Advancing Sketch-Based Network Measurement: A General, Fine-Grained, Bit-Adaptive Sliding Window Framework

Kejun Guo†, Fuliang Li†, Jiaxing Shen‡, Xingwei Wang†✉
†Northeastern University, Shenyang 110819, China, ‡Lingnan University, Hong Kong, China
Email: 2301867@stu.neu.edu.cn, lifuliang@mail.neu.edu.cn, jiaxingshen@LN.edu.hk, wangxw@mail.neu.edu.cn

***Abstract*—Network measurement plays a critical role in numerous network applications that rely on fundamental flow processing tasks such as frequency estimation, heavy hitter detection, and distribution estimation. Sketch has emerged as an efficient approach for network measurement due to its low overhead. However, most sketch-based solutions target static windows while enabling sliding window-based measurement remains an open challenge. This paper introduces two novel general frameworks applicable to diverse sketch models for sliding window-based network measurement: a traditional sliding window framework and a fine-grained flow-level framework. The traditional framework divides the window into parts and uses centralized flushing to remove expired parts. The flow-level framework tracks timestamps to maintain exact flow characteristics over one period, preventing truncation. To optimize memory usage, a bit-wise adaptive allocation algorithm allows dynamic borrowing of unused counter bits. The frameworks are evaluated on sketches for different flow processing tasks. Results show the frameworks are widely generalizable, reduce error substantially compared to existing approaches, and provide more efficient memory usage.**

***Index Terms*—Sketch, approximate estimate, sliding window.**

I. Introduction

A. Background and Motivations

Network measurement is the basis of various network applications, including traffic engineering, anomaly detection and congestion control [1], [2]. They rely on fundamental flow processing tasks such as frequency estimation, heavy hitter detection, and distribution estimation. Existing sketch-based solutions have been widely used due to their ability to achieve high accuracy with low memory usage. Due to low memory usage, sketch can be stored in programmable switch or cache.

In practical applications, people tend to focus on the most recent flows. For example, in intrusion detection systems, it is more concerned about recent attacks. Therefore, sliding window-based sketch solutions are proposed for this purpose. However, for high-speed and massive flows, this presents a challenge as the sliding window requires maintaining the flow state to promptly clear outdated flows. There have been a few sliding window algorithms for sketches, like Sliding sketch [3], S-ACE [4], and so on [5]–[12]. However, they are limited to the following three aspects:

- **Poor generality:** The first is the limitation of the measurement task. ECM [7] and SWCM [8] are based on CM sketch [13] only for frequency estimation and frequency distribution estimation. WCSS [9] and CELL [10] focus on heavy hitter detection. The second is the limitation caused by the framework itself. Although Sliding sketch [3] and Hopping sketch [6] claim that they are general frameworks, we find that they are only general to sketches with k-hash model. For single hash and complex sketches (Hashpipe [14], Elastic sketch [15], and so on), they are no longer general. We will use an example to explain the problem of Sliding sketch in detail in Section III-C.
- **Coarse granularity:** We refer to the sliding windows in those previous papers as traditional sliding windows. Furthermore, the traditional sliding window is coarse-grained. As shown in Fig. 1, with four seconds as a period, the query is executed at the tenth second. The traditional sliding window is to obtain the flow characteristics between the sixth and tenth seconds. But only the flows that start before the sixth second and end after the tenth second has gone through a complete period, and the rest of the flows have not gone through a complete period. In other words, traditional sliding windows only have good estimates for persistent flows ($Flow\_2$), while they have poor estimate for ephemeral flow ($Flow\_1$). However, in existing applications, ephemeral flows are always in the majority. The flow-level sliding window has good estimates for both persistent and ephemeral flows. We emphasize that flow-level sliding window makes sense. For example, in recommendation systems, we make recommendations based on each user's behavior in the recent period, terminating at the last activity of each user rather than the current query time.
- **Poor memory utilization:** Due to the need to clear outdated flows, the existing sliding window algorithm often requires additional buckets, like Sliding sketch. But the existing on-chip memory resources are precious. Although there are solutions like SEAD [16], SALSA [17], and so on [18]–[20] that perform compression optimizations on counters, there

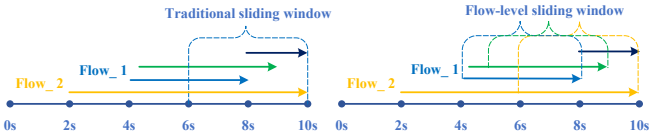


Fig. 1: Two types of the sliding window.

are various limitations. For example, in SEAD, large count values cannot utilize unused bits of small count values. In SALSA, the merging of small count values with large count values may introduce errors.

B. Our Proposed Solution and Contribution

In order to overcome the above shortcomings, we propose the two general sliding window framework and the bit-wise adaptive allocation algorithm respectively. We apply our framework to evaluate on five sketches of three tasks.

Contribution I: We propose two sliding window frameworks. To address the issue of poor generality in the traditional sliding window, we propose a noval traditional sliding window framework, which is simple and effective to adapt to most sketches. Our framework is similar to the Sliding sketch [3]. The difference is that we adopt a solution of centralized refreshing instead of scanning. K-hash sketches, single-hash sketches and complex sketches all can be extended to sliding window models by our framework.

To address the issue of coarse granularity, we propose the fine-grained flow-level sliding window framework, which can obtain the characteristics of the flow over one complete period at any given time. On each insertion, we clear outdated flows based on timestamps to record the characteristics of the flow over one complete period.

Contribution II: We propose the bit-wise adaptive allocation algorithm for counters. We base this on two facts: 1) Since network traffic is skewed, most of the counters in sketch have small count values. 2) Despite the presence of hash collisions, we realize that the probability of several consecutive buckets containing elephant flows is low. Therefore, we let multiple counters share a register array, which can be adaptively occupied between counters according to their count values. Our proposed bit-wise adaptive allocation algorithm can minimize the error by letting large counts take advantage of unused bits from small counts. Our solution preserves the count values as losslessly as possible.

Contribution III: We apply our framework to evaluate on five sketches of three tasks. The experimental results show that our framework is general, fine-grained, and has a high memory utilization. And the accuracy of existing sketch without sliding window support is much higher than other sliding window algorithms after using our framework.

II. RELATED WORK

In this section, we present various sketches that can be used in our framework, as well as state-of-the-art probabilistic data structures for sliding window and approximate estimate.

A. Sketch

Sketch is a kind of probabilistic data structure. At the cost of allowing a certain amount of error, sketch achieves high accuracy with low memory usage. Conventional sketches support fixed window queries and do not support sliding window model. The existing sketch supports a variety of flow processing tasks. In this paper, we focus on the three most fundamental tasks: frequency estimation, heavy hitter detection and frequency distribution estimation. Frequency estimation counts the number of packets in a flow. Typical sketches for frequency estimation include CM sketch [13], Elastic sketch [15], HeavyGuardian [21] and so on. Heavy hitter detection identifies flows exceeding a frequency threshold. Sketches that support heavy hitter tasks include Hashpipe [14], Heavykeeper [22] and so on [23]. Frequency distribution estimation obtains the distribution characteristics of all flow frequencies. Sketches that support frequency distribution estimation tasks include Mrac [24], FlowRadar [25] and so on.

