Cuckoo Counter: A Novel Framework for Accurate Per-Flow Frequency Estimation in Network Measurement

Jiuhua Qi*, Wenjun Li*†‡, Tong Yang‡, Dagang Li* and Hui Li*†
*Shenzhen Graduate School, Peking University, †RCNC, Peng Cheng Laboratory, ‡EECS, Peking University
Email: jiuhuaqi@pku.edu.cn, wenjunli@pku.edu.cn, yang.tong@pku.edu.cn, dgli@pku.edu.cn, lih64@pkusz.edu.cn

***Abstract*—Per-flow frequency estimation plays a fundamental role in network measurement. As a probabilistic data structure, sketch has been extensively investigated and used for per-flow frequency estimation, but most sketch-based proposals in previous literatures cannot achieve high accuracy and high speed simultaneously. Moreover, because each insertion to a sketch causes increment in multiple entries, the over-estimation error will accumulate quickly over time. In this paper, we propose Cuckoo Counter, a compact and accurate framework for per-flow frequency estimation, which employs three novel ideas: (1) kicking out conflicting flows instead of using multiple entries counts to improve accuracy; (2) using different sizes of entries to insulate mice flows from elephant flows, which can handle the skewed data streams efficiently and improve memory utilization; (3) a Cuckoo-like replacement strategy for mice flows, so as to maintain accurate records for elephant flows. To verify the effectiveness and efficiency of our framework, we compared it with two well-known sketches as well as the recent proposed Augmented sketch and Pyramid sketch. Extensive experimental results on three different types of test datasets show that Cuckoo Counter outperforms these sketches considerably.**

***Index Terms*—Network measurement, Frequency Estimation, Sketch**

I. Introduction

Per-flow frequency estimation in network measurement can provide significant information for network operations, such as quality of service, congestion control, capacity planning and anomaly detection [1]–[9]. Many algorithms have been proposed [10]–[13]. In real network scenarios (phone call, videos sensor data network traffic, web clicks and crawls), the massive data comes as a high-speed stream [14]–[17] . However, the traditional method of network measurement is sample-based, which has low accuracy. Nevertheless, it is impractical to record all flows accurately in high-speed streams. So, using sketch a probabilistic data structure to estimate flows frequencies has become popular and widely accepted [8] [16] [18]–[21] . There are many classic sketches proposed in the literatures [4] [22]–[25]. Sketches provide fine-grained measurements and use a fixed small size of memory to summarize traffic statistics of all flows, causing only boundary errors. Furthermore, sketches use multiple hash functions to map a flow to multiple entries to improve the accuracy. But higher accuracy requires more hash functions, which results in lower speed. Hence, most sketches cannot achieve high accuracy and high speed simultaneously.

Another characteristic of network traffic is that it is non-uniformly distributed [8]. In other words, most flows have low frequencies (< 16, called mice flows), while a few flows have very high frequencies (> 40000, called elephant flows). Generally, the network traffic distribution conforms to the Zipfian [26] distribution, while most exiting sketches (CM sketches [22], CU sketches [4] and Count sketches [23]) that use entries of the same size do not work well for the network traffic. That is because if the entry size is allocated according to the size of elephant flows, the higher bits in many entries used to store mice flows are all wasted. This results in low memory utilization. But, if the entry size is allocated according to the size of mice flows, the entry overflow will result in very low measurement accuracy of the elephant flows. The Augmented sketch [14] uses an additional filter on existing sketch S to capture elephant flows dynamically, but when inserting a flow, it will cause many exchanges between the sketch and the filter. The Pyramid sketch [27] proposed lately uses a hierarchical data structure to dynamically accommodate elephant flows and mice flows. While it still requires multiple entries updated for each insertion, which affects the performance. Since for the same memory size, if a flow is counted as many entries, there will be fewer entries to exactly estimate different flows, which will increase the probability of hash collisions and thus affect the overall error.

In this paper, we propose Cuckoo Counter, a compact and accurate framework for per-flow frequency estimation. Cuckoo Counter maintains a similar data structure as the Cuckoo hash [28], with two arrays of N buckets each, and four entries in each bucket, but the entries are of different sizes. Each entry consists of two parts, *i.e.)* fingerprint and counter. The key ideas of our Cuckoo Counter are as follows. (1) We leverage entries with different sizes to count the frequencies of mice flows and elephant flows respectively, which can handle skewed data streams efficiently and improve the memory utilization. The reason we assign four entries to each bucket is that when light collisions occur, we keep flows that are all conflicting in the same bucket as much as possible to avoid kick-outs across buckets. When the current counter of the entry overflows, we relocate the stored flow to a larger entry in the

This work is supported by NSFC (61671001, 61672061), Key Areas R&D Program of Guangdong (2019B010137001), National Keystone R&D Program of China (2017YFB0803204, 2016YFB1000304, 2018YFB1004403), PCL Future Regional Network Facilities for Large-scale Experiments and Applications (PCL2018KP001), Shenzhen Peacock Innovation Program (KQJSCX20180323174744219) and Shenzhen Research Program (JCYJ20170306092030521). Corresponding authors H. Li and W. Li are also with Shenzhen Key Lab of Information Theory & Future Internet Architecture.

