A Pyramid Framework for Sketches in Data Streams

Tong Yang, Yang Zhou, Haowei Zhang, Shigang Chen, *IEEE Fellow*, Hao Jin, Bin Cui, and Xiaoming Li

***Abstract*—Sketch is a probabilistic data structure, and is used to store and query the frequency of any item in a given multiset. Due to its high memory efficiency, it has been applied to various fields in computer science, such as stream database, network traffic measurement, *etc.* The key metrics of sketches for data streams are accuracy, speed, and memory usage. Various sketches have been proposed, but they cannot achieve both high accuracy and high speed using limited memory, especially for skewed datasets. To address this issue, we propose a sketch framework, the Pyramid sketch, which can significantly improve accuracy as well as update and query speed. To verify the effectiveness and efficiency of our framework, we applied our framework to four typical sketches. Extensive experimental results show that the accuracy is improved up to 3.50 times, while the speed is improved up to 2.10 times. We have released our source codes at Github [2].**

I. Introduction

A. Background and Motivation

Given a multiset, estimating the frequency of each item is a critical problem in data stream applications. A `multiset` refers to a set in which each item can appear multiple times. In scenarios such as real-time IP traffic, graph streams, web clicks and crawls, sensor database, and natural language processing (NLP) [3]–[7], the massive data are often organized as high-speed streams, requiring servers to record stream information in real time. Due to the high speed of data streams, it is often impractical to achieve accurate recording and estimating of item frequencies. Therefore, estimation of item frequencies by probabilistic data structures becomes popular and gains wide acceptances [8]–[10]. Sketches are initially designed for the estimation of item frequencies in data streams [11]–[14], and now have been applied to many more fields, such as sparse approximation in compressed sensing [15], natural language processing [16], [17], data graph [18]–[20], and more [21]–[23]. Note that we mainly *focus on the sketches used for frequency estimation in this paper.*

According to our analysis of real datasets and literatures [8], [10], the item frequencies in data streams are often highly skewed. In other words, most items are cold (*i.e.*, have a low frequency), while a few items are hot (*i.e.*, have a high frequency). For convenience, we use `hot items` and `cold items` to represent them in this paper. All existing sketches use counters to store frequencies, but it is difficult to find a proper size for the counters to fit these highly skewed data streams. For example, the frequencies of most items are cold (< 16), while the frequencies of a few hot items are larger than 40,000. Given the size of memory usage, 1) if each counter is 4 bits wide, the number of counters ($\mathcal{C}$) will be large, and the estimation of cold items will be very accurate. Unfortunately, hot items will incur overflows of counters, and this can hardly be acceptable in many applications. 2) If we allocate 16 bits to each counter, the number of counters will decrease to $\mathcal{C}/4$, and the accuracy of the sketch will drop drastically. What is worse, the frequency of the hottest item is `unknown` in many applications, which makes it hard to determine the counter size. Unfortunately, existing sketches (CM sketches [9], CU sketches [24], Count sketches [25], and Augmented sketches [8]) have to allocate enough bits for each counter, thus can hardly work well in real data streams that are highly skewed. The `design goal` of this paper is to *devise a framework which not only prevents counters from overflowing without the need of knowing the frequency of the hottest item in advance, but also can achieve high accuracy, high update speed, and high query speed at the same time.*

B. The Proposed Solution

In this paper, we propose a sketch framework, namely the Pyramid sketch, as it employs a pyramid-shaped data structure. The key idea of our Pyramid framework is *to automatically enlarge the size of the corresponding counters according to the current frequency of the incoming item, while achieving close to 1 memory access and 1 hash computation for each insertion.* Our proposed enlarging strategy uses geometric progression to guarantee that any practical large frequency can be represented within bounded memory usage. The pivotal technique is `counter-pair sharing`, which can significantly improve the accuracy.

Considering the significance of insertion speed in high-speed data streams, we further propose another four techniques: `1) word constraint, 2) word sharing, 3) one hashing, and 4) Ostrich policy`, to accelerate the insertion speed, while keeping the high accuracy.

To verify the universality of our sketch framework, we apply Pyramid to four typical sketches: CM sketches, CU sketches, Count sketches, and A sketches. In real data streams, we recommend using $\mathrm{P_{CU}}$ sketch which represents the CU sketch using Pyramid, as it achieves the highest accuracy and highest insertion speed at the same time.

II. Related Work

The issue of estimation of the item frequency in a multiset is a fundamental problem in databases. The solutions can be divided into the following three categories: sketches, Bloom filter variants, and counter variants.

Sketches: Typical sketches include CM sketches [9], CU sketches [24], Count sketches [25], Augmented sketches

Tong Yang, Yang Zhou, Haowei Zhang, Hao Jin, Bin Cui, and Xiaoming Li are with the Department of Computer and Science, Peking University, Beijing, China.

Shigang Chen is with the Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL 32611 USA. Email: {sgchen@cise.ufl.edu}.

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.

[8], and more [26], [27]. A comprehensive survey about sketch algorithms is provided in the literature [10]. A CM sketch [9] consists of d arrays, A_1...A_d, and each array consists of w counters. There are d hash functions, $h_1(.) \dots h_d(.)$ $(1 \leqslant h(.) \leqslant w)$. When inserting an item e, the CM sketch increments all the `d mapped counters`[1] $A_1[h_1(e)] \dots A_d[h_d(e)]$ by 1. When querying an item e, the CM sketch reports the minimal one among the d mapped counters. The CU sketch [24] is similar to the CM sketch except that it only increments the smallest counter(s)[2] among the d mapped counters for each insertion. The Count sketch [25] is also similar to the CM sketch except that each array is associated with two hash functions. The Augmented sketch targets at improving accuracy by using one additional filter to dynamically capture hot items, suffering from complexities, slow update and query speed. The A sketch adds an additional filter (which is actually a queue with k items) to an existing sketch $\mathcal{T}$. The A sketch is very accurate only for those items in the filter. The authors focus on querying the hot items by using query sets sampled from the whole multiset. That is why their experimental results of the A sketch significantly outperform the CM sketch. Among these sketches, the CU sketch [24] achieves the highest accuracy when using the same amount of memory. Unfortunately, *all these sketches have three shortcomings for skewed datasets: 1) not accurate enough; 2) requiring multiple memory accesses and hash computations for each insertion; 3) requiring to know the approximate frequency of the hottest item in advance.*

Bloom filter variants: The second kind of solutions is based on Bloom filter [28]. A standard Bloom filter can tell whether an item belongs to a set or not, but *cannot* estimate its frequency. Counting Bloom filters (CBF) [29] can be used to estimate the frequency of items in a multiset. CBF is quite similar to the CM sketches, except that CBF uses only one array. Several improvements based on CBF have been proposed, such as the Spectral Bloom Filter (SBF) [30], the Dynamic Count Filter (DCF) [31] and the Shifting Bloom filter [32], [33], and they all can store frequencies of items.

Counter variants: Two typical algorithms are the counter braids [34] and Random Counters [3]. Counter braids can report the frequencies of all items one time by post processing. It needs to know the IDs of all distinct items. The authors of Counter braids also admit that it cannot support instant point query [34]. The estimation method in Random Counters [3] is called CSM. It can achieve fast update speed at the cost of accuracy.

Summary: Although there are various algorithms for frequency estimation of multisets, no existing sketch can achieve high accuracy and one memory access per insertion, especially for skewed datasets.

III. Pyramid Sketch Framework

Our Pyramid framework consists of two key techniques: `counter-pair sharing` and `word acceleration`. Counter-pair sharing is used to *dynamically assign appropriate number of bits for different items with different frequencies*. Word acceleration can achieve one memory access and one hash computation for most updates, significantly accelerating the update speed of 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.

A. Technique I: Counter-pair Sharing

Data Structure: As shown in Figure 1, our Pyramid framework consists of λ layers, where L_i denotes the i^{th} layer. L_i consists of w_i counters where $w_{i+1} = w_i/2$ $(1 \leqslant i \leqslant \lambda - 1)$, and each counter contains δ bits. We represent the j^{th} counter of the i^{th} layer with $L_i[j]$. The first layer L_1 is associated with d independent hash functions $h_i(.)$ $(1 \leqslant i \leqslant d)$, whose outputs are uniformly distributed in the range $[1, w_1]$. The i^{th} layer L_i is associated with the $i + 1^{th}$ layer L_{i+1} in the following way: two adjacent counters at L_i are associated with one counter at L_{i+1}. In other words, $L_i[2j-1]$ and $L_i[2j]$ are associated with $L_{i+1}[j]$. $L_i[2j-1]$ and $L_i[2j]$ are defined as the **sibling counters**. $L_{i+1}[j]$ is defined as the **parent counter** of $L_i[2j-1]$ and $L_i[2j]$. $L_i[2j-1]$ is defined as the **left child counter** of $L_{i+1}[j]$, while $L_i[2j]$ is defined as the **right child counter** of $L_{i+1}[j]$.

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 is only used for recording frequencies. In other words, all the δ bits in the pure counters are used for counting, representing a range $[0, 2^{\delta} - 1]$. The hybrid counter with δ bits is split into three parts: *the left flag, the counting part, and the right flag*. The left flag (1 bit) indicates whether its left child counter is overflowed, while the right flag (1 bit) indicates whether its right child counter is overflowed. The counting part ($\delta - 2$ bits) ranging from 0 to $2^{\delta-2} - 1$ is used for counting. We use $L_i[j].lflag$, $L_i[j].count$, $L_i[j].rflag$ to represent the three parts of counter $L_i[j]$.

There are following three primary operations in our Pyramid framework: insertion, deletion, and query. Initially, all the counters at all the layers are 0.

Insertion: When inserting an item e, we first compute the d hash functions $h_1(e), h_2(e), ..., h_d(e)$ $(1 \leqslant h(.) \leqslant w_1)$ 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 will perform different incrementing operations on these d counters. During the incrementing process, if any of the d counters overflows, we simply record the overflow information in its parent counter. This is called **carryin** operation.

The carryin operation is based on the following observation about practical datasets that are skewed: *the number of overflowed counters is small, and in most cases at most one of the sibling counters overflows.* Consider that $L_1[j]$ overflows. Let its parent counter be $L_2[j']$. Without loss of generality, assume $L_1[j]$ is the left child of $L_2[j']$. We check $L_2[j'].lflag$. If the flag is off, we turn it on and increment $L_2[j'].count$; if the flag is on, we only increment $L_2[j'].count$. If $L_2[j'].count$ does not overflow, insertion ends. Otherwise, we repeat the carryin operation at layer L_2, and the operation will be performed

[1]For convenience, we call them d `mapped counters`.

[2]We use counter(s) because there may be multiple smallest counters.

