SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

April 2024, Vol. 67, Iss. 4, 142101:1–142101:18
https://doi.org/10.1007/s11432-022-3794-1

Jigsaw-Sketch: a fast and accurate algorithm for finding top-*k* elephant flows in high-speed networks

Boyu ZHANG[1], He HUANG[1*], Yu-E SUN[2], Yang DU[1] & Dan WANG[1]

[1]*School of Computer Science and Technology, Soochow University, Suzhou* 215006*, China;*
[2]*School of Rail Transportation, Soochow University, Suzhou* 215006*, China*

Received 17 December 2022/Revised 17 March 2023/Accepted 2 June 2023/Published online 20 March 2024

Abstract Finding top-k elephant flows in high-speed networks is one of the most fundamental network measurement tasks. It is more challenging than per-flow size estimation since the IDs and sizes of top-k flows must be tracked simultaneously. Most existing studies only record the IDs of a small number of elephant flows to fit their estimators in the extremely limited high-speed on-chip memory. However, these solutions need too many memory accesses when a packet arrives to track the elephant flows with high accuracy, which limits their practicability. Therefore, this paper proposes Jigsaw-Sketch, a new algorithm to find the top-k elephant flows with much fewer memory accesses while achieving high memory efficiency and accuracy. In this design, we propose a novel two-stage jigsaw storage scheme, which can capture the candidate top-k flows from massive network steams efficiently, and further find the top-k elephant flows with high memory efficiency and only a few memory accesses for each packet. Extensive experimental results based on real network traces show that Jigsaw-Sketch improves the packet processing throughput by at least 86%, while achieving smaller memory footprints and higher accuracy compared to the SOTA.

Keywords sketch, top-k, network measurement, elephant flow, high-speed networks

1 Introduction

Traffic measurement in high-speed networks plays an essential work in improving network performance [1–9]. One fundamental measurement task is to find top-k elephant flows, which also has many practical applications in databases [10,11], data mining [12,13], and security [14,15]. In practice, a flow is composed of all the packets containing the same flow ID, and a flow ID is usually defined as a combination of packet headers, e.g., the network five tuples[1)]. The number of packets in each flow is described as the flow size. In real networks, the flow size distribution is always highly skewed: most flows are mouse flows with small sizes, while a few flows are elephant flows which contribute a large portion of the network traffic. Finding top-k elephant flows is quite significant because the network managers can perform fine-grained network management on these top-k flows, such as load balancing, congestion control, and anomaly detection.

Different from counting the number of packets (i.e., per-flow size measurement [6,7,16–19]), finding top-k flows is more challenging because it requires tracking the IDs and sizes of top-k flows simultaneously. Accurately tracking the information of all flows is unrealizable due to the mismatch between the limited high-speed on-chip memory (like static random access memory (SRAM)) and the extremely high line rates of modern networks. For example, a one-minute real Internet trace downloaded from CAIDA contains more than one million flows. At the same time, SRAM is usually smaller than 8.25 MB [20] and is shared among many essential network functions. Consequently, only recording the IDs of a small portion of flows (i.e., top-k flow candidates) gains wide acceptance. This method processes each arrival packet to identify elephant flows and select the candidates from them to record IDs. When the measurement ends, the IDs and sizes of top-k flows can be reported based on the recorded candidates.

* Corresponding author (email: huangh@suda.edu.cn)

1) The network five-tuple consists of source IP, destination IP, source port, destination port, and protocol, whose size is 13 bytes.

The recent work puts much effort into improving the accuracy of identifying elephant flows. For example, HeavyKeeper [9, 21] actively removes mouse flows to make room for elephant flows; HeavyGuardian [22] only allows mouse flows to decay the smallest tracked flows' counters so that the noises introduced to elephant flows can be reduced. However, they can hardly be adapted to high-speed network links since they ignore the memory access bottleneck. For instance, a 40 Gbps link fed by minimum-size Ethernet packets (512 bits per packet) has to forward each packet in about 12 ns, allowing only a small number of memory accesses in SRAM. Since many other network functions also require memory accesses, a single traffic measurement function should reduce its memory accesses as much as possible. Unfortunately, existing solutions require too many memory accesses. HeavyKeeper [21] employs a min-heap with k slots to track the largest k flows it has identified. To check whether the arrival flow has been tracked, it needs to perform $O(k)$ memory accesses to traverse the min-heap. ElasticSketch [23], HeavyGuardian [22], and WavingSketch [24] employ a bucket array to identify the candidates and record their IDs and flow sizes. In their design, each flow is mapped to one bucket, and each bucket usually contains eight or more candidates. When a packet arrives, they traverse the corresponding bucket to update the candidates. Since each bucket can reach thousands of bits when storing the entire five-tuple IDs, traversing a whole bucket requires a dozen memory accesses. There are also some methods that employ a Stream-Summary instead of the min-heap [9, 25–27]. The Stream-Summary achieves an $O(1)$ average time complexity to insert or update a flow, but the pointer operations in its design lead to a number of discontiguous memory accesses. Besides, the pointers consume much memory, making the scheme not memory efficient and decreasing its accuracy. In brief, existing work ignores the importance of tracking elephant flows with fewer memory accesses, which has become a bottleneck in state-of-the-art sketches.

In this paper, we propose Jigsaw-Sketch, a new algorithm that can simultaneously achieve high accuracy and high speed. Our algorithm divides packet processing into two stages. The first stage records the fingerprints[2)] and flow sizes in a bucket array to capture the elephant flows and select the candidate top-k flows from them. Then, the second stage records the ID information of the candidate flows in an auxiliary list. The prior methods obtain bucket indexes and fingerprints through Hash functions, and we have to store an entire ID and an extra fingerprint for each candidate flow if following their methods, which is not memory efficient. Therefore, we propose a novel jigsaw storage scheme to realize our two-stage design. It implements the conversion between flow ID and jigsaw, where the jigsaw pieces are a bucket index, a fingerprint, and a residual part. Through this scheme, we can employ bucket indexes and fingerprints to store parts of the ID information and only store the residual parts rather than the entire IDs in the second stage. It reduces the memory consumption to track the candidates and improves our memory efficiency and algorithm accuracy.

Our two-stage design reduces the memory accesses from the following aspects. On the one hand, most packets will only get through the first stage, i.e., they only retrieve the fingerprints rather than the entire IDs. It significantly reduces the data each packet retrieved. On the other hand, when a candidate flow gets to the second stage, its recording position has been determined by the position it is identified in the first stage. In other words, we do not need to perform extra memory accesses to find the recording position. To improve algorithm speed and accuracy further, we adopt another technique: probabilistic replacement strategy. It employs lightweight computations to keep the elephant flows and makes mouse flows easily replaced by elephant flows, enabling us to quickly and accurately capture the elephant flows in each bucket. With the above techniques, Jigsaw-Sketch can simultaneously achieve higher accuracy and packet processing throughput than state-of-the-art algorithms.

This paper makes the following contributions:

- We propose Jigsaw-Sketch, a new algorithm for finding top-k elephant flows. It is based on a novel two-stage design and a novel jigsaw storage scheme, which allow our algorithm to reduce memory accesses while achieving high memory efficiency and accuracy.
- We mathematically prove the error bound of Jigsaw-Sketch and implement Jigsaw-Sketch on software (central processing unit, CPU) and hardware (field programmable gate array, FPGA) platforms[3)].
- We conduct extensive experiments with real Internet traces. Experimental results show that Jigsaw-Sketch is at least 86% faster than the fastest prior work while achieving higher accuracy than the most accurate work using smaller memory footprints.

2) A fingerprint is a string much shorter than the flow ID and lossily compresses the ID information of a flow.

3) The source code of Jigsaw-Sketch and other related algorithms have been made available at https://github.com/duyang92/jigsaw-sketch-paper.

2 Related work

2.1 Per-flow size estimation

The per-flow size estimation task aims to answer the query for each flow's size. State-of-the-art solutions employ various bit/counter sharing strategies in the sketches to reduce memory consumption [6, 7, 16–19]. For example, CM sketch [16] employs a pool of counters to record flow sizes. For each arrival packet, it hashes the packet's flow ID into d counters and increases each of them by 1. When querying a flow's size, it returns the smallest value of the d hashed counters as the estimation. Since counters are shared between different flows, CM sketch can employ much fewer counters than the flows for estimation. However, most per-flow size estimation methods do not store the flow IDs. In other words, they must rely on external means to record the flow IDs, while the means are rarely described.

2.2 Finding top-k elephant flows

Finding top-k elephant flows aims to find the k flows with the largest sizes. Unlike per-flow size estimation, this task requires tracking the flow IDs and sizes of top-k elephant flows simultaneously, making it more challenging. To introduce the state-of-the-art methods and their limitations, we divide them into the following categories according to their flow ID storage schemes.

2.2.1 *Min-heap storage scheme*

It is straightforward to employ a per-flow size estimation structure to estimate all flows' sizes, then use a min-heap with k slots to track top-k elephant flows (e.g., [28, 29]). However, these methods easily misclassify mouse flows as elephant flows due to the counter sharing strategies, limiting their accuracy for top-k finding.

To solve this problem, HeavyKeeper [21] combines each counter with a fingerprint field as a bucket, where the fingerprint is employed to identify flows. Then, it leverages an exponential decay strategy to capture elephant flows. Suppose all the hashed buckets record other flows rather than the arrival flow. The exponential decay strategy will decay the smallest bucket containing counter value c with a probability of b^{-c}, where b is a predefined constant number (e.g., $b = 1.08$). Then, if the counter is decayed to 0, the new flow will be inserted into the bucket. Though this strategy can capture elephant flows accurately, its exponential computation is quite expensive, which limits algorithm speed.

Except for the above limitations, the methods based on min-heap are limited in packet processing throughput due to the large number of memory accesses required by min-heap. In order to track the largest k flows, existing methods always need to check whether the arrival flow has been tracked. In other words, they need to lookup the arrival flow in the min-heap. The lookup process needs to traverse the min-heap and requires $O(k)$ memory accesses, which is unacceptable for high-speed networks. Some studies add a Hash table to accelerate the lookup process. However, the methods still need to access the Hash table and min-heap multiple times to update a tracked flow, which still constrains algorithm speed.

