RGS-Sketch: An Accurate, Invertible, and Mergeable Sketch for Online Super Spreader Detection in High-speed Data Streams

Boyu Zhang
Soochow University
byzhang22@stu.suda.edu.cn

He Huang*
Key Laboratory of Data Intelligence and Advanced Computing in Provincial Universities, Soochow University
huangh@suda.edu.cn

Yu-E Sun
Soochow University
sunye12@suda.edu.cn

Guoju Gao
Soochow University
gjgao@suda.edu.cn

ABSTRACT

Super spreader detection in high-speed data streams is crucial for numerous applications. Although many methods have emerged, existing works can hardly concurrently achieve high memory efficiency, support online detection, enable merging data from different measurement points/periods, and offer invertibility. This makes them unable to satisfy flexible application requirements. This paper proposes RGS-Sketch, a novel sketch designed to address this problem. The core of RGS-Sketch lies in a new mergeable memory sharing design called register group sharing. This design organizes registers into groups as basic memory sharing units, accommodating the high skewness of real-world data streams and offering high memory efficiency. Besides, it enables online detection through the real-time acquisition of a group's state, which also facilitates invertibility. To enhance detection accuracy further, we propose a limited register update strategy. It blocks small flows from updating registers, thereby reducing memory overhead and estimation noises. Extensive experimental results based on four real-world datasets show that RGS-Sketch significantly outperforms the most accurate baselines in accuracy while maintaining a high throughput. Specifically, it improves the F1 scores by up to 0.643 for measurements at a single point/period and up to 0.472 across multiple points/periods.

PVLDB Reference Format:
Boyu Zhang, He Huang, Yu-E Sun, and Guoju Gao. RGS-Sketch: An Accurate, Invertible, and Mergeable Sketch for Online Super Spreader Detection in High-speed Data Streams. PVLDB, 18(4): 1237 - 1249, 2024.
doi:10.14778/3717755.3717779

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/duya886/RGS-Sketch.

1 INTRODUCTION

Super spreader detection in high-speed data streams is fundamental for many applications, such as database query optimization [14], web caching prioritization [1], search engines [16], malicious network attack detection [10, 21, 34], and network hot-spot localization [31]. For example, super spreaders can be hosts with an abnormally large number of distinct connections, indicating various networking issues such as worm propagation or DDoS attacks. In a general model [37], a data stream consists of a sequence of data items that form multiple sub-streams, also called flows. Each data item is a pair composed of a flow ID and an element. The definitions of flow IDs and elements are flexible, depending on the specific application interests, such as IP addresses, user IDs, and product IDs. The number of distinct elements within a flow is called flow cardinality (spread). Consequently, detecting super spreaders can be modeled as identifying the flows with large cardinalities.

Detecting super spreaders in high-speed data streams faces three critical challenges. Firstly, memory constraints pose a significant hurdle. It is desirable to access only high-speed cache memory (e.g., SRAM) to ensure a high item processing throughput. However, the cache memory is limited in capacity and must be shared by many functions, necessitating highly memory-efficient methods. Secondly, mergeability is indispensable to accommodate flexible application requirements. For instance, we can merge the data from multiple separate points for parallel collaborative measurement [35]. Another case is merging the algorithm instances in different measurement periods to detect over longer timescales. Otherwise, multiple instances deployed for different time ranges must run simultaneously, causing much larger overhead. Thirdly, online detection is crucial for real-time applications [19]. However, calculating cardinalities is complex as it involves remembering all elements and eliminating duplicates. To fit in the limited memory, sketches, a family of probabilistic data structures that employ hashing techniques for summarizing stream data, are widely employed. For example, sketches for estimating a single flow's cardinality, such as bitmap [39] and HyperLogLog [13], usually serve as basic sharing units in super spreader detection. To balance memory and accuracy, most sketches require scanning hundreds of bits or registers (small-sized counters) for estimation, which is time-consuming and unsuitable for real-time applications.

Many research efforts have been devoted to super spreader detection, which can be broadly categorized into invertible and non-invertible methods. Non-invertible methods prioritize minimizing memory consumption and estimation noises [7, 37, 43]. However, to locate the super spreaders, they necessitate prior knowledge of flow IDs. Besides, they also suffer from the over-estimation problem

*He Huang is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 18, No. 4 ISSN 2150-8097.
doi:10.14778/3717755.3717779

caused by hash collisions when querying on numerous small flows (i.e., with low cardinalities). Invertible methods, on the other hand, can recover flow IDs just from their own data structures. Most of them rely on uniform-sized cardinality estimators, whose size is determined by the possible maximum flow cardinality [2, 27, 35, 38]. Unfortunately, in real-world data streams, most flows are small, and only a few are large (i.e., with high cardinalities) [15, 19, 20]. Thus, most cardinality estimators cannot fully utilize their memory space, causing memory waste. ExtendedSketch [20] and ExtendedSketch+ [15] dynamically expand small bitmaps to accommodate the skewness, but the memory pointers in their design introduce significant memory overhead. FreeBS-SSD and FreeRS-SSD [19] support online super spreader detection and achieve high memory efficiency through bit/register memory sharing. However, their data structures are non-mergeable, limiting their flexibility. In short, existing work can hardly address the challenges simultaneously.

To this end, we propose *RGS-Sketch*, a novel sketch that simultaneously achieves high memory efficiency, supports online detection, maintains mergeability, and offers invertibility. The core of **RGS**-Sketch lies in a new memory sharing strategy called ***R**egister **G**roup **S**haring*. Specifically, we assign each flow a virtual estimator composed of registers that are organized into uniformly sized groups. Then, each group in the virtual estimator maps to a group in a shared physical register pool. Through this strategy, small flows consume only a few physical registers, while larger flows consume more. Thus, it accommodates the high skewness of real-world data streams, thereby ensuring high memory efficiency. Furthermore, each group is stored within a machine word, allowing a single memory access to retrieve all registers in the group. Then, we can get a real-time estimation of the flow cardinality increment based on the group's state. With buckets that record the IDs and accumulated cardinality estimations of candidate super spreaders, RGS-Sketch provides invertibility and online detection capability. Also, we can merge the register pools from different measurement points/periods and estimate the aggregated flow cardinalities. In addition, we propose the *limited register update strategy* to enhance the algorithm's accuracy further. This technique restricts register updates only to elements of candidate super spreaders. Therefore, it can reduce the memory overhead and estimation noises caused by the numerous small flows. We perform a detailed theoretical analysis and conduct extensive experiments based on four real-world datasets. Compared with the state-of-the-art algorithms, RGS-Sketch significantly improves the F1 scores while maintaining a high throughput.

2 BACKGROUND AND RELATED WORK

2.1 Problem Statement

We consider a general model where a data stream consists of a sequence of data items that form multiple sub-streams, also called flows. Each data item is a pair composed of a flow ID and an element. The flow ID and element definitions are configured according to application requirements, such as IP addresses, user IDs, and product IDs. Measurement is typically conducted in each pre-defined period for a continuous data stream, such as a packet stream on the Internet. Let $F = \{f_1, f_2, f_3, ...\}$ and $E = \{e_1, e_2, e_3, ...\}$ separately represent the flow ID set and element set. Each data item can be denoted as (f, e), where $f \in F, e \in E$. Consequently, each flow is a sub-stream comprising all data items with its ID, and a flow's cardinality is defined as the number of distinct elements in it.

Super spreaders are defined as flows whose cardinality exceeds a pre-defined threshold. Let n_f represent the cardinality of flow f, and n represent the total cardinality of all flows, e.g., $n = \sum_{f \in F} n_f$. Then, a flow is a super spreader if $n_f \geq T$, where T is a pre-defined threshold. According to application needs, T can be a constant or a variable ϕn, where ϕ is a pre-defined fraction threshold. In practice, we usually employ a separate cardinality estimator with tiny memory space to estimate n.

2.2 Related Work

Two main techniques are widely employed in super spreader detection: sampling and sketches. The sampling-based methods usually capture large flows by sampling elements, and then utilize a hash table to record the captured flows and sampled elements [5, 22, 33, 36]. Some studies have pointed out that these methods inherently exhibit poor accuracy and high memory consumption [26]. Non-duplicate sampling [17] and the further work [8, 9] utilize an on-chip/off-chip model and record the captured flows and sampled elements in the off-chip memory (e.g., DRAM) to reduce the demand for the on-chip memory. However, the sampling probability is constrained by the update speed of the off-chip memory and the communication speed between the on-chip and off-chip components.

Sketch-based methods have gained wide attention due to their high memory efficiency and accuracy. Random Aging Streaming Filter [43] detects super spreaders using a two-dimensional bit array with aging operations. BACON Sketch [7] combines Count-Min Sketch [6] and Direct Bitmap [11] to estimate flows' cardinalities. Randomized Error-reduction Sketch [37] splits the noises caused by estimator-level memory sharing into two parts and removes noises by subtracting the complement from the primary part. Geometric-Min Filter [28] identifies the super spreaders by recording the arrival flows' maximum geometric hash values in a two-dimensional counter array. Geometric-Min Filter itself is not invertible. Furthermore, it does not provide cardinality estimations. The methods above lack invertibility, necessitating the adoption of additional mechanisms for recording flow IDs.

Invertible sketches can recover the IDs of super spreaders from their own data structures. This invertibility can be achieved through two primary ways. The first way is to record ID information using indexes and reconstruct flow IDs after the measurement. Connection Degree Sketch [38] employs a three-dimensional bit array for cardinality estimation and reconstructs super spreader IDs based on the Chinese Remainder Theorem. Double Connection Degree Sketch [38] further introduces an additional valid sketch to reduce the false positives when reconstructing IDs. Vector Bloom Filter [27] extracts bits from flow IDs as indexes and recovers the flow IDs by overlapping the indexes. The above methods are not memory-efficient as they ignore the highly skewed flow cardinality distribution when allocating estimators' memory. ExtendedSketch [20] aims to accommodate the skewness by dynamically extending bitmaps. Specifically, it constructs a two-dimensional bucket array where each bucket contains an extensible bitmap for cardinality estimation. Besides, it uses the Chinese Remainder Theorem for ID reconstruction. However, ExtendedSketch confronts

substantial memory overhead of pointers and challenges in predicting final memory consumption. Moreover, reconstructing flow IDs requires enumerating all possible index combinations, which is time-consuming and prone to generating numerous fake flow IDs.

