Cuckoo Counter: Adaptive Structure of Counters for Accurate Frequency and Top-k Estimation

Qilong Shi, Yuchen Xu, Jiuhua Qi, Wenjun Li, Tong Yang, Yang Xu, and Yi Wang

***Abstract*— Frequency estimation and top-k flows identification are fundamental problems in network traffic measurement. Sketch, as a basic probabilistic data structure, has been extensively investigated and used in different management applications. However, few of them is suitable for both estimating frequency and finding top-k flows due to the unbalanced distribution of real-world network streams. By introducing a pre-filtering stage to isolate elephant and mice flows, the recently proposed Augmented Sketch (ASketch) significantly improves accuracy for both tasks. However, it suffers from serious performance degradation because of frequent flow exchanges. In this paper, we propose Cuckoo Counter (CC), an adaptive structure that consists of several buckets organized in a specific way. The size of the entry in each bucket is carefully designed to match the actual distribution of streams. During processing, CC hashes a flow to buckets and uses the idea of cuckoo hashing to relocate the flow if an overflow or collision happens, which contributes to fully utilizing memory. Therefore, the replacement strategy helps CC precisely record elephant flows and cover more mice flows, and also guarantees the throughput. Extensive experimental results show that CC has the highest (Freq.) accuracy, excellent (Heavy hitter / change) accuracy, highest (Top-k) precision, and competitive throughput compared to the state-of-the-art. Specifically, CC improves the throughput and accuracy by around 1 and 2 orders of magnitude respectively compared to the well-known ASketch.**

***Index Terms*— Network measurement, frequency estimation, top-k, sketch.**

Manuscript received 23 June 2021; revised 21 April 2022 and 16 September 2022; accepted 7 December 2022; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor V. Subramanian. This work was supported in part by the Key-Area Research and Development Program of Guangdong Province under Grant 2021B0101400001; in part by the National Key Research and Development Program of China under Grant 2022ZD0115303 and Grant 2020YFB1806400; in part by the Major Key Project of Peng Cheng Laboratory under Grant PCL2021A02; in part by the National Natural Science Foundation of China under Grant 62102203, Grant 62150610497, and Grant 62172108; in part by the Basic Research Enhancement Program of China under Grant 2021-JCJQ-JJ-0483; in part by the China Post-Doctoral Science Foundation under Grant 2020TQ0158 and Grant 2020M682825; in part by the International Post-Doctoral Exchange Fellowship Program of China under Grant PC2021037; and in part by the Open Research Projects of Zhejiang Laboratory under Grant 2022QA0AB07. This paper was presented in part at the ACM/IEEE ANCS, Cambridge, U.K., 25 September 2019 [1] [DOI: 10.1109/ANCS.2019.8901891]. *(Qilong Shi, Yuchen Xu, and Jiuhua Qi contributed equally to this work.) (Corresponding authors: Wenjun Li; Tong Yang.)*

Qilong Shi and Yuchen Xu are with the Department of Computer Science and Technology, Peking University, Beijing 100871, China (e-mail: stallone@pku.edu.cn; xu.yuchen@pku.edu.cn).

Jiuhua Qi is with Sangfor Technologies, Shenzhen 518055, China (e-mail: jiuhuaqi@pku.edu.cn).

Wenjun Li is with the Peng Cheng Laboratory, Shenzhen 518055, China, and also with the School of Engineering and Applied Sciences, Harvard University, Allston, MA 02134 USA (e-mail: wenjunli@seas.harvard.edu).

Tong Yang is with the Department of Computer Science and Technology, Peking University, Beijing 100871, China, and also with the Peng Cheng Laboratory, Shenzhen 518055, China (e-mail: yang.tong@pku.edu.cn).

Yang Xu is with the School of Computer Science, Fudan University, Shanghai 200433, China, and also with the Peng Cheng Laboratory, Shenzhen 518055, China (e-mail: xuy@fudan.edu.cn).

Yi Wang is with the Institute of Future Networks, Southern University of Science and Technology, Shenzhen 518055, China, and also with the Peng Cheng Laboratory, Shenzhen 518055, China (e-mail: wangy37@sustech.edu.cn).

This article has supplementary downloadable material available at https://doi.org/10.1109/TNET.2022.3232098, provided by the authors.

Digital Object Identifier 10.1109/TNET.2022.3232098

I. INTRODUCTION

NETWORK measurements play a central role in computer networks, such as network telemetry [2], [3], anomaly detection [4], [5], [6], [7], capacity planing [8], and caching of forwarding table entries [9], [10]. A real network data stream consists of a sequence of packets, each of which has an ID (we usually use five-tuple as the ID of a network flow: source IP address, destination IP address, source port number, destination port number, and protocol type). We use the term *flow* to represent packets with the same ID. Normally, frequency estimation for each flow is considered the most basic part of network measurements since various applications are centered on it. Meanwhile, the high-frequent flows are usually more relevant in these applications [11]. Many management applications can benefit from a function that can find them efficiently, such as congestion control by dynamically scheduling elephant flows [12], network capacity planning [12], anomaly detection [13]. For instance, filtering out the top-k flows contributes to identifying attackers in DDoS defense [14]. Therefore, from another aspect, it is more important to estimate the frequency of the most frequent flows and find them out as accurately as possible. In this paper, we mainly *focus on the approximate algorithms used for frequency estimation and top-k flows identification in a real network data stream*.

In those applications mentioned above, the network stream is generated at a high rate [15], [16], long term and only one-pass, thus the ordinary large-memory scheme is not affordable or outweighs its benefits. Thus, many probabilistic algorithms that store data temporarily in limited memory are subsequently invented [16], [17], [18], [19], [20]. $Sampling$ [21] is a memory-saving scheme that was proposed very early, but its effectiveness and error bound are hard to guarantee. Some variants of $Bloom\ filter$ [22], [23] can be used to estimate the frequency of flow in a data stream, but they are not specialized in it. In comparison, $sketch$ and $counter$ gain wide acceptance due to their better performance guarantees [24], [25], [26], [27], [28], [29], [30], [31].

Sketch-based structures usually provide an approximate estimation of every flow. In the past, many well-known works were invented (e.g., Count sketch [26], CM sketch [25],

CU sketch [24], Elastic sketch [32]), and they usually consist of many reusable fixed-size counters and several hash functions, making them support $O(d)$ time insertion and query (d is the number of hashes and usually less than 5). Unfortunately, the sketches mentioned above are not well adapted to the real network stream, whose flows' frequency are often highly skewed [17]. In other words, the majority of flows are mice (i.e., have a low frequency), while a few are elephant (i.e., have a high frequency). So they have to allocate enough bits for each counter, thus wasting lots of memory in real scenarios. The Pyramid sketch [33], which is proposed recently to specialize in frequency estimation, addresses the problem by using a hierarchical data structure to dynamically accommodate elephant flows and mice flows while achieving close to $O(1)$ time insertion and query on average. As described above, the sketch's fast update speed for all flows makes it very suitable for frequency estimation. Unfortunately, they are not so good at top-k estimation. The original sketch-based schemes such as CM sketch may store different flows in the same space, thus leading to misclassification between mice flows and elephant flows; the Pyramid sketch, however, will access memory several times when querying elephant flows, which greatly slows down the speed.

Counter-based structures, on the other hand, record accurate information of elephant flows (e.g., Space-Saving [34], Lossy Counting [35], CSS [36], and HeavyKeeper [37]). They usually consist of several fixed-size counters, each storing a <key,value> pair. The key is the flow's ID or fingerprint, and the value is the flow's frequency. Counter-based schemes provide higher accuracy in elephant flow estimation because their replacement policy prefers to preserve large flows in the counters, and each counter has a flow's ID or fingerprint to avoid a elephant flow from being falsely identified as a mice one. So they are quite suitable for top-k estimation. Unfortunately, these fixed-size counter-based schemes store few information of mice flows, making them unfit for frequency estimation.

Recently, some works combine the ideas of the above two schemes to make them suitable for both frequency and top-k estimation [15], [32], [38]. A well-known solution is ASketch [15]. It uses an additional filter (with counters in it) on an existing sketch to aggregate elephant flows early, and the sketch processes the tail of the distribution. It utilizes the skewness of the underlying network stream to improve the accuracy of the most frequent flows by filtering them out earlier, preventing them from misclassification. Unfortunately, during insertion, it will cause many exchanges between the sketch and the filter, which greatly slows down the speed.

From the above description, we can find that the current difficulty mainly lies in *guarantee speed, error, and (top-k) precision at the same time*, for both frequency estimation and top-k estimation. In this paper, we propose a novel structure to address these problems, namely the Cuckoo Counter (CC). It employs a similar data structure as the Cuckoo hashing [39], consisting of m buckets organized in a specific way and B entries with different sizes in each bucket (The size of $entry_i$ increases as i increases). Each entry is a <fingerprint,frequency>-like key-value pair. Each packet is identified by its fingerprint. It makes $O(1)$ access to each bucket and linear traversal within a bucket, thus achieving high throughput during update. It also utilizes the skewness of the underlying network stream. When an overflow happens, CC tries to relocate the flow to a larger-size entry in the same bucket, which is still in one memory access, so as to guarantee that elephant flows are placed in a large-size entry; mice flows are placed in small-size entries. It also achieves high memory utilization. When a collision happens, CC leverages the cuckoo hashing [40] to kick out flows stored in the smallest entries (i.e., the mice flows), and fills each bucket as much as possible while separating elephant flows from mice flows as usual. Through the above key ideas, CC can achieve high accuracy for frequency estimation and high precision for finding top-k flows.

To verify the effectiveness of our work, we conduct extensive experiments and compare CC with typical algorithms. In terms of frequency estimation, CC achieves more than 1 order of magnitude lower error and advantageous throughput compared to CM sketch [25], Pyramid sketch [33], Elastic sketch [32], Nitro sketch [41], MV sketch [11], etc. In terms of finding top-k flows, CC achieves nearly an order of magnitude lower error and highest precision compared to Lossy Counting, Space-Saving, HeavyKeeper, Elastic sketch, MV sketch, etc. Especially, CC outperforms ASketch, which is also designed to work on both two tasks, by around 2 orders of magnitude on average. All related codes are open-sourced at our website [42] and GitHub [43].

The rest of the paper is organized as follows. Section II introduces the background and the related work addressed on our two tasks. Section III presents the data structure and algorithm of Cuckoo Counter. We conduct the mathematical analysis of Cuckoo Counter's upper/lower error bounds, space/time complexity and "error rate" in Section IV, and show experimental results in Section V. Finally, Section VI concludes the paper.

II. Background and Related Work

A. Problem Statement

We first formally define the network measurement task. A network stream $\mathcal{P} = \{p_1, p_2, \ldots, p_N\}$ contains N packets. Each packet belongs to one and only one flow, denoted as e. Packets in a network stream $\mathcal{P}$ can be classified into n non-overlapping flows: $\mathcal{E} = \{e_1, e_2, \ldots, e_n\}$. The number of packets in a flow e_i is called the frequency of e_i (abbreviated as $e_i.f$ or f_i), and so we have $\sum_{i=1}^{n} f_i = N$. A flow also has a unique ID to identify itself, represented as $e_i.id$ (for example, we often use the 5-tuple headers to identify a TCP flow in a network stream). All packets of the same flow have the same ID (i.e., $p_i.id$=$p_j.id$ if $p_i, p_j \in e_k$). CC mainly addresses two measurement tasks in this paper. The first one is **Per-Flow Frequency Estimation** [33]: Given a network stream $\mathcal{P}$, it provides the (approximate) frequency of each flow (i.e., $f_1, f_2, \ldots, f_n$); The second one is **Top-k Estimation** [37]: Given an integer k and a network stream $\mathcal{P}$, it provides a list of k flows from $\mathcal{E}$ with the largest flow sizes (frequency), i.e., $e_1, e_2, \ldots, e_k$. In the rest of this section, we show the related solutions and the state-of-the-art for these two tasks.

B. Frequency Estimation

As mentioned before, the most popular per-flow frequency estimation method in network measurement is sketch, such as CM sketch [25], CU sketch [24], Pyramid sketch [33], Elastic sketch [32], Nitro sketch [41] and MV sketch [11]. The most widely used sketch is the CM sketch. The CM sketch consists of d arrays, denoted by $\mathcal{A}_1 \dots \mathcal{A}_d$, where each array maintains W counters. There are d hash functions, $h_1 \dots h_d$. When inserting a packet p belonging to flow e, the CM sketch adds d mapped counters, $i.e. \mathcal{A}_i[h_i(e)], (1 \leq h_i(e) \leq w, 1 \leq i \leq d)$ by 1. When querying a flow e', it returns the minimum value of the d mapped counters, $i.e., min_{(1 \leq i \leq d)} \mathcal{A}_i[h_i(e')]$. Due to hash collisions, the same counter may be shared by different flows, which results in high error for mice flows. The CU sketch's process is similar to the CM sketch except that it only increases the minimum counters of the d mapped counters by 1 when insertion.