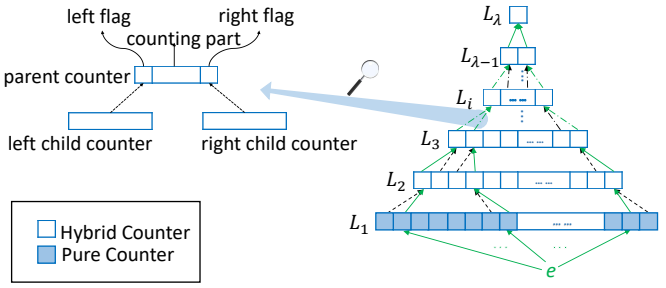
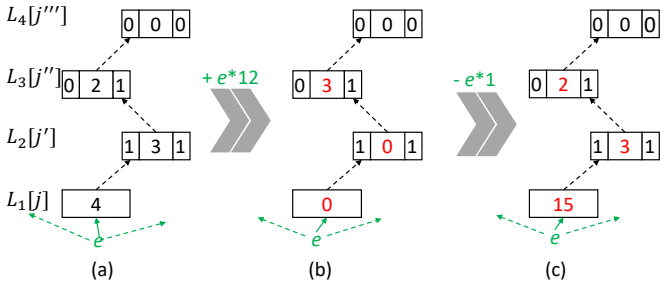


Fig. 1. Counter-pair sharing technique.



layer by layer until there is no overflow. In this way, we actually dynamically assign an appropriate number of higher-order bits to record the incoming items, so as to minimize memory waste.

Example I*:* As shown in Figure 1 (a)(b), every pure counter consists of 4 bits, so does every hybrid counter. In each hybrid counter, the counting parts consist of 2 bits. As shown in the figure, the value of $L_1[j]$, the three parts of $L_2[j']$, $L_3[j'']$ and $L_4[j''']$ are $4, <1,3,1>, <0,2,1>, <0,0,0>$, respectively. Suppose $L_1[j]$ is incremented by 12, $L_1[j]$ overflows, and the carryin operations are performed as follows.

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

Deletion: This framework supports deletions if and only if the sketch used in our framework supports deletions, such as the CM sketch [9], the Count sketch [25], *etc.* To delete an item e, we first check the d mapped counters $L_1[h_1(e)], L_1[h_2(e)], ..., L_1[h_d(e)]$, and perform different deletion strategies according to the specific sketch. *The operation of decrementing a counter is exactly the reverse process of incrementing a counter.* Specifically, to decrement a pure counter $L_1[j]$, if it is non-zero, we just decrement it by 1. Otherwise, we set $L_1[j]$ to the maximum value $(2^\delta - 1)$, and then decrement its parent counter recursively. Without loss of generality, we only show the situation of only accessing or modifying *left flags*. To decrement a hybrid counter $L_i[j]$, $L_i[j].lflag$ must be on. There are the following three cases. 1) If $L_i[j].count$ is larger than 1, we just decrement it by 1. 2) If $L_i[j].count$ is 0, we just set it to $2^{\delta-2} - 1$, and then decrement its parent counter recursively. 3) If $L_i[j].count$ is 1, we first set it to 0, and turn $L_i[j].lflag$ off if the left flag of its parent counter is off.

Example III*:* As shown in Figure 1 (b)(c), the parameters of pure counter and hybrid counters are same as those of Example I. As shown in the figure, the value of $L_1[j]$, the three parts of $L_2[j']$ and $L_3[j'']$ and $L_4[j''']$ are $0, <1,0,1>, <0,3,1>, <0,0,0>$, respectively. Suppose $L_1[j]$ is decremented by 1 and the deletion operations are performed as follows.

1) $L_1[j]$ is set to 15;
2) $L_2[j'].lflag$ keeps on, and $L_2[j'].count$ is set to 3;
3) $L_3[j''].rflag$ keeps on, and $L_3[j''].count$ is set to 2.

Query: When querying an item e, we first compute d hash functions to find the d mapped counters $L_1[h_1(e)]$, $L_1[h_2(e)], ..., L_1[h_d(e)]$. These d mapped counters at layer

Algorithm 1: ReportVal(i, j_i).

```
/* assume that all the child counters
   are the left child counters                    */
1 if i==1 then
2     return L_1[j_1] + ReportVal(i+1, j_{i+1});
3 if L_i[j_i].lflag==false then
4     return 0;
5 if L_i[j_i].lflag==true && L_i[j_i].rflag==true then
6     return (L_i[j_i].count - 1) × 2^{δ+(i-2)×(δ-2)} +
        ReportVal(i+1, j_{i+1})
7 else
8     return L_i[j_i].count × 2^{δ+(i-2)×(δ-2)} +
        ReportVal(i+1, j_{i+1})
```

L_1 share the same array of w_1 counters. Below we focus on describing the operations of querying the first mapped counter $L_1[j_1]$ $(j_1 = h_1(e))$, and the operations for other $d-1$ counters are analogous. Let the parent counter and the ancestor counters of $L_1[j_1]$ be $L_2[j_2]$, $L_3[j_3]$, ..., $L_\lambda[j_\lambda]$, respectively. Without loss of generality, we consider the case where $L_i[j_i]$ is the left child of $L_{i+1}[j_{i+1}]$, $for 1 \le i < \lambda$. According to Algorithm 1, we recursively assemble the counter value top-down, layer by layer, until the left flag is off. The result is a final **reported value** $ReportVal(1, j_1)$, also denoted as $\mathcal{R}(L_1[j_1])$ for convenience. In line $5-8$, if the left flag and right flag of $L_i[j_i]$ are both on, its left child counter and right child counter must have both overflowed *at least* once. Therefore, we subtract 1 from $L_i[j_i].count$, so as to reduce the over-estimation error incurred due to collision in counter-pair sharing. 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 produce the query output based on the specific sketch under use. For example, for CM and CU, we simply report the minimum value of the d reported values.

Example III*:* As shown in Figure 1 (c), the parameters of pure counter and hybrid counters are same as those of Example I. As shown in the figure, the value of $L_1[j]$, the three parts of $L_2[j']$ and of $L_3[j'']$ and $L_4[j''']$ are 15, $<1,3,1>, <0,2,1>, <0,0,0>$. The process of getting $\mathcal{R}(L_1[j])$ is shown as follows.

1) $L_1[j]$ is 15;
2) $L_2[j'].lflag$ is on and $L_2[j'].count$ is 3;
3) $L_3[j''].rflag$ is on and $L_3[j''].count$ is 2;
4) $L_4[j'''].lflag$ is off and the recursive operation ends;

5) The reported value is $15+(3-1)*2^4+(2-0)*2^6 = 175$.

Summary: With our *counter-pair sharing* technique, we can use fine-grained counters (*e.g.*, 4 bit per counter) to record the frequencies of mouse items, elephant items, and very large items (*e.g.*, frequency of 2^{24}). The number of layers is dynamically increasing when the incoming item is very large, but the total memory is fixed as the size of layers follows the decreasing geometric sequence.

B. Technique II: Word Acceleration

Word Constraint Technique: In the word constraint technique, we make two minor modifications: 1) we set the counter size δ to 4 bits; 2) we confine the d mapped counters at layer L_1 to a single machine word. In this way, the average number of memory accesses per insertion or query is significantly reduced. Let the size of a machine word be $\mathcal{W}$ bits. Each machine word is comprised of $\mathcal{W}/4$ counters. Because there are w_1 counters at layer L_1, which translates to $4w_1/\mathcal{W}$ machine words. Layer L_1 is associated with $d+1$ hash functions $h_i(.)$ $(1 \leqslant i \leqslant d+1)$. The first hash function is used to associate each item with a specific word $\boldsymbol{\Omega}$, and the remaining d hash functions are used to identify d mapped counters in the word $\boldsymbol{\Omega}$. Our word constraint technique is based on the following facts: 1) In our framework, each counter is small (e.g., 4 bits), while a machine word is usually 64 bits wide on many of today's CPUs. 2) The size of a machine word can be much larger on GPU platforms. For example, one memory access can read up to 1024 bits in some commodity GPUs [35].

Therefore, one machine word on CPU or GPU can typically contain a reasonably large number of small counters used in our framework. Note that the actual operations of insertion, deletion and query stay the same under the word constraint.

Obviously, after using the word constraint technique, the average numbers of memory accesses per insertion, deletion or query are reduced to around $1/d$, as all the d mapped counters can be read/written within one memory access. Next, we derive an upper-bound of the average number of memory accesses for each insertion as follows. When inserting an item, suppose Pr(layer L_1 overflows) $< \rho$ and Pr(layer L_{i+1} overflows $\mid$ layer L_i overflows) $< \sigma$ (< 1) $(1 \leqslant i < \lambda)$. The average number of memory accesses represented by $\bar{t}$ is determined by the following formula.

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

In our experiments of IP traces, ρ is approximate to 0.05 and σ is approximate to 0.25. Therefore, the average number of memory accesses for each insertion $\bar{t}$ is less than $1+0.05/(1-0.25) \approx 1.07$, which is consistent with our experimental results shown in Figure 18.

On top of the Pyramid sketch with only counter-pair sharing, adding the technique of word constraint helps reduce memory accesses per insertion or query, but incurs severe accuracy loss in the meantime. The main reason for accuracy loss is that after implementing the carryin operation, the probability of collision among counters in the same machine word increases sharply at higher layers. More specifically, given an item e, its

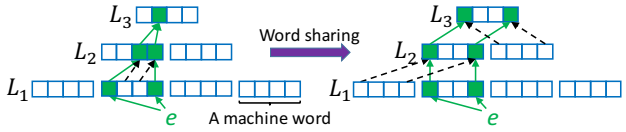


Fig. 2. Example of word sharing technique.

d counters at layer L_1 are mapped to one machine word, while the parent counters of these d counters are mapped to half of a word at layer L_2. The ancestor counters are mapped to even small ranges in a word at higher layers, resulting in more collisions. To address this issue, we propose a new technique as follows.

Word Sharing Technique: The methodology of this technique is managing to make the parent counters and ancestor counters of the d mapped counters always fall in a constant range (*i.e.*, a machine word) instead of smaller and smaller ranges, so as to reduce collisions. Specifically, our word sharing technique works as follows: 1) Similar to the definition of parent counter, left child counter, right child counter, we have left child word, right child word, and parent word, where 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, the collisions in counter-pair sharing are alleviated, and the accuracy is significantly improved.

Example $\mathbb{IV}$*:* In Figure 2, we set $\lambda = 3$, $d = 2$, and $w_1 = 16$. Each machine word consists of 4 counters. Without the word sharing technique, the d mapped counters at layer L_1 belong to a machine word, their parent counters belong to half of a machine word at layer L_2, and their layer-3 ancestors belong to a quarter word, which is one counter in this example. In contrast, with the word sharing technique, the parent/ancestor counters of the d mapped counters always fall in a single machine word at each layer.

One Hashing Technique: More hash computations result in slower speeds for update and query. Ideally, only one hash computation is performed for each insertion, deletion or query. Towards this goal, we propose to use one hashing technique. The idea is to *split the value that a 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. Given a hash function with 32-bit output, we may use the first 16 bits to locate a word in a Pyramid sketch. Suppose a word is 64 bits long. Locating one of the 16 counters in a word requires 4 hash bits. In total, we need 16 hash bits to locate 4 counters in this word. In this way, we can use only one hash computation to handle a Pyramid sketch which originally requires 4 hash computations with at most 2^{16} words at the layer L_1. Similarly, we can use a hash functions with 64-bit output to support a Pyramid sketch ($d = 4$) with at most 2^{48} words, *i.e.*, 2048 TB memory at the layer L_1, which should be large enough for all practical cases.

