF} is the number of slots in each bucket. For Ark filter, $\xi_{AF} = \frac{\overline{n}}{m_{AF} \times (m_{AF}-1)} = \frac{\alpha_{AF} \times b_{AF}}{m_{AF}-1}$. While $bpe_{AF} = \lceil \log_2 m_{AF} \rceil + 1$, therefore, we can

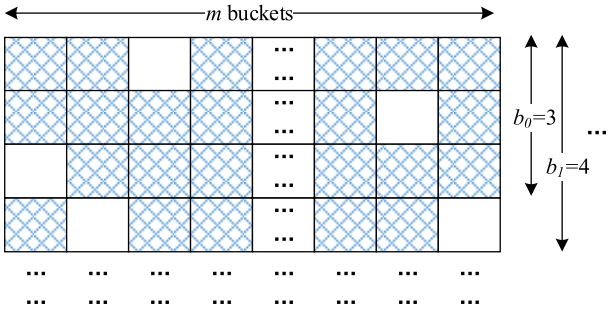


Fig. 5. The framework of DAF. All filters have the same length (m), yet the value of b is changed dynamically for instant capacity alteration.

quantify the relationship between bpe_{AF} and ξ_{AF} as:

$$bpe_{AF} = \lceil \log_2(\alpha_{AF} \times b_{AF}/\xi_{AF} + 1) \rceil + 1. \quad (9)$$

Fig.4 depicts the trade-off between bpe and fpr in Bloom, Cuckoo, and Ark. Bloom filter may have a bit lower fpr than Cuckoo filter and Ark filter when the bpe is small. By contrast, when bpe increases to some extent, Ark filter has the lowest fpr, while Bloom filter gets the highest fpr.

IV. Ark Filter Variants

Based on the above framework, we further redesign Ark filter for various network flow analysis jobs. Intuitively, the sketch data structure should be extended to represent dynamic sets. Besides membership queries, big data algorithms usually rely on sketch data structures to realize approximate yet fast cardinality estimation, frequency estimation, and top-k query. In effect, cardinality estimation is simple in Ark filter by maintaining a global counter. When a new element is inserted successfully, the global counter will be increased by 1. Once an element is removed from the filter, the global counter will be decreased by 1. With such a design, the global counter tells the number of elements represented by the filter precisely. Therefore, this section details how to augment Ark filter to achieve capacity elasticity, frequency query, and top-k query.

A. Ark Filter for Dynamics

The Ark filter introduced in Section III is only capable of static set representation. Despite the support of element deletion, an instant capacity scaling and recycling mechanism are missing in Ark filter.

To this end, we propose the Dynamic Ark filter (DAF) inspired by previous work like DCF [12] and DBF [9] which maintain multiple homogeneous filters so that the indexes of an element in the first filter are also applicable in others. Due to the dependency between element fingerprint and filter length, it is not advisable to launch filters with various lengths in DAF. For instance, if there are two filters with m_1 and m_2 ($m_1 \neq m_2$) buckets, respectively, to check the existence of element x, DAF has to calculate the respective fingerprints in the two filters by accessing the actual element content. By contrast, when $m_1 = m_2$, both the fingerprint and the indexes of candidate buckets can be multiplexed with only one access of the element content. Therefore, DAF prefers to adding Δb arrays of untapped slots while remaining m immutable.

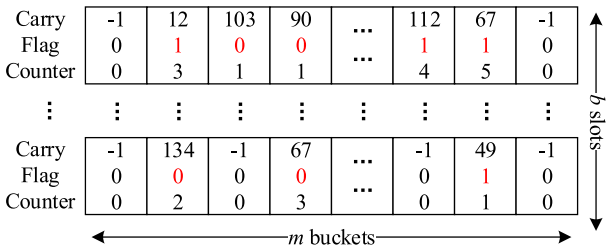


Fig. 6. The framework of Counting Ark filter. A counter field is added to each slot to explicitly count the frequency of the stored element.

At its core, DAF scales up by increasing the value of b adaptively. The number of added arrays Δb is decided by the expected number of future elements. According to the theory in [35] and [36], Δb can be set such that all the coming elements can be represented with high probability. If there are Δn more elements to represent, the value of Δb should be calculated as $\lceil \frac{\Delta n}{Tm} \rceil$, where T is the threshold given in [35] and [36]. A toy example of DAF is depicted in Fig. 5. The initial DAF has three arrays ($b_0 = 3$) and scales up to 4 arrays ($b_1 = 4$) to accommodate more elements. Note that the insertion of a coming element will be handled by the activated arrays (i.e., newly added arrays). To query the membership of x, DAF has to check all the slots in its candidate buckets: if both of them fail to find η_x, DAF returns *false* to show that x is not a member; otherwise, DAF returns *true*. The deletion of x will also check the two buckets and then try to delete x.

When a DAF is sparse enough, it is necessary to release the space for efficiency. A global counter C is implemented to record the number of elements in DAF. If $\frac{C}{m} \leq \bar{T}$, where $\bar{T}$ is the threshold when $b = \bar{b}$, then the number of arrays in DAF can be decreased to $\bar{b}$. To this end, DAF selects the $b - \bar{b}$ arrays that store the least number of elements and reinserts these elements into other arrays. DAF hasn't to access the real element contents for any element reinsertion since the candidate bucket indexes suit all the arrays. Specifically, for a bucket with no more than $\bar{b}$ fingerprints, the fingerprints in the removed slots are pushed to the $b - \bar{b}$ empty slots directly; otherwise, part of the fingerprints in this bucket has to be redirected to their alternative candidate buckets.

B. Ark Filter for Frequency Estimation

Given a set of elements, frequency estimation tells how many times an element x occurs in the set. For example, in network measurement, the selected routers have to estimate the undergoing flow size. Then, further analysis such as heavy changer, heavy hitter, and entropy estimation can be enabled. For a recommendation system, it is necessary to record or estimate how many times a user clicks on a specific product.