B. Sliding Window Model

The existing traditional sliding windows for sketch fall into two main categories. One category is sliding window only for a specific sketch or flow processing task, like S-ACE [4], CELL [10] and so on [7]–[9]. The other category is the general framework, which is generic for various sketches and flow processing tasks, such as Sliding sketch [3] and Hopping sketch [6] and so on [5]. Currently, the most typical sliding window solution is Sliding sketch, which mainly makes use of the characteristics of k-hash model. Sliding sketch uses a scanning pointer to go through sketch one bucket by one bucket repeatedly to ensure that there is always a mapped bucket that records the frequency of queried flows in a slice very close to the sliding window.

C. Optimizations on Counters

Most approximate estimation solutions improve memory utilization by exploiting the skewed characteristics of network traffic, like SEAD [16], SALSA [17], BitSense [18] and so on [19], [20]. SALSA merges overflowing ones with their neighbors to represent larger numbers. We realize that merging neighbors with higher counts with neighbors with lower counts may cause unacceptable errors in single hash model sketch. Under the premise of simultaneously supporting both increase and decrease, the state-of-the-art solution is SEAD. Inspired by floating-point number representation, SEAD uses some sign bits to adjust the magnitude of the counters, enabling the counters to represent larger values. However, it fails to utilize unused bits in other counters.

III. SLIDING SKETCHES

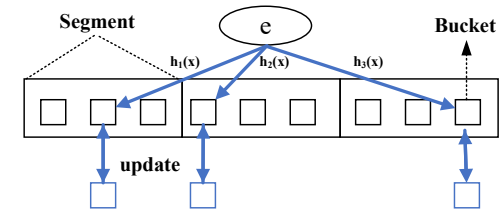
In this section, we propose two sliding window frameworks: the traditional sliding window framework and the flow-level sliding window framework.

A. Definitions of Sliding Window

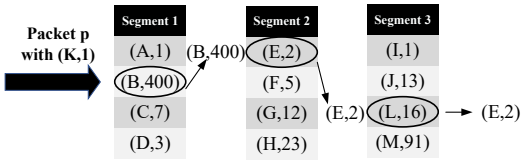
We give the formal definitions of the traditional sliding window and the flow-level sliding window respectively.

Definition 1. ***Traditional sliding window:*** *Traditional sliding window with length N means the union of the packets that arrive in the last N time units.*

Definition 2. ***Flow-level sliding window:*** *Flow-level sliding window with length N means the union of the packets that*



(a) K-hash model sketch.



(b) Non-k-hash model sketch.

Fig. 2: Two categories of sketch models.

arrive in the last N time units for the flow e, which takes the last time that the flow e appears as the end time.

B. Sketch Model

Sketches are divided into two categories: k-hash model and non-k-hash model. The details are as follows:

1) **K-hash model:** As shown in Fig. 2a, the k-hash model sketches consist of k segments, where each segment contains multiple elements and is associated with a hash function. The elements within a segment are called buckets, which could be bits, counters, or key-value pairs depending on specific sketches. When a flow is inserted, it is updated in k segments through k hash functions. Insertion is performed independently for each mapped bucket. The updates of each segment are mutually independent, and each segment can be considered as a copy of the traffic statistics characteristics.

2) **Non-k-hash model:** Non-k-hash model sketches may be the variety of complex sketches. The updates of each segment are not independent. And non-k-hash model sketches do not store multiple copies like k-hash model, but only store a certain flow in one or some bucket. As shown in Fig. 2b, we take Hashpipe [14] as an example. When inserting a flow, in the first segment, key-value pairs in the mapping bucket are directly replaced. In the subsequent segments, if the flow key matches the stored key in the mapping bucket, we increase the count value in the mapping bucket. Otherwise, we evict key-value pairs with the smaller count value towards the next segment and repeat the same operation. It is evident that the key-value pair corresponding to each flow is not stored in all segments.

C. Traditional Sliding Window Framework

In the non-k-hash model, let's discuss the problem in Sliding sketch [3]. First, non-k-hash model sketches do not store multiple copies like the k-hash model sketches, which fundamentally undermines the effectiveness of Sliding sketch. This is because Sliding sketch relies on having copies of the interested flows in each of the k segments to ensure that there is always a segment that records queried flows in a slice very close to the sliding window. Second, in the non-k-hash model sketches, there is a problem of duplicate scanning in Sliding sketch. We still take Hashpipe as an example. As shown in Fig. 2b, assuming we have completed the scanning of $Segment$ 1 and are about to scan $(E, 2)$ in $Segment$ 2. At this moment, a packet p arrives, which displaces $(B, 400)$ from $Segment$ 1 to the position of $(E, 2)$ in $Segment$ 2, resulting in an exchange between $(B, 400)$ and $(E, 2)$. In this scenario, Sliding sketch will rescan the previously scanned $(B, 400)$, which will significantly underestimate the count value of B.

Algorithm 1: Insertion in Traditional Sliding Window

```
1 pos = hash(e) mod row;
2 cur_part ← ((T_cur − T_init) / (N/d)) mod (d + 1);
3 if last_part != cur_part then
4 |   last_part ← cur_part;
  |   // Centralized flushing.
5 |   Sketch.clear(all bucket, cur_part);
6 end
  // The mapping bucket position is pos.
7 Sketch.insert(pos, cur_part);
```

Our framework combines the strategies of both Sliding sketch [3] and S-ACE [4]. Unlike Sliding sketch, we adopt a solution of centralized flushing instead of scanning for generality. Our framework is different from S-ACE in two respects. First, S-ACE equally divides period to place a special data structure called virtual aging counter. We emphasize that we will not simply divide the period equally to place the same sketch but only expand the necessary fields. For example, in Hashpipe, we only expand the count field but not the flow key field. It is necessary to accumulate the count as the overall count to perform the insertion. In other words, we maintain only one sketch within the sliding window and do not maintain a separate sketch for each part, which is not explicitly clarified in S-ACE. Second, S-ACE only supports frequency estimation task and a special data structure called virtual aging counter. Our framework enables most sketches to support sliding window to support almost all tasks.

Inspired by above solutions, we propose our solution:

Initialization: Suppose a period is N seconds. In order to obtain the flow characteristics of the most recent period, we divide a period into d parts. Every part is for $\frac{N}{d}$ seconds. We always maintains the last $d+1$ parts. We use row to represent the number of buckets. We use cur_part and $last\_part$ to represent the current and previous part of the bucket, and initialize cur_part and $last\_part$ to zero. We use T_{cur} to represent the current system time and T_{init} to represent the time when the sketch starts to execute the task.

Insertion: For the incoming flow e, we treat the $d + 1$ parts as a large bucket and perform the normal sketch insert operation. As shown in Alg. 1, we calculate the position of the cur_part and execute the insertion.