Summary: With our *word acceleration* technique, our algorithm only needs one memory access and one hash computation for most items. Indeed, several layers may need to be

updated for some items, and updating a layer requires one memory access. Fortunately, as most items are very small, and only access the first layer. Our experimental results on real data stream show that the average number of layers need to be updated are around 1.07, which is close to one memory access per operation.

C. Further Optimization Method

Ostrich Policy: For sketches which need to know the reported values of the d mapped counters during each insertion (such as the CU sketch), multiple layers need to be accessed. To reduce the number of layer accesses, we propose a novel strategy, namely Ostrich policy. The key idea of Ostrich policy is ignoring the second and higher layers when getting the reported values of the d mapped counters. Here we take the CU sketch as an example. When inserting an item e, suppose the counter(s) 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 that of the d mapped counters. If there are multiple smallest ones, we perform the increment operation on all of them. Through the Ostrich policy, the insertion speed of our framework will be significantly improved.

One may argue that Ostrich policy could degrade the accuracy a lot. Actually, our experimental results show that Ostrich policy does help in improving the accuracy. Let us take the CU sketch as an example to explain this counter-intuitive phenomena. The CU sketch always increments the smallest counter(s) by 1 for each insertion. However, in many cases, the smallest counter(s) are already larger than their real frequencies. 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 contributes to better accuracy. Ostrich policy is one efficient implementation of this new strategy.

D. Advantages over Prior Art

Compared to existing sketches, our Pyramid framework has the following advantages: 1) It is much more memory efficient because any an incoming item with different frequency are dynamically assigned appropriate number of bits. In other words, when using the same memory size, our framework can achieve a much higher accuracy. 2) It is much faster in terms of update and query speed, because it can read or write d mapped counters in one memory access, and only one hash computation is needed. 3) What is more, it also addresses another shortcoming of all existing sketches – word alignment. For example, given a multiset with a maximum frequency of 1000, the counter size in existing sketches should be 10 bits. As a result, some counters will traverse two bytes or two words, which will make the read or write operations of counters inefficient. In contrast, in our Pyramid framework, the word alignment can be perfectly achieved by setting the counter size δ to 4 bits.

IV. Multi-Word Optimization

Below, we show the multi-word constraint technique, which is extended from the word constraint technique introduced in Section III-B, to offer an option to balance the speed

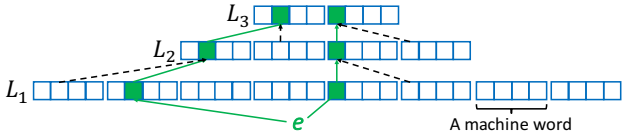


Fig. 3. Example of multi-word constraint technique.

performance and accuracy performance. Specifically, multi-word constraint technique can further improve the accuracy of Pyramid sketch at the cost of a little speed degrading.

In multi-word constraint technique, 1) we set the counter size δ to 4 bits; 2) we confine the d mapped counters at layer L_1 to K ($K \leqslant d$) machine words. Layer L_1 is associated with $d+K$ hash functions $h_i(.)$ ($1 \leqslant i \leqslant d+K$). The first K hash functions are used to associate each item with K specific machine words $\boldsymbol{\Omega}_1 ..., \boldsymbol{\Omega}_K$, and the remaining d hash functions are used to identify d mapped counters in these machine words. The d mapped counters are uniformly assigned to and located in the K machine words $\boldsymbol{\Omega}_1 ..., \boldsymbol{\Omega}_K$. In this way, multi-word constraint increases the randomness of locations of the d mapped counters, which correspondingly improves the accuracy performance of our framework. Similar to the analysis of 1, we get the average number of memory accesses after using multi-word constraint as follows:

$$\bar{t} < K + K \times \sum_{k=0}^{\infty} \rho\sigma^k \quad = K + \frac{K \times \rho}{1-\sigma} \tag{2}$$

Since in our experiments on IP traces, ρ is approximate to 0.05 and σ is approximate to 0.25, $\bar{t}$ is less than $K + K \times 0.05/(1-0.25) \approx 1.07K$. Besides, our multi-word constraint technique is fully compatible with the aforementioned word sharing technique.

Example $\mathbb{V}$*:* In Figure 3, we set $\lambda = 3$, $d = 2$, $w_1 = 32$, and $K = 2$. Each machine word consists of 4 counters. We first use two hash functions to locate two machine words (the 2^{nd} and 5^{th}) at layer L_1. Then we assign each of the two mapped counters to each machine word (uniform assignment). Take the 2^{nd} machine word for example. The 2^{nd} counter in the 1^{st} word and the 2^{nd} counter in the 2^{nd} word at layer L_1 share the 2^{nd} counter in the 1^{st} word at layer L_2. The sharing case between layer L_2 and layer L_3 is similar to this.

The key idea of one hashing technique also applies to multi-word constraint technique. We can split the value produced by a hash function into several segments and use these segments to locate words or counters. For example, suppose there are 2^{16} 64-bit machine words at layer L_1 (the total memory of this Pyramid sketch is 1MB) and $d = 4$. Locating a machine word requires 16 hash bits and locating a counter in one specific machine word requires 4 hash bits. For K-word constraint case, we need $K*16+4*4 = 16(K+1)$ hash bits. Therefore, one hash function with 64-bit output can handle the multi-word case with $K = 2$ or $K = 3$, and two hash functions can handle such case with $K = 4$. In addition, Ostrich policy keeps unchanged after applying multi-word constraint technique.

Other methods and discussions. There are many other methods. For example, we can use only 1 hash function instead of K hash functions, and associate each item with K specific machine words $\Omega_1, \Omega_1+1, \ldots, \Omega_1+K-1$, where Ω_1 derives

from that hash function. In this way, we can also increase the randomness of locations of the d mapped counters, and we just need extra 1 hash function, which reduces the cost of the speed degrading. However, according to our experiments[3], we find that the randomness of this method is not as good as that of using K hash functions.

V. Mathematical Analyses

In this section, we derive the correct rate and error bound when applying Pyramid framework to the CM sketch. We call the CM sketch after using this framework the $\mathrm{P_{CM}}$ sketch. Similarly, we have $\mathrm{P_{CU}}$, $\mathrm{P_C}$, $\mathrm{P_A}$.

A. Claim of No Under-estimation Error

We claim that $\mathrm{P_{CM}}$ sketch has no under-estimation error. Under-estimation error means that *the querying value is smaller than the real frequency*. Due to space limitation, we present the proof in the appendix of the technical report [2].

B. Correct Rate of the P_{CM} sketch

Given a multiset with N distinct items, we build a $\mathrm{P_{CM}}$ sketch. Let N_i[4] $(1 \leqslant i \leqslant \lambda)$ denote the number of distinct items whose corresponding mapped d counters' carryin operations stop exactly at layer L_i. Without loss of generality, we simply assume that the d mapped counters' carryin operations all stop at the same layer.

Let P_i^{acc} denote the probability that one arbitrary counter corresponding with one arbitrary item stores the accurate value at layer L_i. P_i^{acc} equals to the probability that no collision happens in one certain counter at layer L_i:

$$\begin{aligned} P_i^{acc} &= \left(\frac{w_i - 1}{w_i}\right)^{(\Phi_i \times N - 1)\times d} \times \left(\frac{\mathcal{W}/\delta - 1}{\mathcal{W}/\delta}\right)^{d-1} \\ &= \left(1 - \frac{2^{i-1}}{w_1}\right)^{(\Phi_i \times N - 1)\times d} \times \left(1 - \frac{\delta}{\mathcal{W}}\right)^{d-1} \quad (1 \leqslant i \leqslant \lambda) \end{aligned} \tag{3}$$

where

$$\Phi_i = \begin{cases} \frac{\sum_{k=i}^{\lambda} N_k}{N} & (1 \leqslant i \leqslant \lambda) \\ 0 & (i = \lambda + 1) \end{cases} \tag{4}$$

and $\Phi_i - \Phi_{i+1} = N_i/N$ $(1 \leqslant i \leqslant \lambda)$.

Let $\mathcal{P}$ denote the expectation of the probability that one arbitrary counter at layer L_1 reports the accurate result. We calculate every portions of this probability layer by layer and report the average value using the weight of N_i:

$$\begin{aligned} \mathcal{P} &= \frac{\sum_{k=1}^{\lambda}\left[N_k \times \prod_{l=1}^{k} P_l^{acc}\right]}{N} \\ &= \sum_{k=1}^{\lambda}\left[(\Phi_k - \Phi_{k+1}) \times \prod_{l=1}^{k} P_l^{acc}\right] \end{aligned} \tag{5}$$

Let C_r denote the correct rate of the estimation for one arbitrary item. This item suffers from an over-estimation error only in the condition that there are collisions in all the d mapped counters, thus we get:

$$C_r = 1 - (1 - \mathcal{P})^d \tag{6}$$

For the same multiset, we build a CM sketch comprised of d arrays: $\mathrm{A}_1 ... \mathrm{A}_d$, each of which consists of w counters. Let $\mathcal{P}'$ denote the expectation of the probability that one arbitrary counter at one arbitrary array A_i $(1 \leqslant i \leqslant d)$ reports the accurate result. We have:

$$\mathcal{P}' = \left(1 - \frac{1}{w}\right)^{N-1} \tag{7}$$

Let C_r' denote the correct rate of the estimation for one arbitrary item in this CM sketch. Analogous to the derivation of the $\mathrm{P_{CM}}$ sketch, we get:

$$C_r' = 1 - \left(1 - \mathcal{P}'\right)^d \tag{8}$$

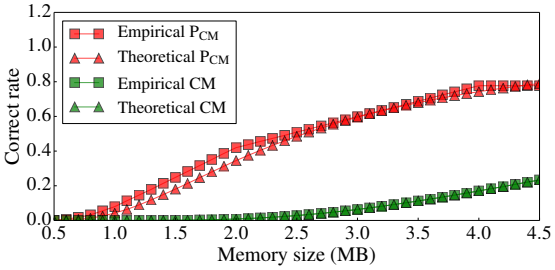


Fig. 4. Comparison of empirical and theoretical correct rate with different memory sizes on one real IP trace.

Figure 4 plots the empirical and theoretical correct rates of the CM and $\mathrm{P_{CM}}$ sketch, on one real IP trace which will be mentioned in the Section VI-A. It shows that the theoretical results are consistent with the empirical results for both the CM and $\mathrm{P_{CM}}$ sketch. As the memory size becomes larger, the theoretical correct rates of the $\mathrm{P_{CM}}$ sketch get closer to the empirical result. This is because the randomicity of hash values can be guaranteed with large enough mapping space. Besides, we observe that the $\mathrm{P_{CM}}$ sketch always significantly outperforms the CM sketch in terms of the correct rate empirically and theoretically.