To this end, we propose Counting Ark filter (CAF), which extends the Ark filter by introducing a counter field into each slot to record the frequency information of the stored element explicitly. As depicted in Fig. 6, with such a design, each slot in CAF has three fields, i.e., the *Carry* field to store the fingerprint information, the *Flag* bit to mark the stored part of the fingerprint, and the *counter* field to counts the frequency of the represented element.

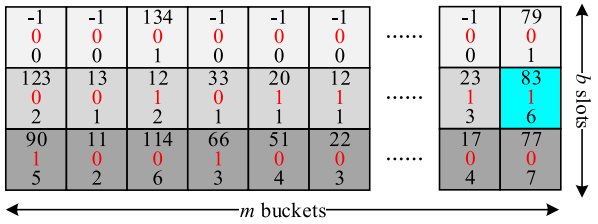


Fig. 7. A toy example of "partial rank" in an ascended CAF. Each bucket has 3 slots, and the elements represented by a bucket are sorted according to their counter values.

Note that operations in CAF can be slightly different from the original Ark filter. When inserting an element x, CAF still follows the "kick-out-and-reallocate" strategy. However, CAF has to additionally check whether x has been stored by its candidate buckets or not. If that commits, only the *counter* field in the accommodation slot will be added up by 1; otherwise, all three fields of the selected slot will be updated. To query the frequency of x, CAF traverses its two candidate buckets to locate its resident slot, then returns the *counter* field of that slot. If x cannot be found, CAF returns false. CAF offers two kinds of deletion operations, i.e., deleting a replica of x or trashing x. To delete a replica of x, CAF locates the fingerprint of x and then decreases the corresponding *counter* field by 1. If the *counter* gets down to 0, the *Carry* and *Flag* fields will also be updated as -1 and 0 respectively to remove x from the filter. By contrast, to trash x, CAF locates the element and then clears the slot by letting $Carry = -1$ and zerolizing both *counter* and *Flag*. Compared with Ark filter, CAF may lead to a bit more time to insert elements since it has to check the existence of the elements before storing them. However, query and deletion in CAF are still time-constant.

As for the filter-level operations, CAF still reserves the ability to compact, subtract and intersect two filters CAF_A and CAF_B. However, CAF has to handle the *counter* field when executing these operations. For an arbitrary element x, suppose CAF_A and CAF_B represent $f_A(x)$ and $f_B(x)$ replicas of x, respectively. In the compact result, the *counter* is calculated as $f_A(x) + f_B(x)$. To subtract CAF_B from CAF_A, the resultant CAF needs to record $f_A(x) - f_B(x)$ replicas of x. By contrast, after the intersection, the generated CAF only keeps $min\{f_A(x), f_B(x)\}$ replicas of x. Certainly, according to their practical needs, users can customize their own definitions of these filter-level operations and attach diverse calculations to the *counter*s. This design flexibility makes CAF general in many cases. For instance, one may regard $max\{f_A(x), f_B(x)\}$ as the *counter* value of x in the union result.

Besides membership query, CAF provides accurate frequency estimation once an element is represented correctly. The number of bits a counter needed is tricky. Too large counters lead to space waste, while small counters incur overflow risk. Basically, the length of a counter is directly determined by the element with the maximum frequency. It is possible to optimize the *counter* field by unequalizing the length of the counter in a bucket so that slots with lengthy counters can store elements with a large frequency, just like the TowerSketch [26], Stingy Sketch [27] do.

C. Ark Filter for Top-k Query

Top-k query returns the k highest ranked flows in a both quick and efficient fashion. We rely on the *counter* field in CAF to record the flow size. Based on CAF, there are two different methodologies to search the filter and find the top-k flows, i.e., *rank while query* and *rank before query*. A *rank while query* strategy ranks all the *counters* in the filter to derive out the top-k elements during the query. This brute-force method guarantees accurate query results and promises no additional space overhead; however, it can be impractical due to its time consumption. State-of-the-art sort algorithms, such as TimSort [37] and Quicksort [38], require $O(mb \times \log(mb))$ time-consumption to rank all the mb counters in the filter. On the contrary, a *rank before query* scheme maintains an extra max (min) heap, which ranks the elements according to their counter values dynamically [39]. The max (min) heap just returns its first (last) k elements to answer the query. This scheme naturally realizes precise and constant-time top-k queries; however, the introduced space overhead of such a method contradicts the design philosophy of sketch. Therefore, the above methods are either time-consuming or space-aggressive.

The sorted CAF. We present a conservative proposal that sorts the counters in a bucket and derives out the top k elements when querying based on the sorted buckets. We call this design as sorted CAF. Specifically, when an element is inserted or updated, its location in the resident bucket will be re-arranged according to its frequency. CAF stores the fingerprints inner a bucket in either an ascending or descending order. In a descended CAF, the first slot of each bucket stores the element with the maximum counter value, the second bucket accommodates the second maximum counter value, so on and so forth. On the contrary, the element with the maximum counter is stored in the last slot in an ascended CAF bucket. Without loss of generality, Fig. 7 presents a toy example of a sorted CAF. In this example, each bucket has three slots, and the elements in each bucket are sorted and stored ascendingly. Whenever the counter value changes, the bucket will re-sort all the elements it records. To answer the top-k query, the ascended CAF has to sort all the last slots in m buckets to derive out the k elements with the top counters.

Accuracy of sorted CAF. In reality, the sorted CAF may lead to incorrect top-k query results. As shown in Fig. 7, suppose the maximal counter is 7, then the element (whose counter is exactly 6) stored in the second slot of the last bucket will not be listed in the query result when $k \in [1, m]$. A naive solution is to search more slots in the filter to compare more elements before listing the results. Generally, let θ denote the number of searched slots in a bucket. Then we have the following Theorems to guarantee the query accuracy.

Theorem 2: In a sorted CAF with m buckets each of which has b slots, for a top-k query, the filter returns correct results when $k \leq \theta$.

