Pyramid Family: Generic Frameworks for Accurate and Fast Flow Size Measurement

Yuanpeng Li*†, Xiang Yu*, Yilong Yang‡, Yang Zhou§, Tong Yang*†, Zhuo Ma‡, Shigang Chen¶

Abstract—Sketches, as a kind of probabilistic data structures, have been considered as the most promising solution for network measurement in recent years. Most sketches do not work well for skewed network traffic. To address this problem, we propose a family of sketch frameworks, namely the Pyramid family. The first member of our Pyramid family is the S-Pyramid framework, which includes two techniques: counter-pair sharing for high accuracy, and word acceleration for fast speed. The second member of our Pyramid family is the Mini-Pyramid framework, which projects the S-Pyramid framework into one counter, bringing more flexibility in application while keeping the accuracy. To demonstrate the generality of our Pyramid family, we apply both frameworks to sketches of CM, CU, Count, and Augmented. To demonstrate the flexibility of the Mini-Pyramid framework, we further apply Mini-Pyramid to SBF and the On-Off sketch. The experimental results show that, the S-Pyramid framework can reduce the ARE by up to 7.12 times compared with the original sketches, while improving the throughput by up to 2.37 times; the Mini-Pyramid framework can reduce the ARE by up to 29.2 times, at the cost of 21.3% lower throughput on average.

Index Terms—Network Measurement, Flow Size Measurement, Sketches, Pyramid Family, Mini-Pyramid Framework

1 Introduction

1.1 Background and Motivation

With the rapid development of the Internet and emergence of new networking paradigms such as SDN (Software-Defined Networking) and TSN (Time-Sensitive Networking), traffic measurement becomes increasingly important to support novel network functions and detect anomalies. One of the most fundamental measurements is flow size measurement. Flow size measurement is the base of many network applications, including alleviation of network congestion [2], [3], SYN flooding attack detection [4], [5], heavy hitter identification [6], [7], heavy change detection [8], [9], packet loss detection [10], [11], traffic burst detection [12], [13], network accounting [14], [15], and more [16], [17], [18].

There are two challenges in network measurement. The first challenge is the limited processing time and memory space. As for high-speed network traffic, any practical solution to per-flow measurement should process each incoming packet at line rate, while ensuring high measurement accuracy. This is challenging to achieve in modern high-speed networks where packet forwarding is performed mostly in hardware platform, requiring to minimize per-packet overhead. To prevent measurement from becoming a bottleneck, it is highly desirable to implement such functions in on-chip memory [19], [20], [21], [22] such as L2/L3 caches on network processors or Block RAM in FPGA. However, on-chip memory has a limited size, imposing a great challenge for the design of extremely compact data structures that can accurately record information for all flows.

The second challenge is the skewed network traffic. It is well known that the flow size distribution tends to be highly *skewed* [14], [15], [23], [24], [25], [26], *i.e.*, most flows are small and a few flows are extremely large. They are called *mice flows* and *elephant flows*, respectively. For example, in a half-hour CAIDA traffic trace [27], 45% of all flows have fewer than 2^2 packets, 94% of all flows have fewer than 2^4 packets, while the size [1] of the largest flow is close to 2^{20}. The skewed traffic poses a challenge for the measurement. Suppose we use counters to record flow size information. On the one hand, if we use 2-bit counters, we can only estimate flow size up to 2^2-1. On the other hand, if we use 32-bit counters, the number of counters will be reduced by 93% (with the same total memory), leading to poor accuracy. Therefore, it's quite challenging to assign suitable counter size and make better use of the memory.

1.2 Prior Art and Limitations

Sketches, as a kind of probabilistic data structure, have been considered as the most promising solution for network measurement in recent years, because they greatly optimize the speed and memory usage at the cost of small error. In other words, most sketches address the challenge of limited processing time and memory space. We divide the sketches into two categories. The first category does not address the latter challenge. They adopt unified big counters, ignoring the skewness of the flow size distribution, and exchanging

*Department of Computer Science and Technology, and National Engineering Laboratory for Big Data Analysis Technology and Application, Peking University, Beijing, China.
†Peng Cheng Laboratory, Shenzhen, China.
‡School of Cyber Engineering, Xidian University, Xi'an, China.
§Department of Computer Science, Harvard University, Brighton, MA, USA.
¶Department of Computer and Information of Science and Engineering, University of Florida, Gainesville, FL, USA.
Tong Yang (yangtongemail@gmail.com) and Zhuo Ma (mazhuo@mail.xidian.edu.cn) are the corresponding authors.
The preliminary version of this paper titled "Pyramid Sketch: a Sketch Framework for Frequency Estimation of Data Streams" was published in the proceedings of the 43nd International Conference on Very Large Data Bases(VLDB) [1], Munich, Germany. August, 2017.

1. In this paper, flow size is defined as the number of packets in the flow.

the simplicity with a considerable amount of space waste. Typical sketches include sketches of Count-Min (CM) [28], Conservative Update (CU) [14], and Count (C) [29].

The second category aims to address the latter challenge. Their data structures are separated into two parts, which record elephant flows and mice flows, respectively. The first part is a key-value (KV) table used to precisely record the ID and size of elephant flows. In addition, it uses auxiliary information (*e.g.*, old count [23], negative votes [15]) to determine whether a flow is an elephant flow or a mice flow. The second part is commonly a small CM sketch used to record the size of mice flows. However, these sketches require to adjust many parameters, such as the memory usage of the two parts, different counter sizes in the two parts, thresholds for replacement strategy, the number of hash functions in the second part. The optimal parameters are sensitive to flow size distribution, memory size, computation resources, network bandwidth, *etc.* Therefore, in industrial deployment, engineers are required to have a deep understanding of the sketches, which hinders the deployment process of sketches to the industrial field. Typical sketches include sketches of Augmented (A) [23] and Elastic [15].

1.3 The Proposed Solution

This paper proposes a family of sketch frameworks, namely the Pyramid family. The key idea is to provide a hierarchical structure for sketches. The first member of our Pyramid family is the standard Pyramid (**S-Pyramid**) framework. This framework has two key techniques: counter-pair sharing for high accuracy, and word acceleration for high speed. **1) Counter-pair sharing:** Through this technique, we can use fine-grained counters (*e.g.*, 4-bit counters) to record both small flows and very large flows. More counters will be used if some counters are overflowed. Specifically, counters can expand into the higher layers of the hierarchy as needed. Thus, any sketch applying the S-Pyramid framework can make better use of memory, which can significantly improve the accuracy. **2) Word acceleration:** This technique consists of three parts: word constraint, word sharing, and one hashing. Through this technique, we can just use one hash computation per operation and one memory access per layer for each operation, while keeping high accuracy. Moreover, the processing overhead for most packets is one memory access and one hash computation, which can significantly improve the processing speed.

The advantage of the S-Pyramid framework is that it can significantly improve both accuracy and speed. The disadvantage of the S-Pyramid framework is that it needs to change the structure of the original sketch, limiting its application range. For sketches using counter arrays (*e.g.*, CM [28], CU [14], Count [29], and ASketch [23]), it is easy to apply the S-Pyramid framework (see Section 6.1-6.4). However, for sophisticated sketches with dedicated structure (*e.g.*, Spectral Bloom filter (SBF) [30], On-Off sketches [17]), it is inconvenient to apply the S-Pyramid framework.

In order to address the above limitation, we propose the second member of our Pyramid family, namely the **Mini-Pyramid** framework. In this framework, we project the S-Pyramid framework into one counter, namely the **M-Pyramid counter**. When applying this framework, we simply replace each counter in the original sketches with an M-Pyramid counter. Compared with the S-Pyramid framework, 1) the major advantage of the Mini-Pyramid framework is the flexibility, *i.e.*, it can be applied to any sketches using counters; 2) the major disadvantage of the Mini-Pyramid framework is the lower speed (about 20% lower). In all, there are two members in our Pyramid family: the standard Pyramid (S-Pyramid) framework and the Mini-Pyramid framework. The S-Pyramid framework using counter-pair sharing and word acceleration techniques can achieve high accuracy and high throughput, while the Mini-Pyramid framework sacrifices speed to gain more flexibility.

To demonstrate the generality of our Pyramid family, we implement both frameworks on sketches of CM [28], CU [14], Count [29], and Augmented [23]. To demonstrate the flexibility of the Mini-Pyramid framework, we further implement Mini-Pyramid on SBF [30] and the On-Off sketch [17]. We compare the accuracy and throughput of sketches before and after applying our frameworks on 3 real traces and a series of synthetic traces. The experimental results show that, 1) the S-Pyramid framework can reduce error by up to 7.12 times compared with the original sketches, while improving throughput by up to 2.37 times; 2) the Mini-Pyramid framework can reduce error by up to 29.2 times, at the cost of 21.3% lower throughput on average. All related source codes are released at Github [31].

1.4 Key Contributions

- We propose the S-Pyramid framework, a sketch framework for simultaneously improving the accuracy and throughput through counter-pair sharing and word acceleration technique. We apply the S-Pyramid framework to sketches of CM [28], CU [14], Count [29], and Augmented [23]. The experimental results show that the S-Pyramid framework improves accuracy by up to 7.12 times compared with the original sketches, while improving throughput by up to 2.37 times.
- We propose the Mini-Pyramid framework, a mini version of the S-Pyramid framework that can be more flexibly applied. We apply the Mini-Pyramid framework to sketches of CM [28], CU [14], Count [29], Augmented [23], SBF [30], and On-Off [17]. The experimental results show that the Mini-Pyramid framework can improve accuracy by up to 29.2 times compared to the original sketches, at the cost of 21.3% lower throughput on average.

2 Related Work

It is a fundamental problem in the field of network measurement to estimate flow sizes in real network. The most promising solution to flow size measurement is sketches. Typical sketches include sketches of CM [28], CU [14], Count [29], Augmented [23], Elastic [15], Nitro [32], and more [11], [17], [18], [33], [34], [35], [36], [37], [38], [39]. These sketches can be divided into two categories based on whether the skewed flow size distribution is considered or not.

The first category is sketches not considering the problem brought by skewed flow size distribution, including sketches of CM [28], CU [14], Count [29]. They are simple but with low accuracy. The CM sketch [28] is the most

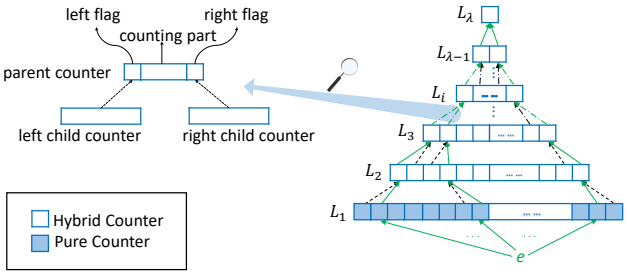


Figure 1. Counter-pair sharing technique.

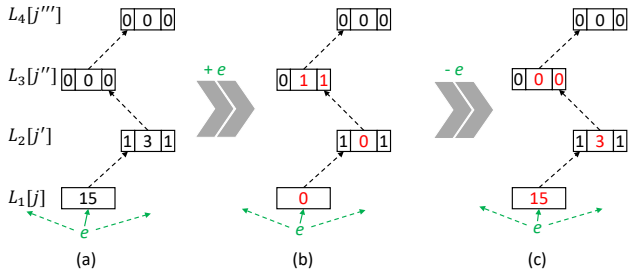


Figure 2. Examples of counter-pair sharing technique.

commonly used sketch in network measurement. It can guarantee no under-estimated error, *i.e.*, the reported value is always no less than the actual size of the queried flow. The CU sketch [14] uses the strategy of conservative update, optimizing the CM sketch for higher accuracy at the cost of not supporting deletion. It still guarantees no under-estimated error. The Count sketch [29] is updated with an equal probability of +1/-1. It achieves unbiased estimation of flow sizes, and is often used in finding top-k frequent flows.

The second category is sketches specially designed for skewed flow size distribution, including sketches of Augmented [23], Elastic [15], and more [36], [37], [38]. They are relatively complex but perform better. The Augmented sketch [23] adds an additional filter to the sketch, aiming to record the elephant flows separately. The Elastic sketch [15] uses Ostracism technique to separate elephant flows from mice flows, and propose a technique to compress sketches.

There are also solutions to flow size measurement using Bloom filter variants. Bloom filters [40] is used to record whether a flow occurs. Its variants expand the bits to counters to support flow size query. Typical Bloom filter variants include Counting Bloom filters (CBF) [41], Spectral Bloom Filters (SBF) [30], and more [42], [43]. Other relevant works include Counter Braids [44], Random Counters [45], and more [46], [47], [48].