C. Error Bound of the P_{CM} sketch

Theorem 1. *For an arbitrary item e_i, let $\hat{f}_i$ denote its estimated frequency and f_i denote its real frequency. Let N denote the number of distinct items and V denote the sum of all items' real frequencies,* i.e., $V = \sum_{k=1}^{N} f_i$. *The Φ_i is defined in Formula 4. Give a small variable ϵ, we have the following guarantee with probability at least $1 - \left(\frac{\Delta}{\epsilon}\right)^d$ (Δ is a constant relying on $N, \Phi_i, w_1, d, \mathcal{W}$ and δ):*

$$\hat{f}_i \leqslant f_i + \epsilon \times V \tag{9}$$

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

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 independent

[3]Due to space limitation, we do not present the figures.

[4]N_i can be derived through the dataset's specific distribution or obtained by experiments.

hash functions, the expectation of this indicator variable can be derived as follows:

$$E\left(I_{i,j,k,l}\right)=\frac{1}{w_l}\times\frac{\Phi_l\times N\times d-d}{\Phi_l\times N\times d-1}+\frac{1}{\mathcal{W}/\delta}\times\frac{d-1}{\Phi_l\times N\times d-1}\quad(1\leqslant l\leqslant\lambda)\tag{10}$$

For convenience, let E_l denote $E\left(I_{i,j,k,l}\right)$, and let β_l denote $2^{\delta+(\delta-2)\times(l-2)}$). We define the variable $X_{i,j}$ as follows:

$$\begin{aligned}X_{i,j}=&(\Phi_1-\Phi_2)\times\sum_{k=1}^{N}\left(f_k\times I_{i,j,k,1}\right)+\\&\sum_{l=2}^{\lambda}\left[(\Phi_l-\Phi_{l+1})\times\sum_{k=1}^{N}\left(\frac{f_k}{\beta_l}\times I_{i,j,k,l}\times\beta_l\right)\right]\\=&\sum_{l=1}^{\lambda}\left[(\Phi_l-\Phi_{l+1})\times\sum_{k=1}^{N}\left(f_k\times I_{i,j,k,l}\right)\right]\end{aligned}\tag{11}$$

Obviously, it can be guaranteed that $X_{i,j}$ is a non-negative variable. $X_{i,j}$ 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:

$$\mathcal{R}(L_1[h_j^1(e_i)])=f_i+X_{i,j}\tag{12}$$

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

$$\begin{aligned}E\left(X_{i,j}\right)&=E\left\{\sum_{l=1}^{\lambda}\left[(\Phi_l-\Phi_{l+1})\times\sum_{k=1}^{N}\left(f_k\times I_{i,j,k,l}\right)\right]\right\}\\&=\sum_{l=1}^{\lambda}\left[(\Phi_l-\Phi_{l+1})\times\sum_{k=1}^{N}\left(f_k\times E_l\right)\right]\\&=\sum_{l=1}^{\lambda}\left[(\Phi_l-\Phi_{l+1})\times E_l\times\sum_{k=1}^{N}f_k\right]\\&=V\times\sum_{l=1}^{\lambda}\left[(\Phi_l-\Phi_{l+1})\times E_l\right]\\&=V\times\Delta\end{aligned}\tag{13}$$

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

$$V=\frac{E(X_{i,j})}{\Delta}\tag{14}$$

Then, by the Markov inequality, we get:

$$\begin{aligned}&Pr\left[\hat{f}_i\geqslant f_i+\epsilon\times V\right]\\&=Pr\left[\forall_j.\ \mathcal{R}(L_1[h_j^1(e_i)])\geqslant f_i+\epsilon\times V\right]\\&=Pr\left[\forall_j.\ X_{i,j}\geqslant\epsilon\times V\right]\\&=Pr\left[\forall_j.\ \frac{X_{i,j}}{E(X_{i,j})}\geqslant\frac{\epsilon}{\Delta}\right]\\&\leqslant\left\{E\left[\frac{X_{i,j}}{E(X_{i,j})}\right]/\frac{\epsilon}{\Delta}\right\}^d\\&=\left(\frac{\Delta}{\epsilon}\right)^d\end{aligned}\tag{15}$$

□

Using the same notations employed in the Theorem 1, we transform the error bound of the CM sketch given by the literature [9] into the following form:

Theorem 2. *Give a small variable ϵ, we have the following guarantee with probability at least $1-\left(\frac{1/w}{\epsilon}\right)^d$ (w is the number of counters of each array in the CM sketch):*

$$\hat{f}_i\leqslant f_i+\epsilon\times V\tag{16}$$

Figure 5 plots the empirical and theoretical guaranteed probabilities of the CM and $\mathrm{P_{CM}}$ sketch, on the synthetic dataset with skewness of 0.0 which will be mentioned in the Section VI-A. It shows the error bound of $\mathrm{P_{CM}}$ sketch is much better than that of the CM sketch. We use the synthetic dataset with skewness of 0.0 to evaluate the worst case performance (see Figure 11 and 12) of sketches, verifying the bounds of CM and $\mathrm{P_{CM}}$. Note that in Figure 11 and 12, the AAE equals to the average $\hat{f}_i-f_i$ of all distinct items.

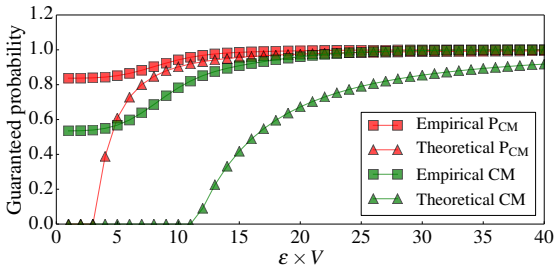


Fig. 5. Comparison of empirical and theoretical error bound on one synthetic dataset with skewness 0.0.

D. Comparison of Space Utilization Ratio

In this subsection, we focus on the space utilization ratio, which is defined as B_{use}/B_{sum}, where B_{use} denotes the number of bits that have changed in the insertion process, and B_{sum} denotes the number of bits allocated in total. The notations are the same as those in Section V-C.

We first derive the space utilization ratio of the CM sketch. For convenience, we use $value^i$ to denote the sum of values of i arbitrary counters.

Suppose the CM sketch consists of r arrays, each array consists of w counters, and each counter consists of δ bits. For B_{sum}, it can be derived as $r*w*\delta$. For B_{use}, since each counter is independent of each other, the expectation of B_{use} can be derived as follows:

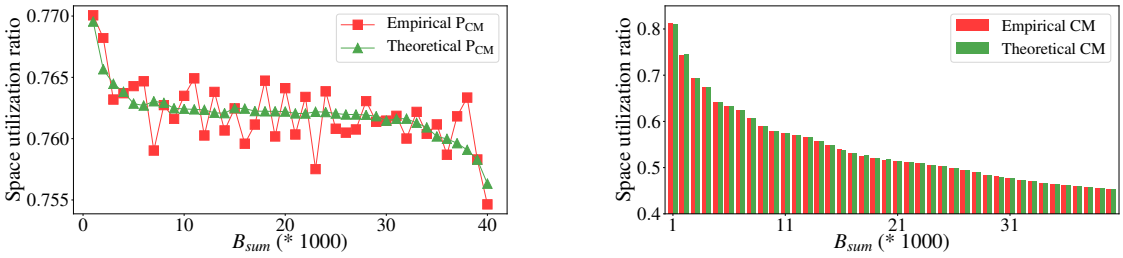
$$E(B_{use})=r*w*\sum_{i=0}^{\delta-1}P(value^1\geqslant 2^i)\tag{17}$$

We use the dynamic programming algorithm to calculate $P(value^1\geqslant 2^i)$. Let $dp_{i,j}^x$ denote the probability that the sum of values of x arbitrary counters equals to j when the first i distinct items are inserted, we have:

$$dp_{i,j}^1=dp_{i-1,j}^1*\frac{w-1}{w}+dp_{i-1,j-f_i}^1*\frac{1}{w}\tag{18}$$

where f_i means the frequency of the item e_i.

Proof. Since there are two conditions for e_i: e_i is mapped to that counter; e_i is not mapped to that counter. Based on



(a) P_{CM}. (b) CM.

Fig. 6. Comparison of empirical and theoretical space utilization ratio on one synthetic dataset with skewness 0.6.

the principle of probability accumulation, we get the Equation 18. □

Then, $P(value^1 \geqslant 2^i)$ can be derived as follows:

$$P(value^1 \geqslant 2^i) = \sum_{j=2^i}^{\infty} dp_{i,j}^1 \tag{19}$$

Next, We derive the space utilization ratio of the P_{CM} sketch. For B_{sum}, it can be derived as $\sum_{i=1}^{\lambda} w_i * \delta$. For B_{use}, we divide it into λ parts. Let B_{use}^i denote the number of bits that have changed in the i^{th} layer. We have:

$$E(B_{use}) = \sum_{i=1}^{\lambda} E(B_{use}^i) \tag{20}$$

The derivation of B_{use}^1 is similar to that in the CM sketch. We have:

$$E(B_{use}^1) = w_1 * \sum_{i=0}^{\delta-1} P(value^1 \geqslant 2^i) \tag{21}$$

We can just modify the transformation equation of dp to calculate $P(value^1 \geqslant 2^i)$. Specifically, we have:

$$dp_{i,j}^1 = \sum_{k=0}^{d} dp_{i-1,j-k*f_i}^1 * \left(\frac{1}{w_1}\right)^k * \left(\frac{w_1-1}{w_1}\right)^{d-k} * \binom{k}{d} \tag{22}$$

where k denotes the number of times that this item is mapped to that counter for every insertion.

For $B_{use}^t (2 \leqslant t \leqslant \lambda)$, since the first/last bit of an arbitrary counter will be changed iff its left/right child counter is overflowed, we have:

$$\begin{aligned} E(B_{use}^t) = {} & 2 * P\left(value^{2^{t-2}} \geqslant 2^{\delta+(t-2)*(\delta-2)}\right) \\ & + \sum_{k=0}^{\delta-2} P\left(value^{2^{t-1}} \geqslant 2^{\delta+(t-2)*(\delta-2)+k}\right) \end{aligned} \tag{23}$$

We can also use the dynamic programming algorithm to calculate $P(value^x)$. Specifically, we have:

$$dp_{i,j}^x = \sum_{k=0}^{d} dp_{i-1,j-k*f_i}^1 * \left(\frac{x}{w_1}\right)^k * \left(\frac{w_1-x}{w_1}\right)^{d-k} * \binom{k}{d} \tag{24}$$

and

$$P(value^x \geqslant y) = \sum_{j=y}^{\infty} dp_{i,j}^x \tag{25}$$

where x and y are constants predefined by the user.

Figure 6 plots the empirical and theoretical space utilization ratio of the CM and P_{CM} sketch, on the synthetic dataset with skewness of 0.6 which will be mentioned in the Section VI-A. It shows that the theoretical space utilization ratio is almost the same as the empirical space utilization ratio. The reason behind is that there is no hypothesis or approximation in the derivation process. Furthermore, we find that as the number of bits allocated increases, the space utilization ratio of the CM sketch significantly reduces, while it causes little impact on the space utilization ratio of the P_{CU} sketch.

VI. Performance Evaluation

A. Experimental Setup

Datasets: We use three kinds of datasets as follows, and we present the frequency distributions of these datasets in the appendix of the technical report [2].