Proof: Whether the sorted CAF returns a correct query result or not entirely depends on the distribution of the top-k elements in the filter. When $k \leq \theta$, even if all the top-k

TABLE I
QUALITATIVE COMPARISON AMONG STRATEGIES FOR TOP-k QUERY, IN TERMS OF ACCURACY, SPACE, AND TIME-COMPLEXITY

Methods	Accuracy	Extra space	Complexity
Rank while query	✓	✗	$O(mb\log(mb))$
Rank before query	✓	✓	$O(1)$
Sorted CAF	✗	✗	$O(m\log m)$

TABLE II
PARAMETER SETTING FOR THROUGHPUT TEST

Sketch	bpe	fpr	# realloca. *Max*	# hash
Bloom	23.11	1.5×10^{-5}	-	17
Cuckoo	20	1.5×10^{-5}	500	3
Quotient	20	1.5×10^{-5}	-	1
Ark	20	1.5×10^{-5}	500	1

elements are represented by a single bucket, they will still be searched by the filter. ■

Theorem 3: According to existing theory [40], when inserting the top-k elements into the filter, the asymptotic expected maximum number of elements a bucket will store is:

$$\Gamma^{-1}(m)\left(1+\frac{\ln(k/m)}{\ln\Gamma^{-1}(m)}+O\left(\frac{1}{\ln^2\Gamma^{-1}(m)}\right)\right). \quad (10)$$

Besides, when representing k elements with the sorted CAF, let λ be an integer variable and denote the number of elements stored by a bucket, then the probability that λ is greater or equal to $i\in[0,k]$ can be bounded as:

$$Pr(\max(\lambda)\geq i)\leq m\binom{k}{i}\frac{1}{m^i}\leq m\left(\frac{e\times k}{m\times i}\right)^i. \quad (11)$$

The proof is omitted. Please refer to [40] for detail.

Corollary 1: Based on Theorem 3, when $\theta\in[1,b-1]$, the probability p that the sorted CAF returns correct top-k query results can be calculated as:

$$\begin{aligned}p&=1-Pr(\max(\lambda)\geq\theta+1)\\&\geq 1-m\binom{k}{\theta+1}\frac{1}{m^{\theta+1}}\geq 1-m\left(\frac{ek}{m(\theta+1)}\right)^{\theta+1}.\end{aligned} \quad (12)$$

When $\theta=b$, $p=1$ since all the slots will be traversed and checked to search out the top k elements.

V. EVALUATION

In this section, we first implement Ark filter and compare it with typical sketches regarding the throughput of element insertion, query, and deletion. Thereafter, we quantify the performance of Ark filter variants for other big data analysis jobs, i.e., dynamic representation, frequency estimation, and top-k query. All experiments are conducted in a machine with an Intel Core i7 processor and 16GB DRAM. In our experiments, if no otherwise statement, the Bloom filter, Adaptive Bloom filter, Shifting Bloom filter and CM-sketch run the Murmurhash algorithm to generate the hash functions they needed, while quotient filter and Ark filter use the python built-in hash function. All results are the average values of 100 executions. All the codes are available at GitHub (https://github.com/fptjy/ArkFilter).

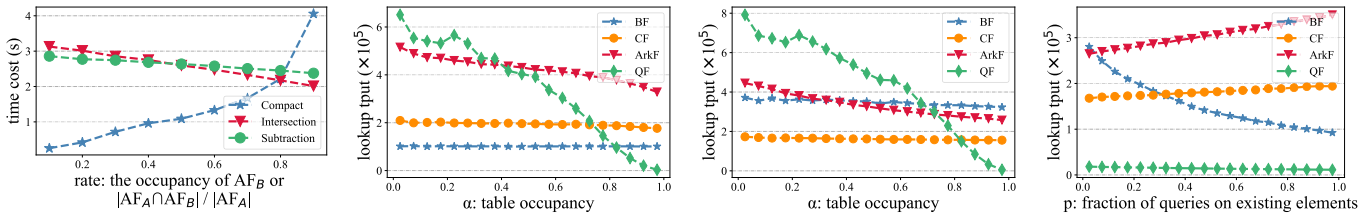
A. Performance of Ark Filter

Before quantifying the element-level performance (i.e., insertion, deletion, query), we first measure the filter-level performance, mainly in terms of the time consumption for compact, intersection, and subtraction. We initiate two filters AF_A and AF_B such that $m_A=m_B=2^{18}$ and $b_A=b_B=2^2$. For compact, the space occupation of AF_A is set as 0.1; then we increase the space utilization of AF_B from 0.1 to 0.9. As shown in Fig. 8(a), the resultant time consumption goes up constantly since more reallocations may be triggered when storing these elements with a single filter. As for intersection and subtraction, AF_A and AF_B have equal space utilization (0.95) yet the ratio of common elements (i.e., $|AF_B\cup AF_A|/|AF_A|$) ranges from 0.1 to 0.9. With more common elements, Ark filter can finish the intersection and subtraction operations faster.

In this section, we compare Ark filter with three typical sketches, including Bloom filter, Cuckoo filter, and the Quotient filter, in terms of the query, insertion, and deletion throughput. Note that these filters use hash functions to map the element directly, therefore, the element content has no impact on the filter performance. For simplicity, we feed these filters with random strings. We record the throughput of the filters at different space occupancy α (ranging from 0.025 to 0.975). We remain the overall false positive rate of the sketches as 1.5×10^{-5}. The default parameters of this experiment are summarized in Table. II.