3 The S-Pyramid Framework

In this section, we present the first member of our pyramid family, the standard Pyramid framework, S-Pyramid for short. The S-Pyramid framework has two key techniques: **counter-pair sharing** and **word acceleration**. Counter-pair sharing is used to *dynamically assign an appropriate number of bits for different flows with different flow sizes*. Word acceleration can *achieve one memory access and one hash computation for most operations*, significantly accelerating the speed of the sketches. We also present one further optimization method: **Ostrich policy**. Note that we introduce the techniques not in isolation, but one at a time on top of all previous techniques.

3.1 Counter-Pair Sharing

Data Structure: As shown in Figure 1, the S-Pyramid framework consists of λ layers. Let L_i denote the i^{th} layer. L_i consists of w_i counters, where $w_{i+1} = w_i/2$ $(1 \leqslant i \leqslant \lambda - 1)$. Each counter contains δ bits. Let $L_i[j]$ denote the j^{th} counter of L_i. The first layer L_1 is associated with d pairwise independent hash functions $h_i(.)$ $(1 \leqslant i \leqslant d)$. L_i is associated with L_{i+1} in the following way: two adjacent counters at L_i are associated with one counter at L_{i+1}, *i.e.*, $L_i[2j-1]$ and $L_i[2j]$ are associated with $L_{i+1}[j]$. We define $L_{i+1}[j]$ as the **parent counter** of $L_i[2j-1]$ and $L_i[2j]$, $L_i[2j-1]$ as the **left child counter** of $L_{i+1}[j]$, and $L_i[2j]$ as the **right child counter** of $L_{i+1}[j]$. $L_i[2j-1]$ and $L_i[2j]$ are defined as the **sibling counters**.

There are two types of counters: **pure counters** and **hybrid counters**. The first layer L_1 consists of pure counters, while the other layers consist of hybrid counters. The pure counter consists of only a *counting part*, ranging $[0, 2^\delta)$. The δ-bit hybrid counter is split into three parts: a 1-bit *left flag*, a $(\delta-2)$-bit *counting part*, and a 1-bit *right flag*. Let $L_i[j].lflag$, $L_i[j].count$, and $L_i[j].rflag$ denote the three parts of counter $L_i[j]$, respectively. The left flag indicates whether its left child counter is overflowed, while the right flag indicates whether its right child counter is overflowed. The counting part ranging $[0, 2^{\delta-2})$ is used for counting the number of overflows.

Initialization: Initially, all counters at all layers are set to 0, *i.e.*, all counting parts are set to 0 and all flags are set to false.

Insertion: When inserting a packet of flow e, we first compute the d hash functions $h_1(e)$, $h_2(e)$, ..., $h_d(e)$ $(1 \leqslant h_i(.) \leqslant w_1)$ to locate the d *mapped counters* $L_1[h_1(e)]$, $L_1[h_2(e)]$, ..., $L_1[h_d(e)]$ at layer L_1. Different sketches perform different incrementing operations on these d counters. If one of the d mapped counter $L_i[j]$ overflows, we perform the following **carry-in** operation: we set the overflowed counter to 0, and then increment its parent counter. Let $L_{i+1}[j']$ denote the parent counter. Note that all parent counters are hybrid counters. When incrementing $L_{i+1}[j']$, we first set its corresponding flag $L_{i+1}[j'].lflag/rflag$ to true, and then increment the counting part $L_{i+1}[j'].count$. If $L_{i+1}[j'].count$ does not overflow, the insertion ends; otherwise, we repeat the carry-in operation on $L_{i+1}[j']$.

Example I: As shown in Figure 2(a)-(b), each pure counter and hybrid counter contain 4 bits. The counting part in each hybrid counter contains 2 bits. The value of $L_1[j]$, the three parts of $L_2[j']$, $L_3[j'']$, and $L_4[j''']$ are 15, $\langle 1,3,1\rangle$, $\langle 0,0,0\rangle$, and $\langle 0,0,0\rangle$, respectively. Suppose $L_1[j]$ is incremented by 1 and overflows, the carry-in operations are performed as follows:

1) $L_1[j]$ is set to 0;
2) $L_2[j'].lflag$ keeps true;
3) $L_2[j'].count$ is set to 0;
4) $L_3[j''].rflag$ is set to true;
5) $L_3[j''].count$ is incremented to 1.

Deletion: Deletion is the reverse operation of insertion, *i.e.*, decrement the flow size by 1. The S-Pyramid framework

Algorithm 1: ReportVal(i, j_i).

```
1  if i==1 then
2      return L_1[j_1] + ReportVal(i+1, j_{i+1})
3  if L_{i-1}[j_{i-1}] is L_i[j_i]'s left child counter and
     L_i[j_i].lflag = False then
4      return 0
5  if L_{i-1}[j_{i-1}] is L_i[j_i]'s right child counter and
     L_i[j_i].rflag = False then
6      return 0
7  if L_i[j_i].lflag = True and L_i[j_i].rflag = True then
8      return (L_i[j_i].count - 1) × 2^{δ+(i-2)×(δ-2)}
         +ReportVal(i+1, j_{i+1})
9  else
10     return L_i[j_i].count × 2^{δ+(i-2)×(δ-2)}
         +ReportVal(i+1, j_{i+1})
```

supports deletion only if the original sketch supports deletion (*e.g.*, CM [28], Count [29]). When deleting a packet of flow e, we first compute the d hash functions to locate the d mapped counters, and then perform the decrementing operation. *The decrementing operation is exactly the reverse process of incrementing*. Specifically, to decrement a pure counter $L_1[j]$, if it is non-zero, we just decrement it by 1. Otherwise, we perform the **carry-down** operation: set $L_1[j]$ to its maximum value $(2^\delta - 1)$, and then decrement its parent counter recursively. There are three cases when decrement a hybrid counter $L_i[j']$:

1) If $L_i[j'].count$ is larger than 1, we simply decrement it by 1.

2) If $L_i[j'].count$ is 1, we first decrement it to 0, and then set $L_i[j'].lflag/rflag$ to false if the corresponding flag of its parent counter is false.

3) If $L_i[j'].count$ is 0, we set it to its maximum value $(2^{\delta-2}-1)$, and then decrement its parent counter recursively.

Example II: As shown in Figure 2(b)-(c), the value of $L_1[j]$, the three parts of $L_2[j']$, $L_3[j'']$, and $L_4[j''']$ are 0, $\langle 1,0,1\rangle$, $\langle 0,1,1\rangle$, and $\langle 0,0,0\rangle$, respectively. Suppose $L_1[j]$ is decremented by 1 to 0, the carry-down operations are performed as follows:

1) $L_1[j]$ is set to 15;
2) $L_2[j'].lflag$ keeps true;
3) $L_2[j'].count$ is set to 3;
4) $L_3[j''].rflag$ is set to false;
5) $L_3[j''].count$ is decremented to 0.

Query: When querying a flow e, we compute the d hash functions to locate the d mapped counters, and then query the mapped counters respectively. Below we describe how to query a single mapped counter $L_1[j_1]$. Let $L_2[j_2]$, $L_3[j_3]$, ..., $L_\lambda[j_\lambda]$ denote the parent counter and the ancestor counters of $L_1[j_1]$. As shown in Algorithm 1, we recursively assemble the counter value top-down, layer by layer, until the corresponding flag is false. Note that if both left and right flags of $L_i[j_i]$ are true (line 5-8), both its left and right child counters must have overflowed at least once. Therefore, we subtract $L_i[j_i].count$ by 1, thus reduce the over-estimation error incurred by the collision in counter-pair sharing. Let $\mathcal{R}(L_1[j_1])$ denote the final **reported value** $ReportVal(1, j_1)$ for convenience. After obtaining the d reported value, $\mathcal{R}(L_1[h_1(e)])$, $\mathcal{R}(L_1[h_2(e)])$, ..., $\mathcal{R}(L_1[h_d(e)])$, we report the query result based on the specific sketch under use. For example, for CM and CU, we simply report the minimum value among the d reported values.

Example III: As shown in Figure 2(b), the value of $L_1[j]$, the three parts of $L_2[j']$, $L_3[j'']$, and $L_4[j''']$ are 0, $\langle 1,0,1\rangle$, $\langle 0,1,1\rangle$, and $\langle 0,0,0\rangle$. The report value operation is performed as follows:

1) $L_1[j]$ is 0;

2) Both $L_2[j'].lflag$ and $L_2[j'].rflag$ are true, and $L_2[j'].count$ is 0;

3) $L_3[j''].rflag$ is true and $L_3[j''].count$ is 1;

4) $L_4[j'''].lflag$ is false and the recursive operation ends;

5) The reported value is $0 + (0-1) \times 2^4 + 1 \times 2^6 = 48$.

Summary and Analysis: Our counter-pair sharing technique is based on the following insight: the practical datasets are skewed, and the number of elephant flows is much less than mice flow. Therefore, only a small number of counters are overflowed, and in most cases at most one of the sibling counters is overflowed. That means the sharing of the parent counter is almost conflict-free. Therefore, with counter-pair sharing technique, most of the flows can be recorded with the appropriate counter(s) size.

3.2 Word Acceleration

In sketch algorithm, memory access and hash computation are two major bottlenecks in speed [32]. Based on the structure of the S-Pyramid framework, we propose three acceleration methods to improve the throughput while ensuring the accuracy.

Word Constraint Technique: In the word constraint technique, we make two minor modifications: 1) we set the counter size δ to 4 bits; 2) as shown in Figure 3(a)-(b), we constrain the d mapped counters at layer L_1 to a single machine word. In this way, the average number of memory accesses per operation is significantly reduced and the speed is significantly improved. Let the size of a machine word be $\mathcal{W}$ bits. Each machine word contains $\mathcal{W}/\delta$ counters. Therefore, layer L_1 contains $\delta w_1/\mathcal{W}$ machine words. In addition, L_1 is associated with $d+1$ hash functions $h_i(.)$ $(1 \leqslant i \leqslant d+1)$. The first hash function is used to map each flow to a specific word $\mathbf{\Omega}$, and the remaining d hash functions are used to locate the d mapped counters in word $\mathbf{\Omega}$. The operations of insertion, deletion and query remain the same under the word constraint technique.

Our word constraint technique is based on the following facts: 1) In the S-Pyramid framework, each counter is small (*e.g.*, 4 bits), while a machine word is usually 64 bits wide on modern CPUs. 2) The size of a machine word can be much larger on other platforms (*e.g.*, 1024 bits on GPU [49]). Therefore, one machine word can typically contain a reasonably large number of small counters used in the S-Pyramid framework. Obviously, after using the word constraint technique, the average number of memory accesses per operation is reduced to around $1/d$, as all the d mapped counters can be read/written within one memory access. We derive the upper-bound on the average number of memory accesses for each insertion as follows.

When inserting a packet, suppose $\Pr(L_1 \text{ overflows}) < \rho$ and $\Pr(L_{i+1} \text{ overflows} | L_i \text{ overflows}) < \sigma$ $(1 \leqslant i < \lambda)$. The

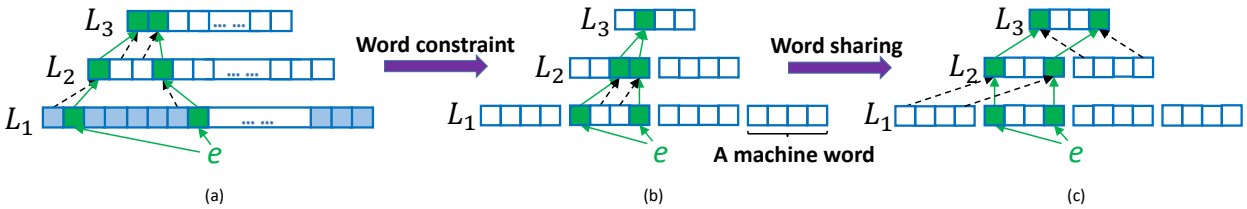


Figure 3. Word constraint and word sharing technique.

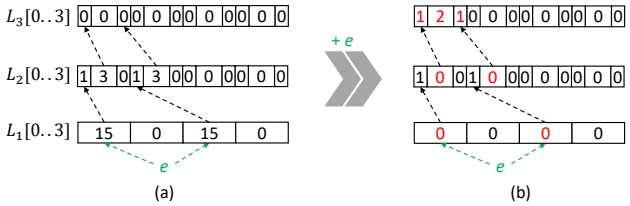


Figure 4. Example of word constraint technique.

