Enhancing Accuracy for Super Spreader Identification in High-Speed Data Streams

Haibo Wang
Department of Computer Science, University of Kentucky, Lexington, KY USA, haibo@ieee.org

ABSTRACT

This paper addresses the challenge of identifying super spreaders within large, high-speed data streams. In these streams, data is segmented into flows, with each flow's spread defined as the number of distinct items it contains. A super spreader is characterized as a flow with a notably large spread. Current compact solutions, known as sketches, are designed to fit within the constrained memory of online devices. However, they struggle to accurately track the spread of all flows due to the substantial memory requirement for monitoring a single flow — a problem exacerbated when numerous flows are involved. To overcome these limitations, this study proposes a more precise sketch-based approach. Our solution introduces an innovative non-duplicate sampler that effectively eliminates duplicates, allowing for accurate post-sampling count of flow spread using only counters. Additionally, it incorporates an exponential-weakening decay technique to highlight large flows, markedly enhancing the accuracy of super spreader identification. We offer a comprehensive theoretical analysis of our method. Trace-driven experiments validate that our approach statistically surpasses existing state-of-the-art solutions in identifying super spreaders. It also demonstrates the lowest time required to restore super spreaders and significantly reduces bandwidth consumption by an order of magnitude when offline restoration is conducted remotely.

PVLDB Reference Format:
Haibo Wang. Enhancing Accuracy for Super Spreader Identification in High-Speed Data Streams. PVLDB, 17(11): 3124 - 3137, 2024.
doi:10.14778/3681954.3681988

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/wanghaibo321/super-spreader-identificationi.

1 INTRODUCTION

The real-time analysis of large, high-rate data streams has a multitude of applications, as highlighted in various studies [19, 23, 40, 48, 59, 63, 70, 79, 83, 84]. Conventionally, a data stream is represented as a sequence of data items, denoted as $d_1, d_2, d_3, d_2, \dots$. The key statistical metrics of interest traditionally include item frequencies within the stream and the stream's cardinality, which is the count of distinct items [31, 38, 41, 42, 42, 56, 65, 66, 69, 71, 78, 78, 84].

However, the traditional model is unsuitable for many sophisticated applications. To capture greater details of a data stream, we adopt a more general model where a data stream consists of data items from multiple sub-streams also called *flows* and each data item is a pair $\langle f, e \rangle$, where f is a flow label that tells which flow the item belongs to, and e is the actual data (also referred to as *element*) of interest in the flow. There are two fundamental flow statistics: *flow size*, defined the number of elements in a flow, and *flow spread* which this paper focuses on, defined as the number of distinct elements in a flow. Flow size is the traditional item frequency if we treat the item label as the flow label. However flow spread is the traditional cardinality only in the special case where we treat the whole stream as a single flow. Of particular importance are large flows whose size or spread exceeds a user-set threshold, called *heavy hitters* and *super spreaders*, respectively.

Identifying super spreaders in this generalized model has significant implications in various domains, such as P2P hot-spot localization [57], web caching prioritization [3, 85], detection of DDoS attacks [2, 51], port scanning measurement [19], and worm propagation detection [11, 58]. To illustrate this in the context of Internet applications, consider a packet stream arriving at a high-speed router, where each packet is modeled as $\langle f, e \rangle$. The definitions of f and e can be arbitrarily based on packet header or payload information to meet specific application requirements. For instance, identifying a super spreader in a network could involve detecting external sources that exhibit large spreads, i.e., flows with a high number of distinct destination addresses. Conversely, by defining destination addresses as flow labels and source addresses as elements, super spreader identification can assist in pinpointing potential victims of DDoS attacks—those internal destination addresses receiving traffic from numerous distinct sources. Another example involves a large server farm analyzing content popularity by tracking distinct user accesses to each file [8], where each file accessed forms a distinct flow. Super spreader identification is also applied in various data analysis systems at Google, such as Sawzall [53], Dremel [49], and PowerDrill [29], to estimate unique user searches for particular keys.

Addressing the challenge of super spreader identification in large, high-rate data streams involves meeting two practical requirements. Firstly, the solution must be sufficiently rapid to match the item arrival rate, ensuring minimal processing overhead per item. For example, in packet streams, modern routers operate at speeds ranging from hundreds of gigabits to terabits per second, equating to processing millions of packets per second [43]. Secondly, memory efficiency is crucial to facilitate software/hardware implementation within the limited cache memory of streaming devices, such as the approximately 10 MB SRAM found on routers and switches. The allocation for measurement functions is further restricted due to other critical routing, performance, and security functions, often limiting safe memory allocation to a small fraction, like 10% or 5%. This severe memory limitation makes it impractical to assign

Proceedings of the VLDB Endowment, Vol. 17, No. 11 ISSN 2150-8097.
doi:10.14778/3681954.3681988

individual spread estimators to each flow in a data stream, which could comprise millions of per-source flows within one hour [17].

In recent decades, the research focus on super spreader identification has evolved from per-flow tracking and sampling, which incur significant computing or memory costs, to designing compact data structures known as *sketches*. Notable examples include DCS [26], FAST [40], CMH [15], OpenSketch, SpreadSketch [60], AROMA [5], and GMF [44]. Among these, SpreadSketch, AROMA, and GMF are recent innovations considered state-of-the-art. However, our experimental results indicate that these sketches still struggle with accuracy. SpreadSketch, for instance, tends to overestimate flow spread and falsely identify super spreaders, while AROMA and GMF exhibit large estimation variances. This is primarily due to the challenge of duplicate removal when measuring flow spread—only the first appearance of an item should increment the spread, with subsequent duplicates ignored. To counteract the impact of duplicates, more sophisticated data structures are required beyond simple counters used for single-flow size measurement. As memory constraints intensify, hash collisions among flows increase, leading to heightened estimation errors. While AROMA adopts a self-adaptive sampling approach, its sampling probability remains small under memory constraints for the same reasons.

This paper approaches the problem from a novel angle. We retain the sophisticated data structures utilized in prior sketches for duplicate removal but repurpose them as a *non-duplicate sampler* rather than a spread estimator. This sampler filters out duplicate items, outputting them only at their first appearance with an evolving probability p. This approach allows us to maintain a simple counter for measuring a flow's spread, incrementing the counter by $\frac{1}{p}$ each time a flow's item is sampled. Furthermore, we incorporate an *exponential-weakening decay* technique to effectively manage hash collisions among flows mapped to the same non-duplicate sampler, aiding in the accurate measurement and differentiation of large flows. We provide a theoretical analysis of our proposed solution's performance. Trace-driven experiments demonstrate that our approach significantly outperforms existing state-of-the-art solutions in terms of super spreader identification accuracy.

2 BACKGROUND

2.1 System Model

A compact data streaming processing system comprises two main components: an online recording module and an offline processing module. The online module is tasked with monitoring the data stream on various online processors, which could include web servers, cache systems, routers, switches, gateways, intrusion detection systems (IDS), online search engines, among others. To manage the high-speed arrival rate of data items, this module is ideally implemented on limited on-chip cache memory, allowing for rapid processing. It is essential for the online module to be lightweight, both in terms of memory footprint and processing overhead, to efficiently handle the continuous data flow.

The offline module's deployment varies depending on the user and the system's constraints. If the system itself serves as the user, like in the case of a local server, the offline module is typically integrated within the online processors. Here, the module operates under more stringent time and resource constraints, necessitating a focus on efficiency and compactness. Conversely, when real-time constraints are not a pressing concern, the offline module can be deployed on a more powerful server, allowing for broader and more resource-intensive processing capabilities.

Operationally, the system divides time into measurement periods, the length of which is determined by the application's requirements. During each period, the online module extracts relevant data from incoming items and updates its data structures. At the period's end, the collected data is offloaded to the offline processing module, which could be located on a server. This process involves resetting the online module's data structures to start anew for the next period. For data offloaded to a server, compactness is key to minimize bandwidth usage. Finally, the server processes this data to identify offline super spreaders.