One of state-of-the-art works, the Pyramid sketch [33], is a layered data structure with λ layers. The number of counters of layer i is half of that of layer $i-1$. The first layer is a normal sketch (like CM) with pure counters, and the second layer and above are hybrid counters used for automatic carry. With this structure, it not only prevents counters from overflowing without the need of knowing the frequency of the elephant flow in advance, but also achieves high accuracy and high throughput at the same time. But whenever querying the elephant flow, it will access multiple layers and thus decreases the speed, making it difficult to perform top-k estimation.

C. Top-k Estimation

For top-k estimation, some algorithms that use sketches mentioned above with other structures were proposed. These solutions follow two basic strategies: $count\text{-}all$ and $admit\text{-}all\text{-}count\text{-}some$.

1) Count-All Strategy: This strategy uses sketches (such as CM sketch [25] or CU sketch [24]) to record the sizes of every flow, and uses a min-heap to keep track of the top-k flows, including the flow IDs and their flow sizes. Take the CM sketch as an example. For each arriving flow, we insert it into a CM sketch, and then update the min-heap by its estimated value. However, a mice flow may be misclassified as an elephant flow as we said previously.

2) Admit-All-Count-Some Strategy: Quite a few algorithms use the admit-all-count-some strategy, including Lossy Counting (LC) [35], and Space-Saving (SS) [34], etc. Take Space-Saving as an example. It only stores the information of some flows in a data structure called Stream-Summary. When a packet arrives, if its ID is not in the summary, the packet will be admitted into the summary, replacing the smallest flow whose size is n_{min}. The new flow's initial size will be set $n_{min}+1$. The main problem is that the strategy drastically over-estimates the sizes of flows since most flows are mice flows.

The state-of-the-art, HeavyKeeper (HK) [37], uses a new method called $exponential\text{-}weakening\ decay$ and a small hash table to store all elephant flows. When the incoming flow is not found in the hashed bucket, HeavyKeeper decays the flow size with a probability, which exponentially decreases as the flow size stroed in the bucket increases. If the flow size is decayed to 0, it will be replaced by a new flow. In this way, mice flows can easily be decayed to 0, while elephant flows are more likely to keep stable in the bucket. But the reported flow size might be under-estimated due to the decay operations. Besides, its fixed-size buckets make it difficult to adapt to the real network flow distribution. As a result, there is still potential for improvement in accuracy and precision when k and datasets are big.

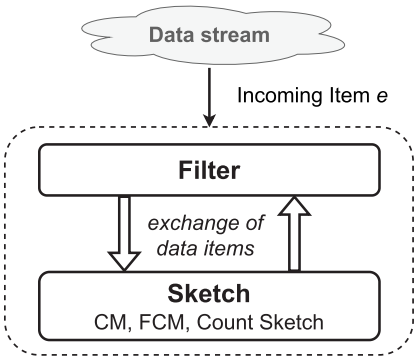


Fig. 1. The ASketch framework.

D. For Both

In fact, all sketches used for frequency estimation can simultaneously perform top-k estimation by adding a min-heap, but the effect may not be good. In the experimental part of Section V, we also compared the top-k estimation of the three excellent sketches designed for frequency estimation: Elastic sketch (EL) [32], Nitro sketch (NI) [41] and MV sketch (MV) [11].

As we said before, some schemes have recently been invented that focus on both tasks at the same time, including Augmented Sketch (ASketch) [15], Cold Filter [38], etc. We select the well-known ASketch as an example. The idea is illustrated in Fig. 1. It adds an additional filter (with counters in it) to an existing sketch Φ, to maintain the top-k flows within this filter. When inserting a packet p belonging to flow e, it scans each flow stored in the filter in order. If e has already been in the filter, it just increments its corresponding counter. Otherwise, it stores e with an initial count of one if there is available space in the filter. If there is no available space, it inserts this packet into the sketch Φ. During insertion, if the frequency of e reported by Φ is larger than the minimum value (associated with the flow e') in the filter, the ASketch needs to expel the flow e' to Φ, and insert e into the filter. Its key point is to use pre-filter to separate elephant flows from mice flows, thus utilizing the skewness of the underlying network stream to improve the accuracy for the most frequent flows by filtering them out earlier, preventing them from misclassification. However, its exchanges greatly slow down the speed, which can also be found in the experimental results in Section V. The structure of Cold Filter is similar to ASketch, but with an opposite function — to captures mice flows. What's more, each packet enters one stage at most once, which will not trigger exchanges and increases the speed compared

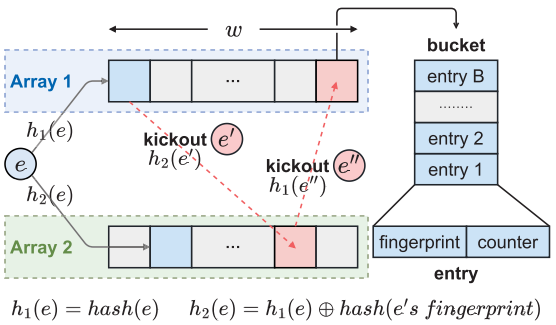


Fig. 2. The data structure of Cuckoo Counter.

to ASketch. However, since Cold Filter is a meta-framework, the limitations of those structures that combined with it still remain.

In a nutshell, although there are various solutions to addressing frequency estimation or top-k estimation, few methods can address both while achieving high speed, high accuracy, and high precision, due to their strategic or structural restrictions.

III. Cuckoo Counter Framework

In this section, we describe the data structure and algorithm of Cuckoo Counter. A data stream processing structure should provide two fundamental interfaces: *Insert()* and *Query()* to support each measurement task. For the frequency estimation and top-k estimation of Cuckoo Counter, we will discuss them separately.

A. Frequency Estimation

1) Data Structure: As shown in Fig. 2, Cuckoo Counter consists of two arrays, $\mathcal{A}_1$ and $\mathcal{A}_2$. We set the number of buckets to m, so each array has w buckets, and each bucket consists of B entries with different sizes: $\{entry_1, entry_2, \ldots, entry_B\}$ (the size of $entry_i$ increases as i increases). An entry is the unit of Cuckoo Counter, and a bucket is the unit of one access. As a result, we want the length of a bucket to be an integer multiple of a machine word (e.g., 64bits, 128bits…) and there will be no waste of memory access. Each entry consists of two parts, $fingerprint$ and $counter$. We employ the fingerprint as identification of the flow. All fingerprints must occupy the same size space in order to comply with the idea of cuckoo hashing. We assume the fingerprint takes up $\mathcal{F}$ bits memory space, and the counter part occupies the remaining space. All these entries can be used to estimate and store the mice flows. But the elephant flows are only stored in $entry_B, entry_{B-1} \ldots$ with larger size. Entries in the middle act as a buffer for counting between mice flows and elephant flows. They can deposit both mice and medium-size flows.

We introduce partial-key cuckoo hashing [40] to derive a flow's alternate location based on its fingerprint. For a packet p belonging to flow e, the details of calculating the index of two candidate buckets are as follows, $h_1(e) = hash(e), h_2(e) = h_1(e) \oplus hash(e's\ fingerprint)$ (since the $hash()$ function is 64 bits long and much larger than m, we later use $\tilde{h}_i(e)$ to refer to $h_i(e)\%w$, which is the number of the upper/lower bucket that e is mapped to).

The $\oplus$ (XOR) in the formula guarantees that $h_1(e)$ can also be computed from $h_2(e)$ and $e's\ fingerprint$, which means that $h_1(e) = h_2(e) \oplus hash(e's\ fingerprint)$. Hence, no matter which array the flow is now in, we can calculate the location of the flow in the other array by its current position and fingerprint:

$$h_{another} = h_{current} \oplus hash(flow's fingerprint)$$

2) Algorithm and Operations: **Insert:** For brevity, we use $\mathcal{A}_i[j][k]$ to refer to $array_i[bucket_j]$ $[entry_k]$. Initially, all entries are set to 0. When inserting a packet p belonging to flow e, we first compute two indexes by hashing, $h_1(e)$ and $h_2(e)$ to find two candidate buckets, $\mathcal{A}_1[\tilde{h}_1(e)]$ and $\mathcal{A}_2[\tilde{h}_2(e)]$. Then we scan all entries $\mathcal{A}_1[\tilde{h}_1(e)][j], \mathcal{A}_2[\tilde{h}_2(e)][j], (1 \le j \le B)$ in these two buckets. If flow e exists, we increment the counter of the corresponding entry by 1. If flow e is a new flow, then we check if there are empty entries in $\mathcal{A}_1[\tilde{h}_1(e)]$ or $\mathcal{A}_2[\tilde{h}_2(e)]$, and if so, insert the packet into the empty entry found firstly and set the value of the counter to 1. If the two buckets are full, we randomly select a flow e' in $\mathcal{A}_1[\tilde{h}_1(e)][1]$ or $\mathcal{A}_2[\tilde{h}_2(e)][1]$ to kick out, and insert flow e into the kicked-out entry. Then relocate the kicked flow e' by the partial-key cuckoo hashing. The flow e' will be inserted into the corresponding bucket of the other array. If that bucket is full too, the flow e'' in the $entry_1$ of the bucket will be kicked out, and the flow e' will be inserted to replace e''. This process will continue until the original flow and the kicked-out flows are inserted successfully, or the kicking out times reach $maxloop$. When the kicking number is $maxloop$, the last kicked out flow will be forced to be inserted into its corresponding bucket, then replace the fingerprint of $entry_1$ by its own fingerprint and the smaller value of these two counters will be stored.

Here we briefly explain why we take the smaller value of counters, since if we take the larger one, we can maintain the good property of no under-estimation error: Because of the overestimation caused by fingerprint collision, the underestimation caused by **min()** operation can offset the error to some extent. And the experimental result of average absolute error (AAE) on CAIDA datasets in Table I verifies our statement (the experimental result of average relative error (ARE) is similar). The Algorithm of insertion is given in Algorithm 1.

We only select an $entry_1$ of $\mathcal{A}_1[\tilde{h}_1(e)]$ or $\mathcal{A}_2[\tilde{h}_2(e)]$ randomly to kick out or insert when the buckets of flow e mapped are full. We make sure that $entry_1$ always records the mice flow in the network stream. When the counter of the entry overflows, such as the value of the counter in $entry_1$ reaches its capacity, we scan other larger entries in the bucket. If there is a larger entry, but its counter value is smaller than the overflowed entry, then swap the two entries. Otherwise, we check entries in the alternative bucket of the other array, if there is an entry Φ with greater size and smaller counter value than the overflowed entry. Then we kick out the original flow in Φ and relocate it. Afterward, we insert the overflowed flow into Φ. This will only introduce error of the frequency of mice flow to other entries, without mistakenly adding the frequency of elephant flow to that the mice flow, which

Algorithm 1 Insert(e)