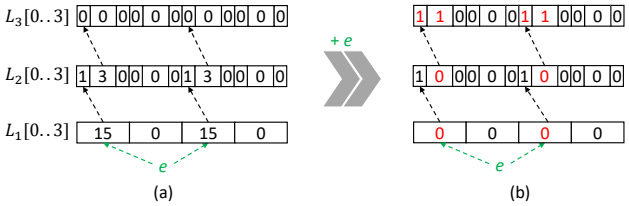


Figure 5. Example of word sharing technique.

average number of memory accesses $\bar{t}$ is determined by the following formula:

$$\bar{t} < 1 + \sum_{k=0}^{\infty} \rho\sigma^k = 1 + \frac{\rho}{1-\sigma} \tag{1}$$

In our experiments on CAIDA trace (see § 7.1), $\rho \approx 0.05$, $\sigma \approx 0.25$. Therefore, the average number of memory accesses for each insertion $\bar{t} \leqslant 1 + 0.05/(1-0.25) \approx 1.07$, which is consistent with our experimental results shown in Figure 18(b).

Example IV: As shown in Figure 4, each word contains 4 counters, and the d mapped counters are constrained to one machine word. Suppose a flow is hashed to $L_1[0]$ and $L_1[2]$, and the counters are incremented by 1 and overflow, the carry-in operations are performed as follows:

1) $L_1[0]$ and $L_1[2]$ are set to 0;

2) $L_2[0].lflag$ and $L_2[1].lflag$ keep true, $L_2[0].count$ and $L_2[1].count$ are set to 0;

3) $L_3[0].lflag$ and $L_3[1].rflag$ are set to true, $L_3[0].count$ is incremented to **2**.

As shown above, applying word constraint technique to the S-Pyramid framework helps reducing memory accesses. However, it also incurs severe accuracy loss. The main reason is that, after implementing the carry-in operation, the probability of collision among counters in the same machine word increases sharply at high layers (see Figure 3(b)). More specifically, given a flow e, its d mapped counters at layer L_1 shares one machine word, while their parent counters shares only half of a word at layer L_2. Their ancestor counters are constrained to smaller and smaller ranges at higher layers, resulting in more and more collisions.

Word Sharing Technique: To address the above issue, we propose a new technique, namely the word sharing technique. The methodology of this technique is making the parent and ancestor counters of the d mapped counters always fall in a constant range, *i.e.*, a machine word, instead of a smaller and smaller range, thus reduce collisions. As shown in Figure 3(b)-(c), our word sharing technique works as follows:

1) Similar to the definition of parent counter, left child counter, and right child counter, we have **parent word**, **left child word**, and **right child word**. The left child word and the right child word are adjacent at layer L_i, sharing the same parent word at the next layer L_{i+1}.

2) The i^{th} counter in the left child word and the i^{th} counter in the right child word share the i^{th} counter in the parent word.

In this way, collisions in counter-pair sharing are alleviated, and the accuracy is significantly improved. As shown in Figure 3(c), with the word sharing technique, the parent and ancestor counters of the d mapped counters always fall in one machine word at each layer. Therefore, no additional collisions occur. Note that word sharing technique keeps the number of counters, and therefore does not result in additional memory waste.

Example V: As shown in Figure 5, suppose a flow is hashed to $L_1[0]$ and $L_1[2]$, and the counters are incremented by 1 and overflow, the carry-in operations are performed as follows:

1) $L_1[0]$ and $L_1[2]$ are set to 0;

2) $L_2[0].lflag$ and $L_2[2].lflag$ keep true, $L_2[0].count$ and $L_2[2].count$ are set to 0;

3) $L_3[0].lflag$ and $L_3[2].lflag$ are set to true, $L_3[0].count$ and $L_3[2].count$ are incremented to **1**.

One Hashing Technique: As mentioned above, hash computation is another speed bottleneck of sketch algorithm. Ideally, only one hash computation is performed for each operation. Towards this goal, we present the one hashing technique. The idea is to *split the value one hash function produces into several segments, and each segment is used to locate a word or a counter*, so as to reduce the hash computation. A hash function usually produces a value of 32 or 64 bits. However, the number of the counters is hardly able to reach the maximum value. Many bits are unused, resulting in computational waste. Given a hash function with 32-bit output, we may use the first 16 bits to locate a word in the S-Pyramid sketch. [2] Suppose a word contains 64 bits and 16

2. We call the sketches applying our frameworks the S-Pyramid sketches and the Mini-Pyramid sketches, respectively.

counters. Locating one of the counters requires 4 hash bits. In total, we need 16 hash bits to locate all 4 counters in the word. In this way, we can use only one hash computation to locate all d mapped counters in an S-Pyramid sketch ($d = 4$) with at most 2^{16} words at the layer L_1. Similarly, we can use a hash function with a 64-bit output to support an S-Pyramid sketch ($d = 4$) with at most 2^{48} words, *i.e.*, 2048 TB memory at layer L_1, which should be large enough for all practical cases.

Summary: With our word acceleration technique, the S-Pyramid framework only needs one memory access and one hash computation for most operations. Although for some flows, multiple layers need to be read/written, which resulting in multiple memory accesses. Fortunately, most flows are mice flows and only access the first layer. Our experimental results on CAIDA show that the average number of memory accesses per insertion can be reduced to 1.07, which is close to one memory access per operation (see Figure 18(b)).

3.3 Further Optimization Method

Ostrich Policy: For sketches that need to get the reported value of the d mapped counters during each insertion (*e.g.*, CU [14]), multiple layers may need to be accessed. To reduce memory accesses, we propose a novel strategy, namely Ostrich policy. The key idea of Ostrich policy is *ignoring the higher layers when getting the reported values of the d mapped counters*. Take the CU sketch as an example. When inserting a packet of flow e, suppose the counter(s) [3] with the *smallest value* among the d mapped counters is $L_1[j]$, we just increment the counter $L_1[j]$ by 1. Note that the *reported value* of $L_1[j]$ is not always the smallest among the d mapped counters. If there are multiple smallest counters, we increment all of them. With Ostrich policy, the insertion speed is significantly improved.

Our experimental results show that Ostrich policy has no negative effect on accuracy (see Section 7.2.1). That may be counter-intuitive. Let's take the CU sketch as an example to illustrate this phenomenon. The CU sketch always increments the smallest counter(s) by 1 in each insertion. However, in some cases, the smallest counter(s) is already over-estimated, *i.e.*, larger than the real value. In such cases, incrementing the smallest counter(s) is not the best strategy, while a new strategy of incrementing the smallest counter(s) with high probability often lead to a better accuracy. Ostrich policy is one efficient implementation of this new strategy.

It is worth noticing that although Ostrich policy has little effect on accuracy, it sacrifices the property of no under-estimated error of the CU sketch. That's because ostrich policy ignore the higher layers and may result in misjudging the minimum counter. Therefore, Ostrich policy can only be used if under-estimated error is acceptable.

4 The Mini-Pyramid Framework

In this section, we present the second member of our Pyramid family, the mini version of the S-Pyramid framework, Mini-Pyramid for short. The Mini-Pyramid framework projects the structure of the S-Pyramid framework into a 32-bit counter, namely the M-Pyramid counter. We focus on 32-bit counters, mainly because they are sufficient to record all flow sizes, and most sketches use 32-bit counters in deployment. From this insight, the 32-bits Mini-Pyramid is more flexible in deployment. For parameter settings, please refer to Section 7.2.2.

3. We use counter(s) because there may be multiple minimum counters.

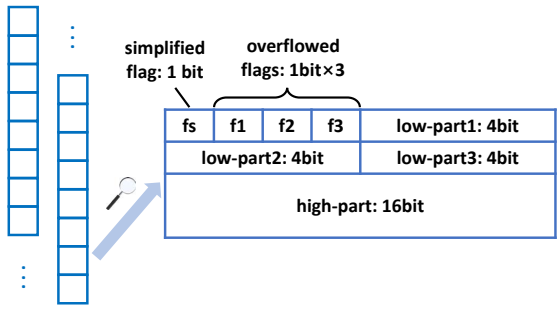


Figure 6. The data structure of the Mini-Pyramid framework.

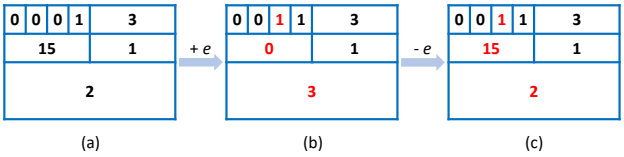


Figure 7. Examples on the operations of the Mini-Pyramid framework.

4.1 Data Structure and Operation

Data Structure: As shown in Figure 6, for the limited memory, our Mini-Pyramid framework consists of only two layers. The first layer consists of three 4-bit counters, we name it the **low-part**; The second layer consists of one 16-bit counter, we name it the **high-part**. Let $L[i]$ denote the i^{th} counter at low-part, and H denote the counter at high-part. We also build three flag bits to indicate whether the corresponding counter is overflowed. Let $flag_i$ denote the i^{th} flag bit.

Insertion: When inserting a packet of flow e, we first compute a hash function $h(e)(1 \leqslant h(.) \leqslant 3)$ to locate the counter $L[h(e)]$ at low-part. Different sketches perform different incrementing operations on the counter. If the counter overflows, we perform the **carry-in** operation: we set $L[h(e)]$ to 0, set the corresponding overflowed flag $flag_{h(e)}$ to true, and increment the counter at high-part H by 1.

Example IV: As shown in Figure 7(a)-(b), $L[2]$, H, and $flag_2$ are 15, 2, and 0, respectively. Suppose $L[2]$ is incremented by 1 and overflows, the carry-in operation is performed as follows:

1) $L[2]$ is set to 0;
2) $flag_2$ is set to true;
3) H is incremented to 3.

Deletion: Like the S-Pyramid framework, the Mini-Pyramid framework supports deletion only if the original sketch supports deletion. When deleting a packet of flow e, we first compute $h(e)$ to locate $L[h(e)]$, and then perform the decrementing operation. If the counter is non-zero, we simply decrement it by 1. Otherwise, we perform the **carry-down** operation: we set $L[h(e)]$ to its maximum value (15), and decrement H by 1. If H is decremented to 0, we then set the overflowed flag $flag_{h(e)}$ to false.

Example V: As shown in Figure 7(b)-(c), $L[2]$, H, and $flag_2$ are 0, 3, and 1, respectively. Suppose $L[2]$ is decremented by 1 to 0, the carry-down operation is performed as follows:

1) $L[2]$ is set to 15;
2) $flag_2$ keeps true;
3) H is decremented to 2.

Query: When querying a flow e, we first compute $h(e)$ to locate $L[h(e)]$. If the overflowed flag $flag_{h(e)}$ is false, we simply report $L[h(e)]$. Otherwise, we assemble H and $L[h(e)]$ and report the value ($H \times 2^4 + L[h(e)]$).

Example VI: As shown in Figure 7(c), $L[2]$, H, and $flag_2$ are 15, 2, and 1, respectively. The query operation on $L[2]$ is performed as follows:

1) $L[2]$ is 15;
2) $flag_2$ is true and H is 2;
3) The reported value is $15 + 2 \times 2^4 = 47$.

Optimization: Like the S-Pyramid framework, the Mini-Pyramid framework can also adapt the **one hashing technique**. When calculating a hash function, we may separate the hash value into two parts, one for locating the M-Pyramid counter, and the other for locating the counter at low-part.

4.2 Simplification

The M-Pyramid counter can record flows of size up to $2^{20} - 1$. However, in some cases (*e.g.*, data center network, see Section 7.1), the maximum flow size exceeds this threshold. To address this issue, we propose the **simplification** operation. As shown in Figure 6, the M-Pyramid counter remains 1 bit for the simplified flag. Let $flag_s$ denote the flag. When $flag_s$ is true, the counter is simplified to a 31-bit general counter. When perform an operation, we first check $flag_s$. If $flag_s$ is false, we treat the counter as an M-Pyramid counter. Otherwise, we treat it as a general counter. Note that if H overflows when incrementing, we have to set $flag_s$ to true, and set the general counter to the maximum value of the M-Pyramid counter plus 1 (2^{20}).

5 Mathematical Analyses

In this section, we first derive the correct rate and error bound of SP_{CM} [4]. We then prove that the error of MP_{CM} is always lower than CM.

5.1 Analyses on the SP_{CM} Sketch

In this subsection, we only give the theorems and the proof of error bound. For further mathematical analyses on the S-Pyramid framework, please refer to the original paper [1].