978-1-7281-4387-3/19/$31.00 © 2019 IEEE

same or the alternative bucket, so as to guarantee that elephant flows are placed in entries with large size. We fixed each bucket to a 64-bit size so that we only need one memory access for per bucket operation, which would achieve fast speed. (2) When serious conflicts occur, we use partial-key cuckoo hashing [29] to kick out flows that are in the smallest entries, thus ensuring that we always kick out mice flows. Therefore, we merely introduce errors among mice flows and make sure that the statistics of the elephant flows are accurate, because elephant flows are generally considered more important than mice flows in most cases. (3) We only use one entry to store the frequency of a flow while ensuring high precision and high speed, thus improving memory utilization. However, we need to introduce fingerprints to identify different flows in the same bucket, which will cause small memory overhead but easy to distinguish elephant flows. Our experiments also show that this memory consumption is entirely acceptable. The experiments illustrate that our Cuckoo Counter improves the insertion throughput by 30%, query throughput by 70% and accuracy by 230% compared with the state-of-the-art Pyramid sketch on average. The source code is available at Github [30].

The rest of this paper is organized as follows. In Section II, we first briefly summarize the related work. We describe our Cuckoo Counter in detail in Section III. Section IV provides experimental results. Finally, Section V draws conclusion and the future work.

II. Relate work

The most popular per-flow frequency estimation method in network measurement is sketches. The classical frequency estimation sketches include CM sketches [22] , CU sketches [4], Count sketches [23], and the recently frequency estimation sketches are Augmented sketch [14] and Pyramid sketches [27]. The most widely used sketch is the CM sketch. A CM sketch consists of d arrays, denoted by $A_{1...}A_d$, where each array maintains W entries. There are *d* hash functions, $h_{1...}h_d$. When inserting a flow *e*, the CM sketch adds all the *d* mapped entries, i.e.) $A_1[h_1(e)]\cdots A_i[h_i(e)], (1 \leq h_i(e) \leq w, 1 \leq i \leq d)$ by 1. When querying a flow *e'*, it returns the minimum values of the d mapped entries, i.e.) $min_{(1\leq i\leq d)}A_i[h_i(e')]$. Due to hash collisions, the same entry may be shared by different flows, which results a high error for mice flows. The CU sketch processes are similar to the CM sketch expect that it only increases the minimum entries of the d mapped entries by 1 when inserting. The Count sketch is also similar to the CM sketch except for assigning two hash functions to each array. The Augmented sketch uses an additional filter (a queue with *k* entries) on existing sketch *S* to capture elephant flows dynamically. It uses prefilter to increase accuracy but increases complexity and results slower update and query speed. The Pyramid sketch is a layered data structure with λ layers. The entries of layer *i* are half of layer $i-1, i \subseteq [1, \lambda]$. It can dynamically adapt to mice flows and elephant flows in network traffic and has high speed and accuracy, but it still uses multiple entries to estimate a flow, which results in low memory utilization and large error.

There are two commonly algorithms of Counter variants(Counter Braids [11] and Randomized Counter Sharing [12]) are used to estimate per-flow frequency. In Counter Braids, due to the post process, the query speed is significantly slow. The Randomized Counter Sharing sacrifices its accuracy for high speed. So, both of them cannot provide high speed and high accuracy at the same time. There are also some other typical counter-based data structures in network measurement, but most of them are used for finding the top-k frequent flows, such as Lossy Counting [1], Space-Saving [31] and Frequent [32].

We are inspired by the Cuckoo hashing [28]. Cuckoo hashing is proposed to solve the hash conflict problem. It has the characteristics of high space efficiency and fast query speed. The Cuckoo filter [29] is a compact variant of a cuckoo hash table that stores only fingerprints. The Cuckoo filter is primarily used to check if a flow exists. To the best of our knowledge, Cuckoo hashing or Cuckoo filter has not been used to perform per-flow frequency estimation in data streams. Meanwhile, the Cuckoo filter is not suitable for frequency estimation in data streams, because it assigns the same sizes of entries and kicks out too many times resulting in very low speed.

Summary: Although there are various algorithms to estimate the frequency of per-flow in network traffic, no existing methods can achieve high accuracy and speed by storing the frequency information of a flow only once.

III. Cuckoo counter framework

In this section, we describe the data structure and algorithm of our Cuckoo Counter. The algorithm includes insertion, query, and deletion.

A. Data Structure

As shown in Figure 1, our Cuckoo Counter consists of two arrays, A_1 and A_2. Each array has N buckets, and each bucket consists of four entries with different sizes, $entry_1$, $entry_2$, $entry_3$, $entry_4$. Entry is the unit of our Cuckoo Counter. The four entries are 12 bits, 12 bits, 16 bits, 24 bits respectively. Therefore, each bucket is 64 bits, and only one memory access is required for each bucket operation. Each entry consists of two parts, fingerprint and counter. We employ the fingerprint of the key as identification of the flow. All fingerprints take up 8bits memory size, therefore, the sizes of counters are 4 bits$\sim$16 bits. $Entry_1.counter$ and $entry_2.counter$ are 4 bits. $Entry_3.counter$ is 8 bits and $entry_4.counter$ is 16 bits. All these entries can be used to estimate and store the mice flows. But the elephant flows only are stored in $entry_4$. $Entry_3$ acts as a buffer for counting between mice flows and elephant flows. It can deposit both mice and medium-size flows.