2.2.2 *Stream-Summary storage scheme*

Stream-Summary has the same function as a min-heap, but it employs linked lists and a Hash table to achieve an $O(1)$ average time complexity when inserting a new flow or increasing a tracked flow's size by 1. It is employed in Space Saving [25], Unbiased Space Saving [27], and the extended version of HeavyKeeper [9]. The former methods employ a Stream-Summary as the whole data structure, while HeavyKeeper employs it to replace the min-heap to improve algorithm speed.

However, Stream-Summary is still limited in speed and has low memory efficiency. The pointer operations in its insertion and update operations will access discontiguous memory multiple times, which decreases its speed. Besides, Stream-Summary needs to store about 7 pointers for each tracked flow, which consumes a significant amount of memory and makes Stream-Summary not memory efficient. Compact Space Saving [26] redesigns Stream-Summary to reduce memory consumption by replacing pointers with array indexes and employing a compact Hash table [30]. Nevertheless, storing indexes still consumes a large amount of memory.

2.2.3 *Bucket storage scheme*

The methods following this scheme employ buckets to capture candidate flows and directly store the flow IDs along with the counters in each bucket [22–24, 31–35]. Specifically, there are two main ways to configure the buckets. The first way only records one flow in each bucket and hashes each flow to multiple buckets to update flow information (e.g., [31–35]). ElasticSketch [23], HeavyGuardian [22], and WavingSketch [24] employ the other way, which hashes each flow to one bucket, where each bucket contains multiple cells to record the IDs and sizes of multiple flows simultaneously. When each packet arrives, these methods traverse the hashed bucket to find whether the flow has been tracked and determine how to update the bucket.

The packet processing speed of the methods following this scheme is faster but still unsatisfactory. The former way has multiple Hash computations and discontiguous memory accesses, which limits its speed. Instead, the latter way only has one Hash computation, and the accessed memory is continuous. However, it usually needs to configure long buckets to achieve high accuracy due to the flow collision problem, which incurs a number of memory accesses. For example, from our experiments, WavingSketch needs to set 16 or more cells in each bucket to achieve high accuracy. Then, each bucket can reach thousands of bits when recording the entire network five-tuple flow IDs, which requires a dozen memory accesses to traverse the bucket. Therefore, methods following this scheme are still limited in packet processing throughput and can hardly fit in high-speed networks.

3 Preliminary

3.1 Problem statement

Consider a packet stream $\mathcal{P} = \{\mathbb{P}_1, \mathbb{P}_2, \ldots, \mathbb{P}_N\}$ during a measurement period. Each packet $\mathbb{P} \in \mathcal{P}$ belongs to a flow, and the definition of flow ID is configured based on application interests, which is usually a combination of packet headers, e.g., the network five tuples. We model the packet stream as a set of flows $\mathcal{F} = \{f_1, f_2, \ldots, f_M\}$, and each flow f_i consists of all the packets carrying its flow ID.

The objective of finding top-k elephant flows is to find the k flows with the largest sizes (i.e., number of packets). Let n_i be the actual size of flow f_i, and $\hat{n}_i$ be the estimated value of n_i. Given an integer k, the output is a list of k flows and their estimated sizes, i.e., $\{(f_{j_1}, \hat{n}_{j_1}), (f_{j_2}, \hat{n}_{j_2}), \ldots, (f_{j_k}, \hat{n}_{j_k})\}$, where f_{j_i} is the flow with the i-th largest estimated size $\hat{n}_{j_i}$.

3.2 Our goals

We aim to design a new algorithm for finding top-k elephant flows, which should achieve high throughput while maintaining high accuracy. Specifically, it should satisfy the following properties:

(1) High throughput. In order to fit in high-speed networks, we want our sketch to achieve high packet processing throughput by reducing the memory accesses for each packet.

(2) High accuracy. Our algorithm should achieve high accuracy to support upper-layer applications. We want it to be more accurate than the most accurate prior work.

(3) High memory efficiency. In order to fit in the limited memory, our design should be memory efficient, which means it can work well even when using very small memory footprints.

4 Algorithm design

4.1 Approach overview

Jigsaw-Sketch aims to achieve a high packet processing throughput while providing high accuracy. To this end, we propose a novel two-stage design along with the jigsaw storage scheme and adopt a probabilistic replacement strategy.

We first introduce the strawman version of the two-stage design to explain its rationale. The two-stage design separates the top-k finding task into a capturing stage and a storage stage. The capturing stage aims to capture candidate top-k flows. For this purpose, we store fingerprints and flow sizes in a bucket array to identify the flows with large sizes. Since the fingerprints are much shorter than flow IDs, this design can significantly reduce the data to be retrieved compared to storing entire IDs, which means the memory accesses can be reduced. As the capturing stage only stores fingerprints, we need to store the IDs of the candidate top-k flows in the storage stage. To avoid performing extra memory accesses for finding the storage position, we design a one-to-one position mapping, which enables each captured flow to directly locate its storage position according to the position it is captured. Such a two-stage design also allows us to select only a few packets of the candidate flows to store the IDs, reducing memory accesses for most packets.

Though the employment of fingerprints in the strawman two-stage design reduces the memory accesses, we need to store a flow ID and an extra fingerprint for each candidate top-k flow, which means extra memory consumption. We propose a novel jigsaw storage scheme to improve this design. It can convert each flow ID to three jigsaw pieces: a bucket index, a fingerprint, and a residual part, and the original flow ID can be recovered from the corresponding pieces. Through this scheme, we can employ the bucket indexes and fingerprints in the capturing stage to store parts of the ID information and store the residual parts rather than the entire IDs in the storage stage. Since the bucket indexes do not occupy memory, the storage of jigsaw pieces can even consume less memory than storing the original ID, improving our memory efficiency and algorithm accuracy.

Besides, we adopt a probabilistic replacement strategy to improve algorithm speed and accuracy further. In the capturing stage, the flows mapped to each bucket will significantly exceed the bucket capacity. In order to capture the candidate top-k elephant flows, the replacement strategy employs the new arrival flow to replace the recorded smallest flow in the mapped bucket using a probability that decreases with the increased recorded flow size. Thus, this strategy tends to keep the elephant flows and replace the mouse flows with new flows. Moreover, the probability calculation formula in the strategy is lightweight, enabling us to capture the elephant flows with high accuracy and high speed.

4.2 Two-stage data structure of Jigsaw-Sketch

As shown in Figure 1, the data structures of Jigsaw-Sketch contain a bucket array B with w buckets and an auxiliary list L with u slots, which are separately employed in the two stages of our algorithm. In bucket array B, each bucket comprises λ cells: the first λ_h cells are heavy cells, and the other $\lambda - \lambda_h$ cells are light cells. The physical structure of each cell is the same, consisting of two fields: a fingerprint field for identifying flows and a counter field to record flow sizes. We will discuss the difference between heavy cells and light cells later. For convenience, we use $B[i][j]$ to represent the j-th cell of the i-th bucket in B, and use $B[i][j]$.FP and $B[i][j].C$ to represent its fingerprint field and counter field, respectively.

We build a one-to-one mapping between the slots in the auxiliary list L and the heavy cells in bucket array B. Each heavy cell $B[i][j]$ is bound to slot $L[i\lambda_h + j]$. Thus, the number of slots is relevant to the number of heavy cells, i.e., $u = w\lambda_h$. Each slot has a residual part field and a signal field. The former stores the residual part of the ID information, and the latter is a 2-bit counter for handling fingerprint collisions. Similarly, the residual part field and signal field of the i-th slot in the auxiliary list L is represented as $L[i]$.RP and $L[i].S$ respectively.

It is worth noting that there are no slots bound to light cells due to the different functionality of light cells compared with heavy cells. We set multiple cells in each bucket to collect the size information of more flows when determining which flows should be tracked. It also reduces the collisions among elephant flows. However, most cells will record mouse flows, and storing the residual parts for them is unnecessary. To improve memory efficiency, we let part cells be light cells without bounding to slots and employ them to track the potential candidate top-k flows and find the candidate flows from them. The experiments in Subsection 6.4 also demonstrate the effectiveness of this design.

4.3 Techniques

4.3.1 *Jigsaw storage scheme*

We propose the jigsaw storage scheme to improve the memory efficiency of our algorithm. It implements the conversion between flow ID and jigsaw, where the jigsaw pieces are a bucket index, a fingerprint, and a residual part, respectively. The design of our jigsaw storage scheme has to meet the following requirements. (1) It must be able to recover the flow IDs from the corresponding jigsaw pieces, like operating a jigsaw. (2) The bit length of the jigsaw should not exceed the ID length. Otherwise, the memory efficiency of our sketch will be compromised. (3) The bucket indexes and fingerprints generated by the scheme have to be well-randomized since their randomness can significantly affect the performance