Input: a packet p belonging to flow e

$fp = fingerprint(e)$, $maxloop \geq 0$;
$h_1(e) = hash(e)$, $h_2(e) = h_1(e) \oplus hash(fp)$;
we use $\mathcal{A}_i[j][k]$ to refer to $array_i[bucket_j][entry_k]$;
$entry.fp$ to refer to $entry.fingerprint$;
$entry.cnt$ to refer to $entry.counter$;
i, $i' \in \{1, 2\}$, $i + i' = 3$, $1 \leq j \leq B$;
if fp == $\mathcal{A}_i[\tilde{h}_i(e)][j].fp$ **then**
  $\mathcal{A}_i[\tilde{h}_i(e)][j].cnt$++;
  if $\mathcal{A}_i[\tilde{h}_i(e)][j]$ *overflow* **then**
    Stay_overflow($\mathcal{A}_i[\tilde{h}_i(e)][j]$);
    if *failed* **then**
      Kick_overflow($\mathcal{A}_i[\tilde{h}_i(e)][j]$);
else if $\mathcal{A}_i[\tilde{h}_i(e)]$ *has an empty entry* **then**
  insert e into the entry;
else
  random choose an $\mathcal{A}_i[\tilde{h}_i(e)][1]$;
  Kickout($maxloop$, $\mathcal{A}_i[\tilde{h}_i(e)][1]$);
  put $\{fp, 1\}$ to $\mathcal{A}_i[\tilde{h}_i(e)][1]$;
Function `Stay_overflow(`$\mathcal{A}_i[\tilde{h}][j]$`)`**:**
  if *has* $\mathcal{A}_i[\tilde{h}][k].cnt$, $(k > j)$ *is smaller* **then**
    swap($\mathcal{A}_i[\tilde{h}][j]$, $\mathcal{A}_i[\tilde{h}][k]$);
    return 1;
  return 0;
Function `Kickout(`$maxloop$, $\mathcal{A}_i[\tilde{h}][j]$`)`**:**
  $rh = h \oplus hash(\mathcal{A}_i[\tilde{h}][j].fp)$;
  if $\mathcal{A}_{i'}[\tilde{rh}][k]$ *is an* ***empty*** *and* ***capable*** *entry* **then**
    put $\mathcal{A}_i[\tilde{h}][j]$ to $\mathcal{A}_{i'}[\tilde{rh}][k]$;
  else if $maxloop$ - -> *0* **then**
    choose a capable entry $\mathcal{A}_{i'}[\tilde{rh}][k]$;
    Kickout($maxloop$, $\mathcal{A}_{i'}[\tilde{rh}][k]$);
    put $\mathcal{A}_i[\tilde{h}][j]$ to $\mathcal{A}_{i'}[\tilde{rh}][k]$;
  else
    choose a capable entry $\mathcal{A}_{i'}[\tilde{rh}][k]$;
    put $\mathcal{A}_i[\tilde{h}][j]$ to $\mathcal{A}_{i'}[\tilde{rh}][k]$ (take **smaller** cnt);
Function `Kick_overflow(`$\mathcal{A}_i[\tilde{h}][j]$`)`**:**
  $rh = h \oplus hash(\mathcal{A}_i[\tilde{h}][j].fp)$;
  if $\mathcal{A}_{i'}[\tilde{rh}][k]$ *is an* ***empty*** *and* ***capable*** *entry* **then**
    put $\mathcal{A}_i[\tilde{h}][j]$ to $\mathcal{A}_{i'}[\tilde{rh}][k]$;
  else
    choose a capable entry $\mathcal{A}_{i'}[\tilde{rh}][k]$;
    Kickout($maxloop$, $\mathcal{A}_{i'}[\tilde{rh}][k]$);
    put $\mathcal{A}_i[\tilde{h}][j]$ to $\mathcal{A}_{i'}[\tilde{rh}][k]$;
  set $\mathcal{A}_i[\tilde{h}][j]$ to 0;

especially improves the accuracy of the frequency estimation of elephant flows.

Query: When querying a flow e, we calculate two indexes firstly, $h_1(e)$ and $h_2(e)$, by partial-key cuckoo hashing. Then we match the fingerprint of e with these fingerprints in $\mathcal{A}_i[\tilde{h}_i(e)][j](i \in \{1, 2\}, 1 \leq j \leq B)$. If matched, we return the counter of the corresponding entry. Then if there is at least 1 empty entry, we return 0. Otherwise, we just return $\mathcal{A}_i[\tilde{h}_i(e)][1].counter$ that is smaller.

TABLE I
COMPARISON OF AAE BETWEEN MIN() AND MAX()

AAE(Insert)	CAIDA	Real-life	IMC
min()	**0.49**	**0.14**	**4.04**
max()	0.56	0.18	4.08

Algorithm 2 Query(e)

Input: a flow e
Output: frequency of e

$fp = fingerprint(e)$;
$h_1(e) = hash(e)$, $h_2(e) = h_1(e) \oplus hash(fp)$;
$i \in \{1, 2\}$, $1 \leq j \leq B$;
if *has* $\mathcal{A}_i[\tilde{h}_i(e)][j].fp$ == fp **then**
  return $\mathcal{A}_i[\tilde{h}_i(e)][j].cnt$;
if *has* $\mathcal{A}_i[\tilde{h}_i(e)][j]$ *is empty* **then**
  return 0;
return $\min_i(\mathcal{A}_i[\tilde{h}_i(e)][1].cnt)$;

TABLE II
COMPARISON OF AAE BETWEEN MIN() AND MAX()

AAE(Query)	CAIDA	Real-life	IMC
min()	**0.49**	**0.14**	**4.04**
max()	0.67	0.21	4.26

Here we also briefly explain why we take the smaller value of counters: 1). For flows that are in the datasets: the experimental result of average absolute error (AAE) on CAIDA datasets in Table II shows that taking the smaller value is better (the experimental result of average relative error (ARE) is similar). This is because for flows that are not stored in the Cuckoo Counter, the flows have a high probability of being low-frequency. 2). For flows that are out of datasets (should return 0 when querying these flows): obviously it is better to take the smaller value. The Algorithm of query is given in Algorithm 2.[1]

Deletion: The deletion operation of Cuckoo Counter is simple. We also compute two indexes of a flow e, $h_1(e)$ and $h_2(e)$, and scan entries in $\mathcal{A}_i[\tilde{h}_i(e)][j](i \in \{1, 2\}, 1 \leq j \leq B)$. If the same fingerprint of flow e in these entries exists, we decrease the corresponding counter by 1. Otherwise, we decrease the bigger $\mathcal{A}_i[\tilde{h}_i(e)][1].counter$ by 1. The reason is the same as that in querying. If after deletion, the counter is 0, we also delete the corresponding invalid fingerprint.

A running example: Fig. 3 shows an running example of Cuckoo Counter. We only show the first three buckets of each array and will set some parameters below. We assume $B = 3$, $maxloop = 2$, $entry_1.cnt$ occupies 4bit (the maximum value

[1]If we use the $max()$ operation in both *Insert()* and *Query()*, it is easy to prove the good property of no under-estimation error at this point. In contrast, the AAE/ARE increases but is acceptable (still ahead of the Pyramid sketch), which we do not elaborate on due to space constraints. It is up to the user to choose whether they want to use the min() or max() operation.

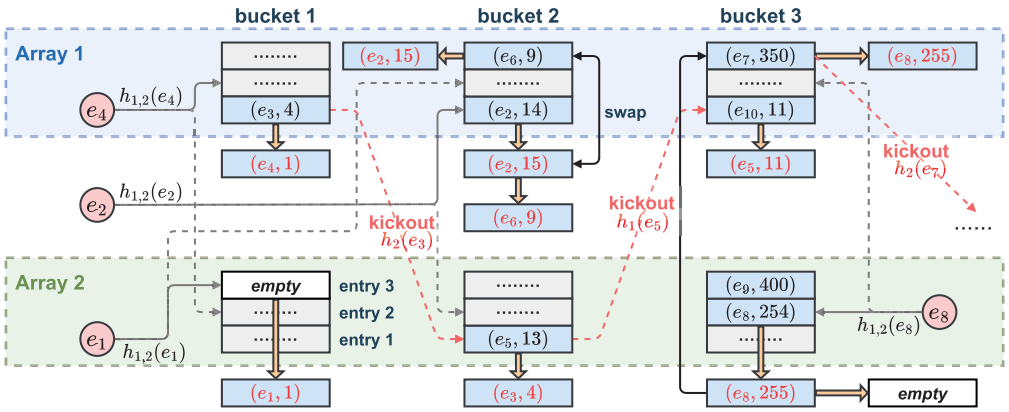


Fig. 3. An example of Cuckoo Counter using partial-key cuckoo hashing.

is 15), $entry_2.cnt$ occupies 8bit (the maximum value is 255), and $entry_3.cnt$ is big that we assume it will not overflow. In this example, given an entry with (e_i, fre), e_i is the flow's fingerprint (it is possible that two different flows have the same fingerprint, but we ignore it for convenience), fre is e_i's frequency. We will insert some packets belonging to some specific flows next. 1). *To insert* e_1*:* its two corresponding buckets are $\mathcal{A}_1[2]$ and $\mathcal{A}_2[1]$. We do not find it, but an empty entry in the $\mathcal{A}_2[1][3]$, so we fill (e_1, 1) into this entry. 2). *To insert* e_2*:* its two corresponding buckets are $\mathcal{A}_1[2]$ and $\mathcal{A}_2[2]$. We search for it in the two buckets and find that it already exists in an entry $\mathcal{A}_1[2][1]$, so we increase e_2 by 1. But at this time it is about to overflow, so we first look for a suitable entry in the same bucket. We then find $entry_3$ with (e_6, 9) and just swap them. 3). *To insert* e_4*:* its two corresponding buckets are $\mathcal{A}_1[1]$ and $\mathcal{A}_2[1]$. We still search in the two buckets, but do not find it, and there are no empty entries. So we randomly select an $entry_1$: $\mathcal{A}_1[1][1]$ with (e_3, 4) to kick out and replace it with (e_4, 1) forcibly. Unfortunately, e_3 does not find an empty entry in its alternate bucket ($\mathcal{A}_2[2]$). So we also select the $entry_1$: $\mathcal{A}_2[2][1]$ with (e_5, 11), kick out and replace it with (e_3, 4) forcibly. Finally, since $maxloop = 2$, e_5 replace $entry_1$ of its alternate bucket: $\mathcal{A}_1[3][1]$ and take the smaller value. So we fill (e_5, 11) into this entry. 4). *To insert* e_8*:* its two corresponding buckets are $\mathcal{A}_1[3]$ and $\mathcal{A}_2[3]$. We still search in the two buckets and found that it already exists in an entry: $\mathcal{A}_2[3][2]$, so we increase e_8 by 1. But at this time it is about to overflow, and the $entry_3$ in e_8's two buckets is too big to swap (with (e_9, 400) and (e_7, 350) in them), so we have to kick out $\mathcal{A}_1[3][3]$ with (e_7, 350), and replace it with (e_8, 255) then empty $\mathcal{A}_2[3][2]$.

B. Top-k Estimation

1) Data Structure: The data structure of Cuckoo Counter is the same as shown before, which consists of two arrays, and each array has w buckets separated into multiple entries with different sizes. To report the top-k most frequent flows, we add an extra heap. This heap is different from the min-heap in the paper of CM-sketch [25] or HeavyKeeper [37], it uses the f_{start} field to record the frequency of a flow when it is first entered into the heap. By adding this field, we can improve the algorithm to filter out some false top-k flows.

Algorithm 3 Update of Top-k

Input: a packet p belonging to flow e

min_{heap} = smallest $heap[x].f_{now}$, $1 \le x \le (1+\varepsilon)k$;
Insert(e); f_{cc} = **Query**(e);
if $e.id == heap[x].ID$ **then**
  $heap[x].f_{now} = f_{cc}$;
else if *has* $heap[x]$ *is empty or* $f_{cc} > min_{heap}$ **then**
  $heap[x].ID = e.id$;
  $heap[x].f_{start} = f_{cc}$;
  $heap[x].f_{now} = f_{cc}$;

Experiments show that this optimization improves the accuracy of top-k estimation compared to ordinary min-heap (refer to Appendix A).

As shown in the left half of Fig. 4. The extra heap consists of $(1+\varepsilon)k$ entries (where ε is a small number, such as 0.01), each represented as $heap[x]$ $(1 \le x \le (1+\varepsilon)k)$, where k is the number of elephant flows to be tracked. Each entry consists of three parts: flow ID, start frequency and current frequency, represented as $heap[x].ID$, $heap[x].f_{start}$ and $heap[x].f_{now}$, respectively.

2) Algorithm and Operations: **Update:** Initially, all entries of heap and Cuckoo Counter are set to 0. When inserting a packet p belonging to flow e, we first compute its fingerprint and insert it into Cuckoo Counter by the partial-key cuckoo hashing as shown in Algorithm 1. After insertion, we also get the frequency of e denoted by f (Insert() and Query() are similar in that they can be done together without affecting speed). Then we start to update the heap. If e lies in heap, we update the corresponding f_{now} field with f. Otherwise, if there exists an empty entry in the heap or f is greater than the minimum f_{now} in the heap, we insert or replace the flow of minimum entry with the information of e (set ID to e, f_{start} and f_{now} to f). The algorithm of insertion is given in Algorithm 3.

Detection: The method for detecting the top-k elephant flows is slightly different. We first compute the fingerprints of all flows in the heap, and then re-traverse the heap. When a flow e has a fingerprint collision and its $(f_{now} - f_{start})$ value is less than a given threshold, we do not report it; otherwise, we report it. The number of filtered flows cannot exceed εk.