2.2 Problem Statement

Traditionally, a data stream is a sequence of data items, denoted as $\ldots d \ldots$. Classical measurements in this model include the frequencies of data items [31, 38, 41, 42, 56, 78, 84] and the stream's *spread* (or cardinality) [42, 78], defined as the count of distinct items in the stream. For instance, in a data stream of d_1, d_2, d_1, d_1, d_1 has a frequency of 3, d_2 has a frequency of 1, and the stream's spread is 2, representing the two distinct items, d_1 and d_2.

This paper utilizes a generalized model where a data stream comprises a continuous sequence of data items $\langle f, e\rangle$, with f representing a flow label and e serving as an element identifier. Data items sharing the same label constitute a flow. In essence, the stream is segmented into multiple sub-streams or flows, each uniquely labeled and containing a set of elements. It's worth noting that this set is a multi-set, as elements can appear multiple times, which will be further clarified through an example. Flow statistics can vary and include: (1) flow size, the number of data items in a flow; (2) the count of different flows; and (3) *flow spread*, the number of distinct elements in a flow. The first two statistics align with those in the traditional model when we substitute f for d and disregard e. However, this paper specifically focuses on flow spread.

In most applications, large flows are of particular interest. Flows with substantial size are referred to as *heavy hitters*, while those with a large spread are known as *super spreaders*. A flow is deemed large if its size or spread reaches a pre-set threshold T. The challenge addressed in this paper is to *design an efficient sketch*, a compact data structure that records the data items of a given stream. After recording, this sketch should be able to output all flow labels associated with super spreaders.

To illustrate this model, consider the gateway of an enterprise network configured to monitor the inbound packet stream for scan detection. Each packet is abstracted as a data item $\langle f, e\rangle$, with the flow label f being the source address from the packet header, and element e representing the destination address/port also found in the packet header.

• Consider a packet flow $\{\langle f_1, e_1\rangle, \langle f_2, e_2\rangle, \langle f_1, e_1\rangle, \langle f_1, e_1\rangle\}$. For the first task, we count that there are three packets in flow f_1 and one packet in flow f_2. Note that the same destination address/port e_1 appears in flow f_1 three times. For the second task, we count that there are two flows. For the third task, we see that the spread of flow f_1 is one and that of flow f_2 is also one.

• Consider a packet flow $\{\langle f_1, e_1\rangle, \langle f_2, e_2\rangle, \langle f_1, e_3\rangle, \langle f_1, e_4\rangle\}$. The answers are all the same as in the previous case except for the spread of f_1 is now 3.

Consider a scenario where an external source (f) sends 1,000,000 packets through the gateway. If these packets target the same destination/port (e), the source's spread is 1. However, if the packets are directed to different destination/port pairs, the spread is 1,000,000, indicating a potential scan of the network. By monitoring the spreads of all sources, the gateway can effectively detect scanners. This application aligns well with our general model but is not compatible with the traditional model. Additional application examples are detailed in the introduction.

Identifying super spreaders is crucial for a variety of uses. On social media platforms, the goal might be to identify trending topics that a significant number of users have discussed [12, 74]. In the context of e-commerce, a product gains the label of a super spreader if a substantial number of unique users review it [36]. For online advertising platforms, monitoring the unique click counts for each advertisement is key to assessing its success [33]. Specifically, ads that attract clicks from a wide audience, termed super spreaders, can prompt further action, such as the introduction of more related ads, to boost profits. We list some specific definitions of super spreaders in various applications in Table 1.

2.3 Heavy Hitters and Super Spreaders

The problem of *heavy-hitter identification* bears close relevance to our study, which focuses on identifying flows with large sizes. The predominant strategy in heavy-hitter identification involves maintaining a small subset of flows, aiming to capture large flows within this set while replacing smaller ones with new candidates. Notable solutions in this domain include Frequent [16], Lossy Counting [47], Space Saving [50], CSS [6], RHHH [7], Heavy Keeper [28], SketchLearner [32], Elastic Sketch [78], Topkapi [46], and Nitrosketch [41]. The data structures employed to manage this subset of flows range from min-heaps [13] and stream summaries [50] to TinyTables [20] and hash tables [78]. Additionally, sampling techniques have proven effective in filtering out smaller flows [41].

To illustrate, let's consider two exemplary solutions. Space Saving [50] maintains a number of flows along with their sizes in a stream summary, offering $O(1)$ time efficiency for updating any flow's size or identifying the smallest flow. Here, the counter for a flow present in the summary is incremented by one upon the arrival of its packet. Conversely, when a packet from a new flow arrives, the smallest flow f in the summary is replaced with this new flow, whose size is set to the size of f plus one. Heavy Keeper [28], on the other hand, utilizes hash tables where each entry stores a flow's ID and a corresponding counter. When a packet from a listed flow arrives, its counter is incremented. If a packet from a new flow arrives and is hashed to an occupied entry, the existing flow's counter undergoes decay. Once this counter decays to zero, the flow is replaced by the new one.

Another example solution to heavy hitter identification, Topkapi [46] has a data structure of a two-dimension array, with each cell consisting of a flow label field and a counting field. It updates the data structure in item recording as follows: Decrease the counting field by 1 deterministically when the flow label differs from existing one stored in the flow label field and increases it by 1 otherwise. Our design differs from Topkapi in 1) solving a different problem; 2) containing a unique HLL sampler in each sampler; and 3) updating the data structure probabilistically rather deterministically.

However, it's important to note that these solutions, designed for heavy hitter identification, are not suitable for super spreader identification, as they rely on counters. Counters, by their nature, are incapable of tracking a flow's spread, which is defined as the count of distinct elements in the flow.

2.4 Prior Art

Before exploring mainstream sketch-based solutions, let's first consider other existing approaches.

Per-flow tracking: In specific intrusion detection systems, such as Snort [55] and FlowScan [54], the system maintains all active connections for each source (defining the source address as the flow label) to identify port scanning activities. This exacting counting approach, though precise, leads to substantial memory consumption. To improve memory efficiency, Estan et al. [21] propose using a bitmap for sources with many connections. However, this method still suffers from high memory consumption as the number of flows can be numerous.

Sampling: To limit the number of distinct items processed and enhance memory efficiency, hash-based sampling methods [9, 35, 64] are proposed to monitor only a fraction of flows. The likelihood of sampling super spreaders is high if the sampling probability is appropriately set. However, as widely acknowledged, sampling-based solutions are less accurate. They require processing a sufficiently large number of items to converge. Below we review the existing sketches for super spreader identification.

DCS [26] uses multiple hash tables, each with a different sampling probability, to store $\langle f, e\rangle$ pairs, from which we can count the number of distinct sampled elements from each flow in each table, produce an estimate adjusted by sampling probability, select the most accurate estimate from different hash tables, and identify super spreaders. By storing the element identifiers (or encoding them in Bloom filters), the memory overhead is very large.

FAST [40] maintains multiple arrays of HLL sketches [24]. For each arrival packet $\langle f, e\rangle$, it splits f into two parts, hashes one part to d HLL arrays, and in each array records e in one HLL sketch for every bit in the second part of f whose value is one. Without storing element identifiers, this method uses less memory than [24, 64]. However, it has to record element e many times in each of d arrays. This still causes significant memory overhead and inaccuracy as each HLL sketch has to be shared by many flows. Moreover, it is very computationally expensive to recover the flow identifiers.

CMH [15] uses CountMin and a min-heap whose counters are replaced by a data structure such as bitmap that can measure flow spread. As we explained earlier, this method has significant memory overhead. For each arrival packet $\langle f, e\rangle$, while recording the element, it queries the spread of flow f. If the estimated spread of f is larger than a threshold, it will report f as a super spreader. However, spread estimation is computationally expensive and not suitable for online per-packet operation. In a similar design, as a general framework, OpenSketch [82] replaces the counter in the CountMin sketch [15] with a bitmap for super spreader identification.