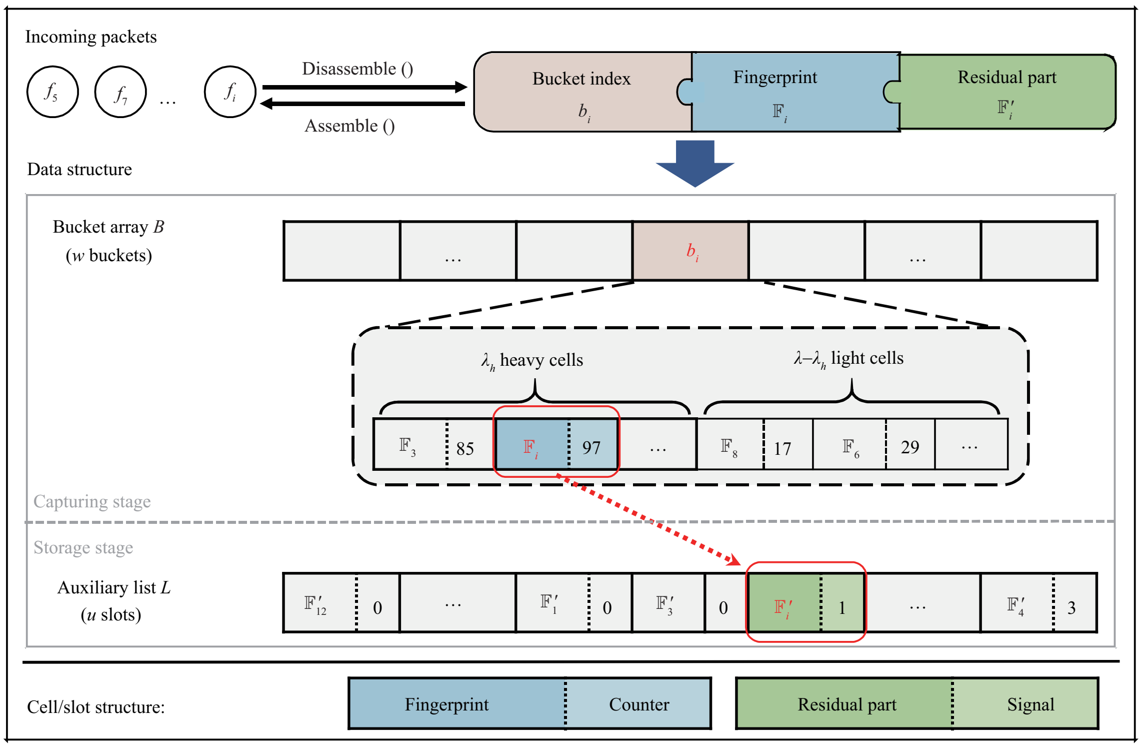

Figure 1 Data structure of Jigsaw-Sketch.

of our algorithm. (4) The scheme must be lightweight in computation overhead so that it will not constrain packet processing throughput.

It is challenging to meet all the requirements simultaneously, and existing methods, such as encryption algorithms and Hash algorithms, can only meet a part of them. For example, aiming to protect the plain text, the encryption algorithms usually output longer encrypted messages or have heavy computation overhead. Besides, they do not consider the problem of generating well-randomized bucket indexes and fingerprints. The Hash algorithms can generate well-randomized bucket indexes and fingerprints, while they do not support recovering the Hash values to original IDs.

To achieve the requirements, we design a simple and invertible disassembling algorithm (Algorithm 1) using the theory of modular inverses, shifting operations (represented by $\ll$ and $\gg$), and exclusive or (XOR) operations (represented by $\oplus$); and the assembling algorithm is the inverse process (Algorithm 2). In the disassembling algorithm, we first randomize the flow ID as shown in line 2 of Algorithm 1. The inverse process is shown in line 11 of Algorithm 2. This invertible randomization step employs the theory of modular inverses: for a pair of modular inverses a and a' with regard to 2^l (i.e., $aa' \bmod 2^l = 1$, which requires a to be relatively prime to 2^l), they can be employed to randomize and recover the flow ID f_i according to the following formulas, where l is the bit length of flow ID.

$$x = f_i a \bmod 2^l, \qquad f_i = xa' \bmod 2^l.$$

This randomization is actually a remapping for the flow ID, avoiding similar IDs getting similar or even the same bucket indexes and fingerprints. However, the value x is not randomized enough to generate well-randomized bucket indexes and fingerprints since the lowest bits of x cannot summarize the information of the entire ID. XOR is widely employed in Hash algorithms for randomization, which is fast and can output the same number of bits as the operands. Thus, we employ XOR to randomize the ID and summarize all ID information to the lowest bits. As shown in lines 3–10 of Algorithm 1, we carefully design the XOR steps to make this process invertible, and the inverse codes are shown in lines 3–10 of Algorithm 2. After the two randomization steps, the randomized value still has the same number of bits as the flow ID, and we can get well-randomized bucket indexes and fingerprints using the modular operations (lines 11 and 12 in Algorithm 1).

Algorithm 1 Disassembling method of the jigsaw storage scheme

```
Input: Flow ID $f_i$.
Output: Bucket index $b_i$, fingerprint $\mathbb{F}_i$, residual part $\mathbb{F}'_i$.
Params: ID bit length $l$, fingerprint bit length $l'$, integer $a$ that is relatively prime to $2^l$.
 1: Function Disassemble($f_i$):
 2: $x \leftarrow f_i a \bmod 2^l$;
 3: $y_1 \leftarrow (x \gg 0) \bmod 2^{\frac{l}{4}}$;    $y_2 \leftarrow (x \gg \frac{l}{4}) \bmod 2^{\frac{l}{4}}$;
 4: $y_3 \leftarrow (x \gg \frac{l}{2}) \bmod 2^{\frac{l}{4}}$;    $y_4 \leftarrow (x \gg \frac{3l}{4}) \bmod 2^{\frac{l}{4}}$;
 5: $y'_1 \leftarrow y_1$;
 6: $y'_2 \leftarrow y_1 \oplus y_2$;
 7: $y'_3 \leftarrow y_1 \oplus y_2 \oplus y_3$;
 8: $y'_4 \leftarrow y_1 \oplus y_2 \oplus y_3 \oplus y_4$;
 9: $y'_4 \leftarrow y'_4 \oplus (y'_4 \gg \frac{l}{8})$;
10: $z \leftarrow (y'_1 \ll \frac{3l}{4}) + (y'_2 \ll \frac{l}{2}) + (y'_3 \ll \frac{l}{4}) + y'_4$;
11: $b_i \leftarrow z \bmod w$;    $z \leftarrow \lfloor \frac{z}{w} \rfloor$;
12: $\mathbb{F}_i \leftarrow z \bmod 2^{l'}$;    $z \leftarrow \lfloor \frac{z}{2^{l'}} \rfloor$;
13: $\mathbb{F}'_i \leftarrow z$;
14: Return $b_i$, $\mathbb{F}_i$, $\mathbb{F}'_i$.
```

Algorithm 2 Assembling method of the jigsaw storage scheme

```
Input: Bucket index $b_i$, fingerprint $\mathbb{F}_i$, residual part $\mathbb{F}'_i$.
Output: Flow ID $f_i$.
Params: ID bit length $l$, fingerprint bit length $l'$, integer $a'$ that is the modular inverse of $a$ with regard to $2^l$.
 1: Function Assemble($b_i, \mathbb{F}_i, \mathbb{F}'_i$):
 2: $z \leftarrow ((\mathbb{F}'_i \ll l') + \mathbb{F}_i)w + b_i$;
 3: $y'_1 \leftarrow (z \gg \frac{3l}{4}) \bmod 2^{\frac{l}{4}}$;    $y'_2 \leftarrow (z \gg \frac{l}{4}) \bmod 2^{\frac{l}{4}}$;
 4: $y'_3 \leftarrow (z \gg \frac{l}{2}) \bmod 2^{\frac{l}{4}}$;    $y'_4 \leftarrow (z \gg 0) \bmod 2^{\frac{l}{4}}$;
 5: $y'_4 \leftarrow y'_4 \oplus (y'_4 \gg \frac{l}{8})$;
 6: $y_1 \leftarrow y'_1$;
 7: $y_2 \leftarrow y'_1 \oplus y'_2$;
 8: $y_3 \leftarrow y'_2 \oplus y'_3$;
 9: $y_4 \leftarrow y'_3 \oplus y'_4$;
10: $x \leftarrow y_1 + (y_2 \ll \frac{l}{4}) + (y_3 \ll \frac{l}{2}) + (y_4 \ll \frac{3l}{4})$;
11: $f_i \leftarrow xa' \bmod 2^l$;
12: Return $f_i$.
```

Next, we analyze the improved memory efficiency of our scheme compared to directly storing fingerprints and flow IDs. For each candidate flow, our scheme can save $\lfloor l' + \log_2 w \rfloor$ bits, where l' is the bit length of fingerprints and w is the number of buckets. For example, when the fingerprints are set to 16 bits and buckets are more than 512 (common in existing work), our scheme can save at least 25 bits for each candidate flow. It significantly improves our algorithm's memory efficiency and allows it to achieve high accuracy using small memory footprints.

4.3.2 *Probabilistic replacement strategy*

In the bucket array of the capturing stage, the flows mapped to each bucket are much more than the cells in the bucket, and we want to keep the elephant flows and expel the mouse flows. Therefore, when a new flow arrives and all the cells in the bucket are occupied, we need to determine whether to replace the smallest recorded flow with the new flow or to keep the recorded flows. The exponential decay strategy [9, 21, 22] has high accuracy, while its exponential computation is expensive, which limits algorithm speed. In contrast, the unbiased replacement strategy [27, 33] and waving counter strategy [24] have light computation overhead, while they have larger estimation errors.

In order to capture the elephant flows accurately with light computation overhead, we adopt a probabilistic replacement strategy. Suppose the new arrival flow is f_i, and the smallest cell in the bucket belongs to f_j, which has a counter value of c. Our strategy will try to replace f_j with f_i using a probability of $p_r = \frac{1}{c}$. This strategy is similar to but different from the unbiased replacement strategy. The unbiased replacement strategy will always increase the counter of f_j from c to $c+1$ and then try to replace f_j with a probability of $p_r = \frac{1}{c+1}$. This "always increment" choice will incur extra errors on the smallest counter, and the rapid increment on the smallest counter will also lead to lower accuracy for finding top-k flows. Overall, the probabilistic replacement strategy we employ has two main advantages. First, its computation is lightweight and will not compromise algorithm throughput. Second, the strategy will not change the flow sizes of the recorded flows if the replacement fails, i.e., a failed replacement will not introduce estimation errors to the recorded flows.

4.4 Operations for finding top-k elephant flows

In this subsection, we give the details of the insertion and query operations for finding top-k elephant flows. Before each measurement, the bucket array and auxiliary list of Jigsaw-Sketch have to be initialized, i.e., we have to set all fingerprints, counters, and the fields in slots to 0.