The second way to achieve invertibility is directly storing flow IDs in memory. Since the memory is limited, the number of recorded flows is constrained. Thus, flows must compete for opportunities to be recorded. SpreadSketch [35] enhances Count-Min Sketch by equipping each bucket with a multiresolution bitmap [11], an ID field, and a hash value level field for competition. FlowFight [2] uses a fixed number of HyperLogLog estimators [13] and maintains a sorted flow list based on rough cardinality estimations. These methods allocate the same memory to each cardinality estimator, thus not memory-efficient. Like ExtendedSketch, ExtendedSketch+ [15] also builds upon the idea of dynamically extending bitmaps. Except for the extensible bitmap, it adds an ID field and a possibility index to each bucket for flow competition. Besides, it adds a quotient set for each bit in the bitmap to ensure a lossless extension strategy. However, the sets also lead to more considerable memory overhead than ExtendedSketch. FreeBS-SSD and FreeRS-SSD [19] achieve high memory efficiency by sharing memory among all flows at a bit/register level. Specifically, they employ a bit/register pool for cardinality estimation and a Stream-Summary [29] for recording the IDs and estimated cardinalities of candidate super spreaders. Each time a flow element arrives, it will be hashed to one bit/register within the whole pool, and the flow can immediately get an increment value of the estimated cardinality. Since large flows tend to use more bits/registers, their designs accommodate the skewed flow cardinality distribution. However, neither FreeBS-SSD nor FreeRS-SSD are mergeable due to their sharing strategies.

3 ALGORITHM DESIGN

3.1 Main Idea

The main idea of RGS-Sketch lies in organizing contiguous registers within a machine word into a group for memory sharing. This design provides fine-grained memory sharing and enables online cardinality tracking. Furthermore, this design is also amenable to merging operations.

Accurately tracking large flows is imperative to facilitate precise super spreader detection. In other words, we must accurately determine which flows surpass others in cardinality during measurement. To this end, we adopt a real-time cardinality tracking technique rooted in computing real-time state-changing probabilities. The basic concept is as follows. Consider a register array whose state can be changed by a new element with a probability p. Suppose a new element alters the array at the current moment. In that case, we can increment the corresponding flow's cardinality estimation by p^{-1}. Thus, we can provide real-time estimations with a structure that accumulates the estimations of flows' cardinality increments.

Considering the requirement for mergeability, one typical memory sharing strategy is vHLL [40], which is designed to estimate the cardinalities of all flows. This strategy assigns each flow a virtual HyperLogLog estimator composed of registers, and each register is randomly drawn from a register pool. Then, we can leverage HyperLogLog's methodology for cardinality estimation. In this strategy, we can merge multiple register pools by taking the maximum values

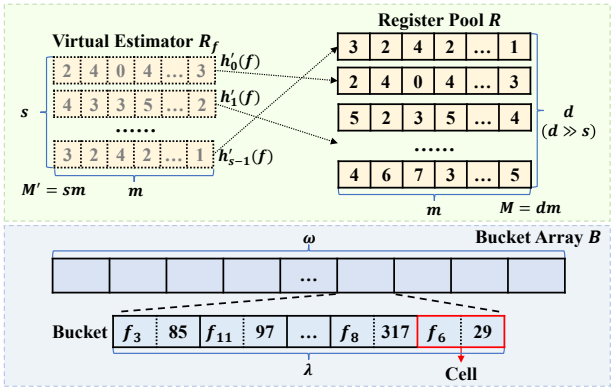

Figure 1: The data structure of RGS-Sketch.

in the corresponding registers. However, when estimating a flow's cardinality, vHLL needs to read all registers within the virtual estimator, involving computing hash functions and accessing discrete memory hundreds of times. This high estimation overhead makes it impractical to capture the super spreaders during measurement. Thus, it is unsuitable for super spreader detection.

In the design of **RGS**-Sketch, we propose ***R**egister **G**roup **S**haring* to reconcile real-time state-changing probability calculation and mergeable fine-grained memory sharing. This strategy organizes registers into groups, where each group is a contiguous memory block. Subsequently, each flow is assigned a virtual HyperLogLog estimator comprised of register groups randomly selected from a physical register pool. Instead of traversing the entire virtual estimator, we only examine one group to compute the group's state-changing probability. When a group is confined to a single machine word, it only involves two hash function calculations to determine the physical group index and one memory access to retrieve the content. Thus, the probability calculation process can be executed in real time. Note that we choose to employ the probability of a group rather than a single register to utilize the averaging effect. This design avoids the large estimation errors caused when items belonging to small flows coincidently alter a register with a small state-changing probability. Meanwhile, this register group sharing approach can still provide high memory efficiency due to the small group size. Additionally, it retains the ability to perform the merging operations based on HyperLogLog's estimation method.

3.2 Data Structure

As depicted in Figure 1, the data structure of RGS-Sketch includes a bucket array B containing ω buckets and a register pool R consisting of M registers. Each bucket comprises λ cells, each including two fields: an ID field for recording the flow ID and a counter field for tracking flow cardinality. We use $B[i][j].ID$ and $B[i][j].C$ to represent the ID and counter field of the j-th cell in the i-th bucket, respectively. The register pool R can be seen as a two-dimensional register array arranged in d rows and m columns, resulting in a total of $M = dm$ registers. Each register can be seen as a small-sized counter, whose size is typically 5 bits like the configuration in HyperLogLog. Within the register pool, we refer to the set of m registers aligned horizontally as a register group, ensuring contiguous

memory allocation for each group. Note that the memory of continuous groups is contiguous in implementation, with no wasted space between the groups. At the beginning of each measurement period, bucket array B and register pool R must be initialized, ensuring that all fields and registers are set to 0.

During the measurement process, each flow f is assigned a virtual HyperLogLog estimator R_f comprising s virtual register groups, where $s \ll d$. Consequently, R_f has $M' = sm$ virtual registers. As configured in HyperLogLog, to facilitate updating the virtual HyperLogLog, M' should be a power of 2. Then, m and s should also be powers of 2. We term these estimators and register groups as "virtual" to emphasize that they exist solely in a logical sense, without incurring any additional memory overhead. Specifically, each virtual group is randomly mapped to a physical group within R through a hash function. This strategy ensures that all flows share the register pool R at a level of register groups.

3.3 Update Operation

The pseudo-code for the update operation is shown in Algorithm 1. For each arrival item (f, e), we first calculate three values: virtual group index $g_v = h(f, e) \mod s$, register index $r = \lfloor h(f, e)/s \rfloor \mod m$ in the group, and the remaining string $\lfloor h(f, e)/(sm) \rfloor$, where $h(\cdot)$ is a hash function that outputs a 32-bit string. The virtual group $R_f[g_v]$ is mapped to a physical group $R[g]$ through $g = h'_{g_v}(f)$, where $h'_i(\cdot)$ are hash functions with a range of $[0, d)$. The hash functions $h'_i(\cdot)$ can be implemented as follows:

$$h'_i(f) = h'(f \oplus C[i]) \text{ or } h'_i(f) = h'(f \mid i), 0 \le i < s, \quad (1)$$

where $h'(\cdot)$ is a master function, $\oplus$ represents the XOR operator, C is an array containing distinct constant integers that are randomly generated, and $\mid$ denotes the concatenation operator.

Subsequently, an element rank value o is calculated using the function $\rho(\lfloor h(f, e)/(sm) \rfloor)$. The function $\rho(\cdot)$ returns one plus the count of leading zeros in the input binary string. For instance, $\rho(10\ldots_2) = 1, \rho(01\ldots_2) = 2$. When the input equals 0, $\rho(\cdot)$ gets the maximum output o_{max} that equals one plus the length of the input binary string. If o is not larger than $R[g][r]$, no further action is taken for the current item. Otherwise, we need to update the bucket array and the register pool. The state-changing probability of group $R[g]$ must be calculated to perform the updates. For an arbitrary item mapped to $R[g]$, the probability that o exceeds the value stored in the corresponding register is calculated as follows:

$$\begin{aligned} p &= \sum_{j=0}^{m-1} \Pr(r = j \wedge o > R[g][j]) \\ &= \frac{1}{m} \sum\nolimits_{0 \le j < m, R[g][j] \ne o_{max}} 2^{-R[g][j]}. \end{aligned} \quad (2)$$

Then, we access bucket array B to update the estimated cardinality of flow f using p. Note that we do not immediately update the register $R[g][r]$. Instead, the decision is deferred until after processing bucket array B. The bucket index of flow f is determined by $b = h''(f)$, where $h''(\cdot)$ is a hash function whose range is $[0, \omega)$. Depending on the mapped bucket $B[b]$, we apply different operations in the following three cases:

Case 1. If a cell $B[b][c]$ recording flow f exists, we increment the cell's counter filed $B[b][c].C$ by p^{-1}. Note that p^{-1} is typically a decimal, which introduces problems in the counter updates. We will discuss it in Section 3.7.

Case 2. No cell recording flow f exists, while empty cells are available in bucket $B[b]$. Assuming the first empty cell is $B[b][c]$, we set its ID field and counter field to f and p^{-1} separately, i.e., let $B[b][c].ID = f$ and $B[b][c].C = p^{-1}$.

Case 3. There is no cell recording flow f, and no empty cell remains in bucket $B[b]$. In this case, we adopt a probability-based replacement strategy to replace the flow ID recorded in the smallest cell. Assuming the cell with the smallest counter is $B[b][c']$, we set $B[b][c'].ID$ to f with a probability of $(B[b][c'].C + 1)^{-1}$. We utilize a hash function $h'''(\cdot)$ with a range of $[0, X)$ to implement this probabilistic replacement, where X is a large enough constant. If $h'''(f, e)$ is less than $(B[b][c'].C + 1)^{-1} X$, we replace the value of $B[b][c'].ID$ as well as increment $B[b][c'].C$ by one. Otherwise, the content of $B[b][c]$ will not be changed.