1) Real IP-Trace Streams: We obtain the real IP traces from the main gateway at our campus. The IP traces consist of flows, and each flow is identified by its five-tuple: source IP address, destination IP address, source port, destination port, and protocol type. The estimation of item frequency corresponds to the estimation of number of packets in a flow. We divide 10M*10 packets into 10 IP traces, and build a sketch with 1MB memory for each trace. Each dataset includes around 1M flows.

2) Web Page Dataset: We downloaded this dataset from the website [36]. It is built from a spidered collection of web html documents. Each item indicates the number of distinct terms in one web page. Since this raw dataset is very large, we use the first 16M items and estimate the frequency of each distinct items.

3) Synthetic Datasets: We generate 11 stream datasets of packet IDs following the **Zipf** [37] distribution with the fixed total frequency of packet IDs (10M) but different skewnesses (from 0.0 to 1.0 with a step of 0.1) and different numbers of distinct items. Our generator's source codes come from a performance testing tool named Web Polygraph [38].

Implementation: We have implemented the sketches of CM [9], CU [24], C [25] and A [8] in C++. We apply our Pyramid framework to these sketches, and the results are denoted as P_{CM}, P_{CU}, P_C, and P_A. For P_{CM}, P_C and P_A, we apply the proposed techniques of counter-pair sharing and word acceleration (including word constraint, word sharing, and one hashing). *We do not apply the multi-word constraint technique by default, unless otherwise specified.* For the P_{CU}

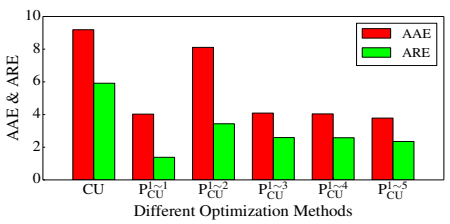


Fig. 7. Comparison of AAE & ARE with different optimization methods on one real IP trace.

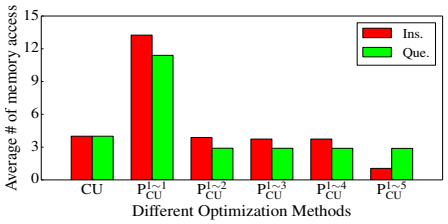


Fig. 8. Comparison of average # memory accesses with different optimization methods on one real IP trace.

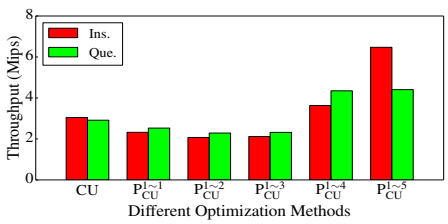


Fig. 9. Comparison of throughput with different optimization methods on one real IP trace.

sketch, we additionally apply the Ostrich policy, which is only suitable for CU. The hash functions used in the sketches are implemented from the 32-bit or 64-bit Bob Hash (obtained from the open source website [39]) with different initial seeds. For the sketches of CM, CU and A, we set the number of arrays to 4 and use 4 32-bit Bob Hashes. For the C sketch, we set the number of arrays to 4 and use 8 32-bit Bob Hashes[5]. For the A sketch, we set its filter size to 32 items as the original paper [8] recommends. For all the above four sketches, we set the counter size to 16 bits in the experiments with the IP traces and synthetic datasets, and to 24 bits in the experiments with the web page dataset, so as to accommodate the maximal frequency of items. For the sketches of P_{CM}, P_{CU}, P_C and P_A, we set the number d of mapped counters to 4, the counter size δ to 4 bits, and the machine word size $\mathcal{W}$ to 64 bits. For each of these Pyramid sketches, we use 1 64-bit Bob hash. In all our experiments with different sketches, we allocate the same amount of memory, 1 MB, by default unless specified otherwise. The number of counters in each experiment can be easily calculated through the allocated memory size and the counter size in the sketch under use. All the implementation source codes are made available at Github [2].

Computation Platform: We performed all the experiments on a machine with 12-core CPUs (24 threads, Intel Xeon CPU E5-2620 @2 GHz) and 62 GB total DRAM memory. CPU has three layers of cache memory: two 32KB L1 caches (one is a data cache and the other is an instruction cache) for each core, one 256KB L2 cache for each core, and one 15MB L3 cache shared by all cores.

B. Metrics

Average Absolute Error (AAE): AAE is defined as $\frac{1}{|\Psi|}\sum_{e_i\in\Psi}|f_i-\widehat{f}_i|$, where f_i is the real frequency of item e_i, $\widehat{f}_i$ is the estimated frequency, and the Ψ is the query set. *We focus on the whole dataset by querying each distinct item only once.* To guarantee that our experiments are conducted head-to-head, we have released the related source codes and datasets at Github [2].

Average Relative Error (ARE): ARE is defined as $\frac{1}{|\Psi|}\sum_{e_i\in\Psi}|f_i-\widehat{f}_i|/f_i$, where the meaning of each notation is the same as that in AAE.

The Average Number of Memory Accesses: The number of memory access is critical when implementing sketches on hardware, such as FPGA and ASIC. Therefore, we measure the average number of memory access of insertion, deletion and query for all sketches.

[5]As mentioned before, in C, each array is associated with 2 hash functions.

Throughput: We perform the insertion and query operations on CPU platform and calculate the throughput using mega-instructions per second (Mips). All the experiments are repeated 100 times to minimize accidental deviations.

C. Effects of Different Techniques

We have proposed five techniques: counter-pair sharing (T_1), word constraint (T_2), word sharing (T_3), one hashing (T_4), and Ostrich policy (T_5). We use $P_{CU}^{1\sim i}$ $(1\leqslant i\leqslant 5)$ to denote the P_{CU} sketch with the first i techniques: T_1,T_2, ..., T_i. Similar notations are introduced for other sketches as $P_{CM}^{1\sim i}$, $P_C^{1\sim i}$, and $P_A^{1\sim i}$ $(1\leqslant i\leqslant 4)$; note that Ostrich policy does not apply to these sketches. In later subsections, we will abbreviate $P_{CM}^{1\sim 4}$ to P_{CM}, $P_{CU}^{1\sim 5}$ to P_{CU}, $P_C^{1\sim 4}$ to P_C, and $P_A^{1\sim 4}$ to P_A.

The experimental performance of the CU sketch in terms of accuracy and speed under the five techniques are shown in Figure 7, 8 and 9. We have the following five observations: 1) The counter-pair sharing technique (T_1) significantly reduces AAE and ARE, and slightly degrades the speeds of both insertion and query. 2) The word constraint technique (T_2) significantly reduces memory accesses per insertion or query, slightly lowers the throughput of both insertion and query, and incurs severe accuracy loss. Lower throughput is caused by extra hash computation for locating a machine word. 3) The word sharing technique (T_3) overcomes the main shortcoming of the word constraint, improving the accuracy without impact on the insertion and query speeds. 4) The one hashing technique (T_4) improves the speeds of insertion and query, while not affecting the accuracy. 5) The Ostrich policy technique (T_5) significantly improves the insertion speed, slightly improve the accuracy, and does not affect the query speed. Note that without using Ostrich policy, $P_{CU}^{1\sim i}$ $(1\leqslant i\leqslant 4)$ always have to access multiple layers to know the reported values of the d mapped counters during each insertion. Therefore, $P_{CU}^{1\sim i}$ $(1\leqslant i\leqslant 4)$ need more memory accesses per query than the final version $P_{CU}^{1\sim 5}$.

D. Accuracy

We apply the Pyramid framework to four typical sketches: the CM, CU, C, and A sketch, and find that P_{CU} achieves the highest accuracy. Therefore, we recommend using the P_{CU} sketch in the application of data streams, and in most experiments, we compare the P_{CU} sketch with the above typical sketches. Note that in each of the parameter settings, the proposed Pyramid framework always improves the accuracies of CM, CU, C, and A. The positive impact of using Pyramid is demonstrated in Figure 10 and 13. In this section, we mainly use the P_{CU} sketch as an example to show the benefits of our Pyramid framework under varied parameter settings.

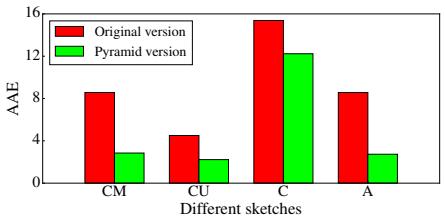


Fig. 10. AAE comparison on the web page dataset.

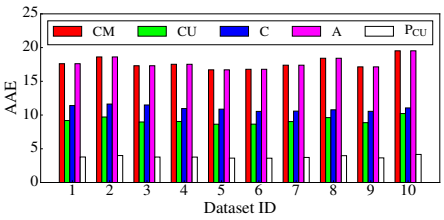


Fig. 11. AAE comparison on different real IP traces.

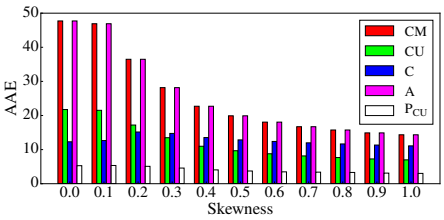


Fig. 12. AAE comparison on synthetic datasets with different skewnesses.

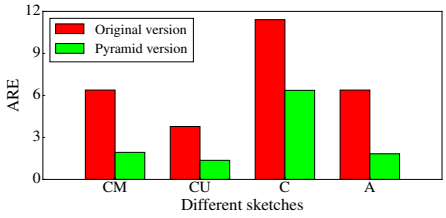


Fig. 13. ARE comparison on the web page dataset.

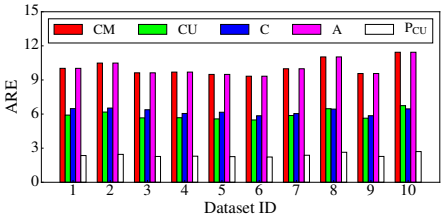


Fig. 14. ARE comparison on different real IP traces.

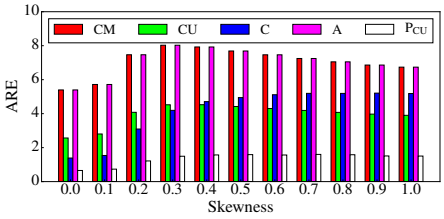


Fig. 15. ARE comparison on synthetic datasets with different skewnesses.

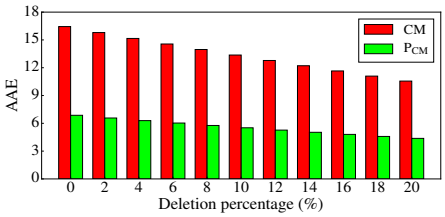


Fig. 16. AAE comparison during deletions on one real IP trace.

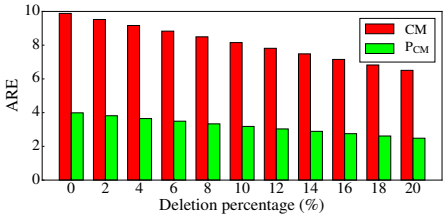


Fig. 17. ARE comparison during deletions on one real IP trace.