4.4.1 *Insertion*

The pseudo-code of the insertion operation is shown in Algorithm 3. For each arrival packet, we first process it in the first stage. Specifically, we extract the packet's flow ID f_i and employ the disassembling method of the jigsaw storage scheme to turn f_i into three pieces: bucket index b_i, fingerprint $\mathbb{F}_i$, and the residual part $\mathbb{F}'_i$ (line 4). According to the bucket index b_i, we can find the mapped bucket $B[b_i]$ of flow f_i and then apply different operations for the following three cases:

Algorithm 3 Insertion

```
Input: Flow ID $f_i$.
 1: Function Insert($f_i$):
 2: $m \leftarrow 0$; //capture the smallest heavy cell
 3: $m' \leftarrow 0$; //capture the smallest cell
 4: $b_i, \mathbb{F}_i, \mathbb{F}'_i \leftarrow$ Disassemble($f_i$);
 5: for $j \leftarrow 0$ to $\lambda - 1$ do
 6:     if $B[b_i][j].C = 0$ then //Case 1
 7:         $B[b_i][j].\mathrm{FP} \leftarrow \mathbb{F}_i$;
 8:         $B[b_i][j].C \leftarrow 1$;
 9:         if $j < \lambda_h$ then
10:             ALSet($b_i, j, \mathbb{F}'_i$);
11:         end if
12:         Return;
13:     end if
14:     if $B[b_i][j].\mathrm{FP}=\mathbb{F}_i$ then //Case 2
15:         $B[b_i][j].C \leftarrow B[b_i][j].C + 1$;
16:         if $j \geqslant \lambda_h$ and $B[b_i][j].C > B[b_i][m].C$ then
17:             Exchange values of $B[b_i][j]$ and $B[b_i][m]$;
18:             ALSet($b_i, m, \mathbb{F}'_i$);
19:         end if
20:         if $j < \lambda_h$ and ($B[b_i][j].C = T$ or ($B[b_i][j].C \geqslant T$ and
            Rand()< $\frac{1}{B[b_i][j].C}$)) then
21:             ALUpdate($b_i, j, \mathbb{F}'_i$);
22:         end if
23:         Return;
24:     end if
25:     if $j < \lambda_h$ and $B[b_i][j].C < B[b_i][m].C$ then
26:         $m \leftarrow j$;
27:     end if
28:     if $B[b_i][j].C < B[b_i][m'].C$ then
29:         $m' \leftarrow j$;
30:     end if
31: end for
32: if Rand()< $\frac{1}{B[b_i][m'].C}$ then //Case 3
33:     $B[b_i][m'].\mathrm{FP} \leftarrow \mathbb{F}_i$;
34:     if $m' < \lambda_h$ then
35:         ALSet($b_i, m', \mathbb{F}'_i$);
36:     end if
37: end if
38: Function ALSet($b_i, j, \mathbb{F}'_i$):
39: $L[b_i\lambda_h + j].\mathrm{RP} \leftarrow \mathbb{F}'_i$;
40: $L[b_i\lambda_h + j].S \leftarrow 0$;
41: Function ALUpdate($b_i, j, \mathbb{F}'_i$):
42: if $L[b_i\lambda_h + j].\mathrm{RP} = \mathbb{F}'_i$ then
43:     $L[b_i\lambda_h + j].S \leftarrow L[b_i\lambda_h + j].S + 1$;
44: else if $L[b_i\lambda_h + j].S > 0$ then
45:     $L[b_i\lambda_h + j].S \leftarrow L[b_i\lambda_h + j].S - 1$;
46: else
47:     $L[b_i\lambda_h + j].\mathrm{RP} \leftarrow \mathbb{F}'_i$;
48: end if
```

Case 1 (lines 6–13). There exists no cell storing fingerprint $\mathbb{F}_i$, and there remain empty cells. Suppose the first empty cell is $B[b_i][j]$. We will set the fingerprint field $B[b_i][j].\mathrm{FP}$ to $\mathbb{F}_i$ and the counter field $B[b_i][j].C$ to 1. Then, if $B[b_i][j]$ is a heavy cell, i.e., $0 \leqslant j < \lambda_h$, we need to access the auxiliary list in the second stage to record $\mathbb{F}'_i$. As shown in the ALSet() function (lines 38–40), the corresponding slot of heavy cell $B[b_i][j]$ is $L[b_i\lambda_h + j]$, and we set the residual part field $L[b_i\lambda_h + j].\mathrm{RP}$ and the signal field $L[b_i\lambda_h + j].S$ to $\mathbb{F}'_i$ and 0, respectively.

Case 2 (lines 14–24). There exists a cell $B[b_i][j]$ storing fingerprint $\mathbb{F}_i$. In this case, we increase the cell's counter field $B[b_i][j].C$ by 1 (line 15). Then, if $B[b_i][j]$ is a light cell, we have to find the smallest heavy cell in the bucket and compare it with $B[b_i][j]$. Suppose the smallest heavy cell is $B[b_i][m]$. If $B[b_i][m].C$ is smaller than $B[b_i][j].C$, we will exchange the contents of $B[b_i][j]$ and $B[b_i][m]$, and set corresponding slot $L[b_i\lambda_h + m]$ to $\langle \mathbb{F}'_i, 0\rangle$ in the second stage.

If $B[b_i][j]$ is a heavy cell, we still need to access the second stage and update the auxiliary list due to the fingerprint collision problem. Considering that the fingerprint of an elephant flow collides with a mouse flow, the recorded flow size will be the sum of their sizes. Moreover, if the mouse flow occupies the elephant flow's slot in the second stage, we can only recover the mouse flow and lose the elephant flow. In order to alleviate the fingerprint collision problem, we choose to access the second stage again when $B[b_i][j].C = T$, and then sample the packets to access the second stage with a probability of $\frac{1}{T}$ when $B[b_i][j].C > T$. The value T is employed to filter out the mouse flows applications do not care about, which can be set to a value much smaller than the concerned elephant flows' sizes and larger than the mouse flows' sizes. Also, it blocks most packets from accessing the auxiliary list in the second stage. The update process on the auxiliary list is shown in the ALUpdate() function (lines 41–48), which can be divided into the following three situations: (1) If $L[b_i\lambda_h + j]$.RP equals $\mathbb{F}'_i$, we increase $L[b_i\lambda_h + j].S$ by 1. (2) If $L[b_i\lambda_h + j]$.RP is different from $\mathbb{F}'_i$ and $L[b_i\lambda_h + j].S > 0$, we decrease $L[b_i\lambda_h + j].S$ by 1. (3) If $L[b_i\lambda_h + j]$.RP is different from $\mathbb{F}'_i$ and $L[b_i\lambda_h + j].S = 0$, we set $L[b_i\lambda_h + j]$.RP to $\mathbb{F}'_i$. Apparently, an elephant flow tends to update the slot multiple times. Then, even though a mouse flow updates the slot by coincidence, it can hardly decrease the signal field to 0 and update the residual part field. In contrast, it is easy for an elephant flow to update the residual part field when a mouse flow occupies it. Thus, this design can further avoid the mouse flow occupying the slot when it collides with an elephant flow.

Case 3 (lines 32–37). There exists no cell storing fingerprint $\mathbb{F}_i$, and all cells are occupied. In this case, we employ the probabilistic replacement strategy to capture the elephant flows. Suppose $B[b_i][m']$ is the smallest cell in the bucket. We try to use $\mathbb{F}_i$ to replace $B[b_i][m']$.FP with a probability of $p_r = \frac{1}{B[b_i][m'].C}$. Then, if the replacement succeeds and $B[b_i][m']$ is a heavy cell, we will access the second stage and set $L[b_i\lambda_h + m']$ to $\langle \mathbb{F}'_i, 0\rangle$.

4.4.2 *Top-k query*

At the end of each measurement period, we traverse each heavy cell in bucket array B and the corresponding slot in auxiliary list L to get top-k elephant flows. For each heavy cell $B[i][j]$, we get the counter value c, bucket index i, fingerprint $\mathbb{F}$, and find the residual part $\mathbb{F}'$ stored in its corresponding slot $L[i\lambda_h + j]$. Then, we employ the assembling method of the jigsaw storage scheme (Algorithm 2) to recover the flow ID and employ c as the estimated flow size. After recovering all candidate top-k flows captured by the heavy cells, we report the k flows with the largest estimated sizes as the top-k elephant flows.

5 Mathematical analysis

In this section, we conduct a mathematical analysis of our algorithm. We show the properties of the probabilistic replacement strategy first and then give two estimation error bounds of our algorithm.

5.1 Theoretical properties of the probabilistic replacement strategy

We analyze the probabilistic replacement strategy in two folds. First, we analyze the replacement strategy from the view of the flows to be replaced.

Theorem 1. The efforts of other flows to replace flow f will cause a downward biased estimation error on the flow size of f. Let X be this estimation error, then the expectation of X is

$$E(X) <= r, \tag{1}$$

where r is the number of times other flows try to replace f.

Proof. Consider a moment t where flow f has the smallest recorded flow size $\hat{n}(t)$ in the bucket. Suppose there are $r(t) \geqslant 1$ packets of other flows that try to replace flow f before the next appearance of flow f. Let $X(t)$ be the downward biased estimation error added on $\hat{n}(t)$. Obviously, if all the $r(t)$ replacements fail, flow f will keep its recorded size $\hat{n}(t)$ unchanged, i.e., $X(t) = 0$. Otherwise, once flow f is replaced by another flow successfully, flow f will lose its recorded size, i.e., $X(t) = \hat{n}(t)$. Since the probability that all replacements fail is $(1 - \frac{1}{\hat{n}(t)})^{r(t)}$, we have

$$E(X(t)) = \hat{n}(t)\left(1 - \left(1 - \frac{1}{\hat{n}(t)}\right)^{r(t)}\right) \leqslant r(t). \tag{2}$$

The inequality in the above formula can be proved using the theory of derivative; thus we omit it. Assuming that $r(t)$ is independent of $\hat{n}(t)$, we have

$$E(X) = \sum_t E(X(t)) \leqslant \sum_t r(t) = r. \tag{3}$$

Next, we analyze the replacement strategy from the view of the flows that try to replace other flows.