Theorem 1 (No Under-estimation Error)**.** *The SP_{CM} sketch has no under-estimation error, where under-estimation error means that the reported value is smaller than the real value.*

Theorem 2 (Correct Rate)**.** *Given a trace with N distinct flows, let N_i ($1 \leqslant i \leqslant \lambda$) denote the number of distinct flows that the carry-in operations of d mapped counters end exactly at layer L_i. Without loss of generality, we assume that all d carry-in operations end at the same layer.*

4. We call the CM sketch applying the S-Pyramid and Mini-Pyramid framework the SP_{CM} and MP_{CM} sketches, respectively for short. Similarly, we have SP_{CU}, MP_{CU}, *etc.*

Let P_i^{acc} denote the probability that one arbitrary counter corresponding with one arbitrary flow records the accurate value at layer L_i.

$$P_i^{acc} = \left(1 - \frac{2^{i-1}}{w_1}\right)^{(\Phi_i \times N - 1) \times d} \times \left(1 - \frac{\delta}{\mathcal{W}}\right)^{d-1} \quad (1 \leqslant i \leqslant \lambda) \tag{2}$$

where

$$\Phi_i = \begin{cases} \frac{\sum_{k=i}^{\lambda} N_k}{N} & (1 \leqslant i \leqslant \lambda) \\ 0 & (i = \lambda + 1) \end{cases} \tag{3}$$

Let $\mathcal{P}$ denote the expectation of the probability that one arbitrary counter at layer L_1 reports the accurate result.

$$\mathcal{P} = \sum_{k=1}^{\lambda} \left[(\Phi_k - \Phi_{k+1}) \times \prod_{l=1}^{k} P_l^{acc} \right] \tag{4}$$

Let C_r denote the correct rate of the estimation for one arbitrary flow.

$$C_r = 1 - (1 - \mathcal{P})^d \tag{5}$$

Theorem 3 (Error Bound)**.** *For an arbitrary flow e_i, let f_i and $\hat{f}_i$ denote its actual and estimated flow size. Let N denote the number of distinct flows and V denote the sum of the real flow sizes of all flows, i.e., $V = \sum_{k=1}^{N} f_i$. The Φ_i is defined in Formula 3. Given a small variable ϵ, we have the following guarantee with probability at least $1 - \left(\frac{\Delta}{\epsilon}\right)^d$: ($\Delta$ is a constant relying on $N, \Phi_i, w_1, d, \mathcal{W}$ and δ):*

$$\hat{f}_i \leqslant f_i + \epsilon \times V \tag{6}$$

Proof. Each layer L_i ($2 \leqslant i \leqslant \lambda$) in the SP_{CM} sketch can be considered to correspond with d *virtual* hash functions $h_1^i(.)$, $h_2^i(.)$, ..., $h_d^i(.)$ ($1 \leqslant h(.) \leqslant w_i$), which are determined by the initial d hash functions at the first layer L_1 and the carry-in operation. Note that in this section, we denote the initial d hash functions by $h_1^1(.)$, $h_2^1(.)$, ..., $h_d^1(.)$ ($1 \leqslant h(.) \leqslant w_1$).

We define an indicator variable $I_{i,j,k,l}$, which is 1 if $h_j^l(e_i) = h_j^l(e_k)$, and 0 otherwise. Due to the pairwise independent hash functions, the expectation of $I_{i,j,k,l}$ can be derived as follows:

$$\begin{aligned} E\left(I_{i,j,k,l}\right) = & \frac{1}{w_l} \times \frac{\Phi_l \times N \times d - d}{\Phi_l \times N \times d - 1} \\ & + \frac{1}{\mathcal{W}/\delta} \times \frac{d-1}{\Phi_l \times N \times d - 1} (1 \leqslant l \leqslant \lambda) \end{aligned} \tag{7}$$

For convenience, let E_l denote $E\left(I_{i,j,k,l}\right)$. We define the variable $X_{i,j}$ as follows:

$$X_{i,j} = \sum_{l=1}^{\lambda} \left[(\Phi_l - \Phi_{l+1}) \times \sum_{k=1}^{N} (f_k \times I_{i,j,k,l}) \right] \tag{8}$$

$X_{i,j}$ is non-negative. It reflects the expectation of the error caused by the collisions happening at all the layers when querying one arbitrary counter at layer L_1. In other words, we have:

$$\mathcal{R}(L_1[h_j^1(e_i)]) = f_i + X_{i,j} \tag{9}$$

The expectation of $X_{i,j}$ is calculated as follows:

$$\begin{aligned}
E(X_{i,j}) &= E\left\{\sum_{l=1}^{\lambda}\left[(\Phi_l-\Phi_{l+1})\times\sum_{k=1}^{N}(f_k\times I_{i,j,k,l})\right]\right\}\\
&=\sum_{l=1}^{\lambda}\left[(\Phi_l-\Phi_{l+1})\times\sum_{k=1}^{N}(f_k\times E(I_{i,j,k,l}))\right]\\
&=\sum_{l=1}^{\lambda}\left[(\Phi_l-\Phi_{l+1})\times\sum_{k=1}^{N}(f_k\times E_l)\right]\\
&=\sum_{l=1}^{\lambda}\left[(\Phi_l-\Phi_{l+1})\times E_l\times\sum_{k=1}^{N}f_k\right]\\
&=\sum_{l=1}^{\lambda}\left[(\Phi_l-\Phi_{l+1})\times E_l\times V\right]\\
&=V\times\sum_{l=1}^{\lambda}\left[(\Phi_l-\Phi_{l+1})\times E_l\right]\\
&=V\times\Delta
\end{aligned} \tag{10}$$

Where Δ denotes $\sum_{l=1}^{\lambda}\left[(\Phi_l-\Phi_{l+1})\times E_l\right]$). Thus,

$$V=\frac{E(X_{i,j})}{\Delta} \tag{11}$$

Then, by the Markov inequality, we get:

$$\begin{aligned}
&Pr\left[\hat{f}_i\geqslant f_i+\epsilon\times V\right]\\
&=Pr\left[\forall_j,\ \mathcal{R}(L_1[h_j^1(e_i)])\geqslant f_i+\epsilon\times V\right]\\
&=Pr\left[\forall_j,\ f_i+X_{i,j}\geqslant f_i+\epsilon\times V\right]\\
&=Pr\left[\forall_j,\ X_{i,j}\geqslant\epsilon\times V\right]\\
&=Pr\left[\forall_j,\ X_{i,j}\geqslant\epsilon\times\frac{E(X_{i,j})}{\Delta}\right]\\
&=Pr\left[\forall_j,\ \frac{X_{i,j}}{E(X_{i,j})}\geqslant\frac{\epsilon}{\Delta}\right]\\
&\leqslant\left\{E\left[\frac{X_{i,j}}{E(X_{i,j})}\right]/\frac{\epsilon}{\Delta}\right\}^d\\
&=\left(\frac{\Delta}{\epsilon}\right)^d
\end{aligned} \tag{12}$$

□

5.2 Analysis on the MP_{CM} Sketch

Theorem 4. *For an arbitrary flow e_i with real flow size f_i, let $\mathcal{C}$ be one of the d mapped counters in the CM sketch. Let $\hat{f}_i^{MP}$ and $\hat{f}_i^{CM}$ denote the reported value of $\mathcal{C}$ with and without applying the Mini-Pyramid framework, respectively. We have*

$$f_i\leqslant\hat{f}_i^{MP}\leqslant\hat{f}_i^{CM} \tag{13}$$

Proof. The structure of Mini-Pyramid is the same as a 2-layer S-Pyramid, thus $f_i\leqslant\hat{f}_i^{MP}$, according to Theorem 1.

Without loss of generality, suppose the located low-part in $\mathcal{C}$ is $L[1]$. Regardless of e_i is overflowed or not, $\hat{f}_i^{MP}\leqslant H\times 2^4+L[1]$, while $\hat{f}_i^{CM}$ is equal to the sizes of all flows mapped to $\mathcal{C}$. Note that H in Mini-Pyramid counts all overflows in $L[1]$, $L[2]$, and $L[3]$, so if we have the flow trace, we can divide H into 3 parts: $H[1]$, $H[2]$, and $H[3]$ ($H=H[1]+H[2]+H[3]$), where $H[i]$ is the number of overflows in $L[i]$ ($1\leqslant i\leqslant 3$). Obviously, the sizes of the flows mapped to sub-counter $L[i]$ are equal to $H[i]\times 2^4+L[i]$. Therefore, the sizes of all flows mapped to $\mathcal{C}$ ($\hat{f}_i^{CM}$) are equal to $\sum_{i=1}^{3}H[i]\times 2^4+L[i]=H\times 2^4+\sum_{i=1}^{3}L[i]\geqslant H\times 2^4+L[1]\geqslant\hat{f}_i^{MP}$. □

As shown in Figure 11(a), the MP_{CM} sketch always achieves more accuracy than the CM sketch, even if the flow size distribution is not skewed.

6 Case Studies

In this section, we introduce 6 typical sketches, and show how our frameworks apply to them. For the former 4 sketches, we apply both S-Pyramid and Mini-Pyramid frameworks. The Mini-Pyramid framework is more flexible, and we further apply it to the latter 2 sketches.

6.1 CM sketches

A CM sketch [28] consists of d arrays, A_1, ..., A_d. Each array contains w counters and is associated with a hash function $h_i(.)$ ($1\leqslant i\leqslant d$, $1\leqslant h_i(.)\leqslant w$). When inserting a packet of flow e, the CM sketch increments all d mapped counters $A_1[h_1(e)]$, ..., $A_d[h_d(e)]$ by 1. When querying a flow e, the CM sketch reports the minimum value among the d mapped counters.

Applying S-Pyramid: For CM, we apply both counter-pair sharing and word acceleration techniques.

Applying Mini-Pyramid: Given a CM sketch, we simply replace each counter in the counter arrays of the CM sketch by an M-Pyramid counter.

6.2 CU sketches

The CU sketch [14] only modifies the insertion process of the CM sketch. The data structure and the query process of the CU sketch are exactly the same as that of the CM sketch. When inserting a packet, the CU sketch only increments the minimum counter(s) among the d mapped counters.

Applying S-Pyramid: For CU, we apply counter-pair sharing, word acceleration techniques, and Ostrich policy.

Applying Mini-Pyramid: Given a CU sketch, we replace each counter in the counter arrays of the CU sketch by an M-Pyramid counter.

6.3 Count Sketches

Similar to the CM sketch, the Count (C) sketch [29] consists of d arrays, A_1, ..., A_d. Each array contains w counters but is associated with two hash functions $h_i(.)$ and $g_i(.)$. $h_i(.)$ is uniformly mapped to one of the w counters, whereas $g_i(.)$ is mapped to -1 or +1 with the same probability. When inserting a packet of flow e, for the i^{th} mapped counter $A_i[h_i(e)]$ ($1\leqslant i\leqslant d$), the Count sketch calculates $g_i(e)$ and adds/subtracts the counter accordingly. When querying a flow e, the Count sketch reports the median value of $A_1[h_1(e)]\times g_1(e)$, ..., $A_d[h_d(e)]\times g_d(e)$.

Applying S-Pyramid: For the Count sketch, we apply counter-pair sharing and word acceleration techniques. Since the S-Pyramid framework does not support negative numbers, we also add bit arrays to record the sign of each counter in the first layer L_1.

Applying Mini-Pyramid: Since the M-Pyramid counter does not support negative numbers, for a Count sketch, besides replacing counters by the M-Pyramid counters, we also add bit arrays to record the sign of each low part in the M-Pyramid counters.

6.4 Augmented Sketches

The Augmented (A) sketch [23] adds an additional filter to an existing sketch (*e.g.*, CM). The filter is an array records the ID, new count, and old count of the elephant flows. When inserting a packet of flow e, ASketch first checks the filter. If e exists in the filter, or the filter is not yet full, ASketch simply updates the filter. Otherwise, ASketch inserts e to the sketch and estimates its flow size. If the estimated flow size is larger than the minimum new count in the filter, we replace that flow with e. When querying a flow e, ASketch first checks whether e is in the filter, if so, ASketch reports the new count directly. Otherwise, ASketch reports the estimated value in the sketch.

Applying S-Pyramid: For ASketch, we apply both counter-pair sharing and word acceleration techniques to the sketch, while keeping the filter unchanged.

Applying Mini-Pyramid: Given an ASketch, we only replace the counters in the counter arrays of the sketch by M-Pyramid counters. The counters in the filter remain the same, because there is no conflict in the filter, so the M-Pyramid counter is not needed.

6.5 Spectral Bloom Filters

The Spectral Bloom Filter (SBF) [30] is quite similar to the CM sketch, except that SBF uses only one counter array. SBF proposes two optimizations to increase accuracy, one of which is called Recurring Minimum (RM). It uses a primary SBF and a secondary SBF to reduce error. When performing an insertion or query operation, if the flow has a RM (*i.e.*, two or more mapped counters report the minimum), it only operates the primary SBF. Otherwise, it performs the insertion or query operation on the secondary SBF additionally.

Applying Mini-Pyramid: Given a SBF using RM, we replace all counter in both primary and secondary SBF by the M-Pyramid counters.