*1) **AAE (Average Absolute Error)**: Our experimental results show that on the web page dataset, the AAEs of the CM, CU, C, and A sketch are 3.01, 2.02, 1.26 and 3.14 times higher than these sketches using our framework, respectively.* Figure 10 plots the AAEs of different sketches on the web page dataset. Since the C sketch needs to record the negative numbers during updates, we use one fifth of total memory to serve as the positive or negative markers. Those markers are accessed only when overflows happen during updates. For the C sketch, when we still confine the d mapped counters within only one machine word, the P_C sketch improves the accuracy only a little because of word constraint. To make a better trade-off, we confine the d mapped counters within two separated machine words, compensating for the accuracy loss caused by word acceleration. Note that for the P_{CM}, P_{CU}, and P_A sketch, *we still use one word constraint*.

Our experimental results show that on different IP traces, the AAEs of the CM, CU, C, and A sketch are 4.74, 2.50, 2.89 and 4.73 times higher than the P_{CU} sketch, respectively. Figure 11 plots the AAEs of different sketches on different real IP traces. The reason why the A sketch has the similar accuracy to the CM sketch is explained in Section VI-B.

Our experimental results show that on skewed datasets, the AAEs of the CM, CU, C, and A sketch are 4.75, 2.31, 3.67 and 4.75 times higher than the P_{CU} sketch, respectively. Figure 12 plots the AAEs of different sketches on synthetic datasets with different skewnesses.

Our experimental results show that on one real IP trace, during the process we delete items up to 20%, the AAE of the CM sketch is 2.39 times higher than the P_{CM} sketch. Figure 16 plots the AAEs of the CM and P_{CM} sketch at a deletion percentage increasing from 0 to 20 by 2, on one real IP trace. We first insert 10 MB items into each sketch, then delete a specific percentage of items and calculate the AAEs of those distinct items with nonzero real frequency after each deletion.

*2) **ARE (Average Relative Error)**:*

Our experimental results show that on the web page dataset, the AREs of the CM, CU, C, and A sketch are 3.31, 2.78, 1.79 and 3.50 times higher than these sketches using our framework, respectively. Figure 13 plots the AREs of different sketches on the web page dataset.

Our experimental results show that on different IP traces, the AREs of the CM, CU, C, and A are 4.24, 2.50, 2.42 and 4.24 times higher than the P_{CU} sketch, respectively. Figure 14 plots the AREs of different sketches on different real IP traces.

Our experimental results show that on skewed datasets, the AREs of the CM, CU, C, and A sketch are about 5.02, 2.83, 3.26 and 5.02 times higher than the P_{CU} sketch, respectively. Figure 15 plots the AREs of different sketches on synthetic datasets with different skewnesses.

Our experimental results show that on one real IP trace, during the process we delete items up to 20%, the ARE of the CM sketch is 2.48 times higher than the P_{CM} sketch. Figure 17 plots the AREs of the CM and P_{CM} sketch at a deletion percentage increasing from 0 to 20 by 2, on one real IP trace.

*3) **Experiments on Memory Size and Word Size**:*

We also conduct experiments on memory size and word size. Due to space limitation, we present results in the appendix of the technical report [2].

*4) **Analysis**:*

We find no matter what metrics we use, after applying the pyramid framework, the accuracy always increases. The main

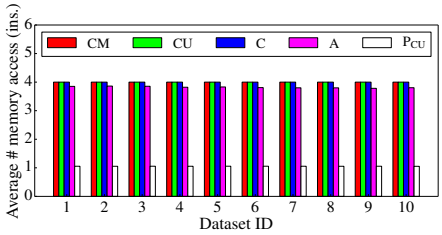


Fig. 18. Comparison of average # memory accesses for each insertion on different real IP traces.

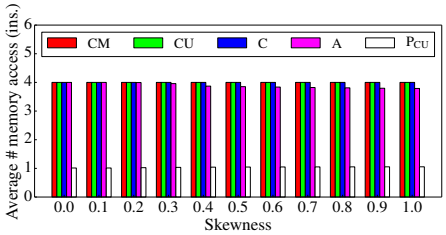


Fig. 19. Comparison of average # memory accesses for each insertion on synthetic datasets with different skewnesses.

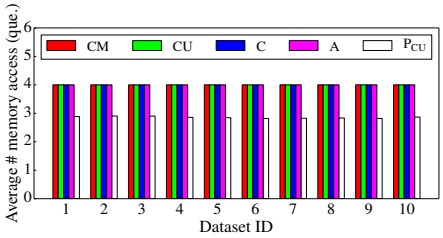


Fig. 20. Comparison of average # memory accesses for each query on different real IP traces.

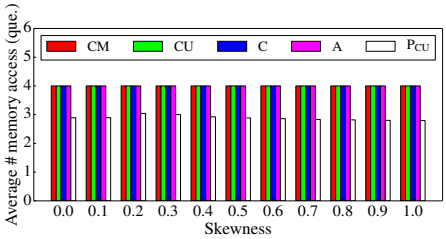


Fig. 21. Comparison of average # memory accesses for each query on synthetic datasets with different skewnesses.

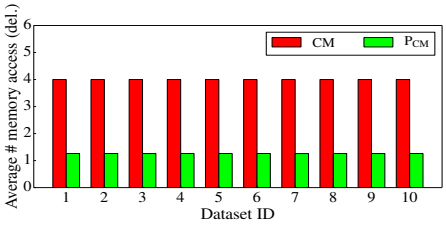


Fig. 22. Comparison of average # memory accesses for each deletion on different real IP traces.

reason is that when using same size of memory, the Pyramid framework uses fine-grained counters (*i.e.*, 4 bits per counter), significantly reducing the memory wastes, and thus achieves higher accuracy. The above experimental results fully verify this conclusion.

E. Speed

We have conducted extensive experiments on speed of the CM, CU, C, and A sketch, and results show that after using the Pyramid framework, the update and query speed of all the sketches are significantly improved. Note that in each of the parameter settings, the Pyramid framework always improves the speeds of CM, CU, C, and A. In this section, we only present the results of $\mathrm{P_{CU}}$.

*1) **The Average Number of Memory Access**:*

In this section, when conducting the statistics for the A sketch, we do not include its number of memory access caused by searching in its filter, while searching the filter probably takes more time than querying the sketch part.

Insertion: *Our experimental results show that on different IP traces, the CM, CU, C, A and P_{CU} sketch need about 4, 4, 4, 3.80 and 1.05 memory accesses for each insertion, respectively.* Figure 18 plots average numbers of memory access of different sketches on different real IP traces during insertions. For the $\mathrm{P_{CU}}$ sketch, the average number is quartered from that of the other sketches by the word constraint method.

Our experimental results show that on skewed datasets, the average numbers of memory access of the CM, CU, C sketch for each insertion keep unchanged (equal to 4), while that of the A sketch is $4.00 \sim 3.79$ *(with a mean of 3.90) and that of the P_{CU} sketch is* $1.01 \sim 1.06$ *(with a mean of 1.04).* Figure 19 plots average numbers of memory access of different sketches for each insertion on synthetic datasets with different skewnesses.

Query: *Our experimental results show that on different IP traces, the CM, CU, C, A and P_{CU} sketch need about 4, 4, 4, 4.00 and 2.85 memory accesses for each query, respectively.* Figure 20 plots average numbers of memory access of different sketches for each query on different real IP traces. We observe that the average number of memory access of the $\mathrm{P_{CU}}$ sketch for each query exceeds 2, owing to the corresponding flags check at the second layer L_2 for each query.

Our experimental results show that on skewed datasets, the average numbers of memory access of the CM, CU, C sketch for each query keep unchanged (equal to 4), while that of the A sketch is about 4.00 and that of the P_{CU} sketch is $2.89 \sim 2.79$ *(with a mean of 2.84).* Figure 21 plots average numbers of memory access of different sketches for each query on synthetic datasets with different skewnesses. We observe that the $\mathrm{P_{CU}}$ sketch performs better on more skewed datasets for query.

Deletion: *Our experimental results show that on different IP traces, the CM and P_{CM} sketch need about 4 and 1.26 memory accesses for each deletion, respectively.* Figure 22 plots average numbers of memory access of different sketches for each deletion on different real IP traces. For the CU sketch, it does not support deletion. For the C and A sketch, their average numbers of memory access for each deletion are similar to that of the CM sketch because they have almost the same structure. We observe that the $\mathrm{P_{CM}}$ sketch considerably outperforms the CM sketch for deletion.

*2) **Throughput**:*

Insertion: *Our experimental results show that on different IP traces, the insertion throughput of the P_{CU} sketch is about 1.97, 2.10, 3.32 and 3.82 times higher than the CM, CU, C and A sketch, respectively.* Figure 23 plots the insertion throughput of different sketches on different real IP traces. We observe that the $\mathrm{P_{CU}}$ sketch always outperforms the other sketches, and at least twice insertion speed.

Our experimental results show that on skewed datasets, the insertion throughput of the P_{CU} sketch is about 1.83, 198, 2.61 and 3.85 times higher than the CM, CU, C and A sketch, re-

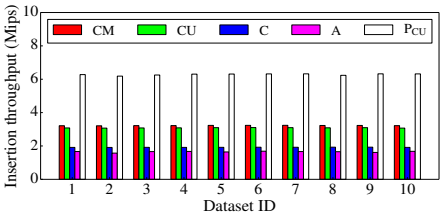


Fig. 23. Comparison of insertion throughput on different real IP traces.

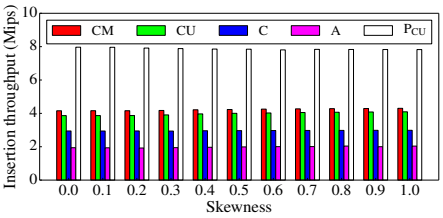


Fig. 24. Comparison of insertion throughput on synthetic datasets with different skewnesses.

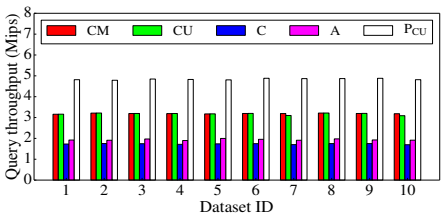


Fig. 25. Comparison of query throughput on different real IP traces.

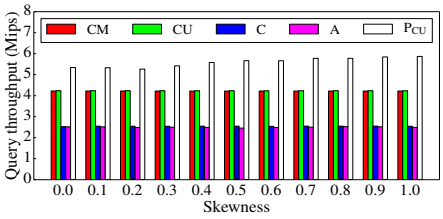


Fig. 26. Comparison of query throughput on synthetic datasets with different skewnesses.

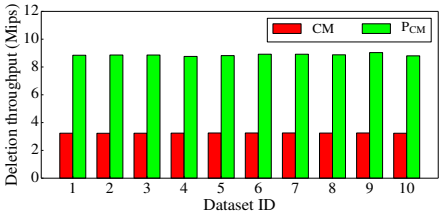


Fig. 27. Comparison of deletion throughput on different real IP traces.

spectively. Figure 24 plots the insertion throughput of different sketches on synthetic datasets with different skewnesses.

Query: *Our experimental results show that on different IP traces, the query throughput of the P_{CU} sketch is about 1.50, 1.50, 2.82 and 2.51 times higher than the CM, CU, C and A sketch, respectively.* Figure 25 plots the query throughput of different sketches on different real IP traces.