Theorem 2. When a flow f tries to replace another flow, the estimation error on its flow size caused by the replacements is unbiased.

Proof. Let f' be the flow with the smallest recorded size $\hat{n}_{\min}$ in the bucket. Once flow f tries to replace flow f', the estimated flow size of f will be incremented by $\hat{n}_{\min}$ with probability $\frac{1}{\hat{n}_{\min}}$. Therefore, the expected increment to the size of flow f is 1, which proves the unbiasedness.

5.2 Estimation error bound analysis

We analyze the estimation error bound of our algorithm when processing a packet stream with N packets that belong to M distinct flows. Let f_i be the i-th largest flow, whose actual size is n_i. The final estimated flow size of f_i is

$$\hat{n}_i = n_i - X_i + Y_i' - Y_i'' + Z_i, \tag{4}$$

where X_i is the decrement from being replaced; Y_i' and Y_i'' are upward and downward estimation errors caused when replacing other flows; Z_i is the increment from fingerprint collision.

5.2.1 *Upper estimation error bound analysis*

Theorem 3. Given the i-th largest flow f_i whose actual flow size is n_i, for any $\epsilon > 0$, the upper bound of estimation error is

$$\Pr\{\hat{n}_i - n_i \geqslant \epsilon N\} \leqslant \frac{1}{\epsilon}\left(\frac{1}{4\lambda w} - \frac{1}{4N} + \frac{1}{w2^{l'}}\right), \tag{5}$$

where l' is the bit length of the fingerprint fields.

Proof. For any two distinct flows f_i and f_j, we use $I_{i,j}$ to indicate that they are mapped to the same bucket and have the same fingerprint. Then we have $E(I_{i,j}) = \frac{1}{w2^{l'}}$. Since Z_i equals the total number of collisions occurring to f_i, we have

$$E(Z_i) = E\left(\sum_{j=1,j\neq i}^{M} I_{i,j}n_j\right) = \sum_{j=1,j\neq i}^{M} E(I_{i,j})n_j \leqslant E(I_{i,j})\sum_{j=1}^{M} n_j = \frac{N}{w2^{l'}}. \tag{6}$$

Next, we analyze the upward estimation error Y_i' caused when replacing other flows. Let $\hat{\mathbb{N}}_{\min}$ be the random variable representing the recorded smallest flow size in the bucket when flow f_i successfully replaces the smallest flow. Suppose flow f_i has tried the replacement for α times before the success, and the smallest flow sizes at the failed replacements are separately $\hat{n}_{\alpha_1}, \hat{n}_{\alpha_2}, \ldots, \hat{n}_{\alpha_\alpha}$. Note that we only consider the upward estimation error, which means $\alpha \leqslant \hat{\mathbb{N}}_{\min} - 1$. Therefore,

$$E(Y_i'|\hat{\mathbb{N}}_{\min} = \hat{n}_{\min}) = \sum_{\alpha=0}^{\hat{n}_{\min}-1}\left(\prod_{\beta=1}^{\alpha}\left(1 - \frac{1}{\hat{n}_{\alpha_\beta}}\right)\right)\frac{\hat{n}_{\min} - \alpha - 1}{\hat{n}_{\min}} \leqslant \sum_{\alpha=0}^{\hat{n}_{\min}-1}\frac{\hat{n}_{\min} - \alpha - 1}{\hat{n}_{\min}} = \frac{\hat{n}_{\min} - 1}{2}. \tag{7}$$

Let $\hat{\mathbb{N}}'_{\min}$ be the random variable representing the smallest flow size recorded in the bucket when answering a query after the measurement ends. We assume that when flow f_i successfully replaces the smallest flow, the value of the smallest recorded flow size is uniformly distributed within the range $[1, \hat{\mathbb{N}}'_{\min}]$ before the query time. Then the probability that $\hat{\mathbb{N}}_{\min}$ equals any integer within this range is $\frac{1}{\hat{\mathbb{N}}'_{\min}}$, and we have

$$E(Y_i'|\hat{\mathbb{N}}'_{\min} = \hat{n}'_{\min}) = \sum_{\hat{n}_{\min}=1}^{\hat{n}'_{\min}} E(Y_i'|\hat{\mathbb{N}}_{\min} = \hat{n}_{\min})\frac{1}{\hat{n}'_{\min}} \leqslant \frac{\hat{n}'_{\min} - 1}{4}. \tag{8}$$

Let $\Pr\{\hat{\mathbb{N}}'_{\min} = \hat{n}'_{\min}\}$ denote the probability that $\hat{\mathbb{N}}'_{\min}$ equals $\hat{n}'_{\min}$, we have

$$E(Y_i') = \sum_{\hat{n}'_{\min}} E(Y_i'|\hat{\mathbb{N}}'_{\min} = \hat{n}'_{\min})\Pr\{\hat{\mathbb{N}}'_{\min} = \hat{n}'_{\min}\} \leqslant \frac{E(\hat{\mathbb{N}}'_{\min}) - 1}{4} \leqslant \frac{\frac{N}{\lambda w} - 1}{4} = \frac{N}{4\lambda w} - \frac{1}{4}. \tag{9}$$

Then, we can use Markov inequality to transform the bound of expectation into the bound of possibility:

$$\begin{aligned}\Pr\{\hat{n}_i - n_i \geqslant \epsilon N\} &= \Pr\{-X_i + Y_i' - Y_i'' + Z_i \geqslant \epsilon N\} \leqslant \Pr\{Y_i' + Z_i \geqslant \epsilon N\} \\ &\leqslant \frac{E(Y_i') + E(Z_i)}{\epsilon N} \leqslant \frac{1}{\epsilon}\left(\frac{1}{4\lambda w} - \frac{1}{4N} + \frac{1}{w2^{l'}}\right). \end{aligned} \tag{10}$$

5.2.2 *Lower estimation error bound analysis*

Theorem 4. Given the i-th largest flow f_i whose actual size is n_i, for any $\epsilon > 0$, the lower estimation error bound is

$$\Pr\{n_i - \hat{n}_i \geqslant \epsilon N\} \leqslant \frac{1}{\epsilon}\left(\binom{i-1}{\lambda-1}\left(\frac{1}{w}\right)^{\lambda}\left(1-\frac{1}{w}\right)^{i-\lambda} + \frac{1}{4\lambda w} - \frac{1}{4N}\right). \tag{11}$$

Proof. For flow f_i, other flows try to replace it only when f_i is the smallest flow recorded in the bucket. In other words, $\lambda - 1$ flows among the largest $i-1$ flows are mapped to the bucket of flow f_i. Let U be the random variable representing the number of flows that is one of the largest $i-1$ flows and mapped to the same bucket with f_i. Obviously, U follows a binomial distribution $B(i-1, \frac{1}{w})$, and we have

$$\Pr\{U = \lambda - 1\} = \binom{i-1}{\lambda-1}\left(\frac{1}{w}\right)^{\lambda-1}\left(1-\frac{1}{w}\right)^{i-\lambda}. \tag{12}$$

Let R be the random variable representing the number of times that other flows try to replace f_i. As proved in Theorem 1, given the value of R, the expectation of the decrement caused by being replaced is $E(X_i|R=r) = r$. When flow f_i is the smallest flow recorded in the bucket, the expectation number of times that other flows try to replace it is

$$E(R|U=\lambda-1) = \frac{1}{w}\left(\sum_{j=i+1}^{M} n_j\right) \leqslant \frac{N}{w}. \tag{13}$$

Besides, the expectation of R is

$$E(R) = E(R|U=\lambda-1)\Pr\{U=\lambda-1\} \leqslant N\binom{i-1}{\lambda-1}\left(\frac{1}{w}\right)^{\lambda}\left(1-\frac{1}{w}\right)^{i-\lambda}. \tag{14}$$

Then, we can get the expectation of the decrement caused by being replaced as follows:

$$E(X_i) = \sum_r E(X_i|R=r)\Pr\{R=r\} = \sum_r r\Pr\{R=r\} = E(R). \tag{15}$$

Next, we analyze the downward estimation error Y_i''. Notice that the estimation error caused by replacing other flows is unbiased, as proved in Theorem 2. Therefore, the downward error's expectation should equal the upward error's expectation.

$$E(Y_i'') = E(Y_i') \leqslant \frac{N}{4\lambda w} - \frac{1}{4}. \tag{16}$$

We use Markov inequality to get the lower estimation error bound

$$\begin{aligned}\Pr\{n_i - \hat{n}_i \geqslant \epsilon N\} &= \Pr\{X_i - Y_i' + Y_i'' - Z_i \geqslant \epsilon N\} \leqslant \Pr\{X_i + Y_i'' \geqslant \epsilon N\} \\ &\leqslant \frac{E(X_i') + E(Y_i'')}{\epsilon N} \leqslant \frac{1}{\epsilon}\left(\binom{i-1}{\lambda-1}\left(\frac{1}{w}\right)^{\lambda}\left(1-\frac{1}{w}\right)^{i-\lambda} + \frac{1}{4\lambda w} - \frac{1}{4N}\right). \end{aligned} \tag{17}$$

We can find that the estimation error bounds are tightly relevant to the number of cells in each bucket. Besides, the lower estimation error caused by being replaced is also relevant to the ranking of the flow.

6 Experimental results

In this section, we evaluate the performance of our algorithm through extensive experiments using real Internet traffic traces downloaded from CAIDA. We first introduce our experimental setup. Next, we prove that the disassembling method of our jigsaw storage scheme can generate well-randomized bucket indexes and fingerprints. Then, we conduct experiments on the parameter settings. At last, we evaluate the performance of Jigsaw-Sketch with different values of k and memory size.