Algorithm 4 Detection of Top-k

Output: an array //storing top-k flows

1 a multiset Φ; $X_{num} = 0$; threshold $= \Delta$;
2 **for** $x = 1; x \leq (1+\varepsilon)k; x$++ **do**
3 $\quad \Phi.insert(fingerprint(heap[x].ID))$;
4 **for** $x = 1; x \leq (1+\varepsilon)k; x$++ **do**
5 $\quad$ **if** $\Phi.count(heap[x].ID) == 1$ *or* $heap[x].f_{now} - heap[x].f_{start} \geq \Delta$ *or* $X_{num} \geq \varepsilon k$ **then**
6 $\quad\quad array.push\_back(heap[x].ID)$;
7 $\quad$ **else**
8 $\quad\quad X_{num}$++;
9 $\quad$ **if** $array.size() == k$ **then**
10 $\quad\quad$ break;
11 **return** $array$;

If the number of filtered flows reaches εk during the traversal process, then we report all subsequent flows. The algorithm of detection is given in Algorithm 4.

Here we explain why this optimization works. Because our CC stores flows with fingerprints, hash collisions may occur between different flows. So the following situation is possible: a mice flow e_0 and a elephant flow e_1 (assuming we are looking for the top-1000 flows, e_1 ranks 500) are mapped into the same bucket, and they have the same fingerprint, so that CC will misclassify e_0 as a large flow. e_0 will enter the heap with both its f_{start} and f_{now} set to a large value (approximate frequency of e_1) and its f_{now} field will hardly grow. e_1 will also enter the heap later, so a false top-k flow such as e_0 has the following characteristics: its fingerprint collides with some other flows in the heap, and its $f_{now} - f_{start}$ value is very small. We tested a variety of datasets and found that the false top-k flows in the heap do fit this characteristic, while the true top-k flows all have large $f_{now} - f_{start}$ values. Therefore, we pre-set a threshold Δ to distinguish the difference between the $f_{now} - f_{start}$ values of true and false top-k flows. If a flow in the heap has a fingerprint conflict and its $f_{now} - f_{start} < \Delta$, we just ignore it. Because some flows are filtered out, we set the heap size to $(1+\varepsilon)k$ to ensure that we end up with k flows. In the best case, we can filter out εk false top-k flows, which improves the top-k precision by εk. Such a change hardly adds extra memory (since $\varepsilon \ll k \ll number\ of\ buckets\ in\ CC$). The filter condition is shown in line 5 of Algorithm 4. For detailed performance comparisons of top-k estimation before and after optimization, please refer to Appendix A.

A running example: The update process is shown in the left half of Fig. 4. 1). *To insert* e_1 (blue arrow): We first get its frequency f_1 in CC, then we find e_1 in the heap and set its f_{now} field to f_1. 2). *To insert* e_2 (green arrow): We also get its frequency f_2 in CC, but we do not find it in the heap and there is no vacancy. We find that f_2 is greater than fn_0 (the smallest f_{now} in the heap), so we replace e_0 with e_2, and set both f_{start} and f_{now} fields to f_2. The detection process is shown in the right half of Fig. 4. Assuming that $threshold = \Delta = 3$, $k = 1000$, and $\varepsilon = 0.01$ (that means we want to find top 1000 frequent flows, and the heap size is 1010). We start traversing the heap: e_1 has no fingerprint collision and $f_{now} - f_{start} = 2039 - 159 = 1880 > \Delta = 3$, so we report it (denoted by "✓" in the red box on the right); e_2 has fingerprint collision, but it meet the threshold, so we report it. e_{500} has no fingerprint collision, we report it even if it does not meet the threshold. e_{800} has fingerprint collision, and it does not meet the threshold ($103 - 102 < 3$), so we ignore it (denoted by "×"). Finally, 1000 flows were reported when we traverse to e_{1003} (which means there are 1000 "✓"s in the red box on the right), and the flows from e_{1004} to e_{1010} were left over.

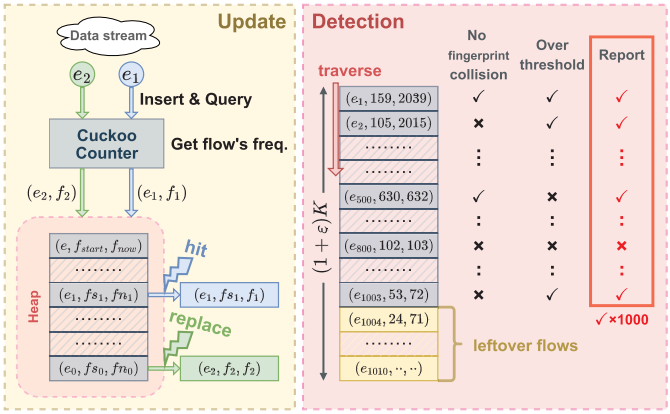


Fig. 4. Cuckoo Counter with an optimized heap.

IV. Mathematical Analysis

In this section, we analyze the error bound for both frequency and top-k estimation of Cuckoo Counter. We also define "error rate", the probability of identifying a non-occurring flow as non-zero, and compare between CC to CM.

We first consider the case of only pure CC (without min-heap). Suppose that CC records a network stream with N packets and n flows. Let e_i be the i^{th} flow, whose actual frequency is f_i. We assume the average size of the $entry_k$ is w_k, the ratio of packets falling in $entry_k$ across all buckets is λ_k, and use $fp()$ to refer to the $fingerprint()$ function. The final frequency estimation of flow e_i is

$$\hat{f}_i = f_i - X_i + Y_i \tag{1}$$

where X_i is the decrement from k kicks (due to the $min()$ operation in the replacement strategy, there will only be under-estimation) and Y_i is the increment from fingerprint collision. The two error bounds of Cuckoo Counter—the lower bound and the upper bound—result from X_i and Y_i respectively. We will calculate them separately.

A. Lower Bound Analysis of Pure CC

First, for $E(X_i)$, we assume that there is no fingerprint collision, which means $Y_i = 0$ in this part. Since there are a total of n flows, but only have $2w \times B$ entries in the structure to store flows, so there will be $n - 2wB$ kicks (each kick will trigger one $min()$ operation). We assume that $\mathcal{T}_{kick}$ represents

TABLE III
COMPARISON OF CUCKOO COUNTER WITH STOA

Freq.	r	w	Space	Insert	Query
CM [25]	$\log\frac{1}{\delta}$	$\frac{2}{\epsilon}$	$O(\frac{1}{\epsilon}\log\frac{1}{\delta})$	$O(\log\frac{1}{\delta})$	$O(\log\frac{1}{\delta})$
NI [41]	$\log\frac{1}{\delta}$	$O(\frac{1}{\epsilon^2 p}+\frac{\sqrt{\log\frac{1}{\delta}}}{\epsilon^2 p^{1.5}\sqrt{m}})$	$O(\frac{\log\frac{1}{\delta}}{\epsilon^2 p}+\frac{\log^{1.5}\frac{1}{\delta}}{\epsilon^2 p^{1.5}\sqrt{m}})$	$O(p\log\frac{1}{\delta})$	$O(\log\frac{1}{\delta})$
MV [11]	$\log\frac{1}{\delta}$	$\frac{2}{\epsilon}$	$O(\frac{1}{\epsilon}\log\frac{1}{\delta}\log n)$	$O(\log\frac{1}{\delta})$	$O(\log\frac{1}{\delta})$
CC	2	$\frac{1}{\delta\epsilon 2^{\mathcal{F}}}$	$O(\frac{1}{\delta\epsilon 2^{\mathcal{F}}})$	$O(1)$	$O(1)$
Top-k	—	—	Space	Update	Detection
SS [34]	—	—	$\min(n, O(K^2\log n))$	$O(1)$	$O(K)$
LC [35]	—	—	$\frac{1}{\epsilon}\log(\epsilon N)+K\log n$	$O(1)$	$O(K)$
HK [37]	—	—	$O(\frac{\gamma}{\epsilon\delta(b-1)})+K\log n$	$O(1)$	$O(K)$
CC	—	—	$O(\frac{1}{\delta\epsilon 2^{\mathcal{F}}})+K\log n$	$O(1)$	$O(K)$

the average number of $min()$ operations applied on each flow, so we have:

$$\mathcal{T}_{kick} = \frac{n - 2wB}{n} \tag{2}$$

The theoretical lower error bound is:

$$Pr[f_i - \hat{f}_i \geq \epsilon N] \leq \frac{\mathcal{T}_{kick}}{\epsilon N} \times (\frac{f_i}{2} - \frac{N\lambda_1}{4w}) \tag{3}$$

Detailed proofs can be found in Appendix B. □

B. Upper Bound Analysis of Pure CC

Second, assuming that the length of fingerprint is $\mathcal{F}$, we have the theoretical upper error bound:

$$Pr[\hat{f}_i - f_i \geq \epsilon N] < min(\frac{1}{w\epsilon 2^{\mathcal{F}}}, \frac{w_1}{\epsilon N}) \tag{4}$$

Detailed proofs can be found in Appendix B. □

C. Upper Bound Analysis of CC & Min-Heap

Next, we discuss the structure of CC & min-heap. Its theoretical upper error bound is:

$$Pr[\hat{f}_i - f_i \geq \epsilon N] \leq \frac{1}{w\epsilon 2^{\mathcal{F}}} \tag{5}$$

Detailed proofs can be found in Appendix B. □

Hence the overall error bounds holds by the Eq. (3), (4), and (5). □

From the above theoretical analysis, it can be seen that the error of the algorithm is inversely proportional to the fingerprint length, the number of entries in each bucket, and the number of buckets.

Furthermore, assuming that the error probability is δ. If we set the results of Eq. (4) and Eq. (5) as δ and ignore the effect of mice flows from Eq. (4), we can obtain the following time and space complexity in Table III (Notice that the r and w in the first line are the depth and width of the sketch, respectively).

Since many of the algorithms CC compared do not have corresponding mathematical analysis given in their original paper, we only select some of them to make a table as above. Note that some algorithms in the above table have self-contained parameters, so please go to the corresponding original paper if necessary.

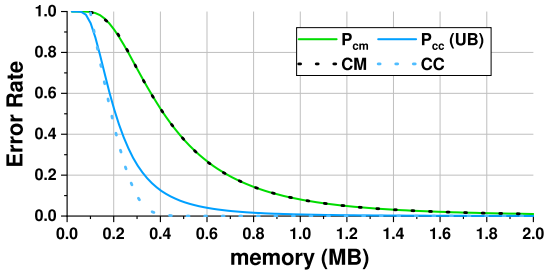


Fig. 5. CM and CC's error rate upper bound vs. memory.

D. Error Rate Comparison Between Pure CC and CM

We define "error rate" as the probability of identifying a non-occurring flow as non-zero after inserting all network stream. We return 0 in line 7 of Algorithm 2 when we do not find the corresponding fingerprint and there is an empty entry in two buckets. And we return the minimum value of two $entry_1$ if there is no empty entry. This raises the question: will we identify more non-occurring flows as non-zero compared to CM sketch?

Suppose d and w_{cm} are the depth and width of CM sketch respectively, w_{cc} is the width of Cuckoo Counter, n represents the type of flow in the network stream Number, M stands for memory usage (unit is MB).

1) For CM Sketch: We define P_{cm} as the error rate of CM, we have:

$$\begin{cases} P_{cm} = (1 - (1 - \frac{1}{w_{cm}})^n)^d \\ d \times w_{cm} \times 2 = M \times 1024 \times 1024 \end{cases} \tag{6}$$

Note that in the paper we take the counter size of CM sketch as 16 bits (2 bytes). Detailed proofs can be found in Appendix B. □

2) For Cuckoo Counter: We also define P_{cc} as the error rate of CC, we have:

$$\begin{cases} P_{cc} < 1 - (1 - \frac{1}{(2w_{cc})})^n \sum_{i=0}^{3} \frac{n^i}{i!(2w_{cc})^i} \\ 2 \times w_{cc} \times 8 = M \times 1024 \times 1024 \end{cases} \tag{7}$$

Note that in the paper we take the bucket size of Cuckoo Counter as 64 bits (8 bytes), and each bucket has 4 entries. Detailed proofs can be found in Appendix B. □

3) Experimental Results: Below we can draw the images of P_{cm} and P_{cc}'s upper bound next, we take $d = 4$, $n = 0.1million = 100000$, and the range of M is 0.1∼2MB.

We can find that: (1). The theoretical upper bound of P_{cc} is much lower than the theoretical results of P_{cm}, and we have also selected different n (e.g. 1 million) to plot, and the results are similar to the above Fig. 5. The real P_{cm} is very close to the theoretical value. (3).The real P_{cc} is slightly lower than the theoretical upper bound.

V. PERFORMANCE EVALUATION

In this section, we conduct a series of experiments. We first introduce the setup and metrics, then respectively show the experimental process of frequency estimation and top-k estimation. Finally, we analyze the experimental results. To guarantee that our experiments are conducted head-to-head, the

source code of the various algorithms we use is either common to the field or open-sourced by their authors, and we have released the related source codes and datasets at our website [42] and GitHub [43].