Our experimental results show that on skewed datasets, the query throughput of the P_{CU} sketch is about 1,33, 1.33, 2.24 and 2.25 times higher than the CM, CU, C and A sketch, respectively. Figure 26 plots the query throughput of different sketches on synthetic datasets with different skewnesses.

Deletion: *Our experimental results show that on different IP traces, the deletion throughput of the P_{CM} sketch is about 2.73 times higher than the CM sketch.* Figure 27 plots the deletion throughput of different sketches on different real IP traces. We observe that the $\mathrm{P_{CM}}$ sketch considerably outperforms the CM sketch for deletion.

*3) **Analysis**:*

As mentioned above, our Pyramid sketch just needs one hash computation per operation, and one memory access per layer for each update or query. Compared to other sketches, the number of hash computations and memory accesses are significantly smaller. Therefore, after applying the pyramid framework, the insertion speed significantly increases.

F. Effects of Multi-Word Optimization

Below, we test the accuracy performance and speed performance of P_{CU} under multi-word constraint with different K ($1 \leqslant K \leqslant 4$). We use $\mathrm{P_{CU}^{K=}}^{i}$ ($1 \leqslant i \leqslant 4$) to denote the $\mathrm{P_{CU}}$ sketch with i-word constraint technique. Obviously, $\mathrm{P_{CU}^{K=1}}$ is equal to the aforementioned $\mathrm{P_{CU}}$. As mentioned in Section IV, for each of $\mathrm{P_{CU}^{K=1}}$, $\mathrm{P_{CU}^{K=2}}$, and $\mathrm{P_{CU}^{K=2}}$, we use one 64-bit Bob Hash; for $\mathrm{P_{CU}^{K=4}}$, we use two 64-bit Bob Hashes.

*1) **Accuracy**:*

AAE: *Our experimental results show that on different IP traces, the AAE of $P_{CU}^{K=1}$ is about 1.36, 1.39, and 1.40 times higher than $P_{CU}^{K=2}$, $P_{CU}^{K=3}$, and $P_{CU}^{K=4}$, respectively.* Figure 28 plots the AAEs of $\mathrm{P_{CU}^{K=}}^{i}$ ($1 \leqslant i \leqslant 4$) on different real IP traces. We observe that changing K from 1 to 2 helps improve the accuracy largely, while changing K from 2 to 3 or from 3 to 4 makes little impact on the accuracy.

Our experimental results show that on skewed datasets, the AAE of $P_{CU}^{K=1}$ is about 1.36, 1.39, and 1.40 times higher than $P_{CU}^{K=2}$, $P_{CU}^{K=3}$, and $P_{CU}^{K=4}$, respectively. Figure 29 plots the AAEs of $\mathrm{P_{CU}^{K=}}^{i}$ ($1 \leqslant i \leqslant 4$) on synthetic datasets with different skewnesses.

ARE: *Our experimental results show that on different IP traces, the ARE of $P_{CU}^{K=1}$ is about 1.32, 1.36, and 1.36 times higher than $P_{CU}^{K=2}$, $P_{CU}^{K=3}$, and $P_{CU}^{K=4}$, respectively.* Figure 30 plots the AREs of $\mathrm{P_{CU}^{K=}}^{i}$ ($1 \leqslant i \leqslant 4$) on different real IP traces.

Our experimental results show that on skewed datasets, the ARE of $P_{CU}^{K=1}$ is about 1.33, 1.35, and 1.35 times higher than $P_{CU}^{K=2}$, $P_{CU}^{K=3}$, and $P_{CU}^{K=4}$, respectively. Figure 31 plots the AREs of $\mathrm{P_{CU}^{K=}}^{i}$ ($1 \leqslant i \leqslant 4$) on synthetic datasets with different skewnesses.

*2) **Speed**:*

Insertion: *Our experimental results show that on different IP traces, $P_{CU}^{K=2}$, $P_{CU}^{K=3}$, and $P_{CU}^{K=4}$ degrade the insertion throughput by about 8.0%, 15.8%, and 41.5%, compared with $P_{CU}^{K=1}$, respectively.* Figure 32 plots the insertion throughput of $\mathrm{P_{CU}^{K=}}^{i}$ ($1 \leqslant i \leqslant 4$) on different real IP traces. We observe that changing K from 1 to 2 or from 2 to 3 degrades the insertion throughput a little, while changing K from 3 to 4 degrades the insertion throughput largely, since $\mathrm{P_{CU}^{K=4}}$ needs two hash functions. Note that the insertion throughput of $\mathrm{P_{CU}^{K=4}}$ is still larger than pure CU sketch.

Our experimental results show that on skewed datasets, $P_{CU}^{K=2}$, $P_{CU}^{K=3}$, and $P_{CU}^{K=4}$ degrade the insertion throughput by about 10.6%, 19.4%, and 38.9%, compared with $P_{CU}^{K=1}$, respectively. Figure 33 plots the insertion throughput of $\mathrm{P_{CU}^{K=}}^{i}$ ($1 \leqslant i \leqslant 4$) on synthetic datasets with different skewnesses.

Query: *Our experimental results show that on different IP traces, $P_{CU}^{K=2}$, $P_{CU}^{K=3}$, and $P_{CU}^{K=4}$ degrade the query throughput by about 11.8%, 18.6%, and 37.7%, compared with*

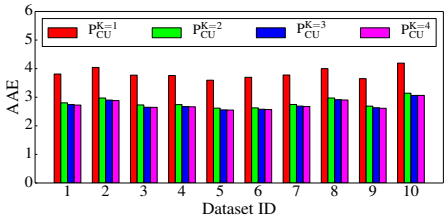


Fig. 28. AAE comparison on different real IP traces.

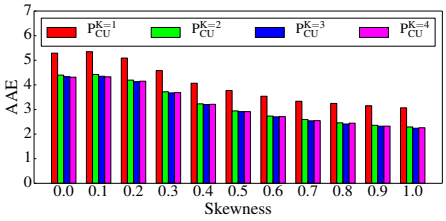


Fig. 29. AAE comparison on synthetic datasets with different skewnesses.

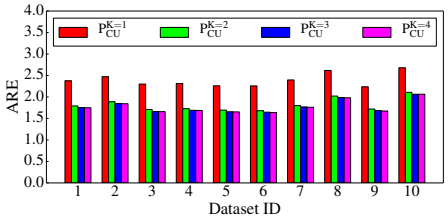


Fig. 30. ARE comparison on different real IP traces.

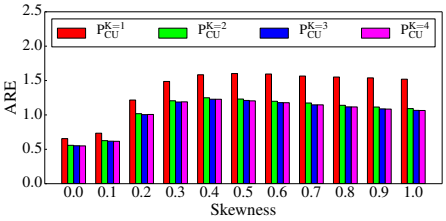


Fig. 31. ARE comparison on synthetic datasets with different skewnesses.

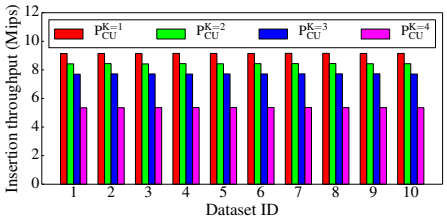


Fig. 32. Comparison of insertion throughput on different real IP traces.

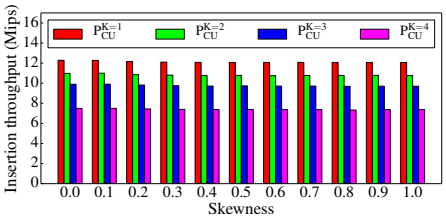


Fig. 33. Comparison of insertion throughput on synthetic datasets with different skewnesses.

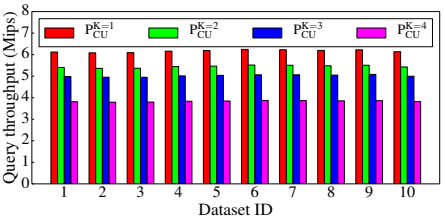


Fig. 34. Comparison of query throughput on different real IP traces.

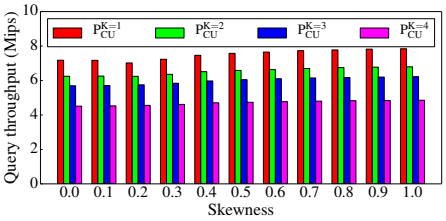


Fig. 35. Comparison of query throughput on synthetic datasets with different skewnesses.

$P_{CU}^{K=1}$, *respectively.* Figure 34 plots the query throughput of $\mathrm{P}_{\mathrm{CU}}^{\mathrm{K}=i}$ $(1 \leqslant i \leqslant 4)$ on different real IP traces. Note that the query throughput of $\mathrm{P}_{\mathrm{CU}}^{\mathrm{K}=4}$ is still larger than pure CU sketch.

Our experimental results show that on skewed datasets, $P_{CU}^{K=2}$, $P_{CU}^{K=3}$, *and* $P_{CU}^{K=4}$ *degrade the query throughput by about* 13.0%, 20.8%, *and* 37.2%, *compared with* $P_{CU}^{K=1}$, *respectively.* Figure 35 plots the query throughput of $\mathrm{P}_{\mathrm{CU}}^{\mathrm{K}=i}$ $(1 \leqslant i \leqslant 4)$ on synthetic datasets with different skewnesses.

VII. Conclusion

Sketches have been applied to various fields. In this paper, we propose a sketch framework - the Pyramid sketch, to significantly improve the update speed and accuracy. We applied our framework to four typical sketches: the CM, CU, Count, and Augmented sketch. Experimental results show that our framework significantly improves both accuracy and speed. We believe our framework can be applied to many more sketches. All related source codes are released at Github [2].

References

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

[2] Source codes of Pyramid sketch and related sketches. https://github.com/zhouyangpkuer/Pyramid_Sketch_Framework.

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

[4] George Kollios, John W Byers, and et al. Robust aggregation in sensor networks. *IEEE Data Eng. Bull.*, 28(1):26–32, 2005.

[5] Peixiang Zhao, Charu C Aggarwal, and Min Wang. gsketch: on query estimation in graph streams. *Proc. VLDB*, 2011.

[6] Amit Goyal, Daume, Hal Iii, and Graham Cormode. Sketch algorithms for estimating point queries in nlp. In *Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, 2012.

[7] Graham Cormode, Theodore Johnson, Flip Korn, S Muthukrishnan, Oliver Spatscheck, and Divesh Srivastava. Holistic udafs at streaming speeds. In *Proc. ACM SIGMOD*, pages 35–46. ACM, 2004.

[8] Pratanu Roy, Arijit Khan, and Gustavo Alonso. Augmented sketch: Faster and more accurate stream processing.

[9] 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.

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

[11] Charu C Aggarwal and S Yu Philip. On classification of high-cardinality data streams. In *SDM*, volume 10, pages 802–813. SIAM, 2010.

[12] Aiyou Chen, Yu Jin, Jin Cao, and Li Erran Li. Tracking long duration flows in network traffic. In *Proc. IEEE INFOCOM*, 2010.

