A Unified Framework for High-Accuracy and Memory-Efficient Per-Flow Cardinality Measurement

Kejun Guo, Fuliang Li, *Member, IEEE*, Yunjie Zhang, Haorui Wan, Jiaxing Shen, *Member, IEEE* Xingwei Wang, *Member, IEEE* and Jiannong Cao, *Fellow, IEEE*

***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 high-accuracy and memory-efficient cardinality measurement 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 high-accuracy super-spreader detection, and a light part implementing compact cardinality estimators for memory-efficient measurement of mouse flows. We address three fundamental technical challenges: (1) adapting the majority vote algorithm to operate with cardinality estimators that lack native support for real-time queries, (2) designing an effective mapping strategy between large and small estimators, and (3) eliminating noise introduced by hash collisions. 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 [2], [3], P2P hot-spot localization [4], port scanning measurement [5], and worm propagation detection [6]. 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 [7], HyperLogLog [8], or Multiresolution Bitmap [9]). 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 [10] 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. Moreover, super-spreaders also refer to flows with high cardinality. From this perspective, elephant flows and super-spreaders can be considered equivalent in cardinality measurement. Therefore, we use these two terms interchangeably in the remainder of this paper. 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 pass through the first layer, hash collisions

A preliminary version of this work was presented at the IEEE International Conference on Network Protocols (ICNP'25) in September 2025 [1].

Kejun Guo, Fuliang Li, Yunjie Zhang, Haorui Wan and Xingwei Wang are with the School of Computer Science and Engineering, Northeastern University, Shenyang 110819, China. E-mail: kejunguo@163.com, lifuliang@cse.neu.edu.cn, 2472095@stu.neu.edu.cn, 2301947@stu.neu.edu.cn, wangxw@mail.neu.edu.cn.

Jiaxing Shen is with the Division of Artificial Intelligence, Lingnan University, Hong Kong. E-mail: jiaxingshen@LN.edu.hk.

Jiannong Cao is with the School of Department of Computing, The Hong Kong Polytechnic University, Hong Kong. E-mail: csjcao@comp.polyu.edu.hk.

(Corresponding author: Fuliang Li.)

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 [11].

NDS [11] 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 high-accuracy super-spreader detection, and a light part implementing compact cardinality estimators for memory-efficient measurement of mouse flows. In contrast to Couper [10], 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 [11]: (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 high-accuracy and 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 high-accuracy super-spreader detection, while the light part utilizes compact cardinality estimators (e.g., Linear Counting [7] 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, we effectively map the large cardinality estimators to smaller ones by using a shared set of hash functions and configuring the array length of each estimator as an integer multiple of those in the other bucket arrays. 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. 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 [12], Count Sketch [13], and so on. To enhance memory efficiency and accuracy, some frequency estimation sketches separate elephant and mouse flows, such as Cold Filter [14], Elastic Sketch [15], and so on [16]. Heavy-hitter detection identifies flows exceeding a predefined threshold. Notable sketches for this task include MV Sketch [17], Elastic Sketch, HeavyKeeper [18] and so on [19]. MV Sketch and Elastic Sketch employ the majority vote algorithm to improve detection accuracy. Cardinality-related sketches, which focus on estimating the number of distinct elements in a flow, are discussed in Section II-B.

B. Per-Flow Cardinality Measurement

Currently, there are many solutions for per-flow cardinality measurement, which can be roughly classified into three categories: per-source tracking, sampling-based solutions, and sketch-based solutions.

Per-source tracking. Traditional intrusion detection systems (e.g., Snort [2] and FlowScan [3]) maintain all active connections for each source to detect port scans or DDoS attacks. However, per-source tracking incurs significant resource overhead.

Sampling-based solutions. While sampling-based solutions [20]–[25] have lower overhead, they suffer from reduced accuracy, particularly for mouse flows. Even for super-spreader detection, as noted in the NDS [11], sampling-based solutions exhibit limited accuracy because they require a substantial volume of processed elements to achieve convergence.

Sketch-based solutions. We separately introduce sketch-based cardinality estimation and sketch-based super-spreader detection.

1) Sketch-based cardinality estimation solutions [10], [26]–[29] are mostly variants of frequency-related sketch. gSkt [28] is a cardinality estimation variant of CM Sketch [12], performing k independent estimations and returning the minimum value. rerSkt [29] follows the strategy similar to Count Sketch [13], dividing background traffic into primary and secondary estimators, with the secondary estimator used to subtract errors from the primary one. Couper [10] is the state-of-the-art solution for cardinality estimation and can be viewed as a variant of Cold Filter [14]. 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.

2) Similarly, sketch-based super-spreader detection solutions [11], [30]–[37] are mostly variants of frequency-related sketch. SpreadSketch [37] adapts CM Sketch [12] and uses a probabilistic approach to record super-spreaders. Similar to the overestimation in CM Sketch, SpreadSketch also suffers from significant estimation errors. AROMA [31] employs a self-adaptive sampling strategy to record super-spreaders and identifies them by locating the flow keys that appear most frequently in the array. However, storing same flow keys multiple times in the array may incur substantial memory overhead. The state-of-the-art solution, NDS [11] 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 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, while NDS achieves high accuracy, its insertion complexity results in a throughput significantly lower than that of counterparts such as SpreadSketch. There are other super-spreader detection sketches [32]–[36], [38], but as described in the NDS, their performance is inferior to NDS, so they are not further discussed.

C. Single-Flow Cardinality Measurement

There are many existing solutions toward single-flow cardinality measurement. We briefly introduce three of the most representative single-flow cardinality estimators.

Linear Counting (LC) [7]: maintains a bit array B of size b and is associated with a hash function $h()$. When inserting an element e, LC uses $h(e)$ to map e to $B[h(e)]$ and sets it to one. During querying, LC counts the number of zero bits in B, denoted as z, and estimates the cardinality c using the maximum likelihood estimator: $\hat{c} = b \ln \frac{b}{z}$.

HyperLogLog (HLL) [8]: 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 lz. Subsequently, HLL updates $A[h_1(e)] = \max(A[h_1(e)], lz)$. 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.

Multi-Resolution-Bitmap (MRB) [9]: consists of R bitmaps, each containing b bits, arranged hierarchically. When inserting an element e, MRB uses $h_1(e)$ to generate a bit string and calculates the number of leading zeros, denoted as r, to assign e to layer r. This ensures that the sampling probability for layer i is $\frac{1}{2^{i+1}}$ for $0 \leq i < R$. During querying, MRB applies LC independently to each layer. Since LC's accuracy decreases as the load factor (the fraction of bits set to "1") increases, MRB only utilizes bitmaps with a load factor below a predefined threshold θ ($\theta = 0.93$).

III. MEC-SKETCH

In this section, we first provide an overview of MEC-Sketch. Then, we separately explain in detail how MEC-Sketch achieves super-spreader detection and cardinality estimation.

