MEC-Sketch: Memory-Efficient Per-Flow Cardinality Measurement in High-Speed Networks

Kejun Guo†, Fuliang Li†✉, Yunjie Zhang†, Haorui Wan†, Jiaxing Shen‡, Xingwei Wang†
†Northeastern University, Shenyang 110819, China, ‡Lingnan University, Hong Kong
Email: kejunguo@163.com, lifuliang@cse.neu.edu.cn, 2472095@stu.neu.edu.cn,
2301947@stu.neu.edu.cn, jiaxingshen@LN.edu.hk, wangxw@mail.neu.edu.cn

***Abstract*—Per-Flow cardinality measurement in high-speed networks is essential for network security and traffic analysis applications. Flow cardinality refers to the number of distinct elements within a flow, such as the number of unique destination IPs associated with a given source IP. While extensive research has been conducted on single-flow cardinality estimation, achieving accurate per-flow cardinality measurement with real-time performance and low memory overhead remains challenging in large-scale network environments, particularly given the highly skewed distribution of flow cardinalities where mouse flows with smaller cardinalities dominate, and elephant flows with larger cardinalities are fewer. This paper introduces MEC-Sketch, a memory-efficient cardinality estimation data structure that leverages the inherently skewed distribution of flow cardinalities in network traffic. MEC-Sketch employs a dual-component architecture: a heavy part utilizing a majority vote algorithm for precise super-spreader detection, and a light part implementing compact cardinality estimators for memory-efficient measurement of mouse flows. We address two fundamental technical challenges: (1) adapting the majority vote algorithms to operate with cardinality estimators that lack native support for real-time queries, and (2) implementing an effective mapping strategy between large estimators in the heavy part and small estimators in the light part during elephant-mouse flow separation. Comprehensive evaluations on real-world network traces demonstrate that MEC-Sketch significantly outperforms state-of-the-art solutions in terms of estimation accuracy, memory efficiency, and computational performance for both cardinality estimation and super-spreader detection tasks.**

***Index Terms*—sketch, cardinality estimation, super-spreader detection, network measurement.**

I. INTRODUCTION

A. Background and Motivation

Per-flow cardinality measurement is critical for various network applications in high-speed networks, including detection of DDoS attacks [1]–[5], P2P hot-spot localization [6], port scanning measurement [7], and worm propagation detection [8], [9]. Each flow can be viewed as a pair $\langle f, e\rangle$, where f is the flow key and e denotes the element of interest. The cardinality of a flow is defined as the number of distinct e corresponding to f. For example, we can treat the source IP as the flow key and the destination IP as the element. The cardinality is defined as the number of unique destination IPs associated with each source IP.

Due to their memory efficiency and constant-time updates, sketch-based per-flow cardinality measurement solutions have been widely adopted. Sketch-based solutions map flows to buckets, each of which contains a cardinality estimator (Linear Counting [10], HyperLogLog [11], or Multiresolution Bitmap [12]). Due to hash collisions, multiple flows may be mapped to the same bucket, introducing estimation errors. Nevertheless, sketch-based solutions achieve high memory efficiency and constant-time updates.

Although existing sketch-based solutions partially alleviate the challenges in this domain, their accuracy, memory efficiency, and real-time performance still fall short of expectations. Per-flow cardinality measurement involves two fundamental tasks: cardinality estimation and super-spreader detection. Cardinality estimation aims to measure the cardinality of all flows, whereas super-spreader detection focuses on identifying flows whose cardinalities exceed a given threshold. Sketches designed for cardinality estimation can often be extended to support super-spreader detection—typically by maintaining a min-heap or similar structure—but they generally exhibit lower memory efficiency and throughput compared to sketches specifically designed for super-spreader detection, which avoid tracking all flows. Therefore, in the following, we use state-of-the-art solutions for cardinality estimation and super-spreader detection to illustrate the limitations of existing sketch-based per-flow cardinality measurement solutions, and introduce our solution accordingly.

Couper [13] is the state-of-the-art solution for cardinality estimation. Given the skewed distribution of flow cardinalities in network traffic, Couper employs a two-layer sketch design, where mouse flows are confined to the first layer and elephant flows overflow into the second layer. It is worth noting that, in frequency-related tasks, elephant flows refer to flows containing a large number of packets. In cardinality-related tasks, elephant flows refer to flows with high cardinality. However, this design faces two key limitations: (1) each insertion requires scanning dozens of bits in the first-layer estimator to determine whether the flow should be promoted, compromising real-time performance; and (2) since elephant flows must also pass through the first layer, hash collisions between elephant and mouse flows are highly likely, resulting in overestimation of mouse flows. Although Couper can be extended to support super-spreader detection by maintaining an additional bucket table, its accuracy, memory efficiency, and throughput are inferior to NDS [14], a solution specifically designed for super-spreader detection.

NDS [14] is the state-of-the-art solution for super-spreader detection. NDS modifies HLL into a non-duplicate sampler, which outputs elements with a varying probability p when they first appear. This allows it to maintain a simple counter to measure flow cardinality, incrementing the counter by $1/p$ every time a successful sample is taken. Additionally, it uses exponential decay strategy to handle hash collisions. However, NDS suffers from two main limitations: (1) due to its sampling nature, it requires processing a large number of elements to achieve convergence; and (2) its low throughput makes it unsuitable for high-speed networks.

An ideal per-flow cardinality measurement solution should support both cardinality estimation and super-spreader detection without compromising the performance of either task. Specifically, compared to state-of-the-art cardinality estimation solutions, it should offer higher throughput, memory efficiency, and estimation accuracy; and compared to state-of-the-art super-spreader detection solutions, it should achieve higher throughput and detection accuracy.

To simultaneously support both cardinality estimation and super-spreader detection without compromising the performance of either task, we propose a dual-component architecture: a heavy part utilizing a majority vote algorithm for precise super-spreader detection, and a light part implementing compact cardinality estimators for memory-efficient measurement of mouse flows. In contrast to Couper, which restricts mouse flows to the first layer and gradually promotes elephant flows to the second layer, our design prioritizes the identification of elephant flows in the heavy part and evicting mouse flows to the light part. This design offers two key advantages: (1) prioritizing elephant flows ensures high throughput for both cardinality estimation and super-spreader detection, avoiding the throughput degradation seen in Couper due to its mouse-first design; and (2) separating elephant and mouse flows minimizes hash collisions between them, thereby reducing the overestimation of mouse flows and improving the accuracy of cardinality estimation. In addition, compared to NDS: (1) our majority voting-based super-spreader detection avoids the need for a large number of traffic to achieve convergence, thereby ensuring high detection accuracy; and (2) we introduce a novel approximate cardinality estimation technique to further enhance the throughput of super-spreader detection. In summary, our solution employs a single sketch to support both cardinality estimation and super-spreader detection without compromising the performance of either task. It achieves higher throughput, memory efficiency, and estimation accuracy than state-of-the-art cardinality estimation solutions, as well as higher throughput and detection accuracy than state-of-the-art super-spreader detection solutions.

B. Proposed Solution and Contributions

Contribution I: We design MEC-Sketch, a novel memory-efficient cardinality estimation sketch. MEC-Sketch comprises two components: a heavy part and a light part. The heavy part employs the majority vote algorithm to enable efficient super-spreader detection, while the light part utilizes compact cardinality estimators (e.g., Linear Counting with a few dozen bits) to achieve memory-efficient cardinality estimation. MEC-Sketch also separates elephant flows from mouse flows to prevent the overestimation of mouse flows.

Contribution II: We address key challenges in transitioning from counter-based to estimator-based designs. First, we observe that the cardinality estimate of the cardinality estimator is proportional to the sum of its register values. Since cardinality estimators do not inherently support real-time queries, we approximate their estimates by dynamically accumulating register values, ensuring compatibility with the majority vote algorithm. Second, during the separation of elephant and mouse flows, we effectively map the large cardinality estimators in the heavy part to smaller ones in the light part by using a shared set of hash functions and configuring the array length of each estimator in the heavy part as an integer multiple of those in the light part. Finally, during the query phase, we enhance estimation accuracy through an *AND*-based noise elimination strategy.

Contribution III: We conduct extensive experiments on two real-world network traces to evaluate MEC-Sketch. Experimental results demonstrate that MEC-Sketch surpasses state-of-the-art sketches in both cardinality estimation and super-spreader detection accuracy. Additionally, MEC-Sketch achieves higher throughput and faster query times compared to most existing solutions.