We introduce partial-key cuckoo hashing [29] to derive a flows alternate location based on its fingerprint. For a flow *e*, the details of calculating the index of two candidate buckets are as follows, $h_1(e) = hash(e)$, $h_2(e) = h_1(e) \oplus hash(e's\ fingerprint)$.

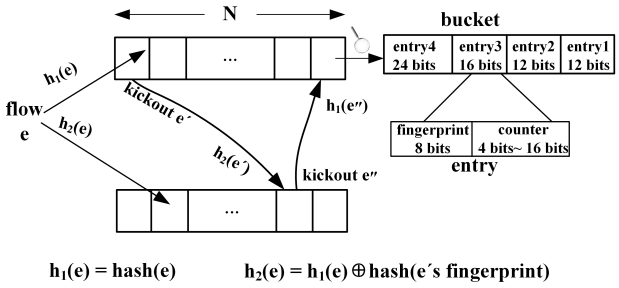


Fig. 1. The data structure of Cuckoo Counter.

The 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 in, we can calculate the location of the flow in another array by its current position and fingerprint:

$$h_{another} = h_{current} \oplus hash(flow's\ fingerprint) \quad (1)$$

B. Algorithm and Operations

Insertion: Initially, all entries are set to 0. When inserting a flow e, we first compute two indexes by hashing, $h_1(e)$ and $h_2(e)$ to find two candidate buckets, $A_1[h_1(e)]$ and $A_2[h_2(e)]$. Then we scan all entries $A_1[h_1(e)][entry_j]$, $A_2[h_2(e)][entry_j], (1 \le j \le 4)$ in these two buckets. If flow e exists, we increment the counter of corresponding entry by 1. If flow e is an new flow, then we check if there are empty entries in $A_1[h_1(e)][entry_j]$ or $A_2[h_2(e)][entry_j]$, and if so, insert the flow into the empty entry which is found firstly and set the value of the counter to 1. If the two buckets are full, we randomly select a flow e' in $A_1[h_1(e)][entry_1]$ or $A_2[h_2(e)][entry_1]$ to kick out, and insert flow *e* into the kicked-out entry. Then relocate the kicked flow *e'* by the partial-key cuckoo hash. The flow *e'* will be inserted into the corresponding bucket of another array. If the bucket is full too, we will kick out the flow *e"* in the $entry_1$ of the bucket to insert the flow *e'*, and replace *e"* again. We keep kicking out and inserting until the original flow and the kicked-out flows are inserted, or the kicking out times reach 2. When the kicking number is 2, we randomly select a bucket from the current kicked-out flow mapped two buckets, then replace the fingerprint of $entry_1$ in the bucket by the fingerprint of current kicked-out flow and add these counters of two flows simply. The Algorithm of insertion is given in Figure2.

We only select an $entry_1$ of $A_1[h_1(e)][entry_j]$ or $A_2[h_2(e)][entry_j]$ randomly to kick out or insert when the buckets of flow *e* mapped are full. Because we make sure that $entry_1$ always record the mice flow in data stream. When the counter of the entry overflows, such as the value of the counter in $entry_1$ is bigger than 16, we scan other larger entries in the bucket. If there is a larger entry, but its counter value is smaller than the overflow entry, then swap the two entries. Otherwise, we check these entries in another alternative bucket, if there is

Algorithm 1: Insert (e)