A. Experiment Setup

1) Test Platform: We performed all experiments on a machine with Intel i7-9700CPU@3.0GHz and 16G DRAM. The OS is Ubuntu 20.04. To reduce the CPU jitter error, we take the average results by running 10 times for each evaluation circularly.

2) Datasets: **i. CAIDA Datasets:** We use the CAIDA trace which collected in Equinix-Chicago monitor from CAIDA [44]. Our experimental CAIDA datasets are the same as that used in [32]. We use a trace with monitoring time of 5 seconds, which contains 165K flows, 2.49M packets. The maximum flow size is 17k. In order to evaluate the performance of our algorithms in large-scale measurement, for experiments on accuracy and throughput, we further use a trace with monitoring time of 2 minutes, which contains 1.71M flows, 53.72M packets. The maximum flow size is 0.93M.

ii. Real-Life Transactional Datasets: We download the RealLife Transactional dataset called WebDocs from website [45]. This dataset is built from a spidered collection of web html documents. More details about the dataset are in [46]. Since the dataset is too large, we cut it into sub-datasets each with a size of 102MB. The frequency of each packet ranges from 1 to 5349.

iii. IMC DC Trace: IMC Data Center Trace [47] is collected from the data centers studied in [48]. The character of data center traffic is that it contains a large number of flows, while simultaneously contains a few extremely large flows. The trace we use contains 1.77M flows, 7.59M packets. The maximum flow size is 3.35M.

iv. Synthetic Datasets: We generate a series of synthetic traces that follow the Zipf [49] distribution using Web Polygraph [50]. The skewness of the traces ranges from 0.0 to 1.0. Each trace contains approximately 1.0M flows, 32.0M packets. The maximum flow sizes range from 62 to 2.22M.

3) Algorithm and Operations: **i. Frequency Estimation:** We implement our Cuckoo Counter in C++, and compare our results with those of CM sketch (CM) [25], CU sketch (CU) [24], Augmented sketch (AS) [15], Pyramid sketch [33], Elastic sketch (EL) [32], Nitro sketch (NI) [41] and MV sketch (MV) [11]. Because the Pyramid framework can apply to different sketches: the CM, CU, and AS, and the Pyramid CU sketch (PCU) has the highest accuracy [33]. Therefore, we use PCU sketch as the representative of Pyramid sketch in our experiments. For CM, CU, AS and PCU, we used the open source C++ code in [33]; For EL, NI and MV, we implemented ourselves.

Notice that we also perform experiments on heavy hitter detection and heavy change detection in this part, because their accuracy is highly determined by the accuracy of frequency estimation. The definitions of heavy hitter and heavy change are as follows.

Heavy Hitter (HH) Detection: reporting flows whose sizes are larger than a predefined threshold.

entry 4 (24 bits)		entry 3 (16 bits)		entry 2 (12 bits)		entry 1 (12 bits)	
fingerprint 8 bits	counter 16 bits	fingerprint 8 bits	counter 8 bits	fingerprint 8 bits	counter 4 bits	fingerprint 8 bits	counter 4 bits

Fig. 6. The specific structure of entries in each 64-bit bucket.

Heavy Change (HC) Detection: reporting flows whose sizes in two adjacent time windows increase or decrease beyond a predefined threshold.

For the Synthetic dataset, we fixed the memory size to 500KB. The size of CM, CU and AS entries are 16 bits. CM and CU allocate 4 arrays and use 4 32-bit Bob hash [51] functions to flows mapping. The AS consists of the widely used CM sketch and a filter. The filter will allocate about 0.4KB additional memory, and the CM sketch of AS also includes 4 arrays and 4 32-bit Bob hash functions. All entries of the PCU are 4 bits, and the number of mapped entries is 4. The PCU use one 64-bit Bob hash function. EL's heavy part contains 8 entries in each bucket, and the depth of the CM sketch in the light part is 1. The depth of the count sketch of NI is 4, and the geometric sampling rate is $p = 0.01$ (recommended value). The depth of MV is 4.

Our Cuckoo Counter has three kinds of entries and two arrays. We fixed each bucket to 64-bit size in order to fit the scale of data streams and achieve one memory access for each bucket operation. The size of each bucket can be also adjusted to match different data streams. In each 64-bit bucket, four entries are distributed with 12 bits, 12 bits, 16 bits and 24 bits, respectively. In each entry, the fingerprint is 8 bits and the rest memory spaces are allocated to the counter, as shown in Fig. 6. We also use the 64-bit Bob hash to find two candidate buckets.

ii. Top-k estimation: For finding top-k flows, we compare our results with Lossy Counting (LC) [35], Space-Saving (SS) [34], Augmented Sketch (AS) [15], HeavyKeeper (HK) [37],Elastic sketch (EL) [32], Nitro sketch (NI) [41] and MV sketch (MV) [11]. HeavyKeeper performs best among all the related algorithms. For LC, SS, AS, EL, NI and MV, we implemented the code ourselves; for HK, we used the open-sourced C++ code.

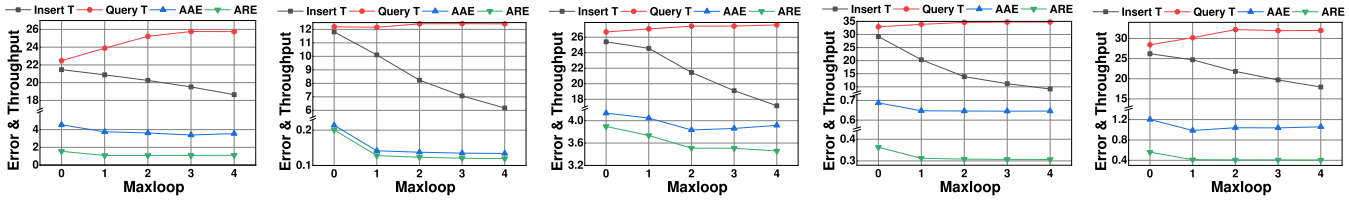
The memory size and k are set manually. The memory size determines the number of buckets in each data structure, and k indicates top-k flows to query, each algorithm reporting the largest k flows estimated. We fix $k = 1000$.

For the Synthetic dataset, we fixed the memory size to 500KB. For HK, EL, NI and MV, the number of buckets in Stream-Summary is k, and the rest memory size is distributed to their main structures. For AS, of which the CM sketch includes 2 arrays. For CC, the number of buckets in Stream-Summary is k, and the rest memory size is used by CC consisting of 2 arrays. Each bucket also has four entries with 4bits, 4bits, 8 bits, and 16bits, and fingerprints also 8 bits.

B. Metrics

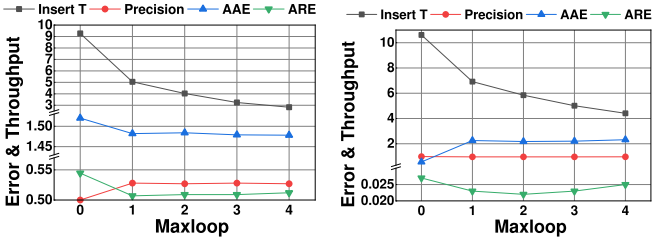
We consider the following metrics.

Throughput: Throughput is used to measure the processing speed of the insertion and query, and is estimated by the running time of the algorithm. It is estimated by the formula N/T, where N is the number of flows, T is the running



(a) CAIDA (b) Real-Life (c) IMC (d) Synthetic (0.0) (e) Synthetic (1.0)

Fig. 7. Frequency Parameter Tuning.



(a) Synthetic (0.0) (b) Synthetic (1.0)

Fig. 8. Top-k Parameter Tuning.

time. We use millions insertions per second (Mps) to represent throughput. All the experiments are repeated 10 times to minimize accidental deviations.

AAE: AAE is defined as $\frac{1}{|\Psi|}\sum_{(e_i\in\Psi)}|f_i-\tilde{f}_i|$, where f_i is the real frequency of flow e_i, $\tilde{f}_i$ is the estimated frequency, and the Ψ is the query set (for frequency estimation, it is all flows; for top-k estimation, it is top-k flows). For the all-flows' query set, the authors of the ASketch [15] use a sampled set of the whole multi-set, mainly focusing on querying the elephant flows. Without knowing the details of the sampling method used by the ASketch paper [15], *we focus on the whole dataset by querying each distinct flow only once*. That is why the ASketch is only a little better than the CM sketch in terms of accuracy in the following experiments.

ARE: ARE is defined as $\frac{1}{|\Psi|}\sum_{(e_i\in\Psi)}|f_i-\tilde{f}_i|/f_i$. These parameters in the formula have the same meaning as in AAE. We explain the reason why AAE and ARE are sometimes larger than anticipated in the experiments. When a sketch uses compact memory (e.g., 100 KB) to process massive data stream (e.g., 10M packets), significant over-estimations of mice flows will become common [33]. Take the real IP traces for example. About 41.8% flows in these data stream only have one packet, while large flows have more than 10,000 packets. When a flow with one packet is estimated as 101, the AAE and ARE will be both 100. This will make AAE and ARE much larger than anticipated.

Precision (HH/HC): Fraction of true heavy flows reported over all reported flows.

Recall (HH/HC): Fraction of true heavy flows reported over all true heavy flows.

F1-score (HH/HC): $\frac{2\times precision\times recall}{precision+recall}$.

Precision (Top-k): Fraction of true top-k flows reported over all the reported k flows.

C. The Effect of Maxloop

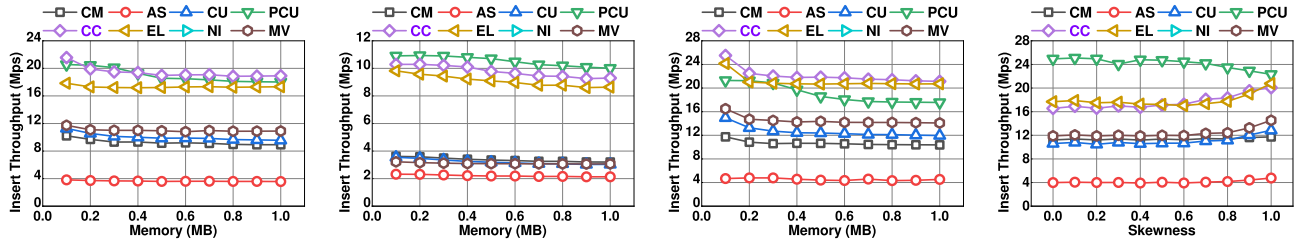
CC has an important parameter: $maxloop$, which determines the maximum number of kickouts for CC. We tune $maxloop$ and see how insert/query throughput, error and precision change to determine the $maxloop$ for subsequent experiments. We fix the memory to 100KB and conduct experiments on various datasets. The tuning results are shown in Fig. 7 and Fig. 8.

Frequency estimation: We can find that for insert throughput, it decreases linearly with the increase of $maxloop$. The reason is that with the arrival of packets, the bucket of CC is gradually filled, which causes more and more kickouts and slows down the speed. For query throughput, it increases slightly as $maxloop$ increases. This is because a higher $maxloop$ makes the CC fully filled, thereby increasing the hit rate of the *Query()* operation. For errors (AAE and ARE), there is a slight drop in $maxloop$ from 0 to 1. When the $maxloop$ is continuously increased, there is no obvious effect. The reason is that, with the arrival of packets, each bucket of CC is full, and the number of kickouts of the *Insert()* operation will reach the upper limit of $maxloop$, thus introducing an error in the kickout replacement strategy (Line 34 of Algorithm 1). As $maxloop$ continues to increase from 1, the advantage of fully utilizing the space offsets the error introduced by kickout. Since the errors are similar when $maxloop \geq 1$, we take $maxloop = 1$ in the frequency estimation experiments.

Top-k estimation: Here we only conduct experiments on two Synthetic datasets (the results of the remaining datasets are similar to Synthetic(1.0)). Note that we did not record the query throughput here, because when k is not very large, the time to query the top-k flows can generally be ignored. For insert throughput, it also drops significantly as $maxloop$ increases. But the rest of the metrics are not the same in the two datasets: in the Synthetic(0.0) dataset (Fig. 8(a)), the error and precision slightly improve when $maxloop$ changes from 0 to 1, and then remain unchanged; in the Synthetic(1.0) dataset (Fig. 8(b)), the precision remains the same, but the error increases instead. The reason is that the elephant flow may be affected by the flow kicked from elsewhere in the process of moving from the small entry in the bucket to the large entry. This effect is limited, and it will not identify an elephant flow as a mice flow (corresponding to no change in precision), but it will slightly increase the error. Since most datasets and data in real networks are highly skewed, we take the "Ostrich Policy" of $maxloop = 0$ in the top-k estimation.