Case 1: If cur_part is equal to $last\_part$, the flow is still located in this part. We just perform the normal sketch insert

Algorithm 2: Query in Traditional Sliding Window

$pos = hash(e) \bmod row,\ s \leftarrow \frac{(T_{cur} - T_{init}) \bmod \frac{N}{d}}{\frac{N}{d}}$;
$cur\_part \leftarrow \frac{(T_{cur} - T_{init})}{\frac{N}{d}} \bmod (d+1),\ ans \leftarrow 0$;
for $j \in range\,[0, d]$ **do**
  $ans\ +=\ bucket[pos][j]$;
end
$ans\ -=\ s \times bucket[pos][(cur\_part+1) \bmod (d+1)]$;
return ans;

operation.

Case 2: If cur_part is not equal to $last\_part$, the $last\_part$ has been completely out of a period. The cur_part of all buckets in sketch is cleared and the value of $last\_part$ is updated to cur_part. Finally, we just perform the normal sketch insert operation.

Query: As shown in Alg. 2, we calculate cur_part and fit parameter s. The entire sketch only needs to calculate cur_part and s once. For query at integer multiples of $\frac{N}{d}$, we support perfect query by summing the latest d parts. For queries at any time, we support the time-unbiased query.

In summary, we mainly adopt a solution of centralized flushing. In the traditional sliding window, it is general for both k-hash model and non-k-hash model sketches. Simultaneously, for query at integer multiples of $\frac{N}{d}$, we support perfect query. For query at any time point, we support the time-unbiased query.

D. Flow-Level Sliding Window Framework

In addition to traditional sliding windows, in some practical scenarios, people expect to get the statistics of the flows through a complete period. The main problem we need to solve is how to maintain a period of state for every flow individually.

Baseline: We introduce the baseline that implements the flow-level sliding window, resembling our traditional sliding window framework. The difference is that we need to maintain $2d + 1$ parts in a bucket. For flow e, the $2d + 1$ parts within the bucket can be represented as $bucket[hash(e) \bmod row][0 : 2d]$. The $bucket[hash(e) \bmod row][(cur\_part - d)\ mod\ (2d + 1) : cur\_part]$ within the bucket is the statistics of the current period, and the remaining d parts are the statistics of the previous period. If there exists a part p in $bucket[hash(e) \bmod row][(cur\_part - d)\ mod\ (2d + 1) : cur\_part]$, and both $bucket[hash(e) \bmod row][p]$ and $bucket[hash(e) \bmod row][(p-d)\ mod\ (2d+1)]$ are non-zero, we consider $bucket[hash(e) \bmod row][(p-d)\ mod\ (2d+1) : p]$ as statistics for a period of flow e.

Obviously, baseline is impractical. On one hand, baseline exhibits excessively high memory usage. On the other hand, flows that do not reach one period may be misidentified. For example, assume that the flow e starts from the end of $bucket[hash(e) \bmod row][(p - d)\ mod\ (2d + 1)]$ and ends at the beginning of $bucket[hash(e) \bmod row][p]$. Even if the flow has not completed a full period, it may still be erroneously identified.

Algorithm 3: Insertion in Flow-Level Sliding Window

1 $pos = hash(e) \bmod row$;
2 $cur\_part \leftarrow \frac{(T_{cur} - T_{start})}{\frac{N}{d}} \bmod (d+1)$;
3 **if** $T_{final} < T_{cur} - N$ *and* $T_{final}! = 0$ **then**
4 $Sketch.clear(pos, all\ part)$;
5 $T_{start}, T_{final} \leftarrow 0$;
6 **end**
7 **if** $T_{final} == 0$ **then**
8 $Sketch.insert(pos, 0)$;
9 $T_{start}, T_{final} \leftarrow T_{cur}$;
10 **end**
11 **else if** $T_{cur} - T_{final} < \frac{N}{d} \times (d+1)$ **then**
12 $Sketch.insert(pos, cur\_part)$;
13 $T_{final} \leftarrow T_{cur}$;
14 **end**
15 **else**
16 $Sketch.clear(pos, [0 : cur\_part])$;
17 $Sketch.insert(pos, cur\_part)$;
18 $temp[0 : d] \leftarrow bucket[pos][0 : d]$;
19 **for** $j \in range\,[0, d]$ **do**
20 $bucket[pos][d - j] \leftarrow$
  $temp[(cur\_part - j + d + 1) \bmod (d+1)]$;
21 **end**
22 $T_{start} += (cur\_part + 1) \times \frac{N}{d}$;
23 $T_{final} \leftarrow T_{cur}$;
24 **end**

To solve the above problems, we propose our solution.

Initialization: We divide a period into d parts. Every part is for $\frac{N}{d}$ seconds. We always maintains the last $d + 1$ parts. We use row to represent the number of buckets. For flow e, the $d + 1$ parts within the bucket can be represented as $bucket[hash(e) \bmod row][0 : d]$. At the same time, we maintain two additional timestamps for every bucket: T_{start} and T_{final}, which represent the system time of the arrival of the first packet and the arrival of the last packet in this bucket respectively. We initialize them to zero. We use T_{cur} to represent the current system time.

Insertion: For the incoming flow e, we perform insertion based on Alg. 3.

Case 1 (line 3-6): When $T_{final} < T_{cur} - N$, the flow recorded in the current bucket is outdated. We clean the bucket.

Case 2 (line 7-10): When the timestamp is zero, it indicates that the flow e has arrived at the bucket for the first time. We insert it into the first part and assign the current system time T_{cur} to T_{start}.

Case 3 (line 11-14): When $T_{cur} - T_{start} < \frac{N}{d} \times (d+1)$, the bucket is not yet full. First, we calculate cur_part to determine which part the flow e should be inserted into the bucket. Then, we insert the flow e into that part of the bucket. Finally, we update T_{final} to the current system time T_{cur}.

Case 4 (line 15-24): Otherwise, it indicates that the flow in the current bucket has exceeded one period. Our purpose is to maintain the flow characteristics within the most recent

Algorithm 4: Query in Flow-Level Sliding Window

$ans \leftarrow 0, pos = hash(e) \bmod row$;
for $j \in range\,[0, d]$ **do**
  $ans+ = bucket[pos][j]$;
end
if $T_{final} - T_{start} \geq N$ **then**
  $s \leftarrow \frac{T_{final} - T_{cur} - N}{\frac{N}{d}}$;
  $ans- = s \times bucket[pos][0]$;
end
return ans;

period. First, we clear several outdated parts before insertion. Then, we shift the parts in the bucket and arrange them in ascending order of time. Finally, we update T_{start} to $T_{start} + (cur\_part + 1) \times \frac{N}{d}$, and update T_{final} to the current time T_{cur}. This ensures that the flow that reaches one period can get its characteristic within the most recent period.

Query: As shown in Alg. 4, we accumulate all the count values in the bucket. To ensure time-unbiased queries, we calculate fit parameter s.