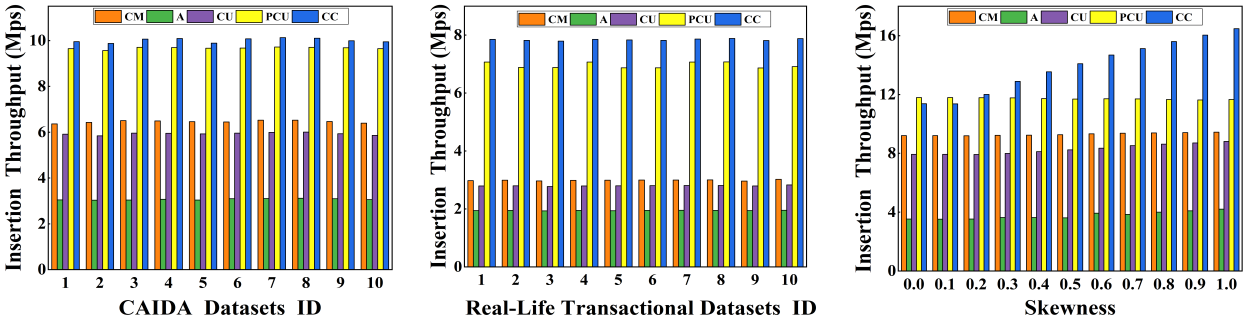
```
1  f = fingerprint(e), maxkick = 2;
2  h1(e) = hash(e), h2(e) = h1(e)⊕ hash(f);
3  i, i' ∈ {1, 2},   i + i' = 3;  1 ≤ j ≤ 4;
4  If A_i[h_i(e)][entry_j].fingerprint == f  then
5     A_i[h_i(e)][entry_j].counter++;
6    If A_i[h_i(e)][entry_j].counter overflow then
7       if has A_i[h_i(e)][entry_k].counter <
              A_i[h_i(e)][entry_j].counter, (k > j) then
8          swap A_i[h_i(e)][entry_k] and A_i[h_i(e)][entry_j];
9       else if A_i'[h_i'(e)][entry_k].counter <
                        A_i[h_i(e)][entry_j].counter  then
10         swap f and A_i'[h_i'(e)][entry_k].fingerprint;
11         swap A_i[h_i(e)][entry_j].counter and
                        A_i'[h_i'(e)][entry_k].counter;
12         h_i(e) = h_i'(e)⊕ hash(f);
           maxkick--;
13         if A_i[ h_i(e)] has an empty entry then
14            insert e into the entry;
15         else Insert(e);
16    end;
17 Else if A_i[h_i(e)] has an empty entry then
18    insert e into the entry;
19 Else
20    for ; maxkick > 0; maxkick--  do
21       if  maxkick == 0 then
22          randomly select an  A_i[h_i(e)][entry_1];
23          replace A_i[h_i(e)][entry_1].fingerprint by f ;
24          A_i[h_i(e)][entry_1].counter + the counter of e;
25       else
26          randomly select an  A_i[h_i(e)][entry_1];
27          swap f and A_i[h_i(e)][entry_1].fingerprint;
28          swap the counter of e
                  and A_i[h_i(e)][entry_1].counter;
29          h_i'(e) = h_i(e)⊕ hash(f);
30          if A_i'[h_i'(e)] has an empty entry then
31             insert e into the entry;
32             break;
33 End;
```

Fig. 2. Insert algorithm

Algorithm 2: Query(e)

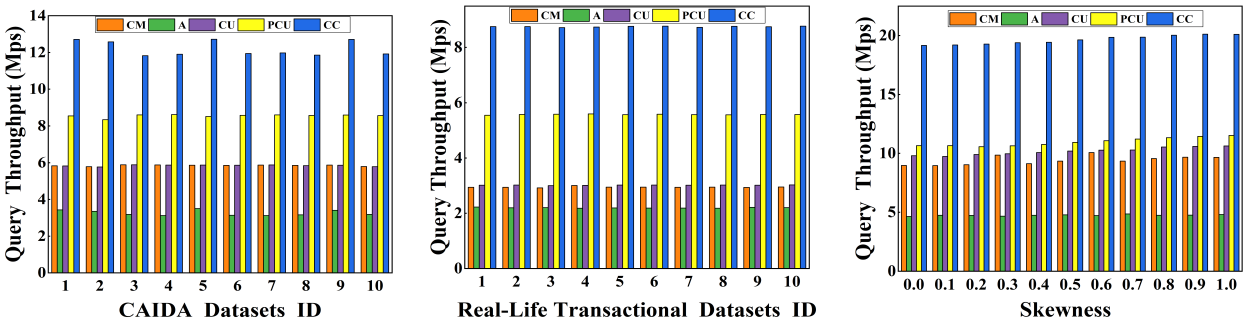
```
1  f = fingerprint(e),  1 ≤ j ≤ 4,  i ∈ {1, 2};
2  h1(e) = hash(e),  h2(e) = h1(e) ⊕ hash(f);
3  If  A_i[h_i(e)][entry_j].fingerprint == f  then
4     return A_i[h_i(e)][entry_j].counter;
5  Else
6     randomly select an entry_1  from  A_i[h_i(e)][entry_1] ;
7     return A_i[h_i(e)][entry_1].counter;
8  End;
```

Fig. 3. Query algorithm



(a) caida datasets (b) real-life transactional datasets (c) synthetic datasets with different skewness

Fig. 4. Comparison of insertion throughput on different datesets.



(a) caida datasets (b) real-life transactional datasets (c) synthetic datasets with different skewness

Fig. 5. Comparison of query throughput on different datesets.

a greater size of the entry *f*, but its counter value is less than the overflow entry. Then we kick out the original flow in *f* and relocate it. Afterward, we insert the overflow flow into *f*. This will only add the frequency of mice flow to the elephant flow erroneously or other mice flows, without mistakenly adding the elephant flow to the mice flow, which greatly improve the accuracy of the frequency estimation especially the frequencies of elephant flows.

An error is introduced for flows that cannot be inserted normally after the maximum number of kicks. However, the probability that cannot be inserted in each mapped bucket is $\frac{5}{N}$, where N is the buckets number in an array. Only when two buckets that are hashed cannot be inserted, will a flow be kicked out. Since, the probability of a flow is kicked out is $\frac{25}{N^2}$. Hence the error introduced by several kicks is tolerable. In reality, an accurate estimate of the frequency of each flow is unrealistic and impossible.