Table 1: Examples of super spreaders in broad database scenarios.

Tasks	f	e	Data type
Hot topics	Topic tag	users	social tweets [12, 74]
Popular products	product ID	users	produce reviews [36]
DDoS attacks	dstIP	srcIP	network traffic [2, 51]
Port scanners	srcIP-dstIP	dstPort	network traffic [19]
Successful ads	ads ID	users	ads clicks [33]
Popular events	serach keys	users	online search stream [53] [49] [29]
High-priority content	file name	users	content access stream [8]

Table 2: Comparison of the proposed sketch with existing solutions. Solutions in bold are considered state of the art.

Group of Solutions	Solutions	Accuracy	Memory Efficiency	Remarks
Sketches	DCS [26]	Low	Low	Capable of storing labels of super spreaders
	FAST [40]	Low	Low	
	CMH [15]	Low	High	
	OpenSketch [82]	Low	High	
	SpreadSketch [60]	Medium	High	
	AROMA [5]	Medium	High	
	GMF [44]	Medium	High	
Per-flow Tracking	Snort [55]	High	Low	Challenged by the sheer number of flows, impacting memory efficiency
	FlowScan [54]	High	Low	
	Estan et al. [21]	High	Low	
Sampling-based Solutions	Paper [9]	Low	High	Effective for a small set of flows. Requires very low sampling probability for memory efficiency
	Paper [35]	Low	High	
	Paper [64]	Low	High	
Other Sketches	Degree Sketch [72]	Low	High	Cannot store labels of super spreaders, hence unable to independently report them
	Vector Bloom Filter [39]	Low	High	
	Extended Sketch [34]	Medium	High	
This paper	Our Solution	High	High	Capable of storing and reporting super spreaders

SpreadSketch [60] can measure the spreads of many flows simultaneously and identify the super spreaders among them. Its data structure follows Count-Min [14] but replaces each counter with a multi-resolution bitmap (MRB) [22], a register and a label field. MRB is designed to measure the spread of one flow. Each flow is mapped to $d > 1$ MRBs and will occupy an MRB by setting the corresponding label field if it has the largest geometric hash value $G(\cdot)$ among all flows mapped to the MRB, where $G(\cdot) = i$ with probability of $\frac{1}{2^i}$, $i \geq 1$. However, SpreadSketch has large estimation error, which will result in low identification accuracy. Specifically, there are severe hash collisions among flows, which "deposit" noise to the MRB with respect to each other. Accordingly, the error is large and the spread estimates usually deviate from real spread positively a lot. For super spreader identification, SpreadSketch will report a large number of false positives, making the performance metric, i.e., F1 score in a low level.

AROMA [5] adopts a self-adaptive sampling strategy [1]. It allocates a bucket array where each bucket stores a flow label and a counter. It first hashes each packet $\langle f, e\rangle$ into a bucket in the array and then produces another hash value of $\langle f, e\rangle$. If the hash value is smaller than the counter of the bucket, the counter is set to this value and the flow label is set to f. Because a flow of higher spread has more distinct elements, they together are more likely to produce small hash values, meaning that more of them will be more likely to stay in the array. Hence, we can identify super spreaders by finding the flow labels that appear most in the array, and estimate their spreads based on their counts in the array. Storing the same flow labels many times in the array can cost significant memory overhead, especially when the flow labels are long (such as 104 bits for each TCP flow label). In addition to TCP flow labels with 104 bits, the long flow labels are common in broad database domains. For instance, mining hot topics in an application of super spreader identification (refer to the last paragraph of Section 2.2), a hot topic can be tens or hundreds of letters, each with 8 bits using ASCII Code, resulting hundreds of thousands of bits for a flow key. Moreover, when there exists a flow of very large spread, it could push other super spreaders of smaller spreads out of the array, causing either failure in identifying some super spreaders or inaccuracy in their spread estimations. Both cases will be evaluated in our experiments.

Geometric-min Filter (GMF) [44] performs flow filtering to only store the candidate flow labels in the hash table whose geometric hash value is larger than a threshold, and then performs further flow

[1]AROMA can also be considered as a sampling based solution but we categorize it into sketches.

spread estimation in existing sketches, i.e., vSketch(HLL) [85]. Its accuracy is limited by vSketch(HLL) for per-flow spread estimation.

Other sketches: Degree Sketch [72] and Vector Bloom Filter [39] assumes the knowledge of the entire flow key space, based on which they can recover the candidate super spreaders offline from the sketches that storing the encoded information during item recording. They have two drawbacks. The first is the huge computational overhead for recovering and the second is flow key space is practically hard to know in advance or nearly infinite. Extended Sketch [34] uses a two-dimensional data structure, where flows are hashed to columns and elements are hashed to rows. They can support reporting hot columns but cannot support reporting super spreaders' flow labels after item recording as they do not store the flow labels in the data structure.

There are a thread of sketches dealing with per-flow spread estimation, with the objective of supporting spread queries given any flow label f. However, these sketches do not store flow labels and consequently these solutions cannot report the super spreaders based on their own sketches. This thread includes CSE[80, 81], vHLL [76, 77], vSketch/bSketch/cSketch [85], FreeBS, FreeRS [73], Randomized Sketches [67, 68], kJoin/skJoin sketches [45]

Summary of existing solutions: In Table 2, we outline the previous solutions to identifying super spreaders. Experimental evidence from [60] shows that approaches such as DCS, CMH, FAST, Degree Sketch, and Vector Bloom Filter fall short in accuracy when compared to SpreadSketch. Degree Sketch, and Vector Bloom Filter are excluded for an additional reason that they do not store the flow labels of super spreaders and hence cannot report those super spreaders independently. Other solutions, such as those in the "per-flow tracking" category of Table 1, are excluded for evaluation because they are not memory efficiency, where memory overhead comes from the large number of flows. Therefore, we regard SpreadSketch, AROMA, and GMF as the leading-edge technologies in this domain. These methods will be thoroughly examined and compared in our evaluation section.

Our design logic. We acknowledge the inevitability of hash collisions in sub-linear sketch structures. Our objective, therefore, is to ensure that larger flows are distinctly prominent and accurately measured, even when they share the same memory space with others. To achieve this goal,, we introduce an innovative approach named the non-duplicate sampler in Section 3.2. This design lays the foundation for our strategy to enable large flows to effectively assert their presence in shared memory environments

3 ALGORITHM DESIGN

In the section, we first review existing sketches on single-flow spread estimation, with a focus on the state of the art, i.e., HLL sketch [24]. After that, we present out novel idea of turning HLL into a non-duplicate sampler. Finally, we present out sketch design on top of HLL non-duplicate samplers.

3.1 HLL Sketch for Single-flow Spread Estimation

To measure the spread of a single flow, most prior work was based on bitmaps [22, 23, 75], FM (Flajolet-Martin) sketches [25], LogLog sketch [18] or HLL (HyperLogLog) sketches [24, 30]. Among them, HLL sketches perform the best, with the largest estimation ranges and the best overall estimation accuracy.