In summary, we achieve the fine-grained flow-level sliding window that supports a complete period of query at any time. Even for flows that do not reach a period, our framework can give the count value and the elapsed time of the flow. It is general for most sketches, extending them to flow-level sliding window. We only need to execute a query once per period to obtain the statistics of the most recent complete period for each flow. At the same time, our framework can incorporate two additional features into existing sketches: providing the time range of the flows and flow rate. Since we have added timestamps, it becomes straightforward to obtain the time range of the flow and calculate the flow rate.

IV. Bit-wise Adaptive Allocation Algorithm

In this section, we propose a bit-wise adaptive allocation algorithm to sketch containing counters, which dynamically allocates the number of bits based on the count values.

A. Algorithm and Operations

It is well known that network traffic is highly skewed. To ensure that counters do not overflow, the number of bits occupied by counters must be set according to the number of bits occupied by the maximum count value, which results in wastage of bits for small count values.

Our proposed solution is based on two key facts: 1) Since network traffic is highly skewed, most of the counters in sketch have small count values. 2) Despite the presence of hash collisions, we realize that the probability of several consecutive buckets containing elephant flows is low. Therefore, we allow several counters to share a set of bits, with each counter adaptively using bits based on its count value.

Initialization: In previous sketch, each bit array occupies 32 or 64 bits as a counter. We use n to represent how many counters share a bit array. As shown in the Fig. 3, the array

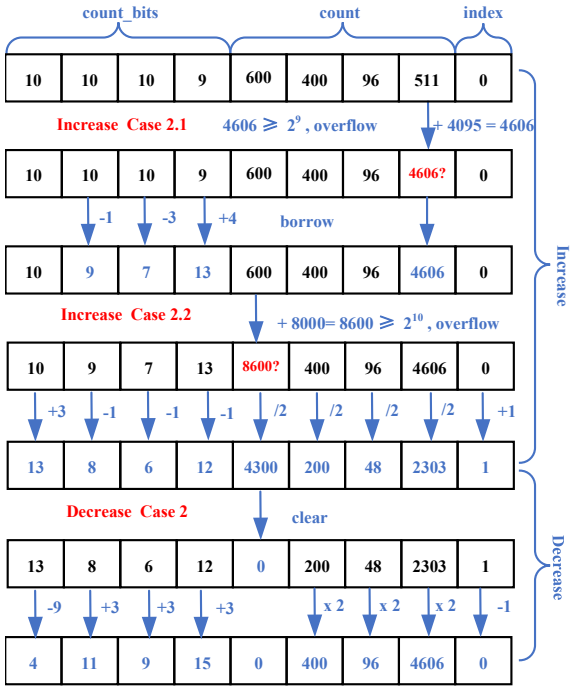


Fig. 3: Example of Bit-wise Adaptive Allocation Algorithm.

consists of a total of 64 bits. Four counters share this array ($n = 4$), which can be read in a single operation on most computers. We use row to represent the number of bit arrays. For a given array of bit arrays $bit\_array[0 : row - 1]$, each bit array is divided into three fields: $count\_bits$, $count$, and $index$. $count\_bits$ denotes the number of bits occupied by the count value. $count$ denotes the count value. $index$ denotes the counting unit. We fix the number of bits occupied by $count\_bits$ and $index$. The counting units represented by $count\_bits$ and $index$ can be user-defined. In this paper, we specify that the counting unit represented by $count\_bits$ is 1, and the counting unit represented by $index$ is a power of 2. Therefore, assuming the maximum count value occupies 32 bits, in Fig. 3, each $count\_bits$ and $index$ occupy 5 bits. Initially, the remaining bits are evenly distributed among $count$.

Increase: For the incoming flow e, we get the value of $index$. We use this value as an exponent of 2 and calculate the result. Then we probabilistically decide whether to accumulate $count$ mapped to by e. If no accumulation occurs, the process ends directly. Otherwise, as shown in Alg. 5:

Case 1 *(line 2-4):* If $count$ mapped to by e does not overflow, $count$ is added directly.

Case 2 *(line 5-37):* Otherwise, it indicates an overflow of $count$ mapped to by e. Firstly, we obtain the remaining bits ($remainder$) of the other $count$ as well as the additional bits needed ($need\_bits$) after the accumulation of $count$ mapped to by e. Then, there are two cases:

- ***Case 2.1*** *(line 6-23):* When $remainder \geq need\_true$, there are still available bits in other $count$. Therefore, we just borrow the remaining bits from the other $count$ and update

Algorithm 5: Increase in Bit-wise Adaptive Allocation

Input: $count\_bits : bit\_array[0 : row-1][0 : n-1]$
$count : bit\_array[0 : row-1][n : 2n-1]$
$index : bit\_array[0 : row-1][2n]$
$pos\_1 \leftarrow [hash(e) \bmod row \times n]$ // n
$pos\_2 \leftarrow hash(e) \bmod n$

1 $temp \leftarrow bit\_array[pos\_1]$;
// Such as $inc = 1$ in CM sketch.
2 **if** $temp[n + pos\_2] + inc < 2^{temp[pos\_2]}$ **then**
3 $temp[n + pos\_2] += inc$;
4 **end**
5 **else**
6 $remainder \leftarrow 0, remain\_bits[0 : n-1] \leftarrow 0$;
7 $flag \leftarrow False$;
8 **for** $j \in range\,[0, n-1]$ **do**
9 **if** $j == pos\_2$ **then** continue ;
10 $remain\_bits[j] \leftarrow$ $temp[j] - \lceil \log_2 temp[n + j] \rceil$;
11 $remainder += remain\_bits[j]$;
12 **end**
13 $need\_bits \leftarrow$ $\lceil \log_2 (temp[n + pos\_2] + inc) \rceil - temp[pos\_2]$;
14 **if** $remainder \geq need\_bits$ **then** $flag \leftarrow True$;
15 $j \leftarrow 0$;
16 **while** $need\_bits > 0$ *and* $j < n$ **do**
17 $min\_bits \leftarrow min(remain\_bits[j], need\_bits)$;
18 $need\_bits -= min\_bits$;
19 $temp[j] -= min\_bits$;
20 $temp[pos\_2] += min\_bits$;
21 $j += 1$;
22 **end**
23 **if** $flag$ **then** $temp[n + pos\_2] += inc$;
24 **else**
25 $index\_add \leftarrow \lceil \frac{need\_bits}{n} \rceil$;
26 $temp[pos\_2] += index\_add \times n$;
27 $temp[2n] += index\_add$;
28 **for** $j \in range\,[0, n-1]$ **do**
29 $temp[j] -= index\_add$;
30 **if** $j == pos\_2$ **then**
31 $temp[j + n] \leftarrow \frac{temp[j+n]+inc}{2^{index\_add}}$;
32 *continue*;
33 **end**
34 $temp[j + n] \leftarrow \frac{temp[j+n]}{2^{index\_add}}$;
35 **end**
36 **end**
37 **end**
38 $bit\_array[pos\_1] \leftarrow temp$;