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 $A_i[h_i(e)][entry_j]$ $(i \in \{1,2\}, 1 \leq j \leq 4)$. If matched, we return the counters value of the corresponding entry. Otherwise, we return $A_i[h_i(e)][entry_1].counter$ randomly. That is because serious hash collisions result in no matching fingerprint, but we can ensure that the flow to be queried is definitely placed in one of the buckets which the flow *e* is mapped. According to the insertion algorithm,the flow with a severe hash conflict stored in $entry_1$ of the mapped buckets randomly. The Algorithm of query is given in Figure3.

Deletion: The deletion operation of our Cuckoo Counter is simple. We also compute two indexes of a flow *e*, $h_1(e)$ and $h_2(e)$, and scan entries in $A_i[h_i(e)][entry_j]$ $(i \in \{1,2\}, 1 \leq j \leq 4)$. If the same fingerprint of flow *e* in these entries exists, we decrease the corresponding counter by 1. Otherwise, we decrease $A_i[h_i(e)][entry_1].counter$ by 1 randomly. The reason is the same as when querying.

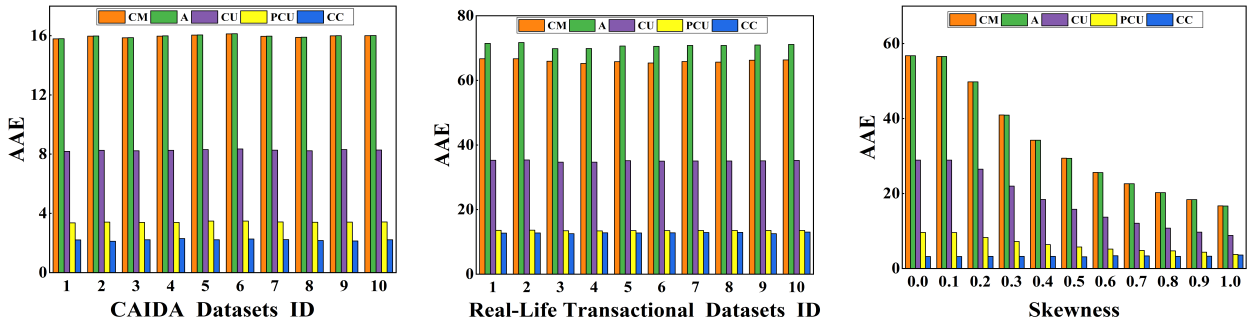
IV. Preformance evaluation

A. Experimental Setup

Datasets: we use the following three datasets.

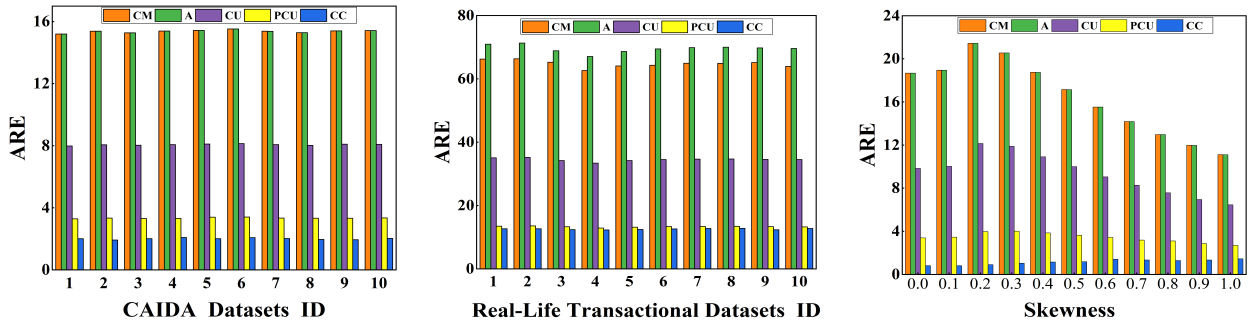
CAIDA Datasets: We use the CAIDA trace which collected in Equinix-Chicago monitor from CAIDA [33]. Our experimental CAIDA datasets are the same as that used with Elastic Sketch [34]. Due to the results by only using source IP address are extremely similar to by using 5-tuple (Source IP, Destination IP, Source Port, Destination Port and Protocol) as flow ID. We identify each flow by its source IP address (4 bytes).

Real-Life Transactional Datasets: We download the Real-Life Transactional dataset called WebDocs from website [35]. This dataset is built from a spidered collection of web html documents. The more details about the dataset are in [36]. Since the dataset is too large, we cut it into sub-datasets with a size of 102MB. The frequency ranges from 1 to 5349.



(a) caida datasets (b) real-life transactional datasets (c) synthetic datasets with different skewness

Fig. 6. Comparison of average absolute error(aae) on different datesets.



(a) caida datasets (b) real-life transactional datasets (c) synthetic datasets with different skewness

Fig. 7. Comparison of average relative error(are) on different datesets.

Synthetic Datasets: We use the Zipfgen program to generate 11 stream datasets (skewness from 0 to 1.0, with intervals of 0.1) that follow the zipf distribution. Each dataset is fixed one million flows and distinct flows with different numbers. The sizes of flows are 13 bytes, and the maximum frequency of flows in each dataset are $16 \sim 55361$.