6.6 On-Off Sketches

The On-Off sketch [17] is specially designed for finding persistent flows. It does some modifications on the basis of the CM sketch. The data structure and the query process of the On-Off sketch remain the same, except that each counter contains a state field with two states: On and Off. All state fields are periodically set to On. When inserting a packet, the On-Off sketch checks all d mapped counters. If a counters' state is On, it will be incremented by 1, and its state will be set to Off. Otherwise, nothing will be done.

Applying Mini-Pyramid: Given an On-Off sketch, we replace all counters in the On-Off sketch by the M-Pyramid counters. We also increase the number of the state fields making them correspond to the low parts in the M-Pyramid counters.

7 Experimental Results

We conduct extensive experiments on both the S-Pyramid framework and the Mini-Pyramid framework, focusing on the following 4 key issues.

- **How do techniques and parameters affect S-Pyramid and Mini-Pyramid?** We take SP_{CM} and SP_{CU} as examples, apply different techniques, and compare the error. We take MP_{CM} as an example, adjust the parameter settings, and select the best parameters.
- **How accurate can both frameworks perform on sketches using counter arrays?** We implement both S-Pyramid and Mini-Pyramid on sketches of CM [28], CU [14], Count [29], and Augmented [23] using C++ program, and evaluate the accuracy of S-Pyramid on single-core CPU.
- **How about the throughput of both framework?** We evaluate the throughput of S-Pyramid and Mini-Pyramid on the above 4 sketches on single-core CPU platforms. We also take CM as an example to evaluate the throughput of both frameworks on multi-core CPU platforms.
- **How about the flexibility of Mini-Pyramid? What is the performance when applying it to sophisticated sketches?** We implement Mini-Pyramid on SBF [30] and the On-Off sketch [17], and compare the accuracy of the sketches with and without applying Mini-Pyramid.

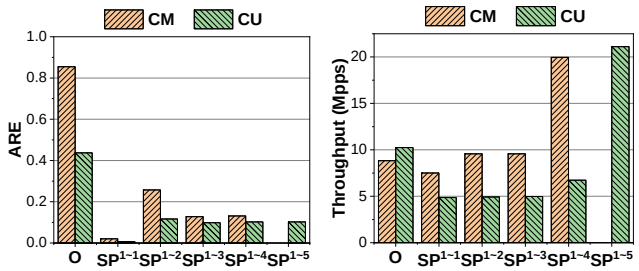
7.1 Experimental Setup

Traces: We use 3 real traces and a series of synthetic traces shown as follows:

- *CAIDA Trace:* As many prior works [15], [34] do, we use the anonymized trace from CAIDA [27]. We use a trace with monitoring time of 1 minute, which contains 254k flows, 27.1M packets. The maximum flow size is 917k. 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 5 minutes, which contains 5.45M flows, 133M packets. The maximum flow size is 1.22M.
- *IMC DC Trace:* IMC Data Center Trace [50] is collected from the data centers studied in [51]. 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 1.40M flows, 10.0M packets. The maximum flow size is 2.43M.
- *Campus Trace:* We also use a trace collected from the campus network for the experiment. The trace contains 384k flows, 2.50M packets. The maximum flow size is 9.98k.
- *Synthetic Traces:* We generate a series of synthetic traces that follow the Zipf [52] distribution using Web Polygraph [53]. The skewness of the traces range from 0.0 to 1.5. Each trace contains approximately 1.00M flows, 32.0M packets. The maximum flow sizes range from 86 to 4.35M.

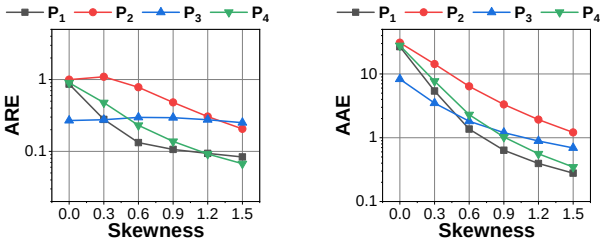
Computation platform: We conduct the experiments on a server with a 18-core CPU (Intel i9-10980XE) and 128GB total DRAM memory. Each core has three levels of cache memory: 64KB L1 cache, 1MB L2 cache, and 24.75MB L3 cache shared by all cores. Each word consists of 64 bits.

Implementation: We implement both the S-Pyramid and Mini-Pyramid frameworks by C++, and apply them to sketches of CM [28], CU [14], Count [29], and Augmented



(a) ARE. (b) Throughput.

Figure 8. Comparison of different techniques/optimization methods on S-Pyramid.



(a) ARE. (b) AAE.

Figure 9. Comparison of different parameter settings on Mini-Pyramid.

[23]. We also apply the Mini-Pyramid framework to SBF [30] and the On-Off sketch [17] to verify its flexibility. For S-Pyramid sketches, we apply the proposed techniques of both counter-pair sharing and word acceleration. For SP_{CU}, we apply Ostrich policy in addition.

For all sketches, we set the number of hash functions d to 4. For original sketches and Mini-Pyramid sketches, we use 32-bit Bob hash [54]; for S-Pyramid sketches, we use 64-bit Bob hash due to the usage of the hash bits by one hash technique. For ASketches, we set the size of the filter to 32. For SBF, we use Recurring Minimum optimization, and set the size of the secondary SBF to 1/2 of the primary SBF. For On-Off sketches, we set the parameters as the original paper [17] recommends.

We verify that randomness has little effect on the performance of Pyramid sketches by repeating experiments, and report the results of a single run as the final experimental results.

Evaluation metrics:

- *Average Relative Error (ARE):* $\frac{1}{n}\sum_{i=1}^{n}\frac{|f_i-\hat{f}_i|}{f_i}$, where n is the number of flows, f_i and $\hat{f}_i$ are the actual and estimated flow sizes respectively.
- *Average Absolute Error (AAE):* $\frac{1}{n}\sum_{i=1}^{n}|f_i-\hat{f}_i|$.
- *F1 Score:* $\frac{2\cdot PR\cdot RR}{PR+RR}$, where PR (Precision Rate) refers to the ratio of true positive instances to all detected instances, and RR (Recall Rate) refers to the ratio of true positive instances to all actual instances.
- *Throughput:* The number of packets operated per unit of time, in million packets per second (Mpps).
- *Average Number of Memory Accesses:* The average number of memory accesses in the sketch per operation [5]. Note that the accesses number of CM, CU, and Count is a constant, which always equal to the number of hash functions d.

7.2 Experiments on Techniques and Settings

In this subsection, we take SP_{CM} and SP_{CU} as examples to compare the performance of the S-Pyramid framework using different techniques. We also take MP_{CM} as an example to compare the performance of Mini-Pyramid framework using different parameter settings.

7.2.1 Experiments on S-Pyramid Techniques

We have proposed 5 techniques/optimization methods in S-Pyramid: counter-pair sharing (T_1), word constraint (T_2), word sharing (T_3), one hashing (T_4), and Ostrich policy (T_5). We use $SP_{CM}^{1\sim i}$ and $SP_{CU}^{1\sim i}$ ($1 \leqslant i \leqslant 5$) to denote SP_{CM} and SP_{CU} with the first i techniques, respectively. We use CAIDA trace for the experiments. In the experiments, we maintain the same memory usage and vary the techniques applied to the sketches. As shown in Figure 8, we have the following 5 insights.

1) Counter-pair sharing (T_1) significantly improves the accuracy, while slightly degrading the throughput of CM, and significantly degrades the throughput of CU.
2) Word constraint (T_2) slightly improves the throughput, while incurring severe accuracy loss.
3) Word sharing (T_3) overcomes the main shortcoming of word constraint, improving the accuracy without negative impact on throughput.
4) One hashing (T_4) significantly improves the throughput of CM, slightly improves the throughput of CU, while not affecting accuracy.
5) Ostrich policy (T_5) significantly improves the throughput of CU, and has little negative effect on accuracy.

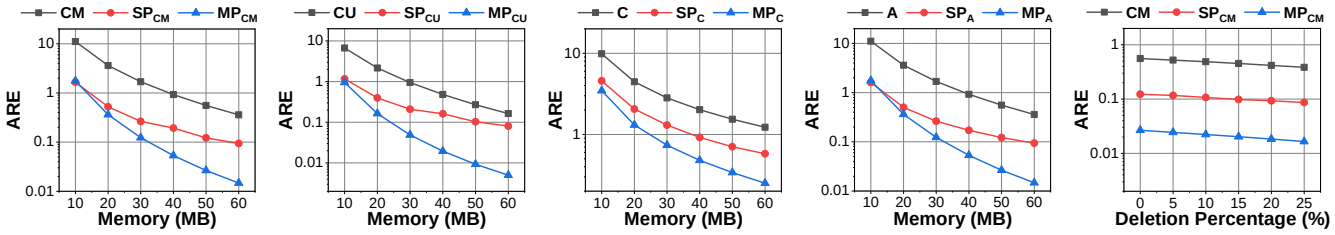
Summary and Analysis: As shown above, counter-pair sharing significantly improves the accuracy, word acceleration and Ostrich policy significantly improve the throughput while sacrificing a little accuracy. It is worth noticing that $T_2 \sim T_4$ do not significantly accelerate SP_{CU}. That's mainly because SP_{CU} needs to get all reported values of the d mapped counters during insertion. This process requires frequent memory accesses at higher layers. Although T_2 guarantees that SP_{CU} only performs 1 memory access per layer, it still performs a large amount of memory accesses, resulting in low throughput. Fortunately, T_5 reduces the memory accesses performed during reporting process. Therefore, $SP_{CU}^{1\sim 5}$ achieves similar throughput as SP_{CM}.

7.2.2 Experiments on Mini-Pyramid Settings

When designing Mini-Pyramid, we mainly consider the following two indicators: 1) accuracy, and 2) the maximum value of the Mini-Pyramid counter. In order to achieve the two goals, we compare the performance of the following 4 parameter settings:

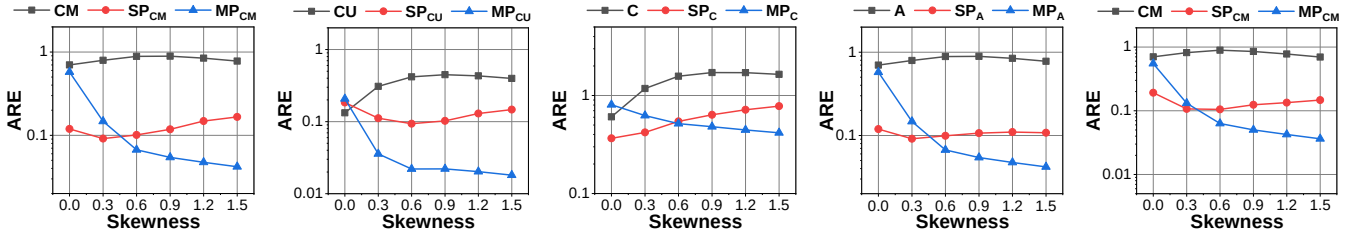
- P_1 (ours): 2 layers, low part consists of 3 4-bit counters, high part consists of 1 16-bit counter.
- P_2: 2 layers, low part consists of 4 2-bit counters, high part consists of 1 19-bit counter.
- P_3: 2 layers, low part consists of 2 8-bit counters, high part consists of 1 13-bit counter.
- P_4: 3 layers, 1^{st} layer consists of 4 2-bit counters, 2^{nd} layer consists of 2 2-bit counters, 3^{rd} layer consists of 1 13-bit counter.

5. Memory accesses to the filter in ASketches is not counted.



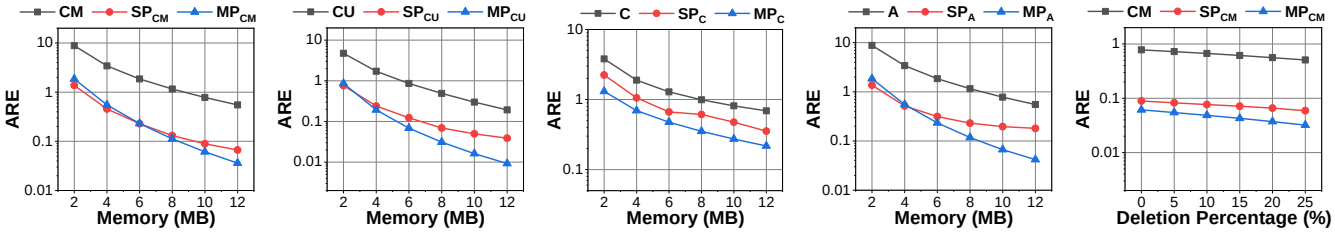
(a) Insertion on CM. (b) Insertion on CU. (c) Insertion on C. (d) Insertion on A. (e) Deletion on CM.

Figure 10. ARE on CAIDA trace.