Algorithm 6: Decrease in Bit-wise Adaptive Allocation

// Such as $dec = 1$ in HeavyKeeper.
1 $temp \leftarrow bit\_array[pos\_1],\ temp[n + pos\_2] -= dec$;
2 $remainder \leftarrow 0, remain\_bits[0 : n-1] \leftarrow 0$;
3 **for** $j \in range\,[0, n-1]$ **do**
4 $remain\_bits[j] \leftarrow temp[j] - \lceil \log_2 temp[n + j] \rceil$;
5 $remainder += remain\_bits[j]$;
6 $temp[j] -= remain\_bits[j]$;
7 **end**
8 **if** $remainder < n$ *or* $temp[2n] == 0$ **then** return;
9 **else**
10 $sub \leftarrow min(\lfloor \frac{remainder}{n} \rfloor, temp[2n])$;
11 $temp[2n] -= sub$;
12 **for** $j \in range\,[0, n-1]$ **do**
13 $temp[j] += sub,\ temp[j + n] \times = 2^{sub}$;
14 **end**
15 $add \leftarrow \lfloor \frac{remainder - n \times sub}{n} \rfloor$;
16 $remainder\_add \leftarrow remainder - n \times (sub + add)$;
17 **for** $j \in range\,[0, n-1]$ **do**
18 $temp[j] += add$;
19 **if** $remainder\_add > 0$ **then**
20 $temp[j] += 1,\ remainder\_add -= 1$;
21 **end**
22 **end**
23 **end**
24 $bit\_array[pos\_1] \leftarrow temp$;

$count\_bits$.

- ***Case 2.2 (line 24-36):*** When $remainder < need\_true$, the available bits of other *count* are still not enough. Firstly, we perform the same steps as in *Case 2.1*. Then, we increment $index$ to enhance the maximum count value that *count* can accommodate. Finally, we subtract the value of $index$ from all $count\_bits$ and update *count*.

Decrease: Similar to **Increase**, we probabilistically decide whether to decrease *count*. If no reduction occurs, the process ends directly. Otherwise, we should determine whether the freed bits allow the value of $index$ to be reduced. We get the number of all available bits ($remainder$) and the value of $index$.

Case 1 (line 1-8): If $remainder < n$ or $index == 0$, it implies that there is no way to decrease the value of $index$.

Case 2 (line 9-23): Otherwise, we can reduce the value of $index$ to improve the accuracy of *count*. Firstly, we calculate sub which represents the maximum value by which $index$ can be decreased. Secondly, we decrement $index$ by sub. Then, we update each $count\_bits$ to $count\_bits + sub$. And we multiply all *count* by 2^{sub}. Finally, we distribute the remaining bits evenly.

Query: We just need to multiply the mapped *count* value by 2^{index}.

The core of our solution lies in bit-wise adaptive allocation. On one hand, it minimizes errors by allowing large count values to utilize unused bits from small count values. On the other hand, our solution strives to preserve all count values as losslessly as possible. Our proposed solution does well to reduce memory usage through the bit-wise adaptive allocation. Taking the maximum count occupying 32 bits as an example, $Memory = 32 \times n$ is transformed into

$Memory = \log_2 32 \times (n+1) + allocated\ bits$. The first half of the expression represents the fixed overhead of our solution, which can be reduced by changing the counting units of $count\_bits$ and $index$ fields. The second half of the expression represents the allocated memory size. People can set $allocated\ bits$ according to the memory situation.

B. Example

As shown in Fig. 3, the counting unit represented by $count\_bits$ is 1, and the counting unit represented by $index$ is a power of 2. Therefore, each $count\_bit$ and $index$ occupy 5 bits. And all $count$ occupy 39 ($64-5\times5=39$) bits together. We can get the corresponding $count$ based on the number of bits recorded in $count\_bits$. We only present key examples.

Case 2.1 in Increase: When $count[3]$ overflows ($511+4095 \geq 2^9$), we check whether there are enough unused bits for $count[3]$ to use. For example, the number of unused bits in $count[1]$ is 1 ($10-\lceil\log_2 400\rceil=1$). $count[3]$ still needs 4 ($\lceil\log_2 4606\rceil-9=4$) bits. There are enough unused bits for $count[3]$ to use. Therefore we perform a borrow operation from other $count$. This operation expands the bit occupancy of $count[3]$, thereby increasing the maximum count value $count[3]$ can accommodate.

Case 2.2 in Increase: When $count[2]$ overflows ($600+8000 \geq 2^{10}$), $count[2]$ still needs 4 ($\lceil\log_2 8600\rceil-10=4$) bits and other $count$ also don't have unused bits ($remainder=0$). Therefore, we increment $index$ by $\lceil\frac{4}{n=4}\rceil$ to enhance the maximum count value that $count$ can accommodate. Then, we update all $count\_bits$ and $count$.

Case 2 in Decrease: When $count[0]$ is cleared to zero, we find that the bits released by $count[0]$ is 13, which are enough to decrease $index$ ($\lfloor\frac{13}{n=4}\rfloor \geq index\_value$). Since a smaller $index$ leads to higher accuracy in $count$, we decrease the value of $index$ while updating $count\_bits$ and $count$.

V. PERFORMANCE EVALUATION

In this section, we apply two types of sliding window models to five sketches: CM sketch [13], Elastic sketch [15], HeavyKeeper [22], Hashpipe [14] and Mrac [24]. We compare them with the state-of-the-art sliding window algorithms under the same memory usage. We also analysis the impact of the number of parts.

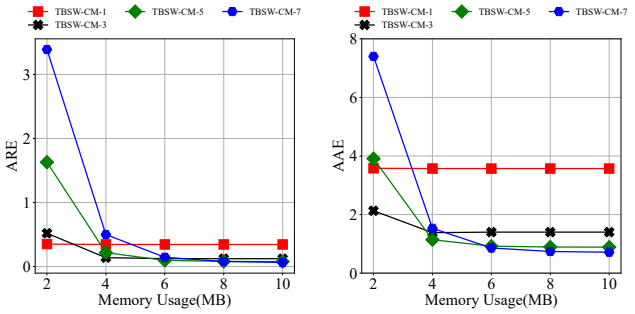
A. Experimental Setup

Traces: We use anonymous IP tracking collected from CAIDA in 2018 [26]. We read 20 million packets and set the length of the sliding window $N=2.5$ million. In the case of aggregation with source IP, there are about $70k$ flows. We carry out experiments in count-based sliding windows. The unit of window size in count-based sliding windows is packet arrival. Each packet arrival can be regarded as a unit time.

Implementation: Our algorithm and all others are implemented in C++. We use k to represent the number of hash functions and d to represent the number of parts of a bucket. We always maintain $d+1$ parts in a bucket. We use n to represent how many counters share a given bit array and set $n=4$. We use an array of length 64 bits, which can be read and written in a single operation on most computers. We don't use the bit-wise adaptive allocation algorithm for the light part of Elastic sketch. For the k-hash model sketches, we set k=10, which refers to the setting of Sliding Sketch [3]. For ECM, SWCM, WCSS, and Lambda, the parameters are set according to the recommendation of the authors. The abbreviations of algorithms in experiments are shown in Table I.