It can be seen that when updating the bucket array, we only need one hash function to locate the bucket and continuous memory accesses to retrieve the cells in it. In contrast, Stream-Summary [29], the Count-Min-like structures [6, 15, 20, 29], and the structures [12, 32] based on Cuckoo Hashing [30] require multiple discrete memory accesses. The updating operations for these structures are more time-consuming. Moreover, Stream-Summary includes numerous memory pointers, harming its memory efficiency. Thus, we choose the bucket array structure for recording the candidate super spreaders. After updating the bucket array, we determine whether to update the register $R[g][r]$. Conventionally, $R[g][r]$ will always be set to the element rank value o. However, this strategy lets the large number of small flows that are unlikely to be super spreaders

Algorithm 1: Update Operation

Input: data item (f, e)

flag $\leftarrow$ false; // indicate if f has been recorded
$g_v \leftarrow h(f, e) \mod s$; $r \leftarrow \lfloor h(f, e)/s \rfloor \mod m$;
$g \leftarrow h'_{g_v}(f)$; $o \leftarrow \rho(\lfloor h(f, e)/(sm) \rfloor)$;
if $o \le R[g][r]$ **then**
  return;
$p \leftarrow \frac{1}{m} \sum_{0 \le j < m, R[g][j] \ne o_{max}} 2^{-R[g][j]}$; $b \leftarrow h''(f)$;
$c' \leftarrow 0$; // capture the smallest cell
for $c \leftarrow 1$ **to** λ **do**
  if $B[b][c].ID = f$ **then** // Case 1
    Increase$(B[b][c].C, p^{-1})$; flag $\leftarrow$ true;
    break;
  if $B[b][c].C = 0$ **then** // Case 2
    $B[b][c].ID \leftarrow f$; Increase$(B[b][c].C, p^{-1})$;
    flag $\leftarrow$ true;
    break;
  if $B[b][c].C < B[b][c'].C$ **then**
    $c' \leftarrow c$;
if flag = false and $h'''(f, e) < \frac{X}{B[b][c'].C+1}$ **then** // Case 3
  $B[b][c'].ID \leftarrow f$; Increase$(B[b][c'].C, 1)$;
  flag $\leftarrow$ true;
if flag = true **then**
  $R[g][r] = o$;

consume significant memory. To solve this problem, we propose a novel *limited register update strategy*. It updates $R[g][r]$ to o only if flow f has been recorded in bucket array B after the previous processing. In other words, this strategy only allows the candidate super spreaders to perform register updates. Through this limited register update strategy, we can avoid the unnecessary memory consumption caused by small flows, thereby enhancing our memory efficiency compared with the traditional approach.

3.4 Merging Operation

Multiple instances of RGS-Sketch, deployed at different measurement points/periods, can be merged to detect the super spreaders within the aggregated data stream. Given a set of I RGS-Sketches RGS^0, RGS^1, ..., RGS^{I-1} sharing identical register-related parameters (e.g., s, d, m, and hash functions), the merging operation is:

$$RGS.R[i][j] = \max_{0 \le k < I} RGS^k.R[i][j], \tag{3}$$

where $RGS.R[i][j]$ and $RGS^k.R[i][j]$ separately represent the j-th register in the i-th group of the merged and the k-th RGS-Sketch. The bucket arrays remain separate and provide flow IDs for the query based on the merged register pool.

3.5 Query Operation

RGS-Sketch includes two query techniques: counter-based query and register-based query, which are employed in different scenarios. The former is employed to estimate flow cardinalities at a single measurement point/period. Meanwhile, the latter estimates flow cardinalities across multiple points/periods.

3.5.1 Counter-based Query. To query a flow f, we first locate its mapped bucket $B[h''(f)]$ and then scan the cells in it. If there exists a cell $B[h''(f)][c]$ recording flow f, we employ the cell's counter value as f's cardinality estimation $\hat{n}_f$, i.e., $\hat{n}_f = B[h''(f)][c].C$. Otherwise, we conservatively estimate its cardinality as 0 if flow f is not recorded.

3.5.2 Register-based Query. This query is performed on the register pool of the sketch RGS merged from I RGS-Sketches RGS^0, RGS^1, ..., RGS^{I-1}. The query method is similar to that of vHLL [40]. Given flow f, we construct its virtual estimator R_f as follows: $R_f[i][j] = R[h'_i(f)][j]$ for $0 \le i < s$ and $0 \le j < m$. Due to memory sharing, R_f records not only the elements of flow f but also those of other flows, which introduces noises. Let $n_{(R)}$ and $n_{(R_f)}$ be the total number of distinct elements recorded by R and R_f, respectively. Additionally, let $n_{f_{(R)}}$ represent the number of f's recorded distinct elements. The recorded elements refer to the elements that arrive when their flows are recorded in the bucket array (including the elements that put their flows into the bucket array), thereby not discarded by the limited register update strategy. Thus, $n_{(R)} \le n$ and $n_{f_{(R)}} \le n_f$. The noise $n_{(R_f)} - n_{f_{(R)}}$ approximately follows the binomial distribution $Bino\left(n_{(R)} - n_{f_{(R)}}, \frac{s}{d}\right)$. Hence, we have

$$\mathbb{E}\left(n_{(R_f)} - n_{f_{(R)}}\right) = \frac{s\left(n_{(R)} - n_{f_{(R)}}\right)}{d}. \tag{4}$$

We can approximate $n_{(R_f)} - n_{f_{(R)}}$ using this expectation. Then,

$$n_{f_{(R)}} \approx \frac{ds}{d-s}\left(\frac{n_{(R_f)}}{s} - \frac{n_{(R)}}{d}\right). \tag{5}$$

It is worth noting that the discrepancy between the estimations of $n_{f_{(R)}}$ and n_f is insignificant. On the one hand, large flows tend to be rapidly captured in the bucket array, thereby minimally impacted by our limited register update strategy. On the other hand, without our strategy, small flows usually struggle to receive accurate estimates due to the overwhelming noises from other flows. In contrast, our strategy effectively mitigates this issue by reducing the noises. By replacing $n_{(R_f)}$ and $n_{(R)}$ with their estimations $\hat{n}_{(R_f)}, \hat{n}_{(R)}$ and estimate n_f using the estimation of $n_{f_{(R)}}$, we have

$$\hat{n}_f = \frac{ds}{d-s}\left(\frac{\hat{n}_{(R_f)}}{s} - \frac{\hat{n}_{(R)}}{d}\right). \tag{6}$$

Using the HyperLogLog estimation formula, we can get

$$\hat{n}_{(R_f)} = \alpha_{M'}\left(M'\right)^2\left(\sum_{i=0}^{s-1}\sum_{j=0}^{m-1} 2^{-R_f[i][j]}\right)^{-1}, \tag{7}$$

$$\hat{n}_{(R)} = \alpha_M M^2\left(\sum_{i=0}^{d-1}\sum_{j=0}^{m-1} 2^{-R[i][j]}\right)^{-1}. \tag{8}$$

Here, α_x is a bias correction constant associated with variable x:

$$\alpha_x = \left(x\int_0^\infty \left(\log_2\left(\frac{2+u}{1+u}\right)\right)^x du\right)^{-1}. \tag{9}$$

Due to the complexity of this formula, it is common practice to employ numerical approximations in real-world applications. Specifically, we often use $\alpha_{16} = 0.673, \alpha_{32} = 0.697, \alpha_{64} = 0.709$, and $\alpha_x = 0.7213\left(1 + 1.079x^{-1}\right)^{-1}$ for $x \ge 128$.

While Formulas (7) and (8) are typically accurate, they may yield large errors when the estimated values are extremely small or large, which need to be corrected. Take $\hat{n}_{(R_f)}$ as an example. The corrected formula is as follows:

$$\hat{n}_{(R_f)} = \begin{cases} M'\ln\left(M'/V'\right) & \hat{n}_{(R_f)} \le 2.5M' \text{ and } V' > 0 \\ -2^{32}\ln\left(1 - \frac{1}{2^{32}}\hat{n}_{(R_f)}\right) & \hat{n}_{(R_f)} \ge \frac{1}{30}2^{32} \\ \hat{n}_{(R_f)} & \text{otherwise} \end{cases}, \tag{10}$$

where V' represents the number of registers equal to 0 in R_f. A similar correction is applied to $\hat{n}_{(R)}$ by replacing $\hat{n}_{(R_f)}$, M', and V' in the above formula with $\hat{n}_{(R)}$, M, and V respectively, where V denotes the number of registers equal to 0 in R.

In the context of a single RGS-Sketch, the counter-based query offers an estimate by leveraging all state changes of R_f, while the register-based query can only rely on the final state. Additionally, memory sharing makes the register-based query susceptible to noises introduced by other flows. As a result, the counter-based query usually yields superior accuracy, which will be proved in Section 4.3. Recall that the register-based query is based on I RGS-Sketches RGS^0, RGS^1, ..., RGS^{I-1}. We can leverage the counter-based query results from the I RGS-Sketches to enhance estimation accuracy further. Let $\hat{n}_f^{(C)}[k]$ represent the counter-based query

result of flow f on the k-th RGS-Sketch, and $\min[k]$ represent the value of the smallest counter in the bucket array of the k-th RGS-Sketch. We can then refine our estimation using the upper and lower bounds:

$$\hat{n}_f = \begin{cases} U & \hat{n}_f > U \\ L & \hat{n}_f < L \\ \hat{n}_f & \text{otherwise} \end{cases}, \tag{11}$$

where $U = \sum_{k=0}^{I-1} \max\{\hat{n}_f^{(C)}[k], \min[k]\}$ and $L = \max_{0 \le k < I} \hat{n}_f^{(C)}[k]$.

3.6 Super Spreader Detection

3.6.1 Detection at a Single Measurement Point/Period. When measurements are conducted at a single point/period, RGS-Sketch supports online and offline super spreader detection. For online detection, after recording each item (f, e), we can use the counter-based query to get the cardinality estimation $\hat{n}_f$ of flow f. If $\hat{n}_f \ge T$, flow f is promptly reported as a super spreader. For offline detection, we systematically scan each cell $B[b][c]$ to retrieve flow ID $B[b][c].ID$ and the flow's cardinality estimation $B[b][c].C$. If $B[b][c].C \ge T$, we report $B[b][c].ID$ as a super spreader.

3.6.2 Detection across Multiple Measurement Points/Periods. Suppose we have collected I RGS-Sketches $RGS^k, 0 \le k < I$ from the I measurement points/periods. Initially, we systematically scan the bucket arrays of all I RGS-Sketches to obtain a set of recorded flow IDs $F' = \{RGS^k.B[b][c].ID, 0 \le k < I, 0 \le b < \omega, 0 \le c < \lambda\}$. Subsequently, for each flow $f \in F'$, we leverage Formula (11) to estimate its cardinality $\hat{n}_f$. If $\hat{n}_f \ge T$, flow f is promptly identified as a super spreader.