The data structure of HLL [24] is an array A of m registers, each of five bits. Consider a flow f, which is recorded in A for spread estimation. For each arrival data item $\langle f, e\rangle$, we perform a uniform hash $h(e) \in [0, m-1)$, which maps the item to a register $A[h(e)]$. We then calculate a geometric hash $G(e)$, which can be implemented by counting the number of leading zeros from another uniform hash $H'(e)$ and then adding one, denoted by z, such that $G(e) = z \geq 1$ with the probability of $\frac{1}{2^z}$. Since each HLL register can count up to 31, we merge the cases of $z > 30$. Accordingly, $G(e)$ follows the distribution:

$$G(e) = \begin{cases} i & \text{with probability of } \frac{1}{2^i} \text{ if } 1 \leq i \leq 30 \\ 31 & \text{with probability of } \frac{1}{2^{30}} \text{ if } i > 30 \end{cases}$$

To record the item, we let $A[h(e)] := \max\{A[h(e)], G(e)\}$. To estimate the flow's spread, denoted as $\hat{n}_f$, we compute

$$\hat{n}_f = \alpha_m \times m^2 \left(\sum_{i=0}^{m-1} 2^{-A[i]}\right)^{-1}$$

where α_m is a constant that can be calculated as $\alpha_m = \frac{0.7213}{1+\frac{1.079}{m}}$ when $m \geq 128$. Refer to [24, 30] for α_m under other values of m. With 5-bit registers, HLL can estimate flow spread up to many billions (specifically $\alpha_m \times m \times 2^{31}$), with a relative standard error of $\frac{1.04}{\sqrt{m}}$.

3.2 Turning HLL Sketch to HLL Sampler

The HLL sketch is hindered by a significant drawback: its high query overhead. Specifically, the computational load involved in generating a spread estimate is substantially greater than the overhead per recorded item. For instance, when $m \geq 128$, the query overhead can be hundreds of times more burdensome [67, 85], as it necessitates accessing m registers compared to a single register access and two hashes required for item recording. This limitation obstructs real-time awareness of flow spreads during item recording. However, we argue that real-time awareness of flow spreads can help significantly improve the accuracy of identifying super spreaders, which will be reflected in our design. To achieve this, we expect to be able to perform real-time queries on the flow's current spread during item recording, motivating us to resolve the issue of high query overhead associated with traditional HLL sketches.

In this paper, we propose a novel interpretation of the HLL sketch. We view it as a non-duplicate sampler, a concept we define in detail hereafter.

Definition 1 (Non-duplicate Sampler). *Any data structure is a non-duplicate sampler if it can block duplicate appearances of any item and samples any item at its first appearance with a probability p. The output of a non-duplicate sampler is the sampled item and its associated probability p. p is not necessarily fixed.*

It's crucial to highlight that an HLL sketch inherently functions as a non-duplicate sampler. When an item $\langle f, e\rangle$ arrives, and it's the item's first appearance, it can only modify the HLL sketch A if $G(e) > A[h(e)]$, a condition that is met with a probability p.

Algorithm 1: *HLLSampler*(f, e)

```
   Input: $A$
   Action: Perform sampling on $\langle f, e\rangle$ in HLL sampler
1  Intilization: $p$=1
2  Recording:
3  if $A[h(e)] < G(e)$ then
4      $p' = p$
5      $p += -\frac{1}{m}2^{-A[h(e)]} + \frac{1}{m}2^{-G(e)}$
6      $A[h(e)] = G(e)$
7      return $p'$
8  return -1  // -1 indicates the item is not sampled
```

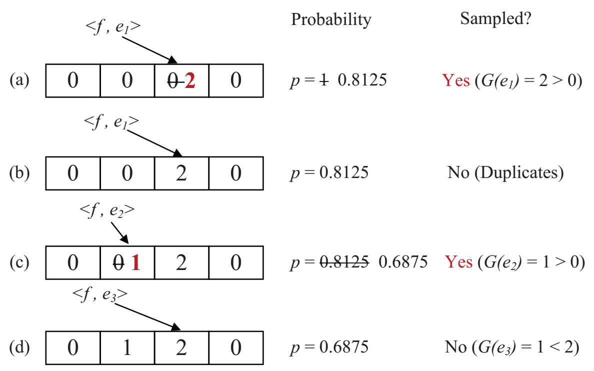


Figure 1: Running example for an HLL sampler to process a sequence of four items: $\langle f, e_1\rangle$ in plot (a), $\langle f, e_1\rangle$ in plot (b), $\langle f, e_2\rangle$ in plot (c), and $\langle f, e_3\rangle$ in plot (d). Each register in the HLL sampler is initialized as 0, and the sampling probability p is set as 1. In plot (a), $\langle f, e_1\rangle$ is hashed to the $h(e_1) = 2$th register. With $G(e_1) = 2 > 0$, $\langle f, e_1\rangle$ is sampled with probability of 1. We set the register value as 2 and p is then updated from 1 to 0.8125 for the next item; In plot (b), a duplicate appearance of $\langle f, e_1\rangle$ arrives. It is hashed to the same $h(e_1) = 2$th register. With $G(e_1) = 2 \leq 2$, $\langle f, e_1\rangle$ is not sampled; In plot (c), $\langle f, e_2\rangle$ is hashed to the $h(e_2) = 1$th register. With $G(e_2) = 1 > 0$, $\langle f, e_2\rangle$ is sampled with probability of 0.8125. We set the register value as 1 and p is then updated from 0.8125 to 0.6875 for the next item; In plot (d), item $\langle f, e_3\rangle$ is hashed to the $h(e_3) = 2$th register but it does not update the value of the register as $G(e_3) = 1 \leq 2$. As a result, the item is not sampled.

$$p = \frac{1}{m}\sum_{i=0}^{m-1}(2^{-A[i]+1} + 2^{-A[i]+2} + \ldots + 2^{-30} + 2^{-30})$$
$$= \frac{1}{m}\sum_{i=0}^{m-1} 2^{-A[i]}. \qquad (1)$$

The above expression for p is derived as any item will be mapped to one of the HLL register $A[i], 0 \leq i < m$ with equal probability, i.e, $\frac{1}{m}$ and the probability for $G(e)$ to be larger than the current value in the register $A[i]$, i.e., $G(e) > A[i]$ is $(2^{-A[i]+1} + 2^{-A[i]+2} + \ldots + 2^{-30} + 2^{-30}) = \frac{1}{2^{A[i]}}$ according to the probability distribution for $G(e)$ in Section 3.1.

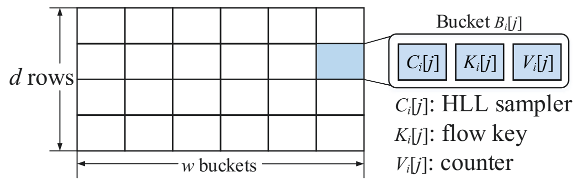


Figure 2: Data structure of the proposed solution.

In cases of duplicate appearances, if an item's initial occurrence has already set $A[h(e)] = G(e)$ when $A[h(e)] < G(e)$, it follows that $A[h(e)] \geq G(e)$. Consequently, the item $\langle f, e\rangle$ won't be able to update the HLL sketch, leading to the packet being disregarded. To circumvent the need to compute p as per equation (1) every time a register is updated, we implement a method where we keep a decimal value for p, initially set to 1. This value is adjusted whenever a register $A[i]$ changes its value from k to k', following the equation: $p += -\frac{1}{m}2^{-k} + \frac{1}{m}2^{-k'}$.

The formal algorithm of turning an HLL sketch to a non-duplicate sampler is provided in Algorithm 1. We also provide a running example in Figure 1. Redefining the HLL sketch as a non-duplicate sampler, termed the *HLL sampler*, allows for the estimation of the spread by maintaining an additional counter v. Whenever the HLL sampler generates an item and a probability p, we update v as $v += 1/p$. In practice, v is an integer and the actual update process is:

$$v += \lceil\frac{1}{p}\rceil, \text{with a probability of} \frac{\frac{1}{p}}{\lceil\frac{1}{p}\rceil}. \qquad (2)$$

HLL samplers effectively resolve the issue of high query overhead associated with traditional HLL sketches. The query overhead with an HLL sampler is minimized to accessing a single integer v. Additionally, the HLL sampler estimates spread using a Markov chain, which provides an unbiased estimation [61]. Unlike the HLL sketch where each register requires 5 bits to prevent overflow (with $G(e) = 31$ having a probability of 2^{-30} and overflow occurring if any register overflows), the HLL sampler can function effectively as long as $p \neq 0$, which is achievable as long as not all m HLL registers reach their maximum value. Thus, we can reduce each HLL register to 4 bits. Accordingly, $G(e) = i$ has a probability of $\frac{1}{2^i}$ for $1 \leq i \leq 14$, or $\frac{1}{2^{14}}$ if $i = 15$.