II. Related work

A. Sketch

Sketch is a probabilistic data structure that employs hash functions to map flows into buckets. To prevent reaching its capacity limit, sketch is reset at the end of each period. Existing sketches support various network measurement tasks and can be broadly classified into two categories: frequency-related and cardinality-related. The two primary frequency-related tasks are frequency estimation and heavy-hitter detection. Frequency estimation determines the number of packets in a flow, with representative sketches including CM Sketch [15], Count Sketch [16], and so on. To enhance memory efficiency and accuracy,some frequency estimation sketches separate elephant and mouse flows, such as Cold Filter [17], Elastic Sketch [18], and so on [19], [20]. Heavy-hitter detection identifies flows exceeding a predefined threshold. Notable sketches for this task include MV Sketch [21], Elastic Sketch, HeavyKeeper [22] and so on [23], [24]. MV Sketch and Elastic Sketch employ the majority vote algorithm to improve detection accuracy. Cardinality-related sketches are discussed in Section II-B.

B. Cardinality Estimation and Super-Spreader Detection

Sketch-based cardinality estimation solutions [13], [25]–[29] are mostly variants of frequency-related sketch. gSkt [27] is a cardinality estimation variant of CM Sketch, performing k independent estimations and returning the minimum value. rerSkt [29] follows the strategy similar to Count Sketch [16], dividing background traffic into primary and secondary estimators, with the secondary estimator used to subtract errors

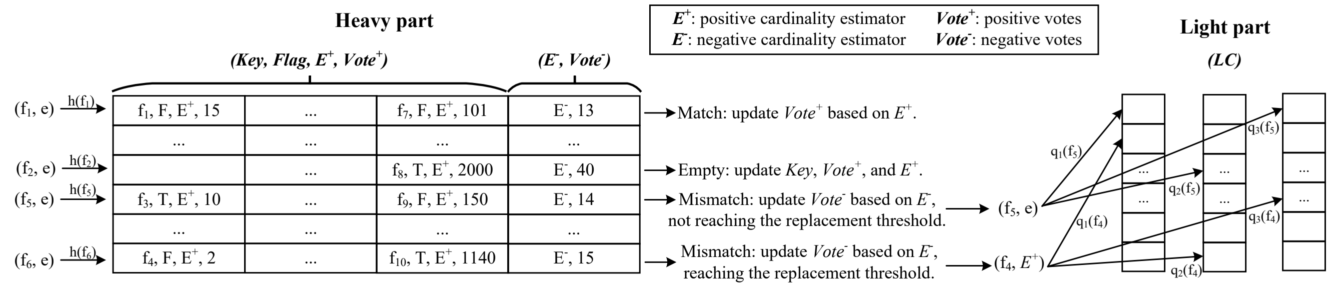


Fig. 1: Overview of MEC-Sketch. The heavy part employs the majority vote algorithm to separate elephant flows from mouse flows, enabling efficient super-spreader detection. The light part utilizes compact cardinality estimators (e.g., Linear Counting with a few dozen bits) to achieve memory-efficient cardinality estimation.

from the primary one. Couper [13] is the state-of-the-art solution for cardinality estimation and can be viewed as a variant of Cold Filter [17]. Given the skewed distribution of flow cardinalities in network traffic, Couper employs a two-layer sketch design, where mouse flows are confined to the first layer and elephant flows overflow into the second layer. This approach effectively separates elephant and mouse flows, enhancing memory efficiency.

Similarly, sketch-based super-spreader detection solutions [14], [30]–[38] are mostly variants of frequency-related sketch. SpreadSketch [38] adapts CM Sketch and uses a probabilistic approach to record super-spreaders. Similar to the overestimation in CM Sketch, SpreadSketch also suffers from significant estimation errors. The state-of-the-art solution, NDS [14] modifies HLL into a non-duplicate sampler, which filters out duplicate elements and outputs them with a varying probability p when they first appear. This allows it to maintain a simple counter to measure flow cardinality, incrementing the counter by $1/p$ every time a successful sample is taken. Additionally, it uses exponential decay strategy to handle hash collisions. However, NDS requires processing a large number of elements to achieve convergence. There are other super-spreader detection sketches [30]–[37], but as described in the NDS, their performance is inferior to NDS.

C. Single-Flow Cardinality Estimator

HyperLogLog (HLL) [11] maintains an array A of m registers and is associated with two hash functions $h_1()$ and $h_2()$. When inserting an element e, HLL uses $h_1(e)$ to determine the target register $A[h_1(e)]$. Then, HLL uses $h_2(e)$ to generate a bit string and calculates the number of leading zeros plus one, denoted as z. Subsequently, HLL updates $A[h_1(e)] = \max(A[h_1(e)], z)$. During querying, HLL estimates the cardinality using the following formula: $\hat{c} = \alpha_m \times m^2 \left(\sum_{i=0}^{m-1} 2^{-A[i]}\right)^{-1}$, where α_m is a constant defined as $\alpha_m = \frac{0.7213}{1+\frac{1.079}{m}}$. If $\hat{c}$ is found to be less than $\frac{5}{2}m$, HLL treats register array as a bitmap and employs LC to estimate result.

Due to space constraints, we will not provide a detailed introduction of other single-flow cardinality estimators [10], [12]. Further details can be found in their respective papers.

III. MEC-Sketch

A. Overview of MEC-Sketch

As shown in Fig. 1, MEC-Sketch consists of two components: a heavy part and a light part. To demonstrate its operation, we consider the insertion and query process of the flow $\langle f, e\rangle$, where f is the flow key and e denotes the element of interest, such as $\langle srcIP, dstIP\rangle$.

When inserting $\langle f, e\rangle$, MEC-Sketch first attempts to insert the flow in the heavy part. If the flow key matches or an empty slot is available, MEC-Sketch directly inserts $\langle f, e\rangle$ and increments the positive votes $Vote^+$ based on the positive cardinality estimator E^+. Otherwise, MEC-Sketch increments the negative votes $Vote^-$ based on the negative cardinality estimator E^- and checks whether the ratio of $Vote^-$ to the minimum $Vote^+$ exceeds the eviction threshold λ. If λ is not reached, MEC-Sketch inserts $\langle f, e\rangle$ into the light part. Otherwise, MEC-Sketch replaces the flow with the minimum $Vote^+$ with $\langle f, e\rangle$, sets the replacement flag $Flag$ to $True$, and moves the evicted flow to the light part.

When querying a flow with flow key f, MEC-Sketch first searches in the heavy part. If a matching flow key is found and the replacement flag $Flag$ is $False$, it indicates that MEC-Sketch records the accurate cardinality. If $Flag$ is $True$, MEC-Sketch performs queries in both the heavy and light parts and mitigates overestimation errors using the algorithm described in Section III-D. If no match is found in the heavy part, MEC-Sketch performs the query in the light part and returns its cardinality.

In the following sections, we will elaborate on the design of MEC-sketch progressively and in detail from three versions: the parallel version, the minimal version, and the cardinality estimation version. The parallel version supports only super-spreader detection and utilizes parallel processing; the minimal version also supports only super-spreader detection but sacrifices parallelism to improve accuracy; the cardinality estimation version supports both cardinality estimation and super-spreader detection. We will begin with the simplest parallel version, followed by the minimal version, and finally, the cardinality estimation version as shown in Fig. 1.

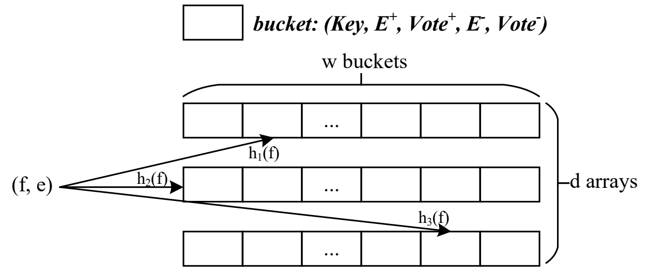


Fig. 2: The data structure of MEC-Sketch (Parallel Version).