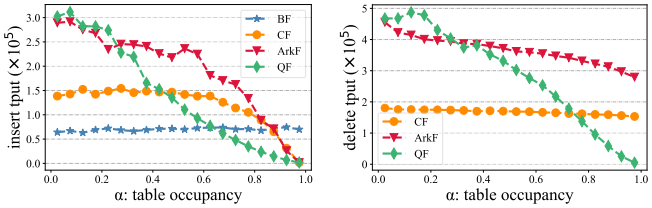
Then, we quantify the query throughput when the queried elements occupy 100% (positive), 0% (negative), and 0% to 100% (hybrid) of the represented elements. As shown in Fig. 8(b), when the filters hold an increasing number of elements, Cuckoo, Quotient, and Ark all lead to decreasing throughput for the positive queries. The reason is that these filters have to check more non-empty slots to conclude the existence of the queried elements when α increases. By contrast, the query throughput of Bloom filter remains stable around 0.99×10^5 irrespective of the increase of α, since it always fetches the corresponding h bits for any membership query. The Quotient filter experiences the most cliffy decrements from 6.57×10^5 to 0.38×10^5 entry/second. Note that the Quotient filter relies on the linear probing technique to resolve the hash collisions so that it has to go over a long "run" to search the target fingerprint when α grows. The curve of Cuckoo filter decreases slowly since it needs to check more non-empty slots but checking them takes just a little extra time. Ark filter experiences a significant throughput drop but remains high even when the filter is full. Compared with Cuckoo filter, Ark filter only checks the quotient or remainder part of the fingerprint, both of which are much shorter than a fingerprint. Besides, Ark filter only executes one hash computation. Consequently, Ark can outperform Cuckoo significantly.

For negative queries, as depicted in Fig. 8(c), all of the sketches show either dramatic (Quotient, Ark) or slight



(a) The filter-level performance. (b) The throughput of positive queries. (c) The throughput of negative queries. (d) The throughput of hybrid queries.

Fig. 8. The filter-level performance and the throughput of different types of queries.



(a) The throughput of insertion. (b) The throughput of deletion.

Fig. 9. The throughput of element insertion and deletion.

(Cuckoo, Bloom) throughput decrements when α grows. A common reason is that all of them need to check more non-empty/non-zero slots/bits to determine the membership of the queried elements. In this case, Bloom can significantly outperform Cuckoo because Bloom returns the negative results once a zero bit is found. The numerical results drop from 7.82×10^5 to 0.60×10^5 for Quotient filter. It is understandable since the "runs" in Quotient filter can be quite lengthy with a higher α. We note that when $\alpha \geq 0.45$, Ark filter has a bit lower throughput than Bloom filter. In effect, the sketches are usually highly utilized. Therefore, we believe the query throughput when $\alpha \geq 0.8$ is of great significance for real applications. From the above experiments, it is evident that Ark filter can significantly outperform others when the space utilization is high, thus committing the practicability of Ark.

Given $\alpha = 0.95$, we vary the fraction of positive elements p in the query set for each sketch (from 0.025 to 0.975), and then record the query throughput with Fig. 8(d). The query throughput of Bloom filter and Quotient filter decreases directly from 2.93×10^5 and 0.18×10^5 to 0.99×10^5 and 0.11×10^5, respectively. By contrast, both Cuckoo filter and Ark filter have significant increments of query throughput from 1.63×10^5 and 2.70×10^5 to 1.91×10^5 and 3.47×10^5, respectively. For Bloom filter, querying more stored elements means it has to check more non-zero bits before exit. Quotient filter checks the *is_occupied* bit of each cell first, thereby pruning unnecessary cell accesses and speeding up the negative queries. The Cuckoo filter and Ark filter return the query results once they find the corresponding fingerprints; thus, they have higher query throughput when p grows. Besides, Ark still outperforms others on a large scale in this experiment.

Fig. 9 indicates the element insertion and deletion throughput of those sketches when the space occupancy α grows gradually. As depicted in Fig. 9(a), the insertion throughput of Bloom filter remains constant while the other three sketches show dramatic declines when α grows. The reason is that they need to handle the overflow of bucket/cell when the filters get crowded. Specifically, Cuckoo and Ark have to execute the *kick-out-and-reallocate* process for overflowed buckets, while Quotient filter must probe an empty cell for the coming fingerprint linearly. Ark filter always has a higher insertion throughput than Quotient and Cuckoo since it only executes one hash computation for each element insertion. Besides, Bloom filter fails to enable element deletion; thus, we quantify the deletion throughput of Ark, Cuckoo, and Quotient in Fig. 9(b). These sketches experience instant deletion throughput drop when α grows. That is a natural result of searching more slots/cells before deleting the fingerprint. Especially, Quotient filter decreases from 5.47×10^5 to 0.54×10^5, making it impractical for real implementation. By contrast, Ark filter outperforms Cuckoo filter consistently.

TABLE III
HASH FUNCTIONS USED IN EXPERIMENTS

Purpose	Ark	CF-F	CF-D	CF-P	CF-M
Fingerprint	built-in	FNV64	DJBhash	PJWhash	Murmurhash
Candidates	–	built-in	built-in	built-in	built-in

Ark filter and cuckoo filter have different mechanisms to decide the candidate buckets of each element. Fig. 10 further quantifies the impact of such difference by recording the time consumption of fingerprint generation and candidate bucket determination for an element. The hash functions used in this experiment (including python built-in hash function and several general hash functions) are listed in Table. III. As shown in Fig.10, to represent more elements, both Ark filter and Cuckoo filter need more time to derive out their candidate buckets. Obviously, Ark filter requires the least time compared Cuckoo filters. This experiment quantitatively indicates Ark filter is able to save time by multiplexing the fingerprint information with the candidate bucket index information.

Putting the above results together, compared with Cuckoo filter, Ark filter has $2.08\times$, $1.34\times$, and $1.68\times$ throughput of deletion, insertion, and hybrid query, respectively; compared with Quotient filter, Ark filter has $4.55\times$, $1.74\times$, and $22.12\times$ throughput of deletion, insertion and hybrid query, respectively; compared with Bloom filter, Ark filter has $2.55\times$ and $2.11\times$ throughput of insertion and hybrid query, respectively. By the above implementation and experiments, we conclude that Ark filter outperforms state-of-the-art sketches significantly by enabling faster element insertion, query, and deletion.