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 $True$,

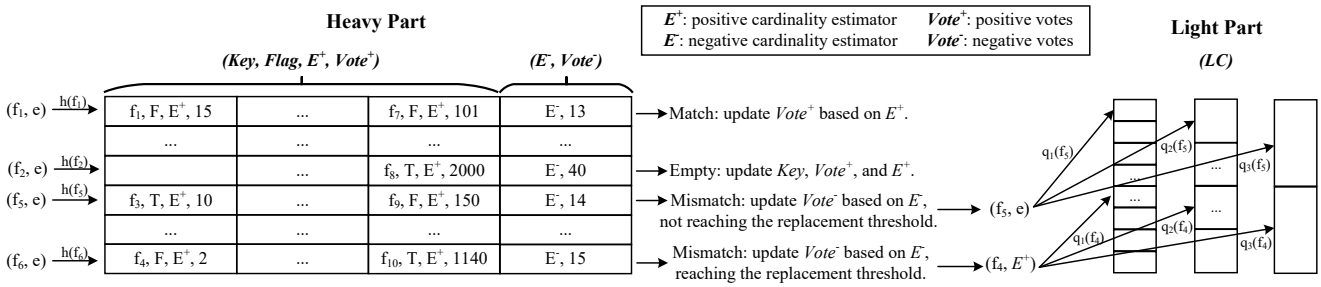


Fig. 1: Overview of MEC-Sketch. The heavy part employs the majority vote algorithm to separate elephant flows from mouse flows, enabling high-accuracy super-spreader detection. The light part utilizes compact cardinality estimators (e.g., LC [7] with a few dozen bits) to achieve memory-efficient cardinality estimation.

MEC-Sketch performs queries in both the heavy and light parts and mitigates overestimation errors using the noise elimination strategy described in Section III-D. If no match is found in the heavy part, MEC-Sketch performs the query in the light part and applies the noise elimination strategy described in Section III-D to eliminate hash collision noise.

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.

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 [15], [17]. 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

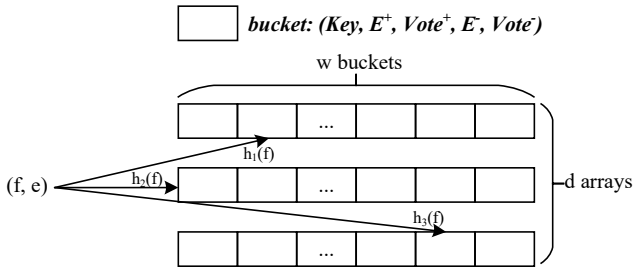


Fig. 2: The data structure of MEC-Sketch (Parallel Version).

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 $Vote = Vote + N$. It is worth noting that while adopting the sampling-based approach proposed in prior work [39] may offer certain advantages, we have implemented the aforementioned accumulation-based strategy to ensure broad compatibility across a wide range of single-flow cardinality estimators.

Data Structure: As shown in Fig. 2, MEC-Sketch comprises d arrays, each containing w buckets. Each bucket can be configured as a bit, a counter, or a key-value pair, depending on the specific functional requirements. In MEC-Sketch, 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

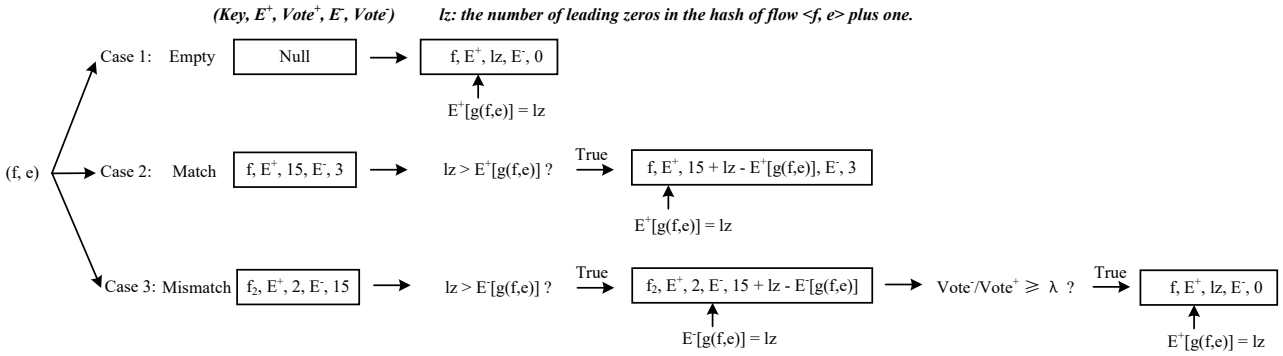


Fig. 3: The main insertion cases of MEC-Sketch (Parallel Version). MEC-Sketch employs the majority vote algorithm to enable efficient super-spreader detection, and approximates their cardinality by dynamically accumulating register values of the cardinality estimator, thereby ensuring the real-time performance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

in the i-th array as $A[i][j]$, with its respective fields represented as $A[i][j].Key, A[i][j].E^+, A[i][j].Vote^+, A[i][j].E^-$, $A[i][j].Vote^-$ $(1 \leq i \leq d, 0 \leq j \leq w-1)$.

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 $A[i][h_i(f)]$ $(1 \leq i \leq d)$. 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 $A[i][h_i(f)].Vote^-/A[i][h_i(f)].Vote^+ \geq \lambda$, 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 $A[i][h_i(f)]$ $(1 \leq i \leq d)$. Among these, it selects the buckets where $A[i][h_i(f)].Key = f$, assuming there are s such buckets $(s \leq d)$. Since these s buckets use the same hash function $g()$ for mapping each element e of flow f to the corresponding register or bit in their cardinality estimator $A[i][h_i(f)].E^+$ via $g(f,e)$, their results can be effectively combined to reduce errors caused by early eviction in the majority vote algorithm. Specifically, we denote the merged estimator as E^m. In HLL, each register in E^m is set to the maximum value among the corresponding registers of the s HLL estimators $A[i][h_i(f)].E^+$. In LC/MRB, each bit in E^m is computed as the bitwise OR of the corresponding bits in the s LC/MRB estimators $A[i][h_i(f)].E^+$. Finally, MEC-Sketch computes the final cardinality as $Est(E^m)$, 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. A mouse flow occupying a bucket is quickly replaced as other flows mapped to the same bucket increment the negative cardinality estimator, driving the ratio of negative to positive votes beyond the eviction threshold. Conversely, an elephant flow accumulates positive votes rapidly due to the presence of numerous distinct elements, significantly reducing the probability of replacement as its vote count increases.

Discussion: It is worth noting that this differs from replacing counters with cardinality estimators in MV-Sketch [17]. Lines 6-9 of Alg. 1 in MV-Sketch cannot be implemented with a cardinality estimator. Additionally, we do not emphasize the novelty of the majority vote algorithm. The key challenge lies in the fact that cardinality estimators do not support real-time updates like counters.