B. MEC-Sketch for Super-Spreader Detection: Parallel Version

One of the design goals of MEC-Sketch is to identify super-spreaders, which are elephant flows with large cardinalities. Due to hash collisions, multiple flows may be mapped to the same bucket, necessitating an approach that selectively retains only elephant flows. To achieve this, MEC-Sketch employs the majority vote algorithm [18], [21]. Specifically, when the incoming flow is not found in the bucket, MEC-Sketch increments the negative vote $Vote^-$. If the threshold λ is reached, the existing flow is replaced with the new flow. This mechanism ensures that mouse flows are quickly evicted, while elephant flows remain stable in the buckets.

Insight: The majority vote algorithm is effective in frequency-related sketches due to real-time counter reading. However, it is not directly applicable to cardinality estimators. For instance, in a typical HLL configuration with 128 registers, each 5 bits in size, estimating cardinality requires scanning all 128 registers, significantly impacting real-time performance. Upon analyzing the classical cardinality estimators in Section II-C, we observe that the sum of register values in their arrays is proportional to the estimated cardinality. This sum can be accumulated in real time, enabling an efficient alternative. To leverage this property, we introduce an additional counter to track the sum of register values, using it as a proxy for cardinality in the majority vote algorithm. The storage overhead of a single 16-bit counter is negligible compared to that of a full cardinality estimator (e.g., an HLL with 128 registers of 5 bits each) while ensuring real-time efficiency. Specifically, in LC and MRB, a higher proportion of ones indicates a larger cardinality. Thus, we maintain an additional counter $Vote$, incrementing it whenever a mapped bit is zero, i.e., $Vote = Vote + 1$. Although the cardinality of MRB depends on its highest occupied level, in practice, we found that counting the number of ones often yields a good approximation to the cardinality and produces favorable experimental results. In HLL, a larger sum of register values corresponds to a greater cardinality. Therefore, we maintain an additional counter $Vote$ as well. If the value stored in a register is less than the inserted value by N, we update the counter as $Vote = Vote + N$.

Data Structure: As shown in Fig. 2, MEC-Sketch comprises d arrays, each containing w buckets. Each bucket consists of five fields: flow key, positive cardinality estimator, positive votes, negative cardinality estimator, and negative votes. Each array is associated with two hash functions, $h_i()$ and $g()$. Given a flow $\langle f, e\rangle$, MEC-Sketch maps f to a bucket using $h_i(f)$ and maps e to the corresponding register or bit in the cardinality estimator using $g(f, e)$. For convenience, we denote the j-th bucket in the i-th array as $A[i][j]$, with its respective fields represented as $A[i][j].Key, A[i][j].E^+, A[i][j].Vote^+, A[i][j].E^-$, $A[i][j].Vote^-$ $(1 \le i \le d, 0 \le j \le w-1)$.

Insertion: Initially, all fields are empty. Given a flow $\langle f, e\rangle$, MEC-Sketch computes d hash functions to map it to d buckets $A[i][h_i(f)]$ $(1 \le i \le d)$. As shown in Fig. 3, taking HLL as an example, in addition to the aforementioned two sets of hash functions ($h()$ and $g()$), HLL also requires a set of hash functions to compute the number of leading zeros. For convenience, we assume that the number of leading zeros of the flow $\langle f, e\rangle$, plus one, is denoted as lz. MEC-Sketch then processes each mapped bucket according to three distinct cases, as described below.

Case 1: If $A[i][h_i(f)].Key = Null$, the bucket is empty. MEC-Sketch assigns f to the flow key field, sets $A[i][h_i(f)].E^+[g(f, e)] = lz$, and increments $A[i][h_i(f)].Vote^+$ by lz.

Case 2: If $A[i][h_i(f)].Key = f$, the bucket already records the cardinality of flow f. If $lz > A[i][h_i(f)].E^+[g(f, e)]$, MEC-Sketch updates $A[i][h_i(f)].E^+[g(f, e)]$ to lz and adjusts $A[i][h_i(f)].Vote^+$ by adding $lz - A[i][h_i(f)].E^+[g(f, e)]$.

Case 3: If $A[i][h_i(f)].Key \neq f$ and $A[i][h_i(f)].Vote^+ > 0$, the bucket does not store the cardinality of flow f. If $lz > A[i][h_i(f)].E^-[g(f, e)]$, MEC-Sketch updates $A[i][h_i(f)].E^-[g(f, e)]$ to lz and increments $A[i][h_i(f)].Vote^-$ by $lz - A[i][h_i(f)].E^-[g(f, e)]$. If the ratio $A[i][h_i(f)].Vote^-/A[i][h_i(f)].Vote^+ \ge \lambda$, MEC-Sketch resets the bucket and inserts $\langle f, e\rangle$ following ***Case 1***. In other words, when the negative votes exceeds λ times the positive votes corresponding to the flow key stored in the bucket, MEC-Sketch replaces the flow with a new one.

Query: To estimate the cardinality of flow f, MEC-Sketch first computes d hash functions to locate d buckets $A[i][h_i(f)]$ $(1 \le i \le d)$. Among these, it selects the buckets where $A[i][h_i(f)].Key = f$ and estimates their cardinality based on $A[i][h_i(f)].E^+$. The final cardinality estimate is the maximum value across the selected buckets: $\max_{1\le i\le d}\{Est(A[i][h_i(f)].E^+)\}$, where $Est()$ represents the cardinality estimation function of the corresponding estimator.

Analysis: By maintaining a counter ($Vote^+$ or $Vote^-$) for each cardinality estimator, MEC-Sketch enables real-time cardinality estimation. Additionally, the majority vote algorithm facilitates effective super-spreader detection.

Discussion: It is worth noting that this differs from replacing counters with cardinality estimators in MV-Sketch [21]. Lines 6-9 of Alg. 1 in MV-Sketch cannot be implemented with a cardinality estimator. Additionally, we do not emphasize the novelty of majority vote algorithm. The key challenge lies in the fact that cardinality estimators do not support real-time updates like counters.

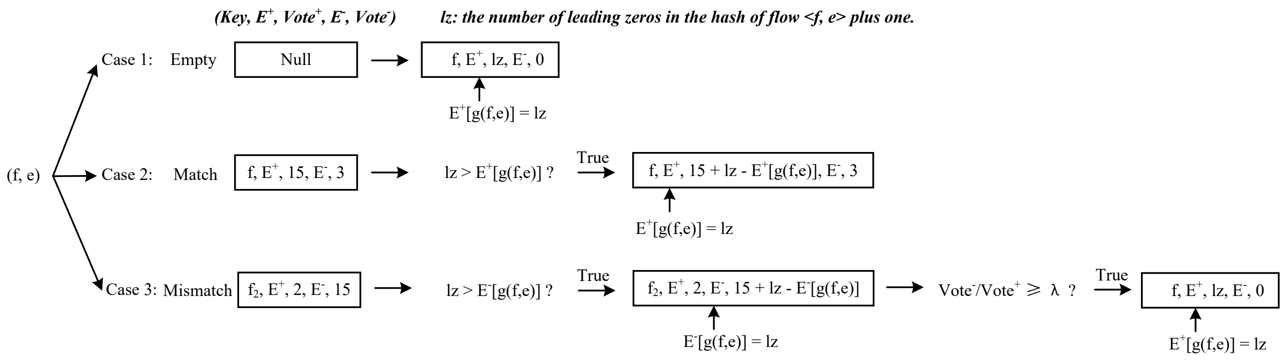


Fig. 3: The main insertion cases of MEC-Sketch (Parallel Version). MEC-Sketch employs the majority vote algorithm to enable efficient super-spreader detection, and approximates their cardinality by dynamically accumulating register values of the cardinality estimator, thereby ensuring the real-time performance of the majority vote algorithm.

C. MEC-Sketch for Super-Spreader Detection: Minimal Version

Insight: We observe that it is unnecessary to maintain a negative cardinality estimator for each positive cardinality estimator. By sacrificing parallelism, accuracy and memory efficiency can be further improved.

Data Structure: As shown in Fig. 4, MEC-Sketch consists of an array of buckets, each containing w slots. Unlike the parallel version, this structure maintains only a negative cardinality estimator and negative votes. This array is associated with two sets of hash functions:$h()$ and $g()$. Given a flow $\langle f, e\rangle$, MEC-Sketch maps f to a bucket using $h(f)$ and maps e to the corresponding register or bit in the cardinality estimator using $g(f, e)$. For convenience, we use $A[i][j]$ to represent the j-th slot in the i-th bucket. The first $w-1$ slots store flow keys, positive cardinality estimators, and positive votes, denoted as $A[i][j].Key$, $A[i][j].E^+$, and $A[i][j].Vote^+$ $(1 \leq i \leq d, 1 \leq j < w-1)$. The last slot of each bucket maintains the negative cardinality estimator and negative votes, represented by $A[i][w-1].E^-$ and $A[i][w-1].Vote^-$, respectively.