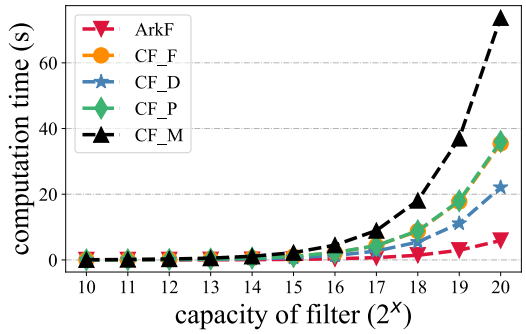
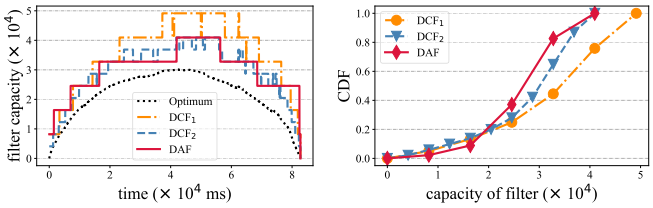


Fig. 10. The time of fingerprint generation and candidate bucket determination.



(a) The filter capacities. (b) The CDF of filter capacities.

Fig. 11. The capacity elasticity of filters for dynamic set representation.

TABLE IV
BPE OF DAF AND DCF AT PEAK TIME

Sketches	DAF	DCF_1	DCF_2
bits per element	19.05	26.13	23.13

B. The Merit of Ark Filter Variants

Here we quantify the performance of the variants of Ark filter for dynamic set representation, frequency estimation, and top$-k$ query, respectively.

1) Quantification of Capacity Elasticity: Dynamic Ark filter (DAF) adjusts its capacity by adding or removing slots from every bucket. The most relevant design is Dynamic cuckoo filter [12] which includes (excludes) untapped (underutilized) Cuckoo filters on demand. In this section, we compare such two typical designs for capacity elasticity with real-world network traces. We choose traffic trace collected by the MAWI group [41] from May 20, 14:00:00, 2021 to May 20, 14:01:25, 2021. There are 7,220,238 IPv4 packets. We use src.IP, dst.IP, src.port, dst.port and protocol in each packet as the key and identify 1,082,109 flows in total. Besides, our tracking results show that the maximum size of this traffic is 30,101 at the peak time. We represent such flows with DAF and DCF and record their capacities at each millisecond. We set $m = 8,192$ for DAF, therefore the value of b should be no less than 4 to accommodate the flows at peak time and the associated fpr is about 4.8×10^{-4}. With such observation, we deploy two versions of DCFs with $b = 4$. Specifically, "DCF_1" has 2,048 buckets while "DCF_2" remains 1,024 buckets.

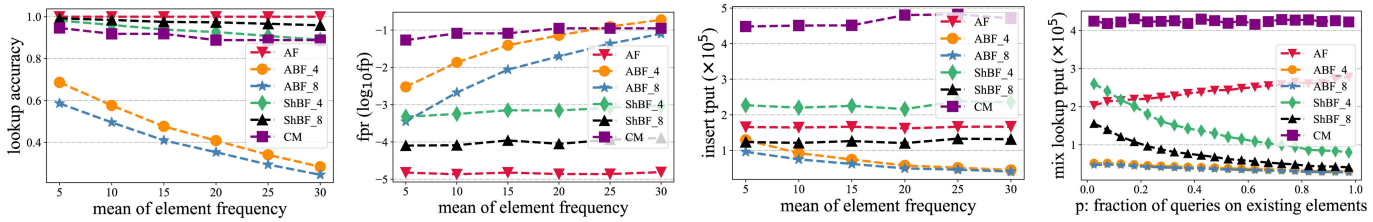
As shown in Fig. 11(a), the optimum capacity (the least number of slots) for such trace changes with time (increase first and then drop constantly after the peak). DAF, "DCF_1", and "DCF_2" all try to scale up or scale down to fit the real capacity demand. "DCF_1" costs the highest capacity than others since its filters are two long and only coarse-grained elasticity is achieved. To represent a small number of flows, the value of b should be no less than 1, while DCF may change the number of filters to fit the flows. That explains why DAF performs no better than DCF at the beginning and ending phases. Fig. 11(b) further presents the CDF of the resultant capacities. DAF and "DCF_2" have the same maximum capacity. However, DAF needs much less capacity to represent 80% of the flows than "DCF_2". The reason is that DAF has a larger space utilization than its competitors on the whole; when b gets larger, its space utilization rises accordingly [35], [36]. Moreover, as shown in Table. IV, DAF saves 27.1% and 17.6% bpe than "DCF_1" and "DCF_2", respectively. In other words, DAF guarantees comparable capacity elasticity as DCF while consuming much less space overhead.

2) Performance of Frequency Estimation: We further evaluate the performance of frequency estimation when Ark filter, Shifting Bloom filter (SBF) [25], Adaptive Bloom filter (ABF) [24], and CM-sketch [7] are adopted. The quantified metrics include accuracy, false positive rate, insertion throughput, and query throughput. We feed these filters with synthetic datasets wherein the data frequency follows a normal distribution. We implement two versions of SBF, as well as ABF, when the number of hash functions for membership representation is set as 4 (SBF_4 in Fig. 12) and 8 (SBF_8 in Fig. 12), respectively. The involved sketches have the same space overhead in our experiments. Generally, ABF and SBF determine the membership of the queried element first before checking the thereafter bits for frequency.

As shown in Fig. 12(a), when the average frequency $\overline{f}$ increases from 5 to 30, both SBF and ABF show significant accuracy drops. By contrast, our Ark filter guarantees 100% frequency estimation accuracy. The essential reason is that Ark filter records the exact frequency directly, while ABF and SBF have to check the non-zero bits to estimate the frequency. Besides, ABF realizes lower accuracy than SBF since it may set more bits to 1 when representing the multiplicity information. For instance, for an element whose frequency is 10, ABF must set 10 bits $h_1(x), \cdots, h_{10}(x)$ to 1 besides the q bits for membership. By contrast, SBF only set the later 10^{th} bit of $h_1(x), \cdots, h_k(x)$. The value of k can be smaller than 10. Consequently, the non-zero bits set by other elements may result in overestimating frequency more easily in ABF. Moreover, using more hash functions to record the existence information of elements can improve the query accuracy by pruning more false positives. As for CM-sketch, it has better accuracy than ABF yet lower accuracy than SBF and Ark filter.