3.7 Discussion on Counter Updates

In Cases 1 and 2 of the update operation, the counter fields to be updated will be increased by p^{-1}. Since p^{-1} is typically a decimal, we cannot simply configure the counter fields as integers. One choice is configuring each counter field as a float-type or a double-type variable. However, in modern machines, a float-type variable usually has only 6 or 7 significant digits, which means large counters may ignore the small increment of p^{-1}. In contrast, a double-type variable has enough significant digits but usually takes up 64 bits and is not memory-efficient. Another choice is configuring the counter fields as integers and using the rounded value of p^{-1} as updates. Unfortunately, the rounded value of p^{-1} is not an unbiased approximation when p is not a uniform random variable. For example, when the recorded elements increase and p decreases from 1 to $\frac{2}{3}$ (not included), the rounded value of p^{-1} will always be $\lfloor p^{-1} \rfloor$. To overcome this issue, we employ a sampling method to update the counter fields when configuring them as integers. Initially, we set the update value to $\lfloor p^{-1} \rfloor$. Subsequently, we increment the update value by one with a probability of $p^{-1} - \lfloor p^{-1} \rfloor$. Finally, we use this update value to update the counters. Let U be a random variable representing the update value. We can get $E(U) = (\lfloor p^{-1} \rfloor + 1)(p^{-1} - \lfloor p^{-1} \rfloor) + \lfloor p^{-1} \rfloor (1 - p^{-1} + \lfloor p^{-1} \rfloor) = p^{-1}$. In other words, this method provides an unbiased approximation. Thus, we adopt this method instead of the rounded value to accurately approximate the value of p^{-1} while maintaining high memory efficiency.

3.8 Optimization on Number of Hash Functions

In the update operation of RGS-Sketch, the execution of four hash functions (i.e., $h(f, e)$, $h'_i(f)$, $h''(f)$, and $h'''(f, e)$) could be expensive in time cost. To mitigate this computational burden, we employ a technique inspired by Kirsch et al.'s work [25] that reduces the number of hash functions to calculate. Specifically, we utilize two hash functions, $H(f)$ and $H'(e)$, to simulate the four hash functions mentioned above. The simulation is achieved as follows:

$$\begin{aligned} h(f, e) &= H(f) + H'(e), \\ h'_i(f) &= H(f) \oplus C[i], 0 \le i < s \\ h''(f) &= H(f), \\ h'''(f, e) &= H(f) + C[s] H'(e), \end{aligned} \tag{12}$$

where $\oplus$ denotes the XOR operator, and C is an array consisting of $s + 1$ randomly generated distinct constant integers larger than 1. Leveraging this approach, we only need to execute two actual hash functions for each arrival item, significantly decreasing the hash computation cost.

4 THEORETICAL ANALYSIS

This section formally analyzes the performance of counter-based and register-based queries. In particular, the register-based query in this section refers to the query method without Formula (11). We give the expectation and variance of each query method's estimation results for super spreaders. Finally, we prove that the counter-based query is more accurate than the register-based query.

4.1 Analysis on Counter-based Query

In real-world data streams, the cardinality distribution is generally highly skewed [15, 19, 20]. Thus, the number of counters in the bucket array is usually much larger than that of super spreaders. In such a case, the likelihood of super spreaders colliding with each other can be negligible. Moreover, super spreaders' cardinalities are usually hundreds to tens of thousands of times greater than those of small flows. A super spreader can easily be inserted into the cell with the smallest counter, even if its mapped bucket is full. Subsequently, the counter will promptly increase its value and cease to be the smallest one in the bucket. Therefore, the flow competition process has little impact on the accuracy of super spreaders. Due to the complexity of the dynamic flow competition process, we draw inspiration from the typical works [15, 18, 19, 35, 41, 42, 44] and assume that each super spreader has an individual counter to track its cardinality estimation to simplify the analysis.

We use the time notation (t) as superscripts to explain the process of state changes. Let $\mathcal{T}_f^{(t)}$ be the set of the first occurrence times of flow f's elements in the data stream until time t. Then, at time $i \in \mathcal{T}_f^{(t)}$, flow f's actual cardinality, its estimation, and the sum of all flow's cardinality are denoted as $n_f^{(i)}$, $\hat{n}_f^{(i)}$, and $n^{(i)}$, respectively. In addition, the state-changing probability of the group mapped by a new element arriving at time i is denoted as $p^{(i)}$.

Theorem 4.1. *For a super spreader f, the expectation and variance of the counter-based query result are*

$$\mathbb{E}\left(\hat{n}_f^{(t)}\right) = n_f^{(t)}, \tag{13}$$

$$Var\left(\hat{n}_f^{(t)}\right) = \sum_{i\in\mathcal{T}_f^{(t)}} \mathbb{E}\left(\frac{1}{p^{(i)}}\right) - n_f^{(t)}, \tag{14}$$

where $\mathbb{E}\left(\frac{1}{p^{(i)}}\right) \approx \frac{1}{\alpha_m M}\left(\frac{d-s}{s}n_f^{(i)} + n^{(i)}\right)$ *when* $\left(\frac{d-s}{s}n_f^{(i)} + n^{(i)}\right) > 2.5M$, *and* α_m *is calculated by Formula (9).*

Proof. Let $\delta^{(i)}$ be the indicator variable for the event that register pool R is changed by the element arriving at time i, and $x^{(i)}$ be the random variable representing the state-changing probability of the element's mapped register group. Then, we have

$$\mathbb{E}\left(\delta^{(i)} \,\middle|\, x^{(i)} = p^{(i)}\right) = p^{(i)}, \tag{15}$$

$$Var\left(\delta^{(i)} \,\middle|\, x^{(i)} = p^{(i)}\right) = p^{(i)} - \left(p^{(i)}\right)^2. \tag{16}$$

The estimation of flow f's cardinality is calculated based on the group state-changing probabilities each time its element changes the register pool, namely,

$$\hat{n}_f^{(t)} = \sum_{i\in\mathcal{T}^{(t)}} \frac{\delta^{(i)}}{p^{(i)}}. \tag{17}$$

Note that the random variables $\delta^{(i)}, i \in \mathcal{T}^{(t)}$ are independent of each other. Given a sequence of group state-changing probabilities $x^{(0)}, x^{(1)}, ..., x^{(t)}$, $\delta^{(i)}$ is only associated with the variable $x^{(i)}$ at the current time. Therefore, we have

$$\begin{aligned}
&\mathbb{E}\left(\hat{n}_f^{(t)} \,\middle|\, x^{(0)} = p^{(0)}, x^{(1)} = p^{(1)}, ..., x^{(t)} = p^{(t)}\right) \\
=&\sum_{i\in\mathcal{T}^{(t)}} \frac{\mathbb{E}\left(\delta^{(i)} \,\middle|\, x^{(i)} = p^{(i)}\right)}{p^{(i)}} \\
=&\sum_{i\in\mathcal{T}^{(t)}} \frac{p^{(i)}}{p^{(i)}} = n_f^{(t)}.
\end{aligned} \tag{18}$$

Then, we can get

$$\begin{aligned}
&\mathbb{E}\left(\hat{n}_f^{(t)}\right) \\
=&\mathbb{E}\left(\mathbb{E}\left(\hat{n}_f^{(t)} \,\middle|\, x^{(0)} = p^{(0)}, x^{(1)} = p^{(1)}, ..., x^{(t)} = p^{(t)}\right)\right) \\
=&\mathbb{E}\left(n_f^{(t)}\right) = n_f^{(t)}.
\end{aligned} \tag{19}$$

Next, we calculate the variance of $\hat{n}_f^{(t)}$. We first calculate the variance under given group state-changing probabilities:

$$\begin{aligned}
&Var\left(\hat{n}_f^{(t)} \,\middle|\, x^{(0)} = p^{(0)}, x^{(1)} = p^{(1)}, ..., x^{(t)} = p^{(t)}\right) \\
=&\sum_{i\in\mathcal{T}^{(t)}} \frac{Var\left(\delta^{(i)} \,\middle|\, x^{(i)} = p^{(i)}\right)}{\left(p^{(i)}\right)^2} \\
=&\sum_{i\in\mathcal{T}^{(t)}} \frac{p^{(i)} - \left(p^{(i)}\right)^2}{\left(p^{(i)}\right)^2} = \sum_{i\in\mathcal{T}^{(t)}} \frac{1}{p^{(i)}} - n_f^{(t)}.
\end{aligned} \tag{20}$$

Then, we have

$$\begin{aligned}
Var\left(\hat{n}_f^{(t)}\right) =& Var\left(\mathbb{E}\left(\hat{n}_f^{(t)} \,\middle|\, x^{(0)} = p^{(0)}, ..., x^{(t)} = p^{(t)}\right)\right) \\
&+ \mathbb{E}\left(Var\left(\hat{n}_f^{(t)} \,\middle|\, x^{(0)} = p^{(0)}, ..., x^{(t)} = p^{(t)}\right)\right) \\
=& Var\left(n_f^{(t)}\right) + \mathbb{E}\left(\sum_{i\in\mathcal{T}^{(t)}} \frac{1}{p^{(i)}} - n_f^{(t)}\right) \\
=& \sum_{i\in\mathcal{T}^{(t)}} \mathbb{E}\left(\frac{1}{p^{(i)}}\right) - n_f^{(t)}.
\end{aligned} \tag{21}$$

It is complex to analyze the exact expression for $\mathbb{E}\left(\frac{1}{p^{(i)}}\right)$. Fortunately, we can get an estimation of $\mathbb{E}\left(\frac{1}{p^{(i)}}\right)$ from the estimation formula of HyperLogLog. Specifically, we treat the register group mapped by f's element at time i as a small HyperLogLog estimator. Suppose the register group mentioned above is $R[g]$. We employ $n_{(R[g])}^{(i)}$ to represent the number of distinct elements recorded by $R[g]$ until time i. According to HyperLogLog's theory, when $n_{(R[g])}^{(i)} > 2.5m$, we have

$$\begin{aligned}
n_{(R[g])}^{(i)} \approx& \mathbb{E}\left(\alpha_m m^2 \left(\sum_{j=0}^{m-1} 2^{-R[g][j]}\right)^{-1} \,\middle|\, n_{(R[g])}^{(i)}\right) \\
\approx& \alpha_m m \mathbb{E}\left(\frac{1}{p^{(i)}} \,\middle|\, n_{(R[g])}^{(i)}\right).
\end{aligned} \tag{22}$$