TABLE I: Abbreviations of algorithms in experiments.

Abbreviation	Full name
TBSW	Traditional Sliding Window + Bit-wise Adaptive Allocation
FBSW	Flow-Level Sliding Window + Bit-wise Adaptive Allocation
Baseline	Baseline in Section III-D
SI	Sliding Sketch [3]
ECM	Exponential Count-Min Sketch [7]
SWCM	Splitter Windowed Count-Min Sketch [8]
WCSS	Window Compact Space-Saving [9]
Lambda	λ-sampling Algorithm [27]



(a) ARE. (b) AAE.

Fig. 4: Experiments with the CM sketch on parameters d.

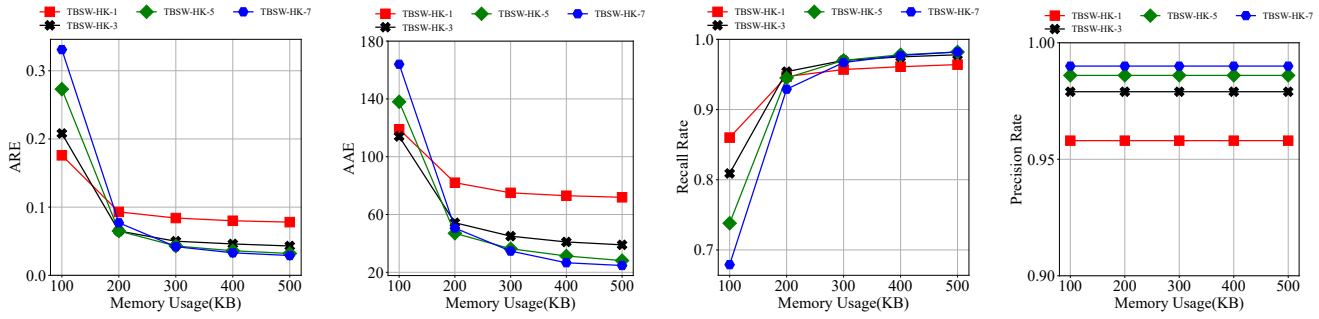
Metrics: We measure the metrics whenever the window slides $\frac{N}{10d}$. We use the average value to represent the experiment result at given parameter setting.

- Average Relative Error (ARE): $\frac{1}{n}\sum_{i=1}^{n}\frac{|f_i-\hat{f}_i|}{f_i}$, where n is the number of flows, and f_i and $\hat{f}_i$ are the actual and estimated flow sizes respectively.
- Average Absolute Error (AAE): $\frac{1}{n}\sum_{i=1}^{n}\left|f_i-\hat{f}_i\right|$, where n is the number of flows, and f_i and $\hat{f}_i$ are the actual and estimated flow sizes respectively.
- Recall Rate: Recall Rate refers to the ratio of reported true instances.
- Precision Rate: Precision Rate refers to the ratio of true instances reported.

B. Experiments on System Parameters

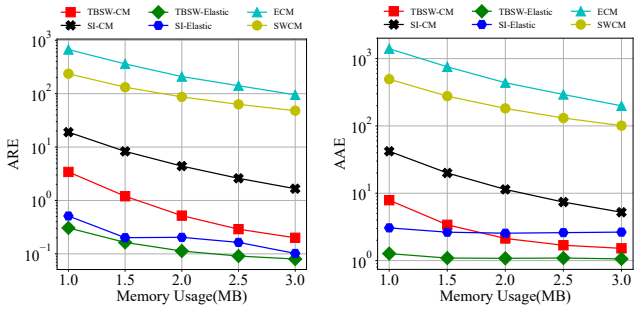
In this section, we analyze the connection between d and memory usage.

TBSW-CM: As shown in Fig. 4, our results show the ARE of TBSW-CM-1 is about 1.5, 4.6 and 9.6 times lower than TBSW-CM-3, TBSW-CM-5 and TBSW-CM-7 respectively when the memory is set to 2MB, but the ARE of TBSW-CM-7 is about 5.4, 1.9 and $1,2$ times lower than TBSW-



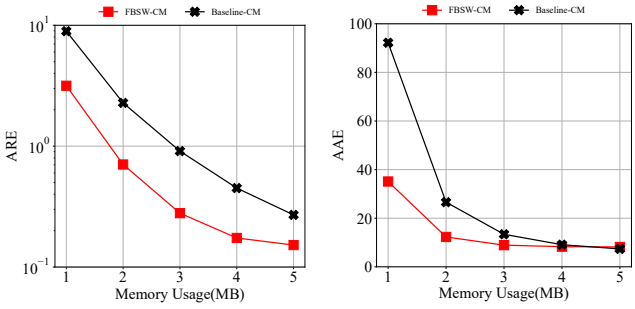
(a) ARE. (b) AAE. (c) Recall Rate. (d) Precision Rate.

Fig. 5: Experiments with the HeavyKeeper on parameters d.



(a) ARE. (b) AAE.

Fig. 6: Frequency estimation on traditional sliding window.



(a) ARE. (b) AAE.

Fig. 7: Frequency estimation on flow-level sliding window.

CM-1, TBSW-CM-3 and TBSW-CM-5 respectively when the memory is set to 10MB.

TBSW-HK: As shown in Fig. 5, our results show that recall rate and precision rate do not differ much when the memory resources exceed 200KB. The ARE of TBSW-HK-1 is the smallest among TBSW-HK-1, TBSW-HK-3, TBSW-HK-5 and TBSW-HK-7 when the memory is set to 100KB, but the ARE of TBSW-HK-1 is the biggest among TBSW-HK-1, TBSW-HK-3, TBSW-HK-5 and TBSW-HK-7 when the memory is set to 500KB.

At small memory, the smaller the value of d is, the higher the accuracy is. However, as the memory increases, the larger the d is, the higher the accuracy is. But the effect caused by d is becoming less and less significant. Therefore, when memory resources are small, we recommend setting the value of d to be small. As memory resources increase, we recommend setting the value of d to be larger. In the following experiments, we set $d = 3$.

C. Experiments on Frequency Estimation

1) Traditional Sliding Window Framework: We compare 6 approaches: TBSW-CM, TBSW-Elastic, SI-CM, SI-Elastic, ECM [7] and SWCM [8]. These include k-hash models and non-k-hash models.

ARE and AAE: As shown in Fig. 6a, in the k-hash models, our results show that the ARE of TBSW-CM is about 8.5, 402 and 167 times lower than SI-CM, ECM and SWCM respectively when the memory is set to 2MB. In the non-k-hash models, the ARE of TBSW-Elastic is about 2, 1850 and 770 times lower than SI-Elastic, ECM and SWCM respectively. As shown in Fig. 6b, in the k-hash models, our results show that the AAE of TBSW-CM is about 5.4, 206 and 85 times lower than SI-CM, ECM and SWCM respectively when the memory is set to 2MB. In the non-k-hash models, the AAE of TBSW-Elastic is about 2.4, 400 and 170 times lower than SI-Elastic, ECM and SWCM respectively.