Insertion: Initially, all fields are empty. Given a flow $\langle f, e\rangle$, MEC-Sketch uses $h(f)$ to locate the corresponding bucket $A[h(f)]$. Taking HLL as an example, we assume that the number of leading zeros of the flow $\langle f, e\rangle$, plus one, is denoted as lz. MEC-Sketch then processes each mapped bucket according to three distinct cases, as described below.

Case 1: If any of the first $w-1$ slots in $A[h(f)]$ contain the flow key f, MEC-Sketch directly inserts the flow $\langle f, e\rangle$.

Case 2: If none of the first $w-1$ slots contain f, but an empty slot is available, MEC-Sketch inserts $\langle f, e\rangle$ into that slot.

The insertion procedures in ***Case 1*** and ***Case 2*** are identical to those in the parallel version. The key difference lies in ***Case 3***. In ***Case 3***, where MEC-Sketch always attempts to evict the slot with the minimum positive vote.

Case 3: If all $w-1$ slots are occupied and none contain f, MEC-Sketch updates the negative cardinality estimator if $lz > A[h(f)][w-1].E^-[g(f,e)]$, setting $A[h(f)][w-1].E^-[g(f,e)] = lz$ and incrementing the negative votes as

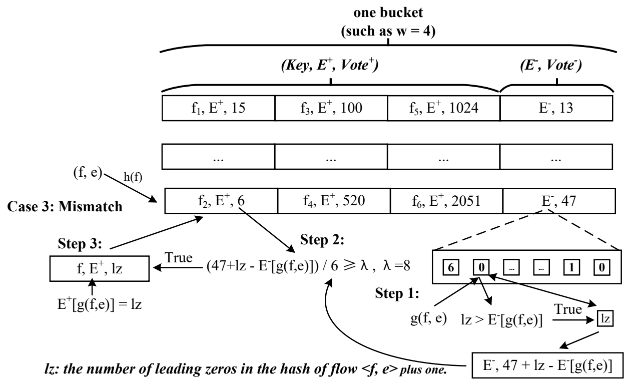
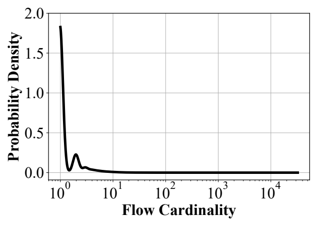


Fig. 4: The data structure and main insertion cases of MEC-Sketch (Minimal Version). MEC-Sketch enhances accuracy and memory efficiency by sacrificing parallelism, specifically by maintaining only a negative cardinality estimator.

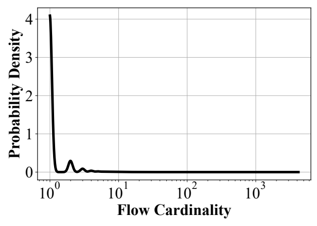
$A[h(f)][w-1].Vote^- += lz - A[h(f)][w-1].E^-[g(f,e)]$. Let j denote the slot with the minimum positive votes. If the eviction condition $A[h(f)][w-1].Vote^-/A[h(f)][j].Vote^+ \geq \lambda$ is met, MEC-Sketch replaces the flow in $A[h(f)][j]$ with $\langle f, e\rangle$ and resets $A[h(f)][w-1]$.

Example: We present an example of ***Case 3***. As shown in Fig. 4, the bucket mapped by flow $\langle f, e\rangle$ does not contain the flow key f. If the number of leading zeros of $\langle f, e\rangle$ plus one, denoted as lz, exceeds the value stored in the HLL register $A[h(f)][w-1].E^-[g(f,e)]$ of the negative cardinality estimator, MEC-Sketch updates this register to lz and sets the negative vote $A[h(f)][w-1].Vote^- = 47 + lz - A[h(f)][w-1].E^-[g(f,e)]$. If the ratio of this updated negative vote $(47 + lz - A[h(f)][w-1].E^-[g(f,e)])$ to the minimum positive vote (6) in the bucket exceeds the threshold ($\lambda = 8$), MEC-Sketch replaces the slot containing the flow with the minimum positive vote with $\langle f, e\rangle$.

Query: To estimate the cardinality of flow f, MEC-Sketch computes $h(f)$ to locate the bucket $A[h(f)]$. If any of the $w-1$ slots contain the same flow key f, the cardinality is



(a) MAWI.



(b) CAIDA.

Fig. 5: The PDF of cardinality distribution in the MAWI and CAIDA datasets. We treat the source IP as the flow key and the destination IP as the element. Mouse flows with smaller cardinalities dominate, while elephant flows with larger cardinalities are fewer.

estimated based on the corresponding cardinality estimator.

Analysis: By sharing a negative cardinality estimator and negative votes among multiple flows, MEC-Sketch significantly improves memory efficiency. Additionally, the presence of multiple slots per bucket enables effective identification of all super-spreaders, even when multiple super-spreaders are mapped to the same bucket.

D. MEC-Sketch for Cardinality Estimation

As shown in Fig. 5, we analyze flow cardinality by considering the number of distinct destination IPs associated with each source IP. The probability density function (PDF) is plotted for 10 million consecutive packets from the MAWI [39] and CAIDA [40] datasets. Mouse flows with smaller cardinalities dominate, while elephant flows with larger cardinalities are fewer. To enhance memory efficiency, a natural approach is to separate elephant and mouse flows. Specifically, compact cardinality estimators (e.g., LC with a few dozen bits) are allocated to the numerous mouse flows, while more complex estimators (e.g., HLL with 128 registers of 5 bits each) are reserved for the relatively few elephant flows.

The primary challenge lies in separating elephant and mouse flows. Existing frequency-related sketches employ two strategies to achieve such separation: (1) restricting mouse flows to the first layer and gradually promoting elephant flows to the second layer; (2) prioritizing the identification of elephant flows in the first layer and evicting mouse flows to the second layer. These approaches also apply to cardinality estimation.

Insight: The state-of-the-art solution, Couper [13], adopts the first strategy but suffers from two key limitations outlined in Section I-A. Therefore, we adopt the second strategy: prioritizing the identification of elephant flows in the first layer and evicting mouse flows to the second layer. Since we have already introduced an efficient method for identifying elephant flows in the previous sections, we now focus on recording mouse flows. However, this introduces an additional challenge: evicting mouse flows to the second layer requires mapping a large cardinality estimator (e.g., an HLL with 128 registers of 5 bits each) to a smaller one (e.g., an LC with a few dozen bits). To address this, MEC-Sketch employs two key strategies. First, the cardinality estimator in the light part utilizes the hash values generated by the hash function $g()$ of the heavy part's estimator. Second, the array length of each cardinality estimator in the heavy part is configured as a factor p of the light part's length. For instance, if the heavy part uses an HLL with 128 registers and the light part employs an LC with 16 bits, the ratio p is $128/16 = 8$. The mapping follows the rule: $LC[i] = 1$ if any non-zero register exists in the range $HLL[i \times 8]$ to $HLL[(i+1) \times 8 - 1]$, where $0 \leq i < 16$. This implies that every group of 8 consecutive HLL registers is mapped to a single bit in the LC. For instance, $HLL[0]$ to $HLL[7]$ correspond to $LC[0]$. An alternative strategy could involve using disproportionate array lengths for the heavy and light parts, applying modulo operations for mapping. However, our solution is computationally simpler and more efficient for real-time applications.