6.1 Experimental setup

Platform. We implement our algorithm on software (CPU) and hardware (NetFPGA) platforms. Specifically, we conduct our evaluation on a server equipped with two Intel Xeon E5-2643 v4 @3.40 GHz CPU and 256 GB RAM. Besides, we also implement our algorithm on a NetFPGA-1G-CML development board containing 16 Mbits Block RAM. When implementing our algorithm with 200 kB memory, we use 8199 of the 203800 total available LUTs. The estimated Fmax given by Vivado HLS is 115.55 MHz, meaning the processing throughput is 115.55 Mps.

Dataset. We use two real Internet traces downloaded from CAIDA4) as our datasets, where flow IDs are defined as the network five-tuples carried by the packets. The two traces are collected at different links and times. For the CAIDA-2016 dataset, we employ one minute of the traces collected at the link from Seattle to Chicago in 2016, which contains 30078201 packets and 1102525 distinct flows, where the largest flow contains 185668 packets. For the CAIDA-2019 dataset, we employ one minute of the traces collected at the link from Sao Paulo to New York in 2019, which contains 36144349 packets and 2607500 distinct flows, where the largest flow contains 162540 packets.

Implementation. We implement our Jigsaw-Sketch (JS for short) and the following solutions (baselines) for comparison: HeavyKeeper (HK) [9, 21], WavingSketch (WS) [24], HeavyGuardian (HG) [22], UA-Sketch (UA) [34], and ChainSketch (CS) [35]. Since the stream-summary version of HeavyKeeper [9] has low memory efficiency, we choose to implement its min-heap version [21] and add a Hash table to accelerate its lookup process, which performs as fast as the stream-summary version and is more memory efficient. All the solutions are implemented in C++ and compiled by a g++ compiler. Besides, the Hash functions are implemented by a well-known fast Hash function, Murmur Hash5). Since the baselines we implement have been proven superior to some other classical methods, such as Space Saving [25], Unbiased Space Saving [27], and ElasticSketch [23], we do not implement them.

Default parameters. The network managers usually get the maximum flow sizes from historical statistics to set the counter size in practice. Considering the largest flow sizes in both datasets are between 2^{17} and 2^{18}, we set the normal counters to 18 bits in all algorithms. We can also implement the counters by active counter techniques [6] to count larger values with fewer bits, while it is beyond this paper's scope. We set the fingerprint fields to 16 bits, which can ensure a small enough error caused by fingerprint collision, as proved in (6). In JS, the number of cells in each bucket is 8, and the number of heavy cells is 4. Besides, we set T to 512 because the sizes of top-k flows in high-speed networks are always much larger, and network managers usually do not care about the mouse flows whose sizes are smaller than 512. Similarly, the Hash chain threshold in ChainSketch is also set to 512. The min-heap capacity of HK is set to k, and the numbers of bucket arrays in HK, UA, and CS are set to 2, 4, and 4 separately. Besides, the maximum Hash chain length in CS is set to 3. Considering the precision and throughput, each bucket of WS and HG contains 16 and 8 cells separately.

6.2 Evaluation metrics

We employ relative standard deviation (RSD) and collision ratio (CR) to evaluate the performance of our jigsaw storage scheme's disassembling method in generating bucket indexes and fingerprints, and employ precision (PR), average relative error (ARE), average absolute error (AAE), and throughput to evaluate our algorithm's performance in finding top-k elephant flows.

RSD. Let x_i be the number of distinct flows mapped to the i-th bucket in a bucket array. The RSD of the number of distinct flows mapped to each bucket is $\mathrm{RSD} = \frac{1}{\bar{x}}\sqrt{\frac{1}{w}\sum_{i=1}^{w}(x_i-\bar{x})^2}$, where $\bar{x} = \frac{M}{w}$, M is the total number of distinct flows, and w is the number of buckets.

4) The CAIDA dataset can be achieved at http://www.caida.org/data/passive.

5) The source code of Murmur Hash is at https://github.com/aappleby/smhasher/tree/master/src/MurmurHash3.cpp.

Table 1 RSD and CR with different numbers of buckets on CAIDA-2016

	Ours		Murmur Hash		Bob Hash		BKDR Hash	
w	RSD ($\times$100)	CR ($\times$100)	RSD ($\times$100)	CR ($\times$100)	RSD ($\times$100)	CR ($\times$100)	RSD ($\times$100)	CR ($\times$100)
512	2.15	3.22	2.15	3.22	2.17	3.24	2.17	3.37
1024	3.07	1.62	3.07	1.63	3.02	1.62	3.05	1.77
2048	4.30	0.82	4.30	0.81	4.28	0.82	4.31	0.96
4096	6.07	0.41	6.07	0.41	6.08	0.42	6.08	0.56
8192	8.59	0.21	8.59	0.21	8.62	0.21	8.59	0.35

CR. Let M_c be the number of collided flows, namely, the number of distinct flows that share bucket indexes and fingerprints with other flows. The CR is defined as $\mathrm{CR} = \frac{M_c}{M}$.

PR. Precision is defined as $\frac{|\Psi \cap \Omega|}{k}$, where Ψ is the estimated set of the top-k elephant flows, and Ω is the actual set.

ARE. ARE is defined as $\frac{1}{k}\sum_{f_i \in \Psi} \frac{|\hat{n}_i - n_i|}{n_i}$, where $\hat{n}_i$ is the estimated size of flow f_i, and n_i is the actual flow size. ARE evaluates the error rate of the estimated flow sizes reported by the algorithm.

AAE. AAE is defined as $\frac{1}{k}\sum_{f_i \in \Psi} |\hat{n}_i - n_i|$, which is similar to ARE.

Throughput. We record the total time used to insert all packets and then calculate the throughput. The throughput is defined as $\frac{N}{t}$, where N is the total number of packets, and t is the total time used to insert all packets. We use million of insertions per second (Mps) to measure the throughput.

6.3 Experiments on the disassembling method of the jigsaw storage scheme

We argue that the disassembling method of our jigsaw storage scheme can generate well-randomized bucket indexes and fingerprints. To prove this, we compare our method with three typical Hash functions: Murmur Hash, Bob Hash, and BKDR Hash. All the compared Hash functions output 32-bit Hash values. Then, we calculate the bucket index b and fingerprint $\mathbb{F}$ as follows: $b = h \bmod w$, $\mathbb{F} = \lfloor h/w \rfloor \bmod 2^{l'}$, where h is the Hash value, w is the number of buckets, and l' is the bit length of the fingerprint.

To evaluate the performance of generating bucket indexes and fingerprints, we first employ the RSD of the number of distinct flows mapped to each bucket as the metric, which shows the uniformity of the distribution of the generated bucket indexes. Then, we compare the CR to evaluate the randomness of fingerprints. When comparing the metrics, we set the fingerprint bit length l' to 16 and vary the bucket number w from 512 to 8192. Each experiment is repeated ten times with different Hash seeds or modular inverses, and the final results are the average values. The Hash seeds of Murmur Hash and Bob Hash, and the modular inverses of our methods are randomly generated. Since the Hash seed of BKDR Hash cannot be set randomly, we set it to the recommended particular values like 31, 131, 1313, and 13131. We employ both datasets to perform the experiments. Due to the space limitation, we omit the results on CAIDA-2019 since they are similar to that on CAIDA-2016.

As shown in Table 1, the RSD and CR of our method are very close to that of the compared Hash functions. Besides, we can find that the CR of our method is even lower than BKDR Hash, since BKDR Hash has poor performance when its Hash seed is set to 31. The results indicate that our method can generate well-randomized bucket indexes and fingerprints as typical Hash functions.

6.4 Experiments on parameter settings

In this subsection, we analyze our parameter settings. Jigsaw-Sketch has three key parameters: the number of buckets w, the number of cells λ in each bucket, and the number of heavy cells λ_h in each bucket. Given a fixed total memory, w is decided by λ and λ_h. Thus, we only conduct experiments on λ and λ_h. Since we can get similar conclusions on both datasets, we only show the results on CAIDA-2016.

Experiments on λ. In this part, we set λ to 4, 8, and 16 to compare the precision and throughput on finding top-1000 elephant flows. λ_h is set to the value that achieves the highest precision. As shown in Figure 2, the precision of Jigsaw-Sketch increases as λ increases. However, a large value of λ leads to a longer bucket, and we need to access memory more times when traversing the bucket. Therefore, the throughput decreases as λ increases. In order to achieve both high precision and high throughput, we select the intermediate value 8 as the default value of λ.

Experiments on λ_h. In this part, we set λ to 8, and compare the precision on finding top-1000 elephant flows for different values of λ_h. As shown in Figure 3(a), Jigsaw-Sketch can achieve the highest

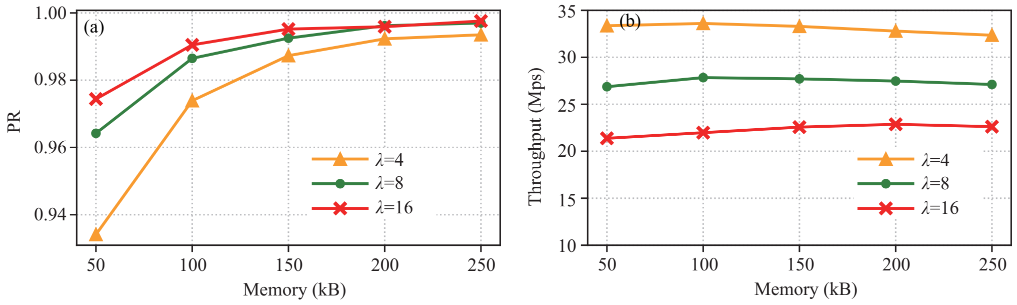

Figure 2 PR and throughput with different values of λ when $k = 1000$. (a) PR; (b) throughput.