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 vote. 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,

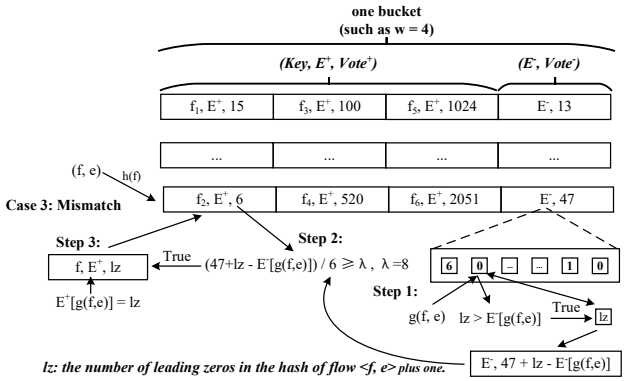
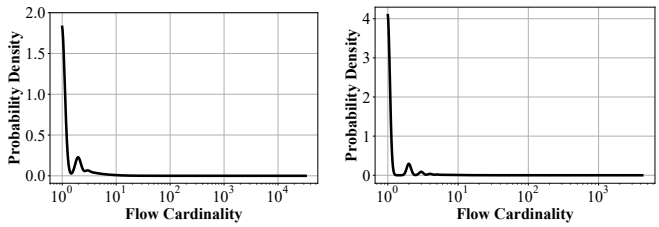


Fig. 4: The data structure and main insertion cases of MEC-Sketch (Minimal Version). MEC-Sketch enhances accuracy and memory efficiency by sacrificing parallelism, specifically by maintaining only a negative cardinality estimator.



(a) MAWI. (b) CAIDA.

Fig. 5: The probability density function of cardinality distribution in the MAWI [40] and CAIDA [41] 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.

positive cardinality estimators, and positive votes, denoted as $A[i][j].Key$, $A[i][j].E^+$, and $A[i][j].Vote^+$ ($1 \leq i \leq d, 0 \leq j < w-1$). 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 $A[h(f)][w-1].Vote^- += lz - A[h(f)][w-1].E^-[g(f,e)]$. 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 estimated based on the corresponding cardinality estimator.

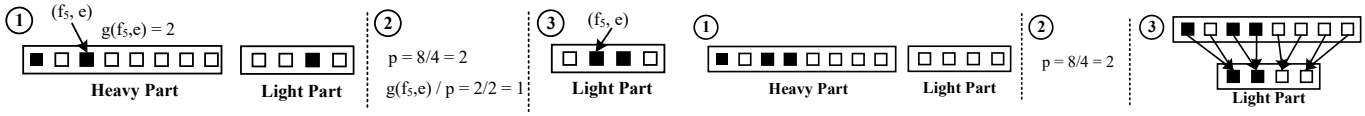
Analysis: By sharing a negative cardinality estimator and negative votes among multiple flows, MEC-Sketch significantly improves memory efficiency. Additionally, the presence of multiple slots per bucket enables effective identification of all super-spreaders, even when multiple super-spreaders are mapped 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 [40] and CAIDA [41] 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 [10], 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 Section III-C, we now focus on recording mouse flows. However, this introduces an additional challenge: evicting mouse flows to the second layer requires mapping a



(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 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.

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, as it can often be implemented using bit shifting, making it 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. Inspired by Tower Sketch [42], the light part consists of k arrays. The i-th array contains l_i $(1 \leq i \leq k)$ buckets, each implemented as a LC. Lower arrays contain more LC buckets with fewer bits, while upper arrays contain fewer buckets with more bits. Each array is associated with a hash function $q_i()$ $(1 \leq i \leq k)$. 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. The array length of each cardinality estimator in the heavy part is configured as p_i $(1 \leq i \leq k)$ times that of the corresponding light part.

Insertion: Initially, all fields are empty. For illustration, we assume that the heavy part employs an HLL with 128 registers. The light part consists of three LC arrays ($k = 3$), where each LC bucket occupies 4, 8, and 16 bits respectively from the lowest to the highest array. Moreover, each LC array is allocated an equal amount of memory, yielding a length ratio of 2:1 between adjacent arrays. 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 $B[i][q_i(f)]$ in the light part and sets the bit $B[i][q_i(f)].LC[g(f,e)/p_i]$ to 1. In practice, p_i is often configured as a power of two, enabling efficient implementation through bit shifting. The $g(f,e)$ is computed in the heavy part and does not need to be recalculated in the light part.

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 $A[h(f)][j]$ 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 $B[i][q_i(f)]$. Then, MEC-Sketch sets $B[i][q_i(f)].LC[b] = 1$ if at least one register among $A[h(f)][j].E^+[b \times p_i]$ to $A[h(f)][j].E^+[(b+1) \times p_i - 1]$ is nonzero, where $0 \leq b < \text{len}(B[i][q_i(f)].LC)$.

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 8 and a corresponding estimator in the light part with a bit array length of 4. 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_i 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

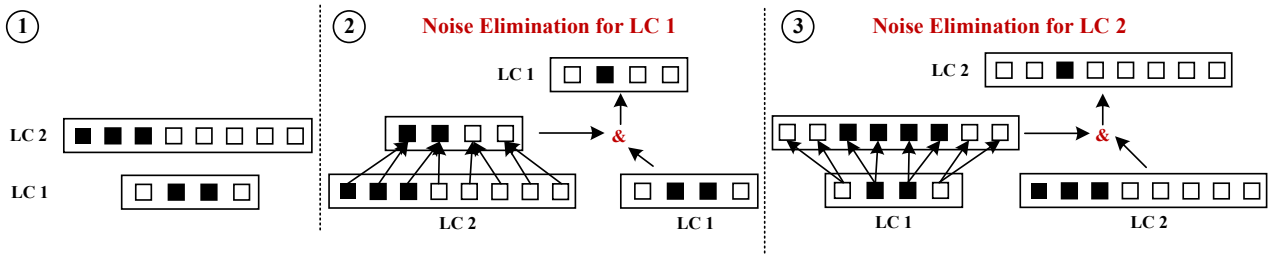


Fig. 7: AND-based noise elimination. ① In the light part, suppose flow f is mapped to two LC buckets in the two lowest LC arrays, as illustrated by LC 1 and LC 2, where LC 1 contains 4 bits and LC 2 contains 8 bits. ② Noise caused by hash collisions in LC 1 is eliminated. ③ Noise caused by hash collisions in LC 2 is eliminated.

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_i consecutive registers in the heavy part's cardinality estimator to a corresponding bit in the light part's estimator. If any of these p_i 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 estimated 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, followed by taking the minimum value among the k LC buckets. As shown in Fig. 7, MEC-Sketch applies an *AND* operation across the k LC buckets before calculating the cardinality. Since all LC buckets across the k arrays use the same hash function $g()$, the *AND* operation 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. It is worth noting that the light part first performs *AND*-based noise elimination on the k LC buckets mapped by flow f, as illustrated in Fig. 7. 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. We apply the above procedure to each of the k noise-eliminated LC buckets and select the minimum value.