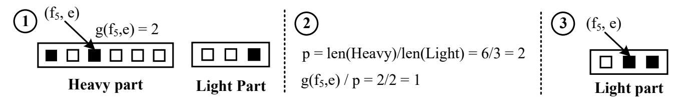
Data Structure: As shown in Fig. 1, MEC-Sketch consists of a heavy part and a light part. It is worth noting that, as discussed in the previous two sections, the light part is unnecessary when MEC-Sketch is used exclusively for super-spreader detection. The heavy part follows the design in Section III-C, with an additional modification: each of the $w-1$ slots in every bucket includes a flag bit, $Flag$, indicating whether eviction has occurred. The array length of each cardinality estimator in the heavy part is set to be p times that of the light part. The light part consists of k arrays, each containing l buckets, where each bucket is an LC with a few dozen bits (e.g., 16 bits). Each array is associated with a hash function $q_i()$ for $1 \leq i \leq k$. Given a flow $\langle f, e\rangle$, MEC-Sketch uses $q_i()$ to map f to an LC bucket, while the hash value from the heavy part's cardinality estimator determines the corresponding bit in the LC. For convenience, $B[i][j]$ denotes the j-th LC bucket in the i-th array of the light part.

Insertion: Initially, all fields are empty. For illustration, we assume that the heavy part employs an HLL with 128 registers and the light part uses an LC with a 16-bit array. The insertion process for the heavy part follows the method in Section III-C and is not repeated here. MEC-Sketch processes each mapped bucket according to three distinct cases, as described below.

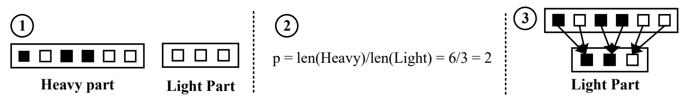
Case 1: If at least one of the $w-1$ slots in the mapped bucket $A[h(f)]$ of the heavy part is empty or matches the flow, MEC-Sketch directly inserts $\langle f, e\rangle$ into that slot.

Case 2: If no matching or empty slots are found, MEC-Sketch inserts $\langle f, e\rangle$ into the negative cardinality estimator $A[h(f)][w-1].E^{-}[g(f,e)]$ in heavy part and checks whether to increment the negative vote $A[h(f)][w-1].Vote^{-}$. If the ratio of the negative vote to the minimum positive vote among the $w-1$ slots does not exceed the threshold λ, MEC-Sketch inserts $\langle f, e\rangle$ into the light part. Specifically, MEC-Sketch uses k hash functions $q_i()$ to locate the mapped bucket $B[q_i(f)]$ in the light part and sets the bit $B[q_i(f)][g(f,e)/p]$ to 1. In practice, p is often configured as a power of two, enabling efficient implementation through bit shifting.

Case 3: Let j denote the slot with the minimum positive vote. If the ratio $A[h(f)][w-1].Vote^{-}/A[h(f)][j].Vote^{+}$ exceeds λ, MEC-Sketch replaces the flow in slot $A[h(f)][j]$



(a) Insertion in the light part when the eviction threshold is not reached.



(b) Insertion in the light part when the eviction threshold is reached.

Fig. 6: The main insertion cases in the light part of MEC-Sketch. MEC-Sketch effectively map the large cardinality estimators in the heavy part to smaller ones in the light part by using a shared set of hash functions and configuring the array length of each estimator in the heavy part as an integer multiple of those in the light part.

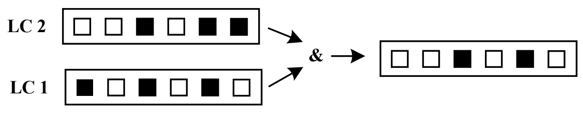


Fig. 7: AND-based noise elimination.

with $\langle f, e \rangle$, sets the eviction flag $A[h(f)][j].Flag$ to $True$, and evicts the replaced flow to the light part. To insert the evicted flow into the light part, MEC-Sketch first uses k hash functions $q_i()$ to locate the mapped bucket $B[q_i(f)]$. Then, MEC-Sketch sets $B[q_i(f)][i] = 1$ if at least one register among $A[h(f)][j].Vote^+[i \times p]$ to $A[h(f)][j].Vote^+[(i+1) \times p]$ is nonzero, where $0 \leq i <$ len(LC).

Example: The operations in the heavy part remain largely unchanged from the minimal version in Section III-C, except for the insertion of evicted flows into the light part. As shown in Fig. 1, MEC-Sketch inserts evicted flows into the light part under two conditions. The first occurs in ***Case 2***, where the flow to be inserted is evicted without reaching the eviction threshold. The second occurs in ***Case 3***, where the flow with the minimum positive vote is evicted upon reaching the eviction threshold. For illustration, consider a cardinality estimator in the heavy part with a register array length of 6 and a corresponding estimator in the light part with a bit array length of 3. The insertion process in these cases is as follows:

(1) As shown in Fig. 6a, when the eviction threshold is not reached, MEC-Sketch uses k hash functions to map the flow to k buckets in light part. Then, MEC-Sketch divides the hash index generated by the heavy part's cardinality estimator by p and uses the quotient as the index for the corresponding bit in the light part's cardinality estimator, setting that bit to 1.

(2) As shown in Fig. 6b, when the eviction threshold is reached, MEC-Sketch maps the large cardinality estimator in the heavy part to the small cardinality estimator in the light part. MEC-Sketch uses k hash functions to map the flow to k buckets in the light part. Then, MEC-Sketch sequentially maps every p consecutive registers in the heavy part's cardinality estimator to a corresponding bit in the light part's estimator. If any of these p registers contain a non-zero value, the corresponding bit in the light part is set to 1.

Query: To estimate the cardinality of flow f, MEC-Sketch first searches in heavy part, which yields three possible cases:

Case 1: If the flow key matches and its eviction flag is $False$, MEC-Sketch directly returns the exact cardinality.

Case 2: If no matching flow key is found in the heavy part, MEC-Sketch queries the light part. It is worth noting that the query in the light part involves a bitwise *AND* operation instead of computing the minimum value. As shown in the Fig. 7, MEC-Sketch applies an *AND* operation across the k LC buckets in the light part before calculating the cardinality. Since all LC buckets across the k arrays of the light part use the same hash function $g()$, the *AND* operation effectively mitigates noise caused by hash collisions.

Case 3: If the flow key matches and its eviction flag is $True$, MEC-Sketch queries both the heavy and light parts. However, their cardinalities cannot be directly summed, as this would lead to overestimation. To illustrate, consider the example in Fig. 8, where four elements of flow f (denoted as C_1) are stored in the heavy part, while three elements (denoted as C_2) are evicted to the light part. A naive summation would yield $|C_1| + |C_2|$, whereas the correct estimate is $|C_1 \cup C_2| = |C_1| + |C_2| - |C_1 \cap C_2|$. Therefore, the key challenge is to determine $|C_1 \cap C_2|$. Leveraging the approach used for mapping a large cardinality estimator to a smaller one, MEC-Sketch maps the heavy part's cardinality estimator to a temporary estimator equivalent to those in the light part. It then applies a bitwise *AND* operation between this temporary estimator and the light part's estimator to approximate $|C_1 \cap C_2|$. Finally, MEC-Sketch computes $|C_1 \cup C_2| = |C_1| + |C_2| - |C_1 \cap C_2|$, ensuring an accurate estimation that accounts for flows marked with an eviction flag in the heavy part.

Theoretical Bounds: Due to the small-scale cardinality estimator (16-bit LC), the error bound in light part is minimal. The error bound of heavy part is comparable to that of Elastic-Sketch [18], with the difference being the substitution of flow frequency with flow cardinality.

Analysis: MEC-Sketch enhances memory efficiency by separating elephant and mouse flows. Compared to the state-of-the-art cardinality estimation solution, Couper [13], MEC-Sketch prioritizes the identification of elephant flows, mitigating the overestimation of mouse flows while maintaining real-time performance. Additionally, MEC-Sketch introduces an efficient strategy for mapping large cardinality estimators to smaller ones. Furthermore, it reduces cardinality overestimation during queries by effectively eliminating noise.