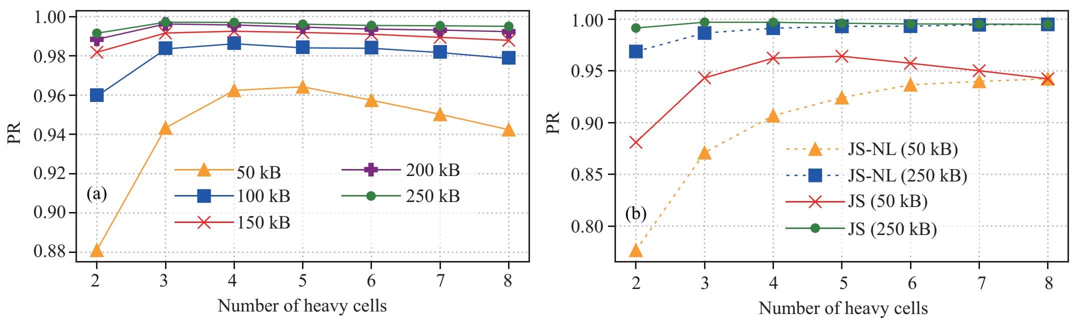

Figure 3 Impact of λ_h when $k = 1000$. (a) PR of JS; (b) PR comparison between JS and JS-NL.

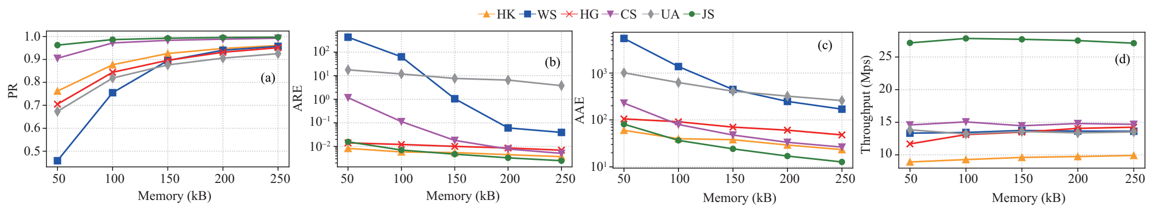

Figure 4 Metrics for varying memory size when $k = 1000$ for CAIDA-2016. (a) PR; (b) ARE; (c) AAE; (d) throughput.

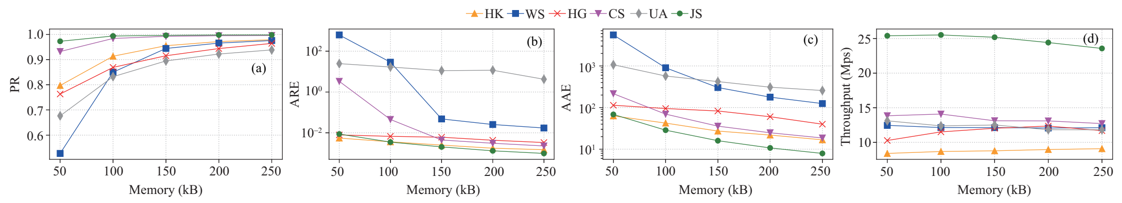

Figure 5 Metrics for varying memory sizes when $k = 1000$ for CAIDA-2019. (a) PR; (b) ARE; (c) AAE; (d) throughput.

precision when λ_h is around half of λ. Besides, we can find that the value of λ_h that achieves the highest precision tends to decrease with the increase of memory size. This situation also happens when λ is 4 or 16. This is because a larger memory size can allocate more buckets, and fewer heavy cells can still contain all the top-k elephant flows mapped in each bucket. Considering the precision under limited memory, we set λ_h to 4 as the default value when $\lambda = 8$.

Remember that each bucket of Jigsaw-Sketch also contains light cells. In order to explain the effectiveness of light cells, we compare two types of Jigsaw-Sketch: Jigsaw-Sketch with light cells (JS) and Jigsaw-Sketch with no light cells (JS-NL). Since the situations are similar with different memory, we only show the results when the memory size is 50 and 250 kB. As shown in Figure 3(b), JS achieves higher precision than JS-NL. For JS-NL, it performs better when it sets more cells in each bucket since it can simultaneously record more flows' sizes. However, most cells record the sizes of mouse flows, and storing the residual part for them is unnecessary. Therefore, replacing some heavy cells with light cells can improve memory efficiency and precision.

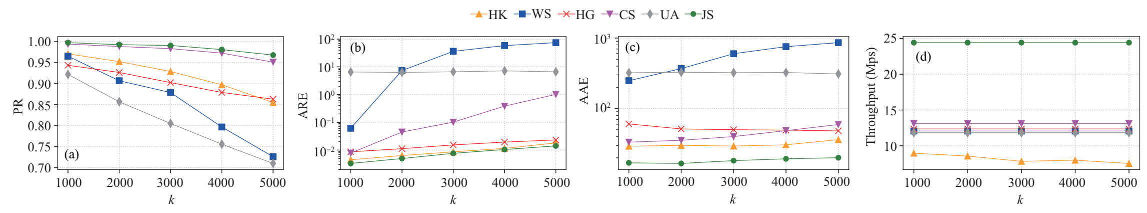

Figure 6 Metrics for varying k when memory size is 200 kB for CAIDA-2016. (a) PR; (b) ARE; (c) AAE; (d) throughput.

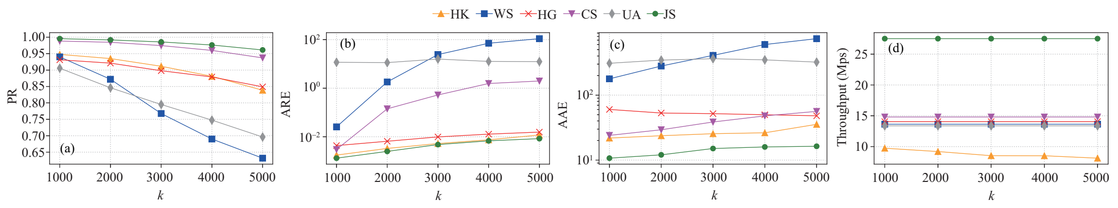

Figure 7 Metrics for varying k when memory size is 200 kB for CAIDA-2019. (a) PR; (b) ARE; (c) AAE; (d) throughput.

6.5 Experiments on precision

In this part, we compare the precision for varying memory size and k. For experiments of varying memory size, we set $k = 1000$ and vary memory size from 50 to 250 kB. We do not set a larger memory size since 250 kB is enough for all the compared algorithms to get high precision. For experiments of varying k, we set the memory size to 200 kB and vary k from 1000 to 5000. We argue that our algorithm outperforms the other methods since it can achieve higher precision while using less memory.

PR vs. memory size. As shown in Figures 4(a) and 5(a), the precision of JS is always the highest, and ChainSketch is the next. For the CAIDA-2016 dataset, the highest precisions of other baselines are 0.762 and 0.961 when the memory size is 50 and 250 kB, separately. In contrast, JS can achieve a precision of 0.962 when the memory size is just 50 kB. This result also outperforms the 0.905 precision of ChainSketch. Besides, when the memory size is 250 kB, the precision of JS reaches 0.997. The gap between the precision values gets smaller as memory increases. However, it is worth noting that it is hard to improve the precision when the precision is already extremely close to 1. Similarly, for the CAIDA-2019 dataset, the precision of JS reaches 0.973 and 0.998 when the memory is 50 and 250 kB separately, significantly outperforming the baselines. These results imply that JS performs much better than the baselines from the view of precision when using the same memory.

PR vs. k. As shown in Figures 6(a) and 7(a), as k increase from 1000 to 5000, the precision of JS decreases from 0.996 to 0.961 for the CAIDA-2016 dataset, and decreases from 0.998 to 0.968 for the CAIDA-2019 dataset. In other words, the precision of JS is always higher than 0.96. The gap between JS and the baselines gets larger as k increases, and JS can improve the precision by at most 0.330 compared to the baselines. This demonstrates that our algorithm can capture the larger flows more accurately from the packet stream.

6.6 Experiments on ARE and AAE

In this part, we focus on the ARE and AAE of the estimated sizes of the reported top-k elephant flows. We also conduct experiments with varying memory sizes and k, and the parameter settings are the same as that in Subsection 6.5.

ARE vs. memory size. As shown in Figures 4(b) and 5(b), the AREs of JS, HK, and HG are very close. This is because the AREs of them are already very small, which are always smaller than 0.02. Specifically, as the memory increases, the AREs of JS, HK, and HG separately decreases from 0.016, 0.009, 0.014 to 0.002, 0.004, 0.007 for the CAIDA-2016 dataset, and decreases from 0.009, 0.005, 0.008 to 0.001, 0.002, 0.003 for the CAIDA-2019 dataset. Apparently, such AREs are small enough to satisfy most applications' requirements. Though CS also has low AREs when the memory size is sufficient, its ARE is much larger when the memory size is not more than 100 kB. This is because there are always some mouse flows misclassified as elephant flows in CS when the memory is tiny.

AAE vs. memory size. As shown in Figures 4(c) and 5(c), when the memory is larger than 100 kB, the AAEs of JS, HK, HG, and CS are all smaller than 100, and JS achieves the smallest AAE among all the methods. As the memory size increases, the AAE of JS decreases from 81.14 to 12.72 for the CAIDA-2016 dataset, and decreases from 68.28 to 7.90 for the CAIDA-2019 dataset. We can find that HG has a larger AAE than HK, though they employ the same exponential decay strategy. One reason is that HK only records the IDs of k flows and can allocate more memory for capturing the elephant flows and estimating their sizes.

ARE vs. k. As shown in Figures 6(b) and 7(b), as k increases from 1000 to 5000, the ARE of JS increases from 0.003 to 0.014 for the CAIDA-2016 dataset, and increases from 0.001 and 0.008 for the CAIDA-2019 dataset, which is always the smallest among all the methods. For CS, the situations that misclassify mouse flows as elephant flows occur more frequently when k increases, significantly increasing its ARE. For HK, the memory consumed by the min-heap and Hash table increases with k. In other words, the memory allocated for buckets to capture elephant flows decreases, which leads to the increase of its ARE.

AAE vs. k. As shown in Figures 6(c) and 7(c), JS achieves the smallest AAE among all the methods. As k varies from 1000 to 5000, the AAE of JS is between 16.65 and 20.08 for the CAIDA-2016 dataset, and between 10.71 and 16.32 for the CAIDA-2019 dataset. In contrast, the AAEs of the baselines are higher than 22.83. These results indicate the accuracy of our algorithm.