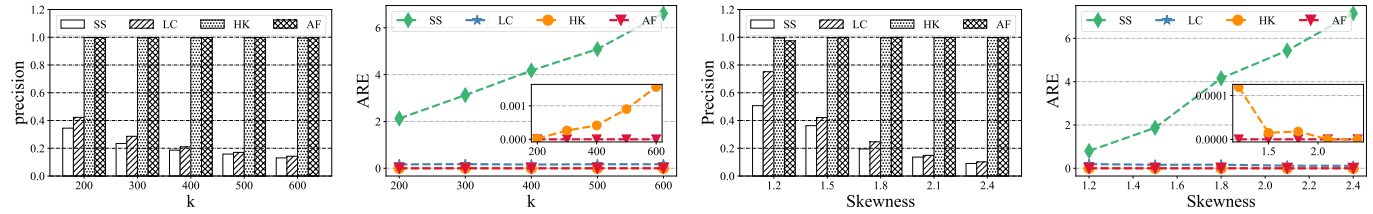
Fig. 12(b) demonstrates the specific false positive rates (fpr) of Ark filter, ABF, SBF, and CM-sketch. Ark filter guarantees no more than 1×10^{-5} fpr, which is orders lower than others. CM-sketch has the highest fpr since it can always overestimate the frequency. For ABF, the fpr is much higher than SBF and Ark filter, and goes wild when frequency increases. ABF sets later $\overline{f}$ bits of the f hash bits to 1 to record a frequency. Thus the increase of $\overline{f}$ leads to more false positives. For SBF, when $\overline{f}$ grows, the filter has to check more bits before reaching a conclusion.

As depicted by Fig. 12(c), Ark filter realizes a high and stable insertion throughput (more than 1.6×10^5). SBF



(a) Frequency estimation accuracy. (b) False positive of the queries. (c) Insertion throughput. (d) Query throughput.

Fig. 12. The frequency estimation performance of filters.



(a) Top$-k$ query accuracy. (b) ARE of Top$-k$ query. (c) Top$-k$ query accuracy. (d) ARE of Top$-k$ query.

Fig. 13. The Top$-k$ estimation performance of filters.

represents any element's membership with the f hash bits and its frequency with the affiliation bits. Therefore the increase of element frequency can barely impact its insertion speed. By contrast, more hash functions (i.e., larger f) slow down element insertions significantly. Unlike Ark filter and SBF, the insertion speed of ABF becomes slow when $\overline{f}$ increases since ABF has to check and set more bits to 1 to record the growing frequencies. Using more hash functions also brings more insertion time for ABF. Besides, CM-sketch realizes the best insertion throughput. The reason is that it just increases the corresponding counters. Such operations need less memory access, therefore can be quite fast.

We have also quantified the query throughput upon sets with a diverse fraction of recorded elements (i.e., r). As shown in Fig. 12(d), the query throughput of Ark filter remains high and keeps increasing when r grows. Ark filter has to check all the $2b$ slots to determine the non-existed elements but may find a stored element fingerprint before going over all the $2b$ slots. On the contrary, ABF and SBF need more time to decide the existence and frequency of a given element. ABF performs the worst since it checks more bits for each query. With more hash functions, the query throughput also decreases for both ABF and SBF. Still, CM-sketch has the highest query throughput because it just checks the corresponding counters for each query, while others have to check more bits or fields.

In brief, Ark filter outperforms others and guarantees accurate and fast frequency query simultaneously and a high level of insertion speed. SBF and ABF need to trade-off the accuracy speed by leveraging the number of hash functions. Besides, the accuracy and speed of SBF and ABF can also be significantly impacted by element frequency. Moreover, Ark filter supports online frequency statistics, while ABF and SBF must know the precise frequency before recording it, thus falling short of representing dynamic or stream datasets. As for CM-sketch, it has great throughput yet suffers from high fpr and low accuracy.

3) Accuracy of Top-k Queries: In this subsection, we compare Ark filter with the existing methods for top-k queries, including Lossy Counting (LC) [30], Space-Saving (SS) [31], and HeavyKeeper (HK) [32]. We quantify the query accuracy and average relative error (ARE) when representing a synthetic stream dataset with the same and fixed space overhead. Just like mentioned by HeavyKeeper [32], we suppose the frequency of flows in the dataset follows a Zipf distribution with a specific skewness. Here, we vary the skewness and value of k to explore their impact on these methods. For experiments of varying k, we set the skewness to 1.5. For experiments with varied skewness, we set the $k = 200$.

As shown in Fig. 13(a) and (b), the top-k query accuracy of Ark filter and HeavyKeeper have near 1.0 accuracy and 0.0 ARE, irrespective of the increment of k. Moreover, the ARE of Ark filter remains at 0 while other methods increase with the growth of k. In effect, the sorted counting Ark filter records each arrived flow precisely and searches the last several slots of each bucket to determine the top-k flows. In this experiment, there are in total 2^{14} buckets whose last two slots are searched. Therefore, Ark filter can respond to the top-k query with high accuracy and export the exact size of top-k flows. HeavyKeeper also performs well because it intelligently omits mouse flows and focuses on recording the elephant ones using an *exponential weakening* decay strategy. By contrast, the accuracy of Space-saving and Lossy Counting strategies remains low when k is small and gets even lower when k grows. Space-saving and Lossy Counting have a much higher ARE than Ark filter and HeavyKeeper. That happens for two main reasons: 1) as the typical *admit-all-count-some* strategies, Space-saving and Lossy Counting only store part of the flows and expel the smallest ones to make room for the new-comings. Such a design causes significant errors, especially when the memory is limited; 2) the difference of flow frequencies among flows gets smaller as k grows from 200 to 600, making it easy for Space-saving and Lossy Counting to regard others as top-k flows.

Fig. 13(c) and (d) show the experiment results when varying the skewness. The precision of both Space-saving and Lossy Counting perform much worse than Ark filter and

HeavyKeeper and decreases dramatically when the dataset is highly skewed. By contrast, Ark filter and HeavyKeeper realize near-accurate top-k queries. When the dataset is highly skewed, some elements whose frequencies are not too high are also lying in the top-k community. However, such elements may be pruned by Space-Saving or Lossy counting, resulting inaccurate query results.