3.3 Super Spreader Identification using HLL Samplers

To monitor various flows in the data stream and pinpoint super spreaders, we utilize a two-dimensional array structure, illustrated in Figure 2. This consists of HLL samplers array C, flow labels array K, and counters array V, each comprising d rows and w columns. We denote the j^{th} HLL sampler, flow label, and counter in the i^{th} array as $C_i[j]$, $K_i[j]$, and $V_i[j]$ respectively, for $0 \leq i < d$, $0 \leq j < w$. These arrays can be collectively considered as a two-dimensional bucket array B, where each $B_i[j]$ contains $C_i[j]$, $K_i[j]$, and $V_i[j]$. Each array C is paired with an independent uniform hash function $h_i(\cdot)$. Initially, all flow labels in K are null, and all HLL samplers and counters are set to 0.

Algorithm 2: Online Item Recording

```
Input: $C, K, V$
Action: Perform online recording for each item $\langle f, e\rangle$
for $i$ from 0 to $d-1$ do
    $p = HLLSampler(f, e)$   // Feed $\langle f, e\rangle$ to HLL sampler; see Algorithm 1
    if $p \neq -1$ then
                        // $\langle f, e\rangle$ passes the HLL sampler
        if $K_i[h_i(f)] = null$ then
            Add Counter $C_i[h_i(f)]$ by $\frac{1}{p}$ according to (2).
            $K_i[h_i(f)] = f$
        else
            if $K_i[h_i(f)] = f$ then
                Add Counter $C_i[h_i(f)]$ by $\frac{1}{p}$ according to (2).
            else
                calculate $p_d = b^{-C_i[h_i(f)]}$
                generate a random value $r \in [0, 1)$
                if $r < p_d$ then
                    decrease $C_i[h_i(f)]$ by $\lceil\frac{1}{p}\rceil$, with probability $\frac{\frac{1}{p}}{\lceil\frac{1}{p}\rceil}$
                    if $C_i[h_i(f)] \leq 0$ then
                        $K_i[h_i(f)] = f, C_i[h_i(f)] * = -1$
```

Algorithm 3: Super Spreader Identification

```
Input: $K, V$
Action: Report super spreaders
empty set $S$
for $i$ = 0 to $d-1$ do
    for $j$ =0 to $w-1$ do
        add $K_i[j]$ to $S$
for $\forall f \in S$ do
    empty set $E$
    for $i$ = 0 to $d-1$ do
        if $K_i[h_i(f)] = f$ then
            add $C_i[h_i(f)]$ to $E$
    $\hat{n}_f$ is the maximum counter's value in $E$.
    if $\hat{n}_f > T$ then
        report $f$ as a super spreader
```

Recording: When an item $\langle f, e\rangle$ arrives, it is mapped to d buckets, $B_i[H_i(f)]$, for $0 \leq i < d$. In each bucket, the item is first processed by the HLL sampler $C_i[H_i(f)]$. If successfully sampled with a probability p, it leads to one of three cases:

• Case 1: If $K_i[h_i(f)] = null$, it means that no flows has passed through the HLL sampler $C_i[h_i(f)]$ before. We set $K_i[j] = f$ and add counter $C_i[h_i(f)]$ by $\frac{1}{p}$ according to (2).

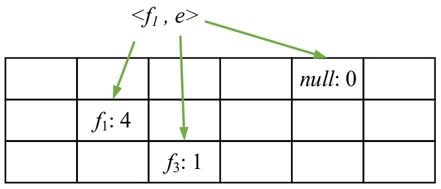


Figure 3: Running example: three recording cases when recording an item $\langle f_1, e\rangle$. $d = 3$ and in each of $d = 3$ mapped buckets, assuming $\langle f_1, e\rangle$ already passes the HLL sampler, which is thus omitted in the figure. Text in each bucket represents *flow key: counter.*

• Case 2: If $K_i[h_i(f)] = f$, we directly add counter $C_i[h_i(f)]$ by $\frac{1}{p}$ according to (2).

• Case 3: If $K_i[h_i(f)] \neq f$, a hash collision is happening. In this case, we introduce the *exponential-weakening decay* (EWD) technique that is adopted in heavy hitter identification to probabilistically keep the large flow dominating among all flows hashed to the bucket $B_i[h_i(f)]$. Specifically, we decrease $C_i[h_i(f)]$ by $\frac{1}{p}$ with a probability $p_d = b^{-C_i[h_i(f)]}$. In the case that $\frac{1}{p}$ is not an integer, we decrease by $\lceil\frac{1}{p}\rceil$, with probability $\frac{\frac{1}{p}}{\lceil\frac{1}{p}\rceil}$. The setting of p_d will be explained shortly. After decreasing, if $C_i[h_i(f)]$ is negative, we will reverse its value and replace the flow label $K_i[h_i(f)]$ with f.

Running example: As shown in Figure 3 where $d = 3$, given an incoming item $\langle f_1, e\rangle$, we compute d hash functions $H_i(f_1) \in [0, w), \forall 0 \leq i < d$ to find one bucket in each row. In each bucket, the item will first be processed by the HLL sampler. For ease of presentation, we assume the item already passed through the HLL sampler with a probability of $p_i, 0 \leq i < d$ in each mapped bucket and thus omit the HLL sampler in the figure. In the mapped bucket of the first row, the flow key field is *null* and the counter is 0, which matches Case 1. Let $p_0 = \frac{1}{2}$ Accordingly, we increases the counter by $\frac{1}{p_0} = 2$ and set the flow key field to f_1. In the mapped bucket of the second row, the flow key is exactly f_1, which matches Case 2. Let $p_1 = \frac{1}{3}$. Accordingly, we directly increases the counter from 4 to $4 + 3 = 7$. In the mapped bucket of the third row, the flow key is not f_1, which matches Case 3. Let $p_2 = \frac{1}{5}$. Accordingly, we decrease the counter by $\frac{1}{p_2} = 5$ with the probability of $p_d = 1.08^{-1}$ (assume $b = 1.08$. If hitting the probability, the decrease will make the counter to be $1 - 5 = -4$ and we will replace the flow key of f_3 with f_1 and reverse the counter from -4 to 4. In another example where $p_2 = \frac{5}{2}$ and $\frac{1}{p_2} = 2.5$ is not an integer, we will further decrease the counter by $\lceil\frac{1}{p_2}\rceil = 3$ with a probability of $\frac{\frac{1}{p_2}}{\lceil\frac{1}{p_2}\rceil} = \frac{2.5}{3} = \frac{5}{6}$.

Why exponential-weakening decay (EWD)? EWD decreases the value of counter $C_i[H_i(f)], 0 \leq i < d$ when a hash collision in the bucket $B_i[h_i(f)]$ happens. Decreasing the counter's value prevents the small flows from occupying the bucket just because it they appear before large flows in the data stream. Small flows, even if they may arrive before large flows and occupy the bucket temporarily, the counter's value will eventually be decreased to zero and the flow key field will be replaced by large flows. This would not be possible if we were to increase or keep the counter unchanged. Moreover, EWD decreases the counter's value with a probability that is an exponential function of the counter's value. An exponential function relative to the counter's value allows smaller values (representing smaller flows) an equitable chance to take the flow label, while larger values (indicating a large flow's presence) are less likely to be replaced. This can make sure that large flows, after adequate number of its distinct items are recorded, will keep occupying the bucket afterwards. Multiple large flows colliding in one HLL sampler is rare due to data skewness in practical datasets. Even in such rare cases, our design is robust as each flow is hashed to d HLL samplers. The parameter b is set as per [28], at $b = 1.08$.