Thus, we have

$$\mathbb{E}\left(\frac{1}{p^{(i)}} \,\middle|\, n_{(R[g])}^{(i)}\right) \approx \frac{n_{(R[g])}^{(i)}}{\alpha_m m}. \tag{23}$$

The elements recorded by $R[g]$ can be divided into two parts: the elements of flow f, and the elements of the other flows. The number of distinct elements in the former part and the latter part approximately follows the binomial distribution $Bino\left(n_f^{(i)}, \frac{1}{s}\right)$ and $Bino\left(n^{(i)} - n_f^{(i)}, \frac{1}{d}\right)$, respectively. Therefore, we can get

$$\mathbb{E}\left(n_{(R[g])}^{(i)}\right) = n_f^{(i)}\frac{1}{s} + \left(n^{(i)} - n_f^{(i)}\right)\frac{1}{d}. \tag{24}$$

Then, we have

$$\begin{aligned}
\mathbb{E}\left(\frac{1}{p^{(i)}}\right) &= \mathbb{E}\left(\mathbb{E}\left(\frac{1}{p^{(i)}} \,\middle|\, n_{(R[g])}^{(i)}\right)\right) \\
&\approx \frac{\mathbb{E}\left(n_{(R[g])}^{(i)}\right)}{\alpha_m m} \\
&= \frac{1}{\alpha_m m}\left(n_f^{(i)}\frac{1}{s} + \left(n^{(i)} - n_f^{(i)}\right)\frac{1}{d}\right) \\
&= \frac{1}{\alpha_m M}\left(\frac{d-s}{s}n_f^{(i)} + n^{(i)}\right),
\end{aligned} \tag{25}$$

when $\left(\frac{d-s}{s}n_f^{(i)} + n^{(i)}\right) > 2.5M$. □

4.2 Analysis On Register-based Query

In the analysis of the register-based query, we still assume that each super spreader has an individual counter to track its cardinality estimation during measurement.

Theorem 4.2. *For a super spreader f, the expectation and variance of the register-based query result are*

$$\mathbb{E}\left(\hat{n}_f^{(t)}\right) = n_f^{(t)}, \tag{26}$$

$$\begin{aligned} &Var\left(\hat{n}_f^{(t)}\right) \\ &\approx \left(\frac{M}{M-M'}\right)^2 \left(\frac{1.04^2}{M'}\left(n_f^{(t)} + \left(n^{(t)} - n_f^{(t)}\right)\frac{M'}{M}\right)^2\right. \\ &+ \left(n^{(t)} - n_f^{(t)}\right)\frac{M'}{M}\left(1-\frac{M'}{M}\right) \\ &\left. + \left(\frac{M'}{M}\right)^2 \frac{1.04^2}{M}\left(n^{(t)}\right)^2\right). \end{aligned} \tag{27}$$

Proof. The proof is very similar to the analysis in Section 6 of vHLL [40]. Due to the page limit, we omit the proof. □

4.3 Discussions

In this part, we prove that the counter-based query outperforms the register-based query in accuracy. This explains why we employ the counter-based query rather than the register-based query for offline estimation when measuring at a single point/period. We employ $\hat{n}_f^{(t,C)}$ and $\hat{n}_f^{(t,R)}$ to represent the estimated results of the counter-based query and register-based query, respectively.

Theorem 4.3. *For a super spreader f, when $m \geq 8$, we have*

$$Var\left(\hat{n}_f^{(t,C)}\right) \lessapprox Var\left(\hat{n}_f^{(t,R)}\right). \tag{28}$$

Proof. From HyperLogLog [13] and its improvement [16], we can find that $\mathbb{E}\left(\alpha_m m^2 \left(\sum_{j=0}^{m-1} 2^{-R[g][j]}\right)^{-1} \middle| n_{(R[g])}^{(i)}\right) \gtrapprox n_{(R[g])}^{(i)}$ when $n_{(R[g])}^{(i)} < 2.5M$. When $n_{(R[g])}^{(i)} = 0$, the difference between them gets the largest, which equals $\alpha_m m$. Then, according to Formula (25), we can derive that, when $\left(\frac{d-s}{s}n_f^{(i)} + n^{(i)}\right) = 0$, the difference between $\mathbb{E}\left(\frac{1}{p^{(i)}}\right)$ and $\frac{1}{\alpha_m M}\left(\frac{d-s}{s}n_f^{(i)} + n^{(i)}\right)$ gets the largest, which equals 1. In other words, we can get $\mathbb{E}\left(\frac{1}{p^{(i)}}\right) \lessapprox \frac{1}{\alpha_m M}\left(\frac{d-s}{s}n_f^{(i)} + n^{(i)}\right) + 1$. Thus, from Theorem 4.1, we have

$$\begin{aligned} Var\left(\hat{n}_f^{(t,C)}\right) &\lessapprox \sum_{i\in\mathcal{T}_f^{(t)}} \left(\frac{1}{\alpha_m M}\left(\frac{d-s}{s}n_f^{(i)} + n^{(i)}\right) + 1\right) - n_f^{(t)} \\ &\leq \sum_{j=1}^{n_f^{(t)}} \frac{(d-s)}{\alpha_m M s} j + \frac{n_f^{(t)} n^{(t)}}{\alpha_m M} \\ &\leq \frac{(d-s)}{2\alpha_m M s}\left(\left(n_f^{(t)}\right)^2 + n_f^{(t)}\right) + \frac{n_f^{(t)} n^{(t)}}{\alpha_m M} \\ &\leq \frac{1}{2\alpha_m M'}\left(n_f^{(t)}\right)^2\left(1 + \frac{1}{n_f^{(t)}}\right) + \frac{n_f^{(t)} n^{(t)}}{\alpha_m M}. \end{aligned} \tag{29}$$

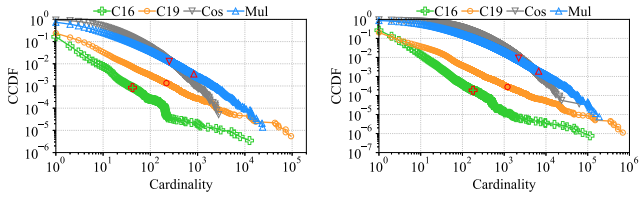

Figure 2: CCDF of the first segments' flow cardinalities (left) and the entire datasets' flow cardinalities (right).

We can get $\alpha_8 \approx 0.6256$ from Formula (9). In addition, according to the numerical approximations, α_m increases with the value of m. Thus, $Var\left(\hat{n}_f^{(t,C)}\right) \lessapprox \frac{0.7992}{M'}\left(n_f^{(t)}\right)^2\left(1 + \frac{1}{n_f^{(t)}}\right) + \frac{1.5985}{M} n_f^{(t)} n^{(t)}$ when $m \geq 8$. In contrast, from Theorem 4.2, we can get

$$\begin{aligned} &Var\left(\hat{n}_f^{(t,R)}\right) \\ &\gtrapprox \left(\frac{M}{M-M'}\right)^2 \frac{1.04^2}{M'}\left(n_f^{(t)} + \left(n^{(t)} - n_f^{(t)}\right)\frac{M'}{M}\right)^2 \\ &\geq \frac{1.04^2}{M'}\left(n_f^{(t)}\right)^2 + 2\frac{1.04^2}{M} n_f^{(t)} n^{(t)} \\ &= \frac{1.0816}{M'}\left(n_f^{(t)}\right)^2 + \frac{2.1632}{M} n_f^{(t)} n^{(t)}. \end{aligned} \tag{30}$$

We can find that $0.7992\left(1 + \frac{1}{n_f^{(t)}}\right)$ is always smaller than 1.0816 when $n_f^{(t)} \geq 3$, which is common for a super spreader. Therefore, Theorem 4.3 holds, proving that the counter-based query is more accurate than the register-based query. □

5 EXPERIMENTAL RESULTS

5.1 Experimental Setup

Dataset: Our experiments utilize four real-world datasets, including two ten-minute network traces and two e-commerce datasets. The network traces, originating from CAIDA [3, 4], were collected in 2016 and 2019, respectively. These traces' flow IDs and elements are defined as packets' source and destination addresses. The e-commerce datasets are separately 5-month e-commerce behavior data from a medium cosmetics shop [24] and 1-month data from a large multi-category online store [23]. We use the product IDs as flow IDs and user IDs as elements. These datasets contain about 293M, 372M, 21M, and 67M items, respectively. Each dataset is divided into ten segments of equal temporal duration. For convenience, we use the abbreviations C16, C19, Cos, and Mul to represent these datasets. In practice, the threshold T for detecting super spreaders is determined by applications, and the detection is harder with a lower threshold. To demonstrate the superiority of our algorithm, we tune the threshold to keep the number of super spreaders to 500. In this case, state-of-the-art solutions can hardly work well with extremely limited memory. We show the CCDF of flow cardinalities in the four datasets' first segments and the entire datasets in Figure 2, where the red points indicate the thresholds for detecting super spreaders.

Implementation: We implement our algorithm and the state-of-the-art algorithms in C++ on a server equipped with two Intel Xeon E5-2643 v4 @3.40GHz CPUs and 256 GB RAM. Each CPU has 20 MB of L3 cache that is shared by all cores. We implement the following baselines in C++ for comparison: SpreadSketch (SS) [35], ExtendedSketch (ES) [20], ExtendedSketch+ (ES+) [15], FlowFight (FF) [2], and FreeRS-SSD (FR) [19]. We employ FreeRS-SSD rather than FreeBS-SSD as the baseline since their performance is very similar, while FreeRS-SSD can record more distinct elements.

Default Parameters: In our experiments, the register size is set to 5 bits, and each memory pointer is calculated as 32 bits. For FR, we allocate 80% of the total memory to build the Stream-Summary to achieve the best performance. For our algorithm, the default memory ratio of the bucket array is 35% for measurements at a single point/period and 10% for measurements across multiple points/periods. The bucket size λ is set to 8. Besides, each virtual estimator has 256 registers. For SS, the size of multiresolution bitmaps is configured based on the relative error of 0.1 and the maximum flow cardinality [35]. For FF, we set the size of the base zone to 500, and the remaining memory is divided equally between the intermediate zone and the promotion zone. For ES and ES+, we only consider the initial memory of their data structure, and the initial size of each bitmap is set to 32 bits. Actually, their final memory consumption is much larger than the initial memory. Besides, the thresholds of their bitmaps' fullness ratio determining the extension condition are set to 0.75 [15, 20]. ES employs bitmaps' extension times to identify abnormal bitmaps and reconstruct flow IDs using the index of these abnormal bitmaps. However, the reconstruction process is time-consuming and can generate many fake flows. Thus, we tune the threshold on extension times to control the number of total generated flow IDs and the number of reported super spreaders to be less than one billion and five thousand, respectively.