E. Comparison with Elastic Sketch

While some may perceive similarities between MEC-Sketch and Elastic Sketch [18], they differ fundamentally in the following aspects:

(1) Objective: MEC-Sketch is designed to address cardinality estimation and super-spreader detection, whereas Elastic

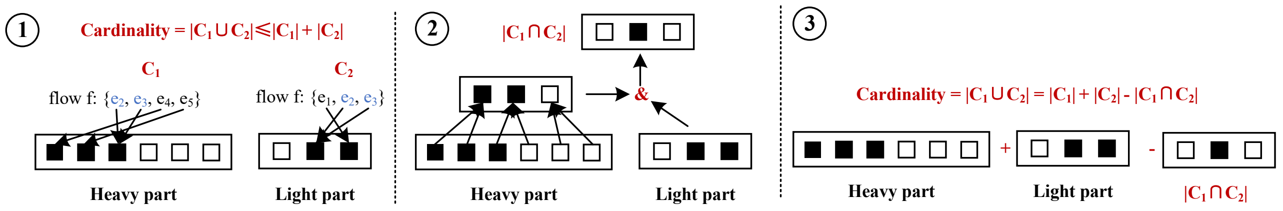


Fig. 8: Reducing overestimation errors. ① A naive summation would yield $|C_1|+|C_2|$, whereas the correct estimate is $|C_1 \cup C_2|$. ② MEC-Sketch maps the heavy part's cardinality estimator to a temporary estimator equivalent to those in the light part. It then applies a bitwise *AND* operation between this temporary estimator and the light part's estimator to approximate $|C_1 \cap C_2|$. ③ MEC-Sketch computes $|C_1 \cup C_2| = |C_1| + |C_2| - |C_1 \cap C_2|$ as the final estimate.

Sketch focuses on frequency-related tasks.

(2) Real-time Performance and Cardinality Mapping: Even if Elastic Sketch's counters are replaced with cardinality estimators, it still suffers from poor real-time performance and the inability to efficiently map large cardinality estimators in the heavy part to smaller ones in the light part—challenges that MEC-Sketch explicitly addresses. Even after replacing the counters in Elastic Sketch with cardinality estimators and adapting our mapping strategy, experimental results show that, in super-spreader detection, MEC-Sketch achieves a throughput improvement of 26.5 times. In cardinality estimation, MEC-Sketch reduces the error by 2-3 times and improves throughput by 17.5 times compared to Elastic-Sketch.

(3) Insertion and Query Mechanisms: MEC-Sketch employs distinct insertion and query algorithms. The distinction in query algorithms is clearly observable. The key distinction in insertion lies in MEC-Sketch's avoidance of Elastic Sketch's replacement insertion operation, which would otherwise lead to significant cardinality overestimation for mouse flows.

(4) Parallelism: Elastic Sketch lacks the parallelized implementation proposed in Section III-B of this paper.

IV. Experimental Results

A. Experimental Setup

Dataset: We use two real-world network traces and conduct experiments based on source-destination IP pairs. These two network traces have been widely used in previous studies [41]–[43]. Specifically, we treat the source IP as the flow key and the destination IP as the element. Thus, the cardinality is defined as the number of unique destination IPs associated with each source IP. It is worth noting that, in addition to defining flow keys and elements as source-destination IP pairs, MEC-Sketch is also applicable to other use cases with different definitions, as the underlying principles remain similar.

(1) MAWI [39]: This dataset, maintained by the MAWI Working Group of the WIDE Project, contains traffic traces. We extract 10 million consecutive packets, yielding approximately 56,000 flows when aggregated by source IP and about 898,000 flows when aggregated by source-destination IP pairs.

(2) CAIDA [40]: This dataset comprises anonymized IP traces collected by CAIDA in 2018. We extract 10 million consecutive packets, resulting in approximately 150,000 flows when aggregated by source IP and around 458,000 flows when aggregated by source-destination IP pairs.

TABLE I: Number of true super-spreaders in the CAIDA and MAWI datasets under different super-spreader threshold.

Threshold	100	200	300	400	500	600	700	800
MAWI	797	530	424	347	291	210	142	119
CAIDA	325	140	73	53	43	40	37	32

Experimental Settings: For elephant flows, HLL and MRB provide more accurate estimates than LC, while for mouse flows, LC is simple yet sufficiently accurate. Therefore, MEC-Sketch employs HLL or MRB as its cardinality estimator in the heavy part and LC as the cardinality estimator in the light part. Each HLL in MEC-Sketch consists of 128 registers, which is a classical setting. Each MRB comprises 8 levels, with each level containing an LC of 64 bits. The eviction threshold is set to $\lambda = 8$. For super-spreader detection, MEC-Sketch excludes the light part. In the parallel version of MEC-Sketch, the number of arrays is set to $d = 3$. In the minimal version, the number of slots per bucket is $w = 8$. For cardinality estimation, 30% of memory is allocated to the heavy part, while the light part contains $k = 3$ LC arrays, each 16 bits in size. We conduct experiments on parameter settings, which are omitted here due to space constraints. All experiments are conducted on a machine with an Intel Core i9-13900H processor (2.6 GHz, 14 cores, 20 threads) and 64GB of DDR4 memory. MEC-Sketch can be easily accommodated on any existing machine, as its size is less than 500KB. The code is implemented in C++, and is available at Github [44].

Abbreviations: The following abbreviations are used:

- MEC-P: The parallel version of MEC-Sketch for super-spreader detection, as described in Section III-B.
- MEC-M: The minimal version of MEC-Sketch for super-spreader detection, as described in Section III-C.
- MEC-C: The version of MEC-Sketch for cardinality estimation, as described in Section III-D.

B. Experiments on Super-Spreader Detection

We compare MEC-Sketch with the state-of-the-art super-spreader detection sketches: SpreadSketch [38] and NDS [14].

(1) Accuracy under the MAWI dataset: The first set of experiments evaluates performance by varying the super-spreader threshold, with each sketch allocated 60 KB of

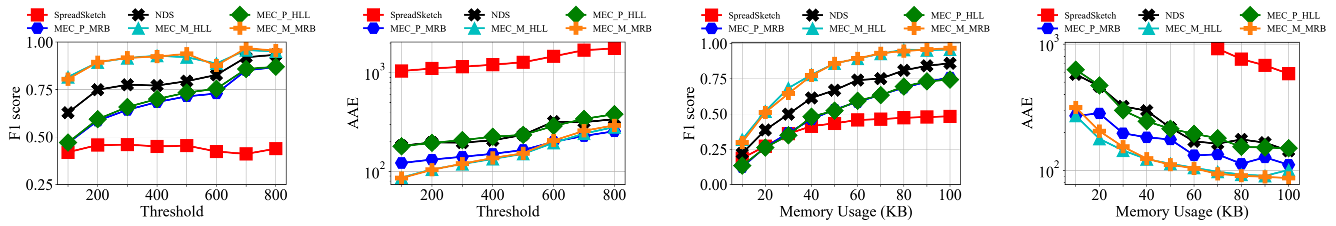


Fig. 9: Experimental Results on Super-Spreader Detection using the MAWI Dataset.

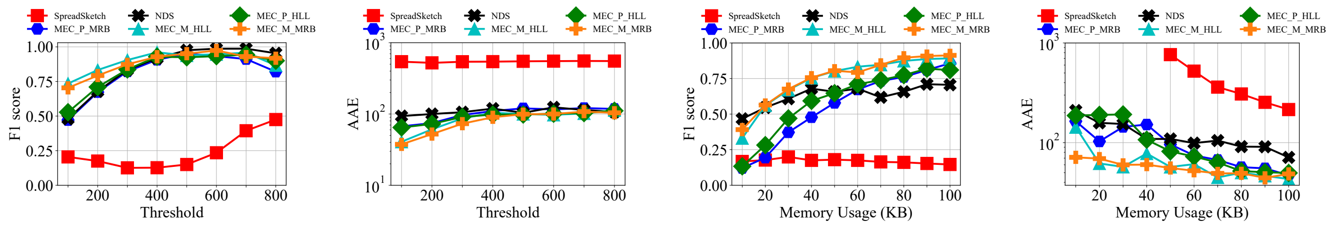
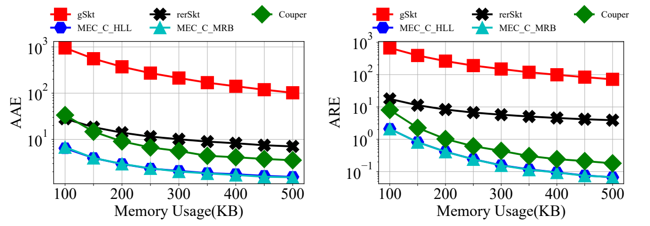
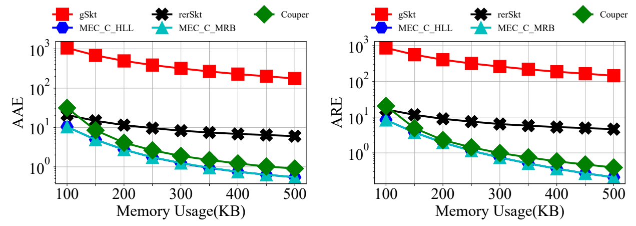


Fig. 10: Experimental Results on Super-Spreader Detection using the CAIDA Dataset.