Parameter Setting: We implement our Cuckoo Counter in C++ [30]. We compare Cuckoo Counter with classic CM sketch, CU sketch and the recently Augmented sketch,Pyramid sketch. Because the PCU sketch has the best performance, we use PCU sketch as the representative of Pyramid sketch in our experiments. We allocate 100KB of memory size to each experiment. The size of CM sketch, CU sketch and Augmented sketch entries are 16 bits. CM sketch and CU sketch allocate 4 arrays and use 4 32-bit Bob hash [37] functions to flows mapping. The Augmented sketch consists of the widely used CM sketch and a filter. The filter will allocate about 0.4KB additional memory, and the CM sketch of Augmented sketch also includes 4 arrays and 4 32-bit Bob hash functions. All entries of the PCU sketch are 4 bits, and the number of mapped entries is 4. The PCU sketch use 1 64-bit Bob hash function. Our Cuckoo Counter has three kinds of entries and two arrays. The memory sizes of these counters in entries are 4 bits, 8 bits, 16 bits respectively, and the fingerprints are 8 bits. We also use a 64-bit Bob hash to find two candidate buckets.

Test Platform: we performed all the experiments on the server NF5 280M4. The server has 12 core CPUs (24 threads, Intel (R) Xeon (R) CPU E5-2620 v3 @2.40 GHz) and 32GB total DRAM memory.

B. Metrics

Three parameters are mainly evaluated in our experiments, Throughput, Average Absolute Error (AAE), and Average Relative Error (ARE).

Throughput: Throughput is used to measure the processing speed of the algorithm and is estimated by the running time of the algorithm. Its estimated by the formula $\frac{N}{T}$, where N is the number of flows, T is the running time. We use millions of per second (Mps) to represent throughput.

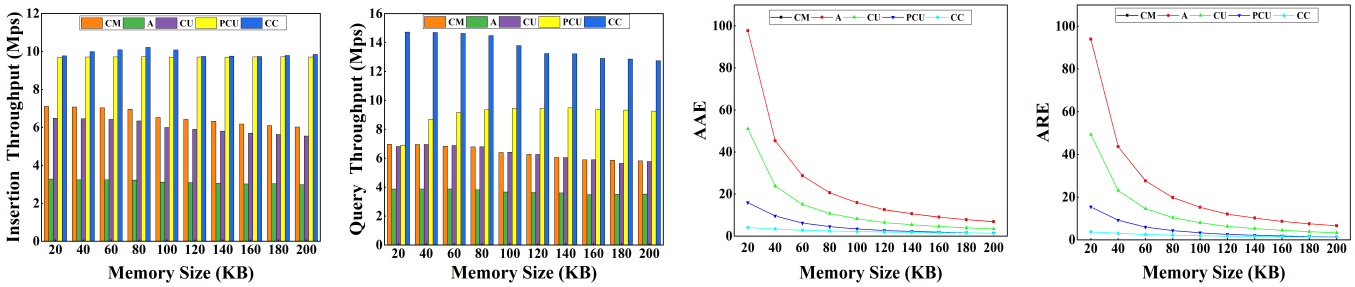
AAE: AAE is defined as $\frac{1}{\psi}\Sigma_{(e_i \in \psi)}|f_i - \tilde{f}_i|$, where f_i is the real frequency of flow e_i, $\tilde{f}_i$ is the estimated frequency, and ψ is the number of different flows.

ARE: ARE is defined as $\frac{\frac{1}{\psi}\Sigma_{(e_i \in \psi)}|f_i - \tilde{f}_i|}{\tilde{f}_i}$. These parameters in the formula have the same meaning as in AAE.

C. Performance

In this part, we illustrate the performance of our Cuckoo Counter by insertion throughput, query throughput, AAE and ARE. We use CC as an abbreviation of Cuckoo Counter, while CM, A, CU, PCU are used as the abbreviations of the CM sketch, Augmented sketch, CU sketch and PCU sketch respectively in the figures.

Insertion Throughput: We use 10 CAIDA sub-traces to test. The Figure4(a) shows that the insertion throughput of



(a) insertion throughput (b) query throughput (c) aae (d) are

Fig. 8. Comparison of insertion throughput, query throughput, aae and are on different memory sizes.

Cuckoo Counter is 1.56, 3.27, 1.70 times higher than the insertion throughput of the CM sketch, Augmented sketch and CU sketch severally, and slightly higher than the PCU sketch. As Figure4(b) displays, the insertion throughput of Cuckoo Counter is 2.64, 4.0, 2.79, 1.14 times higher than the insertion throughput of the CM sketch, Augmented sketch, CU sketch and PCU sketch on 10 Real-Life Transactional Datasets. We also use the skewness datasets (skewness from 0 to 1.0, with intervals of 0.1) to experiment. The results show that the insertion throughput of Cuckoo Counter is 1.23∼1.75, 3.22∼3.92, 1.43∼1.86, 0.96∼1.41 times higher than the insertion throughput of the CM sketch, Augmented sketch, CU sketch and PCU sketch in Figure4(c).