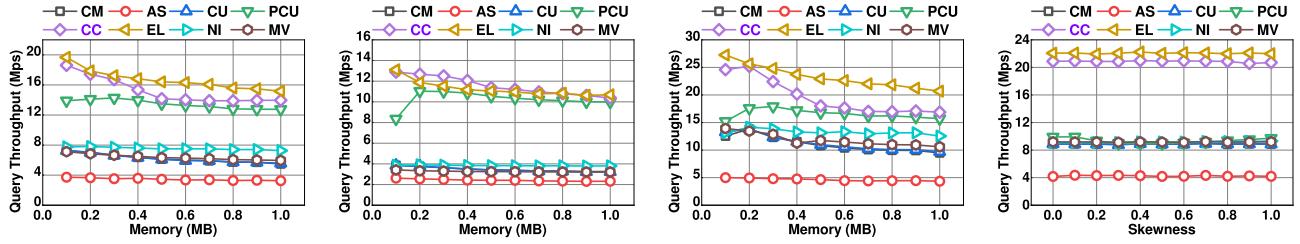
D. Experiments on Per-Flow Frequency Estimation

In this part, we illustrate the performance of our Cuckoo Counter by *insert throughput*, *query throughput*, *AAE*, *ARE*, *HH accuracy*, *HC accuracy*, *Large-scale measurement accuracy*. We use CC as an abbreviation of Cuckoo Counter, while



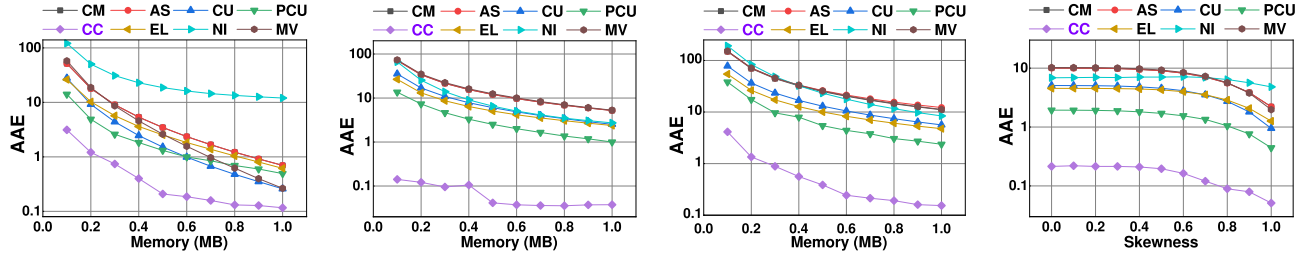
(a) CAIDA (b) Real-Life (c) IMC (d) Synthetic

Fig. 9. Insert throughput vs. **memory and skewness**.



(a) CAIDA (b) Real-Life (c) IMC (d) Synthetic

Fig. 10. Query throughput vs. **memory and skewness**.



(a) CAIDA (b) Real-Life (c) IMC (d) Synthetic

Fig. 11. AAE vs. **memory and skewness**.

CM, AS, CU, PCU, EL, NI, MV are used as the abbreviations of the CM sketch, Augmented sketch, CU sketch, PCU sketch, Elastic sketch, Nitro sketch and MV sketch respectively in the figures.

Insert Throughput: Fig. 10(a) shows the insert throughput vs. memory of various sketches on CAIDA dataset. Note that due to the sampling rate $p = 0.01$, the insert throughput of NI is very high ($>$ 100Mps), which is achieved at the loss of a certain accuracy (see Fig. 11 and Fig. 12), so we just ignore the insert throughput analysis of NI in this subsection.

Among them, PCU, CC, EL achieve the highest speed, because they only need to calculate a very few hash functions, while the rest of the sketches such as CM, AS, CU, MV need to calculate the hash function 4 times (the same depth as their sketches). Note that PCU (and other sketches as well) sees a slight throughput drop as the memory size increases, since it cannot be entirely put in the cache and the memory access latency increases.

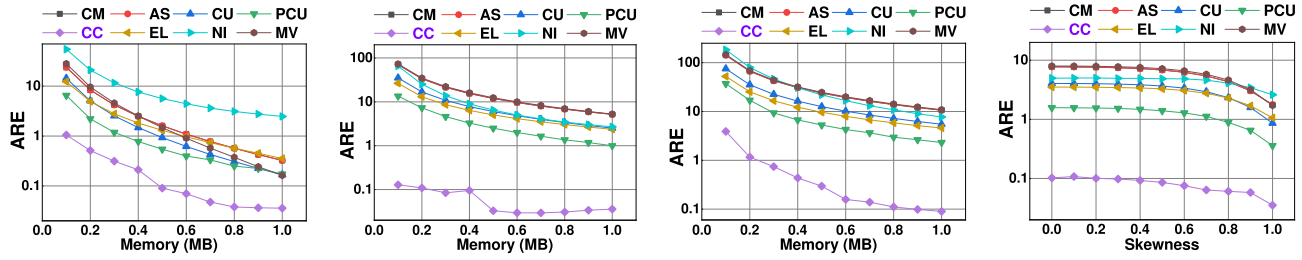
Fig. 10(b) and 10(c) are almost the same as Fig. 10(a). It is worth noting that Fig. 10(d) shows that PCU has higher throughput at lower skewness and lower throughput at higher skewness, while CC and EL are the opposite. This is because the PCU is a pyramid structure, and a small amount of elephant flows will make it frequently carry to the upper layer, which will increase the number of memory access and slow down the speed. However, since CC and EL record the flow's ID/fingerprint, the appearance of elephant flows will increase the number of memory hits. Thereby speeding up the insert throughput.

Query Throughput: Fig. 10(a), 10(b) and 10(c) shows the query throughput vs. memory of various sketches on different datasets. We can find that CC is only slightly slower than EL, and is faster than PCU and other sketches (especially when the memory is 100KB, the throughput rate of CC is 1.33-1.61$\times$ that of PCU at this time, because PCU needs to frequently access the upper layer, which increases the number of memory accesses).

Fig. 10(d) shows that under various skewness, the query throughput of CC is more than 2.00$\times$ that of sketches except EL, and it is close to EL.

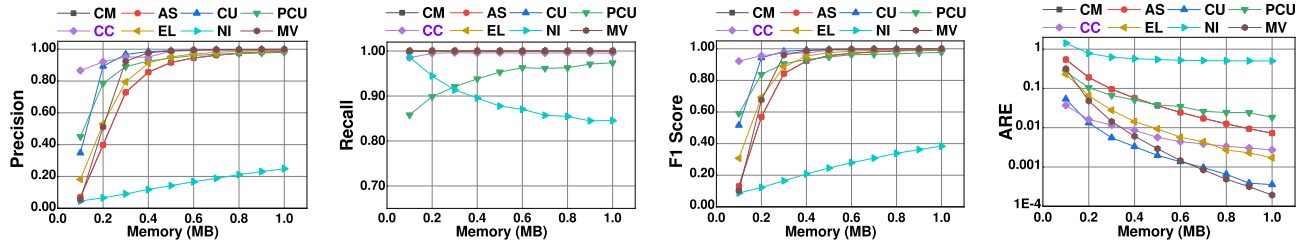
AAE: Fig. 11 shows the AAE vs. memory/skewness of various sketches on different datasets. It can be observed that under all memory and skewness, CC's AAE is at least 10$\times$ ahead of the rest of the sketches. Especially on the real-life dataset, CC leads by more than 100$\times$. In addition, it can be seen from Fig. 11(a), 11(c) and 11(d) that NI's AAE is not competitive when the memory is large or the skewness is large, which also confirms what we mentioned above: the high insert throughput of NI is caused by a low sampling rate and accuracy sacrifice.

ARE: Fig. 12 shows the ARE vs. memory/skewness of various sketches on different datasets. The graph of ARE is



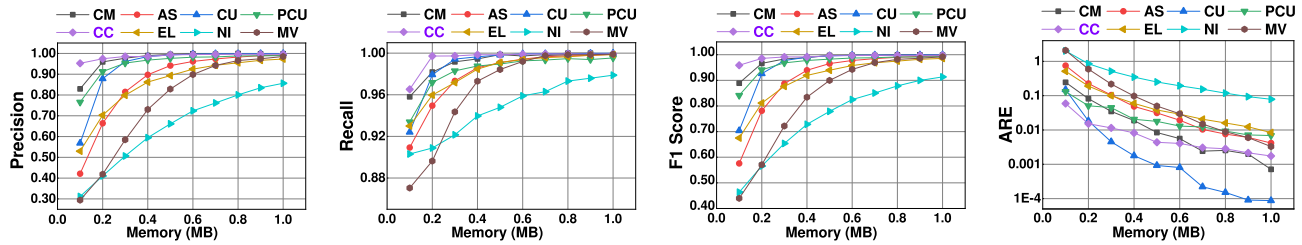
(a) CAIDA (b) Real-Life (c) IMC (d) Synthetic

Fig. 12. ARE vs. **memory and skewness**.



(a) Precision (b) Recall (c) F1 Score (d) ARE

Fig. 13. Heavy hitter detection.



(a) Precision (b) Recall (c) F1 Score (d) ARE

Fig. 14. Heavy change detection vs. **memory**.

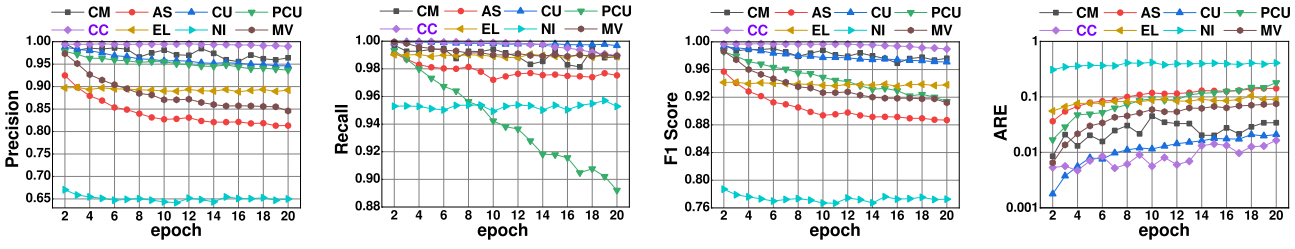
very similar to that of AAE. Compared with PCU (which has the best performance in the rest of the sketches), the ARE of CC is reduced by 4.92-105.31×. Both the AAE and ARE of CC outperforming other algorithms benefits from its cuckoo strategy (storing more flows with best effort) and adaptive design (making best use of limited memory).

Heavy Hitter Detection: Fig. 13 compares the accuracy vs. memory of CC with that of other sketches in heavy hitter detection. We use one CAIDA dataset and set the HH threshold to 0.002% of the total number of packets in dataset. NI has the lowest accuracy due to its sampling-based algorithm. The advantage of CC is that it maintains a high F1 score and precision when the memory is less than 200KB. We can also find that most sketches maintain almost 100% recall. This is due to their nature of overestimating frequency and increasing the false positive rate, which affects precision but not recall. Note that CU and MV have the smallest ARE, while CC and EL are close behind. The reason is that CC's kickout and replacement strategy may introduce errors when a heavy hitter is transferred from a small entry to a large entry in a bucket. We regard this as a design trade-off. In general, CC has the highest F1 score, precision, recall and competitive ARE.

Heavy Change Detection: Fig. 14 compares the accuracy vs. memory of CC with that of other sketches in heavy change detection. We use one CAIDA dataset and divide it into 10 epochs and look at the accuracy within the first epoch. We set the HC threshold as 0.001% of the total number of packets in dataset. Again, NI has the lowest accuracy due to sample rate. At this time, CC has the highest F1 score, precision and recall under all sizes of memory, especially when the $memory$ ≤200KB. CU has the lowest ARE, followed by CC and CM.