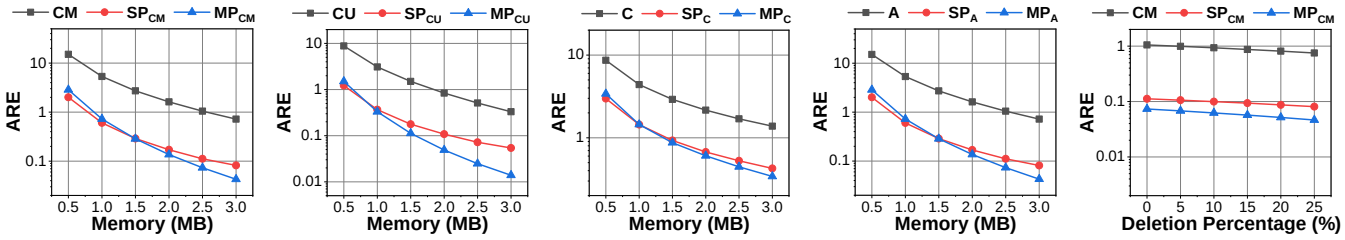
(a) Insertion on CM. (b) Insertion on CU. (c) Insertion on C. (d) Insertion on A. (e) Deletion on CM.

Figure 11. ARE on synthetic traces.



(a) Insertion on CM. (b) Insertion on CU. (c) Insertion on C. (d) Insertion on A. (e) Deletion on CM.

Figure 12. ARE on IMC DC trace.



(a) Insertion on CM. (b) Insertion on CU. (c) Insertion on C. (d) Insertion on A. (e) Deletion on CM.

Figure 13. ARE on campus trace.

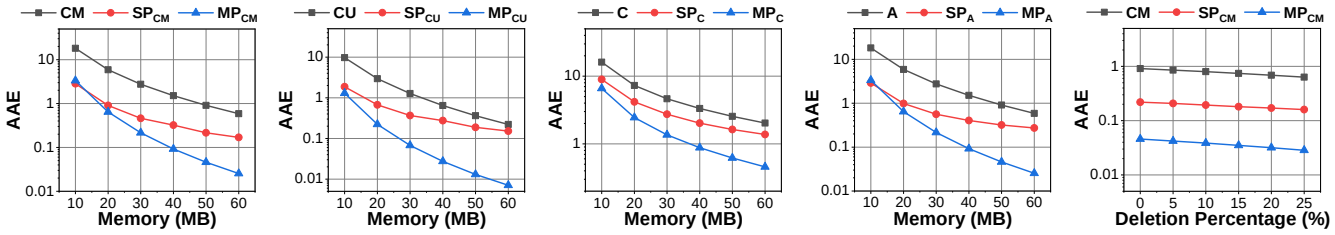
In order to test the performance of Mini-Pyramid on different flow size distributions, we use synthetic traces for the experiments. According to the results in Figure 9, we have the following 2 insights.

1) Our parameter settings (P_1) achieve lower ARE than other parameter settings on most distributions. Although P_3 and P_4 have lower ARE on some traces, P_1 can always achieve similar ARE.
2) Our parameter settings (P_1) achieve lower AAE than other parameter settings on most distributions. Note that P_1 performs better in AAE than in ARE. When skewness is 1.5, P_1 has higher ARE but achieves lower AAE than P_4.

Summary and Analysis: As shown above, our parameter settings (P_1) outperform other parameter settings on most flow size distributions. Therefore, we recommend using P_1 in Mini-Pyramid. It is worth noticing that the low part of P_1 has the same counter size as S-Pyramid and many prior works (*e.g.*, HeavyGuardian [36], Cold filter [37]) – 4 bits, which is a typical threshold between mice flow and elephant flow.

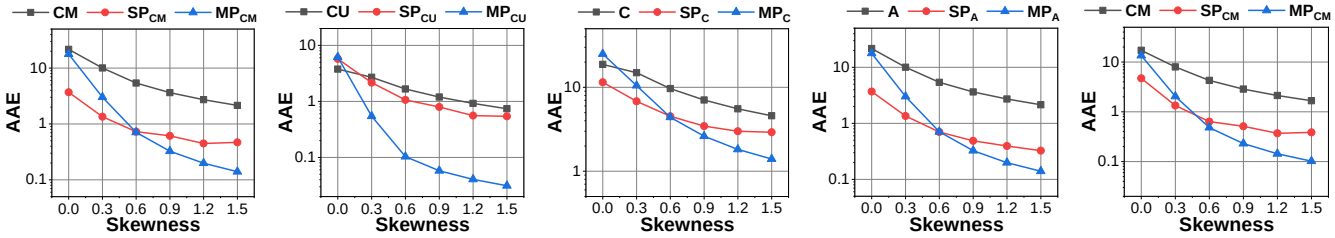
7.3 Experiments on Accuracy

In this subsection, we evaluate the accuracy of both S-Pyramid and Mini-Pyramid. We implement 4 typical sketches and use all 4 traces for the experiment. In our experiments on synthetic traces, we maintain the same memory usage and deletion percentage, and vary the skewness by using different traces. In other experiments, we maintain the same amount of packets, and vary the memory usage of sketches.



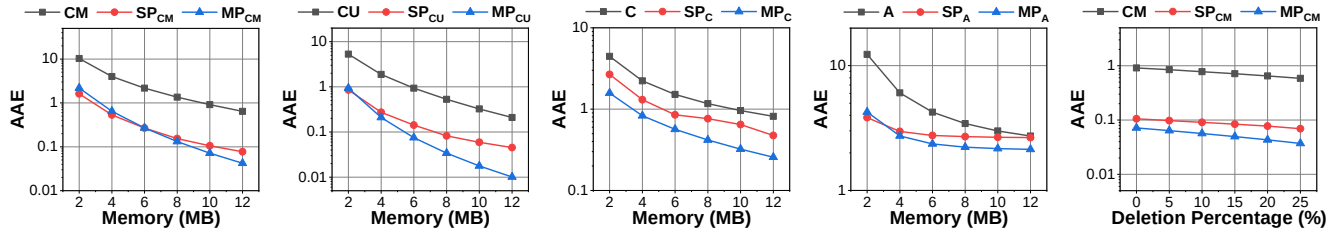
(a) Insertion on CM. (b) Insertion on CU. (c) Insertion on C. (d) Insertion on A. (e) Deletion on CM.

Figure 14. AAE on CAIDA trace.



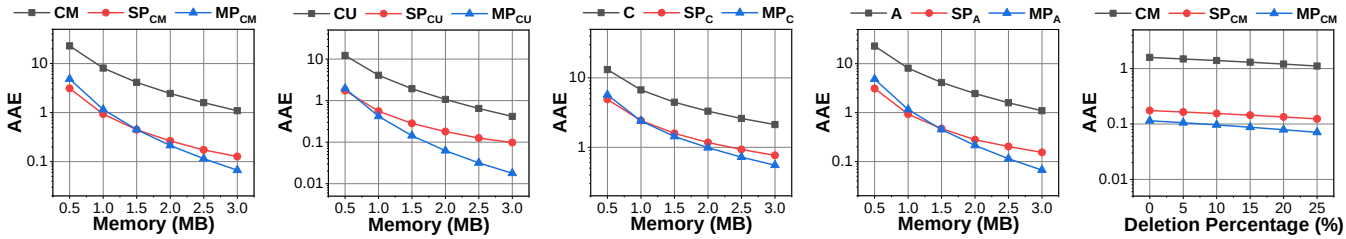
(a) Insertion on CM. (b) Insertion on CU. (c) Insertion on C. (d) Insertion on A. (e) Deletion on CM.

Figure 15. AAE on synthetic traces.



(a) Insertion on CM. (b) Insertion on CU. (c) Insertion on C. (d) Insertion on A. (e) Deletion on CM.

Figure 16. AAE on IMC DC trace.



(a) Insertion on CM. (b) Insertion on CU. (c) Insertion on C. (d) Insertion on A. (e) Deletion on CM.

Figure 17. AAE on campus trace.

ARE (Figure 10-13): The experimental results show that, compared with the original sketches, S-Pyramid reduces the ARE by 4.32 times on average, and Mini-Pyramid reduces the ARE by 14.0 times on average.

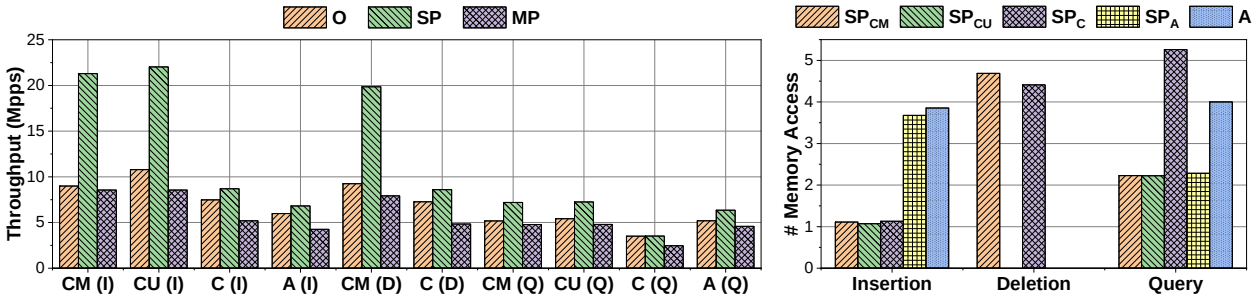
As shown in Figure 10, on CAIDA trace, when using 50MB of memory, the ARE of SP_{CM}, SP_{CU}, SP_C, and SP_A reach 4.57, 2.63, 2.18, and 4.57 times lower than the original sketches, respectively; the ARE of MP_{CM}, MP_{CU}, MP_C, and MP_A reach 21.0, 29.2, 4.54, and 21.0 times lower, respectively. During the process of deletion, the ARE of SP_{CM} and MP_{CM} reach 4.52 and 22.1 times lower than the original sketches on average, respectively.

As shown in Figure 11, on synthetic traces, the ARE of Mini-Pyramid decreases with the increasing of skewness. When skewness is 0.3, the ARE of MP_{CM}, MP_{CU}, MP_C, and MP_A are 0.147, 0.036, 0.625, and 0.147, respectively; when skewness is 1.5, the ARE of MP_{CM}, MP_{CU}, MP_C, and MP_A reach 0.042, 0.018, 0.416, and 0.042, respectively.

As shown in Figure 12-13, on IMC DC trace, the ARE of S-Pyramid and Mini-Pyramid reach 5.34 and 8.76 times lower than the original sketches on average, respectively. On campus trace, the ARE of S-Pyramid and Mini-Pyramid reach 7.12 and 10.0 times lower on average, respectively.

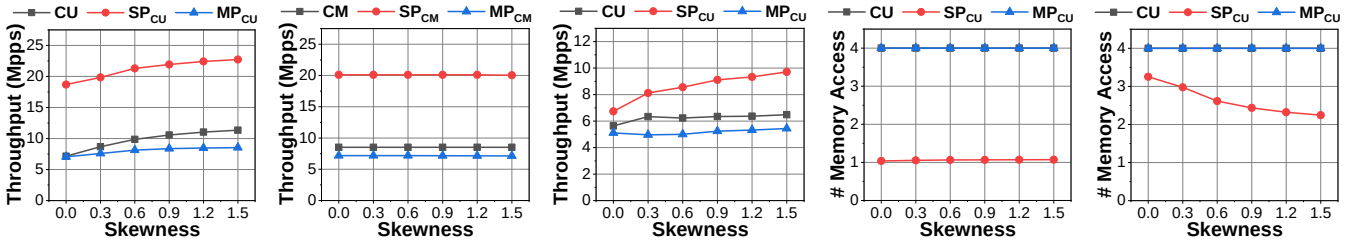
AAE (Figure 14-17): The experimental results show that, compared with the original sketches, S-Pyramid reduces the AAE by 3.58 times on average, and Mini-Pyramid reduces the AAE by 13.2 times on average.

As shown in Figure 14, on CAIDA trace, when using 50MB of memory, the AAE of SP_{CM}, SP_{CU}, SP_C, and SP_A reach 4.23, 1.93, 1.56, and 2.85 times lower than the original



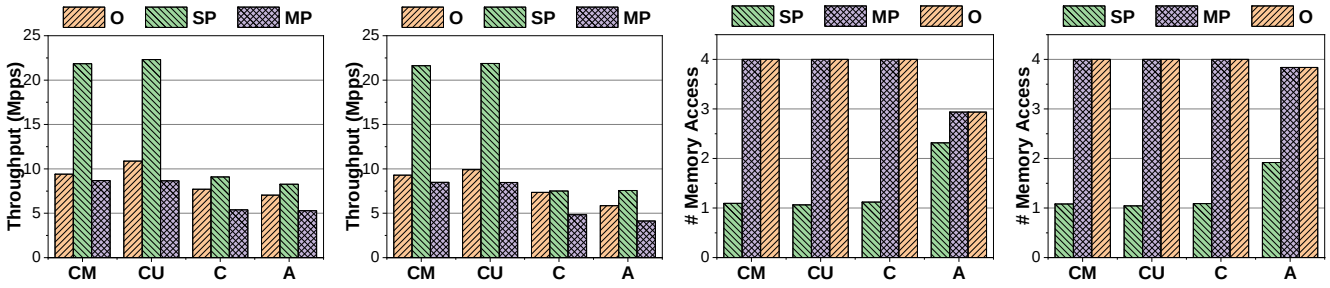
(a) Throughput. (b) # memory access.