We can see that when skewness is 0 or 0.1, the insertion throughput of Cuckoo Counter is merely smaller than PCU sketch. That is because we fixed 1 million flows on each skewness datasets. The smaller skewness means more different flows in the data stream, causing in more kicked-out flows when insertion. It will reduce the insertion throughput of Cuckoo Counter. Cuckoo Counter has a higher insertion throughput than CM sketch and CU sketch because Cuckoo Counter only needs to hash three times (an average of 1.7 times was shown in our experiment) in the worst case(when the memory is nearly full), while CM sketch and CU sketch four times per insert.

Query throughput: As Figure5(a) shows, the query throughput of Cuckoo Counter is 2.18, 3.70, 2.18, 1.50 times higher than the CM sketch, Augmented sketch CU sketch and PCU sketch on different CAIDA datasets. As Figure5(b) shows, the query throughput of Cuckoo Counter is 2.98, 3.94, 2.90, 1.58 times higher than the CM sketch, Augmented sketch CU sketch and PCU sketch on different Real-Life Transactional Datasets. As Figure5(c) shows, the query throughput of Cuckoo Counter is 2.10, 4.12, 1.96, 1.72 times higher than the CM sketch, Augmented sketch, CU sketch and PCU sketch on skewness datasets.

Due to Cuckoo Counter only need check two buckets when querying a flow . The query throughput of Cuckoo Counter is better performance than other four sketches.

AAE: Our experimental results show that on different CAIDA datasets, the AAEs of CM sketch, Augmented sketch, CU sketch and PCU sketch are 7.14, 7.14, 3.70, 1.52 times higher than the AAE of the Cuckoo Counter in Figure6(a). As Figure6(b) shows, the AAEs of CM sketch, Augmented sketch, CU sketch and PCU sketch are 5.23, 5.60, 2.77, 1.06 times higher than the AAE of the Cuckoo Counter on different Real-Life Transactional datasets. As Figure6(c) shows that on different skewness datasets, the AAEs of CM sketch, Augmented sketch, CU sketch and PCU sketch are 4.65∼17.70, 4.64∼17.68, 2.45∼9.02,1.05∼3.00 times higher than the AAE of the Cuckoo Counter.

ARE: Our experimental results show that on different CAIDA datasets, the AREs of CM sketch, Augmented sketch, CU sketch and PCU sketch are 7.56, 7.56, 3.97,1.64 times higher than the ARE of the Cuckoo Counter in Figure7(a). The results show that on different Real-Life Transactional datasets, the AREs of CM sketch, Augmented sketch, CU sketch and PCU sketch are 5.23, 5.60, 2.77, 1.07 times higher than the ARE of the Cuckoo Counter in Figure7(b). the results show that on different skewness datasets, the AREs of CM sketch, Augmented sketch, CU sketch and PCU sketch are 7.67∼23.08, 7.66∼23.08, 4.46∼12.16, 1.86∼4.20 times higher than the ARE of the Cuckoo Counter in Figure7(c).

Memory size: We test the insertion throughout, query throughout, AAE and ARE as memory size changed on CAIDA datasets. We set the memory size from 20KB to 200KB at each interval of 20KB. The relationship between insertion throughput, query throughput, AAE and ARE and the memory size displays in Figure8. We can observe that when memory is smaller, Cuckoo Counter provides better performance than the other four sketches.

V. Conclution and Future Work

In this paper, we propose a novel framework for per-flow frequency estimation in network measurement, called Cuckoo Counter. Experimental results demonstrate that our framework provides higher accuracy and speed than the sketch-based methods of per-flow frequency estimation in network traffic. Meanwhile, since most of the errors of Cuckoo Counter come from mice flows, Cuckoo Counter provides more accurate statistics of elephant flows than mice flows. Our Cuckoo Counter is extremely suitable for heavy hitter detection with high accuracy. In the future, we will carry out theoretical analysis of Cuckoo Counter and apply it to more scenarios, such as heavy hitter detection.

REFERENCES

[1] G. S. Manku and R. Motwani, "Approximate frequency counts over data streams," in *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*. Elsevier, 2002, pp. 346–357.

[2] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One sketch to rule them all: Rethinking network flow monitoring with univmon," in *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM, 2016, pp. 101–114.

[3] R. Ben Basat, G. Einziger, R. Friedman, M. C. Luizelli, and E. Waisbard, "Constant time updates in hierarchical heavy hitters," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 2017, pp. 127–140.

[4] C. Estan and G. Varghese, "New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice," *ACM Transactions on Computer Systems (TOCS)*, vol. 21, no. 3, pp. 270–313, 2003.

[5] X. Li, F. Bian, M. Crovella, C. Diot, R. Govindan, G. Iannaccone, and A. Lakhina, "Detection and identification of network anomalies using sketch subspaces," in *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*. ACM, 2006, pp. 147–152.

[6] B. Krishnamurthy, S. Sen, Y. Zhang, and Y. Chen, "Sketch-based change detection: methods, evaluation, and applications," in *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*. ACM, 2003, pp. 234–247.

[7] M. Yoon, T. Li, S. Chen, and J.-K. Peir, "Fit a spread estimator in small memory," in *IEEE INFOCOM 2009*. IEEE, 2009, pp. 504–512.