5.2 Evaluation Metrics

F1 Score: the harmonic mean of precision (PR) and recall (RC), namely, $F1 = \frac{2\times PR\times RC}{PR+RC}$. Precision is the ratio of true super spreaders detected to all reported super spreaders, which is defined as $\frac{|\Psi\cap\Omega|}{|\Psi|}$, where Ψ is the set of reported super spreaders, and Ω is set of true super spreaders. Recall is the ratio of true super spreaders detected to all true super spreaders, which is defined as $\frac{|\Psi\cap\Omega|}{|\Omega|}$.

Average Relative Error (ARE): the average relative error of the estimated cardinality of reported super spreaders, which is defined as $\frac{1}{|\Psi|}\sum_{f\in\Psi}\frac{|\hat{n}_f-n_f|}{n_f}$. Note that ExtendedSketch may report fake flows. In the calculation, we let the true cardinalities of those fake flows be 1 to allow the division operation.

Throughput: the number of data items processed per second, which is defined as $\frac{N}{S}$, where N is the total number of data items, and S is the seconds used to process all items. We use Millions of updates per second (Mps) to measure the throughput.

5.3 Experiments on Parameter Settings

Let β be the memory ratio of the bucket array and N_{mem} be the number of bits in the total given memory. Suppose the ID and counter fields in each cell are both 32 bits. Then, each bucket occupies 64λ bits. Thus, there are $\omega = \left\lfloor\frac{N_{mem}\beta}{64\lambda}\right\rfloor$ buckets and

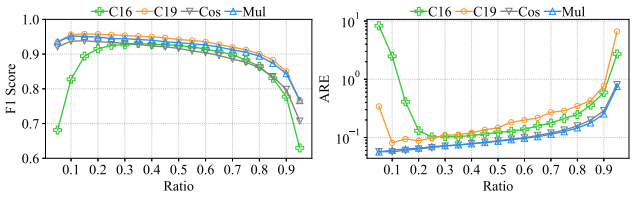

Figure 3: F1 score and ARE for different memory ratios at a single point/period.

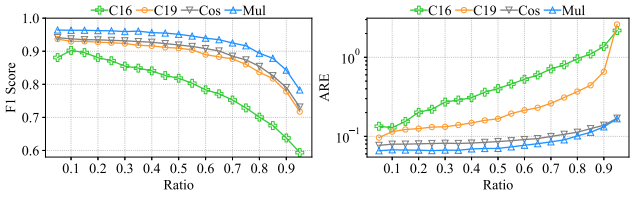

Figure 4: F1 score and ARE for different memory ratios across multiple points/periods.

$M = \left\lfloor\frac{1}{5}\left(N_{mem} - 64\lambda\left\lfloor\frac{N_{mem}\beta}{64\lambda}\right\rfloor\right)\right\rfloor$ registers. In this part, we analyze the settings on memory ratio β and register group size m.

For the measurements at a single point/period, we get the average counter-based query results of the ten segments of each dataset with the total memory varying from 100 KB to 500 KB. For the measurements across multiple points/periods, the total memory is fixed to 500 KB. We treat each dataset segment as the data stream at a single measurement point/period. Then, we utilize the merging operation along with the register-based query to detect super spreaders within the aggregated stream. We vary the number of points/periods from 2 to 10 and repeat each experiment ten times with different hash seeds to obtain the average results.

Firstly, we set $m = 8$ and vary the memory ratio. The F1 scores and AREs are shown in Figure 3 and Figure 4. We can find that when the memory ratio rises, the F1 scores tend to increase first and then decrease. The lowest AREs are achieved near where we get the highest F1 scores. However, the optimal memory ratio differs for different datasets due to their different flow distributions. In dataset C16, the super spreader threshold is much lower than the other datasets, leading to more intense flow competition. Thus, a higher memory ratio can reduce the errors caused by flow competition, especially when the memory is very small (e.g., 100 KB to 300 KB in the single-point/period measurements). When the memory is larger (e.g., 500 KB in the multiple-points/periods measurements) or the threshold is higher, a smaller memory ratio can ensure sufficient buckets, thereby maintaining low flow competition errors. We set the default memory ratio to 0.35 (single-point/period) and 0.1 (multiple-points/periods) to maximize the lowest F1 scores of the four datasets. The performance of our algorithm is stable, and the performance with the selected memory ratio only declines slightly compared with the best performance on each dataset.

Next, we change m to 4, 8, 16, 32, and 64. The memory ratios maximizing the lowest F1 scores are still around 0.35 and 0.1 for each value of m. Thus, we still set the default memory ratio to 0.35 and 0.1. As shown in Figure 5, when m increases, the F1 score of each dataset

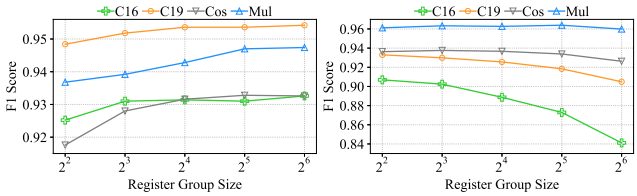

Figure 5: F1 score for different group size m at a single point/period (left) and across multiple points/periods (right).

Table 1: Throughput (Mps) for different group size m.

Dataset \ m	4	8	16	32	64
C16	31.28	30.28	26.28	22.14	16.95
C19	39.39	39.61	34.82	31.20	26.00
Cos	26.73	26.74	22.83	19.74	15.50
Mul	33.37	33.12	29.85	26.75	22.47

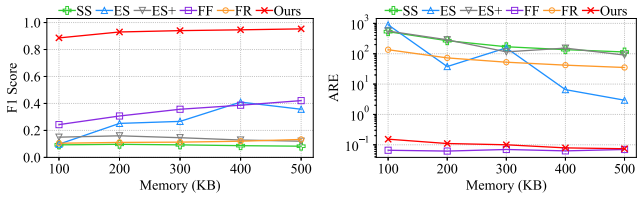

Figure 6: F1 score and ARE on dataset C16 at a single point/period.

slightly increases for measurements at a single point/period. This is because when the group stage-changing probability is calculated based on more registers, the averaging effect can better mitigate the impact of items belonging to small flows but coincidently having very large element rank values. However, a larger group size also leads to fewer groups in a virtual estimator, making the noise distribution less uniform during the memory sharing, thereby decreasing the accuracy of the register-based query. Thus, the F1 score has a more pronounced decrease for measurements across multiple points/periods. Besides, as shown in Table 1, the throughput decreases since there are more memory accesses and operations when calculating larger groups' state-changing probabilities. We can find that the throughput has no significant decrease when m changes from 4 to 8, and the total F1 score is a little higher when $m = 8$. Therefore, we set 8 as the default value of m. Then, each group's memory is limited in a 64-bit machine word and byte-aligned.

5.4 Accuracy at a Single Point/Period

In this section, we evaluate the super spreader detection accuracy when performing measurements at a single point/period. We vary the total memory from 100 KB to 500 KB and calculate the average results across the ten segments of each dataset.

As shown in Figure 6-9, the F1 scores of our algorithm are always the highest in the four datasets. Besides, the gaps between our algorithm and the baselines increase when the memory gets smaller. When the memory is 100 KB, compared with the most accurate

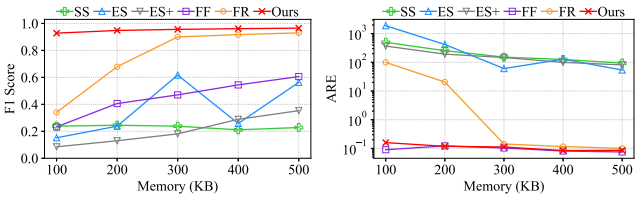

Figure 7: F1 score and ARE on dataset C19 at a single point/period.

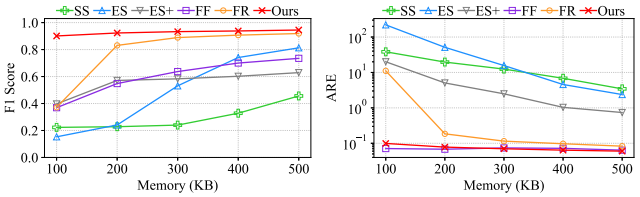

Figure 8: F1 score and ARE on dataset Cos at a single point/period.

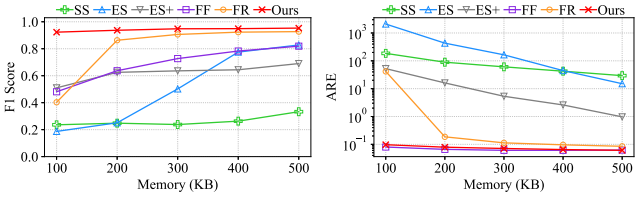

Figure 9: F1 score and ARE on dataset Mul at a single point/period.

baselines, our algorithm improves the F1 score by 0.643, 0.587, 0.504, and 0.413 for datasets C16, C19, Cos, and Mul, respectively. The ARE of our algorithm is a little higher than FF. However, FF has such a low ARE since it has a severe under-estimation problem. When the memory is varied or the dataset changes, the results of ES have large fluctuations because the bitmap extension times have significant variations. ES relies on the extension times to identify abnormal bitmaps and reconstruct IDs using the Chinese Remainder Theory. Thus, the significant variation of bitmap extension times can lead to significant differences in ID reconstruction and finally impact the detection accuracy.

Our algorithm significantly improves the accuracy, especially when memory is limited, mainly for three reasons. First, the proposed register group sharing strategy allows smaller flows to consume less memory, thereby providing high memory efficiency. Besides, since the memory consumption of a flow at most equals the size of a virtual estimator, this strategy can also avoid the excessive memory consumption of the flows with extremely large cardinalities. Second, our algorithm can accurately estimate flows' cardinalities in real time with the help of the group state-changing probability and the highly memory-efficient bucket array structure. Thus, we can accurately capture the super spreaders during measurement. Though FR also employs fine-grained memory sharing and can make accurate real-time estimations, its Stream-Summary

Table 2: Accuracy of RGS-Sketch when measuring the entire datasets with 100 KB of memory at a single point/period.