Figure 18. Throughput and the average number of memory accesses on CAIDA trace, where **O**, **P**, and **MP** represent the original sketches, the S-Pyramid sketches, and the Mini-Pyramid sketches, respectively; **I**, **D**, and **Q** represent insertion, deletion, and query, respectively.



(a) Insertion T on CU. (b) Deletion T on CM. (c) Query T on CU. (d) Insertion MA on CU. (e) Query MA on CU.

Figure 19. Throughput and the average number of memory accesses on synthetic traces, where T and MA represent throughput and the average number of memory accesses for short, respectively.



(a) T on IMC DC trace. (b) T on campus trace. (c) MA on IMC DC trace. (d) MA on campus trace.

Figure 20. Throughput and the average number of memory accesses on insertion on other traces.

sketches, respectively; the AAE of MP_{CM}, MP_{CU}, MP_C, and MP_A reach 19.8, 27.8, 4.12, and 19.8 times lower, respectively. During the process of deletion, the AAE of SP_{CM} and MP_{CM} reach 4.08 and 21.0 times lower than the original sketches on average, respectively.

As shown in Figure 15, on synthetic traces, the AAE of Mini-Pyramid decreases with the increasing of skewness. When skewness is 0.3, the AAE of MP_{CM}, MP_{CU}, MP_C, and MP_A are 3.01, 0.54, 10.49, and 3.01, respectively; when skewness is 1.5, the AAE of MP_{CM}, MP_{CU}, MP_C, and MP_A reach 0.14, 0.03, 1.40, and 0.14, respectively.

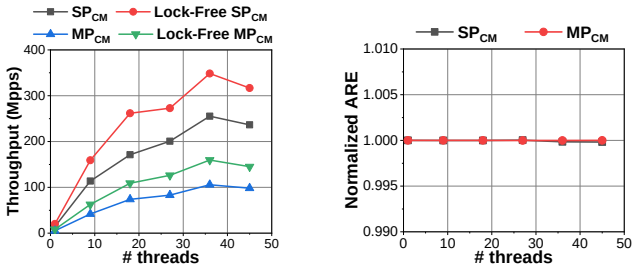
As shown in Figure 16-17, on IMC DC trace, the AAE of S-Pyramid and Mini-Pyramid reach 4.34 and 6.98 times lower than the original sketches on average, respectively. On campus trace, the AAE of S-Pyramid and Mini-Pyramid reach 6.42 and 9.78 times lower on average, respectively.

Summary and Analysis: As shown above, both S-Pyramid and Mini-Pyramid achieve higher accuracy than the original sketches. That's mainly because both frameworks provide hierarchical structures for sketches. Therefore, they can dynamically assign appropriate number of bit of each flows, which reduce the memory waste. Note that when skewness is 0, the performance of Mini-Pyramid is not as good. That's mainly because Mini-Pyramid is designed based on the skewness of the network traffic, however, skewness $=$ 0 means the traffic is normalized distributed. Therefore, most of the low-parts overflow, and Mini-Pyramid cannot separate the elephant flows and the mice flows. It is worth noticing that S-Pyramid still performs well in this case. This is because S-Pyramid has more granularity, so that it can roughly allocate the bit numbers for each flow even if the trace is not skewed.

7.4 Experiments on Throughput

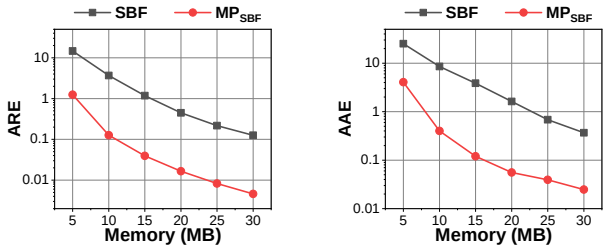
In this subsection, we evaluate the throughput of both S-Pyramid and Mini-Pyramid. For experiments on single-core, we implement 4 typical sketches and use all 4 traces. For experiments on multi-core, we take CM as an example and use CAIDA trace for the experiment.

Normally, multi-thread programs use locks to avoid conflict. However, locks reduce the efficiency of multi-core parallelism, which in turn reduce the throughput. Fortu-



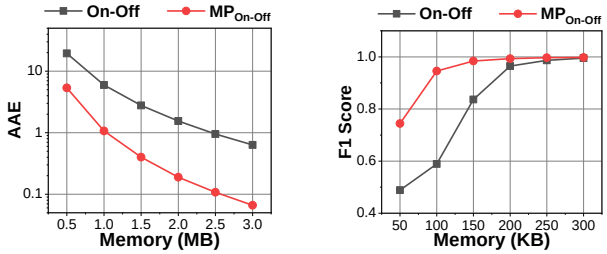
(a) Throughput. (b) Normalized ARE.

Figure 21. Insertion throughput on multi-core CPU. Normalized ARE represents the ratio of ARE using multi-core to that using single-core.



(a) ARE. (b) AAE.

Figure 22. Application on the Spectral Bloom filter.



(a) Persistence Estimation. (b) Finding Persistent Items.

Figure 23. Application on the On-Off sketch.

nately, for sketches, lock-free is possible. That's mainly because sketches are probabilistic data structures with error, therefore, error cause by read/write is acceptable. Besides, the hash conflicts of sketches are controllable. Therefore, there will be few read/write conflicts, causing little error. We will verify this point in the following experiments.

Single-Core Throughput (Figure 18(a), 19(a)-(c), 20(a)-(b)): The experimental results show that, compared with the original sketches, S-Pyramid improves the throughput by 1.50 times on average, and Mini-Pyramid lower the throughput by 21.3% on average.

As shown in Figure 18, on CAIDA trace, the insertion throughput of SP_{CM}, SP_{CU}, SP_C, and SP_A reach 2.37, 2.04, 1.16, and 1.13 times higher than the original sketches, respectively; that of MP_{CM}, MP_{CU}, MP_C, and MP_A are 95.1%, 79.3%, 69.2%, and 71.1% of the original sketches, respectively. The deletion throughput of SP_{CM} and SP_C reach 2.15 and 1.18 times higher than the original sketches, respectively; that of MP_{CM} and MP_C are 85.6% and 66.6% of the original sketches, respectively. The query throughput of SP_{CM}, SP_{CU}, SP_C, and SP_A reach 1.38, 1.33, 1.01, and 1.22 times higher than the original sketches, respectively; that of MP_{CM}, MP_{CU}, MP_C, and MP_A are 92.3%, 88.5%, 70.3%, and 88.4% of the original sketches, respectively.

As shown in Figure 19(a)-(c), on synthetic traces, the insertion and query throughput of S-Pyramid increase with the increasing of skewness. When skewness is 0.3, the insertion and query throughput of SP_{CU} are 19.8Mpps and 8.12Mpps, respectively; when skewness is 1.5, the insertion and query throughput of SP_{CU} reach 22.7Mpps and 9.71Mpps, respectively.

As shown in Figure 20(a)-20(b), the insertion throughput of S-Pyramid on IMC DC and campus trace reach 1.68 and 1.71 times higher than the original sketches on average, respectively; that of Mini-Pyramid on IMC DC and campus trace are 20.7% and 21.7% lower on average, respectively.

The average number of memory accesses (Figure 18(b), 19(d)-(e), 20(c)-(d)): The experimental results show that, compared with the original sketches, S-Pyramid reduces the average number of memory accesses to 1.75 on average.

As shown in Figure 18(b), on CAIDA trace, for each insertion, the average number of memory accesses of SP_{CM}, SP_{CU}, SP_C, and SP_A reach 1.11, 1.07, 1.13, and 3.67, respectively, while that of CM, CU, C, and A are 4, 4, 4, and 3.85, respectively. For each query, the average number of memory accesses of SP_{CM}, SP_{CU}, and SP_A reach 2.23, 2.22, and 2.28, respectively, while that of CM, CU, and A are 4, 4, and 4.00, respectively.

As shown in Figure 19(d)-(e), on synthetic traces, for each query, the average number of memory accesses of S-Pyramid decreases with the increasing of skewness. When skewness is 0.3, the average number of memory accesses of SP_{CU} is 2.98; when skewness is 1.5, the average number of memory accesses of SP_{CU} reaches 2.24.

As shown in Figure 20(c)-(d), for each insertion, the average number of memory accesses of S-Pyramid on IMC DC trace and campus trace reach 1.40 and 1.28 on average, respectively.

Multi-Core Throughput (Figure 21): The experimental results show that, the lock-free Pyramid sketches significantly increase the throughput, while having little negative effect on accuracy.

As shown in Figure 21(a), when using 36 threads, the throughput of lock-free SP_{CM} is 1.36 times higher than that using locks, and the throughput of MP_{CM} is 1.51 times higher. Meanwhile, for both SP_{CM} and MP_{CM}, the difference between the ARE of the lock-free version and the lock version is less than 0.1%. Therefore, the experiment result further verifies that read/write conflicts caused by multi-thread version have little negative effect on accuracy.

Summary and Analysis: As shown above, on single-core CPU platforms, S-Pyramid has a better throughput than the original sketches. It can significantly reduce the number of memory accesses and hash calculation. It is worth noticing that the average number of memory accesses during insertion is smaller than that during query, and the throughput is also higher. This is because during insertion, the parent counter need to be accessed only if an overflow occurs, and overflows occur infrequently. Therefore S-Pyramid can significantly improve the insertion throughput. Besides, both lock-free S-Pyramid and lock-free Mini-Pyramid bring little error. Therefore, Pyramid can also achieve high throughput on multi-core CPU platforms.

7.5 Experiments on Flexibility

In this subsection, we evaluate the flexibility of Mini-Pyramid. We implement the On-Off sketch [17] and SBF [30], and apply Mini-Pyramid to them, in order to verify that Mini-Pyramid can be applied to sophisticated sketches and significantly improve the accuracy. We use CAIDA trace for the experiments.

SBF (Figure 22): The experimental results show that, applying the Mini-Pyramid framework can improve the accuracy of the SBF by 22.7 times on average. As shown in Figure 22, when using 25MB of memory, the ARE of MP_{SBF} is 26.3 times lower than SBF; the AAE of MP_{SBF} is 17.4 times lower.

On-Off Sketches (Figure 23): The experimental results show that, applying the Mini-Pyramid framework can improve the accuracy of the On-Off sketch. As shown in Figure 23(a), on persistence estimation task, when using 2.5MB of memory, the AAE of $MP_{On\text{-}Off}$ is 8.87 times lower than On-Off. As shown in Figure 23(b), on finding persistent items task, when using 100KB of memory, the F1 score of $MP_{On\text{-}Off}$ is 0.946, while that of On-Off is 0.589.

Summary and Analysis: As shown above, by simply replacing counters in the original sketches with M-Pyramid counters, both SBF and On-Off sketches achieve higher accuracy, which shows the flexibility of Mini-Pyramid. Note that the On-Off sketch is the state-of-arts in finding persistent flows, therefore, $MP_{On\text{-}Off}$ can be considered as a more accurate work in this task.

8 Conclusion

Sketches are the bases of many Internet applications. However, most sketches do not work well for skewed network traffic. To address the above problem, in this paper, we propose a sketch framework family – the Pyramid family. The first member, the S-Pyramid framework, can significantly improve both accuracy and speed. The second member, the Mini-Pyramid framework, is more flexible in application, but with lower speed. We apply both frameworks to sketches of CM [28], CU [14], Count [29], and Augmented [23], and further apply the Mini-Pyramid framework to SBF [30] and the On-Off sketch [17]. The experimental results show that both frameworks significantly improve the accuracy, and the S-Pyramid framework also significantly improves speed. All related source codes are released at Github [31].

Acknowledgment

We would like to thank our editors Matthew Caesar, Ness Shroff, Carrie Stein, and the anonymous reviewers for their thoughtful feedback. We would like to thank Yuhan Wu and Zheng Zhong for their helpful suggestions and discussions. This work is supported by Key-Area Research and Development Program of Guangdong Province 2020B0101390001, National Natural Science Foundation of China (NSFC) (No. U20A20179), the project of "FANet: PCL Future Greater-Bay Area Network Facilities for Large-scale Experiments and Applications" (No. LZC0019), and Computing and Network Innovation Lab, Huawei Cloud.