[8] G. Cormode, "Sketch techniques for approximate query processing," *Foundations and Trends in Databases. NOW publishers*, 2011.

[9] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic: measurements & analysis," in *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement*. ACM, 2009, pp. 202–208.

[10] A. Kumar, J. Xu, and J. Wang, "Space-code bloom filter for efficient per-flow traffic measurement," *IEEE Journal on Selected Areas in Communications*, vol. 24, no. 12, pp. 2327–2339, 2006.

[11] Y. Lu, A. Montanari, B. Prabhakar, S. Dharmapurikar, and A. Kabbani, "Counter braids: a novel counter architecture for per-flow measurement," *ACM SIGMETRICS Performance Evaluation Review*, vol. 36, no. 1, pp. 121–132, 2008.

[12] T. Li, S. Chen, and Y. Ling, "Per-flow traffic measurement through randomized counter sharing," *IEEE/ACM Transactions on Networking (ToN)*, vol. 20, no. 5, pp. 1622–1634, 2012.

[13] Y. Zhou, P. Liu, H. Jin, T. Yang, S. Dang, and X. Li, "One memory access sketch: a more accurate and faster sketch for per-flow measurement," in *GLOBECOM 2017-2017 IEEE Global Communications Conference*. IEEE, 2017, pp. 1–6.

[14] P. Roy, A. Khan, and G. Alonso, "Augmented sketch: Faster and more accurate stream processing," in *Proceedings of the 2016 International Conference on Management of Data*. ACM, 2016, pp. 1449–1463.

[15] A. Dobra, M. Garofalakis, J. Gehrke, and R. Rastogi, "Processing complex aggregate queries over data streams," in *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. ACM, 2002, pp. 61–72.

[16] G. Cormode and M. Hadjieleftheriou, "Finding frequent items in data streams," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1530–1541, 2008.

[17] G. Cormode, M. Garofalakis, P. J. Haas, C. Jermaine *et al.*, "Synopses for massive data: Samples, histograms, wavelets, sketches," *Foundations and Trends® in Databases*, vol. 4, no. 1–3, pp. 1–294, 2011.

[18] C. C. Aggarwal and P. S. Yu, "On classification of high-cardinality data streams," in *Proceedings of the 2010 SIAM International Conference on Data Mining*. SIAM, 2010, pp. 802–813.

[19] A. Chen, Y. Jin, J. Cao, and L. E. Li, "Tracking long duration flows in network traffic," in *2010 Proceedings IEEE INFOCOM*. IEEE, 2010, pp. 1–5.

[20] G. Cormode and M. Garofalakis, "Sketching streams through the net: Distributed approximate query tracking," in *Proceedings of the 31st international conference on Very large data bases*. VLDB Endowment, 2005, pp. 13–24.

[21] D. Thomas, R. Bordawekar, C. C. Aggarwal, and S. Y. Philip, "On efficient query processing of stream counts on the cell processor," in *2009 IEEE 25th International Conference on Data Engineering*. IEEE, 2009, pp. 748–759.

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

[23] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," in *International Colloquium on Automata, Languages, and Programming*. Springer, 2002, pp. 693–703.

[24] R. Schweller, Z. Li, Y. Chen, Y. Gao, A. Gupta, Y. Zhang, P. A. Dinda, M.-Y. Kao, and G. Memik, "Reversible sketches: enabling monitoring and analysis over high-speed data streams," *IEEE/ACM Transactions on Networking (ToN)*, vol. 15, no. 5, pp. 1059–1072, 2007.

[25] Y. Li, R. Miao, C. Kim, and M. Yu, "Flowradar: A better netflow for data centers," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, 2016, pp. 311–324.

[26] D. M. Powers, "Applications and explanations of zipf's law," in *Proceedings of the joint conferences on new methods in language processing and computational natural language learning*. Association for Computational Linguistics, 1998, pp. 151–160.

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

[28] R. Pagh and F. F. Rodler, "Cuckoo hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.

[29] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo filter: Practically better than bloom," in *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. ACM, 2014, pp. 75–88.

[30] "The source codes of cuckoo counter." https://github.com/OceanTaraxa/Cuckoo_Counter_Framework.git.

[31] A. Metwally, D. Agrawal, and A. El Abbadi, "Efficient computation of frequent and top-k elements in data streams," in *International Conference on Database Theory*. Springer, 2005, pp. 398–412.

[32] E. D. Demaine, A. López-Ortiz, and J. I. Munro, "Frequency estimation of internet packet streams with limited space," in *European Symposium on Algorithms*. Springer, 2002, pp. 348–360.

[33] "The caida traces." http://www.caida.org/data/overview/.

[34] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, "Elastic sketch: Adaptive and fast network-wide measurements," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. ACM, 2018, pp. 561–575.

[35] "Real-life transactional dataset." http://fimi.ua.ac.be/data/.

[36] "Webdoca dataset." http://fimi.uantwerpen.be/data/webdocs.pdf.

[37] "Hash website." http://burtleburtle.net/bob/hash/evahash.html.