(a) Experimental Results using the MAWI Dataset.



(b) Experimental Results using the CAIDA Dataset.

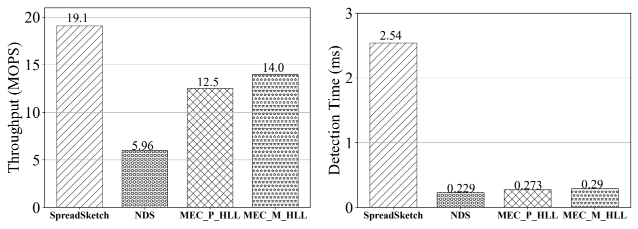
Fig. 11: Experimental Results on Cardinality Estimation.

memory. As shown in Table I, the threshold ranges from 100 to 800, resulting in the number of true super-spreaders varying from 797 to 119. As shown in Fig. 9, compared to NDS, MEC_M_HLL and MEC_M_MRB achieve maximum F1 score improvements of 18.8% and 17.7%, with average improvements of 10.9% and 11%, respectively. The F1 scores of MEC_P_HLL and MEC_P_MRB are slightly lower than that of NDS. Compared to SpreadSketch, MEC_M_HLL, MEC_M_MRB, MEC_P_HLL, and MEC_P_MRB achieve maximum F1 score improvements of 54.7%, 55.7%, 44.7%, and 44%, with average improvements of 46.9%, 47%, 26.5%, and 25.3%, respectively. In terms of AAE, compared to NDS, MEC_M_HLL, MEC_M_MRB, and MEC_P_MRB achieve maximum reductions of 2.09, 2.12, and 1.6 times, with average reductions of 1.61, 1.58, and 1.43 times, respectively. Compared to SpreadSketch, MEC_M_HLL, MEC_M_MRB, MEC_P_HLL, and MEC_P_MRB achieve maximum AAE reductions of 12, 12.2, 5.79, and 8.55 times, with average reductions of 8.81, 8.66, 5.31, and 7.81 times, respectively.

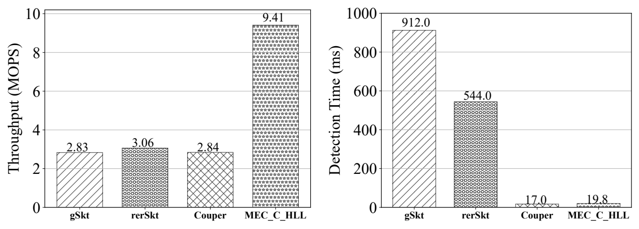
The second set of experiments evaluates performance by varying the memory allocation for each sketch, with the super-spreader threshold set to 200. As shown in Fig. 9, compared to NDS, MEC_M_HLL and MEC_M_MRB achieve maximum F1 score improvements of 19% and 19%, with average improvements of 14.6% and 13.9%, respectively. The F1 scores of MEC_P_HLL and MEC_P_MRB are slightly lower than those of NDS. Compared to SpreadSketch, MEC_M_HLL, MEC_M_MRB, MEC_P_HLL, and MEC_P_MRB achieve maximum F1 score improvements of 48.3%, 48.3%, 26%, and 27.5%, with average improvements of 38.3%, 37.6%, 11.2%, and 11.2%, respectively. In terms of AAE, compared to NDS, MEC_M_HLL, MEC_M_MRB, MEC_P_HLL, and MEC_P_MRB achieve maximum reductions of 2.61, 2.4, 1.21, and 2.13 times, with average reductions of 1.98, 1.94, 1.02, and 1.49 times, respectively. Compared to SpreadSketch, MEC_M_HLL, MEC_M_MRB, MEC_P_HLL, and MEC_P_MRB achieve maximum AAE reductions of 24.2, 21, 10.3, and 18.2 times, with average reductions of 13.5, 13, 6.66, and 10.1 times, respectively.

(2) Accuracy under the CAIDA dataset: As shown in Fig. 10, the results under the CAIDA dataset are consistent with those under the MAWI dataset. Due to space limitations, detailed discussion is omitted.

Analysis: In summary, the minimal version outperforms previous state-of-the-art super-spreader detection sketches, achieving the highest F1 score and the lowest error. The parallel version yields F1 scores comparable to or slightly lower than existing methods, but with reduced error. This is because NDS is better suited for datasets with higher skewness. Furthermore, we conduct experiments on datasets with



(a) Experimental Results on Super-Spreader Detection.



(b) Experimental Results on Cardinality Estimation.

Fig. 12: Experiments on Throughput and Detection Time using the MAWI Dataset.

lower skewness, where the parallel version outperforms NDS, although these results are omitted due to space constraints. Additionally, the non-monotonic behavior of SpreadSketch in the figure is due to its low precision when memory is limited, resulting from overestimation errors.

C. Experiments on Cardinality Estimation

We compare MEC-Sketch with state-of-the-art cardinality estimation sketches: gSkt [27], rerSkt [29], and Couper [13].

(1) Accuracy under the MAWI dataset: We evaluate the error under different memory allocations. As shown in Fig. 11, compared to Couper, rerSkt, and gSkt, MEC_C_HLL reduces AAE by up to 5.21, 4.92, and 148 times, with average reductions of 2.98, 4.68, and 105 times, respectively. MEC_C_MRB reduces AAE by up to 4.94, 5, and 140 times, with average reductions of 2.99, 4.72, and 105 times, respectively. Compared to Couper, rerSkt, and gSkt, MEC_C_HLL reduces ARE by up to 3.82, 58.1, and 1093 times, with average reductions of 2.78, 34.5, and 817 times, respectively. MEC_C_MRB reduces ARE by up to 3.82, 57.5, and 1115 times, with average reductions of 2.8, 34.8, and 824 times, respectively.

(2) Accuracy under the CAIDA dataset: As shown in Fig. 11, the results under the CAIDA dataset are consistent with those from the MAWI dataset. Due to space constraints, detailed discussion is omitted.

Analysis: In summary, compared to existing state-of-the-art cardinality estimation sketches, MEC-Sketch achieves the lowest error. This is primarily due to its consideration of network traffic skew characteristics, which are addressed through a dual-component design to achieve maximum memory efficiency. Furthermore, compared to the state-of-the-art solution, Couper, MEC-Sketch's approach of prioritizing the identification of elephant flows while evicting mouse flows to the second layer is more efficient. We also conduct experiments on datasets with lower skewness and even fully uniform distributions, where MEC-Sketch still outperforms Couper, though these results are omitted due to space constraints.

D. Experiments on Throughput and Detection Time

We evaluate the throughput and detection time on the MAWI dataset. To maximize the font size on the horizontal axis, we present only the results of MEC-Sketch integrated with HLL, as the results with MRB integration are similar.

Throughput and detection time under the MAWI dataset: In super-spreader detection, as shown in Fig. 12, with a super-spreader threshold set to 200 and 60KB of memory allocated to each sketch, MEC-Sketch achieves an update throughput approximately twice that of NDS. Although its update throughput is lower than that of SpreadSketch, MEC-Sketch demonstrates higher accuracy. Both MEC-Sketch and NDS detect all super-spreaders within 1ms in most cases, while SpreadSketch requires approximately 2.5ms. In cardinality estimation, as shown in Fig. 12, the update throughput of MEC-Sketch is 2 to 3 times that of Couper, rerSkt, and gSkt. MEC-Sketch and Couper exhibit the lowest query times.

Analysis: In terms of software platforms, MEC-Sketch's throughput surpasses almost all existing solutions. We will explore its extension to hardware platforms in future work.

V. Conclusion

In this paper, we propose MEC-Sketch, a memory-efficient cardinality estimation sketch. MEC-Sketch consists of two components: a heavy part and a light part. The heavy part employs the majority vote algorithm to enable efficient super-spreader detection, while the light part utilizes compact cardinality estimators to achieve memory-efficient cardinality estimation. Furthermore, we address key challenges in transitioning from counter-based to estimator-based designs. For instance, MEC-Sketch represents its cardinality by dynamically accumulating register values, thereby ensuring real-time performance of the majority vote algorithm. Additionally, MEC-Sketch effectively maps the large cardinality estimators in the heavy part to smaller ones in the light part by using a shared set of hash functions and configuring the array length of each estimator in the heavy part as an integer multiple of those in the light part. Extensive experiments demonstrate that MEC-Sketch outperforms previous state-of-the-art solutions for cardinality estimation and super-spreader detection in both accuracy and performance.