References

[1] Tong Yang, Yang Zhou, Hao Jin, Shigang Chen, and Xiaoming Li. Pyramid sketch: A sketch framework for frequency estimation of data streams. *Proceedings of the VLDB Endowment*, 10(11):1442–1453, 2017.

[2] Nick Duffield, Carsten Lund, and Mikkel Thorup. Learn more, sample less: control of volume and variance in network measurement. *IEEE Transactions on Information Theory*, 51(5):1756–1775, 2005.

[3] Jiaqi Zheng, Hong Xu, Guihai Chen, and Haipeng Dai. Minimizing transient congestion during network update in data centers. In *Network Protocols (ICNP), 2015 IEEE 23rd International Conference on*, pages 1–10. IEEE, 2015.

[4] David Plonka. Flowscan: A network traffic flow reporting and visualization tool. In *LISA*, pages 305–317, 2000.

[5] Hongli Xu, Zhuolong Yu, Chen Qian, Xiang-Yang Li, Zichun Liu, and Liusheng Huang. Minimizing flow statistics collection cost using wildcard-based requests in sdns. *IEEE/ACM Transactions on Networking*, 25(6):3587–3601, 2017.

[6] Xenofontas Dimitropoulos, Paul Hurley, and Andreas Kind. Probabilistic lossy counting: an efficient algorithm for finding heavy hitters. *ACM SIGCOMM CCR*, 38(1):5–5, 2008.

[7] Ran Ben Basat, Gil Einziger, Roy Friedman, Marcelo C Luizelli, and Erez Waisbard. Constant time updates in hierarchical heavy hitters. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 127–140, 2017.

[8] Balachander Krishnamurthy, Subhabrata Sen, Yin Zhang, and Yan Chen. Sketch-based change detection: methods, evaluation, and applications. In *ACM IMC*, pages 234–247. ACM, 2003.

[9] Robert Schweller, Ashish Gupta, Elliot Parsons, and Yan Chen. Reversible sketches for efficient and accurate change detection over network data streams. In *ACM IMC*, pages 207–212. ACM, 2004.

[10] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Flowradar: A better netflow for data centers. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*, pages 311–324, 2016.

[11] Yikai Zhao, Kaicheng Yang, Zirui Liu, Tong Yang, Li Chen, Shiyi Liu, Naiqian Zheng, Ruixin Wang, Hanbo Wu, Yi Wang, et al. Lightguardian: A full-visibility, lightweight, in-band telemetry system using sketchlets. In *NSDI*, pages 991–1010, 2021.

[12] Rafal Maison and Maciej Zakrzewicz. Prediction-based load shedding for burst data streams. *Bell Labs Technical Journal*, 16(1):121–132, 2011.

[13] Zheng Zhong, Shen Yan, Zikun Li, Decheng Tan, Tong Yang, and Bin Cui. Burstsketch: Finding bursts in data streams. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2375–2383, 2021.

[14] Cristian Estan and George Varghese. New directions in traffic measurement and accounting. *ACM SIGMCOMM CCR*, 32(4), 2002.

[15] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Adaptive measurements using one elastic sketch. *IEEE/ACM Transactions on Networking*, 27(6):2236–2251, 2019.

[16] Xiaoqi Chen, Shir Landau-Feibish, Mark Braverman, and Jennifer Rexford. Beaucoup: Answering many network traffic queries, one memory update at a time. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 226–239, 2020.

[17] Yinda Zhang, Jinyang Li, Yutian Lei, Tong Yang, Zhetao Li, Gong Zhang, and Bin Cui. On-off sketch: a fast and accurate sketch on persistence. *Proceedings of the VLDB Endowment*, 14(2):128–140, 2020.

[18] Qun Huang, Siyuan Sheng, Xiang Chen, Yungang Bao, Rui Zhang, Yanwei Xu, and Gong Zhang. Toward nearly-zero-error sketching via compressive sensing. In *18th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 21)*, pages 1027–1044, 2021.

[19] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[20] Wenjun Jiang, Jie Wu, Feng Li, Guojun Wang, and Huanyang Zheng. Trust evaluation in online social networks using generalized network flow. *IEEE Transactions on Computers*, 65(3):952–963, 2016.

[21] Giuseppe Ascia, Vincenzo Catania, Maurizio Palesi, and Davide Patti. Implementation and analysis of a new selection strategy for adaptive routing in networks-on-chip. *IEEE Transactions on Computers*, 57(6):809–820, 2008.

[22] Assaf Shacham, Keren Bergman, and Luca P Carloni. Photonic networks-on-chip for future generations of chip multiprocessors. *IEEE Transactions on Computers*, 57(9):1246–1260, 2008.

[23] Pratanu Roy, Arijit Khan, and Gustavo Alonso. Augmented sketch: Faster and more accurate stream processing. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1449–1463, 2016.

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

[25] Matthew Roughan, Yin Zhang, Walter Willinger, and Lili Qiu. Spatio-temporal compressive sensing and internet traffic matrices (extended version). *IEEE/ACM Transactions on Networking*, 20(3):662–676, 2011.

[26] Theophilus Benson, Aditya Akella, and David A Maltz. Unraveling the complexity of network management. In *NSDI*, 2009.

[27] The CAIDA Anonymized Internet Traces. http://www.caida.org/data/overview/.

[28] Graham Cormode and S Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[29] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. In *Automata, Languages and Programming*. Springer, 2002.

[30] Saar Cohen and Yossi Matias. Spectral bloom filters. In *ACM SIGMOD*, pages 241–252, 2003.

[31] Source code of the Pyramid family [online]. https://github.com/Pyramid-Family/Pyramid-Family.

[32] Zaoxing Liu, Ran Ben-Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. Nitrosketch: Robust and general sketch-based monitoring in software switches. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 334–350. 2019.

[33] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 101–114, 2016.

[34] Qun Huang, Xin Jin, Patrick PC Lee, Runhui Li, Lu Tang, Yi-Chao Chen, and Gong Zhang. Sketchvisor: Robust network measurement for software packet processing. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 113–126, 2017.

[35] Tong Yang, Haowei Zhang, Jinyang Li, Junzhi Gong, Steve Uhlig, Shigang Chen, and Xiaoming Li. Heavykeeper: An accurate algorithm for finding top-k elephant flows. *IEEE/ACM Transactions on Networking*, 27(5):1845–1858, 2019.

[36] Tong Yang, Junzhi Gong, Haowei Zhang, Lei Zou, Lei Shi, and Xiaoming Li. Heavyguardian: Separate and guard hot items in data streams. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2584–2593, 2018.

[37] Yang Zhou, Tong Yang, Jie Jiang, Bin Cui, Minlan Yu, Xiaoming Li, and Steve Uhlig. Cold filter: A meta-framework for faster and more accurate stream processing. In *Proceedings of the 2018 International Conference on Management of Data*, pages 741–756, 2018.

[38] Tong Yang, Siang Gao, Zhouyi Sun, Yufei Wang, Yulong Shen, and Xiaoming Li. Diamond sketch: Accurate per-flow measurement for big streaming data. *IEEE Transactions on Parallel and Distributed Systems*, 30(12):2650–2662, 2019.

[39] Yinda Zhang, Zaoxing Liu, Ruixin Wang, Tong Yang, Jizhou Li, Ruijie Miao, Peng Liu, Ruwen Zhang, and Junchen Jiang. Cocosketch: high-performance sketch-based measurement over arbitrary partial key query. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 207–222, 2021.

[40] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[41] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM transactions on networking*, 8(3):281–293, 2000.

[42] Josep Aguilar-Saborit, Pedro Trancoso, Victor Muntes-Mulero, and Josep-Lluis Larriba-Pey. Dynamic count filters. *ACM SIGMOD Record*, pages 26–32, 2006.

[43] Tong Yang, Alex X Liu, Muhammad Shahzad, Dongsheng Yang, Qiaobin Fu, Gaogang Xie, and Xiaoming Li. A shifting framework for set queries. *IEEE/ACM Transactions on Networking*, 25(5):3116–3131, 2017.

[44] Yi Lu, Andrea Montanari, Balaji Prabhakar, Sarang Dharmapurikar, and Abdul Kabbani. Counter braids: a novel counter architecture for per-flow measurement. In *ACM SIGMETRICS*, 2008.

[45] Tao Li, Shigang Chen, and Yibei Ling. Per-flow traffic measurement through randomized counter sharing. *IEEE/ACM Transactions on Networking*, 20(5):1622–1634, 2012.

[46] Chengchen Hu, Bin Liu, Hongbo Zhao, Kai Chen, Yan Chen, Chunming Wu, and Yu Cheng. Disco: Memory efficient and accurate flow statistics for network measurement. In *Distributed Computing Systems (ICDCS), 2010 IEEE 30th International Conference on*, pages 665–674. IEEE, 2010.

[47] Haipeng Dai, Muhammad Shahzad, Alex X Liu, and Yuankun Zhong. Finding persistent items in data streams. *Proceedings of the VLDB Endowment*, 10(4):289–300, 2016.

[48] Xilai Liu, Yan Xu, Peng Liu, Tong Yang, Jiaqi Xu, Lun Wang, Gaogang Xie, Xiaoming Li, and Steve Uhlig. Sead counter: Self-adaptive counters with different counting ranges. *IEEE/ACM Transactions on Networking*, 2021.

[49] Cuda toolkit documentation. http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#coalesced-access-to-global-memory.

[50] Data Set for IMC 2010 Data Center Measurement. http://pages.cs.wisc.edu/~tbenson/IMC10_Data.html.

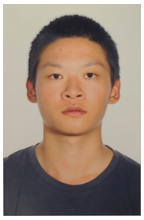
[51] Theophilus Benson, Aditya Akella, and David A Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 267–280, 2010.

[52] David MW Powers. Applications and explanations of Zipf's law. In *EMNLP-CoNLL*. Association for Computational Linguistics, 1998.

[53] Alex Rousskov and Duane Wessels. High-performance benchmarking with web polygraph. *Software: Practice and Experience*, 34(2):187–211, 2004.

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

Yuanpeng Li is currently pursuing the B.S. degree in the School of Electronics Engineering and Computer Science, Peking University, advised by Tong Yang. His research interests include network measurements, sketches, bloom filters, and hash tables.



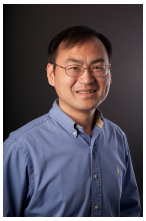
Xiang Yu is an undergraduate majoring in computer science in the school of EECS, Peking University. He is currently following Tong Yang doing researches and his research interests include network measurements, sketches, Bloom filters, and KV stores.

Yilong Yang received his B.S. degree in network engineering from Xidian University in 2019. He is currently a master's student in the School of Cyber Engineering, Xidian University. His research interests cover analysis of protocols, deep learning neural network, and network measurement.

Yang Zhou graduated (summa cum laude) from Peking University, advised by Tong Yang. He is currently pursuing the Ph.D. degree with Harvard University, advised by Minlan Yu. He is broadly interested in streaming algorithms, networked systems, and data-intensive systems.

Tong Yang received his Ph.D. degree in Computer Science from Tsinghua University in 2013. He visited Institute of Computing Technology, Chinese Academy of Sciences (CAS) China from 2013.7 to 2014.7. Now he is an associate professor in Computer Science Department, Peking University. His research interests include network big data, sketches, network measurement, Bloom filters, IP lookups, KV stores, hash tables, *etc.* He published papers in SIGCOMM, SIGKDD, SIGMOD, SIGCOMM CCR, VLDB, ATC, ToN, TC, ICDE, INFOCOM, *etc.*

Zhuo Ma received his Ph.D. degree in Computer Architecture from Xidian University, Xi'an, China, in 2010. Now, he is a professor at school of Cyber Engineering, Xidian University. His research interests include cryptography, machine learning in cyber security, and Internet of things security.



Shigang Chen received his M.S. and Ph.D. degrees in Computer Science from University of Illinois at Urbana-Champaign in 1996 and 1999, respectively. Prior to that, he received his B.S. degree in Computer Science from University of Science and Technology of China in 1993. After graduating from UIUC, he worked with Cisco Systems on network security for three years and helped starting a network security company, Protego Networks. He joined University of Florida as an assistant professor in 2002, and was promoted to associate professor in 2008 and to professor in 2013. He was a recipient of IEEE Communications Society Best Tutorial Paper Award in 1999, NSF CAREER Award in 2007, and Cisco University Research Award in 2007, 2012. He published 200+ peer-reviewed journal/conference papers and has 13 US patents. He holds University of Florida Research Foundation Professorship in 2017-2020 and University of Florida Term Professorship in 2017-2020. He is an IEEE Fellow and an ACM Distinguished Member.