[13] Zaoxing Liu, Antonis Manousis, and et al. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proc. ACM SIGCOMM*, 2016.

[14] Dina Thomas, Rajesh Bordawekar, and et al. On efficient query processing of stream counts on the cell processor. In *Proc. IEEE ICDE*, 2009.

[15] Anna C Gilbert, Martin J Strauss, Joel A Tropp, and Roman Vershynin. One sketch for all: fast algorithms for compressed sensing. In *Proc. ACM STOC*, 2007.

[16] David Talbot and Miles Osborne. Smoothed bloom filter language models: Tera-scale lms on the cheap. In *EMNLP-CoNLL*, pages 468–476, 2007.

[17] Benjamin Van Durme and Ashwin Lall. Probabilistic counting with randomized storage. In *IJCAI*, pages 1574–1579, 2009.

[18] Neoklis Polyzotis, Minos Garofalakis, and Yannis Ioannidis. Approximate xml query answers. In *Proc. ACM SIGMOD*, 2004.

[19] Wenjie Zhang, Xuemin Lin, Ying Zhang, Ke Zhu, and Gaoping Zhu. Efficient probabilistic supergraph search. *IEEE Transactions on Knowledge & Data Engineering*, 28(4):965–978, 2016.

[20] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. Efficient subgraph matching by postponing cartesian products. pages 1199–1214, 2016.

[21] Zhewei Wei, Xuancheng Liu, Feifei Li, Shuo Shang, Xiaoyong Du, and Ji Rong Wen. Matrix sketching over sliding windows. In *International Conference on Management of Data*, pages 1465–1480, 2016.

[22] Ke Yi, Feifei Li, Graham Cormode, Marios Hadjieleftheriou, George Kollios, and Divesh Srivastava. Small synopses for group-by query verification on outsourced data streams. *Acm Transactions on Database Systems*, 34(3):1–42, 2009.

[23] Chengyuan Zhang, Ying Zhang, Wenjie Zhang, and Xuemin Lin. Inverted linear quadtree: Efficient top k spatial keyword search. In *IEEE International Conference on Data Engineering*, pages 901–912, 2013.

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

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

[26] Tong Yang, Lingtong Liu, Yibo Yan, Muhammad Shahzad, Yulong Shen, Xiaoming Li, Bin Cui, and Gaogang Xie. Sf-sketch: A fast, accurate, and memory efficient data structure to store frequencies of data items. In *Proc. IEEE ICDE*, 2017.

[27] Zhou Yang, Liu Peng, Jin Hao, Yang Tong, Dang Shoujiang, and Li Xiaoming. One memory access sketh: a more accurate and faster sketch for per-flow measurement. IEEE Globecom, 2017.

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

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

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

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

[32] Tong Yang, Alex X Liu, Muhammad Shahzad, Yuankun Zhong, Qiaobin Fu, Zi Li, Gaogang Xie, and Xiaoming Li. A shifting bloom filter framework for set queries. *Proc. VLDB*, 9(5):408–419, 2016.

[33] Tong Yang, Alex X Liu, Muhammad Shahzad, Dongsheng Yang, Qiaobin Fu, Gaogang Xie, and Xiaoming Li. A shifting framework for set queries. *IEEE/ACM ToN*, 2017.

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

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

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

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

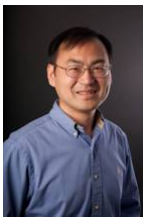
[38] Alex Rousskov and Duane Wessels. High-performance benchmarking with web polygraph. *Software: Practice and Experience*, 34(2):187–211, 2004.

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

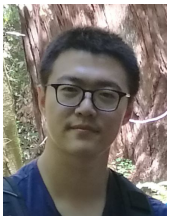
Tong Yang received his Ph.D. degree in Computer Science from Tsinghua University in 2013. He visited Institute of Computing Technology, Chinese Academy of Sciences (CAS) China from 2013.7 to 2014.7. Now he is an assistant professor in Computer Science Department, Peking University. His research interests include IP lookups, Bloom filters, sketches and KV stores.

Yang Zhou is a senior student at Peking University, advised by Tong Yang. He is broadly interested in networking algorithm and system. He has published a few papers about network measurement and approximate data stream processing.

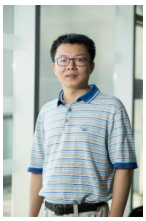
Haowei Zhang is an undergraduate at Peking University. His mentor is Tong Yang. He is interested in the filed of filters and sketches.



Shigang Chen (A'03-M'04-SM'12-F'16) received the B.S. degree from the University of Science and Technology of China in 1993, and the M.S. and Ph.D. degrees from the University of Illinois at UrbanaChampaign in 1996 and 1999, respectively, all in computer science. He served on the technical advisory board for Protego Networks from 2002 to 2003. He was with Cisco Systems for three years. Since 2002, he has been a Professor with the Department of Computer and Information Science and Engineering, University of Florida. He has authored over 100 peer-reviewed journal/conference papers. He holds 11 U.S. patents. His research interests include computer networks, internet security, wireless communications, and distributed computing. He received the IEEE Communications Society Best Tutorial Paper Award in 1999 and the NSF CAREER Award in 2007. He is an Associate Editor of the IEEE/ACM TRANSACTIONS ON NETWORKING, *Computer Networks*, and the IEEE TRANSACTIONS ON VEHICULAR TECHNOLOGY. He served in the Steering Committee of the IEEE IWQoS from 2010 to 2013.



Hao Jin is a junior student at Peking University. He is broadly interested in networking algorithm and machine learning theory.



Bin Cui is a professor in the School of EECS and director of Institute of Network Computing and Information Systems, Peking University. His research interests include database system architectures, query and index techniques, big data management, and mining. He is currently on the editorial boards of the VLDB Journal, Distributed and Parallel Databases Journal, Information Systems, and Frontier of Computer Science. He is the winner of the Microsoft Young Professorship award and the CCF Young Scientist award.

Xiaoming Li is a professor in computer science and technology and the director of Institute of Network Computing and Information Systems (NCIS) at Peking University, China. His current research interest is in search engine and web mining. He led the effort of developing a Chinese search engine (Tianwang) since 1999, and is the founder of the Chinese web archive (Web InfoMall).

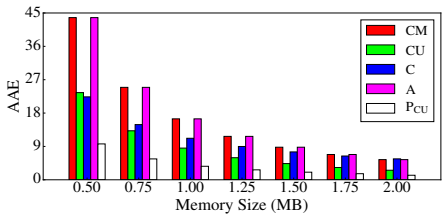


Fig. 36. AAE comparison with different memory sizes on one real IP trace.

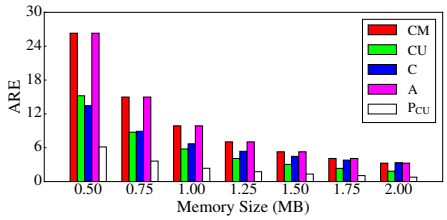


Fig. 37. ARE comparison with different memory sizes on one real IP trace.

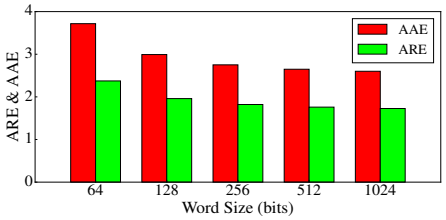


Fig. 38. AAE & ARE of P_{CU} with different word sizes on one real IP trace.

VIII. Appendix

A. Frequency Distributions of datasets

1) Real IP-Trace Streams: The flow characteristics in the datasets mentioned in SectionVI-A are similar. Take the first dataset as an example. The flow size ranges from 1 to 25,429 with a mean of 9.27 and a variance of 11,361. The flow size of each dataset ranges from 1 to around 25,000 with a mean of around 9.30, and the variance is around 11,000. *Note that 41.8% flows only have one packet.* In these datasets, the maximum flows have $15,427 \sim 31,808$ packets. The length of items in each experimental trace is $22 \sim 46$ $bytes$.

2) Web Page Dataset: For this web page dataset with 16M items, it contains 598,688 distinct items. The frequency ranges from 1 to 75,457, with a mean of 29.29 and a variance of 202,826.

3) Synthetic Datasets: The maximum frequency of items in each dataset is $28 \sim 21423$. The length of items in each dataset is all 13 $bytes$.

B. Proof of No Under-estimation Error

According to the implementation of Pyramid sketch framework, there will always be a layer where the overflow will not occur in practice. Through following steps, we will prove that if no overflow occurs at layer L_n $(1 \leqslant n \leqslant \lambda)$, the P_{CM} sketch has no under-estimation error.

Step 1: Suppose $n = 1$. According to the implementation of our framework, layer L_1 performs exactly the same as the CM sketch if no overflow occurs at Layer L_1. Since the CM sketch has no under-estimation error [9], the P_{CM} sketch also has no under-estimation in this case.

Step 2: Suppose $n = k - 1$ $(k \geqslant 2)$. The P_{CM} sketch has no under-estimation error.

Step 3: Suppose $n = k$. For one certain counter $L_{k-1}[j]$ at layer L_{k-1}, its actual value is determined by $L_{k-1}[j].count$, and the number of overflows which is recorded in the counting part of its parent counter $L_k[j']$. However, the sibling counter of $L_{k-1}[j]$ may also contribute to $L_k[j'].count$. Thus, the actual number of overflows of $L_{k-1}[j]$ is no larger than $L_k[j'].count$. In this way, if we recover the value of $L_{k-1}[j]$ according to the $L_k[j'].count$, the value we get will be no less than the actual value of $L_{k-1}[j]$ before the overflow. Assuming the counting part of counters at layer L_{k-1} to be large enough, the P_{CM} sketch will not overflow at layer L_{k-1}, with the counting part of $L_{k-1}[j]$ no less than the actual value. Therefore, in this case, the P_{CM} sketch has no under-estimation error.

According to the above three steps, it can be concluded that the P_{CM} sketch has no under-estimation error

C. Experiments on Memory Size and Word Size

We can obverse that P_{CU} achieves the best accuracy, therefore, we mainly compare P_{CU} with the four typical sketches by varying the memory size or word size.

Our experimental results show that on different memory sizes, the AAEs of the CM, CU, C, and A sketch are 4.52, 2.43, 2.31 and 4.52 times higher than the P_{CU} sketch, respectively. Figure 36 plots the AAEs of different sketches on different memory sizes increasing from 0.50MB to 2.00MB with a step of 0.25MB, on one real IP trace.

Our experimental results show that on different memory sizes, the AREs of the CM, CU, C, and A sketch are 4.28, 2.48, 2.19 and 4.28 times higher than the P_{CU} sketch, respectively. Figure 37 plots the AREs of different sketches on different memory sizes increasing from 0.50MB to 2.00MB with a step of 0.25MB, on one real IP trace.

Our experimental results show that as the word size increases from 64 bits to 1024 bits, the AAE of the P_{CU} sketch decreases from 3.72 to 2.60 and the ARE of the P_{CU} sketch decreases from 2.37 to 1.73. Figure 38 plots the AREs and AAEs of the P_{CU} sketch on different word sizes multiplying from 64 bits to 1024 bits by 2, on one real IP trace.