Theoretical Bounds: Due to the utilization of small-scale cardinality estimators (specifically 4, 8, and 16-bit LCs), the error bound of the light part is minimized. The error bound for the heavy part is comparable to that of Elastic Sketch [15], with the distinction that flow cardinality is substituted for flow frequency in the theoretical derivation.

Analysis: MEC-Sketch enhances memory efficiency by separating elephant and mouse flows. Compared to the state-of-the-art solution, Couper [10], 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. It is worth noting that prior studies [37] typically perform bitwise-*AND* operations between homogeneous cardinality estimators of the same size. In contrast, due to our fine-grained hierarchical design, the bitwise-*AND* operations in MEC-Sketch are performed across heterogeneous cardinality estimators of varying sizes.

E. Comparison with Prior Approaches

1) Comparison with Elastic Sketch [15]: While some may perceive similarities between MEC-Sketch and Elastic Sketch, they differ fundamentally in the following aspects:

(1) Objective: MEC-Sketch is designed to address cardinality-related tasks, whereas Elastic Sketch focuses on frequency-related tasks.

(2) Data Structure: The light part of Elastic Sketch adopts a standard CM Sketch, whereas the light part of MEC-Sketch employs a more fine-grained hierarchical design, achieving higher memory efficiency and improved accuracy.

(3) Real-Time Performance: Even if Elastic Sketch's counters are replaced with cardinality estimators, it suffers from poor real-time performance. This is because Elastic Sketch relies on the majority vote algorithm, which requires traversing the memory of the cardinality estimators and computing the vote counts (i.e., cardinalities) for each insertion. However, unlike compact counters (e.g., 32-bit), cardinality estimators (e.g., an HLL with 128 registers of 5 bits each) typically occupy much larger memory. As a result, the traversal operation severely degrades throughput. In contrast, MEC-Sketch effectively addresses this challenge by leveraging the accumulative property of cardinality estimators. As demonstrated in the experimental results in Section IV-E, compared with

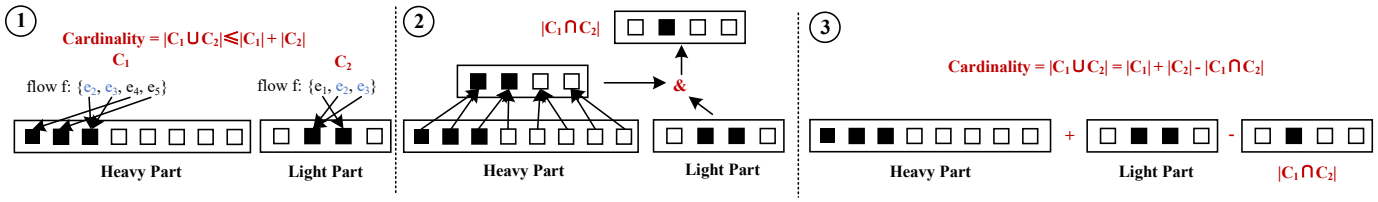


Fig. 8: Reducing overestimation errors. ① A naive summation would yield $|C_1|+|C_2|$, whereas the correct estimate is $|C_1 \cup C_2|$. ② MEC-Sketch maps the heavy part's cardinality estimator to a temporary estimator equivalent to those in the light part. It then applies a bitwise *AND* operation between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

Elastic Sketch, MEC-Sketch improves the throughput of super-spreader detection by $23.6\times$ and the throughput of cardinality estimation by $14.1\times$.

(4) Insertion and Query Mechanisms: MEC-Sketch employs distinct insertion and query algorithms. First, 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. Second, the key distinction in query lies in that Elastic Sketch simply sums the values from the heavy and light parts to obtain the final estimate, whereas MEC-Sketch further reduces estimation error by applying an *AND*-based noise elimination and subtracting the intersection between the heavy and light parts. As demonstrated in the experimental results in Section IV-E, for cardinality estimation, MEC-Sketch reduces the error by $6.94\times$ compared with Elastic Sketch.

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

2) Comparison with Couper [10]: Couper can be viewed as a variant of Cold Filter [14]. 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. However, this design faces two key limitations:

(1) Real-time Performance: Each insertion requires scanning dozens of bits in the first-layer estimator to determine whether the flow should be promoted, compromising real-time performance.

(2) Accuracy: Since elephant flows must pass through the first layer, hash collisions between elephant and mouse flows are highly likely, resulting in overestimation of mouse flows.

3) Comparison with Pyramid Sketch [43]: Pyramid Sketch is designed based on a progressive escalation mechanism, whereas MEC-Sketch is designed based on an eviction mechanism. Therefore, we consider MEC-Sketch to be architecturally distinct from the Pyramid Sketch.

IV. Experimental Results

In this section, we extensively evaluate the performance of MEC-Sketch in per-flow cardinality measurement.

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 [44]–[49]. 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 [40]: 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 [41]: 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.

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 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. From the lowest to the highest array, the LC bucket sizes are 4, 8, and 16 bits, respectively. Moreover, each LC array is allocated an equal amount of memory, yielding a length ratio of 2:1 between adjacent arrays. 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 [50].

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-

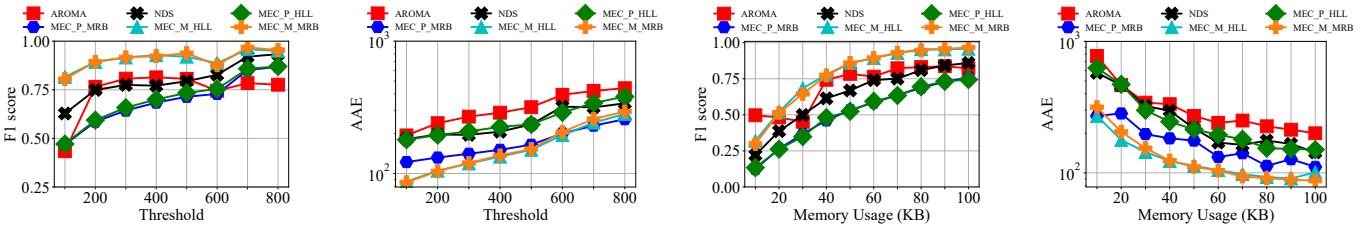


Fig. 9: Experimental Results on Super-Spreader Detection using the MAWI Dataset.