We want to further detail the novelty of EWD by comparing it with the recording operation of one of the state-of-the-art solutions, i.e., SpreadSketch [60]. Consider three flows colliding in one bucket. SpreadSketch will store the flow label of one of the three flows as the candidate super spreader and take the sum of these three as the estimate of the candidate super spreader. If the three flows are two small flows and one large non-super spreader, the candidate super spreader will likely be the large non-super spreader. SpreadSketch, with the addition of two small flows' spreads, will estimate the candidate super spreader's spread larger than the threshold and report it as a super spreader. In contrast, EWD will not have this issue, as it will not add the small flows' spreads to the counters in the case of flow label mismatch. The small flows will almost make no difference to the counter's value, a circumstance that we expect. This is because EWD decreases the counter's value with a probability p_d and as the large non-super spreader begins to dominate the bucket, p_d will be extremely small and can almost be neglected.

Our contribution with exponential-weakening decay. Our main contribution is adapting the exponential-weakening decay technique, commonly used in heavy hitter identification, for super spreader identification. This adaptation overcomes the need for real-time spread knowledge, which is usually acquired by accessing the entire HLL sketch. By reinterpreting the HLL sketch as a non-duplicate HLL sampler, we simplify the process to using a counter behind the sampler for spread counting. This approach not only allows for the application of EWD in super spreader identification but also paves the way for further innovations in heavy hitter identification. Our experimental results demonstrate significant accuracy improvements with the EWD technique. Additional benefits of this reinterpretation are discussed in Section 3.2.

Super spreader identification: To identify the super spreaders, we first fetch all flow labels in K, which comprise a set S of candidate super spreaders. For each flow label $f \in S$, we access its d hashed flow labels, $K_i[h_i(f)]$, $0 \le i < d$. If $K_i[h_i(f)] = f$, f is the dominant flow in the bucket $B_i[h_i(f)]$, and consequently, add the counter $C_i[h_i(f)]$ to estimate set E. There are at most d counters and at least one counter in E. We return the maximum counter's value in E as the estimated spread of f, denoted as $\hat{n}_f$, as we will prove shortly the EWD technique will only make the estimate smaller than the real value, i.e., no over-estimation in each counter, as will prove in Section 4. If $\hat{n}_f \ge T$, report f as a super spreader, otherwise, do nothing. The detailed item recording and super spreader identification operations are given in Algorithms 2 and 3, respectively.

After recording the whole data stream, we only need K and C for identifying super spreaders. The major memory, i.e., C can be directly reset for recording in the next time window. This guarantees a very good property: Compared to other sketch-based solutions that use the whole data structure, our solution requires much less transmission bandwidth if the offline super spreader identification process is done remotely in some scenarios such as cyber security and network monitoring. This will be evaluted in Section 5.

4 ANALYSIS

In this section, we first prove that the proposed sketch is expected to under-estimate the spread of large flows in each array, which provides a theoretical support of returning the maximum value as the spread estimate in Algorithm 3. After that this section analyzes the estimation accuracy.

Theorem 1. *Let $\hat{n}_f$ be the spread estimate of the large flow f produced by Algorithm 3 and n_f be the real spread of f. We must have*

$$E(\hat{n}_f) \le n_f. \tag{3}$$

Proof. Consider the i^{th} array, $0 \le i < d$. We prove by induction. At the beginning, i.e., time $t = 0$, the counter $V_i[h_i(f)] = 0$ and n_f at time $t = 0$ is zero as well. Therefore, Eq. (3) holds.

Now let Eq. (3) holds at time $t \le 0$ and $n_f = v$. When a new item arrives at $t + 1$ there are three cases in recording operations in Section 3.3 according to Algorithm 2. In Case 1, the new item will not be switched to this case. In Case 2, the new item belongs to f and we increase the value of $V_i[h_i(f)]$ by $\lceil \frac{1}{p} \rceil$ with the probability of $\frac{\frac{1}{p}}{\lceil \frac{1}{p} \rceil} * p = frac1\lceil \frac{1}{p} \rceil$. The expected value increase is 1. Since n_f becomes n_f+1, Eq. (3) holds at time t+1. In Case 3, the new item does not belong to flow f. In this case, we decrease the value of $V_i[h_i(f)]$ by $\lceil \frac{1}{p} \rceil$ with the probability of $\frac{\frac{1}{p}}{\lceil \frac{1}{p} \rceil} * p * p_d = frac1\lceil \frac{1}{p} \rceil * p_d$. The expected value increase is $p_d \le 1$. Therefore, Eq. (3) holds at time $t + 1$. The theorem holds. □

Theorem 1 explains why we use the maximum value as the spread estimate of f in line 10 of Algorithm 3.

Theorem 2. *Let $n_{f,i}$ be the spread estimate of f in the i^{th} array. Assuming that once the flow label of a large flow f occupies the hashed flow label, it is held all the time during recording. Given a small positive number ϵ and an large flow f whose spread is n_f, we have*

$$Pr(|n_{f,i} - n_f| \le \epsilon N) \le \frac{1}{\epsilon w n_f (b-1)}, \tag{4}$$

where w the width of each array, b is the exponential base and N is the total spread in the data stream.

The proof can be inferred from [28] and is omitted due to space limit. Theorem 2 bounds the absolute error of the spread estimate on flow f. We connect the average absolute error to super spreader identification: a non-super spreader is identified as a super spreader, called a false positive, if $\max_{0 \le i < d} n_{f,i} > T$, which is equivalent to

Table 3: Number of true super spreaders in the CAIDA data set under different super spreader threshold.

Threshold	100	200	300	400	500	600	700	800
Number	309	146	88	55	49	36	34	32

Table 4: Number of true super spreaders in the E-commerce data set under different super spreader threshold.

Threshold	200	300	400	500	600	700	800
Number	659	263	146	75	52	35	24

$\exists 0 \le i < d$, the error $n_{f,i} - n_f > T - n_f$. Likewise, a super spreader is identified as a non-super spreader, called a false negative, if $\max_{0 \le i < d} n_{f,i} < T$, which is equivalent to $\forall 0 \le i < d$, the error $n_f - n_{f,i} > n_f - T$. False positives and false negatives will be adopted as performance metrics in evaluation.

5 EVALUATION

5.1 Experimental Setup

We carried out the implementation of our proposed methodology for super spreader identification, along with three leading solutions in the field: AROMA [5], GMF [44], and SpreadSketch [60]. These implementations were executed on a CPU platform, specifically on a machine equipped with an Intel Core i7-8700 3.2GHz CPU and 16GB of memory. It's important to emphasize that our inclusion of these state-of-the-art solutions aims to deliver a comprehensive and thorough comparison of accuracy in identifying super spreaders. This selection is based on our extensive knowledge and commitment to achieving high accuracy in this area. For insights into why other sketch-based solutions aren't categorized as state-of-the-art, please refer to Section 2.4.

Data sets. Our evaluation utilizes two real-world data sets. The first is the CAIDA data set, comprising real Internet traffic traces sourced from CAIDA [62]. This data set includes 10 traces, each containing tens of millions of packets. We conducted separate experiments on each of these data streams and present the averaged results. In this data set, the flow label f is defined as the destination address in each packet's header, with each trace containing around 110k flows and approximately 430k distinct data items. The element e is the source address, also from the packet header. A flow is all packets directed towards the same destination, and the flow spread represents the number of distinct sources communicating with a destination. Anomalies in flow spread could indicate flash crowds in service requests or denial-of-service attacks against a destination service, necessitating attention from service administrators. Identifying super spreaders helps detect such anomalies. Table 3 displays the number of super spreaders for various threshold values.