6.7 Experiments on throughput

In this part, we compare Jigsaw-Sketch and the related work from the view of throughput. We first vary the memory size from 50 to 250 kB to compare the insertion throughput when $k = 1000$. Then, we set the memory size to 200 kB and vary k from 1000 to 5000. We argue that Jigsaw-Sketch is much faster than the baselines.

Throughput vs. memory size. As shown in Figures 4(d) and 5(d), JS is always the fastest among all the methods. For the CAIDA-2016 dataset, the average throughput of JS is 27.46 Mps, which is 2.89, 2.03, 2.06, 2.03, and 1.87 times as large as that of HK, WS, HG, UA, and CS, respectively. For the CAIDA-2019 dataset, the average throughput of JS is 24.81 Mps, which is 2.83, 2.04, 2.14, 2.01, and 1.86 times as large as the baselines. We can find that the throughput of HK is the lowest, though we have added a Hash table to accelerate its lookup process. This is because HK still needs to access discontiguous memory multiple times, limiting its throughput. Since JS, WS, HG, UA, and CS do not need to perform extra memory accesses to find the storage position of a candidate top-k flow, they have higher throughputs. JS is much faster than them since it leverages the advantages of contiguous memory in the "multiple cells in one bucket" strategy and reduces memory accesses by reducing the bit length of each bucket through our two-stage design. We can also find that the throughput of HG is close to WS, though the bucket length in HG is nearly half of that in WS. It is because the exponential decay strategy employed by HG is expensive on computation and consumes more time than WS's strategy.

Throughput vs. k. As shown in Figures 6(d) and 7(d), JS always outperforms the other algorithms when k varies from 1000 to 5000. Specifically, when using the CAIDA-2016 dataset, the average throughput of JS is 27.51 Mps, which is 3.12, 2.02, 1.96, 2.06, and 1.86 times as large as that of HK, WS, HG, UA, and CS, respectively. When using the CAIDA-2019 dataset, the average throughput of JS is 24.42 Mps, which is 2.99, 2.02, 1.98, 2.06, and 1.86 times as large as the baselines. The parameters in JS, WS, HG, UA, and CS do not change when varying k. Thus, their throughputs are unchanged. For HK, increasing k will increase the memory of the min-heap and Hash table, and reduce the buckets in its bucket arrays, slightly decreasing its throughput since the exponential decay operations are performed more frequently.

7 Conclusion

In this paper, we propose a new algorithm for finding top-k elephant flows, called Jigsaw-Sketch, which can simultaneously achieve high accuracy and high packet processing speed. Jigsaw-Sketch is based on a novel two-stage design and a new jigsaw storage scheme, which can significantly reduce memory accesses while improving our algorithm's memory efficiency. In addition, we adopt a lightweight and accurate probabilistic replacement strategy to capture elephant flows, further improving the accuracy and speed of our algorithm. We implement Jigsaw-Sketch on both CPU and FPGA platforms. The experimental results show that Jigsaw-Sketch can always achieve the highest accuracy and speed compared to the state-of-the-art.

Acknowledgements This work was supported in part by National Natural Science Foundation of China (Grant Nos. 62332013, 62072322, U20A20182, 62202322), Natural Science Foundation of Jiangsu Province (Grant No. BK20210706), and Jiangsu Planned Projects for Postdoctoral Research Funds (Grant No. 2021K165B).

References

1 Sivaraman A, Subramanian S, Alizadeh M, et al. Programmable packet scheduling at line rate. In: Proceedings of ACM SIGCOMM Conference, Florianopolis, 2016. 44–57

2 Huang H, Sun Y E, Chen S, et al. You can drop but you can't hide: k-persistent spread estimation in high-speed networks. In: Proceedings of IEEE Conference on Computer Communications (INFOCOM), Honolulu, 2018. 1889–1897

3 Rottenstreich O, Tapolcai J. Optimal rule caching and lossy compression for longest prefix matching. IEEE ACM Trans Netw, 2016, 25: 864–878

4 Yu M, Jose L, Miao R. Software defined traffic measurement with OpenSketch. In: Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI), Lombard, 2013. 29–42

5 Sun Y E, Huang H, Ma C, et al. Online spread estimation with non-duplicate sampling. In: Proceedings of IEEE Conference on Computer Communications (INFOCOM), Toronto, 2020. 2440–2448

6 Zhou Y, Zhou Y, Chen S, et al. Highly compact virtual active counters for per-flow traffic measurement. In: Proceedings of IEEE Conference on Computer Communications (INFOCOM), Honolulu, 2018. 1–9

7 Zhao Y, Yang K, Liu Z, et al. LightGuardian: a full-visibility, lightweight, in-band telemetry system using sketchlets. In: Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2021. 991–1010

8 Du Y, Huang H, Sun Y E, et al. Self-adaptive sampling for network traffic measurement. In: Proceedings of IEEE Conference on Computer Communications (INFOCOM), Vancouver, 2021. 1–10

9 Yang T, Zhang H, Li J, et al. HeavyKeeper: an accurate algorithm for finding top-k elephant flows. IEEE ACM Trans Netw, 2019, 27: 1845–1858

10 Ilyas I F, Beskales G, Soliman M A. A survey of top-k query processing techniques in relational database systems. ACM Comput Surv, 2008, 40: 1–58

11 Soliman M A, Ilyas I F, Chang K C C. Probabilistic top-k and ranking-aggregate queries. ACM Trans Database Syst, 2008, 33: 1–54

12 Cheung Y L, Fu A W C. Mining frequent itemsets without support threshold: with and without item constraints. IEEE Trans Knowl Data Eng, 2004, 16: 1052–1069

13 Alsaudi A, Altowim Y, Mehrotra S, et al. TQEL: framework for query-driven linking of top-k entities in social media blogs. Proc VLDB Endow, 2021, 14: 2642–2654

14 Lakhina A, Crovella M, Diot C. Characterization of network-wide anomalies in traffic flows. In: Proceedings of the 4th ACM SIGCOMM Conference on Internet measurement (IMC), Taormina Sicily, 2004. 201–206

15 Zhang Y, Fang B X, Zhang Y Z. Identifying heavy hitters in high-speed network monitoring. Sci China Inf Sci, 2010, 53: 659–676

16 Cormode G, Muthukrishnan S. An improved data stream summary: the count-min sketch and its applications. J Algorithms, 2005, 55: 58–75

17 Lu Y, Montanari A, Prabhakar B, et al. Counter braids: a novel counter architecture for per-flow measurement. SIGMETRICS Perform Eval Rev, 2008, 36: 121–132

18 Chen M, Chen S, Cai Z. Counter tree: a scalable counter architecture for per-flow traffic measurement. IEEE ACM Trans Netw, 2016, 25: 1249–1262

19 Li H, Chen Q, Zhang Y, et al. Stingy sketch: a sketch framework for accurate and fast frequency estimation. Proc VLDB Endow, 2022, 15: 1426–1438

20 Yang T, Xu J, Liu X, et al. A generic technique for sketches to adapt to different counting ranges. In: Proceedings of IEEE Conference on Computer Communications (INFOCOM), Paris, 2019. 2017–2025

21 Gong J, Yang T, Zhang H, et al. HeavyKeeper: an accurate algorithm for finding top-k elephant flows. In: Proceedings of USENIX Annual Technical Conference (ATC), Boston, 2018. 909–921

22 Yang T, Gong J, Zhang H, et al. HeavyGuardian: separate and guard hot items in data streams. In: Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, London, 2018. 2584–2593

23 Yang T, Jiang J, Liu P, et al. Elastic sketch: adaptive and fast network-wide measurements. In: Proceedings of the ACM SIGCOMM Conference, Budapest, 2018. 561–575

24 Li J, Li Z, Xu Y, et al. Wavingsketch: an unbiased and generic sketch for finding top-k items in data streams. In: Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2020. 1574–1584

25 Metwally A, Agrawal D, Abbadi A E. Efficient computation of frequent and top-k elements in data streams. In: Proceedings of International Conference on Database Theory, Edinburgh, 2005. 398–412

26 Ben-Basat R, Einziger G, Friedman R, et al. Heavy hitters in streams and sliding windows. In: Proceedings of IEEE Conference on Computer Communications (INFOCOM), San Francisco, 2016. 1–9

27 Ting D. Data sketches for disaggregated subset sum and frequent item estimation. In: Proceedings of International Conference on Management of Data (SIGMOD), Houston, 2018. 1129–1140

28 Homem N, Carvalho J P. Finding top-k elements in data streams. Inf Sci, 2010, 180: 4958–4974

29 Charikar M, Chen K, Farach-Colton M. Finding frequent items in data streams. In: Proceedings of International Colloquium on Automata, Languages, and Programming, Malaga, 2002. 693–703

30 Einziger G, Friedman R. Counting with tinytable: every bit counts! In: Proceedings of International Conference on Distributed Computing and Networking (ICDCN), Singapore, 2016. 1–10

31 Yu X, Xu H, Yao D, et al. CountMax: a lightweight and cooperative sketch measurement for software-defined networks. IEEE ACM Trans Netw, 2018, 26: 2774–2786

32 Tang L, Huang Q, Lee P P C. MV-Sketch: a fast and compact invertible sketch for heavy flow detection in network data streams. In: Proceedings of IEEE Conference on Computer Communications (INFOCOM), Paris, 2019. 2026–2034

33 Zhang Y, Liu Z, Wang R, et al. CocoSketch: high-performance sketch-based measurement over arbitrary partial key query. In: Proceedings of the ACM SIGCOMM Conference, 2021. 207–222

34 Ye J, Li L, Zhang W, et al. UA-Sketch: an accurate approach to detect heavy flow based on uninterrupted arrival. In: Proceedings of the 51st International Conference on Parallel Processing, Bordeaux, 2022. 1–11

35 Huang J, Zhang W, Li Y, et al. ChainSketch: an efficient and accurate sketch for heavy flow detection. IEEE ACM Trans Netw, 2022, 31: 738–753