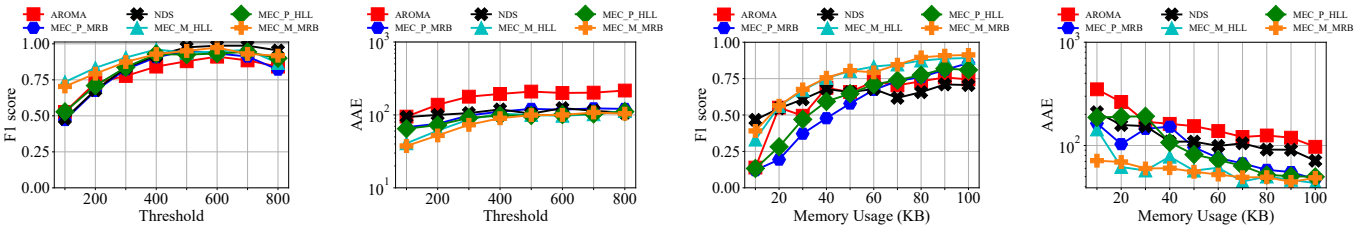
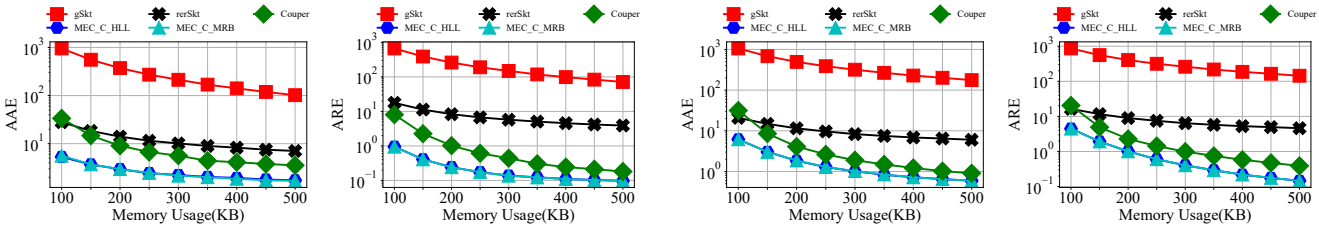


Fig. 10: Experimental Results on Super-Spreader Detection using the CAIDA Dataset.



(a) Experimental Results using the MAWI Dataset.

(b) Experimental Results using the CAIDA Dataset.

Fig. 11: Experimental Results on Cardinality Estimation.

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

spreader detection, as described in Section III-C.

- MEC-C: The version of MEC-Sketch for cardinality estimation, as described in Section III-D.

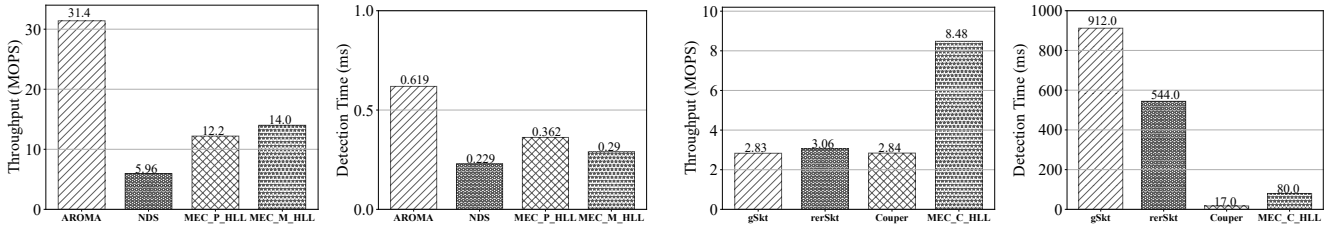
Metrics: We evaluate the following metrics.

- *Precision:* the ratio of true super-spreaders detected over all super-spreaders reported.
- *Recall:* the ratio of true super-spreaders detected over all true super-spreaders reported.
- *F1 Score:* the harmonic average of precision and recall.
- *Average Absolute Error (AAE):* $\frac{1}{m}\sum_{i=1}^{m}|n_i - \hat{n}_i|$, where m is the number of flows, n_i and $\hat{n}_i$ are the actual and estimated cardinality respectively.
- *Average Relative Error (ARE):* $\frac{1}{m}\sum_{i=1}^{m}\frac{|n_i-\hat{n}_i|}{n_i}$.
- *Throughput:* the number of packets processed per second.
- *Detection Time:* the time for querying all super-spreaders or querying the cardinality of all flows.

B. Experiments on Super-Spreader Detection

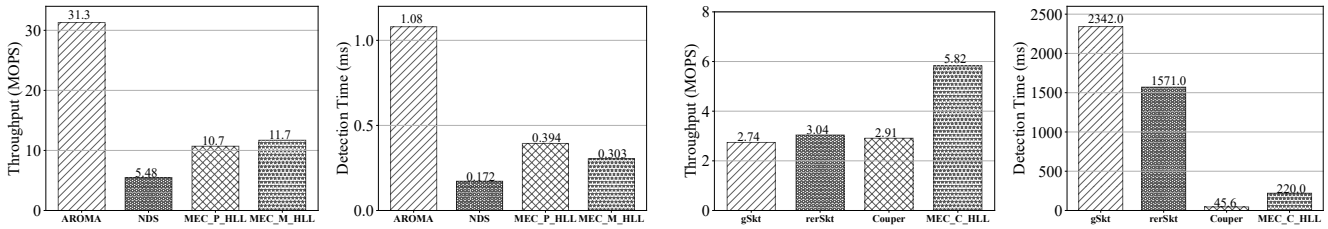
In this section, we compare MEC-Sketch with the state-of-the-art super-spreader detection sketches: AROMA [31] and NDS [11]. The parameter settings for AROMA and NDS follow those in their respective papers and open-source implementations. Comparisons with other methods [32]–[38] are omitted, as both AROMA and NDS have been shown to outperform all other existing solutions.

(1) Accuracy under the MAWI dataset: The first set of experiments evaluates performance by varying the super-spreader threshold, with each sketch allocated 60KB of 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 AROMA, MEC_M_HLL and MEC_M_MRB achieve maximum F1 score improvements of 38.2% and 37.1%, with average improvements of 16.7% and 16.8%, 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 AROMA, MEC_M_HLL, MEC_M_MRB, MEC_P_HLL, and MEC_P_MRB achieve maximum AAE reductions of 2.29, 2.31, 1.36, and 1.96 times, with average reductions of 2.04, 2, 1.25, and 1.83 times, respectively.



(a) Experimental Results on Super-Spreader Detection. (b) Experimental Results on Cardinality Estimation.

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



(a) Experimental Results on Super-Spreader Detection. (b) Experimental Results on Cardinality Estimation.

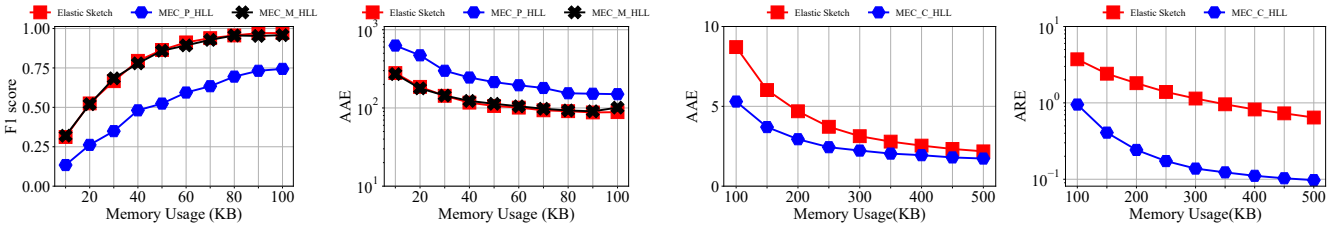
Fig. 13: Experimental Results on Throughput and Detection Time using the CAIDA Dataset.