The second data set is the e-commerce data set, acquired from an actual e-commerce website [1]. This data set consists of three files, of which we use the visitor behavior data. Each record in this file corresponds to a product view, detailing properties such as visitor ID, timestamp, item ID, etc. It encompasses about 1.4M visitors and 235k items, with roughly 1.2M distinct item-visitor pairs ($\langle f, e \rangle$). Here, the flow label f is the item ID, and element e is the visitor ID. Views of the same item constitute a flow, with the flow spread indicating the item's popularity (number of distinct visitors viewing it). Thus, a super spreader represents a popular product item. Table 4 shows the number of super spreaders at different thresholds, illustrating that *the e-commerce data set demands higher accuracy in super spreader identification compared to the CAIDA data set due to the greater number of flows around the threshold.*

Performance metrics. We evaluate all sketches' performance in three aspects. The first is accuracy. We first define four terms to facilitate the understanding of the performance metrics. *True Positive (TP)* : Number of actual super spreaders identified. *False Positive (FP)* : Number of non-super spreaders incorrectly identified as super spreaders. *True Negative (TN)* : Number of non-super spreaders correctly not identified as super spreaders. *False Negative (FN)* : Number of actual super spreaders not identified.

We have connected the false positives and false negatives to the estimation error at the end of Section 4. Note that true negatives and true positives are the opposite of them, respectively. We use the following metrics to evaluate the accuracy performance throughout the evaluation. *Precision*: $\frac{TP}{TP+FP}$, indicating the likelihood that a reported super spreader is indeed a super spreader. *Recall*: $\frac{TP}{FN+TP}$, reflecting the probability of correctly reporting a real super spreader. *F1 Score*: $\frac{2}{recall^{-1}+precision^{-1}}$. It is the harmonic mean of precision and recall. The second is recording overhead, assessed by throughput, measured as the number of items processed per second during online encoding (Mdps). The third is restoring time, defined as the time to restore super spreaders after item recording. The unit is second.

Parameter settings. The HLL for measuring single-flow spread used in GMF contains 128 HLL registers, each of 5 bits, which are the same setting as the original papers. Each bucket in SpreadSketch contains an MRB with 438 bits for measuring single flow's spread, a register of 4 bits, and a label field of 32 bits for storing flow labels, which is the same as the original paper. In the implementation of AROMA, each flow label field is 32 bits, adequate for storing a flow label and each hash value part is 32 bits. Both are the same as the original paper. Each HLL sampler in our solution contains 128 registers. d=4, which is the classical setting [14, 28].

5.2 Accuracy Comparison under Two Data Sets

Accuracy under the CAIDA data set: We first show the experimental results of F1 score, precision, and recall with respect to the super spreader threshold T under the CAIDA data set in Figure 4. We want to stress that varying T is equivalent to varying the number of real super spreaders. Their relation can be found in Tables 3 and 4 for the CAIDA and e-commerce data set, respectively. The memory allocation for each sketch is 2Mb. The threshold varies from 100 to 800 with a step length of 100. As shown in Table 3, this threshold range yield to the number of true super spreaders from 309 to 32, which is a common setting [60, 67]. The first plot in Figure 4 shows that the proposed solution (the red line) can improve the F1 score visually. Existing state of the art has similar performance with a slight better performance from AROMA. In numbers, the proposed solution improves the F1 score by 1.7%, 13.0%, 11.4%, 8.3%, 13.2%, 8.09%, 6.7% and 7.3%, 21.2%, 19.8%, 13.6%, 11.7%, 18.1%, 15.1%, 13.1% and 9.3%, and 5.5%, 14.2%, 10.3%, 9.7%, 14.8%, 10.5%, 9.1%

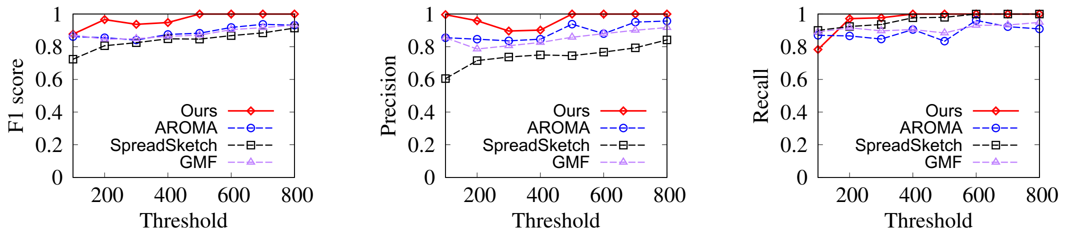


Figure 4: Accuracy comparison vs. threshold in super spreader identification under 2Mb memory and the CAIDA data set.

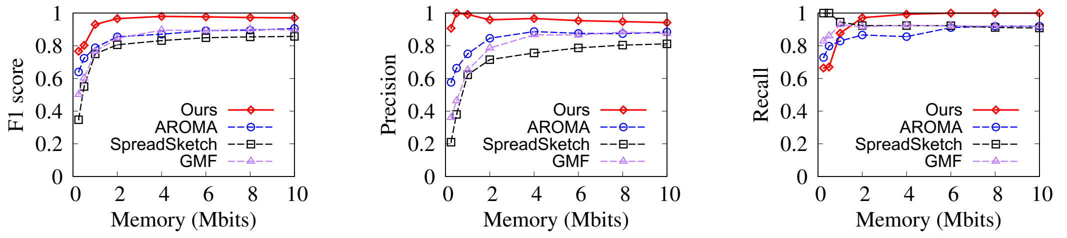


Figure 5: Accuracy comparison vs. memory allocation in super spreader identification under $T = 200$ and the CAIDA data set.

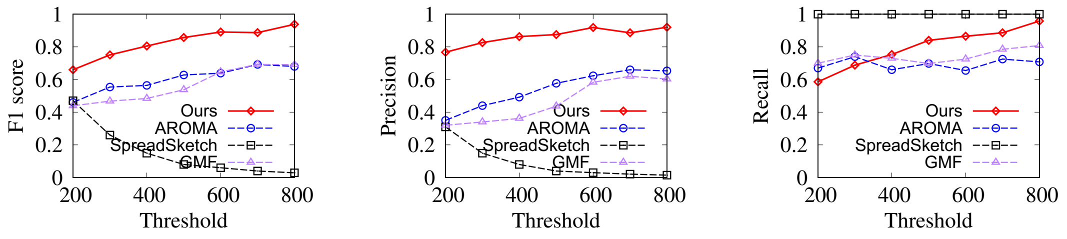


Figure 6: Accuracy comparison vs. threshold in super spreader identification under 2Mb memory and the e-commerce data set.

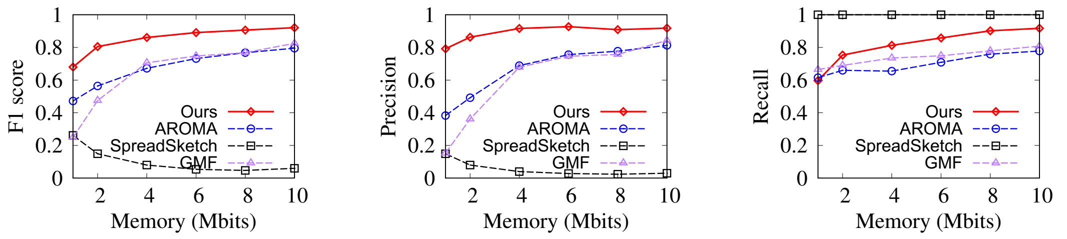


Figure 7: Accuracy comparison vs. memory allocation in super spreader identification under $T = 400$ and e-commerce data set.

and 7.2%, compared to AROMA, SpreadSketch and GMF, under the threshold from 100 to 800, respectively. Next we go to the performance in precision and recall. In the second plot of Figure 4, we find SpreadSketch suffers from a low precision, meaning that there are many false positives. The reason is that SpreadSketch usually over estimate the spread of a flow — there are hash collisions among flows in a single MRB and SpreadSketch uses the estimate produced by MRB as the spread of the super spreaders, which carries noise from other flows. This is consistent with the recall results in the third plot of Figure 4, where the recall of SpreadSketch is almost 1 and there is hardly false negatives. AROMA is based on sampling. Its spread estimate is unbiased, so its precision and recall are pretty balanced. AROMA is not as accurate as the proposed solution due to its low sampling probability. GMF uses vSketch(HLL) [85] to produce the spread estimate for candidate super spreaders. Although its estimation is unbiased, it is not as accurate as it subtracts identical noise for each flow, which is not true in real noise distributions.