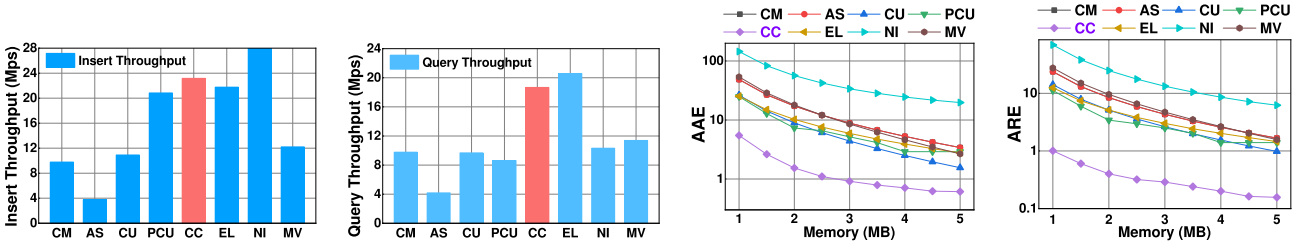
Fig. 15 compares the accuracy vs. epoch of CC with that of other sketches in heavy change detection. We use two CAIDA datasets and divide it into 20 epochs and see how the accuracy changes with epoch. We again set the HC threshold as 0.001% of the total number of packets in datasets. We set the memory to 500KB. Again, NI has the lowest accuracy. Here, the degree of fluctuation of each line reflects the stability of each sketch for heavy change detection. As the epoch increases, CC has nearly 100% F1 score, precision and the lowest ARE, while CU has the highest recall. We find that the recall of the PCU decrease significantly with the increase of epoch, which also reflects the low stability of the PCU for heavy change detection. Note that the ARE curves (Fig. 15(d)) of each sketch are on the rise, which is also very understandable: As more and more packets are inserted, the hash collision between different flows will increase, so the estimation error of the sketch must be larger and larger.

Large-Scale Measurement: We further use a CAIDA trace with monitoring time of 2 minutes to evaluate the performance of CC in large-scale measurement. We vary the memory from



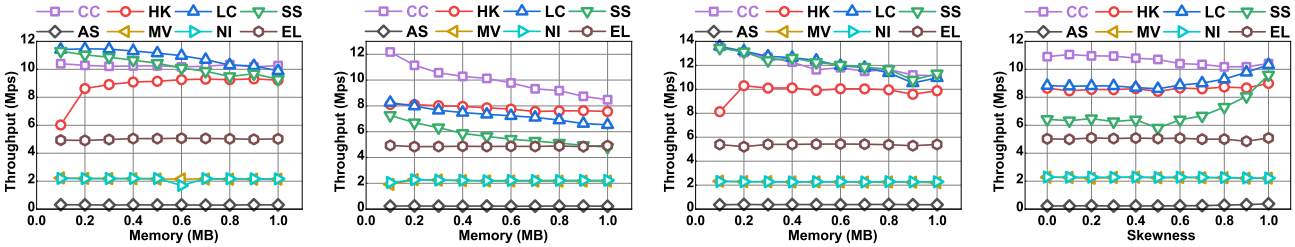
(a) Precision (b) Recall (c) F1 Score (d) ARE

Fig. 15. Heavy change detection vs. **epoch**.



(a) Insert Throughput (b) Query Throughput (c) AAE (d) ARE

Fig. 16. Large-Scale Measurement.



(a) CAIDA (b) Real-Life (c) IMC (d) Synthetic

Fig. 17. **Throughput** vs. **memory and skewness**.

1MB to 5MB. Since the throughput results are similar across memories, we take the results only once. As shown in Fig. 16, CC's insert throughput is only worse than NI (due to the sampling rate) and query throughput is only slightly worse than EL for large datasets. As before, CC achieves a substantial lead in AAE and ARE.

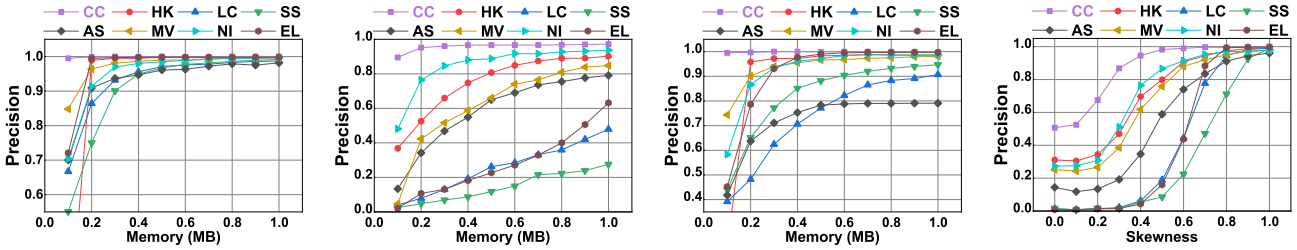
E. Experiments on Top-k Estimation

In this part, we illustrate the performance of our Cuckoo Counter applied to finding top-k flows. The main metrics are insert throughput, top-k precision, AAE and ARE. We compare our Cuckoo Counter (CC) with related Lossy Counting (LC), Space-Saving (SS), Augmented sketch (AS), Heavy Keeper (HK), MV sketch (MV), Nitro sketch (NI) and Elastic sketch (EL).

Throughput: Fig. 17 shows the insert throughput vs. memory/skewness of various sketches on different datasets. We find that in the Real-Life dataset (Fig. 17(b)), CC far outperforms all other data structures in throughput. Especially when the memory is small (such as 100KB), CC's throughput is 1.69× that of HK. In the CAIDA and IMC datasets (Fig. 17(a) and Fig. 17(c)), the throughput of CC is almost the same as that of LC and SS (all three are the highest), and it steadily exceeds HK. In synthetic dataset (Fig. 17(d)), the advantage of CC is also large, especially when the skewness is small (i.e., <0.5).

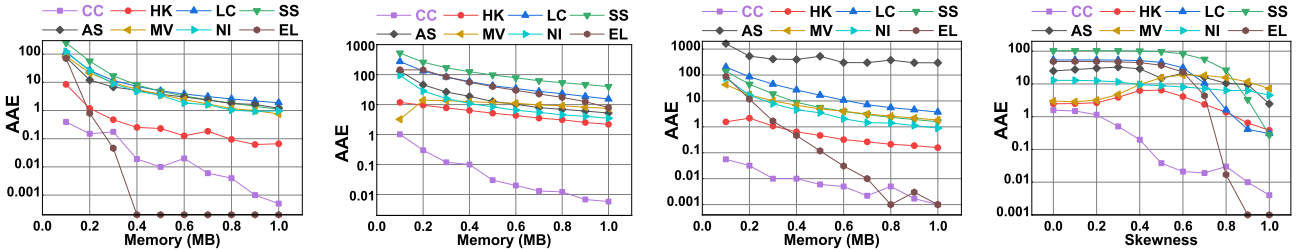
Precision: Fig. 18 shows the top-k precision vs. memory/skewness of various sketches on different datasets. We can find that CC maintains the highest accuracy in almost all memory (Fig. 18(a), 18(b) and 18(c)). And the minimum required memory for CC to reach 95% precision is the least (100KB for the CAIDA dataset, 300KB for the Real-Life dataset, and 100KB for the IMC dataset). Especially when the memory is small (≤100KB), the precision of CC can reach more than 99% (Fig. 18(a)) and 98% (Fig. 18(c)), and is 2.44-5.61× that of HK. Similarly, CC maintains the highest accuracy under all skewness, and is the first to achieve 95% precision at 0.5 skewness.

Next we explain why each algorithm performs poorly when skewness is low. Take the Synthetic dataset (skewness=0.0) we use as an example: at this time, the data distribution is very flat, the frequency of flows ranked 646∼1099 is 51, and the frequency of flows ranked 1100∼1807 is 50. Since the current top-k algorithms are probabilistic algorithms, it is very difficult for us to distinguish the two flows whose frequencies differ by 1, which leads to one of the following three reasons (depending on the characteristics of each algorithm): (1). There are many flows with frequency=51 are underestimated. (2). There are many flows with frequency=50 are overestimated. (3). Both of the above. This leads to huge precision errors. Datasets with high skewness will not have this problem: the frequency of flows ranked 1∼1000 is different,



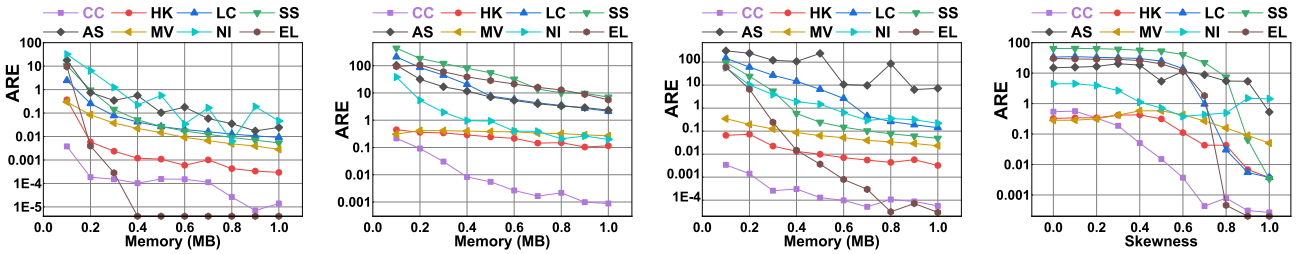
(a) CAIDA (b) Real-Life (c) IMC (d) Synthetic

Fig. 18. Precision vs. memory and skewness.



(a) CAIDA (b) Real-Life (c) IMC (d) Synthetic

Fig. 19. AAE vs. memory and skewness.



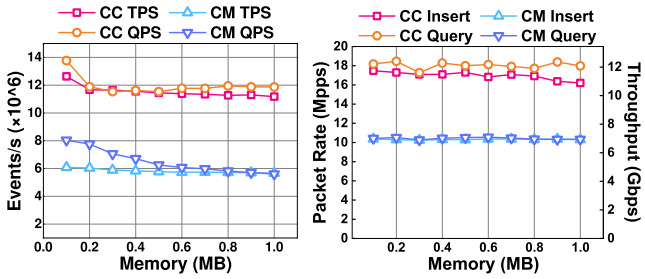
(a) CAIDA (b) Real-Life (c) IMC (d) Synthetic

Fig. 20. ARE vs. memory and skewness.

and it is only necessary to ensure that the relative order of frequencies is correct.

AAE: Fig. 19 shows the top-k AAE vs. memory/skewness of various sketches on different datasets. We find that in the Real-Life dataset, the AAE of CC is overall better than all other sketches, especially 11.60-97.18× better than HK. In the CAIDA and IMC datasets, it is also better than all sketches except EL, and only worse than EL when the memory is slightly larger (Fig. 19(a) and 19(c)). This is because the bucket of EL's heavy part stores the complete flow ID. When the memory is large, it can completely save the top-k flows' information, so it is only accurate when the memory is large. As can be seen from Fig. 19(d), under different skewness, CC maintains a huge lead over HK and other sketches, except that it is worse than EL when the skewness is slightly larger. When the skewness increases, the elephant flow effect of the data stream becomes more significant (reflected in the increase in the number of elephant flows). At this time, the adaptive structure of CC plays a role, making the elephant flow automatically adjust to the larger entry in the bucket.

ARE: Fig. 20 shows the top-k ARE vs. memory / skewness of various sketches on different datasets. The experimental results of ARE are similar to those of AAE. On the Real-Life dataset, CC leads HK and other sketches by more than 0.5 orders of magnitude. It also beats other HK and other sketches on the remaining datasets, and is only worse than EL when the memory or skewness is slightly larger.



(a) Redis (b) BESS

Fig. 21. Speed on software platforms.

F. Integration Into Redis

In order to systematically verify the effectiveness of CC on the software platform, we integrated and tested CC on top of Redis [52], which is often used as a distributed, in-memory storage system. We measure throughput under different memory. To complete this experiment, we rewrite CC in RedisBloom [53] (a module provides that probabilistic data structures for Redis) and add it to the Redis configuration file. The test platform is the same as the previous experiment, and the Redis version number is 6.2.6. We use the CAIDA

dataset, with 2.48M packets and 0.17M streams, and vary the memory from 0.1 to 1.

As shown in Fig. 21(a), we regard an insertion or query equally as an event, and Transactions Per Second (TPS) and Queries Per Second (QPS) can be regarded as the insert/query throughput before. We find that the TPS and QPS of CC are nearly 2 times higher than those of CM. The results of AAE and ARE are almost the same as the previous CPU simulation, so they are omitted.

G. Integration Into BESS

In order to verify the effectiveness of CC on the network platform, we integrated and tested CC on top of BESS [54], which is a programmable platform for vSwitch dataplane. We measure packet rate and throughput under different memory. To complete this experiment, we implement the sketching module of Cuckoo Counter as a plugin in the data plane processing pipeline. We use the $gen\_packet()$ function in BESS to randomly generate packets of size=64B.

As shown in Fig. 21(b), we measured packet rate (Mpps) and throughput (Gbps). For 64B packets, 10Gbps throughput is equivalent to 14.88Mpps, and 20Gbps equals to 29.76Mpps. We can find that the insert and query speed of CC are about 1.6-1.8$\times$ that of CM. We also omit the error.