Dataset	C16	C19	Cos	Mul
F1 score	0.915	0.921	0.884	0.913
ARE	0.205	0.190	0.102	0.098

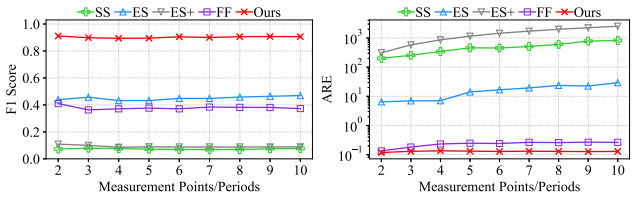

Figure 10: F1 score and ARE on dataset C16 across multiple points/periods.

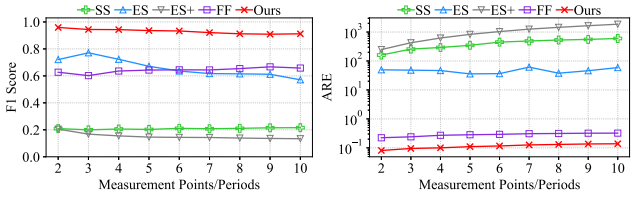

Figure 11: F1 score and ARE on dataset C19 across multiple points/periods.

structure harms its memory efficiency due to the large number of memory pointers. Moreover, the other baselines have to use coarse estimations to capture the super spreaders. Third, the proposed limited register update strategy further enhances memory efficiency by preventing small flows from consuming memory. The experiments in Section 5.7 also prove that this update strategy significantly improves detection accuracy when the memory is extremely limited.

To demonstrate our algorithm's performance in extreme scenarios, we process the whole ten segments with only 100 KB of memory. The results are shown in Table 2. Due to the high memory efficiency, the performance only shows a slight decrease compared to the results on single dataset segments with 100 KB of memory. Moreover, since the difference between super spreaders and small flows increases, the F1 score on dataset C16 even rises.

5.5 Accuracy across Multiple Points/Periods

We utilize the merging operation to merge the sketch instances. Then, we evaluate the accuracy when measuring across multiple points/periods. We set the memory to 500 KB and vary the number of points/periods from 2 to 10. FR is not included in the experiments since it does not support the merging operation.

As shown in Figure 10-13, our algorithm's F1 scores and AREs are always the best compared with the baselines. Specifically, our algorithm achieves an F1 score of at least 0.894, 0.909, 0.911, and 0.947 for datasets C16, C19, Cos, and Mul, respectively. In contrast, the F1 scores of the baselines are not greater than 0.469, 0.771, 0.679, and 0.782, respectively. Compared with the most accurate baselines,

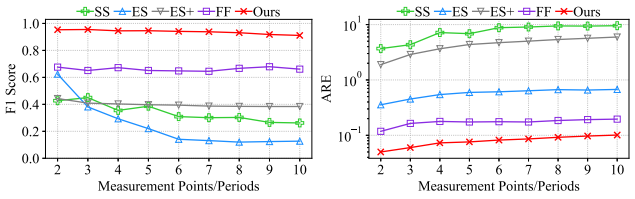

Figure 12: F1 score and ARE on dataset Cos across multiple points/periods.

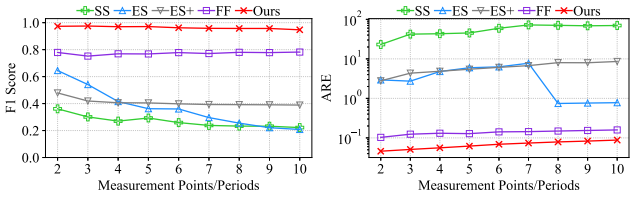

Figure 13: F1 score and ARE on dataset Mul across multiple points/periods.

our algorithm improves the F1 score by up to 0.472, 0.288, 0.304, and 0.223 separately for datasets C16, C19, Cos, and Mul. Note that our algorithm is different from vHLL. Since our algorithm can accurately capture large flows' IDs, it avoids the over-estimation problem in vHLL caused by querying on numerous small flows.

The AREs of SS, ES, and ES+ are quite large. For SS and ES+, this is because they employ estimator-level sharing strategies, leading to large sharing noises that can hardly be removed when memory is insufficient to build enough estimators. Though the initial memory of each estimator in ES+ is only 32 bits, the number of estimators is still small since the memory pointers consume much memory. Moreover, the sharing noises also lead to over-estimation problems in SS and ES+. Like SS and ES+, ES also has the sharing noises that cause large AREs. However, due to its extension strategy, ES also has under-estimation problems after performing the merging operations. We can find that when the number of points/periods increases, ES's F1 scores have a more apparent decrease for datasets Cos and Mul. This is because datasets C16 and C19 contain more duplicate elements, mitigating the loss during merging.

5.6 Item Processing Throughput

In this part, we compare our algorithm with the baselines from the perspective of throughput. In the experiments, we set the memory ratio of the bucket array in our algorithm to 0.35 and vary the total memory from 100 KB to 500 KB.

We measure the recording throughput first and do not perform queries actively. As shown in the left part of Figure 14, each dataset's columns in the bar chart represent the average throughputs with different memory, and the error bar presents the minimum and maximum throughputs. We can find that our algorithm is second only to FR. FR is faster mainly because it employs the traditional strategy. Our algorithm's throughput can be comparable to FR if it also uses the traditional strategy, as shown in the next subsection.

Then, we estimate the cardinality of the item's flow after processing each arrival item and measure the online query throughput.

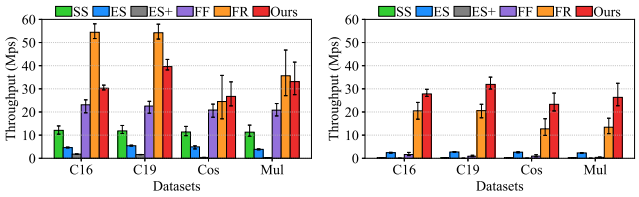

Figure 14: Recording throughput (left) and online query throughput (right) of all compared methods.

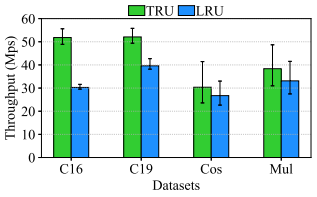

Figure 15: Recording throughput of register update strategies.

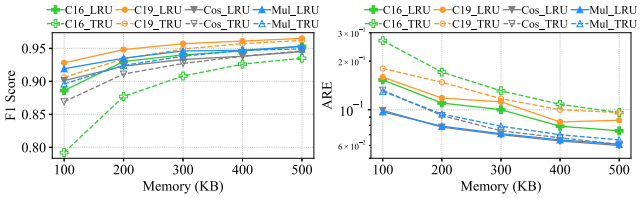

Figure 16: F1 score and ARE of different register update strategies at a single point/period.

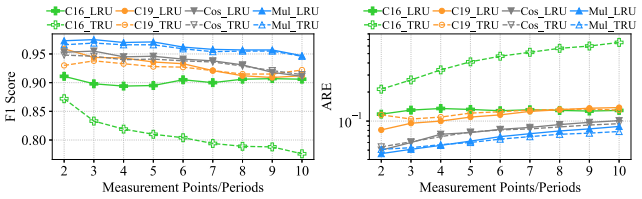

Figure 17: F1 score and ARE of different register update strategies across multiple points/periods.

Note that SS, ES, ES+, and FF are not designed for online detection. As shown in Figure 14, our algorithm's online query throughput is the highest, nearly eleven times as high as that of the fastest non-online algorithm. FR also has a high throughput. This is because our algorithm and FR only need to access the bucket array or the Stream-Summary to get the estimation. Since the Stream-Summary needs multiple discrete memory accesses, FR is slower than our algorithm. SS and ES+ have extremely low throughputs since they must combine the mapped bitmaps via the bitwise AND operation to minimize the sharing noise, which is time-consuming. Moreover, ES+ needs to unify the mapped bitmaps. In contrast, ES performs the bitwise AND operation only when all the mapped bitmaps have not expanded. Once a mapped bitmap has been expanded, ES only accesses the counters recording the number of ones in the bitmaps. Since FF must traverse the whole mapped HyperLogLog to get the estimation, it also has a lower throughput than ES.

5.7 Impacts of Limited Register Update Strategy

In this part, we evaluate the impacts of our limited register update strategy on the recording throughput and accuracy. For comparison, we construct a variant of our algorithm that replaces our limited register update strategy with the traditional one. Then, we denote our algorithm and the variant as LRU and TRU, respectively. Besides, the prefixes C16, C19, Cos, and Mul in the legends represent the datasets. Since the register update strategies have little impact on the query overhead, we only show the recording throughput.

As shown in Figure 15, the traditional register update strategy exhibits superior throughputs. This superiority stems from its lower frequency of accessing the bucket array. Once an item arrives with a higher element rank value than its mapped register, the register must be updated. It prevents the subsequent items mapped to this register with the same or smaller rank values (yet exceeding the register's original value) from accessing the bucket array. For our limited register strategy, the register may not be updated even if an item's rank value exceeds its mapped register. In this case, the subsequent items with the same or smaller rank values can still access the bucket array and try the replacement, incurring lower throughput.

However, as shown in Figure 16 and Figure 17, the limited register update strategy has a superiority in detection accuracy. For dataset C16, our strategy improves the F1 score by up to 0.094 for a single point/period, and by up to 0.130 across multiple points/periods. Besides, our strategy's ARE can also be up to 1.75 and 5.06 times smaller than the traditional strategy, respectively. The gains for the other datasets are less pronounced because their super spreader thresholds are much higher, which means small flows have less impact on the estimation accuracy of super spreaders. When the memory or the number of points/periods is large, our strategy's performance is worse than the traditional strategy, but the gap is tiny. Overall, our limited register update strategy outperforms the traditional register update strategy in accuracy.

6 CONCLUSION

This paper introduces a novel algorithm named RGS-Sketch, tailored for detecting super spreaders in high-speed data streams. The core of RGS-Sketch is founded on a new memory sharing approach called register group sharing. This method shares memory at a level of register groups, providing high memory efficiency, facilitating online detection, and offering mergeability and invertibility. Additionally, we propose a limited register update strategy to enhance our algorithm's detection accuracy by minimizing the memory consumption of small flows. Extensive experiments using real-world datasets reveal that RGS-Sketch consistently achieves the highest F1 scores compared with the state-of-the-art algorithms while maintaining a high throughput.