In summary, Ark filter outperforms its same-kinds such as Cuckoo filter, Bloom filter, and Quotient filter, in terms of insertion and query. Its variants also support dynamic set representation, frequency estimation, and top-k query elegantly with comparable or better performance than their competitors.

VI. Discussion

We further discuss several essential issues about the Ark filter as follows.

The dependency between fingerprint length and Ark filter length. In Ark filter, given the filter length m, the fingerprint of any element must lie in the range $[0, m(m-1)]$. Such a tight dependency enables the multiplexing of the fingerprint information and the candidate bucket information. Therefore, we can deduce the indices of candidate buckets directly based on the fingerprint and filter length (using the quotient or remainder). However, this dependency also limits the design flexibility of Ark filter. A possible solution for this problem is to divide the filter into multiple segments. With such a design, the dependency is limited within each segment, reducing the bpe cost and guaranteeing better flexibility consequently.

The use of flag bit. In Ark filter, the flag bit is implemented to explicitly indicate the carry field stores a quotient (remaining the bit as 0) or a remainder (setting the bit as 1). For a query, the Ark filter will check whether the bucket indexed by the quotient stores the remainder or vice versa. This is helpful when the filter responds to a query. We also note that it is possible to remove the flag bit from the filter for higher query and insertion throughput. However, this alteration will undoubtedly double the false positive rate of Ark filter, since the filter cannot distinguish the quotient part from the remainder part. For better accuracy, the flag bit is suggested; for higher throughput, the flag bit can be removed. Such a choice is left to the users.

Future work. The future work of this paper is mainly two-fold. First, more variants can be designed to further enrich the functionality of Ark filter. This paper details how to support dynamic representation, multiplicity estimation, and top-k estimation, while additional data analysis jobs such as affiliation query, weighted set representation, element decay, etc. Second, the performance of Ark filter can be improved with additional alterations. Besides the mentioned redesigns about the flag bit and segment above, the counter fields in CAF and sorted CAF can be optimized to save space. Moreover, cross-checking [42], bit resetting [43], and hash remapping [44] techniques can also be employed to decrease the false positive rate of Ark filter.

VII. Conclusion

This paper presents the Ark filter design for set representation with the ambition of higher throughput, better flexibility, and more functionalities. At its core, Ark filter relies on the fingerprint of each element to index its candidate buckets such that no hash calculations nor *XOR* operation is needed during reallocations. For any element x, its quotient and the remainder upon the filter length are calculated to index its candidate buckets. Then multiple variants of Ark filter are proposed to enrich its functionalities. Theoretical analysis indicates that, given the same false positive rate, the bpe of Ark is no more than Cuckoo. Besides, comprehensive experiments commit that, Ark outperforms Bloom, Cuckoo, and Quotient significantly. Besides, the proposed variants support dynamic set representation, frequency estimation, and top-k query elegantly with comparable or better performance than their competitors.

Acknowledgment

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

References

[1] B. Li, J. Springer, G. Bebis, and M. H. Gunes, "A survey of network flow applications," *J. Netw. Comput. Appl.*, vol. 36, pp. 567–581, Jan. 2013.

[2] 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.

[3] S. Geravand and M. Ahmadi, "Bloom filter applications in network security: A state-of-the-art survey," *Comput. Netw.*, vol. 57, no. 18, pp. 4047–4064, Dec. 2013.

[4] A. Broder and M. Mitzenmacher, "Network applications of Bloom filters: A survey," *Internet Math.*, vol. 1, no. 4, pp. 485–509, 2004.

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

[6] B. Fan, D. Andersen, M. Kaminsky, and M. Mitzenmacher, "Cuckoo filter: Practically better than Bloom," in *Proc. ACM CoNEXT*, 2014, pp. 75–88.

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

[8] M. Bender et al., "Don't thrash: how to cache your hash on flash," in *Proc. USENIX HotStorage,* 2011, pp. 1–5.

[9] 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.

[10] 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.

[11] M. T. Goodrich and M. Mitzenmacher, "Invertible Bloom lookup tables," in *Proc. 49th Annu. Allerton Conf. Commun., Control, Comput. (Allerton)*, Sep. 2011, pp. 792–799.

[12] H. Chen, L. Liao, H. Jin, and J. Wu, "The dynamic cuckoo filter," in *Proc. IEEE 25th Int. Conf. Netw. Protocols (ICNP)*, Oct. 2017, pp. 1–10.

[13] 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.

[14] 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.

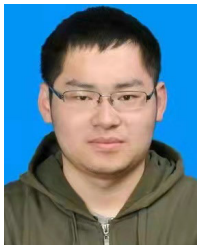
[15] D. Eppstein, M. T. Goodrich, F. Uyeda, and G. Varghese, "What's the difference? Efficient set reconciliation without prior context," in *Proc. ACM SIGCOMM*, 2011, pp. 218–229.

[16] M. Mitzenmacher, S. Pontarelli, and P. Reviriego, "Adaptive cuckoo filters," in *Proc. SIAM ALENEX*, New Orleans, LA, USA, 2018, pp. 1–16.