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 AROMA, MEC_M_HLL and MEC_M_MRB achieve maximum F1 score improvements of 23.2% and 19.3%, with average improvements of 8.14% and 7.48%, 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 AROMA, MEC_M_HLL, MEC_M_MRB, MEC_P_HLL, and MEC_P_MRB achieve maximum AAE reductions of 2.86, 2.69, 1.47, and 2.86 times, with average reductions of 2.46, 2.42, 1.28, and 1.87 times, respectively.

(2) Accuracy under the CAIDA dataset: The first set of experiments evaluates performance by varying the super-spreader threshold, with each sketch allocated 60 KB of memory. As shown in Table I, the threshold ranges from 100 to 800, resulting in the number of true super-spreaders varying from 325 to 32. As shown in Fig. 10, compared to NDS, MEC_M_HLL and MEC_M_MRB achieve maximum F1 score improvements of 25.5% and 22.4%, with average improvements of 4.11% and 3.23%, respectively. The F1 scores of MEC_P_HLL and MEC_P_MRB are nearly equal to that of NDS. Compared to AROMA, MEC_M_HLL, MEC_M_MRB, MEC_P_HLL, and MEC_P_MRB achieve maximum F1 score improvements of 20.8%, 17.7%, 8.7%, and 7.2%, with average improvements of 9.39%, 8.5%, 3.91%, and 1.22%, 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 2.29, 2.49, 1.45, and 1.38 times, with average reductions of 1.34, 1.44, 1.2, and 1.07 times, respectively. Compared to AROMA, MEC_M_HLL, MEC_M_MRB, MEC_P_HLL, and MEC_P_MRB achieve maximum AAE reductions of 2.32, 2.64, 2.13, and 1.83 times, with average reductions of 2.07, 2.23, 1.91, and 1.7 times, respectively.

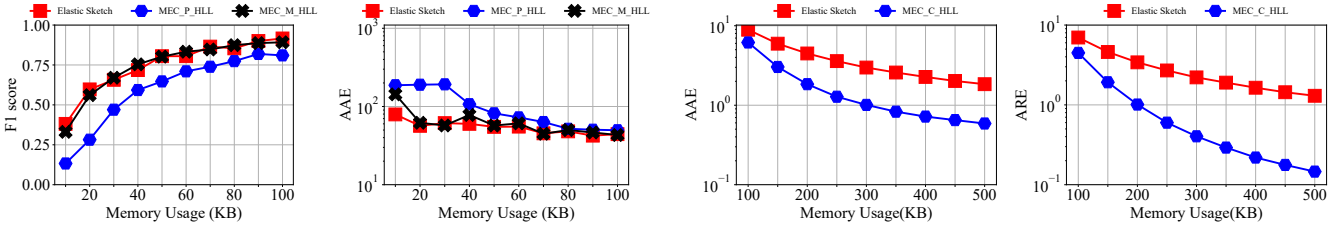
The second set of experiments evaluates performance by varying the memory allocation for each sketch, with the super-spreader threshold set to 200. As shown in Fig. 10, compared to NDS, MEC_M_HLL and MEC_M_MRB achieve maximum F1 score improvements of 22.9% and 23.9%, with average improvements of 11.2% and 12.2%, respectively. As the memory increases, the F1 scores of MEC_M_HLL and MEC_M_MRB gradually surpass those of NDS. Compared to AROMA, MEC_M_HLL and MEC_M_MRB achieve maximum F1 score improvements of 19.2% and 25.2%, with average improvements of 12.5% and 13.5%, 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 2.69, 2.95, 1.8, and 1.64 times, with average reductions of 1.94, 2.1, 1.31, and 1.34 times, respectively. Compared to AROMA, MEC_M_HLL, MEC_M_MRB, MEC_P_HLL, and MEC_P_MRB achieve maximum AAE reductions of 4.27, 4.91, 2.41, and 2.56 times, with average reductions of 2.67, 2.93, 1.81, and 1.85 times, respectively.

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 lower skewness, where the parallel version outperforms NDS, although these results are omitted due to space constraints.



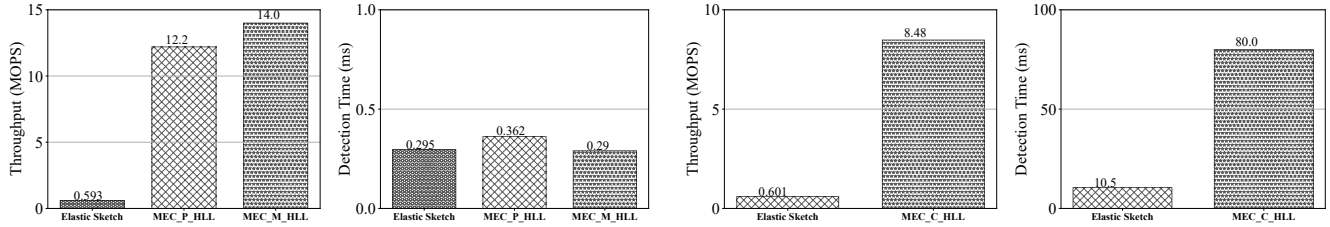
(a) Experimental Results on Super-Spreader Detection. (b) Experimental Results on Cardinality Estimation.

Fig. 14: Accuracy Comparison between Elastic Sketch and MEC-Sketch using the MAWI Dataset.



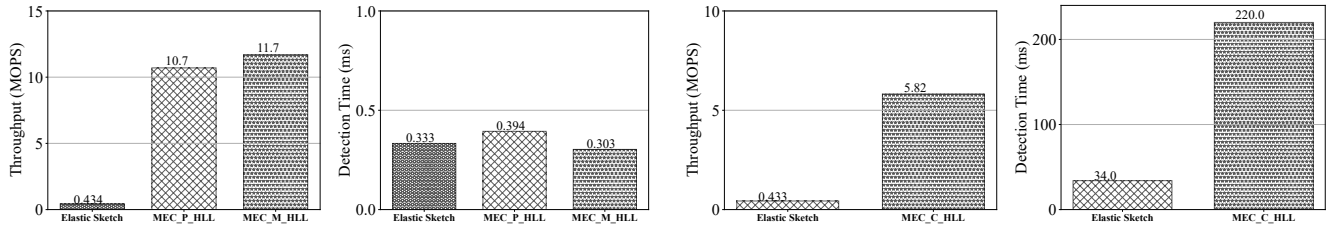
(a) Experimental Results on Super-Spreader Detection. (b) Experimental Results on Cardinality Estimation.