VI. Conclusion

Frequency estimation and top-k flows identification are two fundamental problems in network traffic measurement. To work well on both tasks, we propose Cuckoo Counter (CC), an adaptive structure that consists of several buckets organized in a specific way. In summary, CC employs three key ideas to achieve our design goals: 1) *Memory efficient*: The size of the entry in each bucket is carefully designed to count the frequencies of mice flows and elephant flows respectively, which can handle skewed data streams efficiently and improve the memory utilization; 2) *High speed*: When an overflow happens, CC tries to relocate the flow to a larger-size entry in the same bucket within O(1) memory access, so that elephant flows and mice flows can be relocated into suitable entries without sacrificing update performance; 3) *High accuracy and high precision*: When serious conflicts happens, CC leverages the partial-key cuckoo hashing to kick out flows stored in the smallest entries, fills each bucket as much as possible to improve memory utilization without losing too much accuracy, and naturally preserves more elephant flows. Experimental results show that the CC can outperform the state-of-the-art and achieve very high accuracy with fairly limited memory usage in frequency estimation and finding top-k flows.

Acknowledgment

The authors would like to thank the anonymous reviewers for their thoughtful suggestions. Qilong Shi, Yuchen Xu and Jiuhua Qi conducted this work under the guidance of Wenjun Li and Tong Yang.

References

[1] J. Qi, W. Li, T. Yang, D. Li, and H. Li, "Cuckoo counter: A novel framework for accurate per-flow frequency estimation in network measurement," in *Proc. ACM/IEEE Symp. Archit. Netw. Commun. Syst. (ANCS)*, Sep. 2019, pp. 1–7.
[2] Q. Huang et al., "SketchVisor: Robust network measurement for software packet processing," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2017, pp. 113–126.
[3] M. Yu, L. Jose, and R. Miao, "Software defined traffic measurement with opensketch," in *Proc. USENIX NSDI*, 2013, p. 29–42.
[4] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One sketch to rule them all: Rethinking network flow monitoring with UnivMon," in *Proc. ACM SIGCOMM Conf.*, Aug. 2016, pp. 101–114.
[5] R. B. Basat, G. Einziger, R. Friedman, M. C. Luizelli, and E. Waisbard, "Constant time updates in hierarchical heavy hitters," in *Proc. ACM SIGCOMM*, 2017, pp. 127–140.
[6] L. Tang, Q. Huang, and P. P. C. Lee, "SpreadSketch: Toward invertible and network-wide detection of superspreaders," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, Jul. 2020, pp. 1608–1617.
[7] Q. Huang and P. P. C. Lee, "LD-sketch: A distributed sketching design for accurate and scalable anomaly detection in network data streams," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, Apr. 2014, pp. 1420–1428.
[8] Q. Huang, P. P. C. Lee, and Y. Bao, "Sketchlearn: Relieving user burdens in approximate measurement with automated statistical inference," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2018, pp. 576–590.
[9] O. Rottenstreich and J. Tapolcai, "Optimal rule caching and lossy compression for longest prefix matching," *IEEE/ACM Trans. Netw.*, vol. 25, no. 2, pp. 864–878, Apr. 2017.
[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. 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.
[12] A. Sivaraman et al., "Programmable packet scheduling at line rate," in *Proc. ACM SIGCOMM Conf.*, Aug. 2016, pp. 44–57.
[13] A. Lakhina, M. Crovella, and C. Diot, "Characterization of network-wide anomalies in traffic flows," in *Proc. ACM SIGCOMM*, 2004, pp. 201–206.
[14] C. Graham, K. Flip, M. Shanmugavelayutham, and S. Divesh, "Finding hierarchical heavy hitters in data streams," in *Proc. ACM VLDB*, 2003, pp. 464–475.
[15] P. Roy, A. Khan, and G. Alonso, "Augmented sketch: Faster and more accurate stream processing," in *Proc. ACM SIGMOD*, 2016, pp. 1449–1463.
[16] G. Cormode and M. Hadjieleftheriou, "Finding frequent items in data streams," *Proc. VLDB Endowment*, vol. 1, no. 2, pp. 1530–1541, 2008.
[17] G. Cormode, "Sketch techniques for approximate query processing," in *Foundations and Trends in Databases*. Norwell, MA, USA: NOW, 2011.
[18] A. Chen, Y. Jin, J. Cao, and L. E. Li, "Tracking long duration flows in network traffic," in *Proc. IEEE INFOCOM*, Mar. 2010, pp. 1–5.
[19] G. Cormode and M. Garofalakis, "Sketching streams through the net: Distributed approximate query tracking," in *Proc. ACM VLDB*, 2005, pp. 13–24.
[20] D. Thomas, R. Bordawekar, C. C. Aggarwal, and P. S. Yu, "On efficient query processing of stream counts on the cell processor," in *Proc. IEEE 25th Int. Conf. Data Eng.*, Mar. 2009, pp. 748–759.
[21] P. D. Amer and L. N. Cassel, "Management of sampled real-time network measurements," in *Proc. IEEE LCN*, Jan. 1989, pp. 62–63.
[22] 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.
[23] O. Rottenstreich, Y. Kanizo, and I. Keslassy, "The variable-increment counting Bloom filter," *IEEE/ACM Trans. Netw.*, vol. 22, no. 4, pp. 1092–1105, Aug. 2013.
[24] C. Estan and G. Varghese, "New directions in traffic measurement and accounting," in *Proc. ACM SIGCOMM*, 2002, pp. 323–336.
[25] 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, Apr. 2005.
[26] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," in *Proc. ICALP*. Cham, Switzerland: Springer, 2002.
[27] H. Dai, M. Li, A. X. Liu, J. Zheng, and G. Chen, "Finding persistent items in distributed datasets," *IEEE/ACM Trans. Netw.*, vol. 28, no. 1, pp. 1–14, Feb. 2020.
[28] R. Schweller et al., "Reversible sketches: Enabling monitoring and analysis over high-speed data streams," *IEEE/ACM Trans. Netw.*, vol. 15, no. 5, pp. 1059–1072, Oct. 2007.

[29] C. Min and S. Chen, "Counter tree: A scalable counter architecture for per-flow traffic measurement," *IEEE/ACM Trans. Netw.*, vol. 25, no. 2, pp. 1249–1262, Apr. 2017.

[30] T. Li, S. Chen, and Y. Ling, "Per-flow traffic measurement through randomized counter sharing," *IEEE/ACM Trans. Netw.*, vol. 20, no. 5, pp. 1622–1634, Oct. 2012.

[31] Y. Li, R. Miao, C. Kim, and M. Yu, "FlowRadar: A better NetFlow for data centers," in *Proc. USENIX NSDI*, 2016, pp. 311–324.

[32] T. Yang et al., "Elastic sketch: Adaptive and fast network-wide measurements," in *Proc. ACM SIGCOMM*, 2018, pp. 561–575.

[33] T. Yang, Y. Zhou, H. Jin, S. Chen, and X. Li, "Pyramid sketch: A sketch framework for frequency estimation of data streams," *Proc. VLDB Endowment*, vol. 10, no. 11, pp. 1442–1453, Aug. 2017.

[34] A. Metwally, D. Agrawal, and A. El Abbadi, "Efficient computation of frequent and top-K elements in data streams," in *Proc. ICDT*. Cham, Switzerland: Springer, 2005.

[35] G. S. Manku and R. Motwani, "Approximate frequency counts over data streams," in *Proc. ACM VLDB*, 2002, pp. 346–357.

[36] R. Ben-Basat, G. Einziger, R. Friedman, and Y. Kassner, "Heavy hitters in streams and sliding windows," in *Proc. 35th Annu. IEEE Int. Conf. Comput. Commun. (INFOCOM)*, Apr. 2016, pp. 1–9.

[37] 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.

[38] Y. Zhou et al., "Cold filter: A meta-framework for faster and more accurate stream processing," in *Proc. ACM SIGMOD*, 2018, pp. 741–756.

[39] R. Pagh and F. F. Rodler, "Cuckoo hashing," *J. Algorithms*, vol. 51, no. 2, pp. 122–144, May 2004.

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

[41] Z. Liu et al., "NitroSketch: Robust and general sketch-based monitoring in software switches," in *Proc. ACM SIGCOMM*, 2019, pp. 334–350.

[42] *Our Website*. Accessed: 2022. [Online]. Available: http://www.wenjunli.com/CuckooCounter

[43] *Our Open Souce Github*. Accessed: 2022. [Online]. Available: https://github.com/wenjunpaper/CuckooCounter

[44] *The Caida Traces*. Accessed: 2022. [Online]. Available: http://www.caida.org/data/overview/

[45] *Real-Life Transactional dataset*. Accessed: 2022. [Online]. Available: http://fimi.ua.ac.be/data/

[46] *Webdoca Dataset*. Accessed: 2022. [Online]. Available: http://fimi.uantwerpen.be/data/webdocs.pdf

[47] *Data Set for IMC 2010 Data Center Measurement*. Accessed: 2022. [Online]. Available: https://pages.cs.wisc.edu/ tbenson/IMC10_Data.html

[48] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proc. ACM SIGCOMM*, 2010, pp. 267–280. Accessed: 2010.

[49] D. M. Powers, "Applications and explanations of Zipf's law," in *Proc. ACM NeMLaP3/CoNLL*, Jan. 1998, pp. 151–160.

[50] A. Rousskov and D. Wessels, "High-performance benchmarking with web polygraph," *Softw., Pract. Exper.*, vol. 34, no. 2, pp. 187–211, 2004.

[51] *Hash Website*. Accessed: 2022. [Online]. Available: http://burtleburtle.net/bob/hash/evahash.html

[52] *Redis Website*. Accessed: 2022. [Online]. Available: https://redis.io

[53] *Redisbloom*. Accessed: 2022. [Online]. Available: https://github.com/RedisBloom/RedisBloom

[54] *Bess Website*. Accessed: 2022. [Online]. Available: https://github.com/NetSys/bess

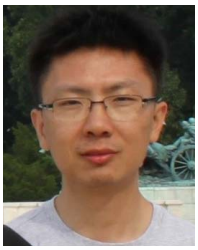
Qilong Shi is currently a Graduate Student with the Department of Computer Science and Technology, School of Electronics Engineering and Computer Science, Peking University. He conducts research work under the guidance of Prof. Tong Yang and Dr. Wenjun Li at the Institute of Network Computing and Information Systems, Peking University. His research interests include sketch, massive data stream processing, and network measurement.

Yuchen Xu is currently a Graduate Student with the Department of Computer Science and Technology, School of Electronics Engineering and Computer Science, Peking University. He conducted this work under the guidance of Dr. Wenjun Li and Prof. Tong Yang at the Institute of Network Computing and Information Systems, Peking University. His research interests include network big data, sketches, and network measurement.

Jiuhua Qi received the B.Sc. degree from the School of Electronic Science and Applied Physics, Hefei University of Technology, in 2017, and the M.Sc. degree in computer science and technology from the School of Electronic and Computer Engineering, Peking University, in 2020. He is currently an Engineer with Sangfor Technologies Inc. His research interests include sketches, cuckoo hashing, and network measurements.

Wenjun Li received the Ph.D. degree from Peking University in 2020. From 2020 to 2021, he worked as a Post-Doctoral Fellow at the Peng Cheng Laboratory. From 2014 to 2015, he also worked as a Research Engineer with Huawei Technologies Company Ltd. He is currently a Post-Doctoral Fellow at Harvard University and an Associate Researcher at the Peng Cheng Laboratory. His research interests include programmable network data plane and network algorithms.

Tong Yang received the Ph.D. degree in computer science from Tsinghua University in 2013. He visited the Institute of Computing Technology, Chinese Academy of Sciences (CAS). He is currently an Associate Professor with the Department of Computer Science and Technology, Peking University. He has published more than ten papers in SIGCOMM, SIGKDD, SIGMOD, and NSDI. His research interests include network measurements, sketches, IP lookups, bloom filters, and KV stores.



Yang Xu received the Ph.D. degree from Tsinghua University in 2007. He is currently a Yaoshihua Chair Professor with the School of Computer Science, Fudan University. Prior to joining Fudan University, he was a Faculty Member of the Tandon School of Engineering, New York University. He has published about 100 papers and holds more than ten U.S. and international granted patents on various aspects of networking and computing. His research interests include SDN, DCN, NFV, and edge computing.

Yi Wang received the Ph.D. degree in computer science and technology from Tsinghua University in July 2013. He is currently a Research Professor with the Institute of Future Networks, Southern University of Science and Technology. His research interests include future network architectures, information centric networking, software-defined networks, and the design and implementation of high-performance network devices.