[17] 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)*, Jul. 2021, pp. 57–67.
[18] D. Eppstein, "Cuckoo filter: Simplification and analysis," 2016, *arXiv:1604.06067*.
[19] L. Luo, D. Guo, O. Rottenstreich, R. T. B. Ma, X. Luo, and B. Ren, "The consistent cuckoo filter," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, Apr. 2019, pp. 712–720.
[20] A. D. Breslow and N. S. Jayasena, "Morton filters: Faster, space-efficient cuckoo filters via biasing, compression, and decoupled logical sparsity," *Proc. VLDB Endowment*, vol. 11, no. 9, pp. 1041–1055, 2018.
[21] M. Wang, M. Zhou, S. Shi, and C. Qian, "Vacuum filters: More space-efficient and faster replacement for Bloom and cuckoo filters," *Proc. VLDB Endowment*, vol. 12. no. 2, pp. 197–210, 2019.
[22] P. Pandey, M. Bender, R. Johnson, and R. Patro, "A general-purpose counting filter: Making every bit count," in *Proc. ACM ICMD*, 2017, pp. 775–787.
[23] F. Deng and D. Rafiei, "New estimation algorithms for streaming data: Count-min can do more," *Webdocs. Cs. Ualberta. Ca*, 2007. [Online]. Available: http://www.cs.ualberta.ca/~fandeng/paper/cmm.pdf
[24] 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.
[25] T. Yang, A. Liu, M. Shahzad, Y. Zhong, Q. Fu, and Z. Li, "A shifting Bloom filter framework for set queries," in *Proc. IEEE VLDB*, Sep. 2016, pp. 408–419.
[26] K. Yang et al., "SketchINT: Empowering INT with towersketch for per-flow per-switch measurement," in *Proc. IEEE 29th Int. Conf. Netw. Protocols (ICNP)*, Nov. 2021, pp. 1–12.
[27] H. Li, Q. Chen, Y. Zheng, T. Yang, and B. Cui, "Stingy sketch: A sketch framework for accurate and fast frequency estimation," *Proc. VLDB Endowment*, vol. 15, no. 7, pp. 1426–1438, 2022.
[28] Y. Zhou et al., "Cold filter: A meta-framework for faster and more accurate stream processing," in *Proc. SIGMOD/PODS*, 2018, pp. 741–756.
[29] Y. Zhang et al., "CocoSketch: High-performance sketch-based measurement over arbitrary partial key query," in *Proc. ACM SIGCOMM*, 2021, pp. 207–222.
[30] G. S. Manku and R. Motwani, "Approximate frequency counts over data streams," in *Proc. 28th Int. Conf. Very Large Databases (VLDB)*, 2002, pp. 346–357.
[31] A. Metwally, D. Agrawal, and A. E. Abbadi, "Efficient computation of frequent and top-k elements in data streams," in *Proc. ICDT*, 2005, pp. 398–412.
[32] T. Yang et al., "HeavyKeeper: An accurate algorithm for finding top-k elephant flows," *IEEE/ACM Trans. Netw.*, vol. 27, no. 5, pp. 1845–1858, Oct. 2019.
[33] B. Fan, D. G. Andersen, and M. Kaminsky, "MemC3: Compact and concurrent MemCache with dumber caching and smarter hashing," in *Proc. USENIX NSDI*, 2013, pp. 1–14.
[34] A. Rukhin et al., "A statistical test suite for random and pseudorandom number generators for cryptographic applications," *Appl. Phys. Lett.*, vol. 22, no. 7, pp. 179–1645, 2010.
[35] M. Dietzfelbinger, A. Goerdt, M. Mitzenmacher, A. Montanari, R. Pagh, and M. Rink, "Tight thresholds for cuckoo hashing via XORSAT," in *Proc. ICALP*, 2010, pp. 213–225.
[36] N. Fountoulakis, M. Khosla, and K. Panagiotou, "The multiple-orientability thresholds for random hypergraphs," in *Proc. 22nd Annu. ACM-SIAM Symp. Discrete Algorithms*, Jan. 2011, pp. 1222–1236.
[37] N. Auger, C. Nicaud, and C. Pivoteau. (2015). *Merge Strategies: From Merge Sort to Timsort*. [Online]. Available: https://hal-upec-upem.archives-ouvertes.fr/hal-01212839
[38] C. Hoare, "Quicksort," *Comput. J.*, vol. 5, no. 1, pp. 1–16, 1962.
[39] X. Yu, H. Xu, D. Yao, H. Wang, and L. Huang, "CountMax: A lightweight and cooperative sketch measurement for software-defined networks," *IEEE/ACM Trans. Netw.*, vol. 26, no. 6, pp. 2774–2786, Dec. 2018.
[40] G. H. Gonnet and R. Baeza-Yates, *Handbook of Algorithms and Data Structures: In Pascal and C*. 1991.
[41] WIDE MAWI WorkingGroup. *Traffic Trace Info*. Accessed: Apr. 6, 2021. [Online]. Available: https://mawi.wide.ad.jp/mawi/samplepoint-F/2021/202105201400.html
[42] H. Lim, N. Lee, J. Lee, and C. Yim, "Reducing false positives of a Bloom filter using cross-checking Bloom filters," *Appl. Math. Inf. Sci.*, vol. 8, no. 4, pp. 1865–1877, 2014.
[43] K. Huang, J. Zhang, D. Zhang, and G. Xie, "A multi-partitioning approach to building fast and accurate counting Bloom filters," in *Proc. IEEE 27th Int. Symp. Parallel Distrib. Process. (IPDPS)*, Cambridge, MA, USA, May 2013, pp. 1159–1170.
[44] S. Z. Kiss, E. Hosszu, J. Tapolcai, L. Ronyai, and O. Rottenstreich, "Bloom filter with a false positive free zone," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, Honolulu, HI, USA, Apr. 2018, pp. 1412–1420.

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 Researcher with the School of Systems, National University of Defense Technology. His research interests include data structure and distributed networking systems.

Pengtao Fu received the bachelor's and master's degrees from the National University of Defense Technology, China, in 2020 and 2022, respectively. His research interests include data structure and network measurement.



Shangsen Li received the B.S. degree in automation from Northeastern University, Shenyang, China, in 2019, and the M.S. degree in management science and engineering from the 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.

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.

Qianzhen Zhang received the Ph.D. degree from the National University of Defense Technology, China, in 2022. He is currently a Lecturer with the College of Systems Engineering, National University of Defense Technology. His research interests include continuous subgraph matching, graph data analytics, and knowledge graph.

Huaimin Wang received the Ph.D. degree in computer science from the National University of Defense Technology (NUDT) in 1992. He has published more than 100 research papers in peer-reviewed international conferences and journals. His current research interests include middleware, software agent, and trustworthy computing. He has been awarded the "Chang Jiang Scholars Program" Professor and the Distinct Young Scholar.