Acknowledgments

We would like to thank our shepherd and the anonymous reviewers for their thoughtful feedback. This work is supported by the National Natural Science Foundation of China under Grant Nos. U22B2005, 62432003 and 62032013; the Liaoning Revitalization Talents Program under Grant No. XLYC2403086; and the financial support of Lingnan University (LU) under Grant No. DB23A9.

References

[1] M. Roesch *et al.*, "Snort: Lightweight intrusion detection for networks." in *Lisa*, vol. 99, no. 1, 1999, pp. 229–238.

[2] F. D. Plonka, "A network traffic flow reporting and visualization tool usenix association," in *Proceedings of the 14th Systems Administration Conference (LISA*, 2000.

[3] A. Akella, A. Bharambe, M. Reiter, and S. Seshan, "Detecting ddos attacks on isp networks," in *Proceedings of the Workshop on Management and Processing of Data Streams*, 2003, pp. 1–2.

[4] J. Mirkovic and P. Reiher, "A taxonomy of ddos attack and ddos defense mechanisms," *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 2, pp. 39–53, 2004.

[5] S. K. Fayaz, Y. Tobioka, V. Sekar, and M. Bailey, "Bohatei: Flexible and elastic ddos defense," in *24th USENIX security symposium (USENIX Security 15)*, 2015, pp. 817–832.

[6] S. Sen and J. Wang, "Analyzing peer-to-peer traffic across large networks," in *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, 2002, pp. 137–150.

[7] Z. Durumeric, M. Bailey, and J. A. Halderman, "An internet-wide view of internet-wide scanning," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 65–78.

[8] S. Chen and Y. Tang, "Slowing down internet worms," in *24th International Conference on Distributed Computing Systems, 2004. Proceedings.* IEEE, 2004, pp. 312–319.

[9] S. Singh, C. Estan, G. Varghese, and S. Savage, "Automated worm fingerprinting." in *OSDI*, vol. 4, 2004, pp. 4–4.

[10] K.-Y. Whang, B. T. Vander-Zanden, and H. M. Taylor, "A linear-time probabilistic counting algorithm for database applications," *ACM Transactions on Database Systems (TODS)*, vol. 15, no. 2, pp. 208–229, 1990.

[11] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier, "Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm," *Discrete mathematics & theoretical computer science*, no. Proceedings, 2007.

[12] C. Estan, G. Varghese, and M. Fisk, "Bitmap algorithms for counting active flows on high speed links," in *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, 2003, pp. 153–166.

[13] X. Song, J. Zheng, H. Qian, S. Zhao, H. Zhang, X. Pan, and G. Chen, "In search of a memory-efficient framework for online cardinality estimation," *IEEE Transactions on Knowledge and Data Engineering*, 2024.

[14] H. Wang, "Enhancing accuracy for super spreader identification in high-speed data streams," *Proceedings of the VLDB Endowment*, vol. 17, no. 11, pp. 3124–3137, 2024.

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

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

[17] Y. Zhou, T. Yang, J. Jiang, B. Cui, M. Yu, X. Li, and S. Uhlig, "Cold filter: A meta-framework for faster and more accurate stream processing," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 741–756.

[18] 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, pp. 561–575.

[19] 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, pp. 220–238.

[20] Y. Li, F. Wang, X. Yu, Y. Yang, K. Yang, T. Yang, Z. Ma, B. Cui, and S. Uhlig, "Ladderfilter: Filtering infrequent items with small memory and time overhead," *Proceedings of the ACM on Management of Data*, vol. 1, no. 1, pp. 1–21, 2023.

[21] L. Tang, Q. Huang, and P. P. Lee, "Mv-sketch: A fast and compact invertible sketch for heavy flow detection in network data streams," in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 2019, pp. 2026–2034.

[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*, vol. 27, no. 5, pp. 1845–1858, 2019.

[23] 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, pp. 2584–2593.

[24] Y. Zhao, W. Zhou, W. Han, Z. Zhong, Y. Zhang, X. Zheng, T. Yang, and B. Cui, "Achieving top-k-fairness for finding global top-k frequent items," *IEEE Transactions on Knowledge and Data Engineering*, 2024.

[25] Q. Xiao, S. Chen, M. Chen, and Y. Ling, "Hyper-compact virtual estimators for big network data based on register sharing," in *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 2015, pp. 417–428.

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

[27] Y. Zhou, Y. Zhang, C. Ma, S. Chen, and O. O. Odegbile, "Generalized sketch families for network traffic measurement," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 3, no. 3, pp. 1–34, 2019.

[28] Y.-E. Sun, H. Huang, C. Ma, S. Chen, Y. Du, and Q. Xiao, "Online spread estimation with non-duplicate sampling," in *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE, 2020, pp. 2440–2448.

[29] H. Wang, C. Ma, O. O. Odegbile, S. Chen, and J.-K. Peir, "Randomized error removal for online spread estimation in data streaming," *Proceedings of the VLDB Endowment*, vol. 14, no. 6, 2021.

[30] R. B. Basat, X. Chen, G. Einziger, S. L. Feibish, D. Raz, and M. Yu, "Routing oblivious measurement analytics," in *2020 IFIP Networking Conference (Networking)*. IEEE, 2020, pp. 449–457.

[31] C. Ma, S. Chen, Y. Zhang, Q. Xiao, and O. O. Odegbile, "Super spreader identification using geometric-min filter," *IEEE/ACM Transactions on Networking*, vol. 30, no. 1, pp. 299–312, 2021.

[32] Y. Liu, W. Chen, and Y. Guan, "Identifying high-cardinality hosts from network-wide traffic measurements," *IEEE Transactions on Dependable and Secure Computing*, vol. 13, no. 5, pp. 547–558, 2015.

[33] S. Ganguly, M. Garofalakis, R. Rastogi, and K. Sabnani, "Streaming algorithms for robust, real-time detection of ddos attacks," in *27th International Conference on Distributed Computing Systems (ICDCS'07)*. IEEE, 2007, pp. 4–4.

[34] P. Wang, X. Guan, T. Qin, and Q. Huang, "A data streaming method for monitoring host connection degrees of high-speed links," *IEEE Transactions on Information Forensics and Security*, vol. 6, no. 3, pp. 1086–1098, 2011.

[35] M. Yu, L. Jose, and R. Miao, "Software defined traffic measurement with opensketch," in *10th USENIX symposium on networked systems design and implementation (NSDI 13)*, 2013, pp. 29–42.

[36] W. Liu, W. Qu, J. Gong, and K. Li, "Detection of superpoints using a vector bloom filter," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 3, pp. 514–527, 2015.

[37] X. Jing, Z. Yan, H. Han, and W. Pedrycz, "Extendedsketch: Fusing network traffic for super host identification with a memory efficient sketch," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 6, pp. 3913–3924, 2021.

[38] L. Tang, Q. Huang, and P. P. Lee, "Spreadsketch: Toward invertible and network-wide detection of superspreaders," in *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE, 2020, pp. 1608–1617.

[39] https://mawi.wide.ad.jp/mawi/samplepoint-F/2021/202109011400.html.

[40] https://catalog.caida.org/dataset/passive_2018_pcap.

[41] K. Guo, F. Li, J. Shen, X. Wang, and J. Cao, "Distributed sketch deployment for software switches," *IEEE Transactions on Computers*, 2024.

[42] K. Guo, F. Li, J. Shen, and X. Wang, "Advancing sketch-based network measurement: A general, fine-grained, bit-adaptive sliding window framework," in *2024 IEEE/ACM 32nd International Symposium on Quality of Service (IWQoS)*. IEEE, 2024, pp. 1–10.

[43] Y. Liu, K. Guo, F. Li, J. Shen, and X. Wang, "La-sketch: An adaptive level-aware sketch for efficient network traffic measurement," in *2025 IEEE/ACM 33nd International Symposium on Quality of Service (IWQoS)*. IEEE, 2025, pp. 1–10.

[44] https://github.com/QingYeyyds/MEC-Sketch.