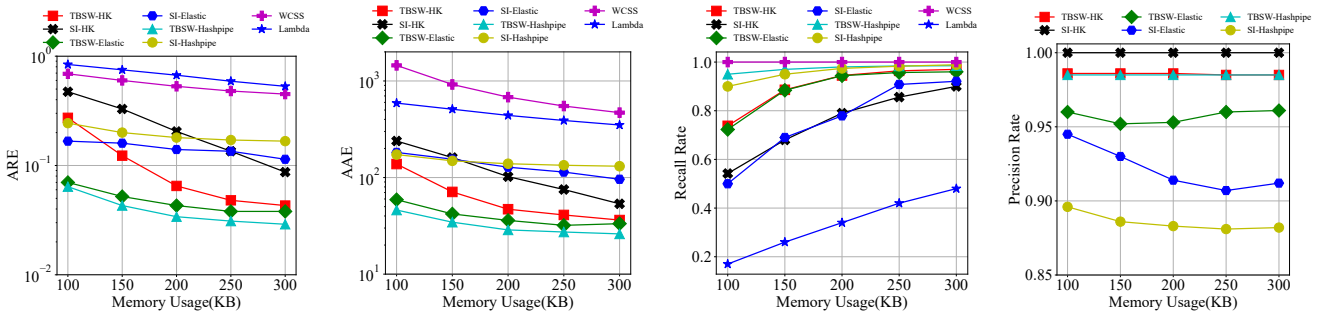
2) Flow-Level Sliding Window Framework: We compare 2 approaches: FBSW-CM and Baseline-CM.

ARE and AAE: As shown in Fig. 7a, our results show that the ARE of FBSW-CM is about 3.3 times lower than Baseline-CM respectively when the memory is set to 3MB. As shown in Fig. 7b, our results show that the AAE of FBSW-CM and Baseline-CM is nearly equal.

D. Experiments on Heavy Hitter Detection

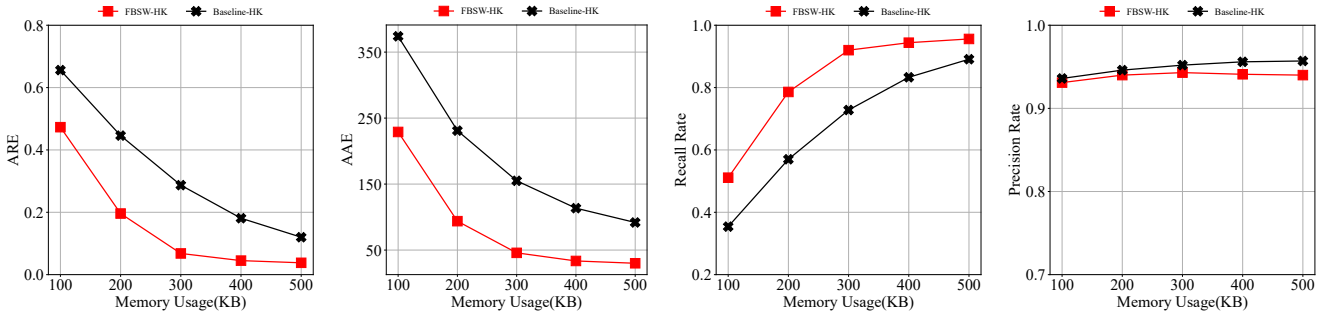
1) Traditional Sliding Window Framework: We compare 8 approaches: TBSW-HK, TBSW-Elastic, TBSW-Hashpipe, SI-HK, SI-Elastic, SI-Hashpipe, WCSS [9] and Lambda [27]. These include k-hash models and non-k-hash models.

ARE and AAE: As shown in Fig. 8a, our results show that the ARE of TBSW-HK is about 3, 10 and 8 times lower than SI-HK, Lambda and WCSS respectively when the memory is



(a) ARE. (b) AAE. (c) Recall Rate. (d) Precision Rate.

Fig. 8: Heavy hitter detection on traditional sliding window.



(a) ARE. (b) AAE. (c) Recall Rate. (d) Precision Rate.

Fig. 9: Heavy hitter detection on flow-level sliding window.

set to 200KB. The ARE of TBSW-Elastic is about 3, 15 and 12 times lower than SI-Elastic, Lambda and WCSS respectively. The ARE of TBSW-Hashpipe is about 5, 20 and 16 times lower than SI-Hashpipe, Lambda and WCSS respectively. As shown in Fig. 8b, our results show that the AAE of TBSW-HK is about 2.2, 9.4 and 14.5 times lower than SI-HK, Lambda and WCSS respectively when the memory is set to 200KB. The AAE of TBSW-Elastic is about 3.5, 12.2 and 18.9 times lower than SI-Elastic, Lambda and WCSS respectively. The AAE of TBSW-Hashpipe is about 4.8, 15.3 and 23.7 times lower than SI-Hashpipe, Lambda and WCSS respectively.

Recall Rate and Precision Rate: As shown in Fig. 8c, our results show that the recall rate of TBSW-HK is about 15% and 60% higher than SI-HK and Lambda respectively when the memory is set to 200KB. The recall rate of TBSW-Elastic is about 15% and 60% higher than SI-Elastic and Lambda respectively. The recall rate of TBSW-Hashpipe is about 65% higher than Lambda respectively. As shown in Fig. 8d, our results show that the precision rate of TBSW-HK, TBSW-Hashpipe and SI-HK is close to 1. The precision rate of TBSW-Hashpipe is about 10% higher than SI-Hashpipe. The precision rate of TBSW-Elastic is always higher than SI-Elastic.

2) Flow-Level Sliding Window Framework: We compare 2 approaches: FBSW-HK and Baseline-HK.

ARE and AAE: As shown in Fig. 9a, our results show that the ARE of FBSW-HK is about 4.2 times lower than Baseline-HK respectively when the memory is set to 300KB. As shown in Fig. 9b, our results show that the AAE of FBSW-HK is about 3.5 times lower than Baseline-HK respectively when the memory is set to 300KB.

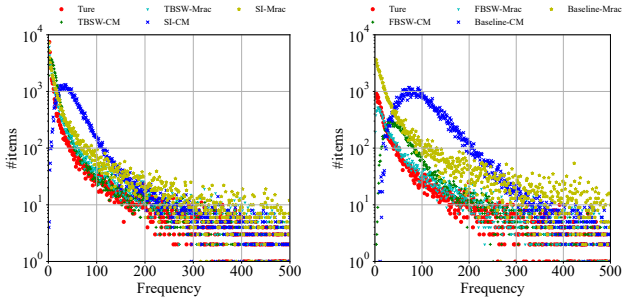
Recall Rate and Precision Rate: As shown in Fig. 9c, our results show that the recall rate of FBSW-HK is about 20% higher than recall rate of Baseline-HK respectively when the memory is set to 300KB. As shown in Fig. 9d, our results show that the precision rate of FBSW-HK and Baseline-HK is approximately equal.