Fig. 15: Accuracy Comparison between Elastic Sketch and MEC-Sketch using the CAIDA Dataset.



(a) Experimental Results on Super-Spreader Detection. (b) Experimental Results on Cardinality Estimation.

Fig. 16: Throughput and Detection Time Comparison between Elastic Sketch and MEC-Sketch using the MAWI Dataset.



(a) Experimental Results on Super-Spreader Detection. (b) Experimental Results on Cardinality Estimation.

Fig. 17: Throughput and Detection Time Comparison between Elastic Sketch and MEC-Sketch using the CAIDA Dataset.

C. Experiments on Cardinality Estimation

In this section, we compare MEC-Sketch with state-of-the-art cardinality estimation sketches: gSkt [28], rerSkt [29], and Couper [10]. The implementation follows the code provided in [10], with parameter settings consistent with those in the respective papers.

(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 6.34, 5.28, and 180 times, with average reductions of 3.01, 4.57, and 105 times, respectively. MEC_C_MRB reduces AAE by up to 5.96, 4.96, and 169 times, with average reductions of 3, 4.6, and 105 times, respectively. Compared to Couper, rerSkt, and gSkt, MEC_C_HLL reduces ARE by up to 8.48, 41.7, and 1098 times, with average reductions of 3.73, 36, and 923 times, respectively. MEC_C_MRB reduces ARE by up to 8.48, 42, and 1098 times, with average reductions of 3.74, 36.2, and 927 times, respectively.

(2) Accuracy under the CAIDA 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 5.11, 10.2, and 320 times, with average reductions of 2.3, 7.66, and 281 times, respectively. MEC_C_MRB reduces AAE by up to 5.1, 10.2, and 318 times, with average

reductions of 2.29, 7.62, and 279 times, respectively. Compared to Couper, rerSkt, and gSkt, MEC_C_HLL reduces ARE by up to 4.58, 31.6, and 986 times, with average reductions of 2.76, 16.7, and 618 times, respectively. MEC_C_MRB reduces ARE by up to 4.58, 31.6, and 986 times, with average reductions of 2.76, 16.7, and 617 times, respectively.

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

In this section, we evaluate the throughput and detection time of MEC-Sketch, with results averaged over ten experiments. 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.

(1) 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 AROMA, MEC-Sketch demonstrates higher accuracy. Both MEC-Sketch and NDS detect all super-spreaders within 0.5ms in most cases, while AROMA requires approximately 0.619ms. 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.

(2) Throughput and detection time under the CAIDA dataset: In super-spreader detection, as shown in Fig. 13, 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 AROMA, MEC-Sketch demonstrates higher accuracy. Both MEC-Sketch and NDS detect all super-spreaders within 0.5ms in most cases, while AROMA requires approximately 1.08ms. In cardinality estimation, as shown in Fig. 13, the update throughput of MEC-Sketch is 2 times that of Couper, rerSkt, and gSkt. MEC-Sketch and Couper exhibit the lowest query times.

Analysis: In terms of software platform performance, MEC-Sketch outperforms nearly all existing solutions in throughput, while maintaining a detection time close to the minimum.

E. Comparison with Elastic Sketch

In this section, we replace the counters in Elastic Sketch with cardinality estimators and apply our proposed mapping strategy between the heavy and light parts to enable a fair comparison with MEC-Sketch. Specifically, each counter in the heavy part is replaced by a HLL with 128 registers, while each counter in the light part is replaced by a 16-bit LC. In super-spreader detection, to ensure a fair comparison with the parallel and minimal versions of MEC-Sketch, we retain only the heavy part of Elastic Sketch. In cardinality estimation, we adopt the cardinality estimation version of MEC-Sketch and use the full Elastic Sketch including its light part.

(1) Accuracy: As shown in Fig. 14 and 15, in super-spreader detection, MEC_M_HLL achieves F1 scores and AAE comparable to Elastic Sketch, whereas MEC_P_HLL exhibits slightly lower performance. In cardinality estimation, compared with Elastic Sketch, MEC_C_HLL reduces AAE and ARE by average factors of 1.45 and 6.94, respectively, on the MAWI dataset, and by average factors of 2.68 and 5.37, respectively, on the CAIDA dataset.

(2) Throughput and detection time: As shown in Fig. 16 and 17, in super-spreader detection with 60KB of memory allocated to each sketch, the update throughput of MEC_P_HLL and MEC_M_HLL on the MAWI dataset is 20.6 and 23.6 times higher than that of Elastic Sketch, respectively; on the CAIDA dataset, the corresponding throughput improvements are 24.7 and 27 times. All three methods can detect all super-spreaders within 1ms. In cardinality estimation, the update throughput of MEC_C_HLL is 14.1 times higher than that of Elastic Sketch on the MAWI dataset and 13.4 times higher on the CAIDA dataset, while its detection time is slightly higher than that of Elastic Sketch.

Analysis: In summary, MEC-Sketch achieves accuracy comparable to Elastic Sketch in super-spreader detection, while providing lower estimation error in cardinality estimation. Moreover, MEC-Sketch attains throughput that is tens of times higher than that of Elastic Sketch. These improvements primarily arise from its approximate cardinality estimation technique and its *AND*-based noise elimination strategy.

V. Conclusion

In this paper, we propose MEC-Sketch, a high-accuracy and 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 high-accuracy 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 to smaller ones by using a shared set of hash functions and configuring the array length of each estimator as an integer multiple of those in the other bucket arrays. Moreover, during the query phase, MEC-Sketch enhances estimation accuracy through an *AND*-based noise elimination strategy. 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.

VI. Limitations and Future Work

Although MEC-Sketch has demonstrated strong performance on software platforms, its deployment and feasibility on hardware platforms remain unexplored. In future work, we intend to investigate its hardware implementation and assess its practical viability.

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. 62572105 and U22B2005, as well as the LiaoNing Revitalization Talents Program under Grant No. XLYC2403086.

References

[1] K. Guo, F. Li, Y. Zhang, H. Wan, J. Shen, and X. Wang, "Mec-sketch: Memory-efficient per-flow cardinality measurement in high-speed networks," in *2025 IEEE 33rd International Conference on Network Protocols (ICNP)*. IEEE, 2025, pp. 1–11.

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

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

[4] 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.

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, "Elastic sketch: Adaptive and fast network-wide measurements," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 2018, pp. 561–575.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] N. Kamiyama, T. Mori, and R. Kawahara, "Simple and adaptive identification of superspreaders by flow sampling," in *IEEE INFOCOM 2007-26th IEEE International Conference on Computer Communications*. IEEE, 2007, pp. 2481–2485.

[21] Y. Du, H. Huang, Y.-E. Sun, S. Chen, G. Gao, X. Wang, and S. Xu, "Short-term memory sampling for spread measurement in high-speed networks," in *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*. IEEE, 2022, pp. 470–479.