ACKNOWLEDGMENTS

This research was supported in part by the National Natural Science Foundation of China (NSFC) under Grants 62332013, and 62472298, and in part by the Project Funded by the Priority Academic Program Development of Jiangsu Higher Education Institutions.

REFERENCES

[1] Jesse Anton, Lawrence Jacobs, Xiang Liu, Jordan Parker, Zheng Zeng, and Tie Zhong. 2002. Web caching for database applications with Oracle Web Cache. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 594–599.

[2] Valerio Bruschi, Salvatore Pontarelli, Jerome Tollet, Dave Barach, and Giuseppe Bianchi. 2021. FlowFight: High performance–low memory top-k spreader detection. *Computer Networks* 196 (2021), 108239.

[3] CAIDA. 2016. The CAIDA UCSD Anonymized Internet Traces 2016. https://catalog.caida.org/dataset/passive_2016_pcap. Accessed: 2019-7-28.

[4] CAIDA. 2019. The CAIDA UCSD Anonymized Internet Traces 2019. https://catalog.caida.org/dataset/passive_2019_pcap. Accessed: 2021-12-27.

[5] Jing Cao, Yu Jin, Aiyou Chen, Tian Bu, and Z-L Zhang. 2009. Identifying high cardinality internet hosts. In *Proceedings of the IEEE Conference on Computer Communications*. 810–818.

[6] Graham Cormode and Shan Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms* 55, 1 (2005), 58–75.

[7] Damu Ding, Marco Savi, Federico Pederzolli, Mauro Campanella, and Domenico Siracusa. 2021. In-network volumetric DDoS victim identification using programmable commodity switches. *IEEE Transactions on Network and Service Management* 18, 2 (2021), 1191–1202.

[8] Yang Du, He Huang, Yu-E Sun, Shigang Chen, and Guoju Gao. 2021. Self-adaptive sampling for network traffic measurement. In *Proceedings of the IEEE Conference on Computer Communications*. 1–10.

[9] Yang Du, He Huang, Yu-E Sun, Shigang Chen, Guoju Gao, Xiaoyu Wang, and Shenghui Xu. 2022. Short-term memory sampling for spread measurement in high-speed networks. In *Proceedings of the IEEE Conference on Computer Communications*. 470–479.

[10] Zakir Durumeric, Michael Bailey, and J Alex Halderman. 2014. An Internet-Wide view of Internet-Wide scanning. In *Proceedings of the USENIX Security Symposium*. 65–78.

[11] Cristian Estan, George Varghese, and Mike Fisk. 2003. Bitmap algorithms for counting active flows on high speed links. In *Proceedings of the ACM SIGCOMM conference on Internet measurement*. 153–166.

[12] Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. 2014. Cuckoo filter: Practically better than bloom. In *Proceedings of the ACM International on Conference on emerging Networking Experiments and Technologies*. 75–88.

[13] Philippe Flajolet, Eric Fusy, Olivier Gandouet, and Frederic Meunier. 2007. HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. *DMTCS Proceedings* (2007), 127–146.

[14] Philippe Flajolet and G Nigel Martin. 1985. Probabilistic counting algorithms for data base applications. *Journal of computer and system sciences* 31, 2 (1985), 182–209.

[15] Hui Han, Xuyang Jing, Zheng Yan, and Witold Pedrycz. 2023. ExtendedSketch+: Super host identification and network host trust evaluation with memory efficiency and high accuracy. *Information Fusion* 92 (2023), 300–312.

[16] Stefan Heule, Marc Nunkesser, and Alexander Hall. 2013. Hyperloglog in practice: Algorithmic engineering of a state of the art cardinality estimation algorithm. In *Proceedings of the International Conference on Extending Database Technology*. 683–692.

[17] He Huang, Yu-E Sun, Chaoyi Ma, Shigang Chen, Yang Du, Haibo Wang, and Qingjun Xiao. 2021. Spread estimation with non-duplicate sampling in high-speed networks. *IEEE/ACM Transactions on Networking* 29, 5 (2021), 2073–2086.

[18] He Huang, Jiakun Yu, Yang Du, Jia Liu, Haipeng Dai, and Yu-E Sun. 2023. Memory-Efficient and Flexible Detection of Heavy Hitters in High-Speed Networks. *Proceedings of the ACM on Management of Data* 1, 3 (2023), 1–24.

[19] Peng Jia, Pinghui Wang, Yuchao Zhang, Xiangliang Zhang, Jing Tao, Jianwei Ding, Xiaohong Guan, and Don Towsley. 2020. Accurately estimating user cardinalities and detecting super spreaders over time. *IEEE Transactions on Knowledge and Data Engineering* 34, 1 (2020), 92–106.

[20] Xuyang Jing, Zheng Yan, Hui Han, and Witold Pedrycz. 2021. Extendedsketch: Fusing network traffic for super host identification with a memory efficient sketch. *IEEE Transactions on Dependable and Secure Computing* 19, 6 (2021), 3913–3924.

[21] Xuyang Jing, Zheng Yan, and Witold Pedrycz. 2018. Security data collection and data analytics in the internet: A survey. *IEEE Communications Surveys & Tutorials* 21, 1 (2018), 586–618.

[22] Noriaki Kamiyama, Tatsuya Mori, and Ryoichi Kawahara. 2007. Simple and adaptive identification of superspreaders by flow sampling. In *Proceedings of the IEEE Conference on Computer Communications*. 2481–2485.

[23] MICHAEL KECHINOV. 2020. eCommerce behavior data from multi category store. https://www.kaggle.com/datasets/mkechinov/ecommerce-behavior-data-from-multi-category-store. Accessed: 2024-04-10.

[24] MICHAEL KECHINOV. 2020. eCommerce Events History in Cosmetics Shop. https://www.kaggle.com/datasets/mkechinov/ecommerce-events-history-in-cosmetics-shop. Accessed: 2024-04-10.

[25] Adam Kirsch and Michael Mitzenmacher. 2008. Less hashing, same performance: Building a better Bloom filter. *Random Structures & Algorithms* 33, 2 (2008), 187–218.

[26] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. 2016. FlowRadar: A better NetFlow for data centers. In *Proceedings of the USENIX symposium on networked systems design and implementation*. 311–324.

[27] Weijiang Liu, Wenyu Qu, Jian Gong, and Keqiu Li. 2015. Detection of superpoints using a vector bloom filter. *IEEE Transactions on Information Forensics and Security* 11, 3 (2015), 514–527.

[28] Chaoyi Ma, Shigang Chen, Youlin Zhang, Qingjun Xiao, and Olufemi O Odegbile. 2021. Super spreader identification using geometric-min filter. *IEEE/ACM Transactions on Networking* 30, 1 (2021), 299–312.

[29] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. 2005. Efficient computation of frequent and top-k elements in data streams. In *Proceedings of the International conference on database theory*. 398–412.

[30] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo hashing. *Journal of Algorithms* 51, 2 (2004), 122–144.

[31] Subhabrata Sen and Jia Wang. 2002. Analyzing peer-to-peer traffic across large networks. In *Proceedings of the ACM SIGCOMM Workshop on Internet measurment*. 137–150.

[32] Qilong Shi, Yuchen Xu, Jiuhua Qi, Wenjun Li, Tong Yang, Yang Xu, and Yi Wang. 2023. Cuckoo counter: Adaptive structure of counters for accurate frequency and top-k estimation. *IEEE/ACM Transactions on Networking* 31, 4 (2023), 1854–1869.

[33] Xingang Shi, Dah-Ming Chiu, and John CS Lui. 2010. An online framework for catching top spreaders and scanners. *Computer Networks* 54, 9 (2010), 1375–1388.

[34] Sumeet Singh, Cristian Estan, George Varghese, and Stefan Savage. 2004. Automated Worm Fingerprinting. In *Proceedings of the Symposium on Operating System Design and Implementation*. 45–60.

[35] Lu Tang, Yao Xiao, Qun Huang, and Patrick PC Lee. 2022. A high-performance invertible sketch for network-wide superspreader detection. *IEEE/ACM Transactions on Networking* 31, 2 (2022), 724–737.

[36] Shobha Venkataraman, Dawn Xiaodong Song, Phillip B Gibbons, and Avrim Blum. 2005. New streaming algorithms for fast detection of superspreaders. In *Proceedings of the Network and Distributed System Security Symposium*. 1–18.

[37] Haibo Wang, Chaoyi Ma, Olufemi O Odegbile, Shigang Chen, and Jih-Kwon Peir. 2021. Randomized error removal for online spread estimation in data streaming. *Proceedings of the VLDB Endowment* 14, 6 (2021), 1040–1052.

[38] Pinghui Wang, Xiaohong Guan, Tao Qin, and Qiuzhen Huang. 2011. A data streaming method for monitoring host connection degrees of high-speed links. *IEEE Transactions on Information Forensics and Security* 6, 3 (2011), 1086–1098.

[39] Kyu-Young Whang, Brad T Vander-Zanden, and Howard M Taylor. 1990. A linear-time probabilistic counting algorithm for database applications. *ACM Transactions on Database Systems (TODS)* 15, 2 (1990), 208–229.

[40] Qingjun Xiao, Shigang Chen, Min Chen, and Yibei Ling. 2015. Hyper-compact virtual estimators for big network data based on register sharing. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. 417–428.

[41] Tong Yang, Junzhi Gong, Haowei Zhang, Lei Zou, Lei Shi, and Xiaoming Li. 2018. Heavyguardian: Separate and guard hot items in data streams. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2584–2593.

[42] Tong Yang, Haowei Zhang, Jinyang Li, Junzhi Gong, Steve Uhlig, Shigang Chen, and Xiaoming Li. 2019. HeavyKeeper: an accurate algorithm for finding Top-k elephant flows. *IEEE/ACM Transactions on Networking* 27, 5 (2019), 1845–1858.

[43] MyungKeun Yoon and Shigang Chen. 2011. Detecting stealthy spreaders by random aging streaming filters. *IEICE transactions on communications* 94, 8 (2011), 2274–2281.

[44] Bohan Zhao, Xiang Li, Boyu Tian, Zhiyu Mei, and Wenfei Wu. 2021. Dhs: Adaptive memory layout organization of sketch slots for fast and accurate data stream processing. In *Proceedings of the ACM SIGKDD Conference on Knowledge Discovery & Data Mining*. 2285–2293.