E. Experiments on Real-time Frequency Distribution Estimation

1) Traditional Sliding Window Framework: With the memory set to 1 MB, we compare 4 approaches: TBSW-CM, TBSW-Mrac, SI-CM and SI-Mrac. These include k-hash models and non-k-hash models.

Real-time Frequency Distribution: As shown in Fig. 10a, the estimated frequency distribution of TBSW-CM and TBSW-Mrac is closer to the real frequency distribution, while that of the SI-CM and SI-Mrac suffers from large errors.

2) Flow-Level Sliding Window Framework: With the memory set to 1 MB, we compare 4 approaches: FBSW-CM, FBSW-Mrac, Baseline-CM and Baseline-Mrac.



(a) Traditional sliding window. (b) Flow-level sliding window.

Fig. 10: Real-time frequency distribution.

Real-time Frequency Distribution: As shown in Fig. 10b, the estimated frequency distribution of FBSW-CM and FBSW-Mrac is closer to the real frequency distribution. The Baseline-CM and Baseline-Mrac deviates from the true distribution of frequencies by a large margin.

VI. CONCLUSION

Network measurement in sliding window is an important and challenging work. Although there are some sliding window algorithms, they have the limitations of poor generality, coarse-grained, and poor memory utilization. In this paper, we present the two general framework for the sliding window, and the bit-wise adaptive allocation algorithm. We use our framework to three fundamental queries in sliding window: frequency estimation, heavy hitter detection, and frequency distribution estimation. Experimental results show that after using our framework, the above sketches that do not support sliding window achieves much higher accuracy and much lower memory usage than the current state-of-the-art.

ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China under Grant Nos. U22B2005, 62032013, 92267206 and 62072091, and supported by Applied Basic Research Program Project of Liaoning Province (2023JH2/101300192), and financial support of Lingnan University (LU) (DB23A9) and Lam Woo Research Fund at LU (871236).

REFERENCES

[1] Q. Huang, X. Jin, P. P. Lee, R. Li, L. Tang, Y.-C. Chen, and G. Zhang, "Sketchvisor: Robust network measurement for software packet processing," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, 2017.

[2] F. Li, K. Guo, J. Shen, and X. Wang, "Effective network-wide traffic measurement: A lightweight distributed sketch deployment," in *IEEE INFOCOM 2024-IEEE Conference on Computer Communications*. IEEE, 2024.

[3] X. Gou, L. He, Y. Zhang, K. Wang, X. Liu, T. Yang, Y. Wang, and B. Cui, "Sliding sketches: A framework using time zones for data stream processing in sliding windows," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020.

[4] Y. Zhou, Y. Zhou, S. Chen, and Y. Zhang, "Per-flow counting for big network data stream over sliding windows," in *2017 IEEE/ACM 25th International Symposium on Quality of Service (IWQoS)*. IEEE, 2017.

[5] H. Sun, J. Li, J. He, J. Gui, and Q. Huang, "Omniwindow: A general and efficient window mechanism framework for network telemetry," in *Proceedings of the ACM SIGCOMM 2023 Conference*, 2023.

[6] Z. Fan, Y. Zhang, S. Dong, Y. Zhou, F. Liu, T. Yang, S. Uhlig, and B. Cui, "Hoppingsketch: More accurate temporal membership query and frequency query," *IEEE Transactions on Knowledge and Data Engineering*, 2022.

[7] O. Papapetrou, M. Garofalakis, and A. Deligiannakis, "Sketch-based querying of distributed sliding-window data streams," *Proc. VLDB Endow.*, 2012.

[8] N. Rivetti, Y. Busnel, and A. Mostefaoui, "Efficiently summarizing data streams over sliding windows," in *2015 IEEE 14th International Symposium on Network Computing and Applications*. IEEE, 2015.

[9] R. Ben-Basat, G. Einziger, R. Friedman, and Y. Kassner, "Heavy hitters in streams and sliding windows," in *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*. IEEE, 2016.

[10] R. Shahout, R. Friedman, and D. Adas, "Cell: counter estimation for per-flow traffic in streams and sliding windows," in *2021 IEEE 29th International Conference on Network Protocols (ICNP)*. IEEE, 2021.

[11] M. Datar, A. Gionis, P. Indyk, and R. Motwani, "Maintaining stream statistics over sliding windows," *SIAM journal on computing*, 2002.

[12] Y. Zhu and D. Shasha, "Statstream: Statistical monitoring of thousands of data streams in real time," in *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*. Elsevier, 2002.

[13] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, 2005.

[14] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, "Heavy-hitter detection entirely in the data plane," in *Proceedings of the Symposium on SDN Research*, 2017.

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

[16] X. Liu, Y. Xu, P. Liu, T. Yang, J. Xu, L. Wang, G. Xie, X. Li, and S. Uhlig, "Sead counter: Self-adaptive counters with different counting ranges," *IEEE/ACM Transactions on Networking*, 2021.

[17] R. B. Basat, G. Einziger, M. Mitzenmacher, and S. Vargaftik, "Salsa: self-adjusting lean streaming analytics," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 2021.

[18] R. Ding, S. Yang, X. Chen, and Q. Huang, "Bitsense: Universal and nearly zero-error optimization for sketch counters with compressive sensing," in *Proceedings of the ACM SIGCOMM 2023 Conference*, 2023.

[19] Q. Shi, C. Jia, W. Li, Z. Liu, T. Yang, J. Ji, G. Xie, W. Zhang, and M. Yu, "Bitmatcher: Bit-level counter adjustment for sketches," in *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, 2024.

[20] J. Gong, T. Yang, Y. Zhou, D. Yang, S. Chen, B. Cui, and X. Li, "Abc: a practicable sketch framework for non-uniform multisets," in *2017 IEEE International Conference on Big Data (Big Data)*. IEEE, 2017.

[21] T. Yang, J. Gong, H. Zhang, L. Zou, L. Shi, and X. Li, "Heavyguardian: Separate and guard hot items in data streams," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2018.

[22] T. Yang, H. Zhang, J. Li, J. Gong, S. Uhlig, S. Chen, and X. Li, "Heavykeeper: an accurate algorithm for finding top-k elephant flows," *IEEE/ACM Transactions on Networking*, 2019.

[23] A. Metwally, D. Agrawal, and A. El Abbadi, "Efficient computation of frequent and top-k elements in data streams," in *International conference on database theory*. Springer, 2005.

[24] A. Kumar, M. Sung, J. Xu, and J. Wang, "Data streaming algorithms for efficient and accurate estimation of flow size distribution," *ACM SIGMETRICS Performance Evaluation Review*, 2004.

[25] Y. Li, R. Miao, C. Kim, and M. Yu, "Flowradar: A better netflow for data centers," in *13th USENIX symposium on networked systems design and implementation (NSDI 16)*, 2016.

[26] "Caida anonymized internet traces 2018 dataset," http://www.caida.org/.

[27] R. Y. Hung, L.-K. Lee, and H.-F. Ting, "Finding frequent items over sliding windows with constant update time," *Information Processing Letters*, 2010.