[22] Y. Du, H. Huang, Y.-E. Sun, S. Chen, and G. Gao, "Self-adaptive sampling for network traffic measurement," in *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*. IEEE, 2021, pp. 1–10.

[23] H. Huang, Y.-E. Sun, C. Ma, S. Chen, Y. Du, H. Wang, and Q. Xiao, "Spread estimation with non-duplicate sampling in high-speed networks," *IEEE/ACM Transactions on Networking*, vol. 29, no. 5, pp. 2073–2086, 2021.

[24] 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.

[25] H. Huang, Y.-E. Sun, C. Ma, S. Chen, Y. Zhou, W. Yang, S. Tang, H. Xu, and Y. Qiao, "An efficient k-persistent spread estimator for traffic measurement in high-speed networks," *IEEE/ACM Transactions on Networking*, vol. 28, no. 4, pp. 1463–1476, 2020.

[26] 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.

[27] 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.

[28] 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.

[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] K. Guo, F. Li, J. Shen, H. Wan, S. Chen, and M. Hou, "One-sketch: A unified framework for per-flow cardinality measurement with flexible bias control," in *IEEE INFOCOM 2026-IEEE Conference on Computer Communications*. IEEE, 2026.

[31] R. Ben-Basat, G. Einziger, S. L. Feibish, J. Moraney, B. Tayh, and D. Raz, "Routing-oblivious network-wide measurements," *IEEE/ACM Transactions on Networking*, vol. 29, no. 6, pp. 2386–2398, 2021.

[32] 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.

[33] 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.

[34] 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.

[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] 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.

[38] 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.
[39] D. Ting, "Streamed approximate counting of distinct elements: Beating optimal batch methods," in *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2014, pp. 442–451.
[40] "MAWI Dataset," https://mawi.wide.ad.jp/mawi/samplepoint-F/2021/202109011400.html, accessed: 2026-05-24.
[41] "Anonymized Internet Traces 2018," https://catalog.caida.org/dataset/passive_2018_pcap, accessed: 2026-05-24.
[42] K. Yang, S. Long, Q. Shi, Y. Li, Z. Liu, Y. Wu, T. Yang, and Z. Jia, "Sketchint: Empowering int with towersketch for per-flow per-switch measurement," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 11, pp. 2876–2894, 2023.
[43] Y. Li, X. Yu, Y. Yang, Y. Zhou, T. Yang, Z. Ma, and S. Chen, "Pyramid family: Generic frameworks for accurate and fast flow size measurement," *IEEE/ACM Transactions on Networking*, vol. 30, no. 2, pp. 586–600, 2021.
[44] F. Li, K. Guo, Y. Liu, J. Shen, X. Wang, and J. Cao, "A unified configuration framework for heterogeneous sketches," *IEEE Transactions on Networking*, 2026.
[45] K. Guo, F. Li, Y. Liu, J. Shen, and X. Wang, "Ra-sketch: A unified framework for rapid and accurate sketch configurations," in *2025 IEEE 33rd International Conference on Network Protocols (ICNP)*. IEEE, 2025, pp. 1–11.
[46] 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.
[47] K. Guo, F. Li, J. Shen, X. Wang, and J. Cao, "Distributed sketch deployment for software switches," *IEEE Transactions on Computers*, 2024.
[48] 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.
[49] F. Li, K. Guo, J. Shen, and X. Wang, "Effective network-wide traffic measurement: A lightweight distributed sketch deployment," in *IEEE INFOCOM 2024-IEEE Conference on Computer Communications*. IEEE, 2024, pp. 181–190.
[50] "Source Code," https://github.com/QingYeyyds/MEC-Sketch, accessed: 2026-05-24.

Kejun Guo received the B.Sc. degree in computer science from Northeastern University, China in 2023. He is currently pursuing the Ph.D. degree with the School of Computer Science and Engineering, Northeastern University, China. His research interests include data stream interests, network measurement, and network security. He is the recipient of the IEEE ICNP 2025 Best Paper Award and the IEEE/ACM IWQoS 2025 Best Student Paper Runner-up Award.

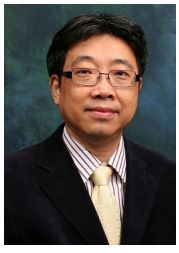
Fuliang Li (Member, IEEE) received the B.Sc. degree in computer science from Northeastern University, China, in 2009, and the Ph.D. degree in computer science from Tsinghua University, China, in 2015. He is currently a Professor with the School of Computer Science and Engineering, Northeastern University. He has published more than 50 journals/conference papers. His research interests include network management and measurement, cloud computing, and network security.

Yunjie Zhang received the B.Sc. degree in computer science from Shenyang University of Chemical Technology, China in 2024. He is currently pursuing the M.Sc. degree with the School of Computer Science and Engineering, Northeastern University, China. His research area is computer networking, with particular interests in network measurement.

Haorui Wan received the B.Sc. degree in computer science from Northeastern University, China in 2023. He is currently pursuing the Ph.D. degree with the School of Computer Science and Engineering, Northeastern University, China. His research interests include network system optimization.

Jiaxing Shen (Member, IEEE) is an Assistant Professor with the Division of Artificial Intelligence at Lingnan University. He received the B.E. degree in Software Engineering from Jilin University in 2014, and the Ph.D. degree in Computer Science from the Hong Kong Polytechnic University in 2019. He was a visiting scholar at the Media Lab, Massachusetts Institute of Technology in 2017. His research interests include mobile computing, data mining, and IoT systems. His research has been published in top-tier journals such as IEEE TMC, ACM TOIS, ACM IMWUT, and IEEE TKDE. He was awarded conference best paper twice including one from IEEE INFOCOM 2020.

Xingwei Wang (Member, IEEE) received the B.S., M.S., and Ph.D. degrees in computer science from Northeastern University in 1989, 1992, and 1998, respectively. He is currently a Professor with the School of Computer Science and Engineering, Northeastern University. He has published more than 100 journal articles, books, book chapters, and refereed conference papers. His research interests include cloud computing, future internet, and others. He has received several best paper awards.



Jiannong Cao (Fellow, IEEE) received the M.Sc. and Ph.D. degrees in computer science from Washington State University, Pullman, WA, USA, in 1986 and 1990, respectively. He is currently a Chair Professor with the Department of Computing, The Hong Kong Polytechnic University (PolyU), Hong Kong. He is also the Dean of Graduate School, the Director of Research Institute of Artificial Intelligent of Things, and the Internet and Mobile Computing Lab, and the Vice Director of the University's Research Facility in Big Data Analytics, PolyU. He has coauthored five books, coedited nine books, and authored or coauthored over 500 papers in major international journals and conference proceedings. His research interests include distributed systems and blockchain, wireless sensing and networking, Big Data and machine learning, and mobile cloud and edge computing.