The second set of experiments evaluate the performance by varying the memory allocations to each sketch, from 256Kb to 10Mb, presented in Figure 5. The threshold T is set as 200, where there are 146 true super spreaders. All sketches benefit from large memory but the proposed solution maintains the highest F1 score. Under

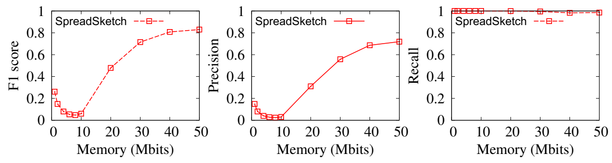


Figure 8: Accuracy performance of SpreadSketch under large memory allocations. The threshold is set as 400 and we use the e-commerce data set. This explains SpreadSketch performs poorly under 2Mb memory and need more memory.

small memory allocations, e.g., 256 kbits, the proposed solution still manage to identify super spreaders with 0.8 of F1 score. While other sketches suffer. For example, SpreadSketch has n F1 score of below 0.4 and a precision of 0.2, meaning that 4 out of 5 reported super spreaders by SpreadSketch are non super spreaders.

Accuracy under the e-commerce data set: Now we use a more challenging data set, which has a denser distribution of flows in the spread range [200, 800] than the CAIDA data set, as shown in Tables 3 and 4. This means the solutions are expected to estimate the spread of super spreaders and non super spreaders (especially those with close spread to the threshold) accurately in order to differentiate them. The memory is set as 2Mb. We first present the accuracy under different threshold T, presented in Figure 6, where we find existing sketch, i.e., AROMA, SpreadSketch and GMF, has a significant accuracy drop compared to the results under the CAIDA data set in Figure 4. The proposed solution still achieves a satisfying performance, ranging from 0.65 to 0.93 under different values of threshold. In numbers, the proposed solution increases the F1 score by 43.4%, 35.5%, 42.7%, 36.4%, 39.4%, 28.1% and 37.9%, 40.4%, 188.8%, 440.2%, 971.2%, 1185.0%, 2117.5% and 3134.4%, and 50.2%, 60.5%, 66.3%, 59.4%, 37.8%, 28.0%, 35.8%, compared to AROMA, SpreadSketch and GMF, under the threshold from 200 to 800, respectively.

One may doubt the correctness of implementation of SpreadSketch as its identification accuracy is so low despite that the e-commerce data set is more challenging: The F1 score is always below 0.5 and below 0.2 if the threshold $T \geq 400$. We explain this counter-intuitive performance by extending the memory range from [1Mb, 10Mb] to [1Mb, 50Mb]. The results are shown in Figure 8. The threshold is set as 400. Figure 8 shows that the F1 score of SpreadSketch decreases at first, bottoms at memory allocation of 8Mb and increases afterwards. This is because of two-way factors. One is the number of buckets to store candidate super spreaders, and the other is the estimation error. Recall that SpreadSketch has over-estimation issue: flows may collide in one bucket and deposit noise to the bucket. SpreadSketh returns the minimum to reduce the noise. However, the noise level is too large that if the memory is less than 10Mb, many flows (including all real super spreaders and many non super spreaders) will reach the threshold and be reported as super spreaders. Smaller memory will result in fewer number of buckets, fewer number of candidate super spreaders and consequently fewer false positives. This can be verified by the Recall results of always being 1 and increasing precision results as the memory decreases. When the memory is larger than 10Mb, noise level is smaller and flows are more accurately measured. As a

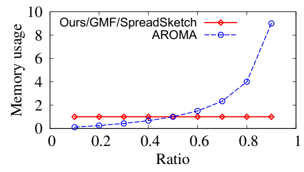
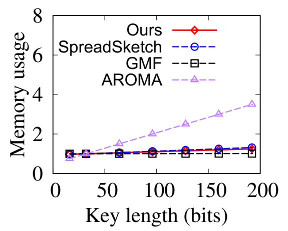


Figure 9: Memory needed to achieve the same super spreader identification accuracy, when inserting an artificial large flow, normalized to the case of no inserted flows. Ratio in x-axis means the rate of the spread of the inserted flow over the total spread of the data set.



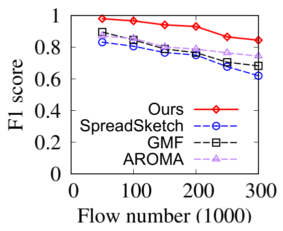


Figure 10: Memory needed for different length of flow labels, normalized to the case of 32 bits per flow label.

Figure 11: Accuracy comparison vs. the number of flows under 2Mb memory and the CAIDA data set.

result, the precision results and F1 score keep increasing as memory increases.

The second set of experiments are conducted by varying the memory allocations. The threshold is set as 400 for a adequate number of true super spreaders. We start from 1Mb, a larger memory than that under the CAIDA data set, to 10Mb. The results are plotted in Figure 7. The results are consistent with those under the CAIDA data set and we omit the description here.

5.3 Accuracy under Two Extreme Cases

Case 1: A dominant flow in the data stream. The practical data set is usually skewed, and will become even more skewed when there are abnormals. To simulate this case, we insert an artificial flow with a large spread to the CAIDA data set. Let x be the spread of the inserted flow. We define the *ratio* as the rate of x over the total number of spread in the data set. We vary the value of the ratio from 0.1 to 0.9, feed the new data set to all sketches separately and evaluate their accuracy performance. The metric adopted is the normalized memory usage to the case of the raw CAIDA data set in order to get the same accuracy performance. The results are shown in Figure 9. As the large flow dominates the whole data set more and more severely, the memory usage needed for keep the accuracy level unchanged grows faster. The reason is that AROMA is based on sampling. The large flow's items will overwhelm the whole data structure, making the sampling probability for other flows' item very small. To maintain the same probability, AROMA needs a lot of additional memory. SpreadSketch, GMF and the proposed solution are not affected by the inserted large flow, as any flow, no matter

Table 5: Recording throughput (million items per second) comparison, under 2Mb memory and CAIDA data set.

Sketches	Ours	AROMA	SpreadSketch	GMF
Throughput	10.6	38.8	14.2	20.5

Table 6: Restoring time (second) for reporting super spreaders. The data set used is CAIDA and the memory is 2Mb.

Sketches	Ours	AROMA	SpreadSketch	GMF
Time (s)	0.001	0.002	0.026	0.503

small or large can only be hashed to one or d (which is usually 4) plug-ins, rather the whole data structure.

Case 2: Long key for flow label. In some cases, such as finding hot topics in social networks or keywords on search engine, the flow label (which is the topic or the keywords) may be a long string, which needs more bits to be stored. In the section, we vary the flow label from 16 bits to 192 bits where 32 bits per flow label is the default setting. We still use the metric of memory usage needed for maintaning the same accuracy level, normalized to that under the default setting of 32 bits. The results are shown in Figure 10, where we find AROMA needs an increasing memroy with a large slope as the the length of the flow label increases. The reason is that AROMA's data structure is an array of slots, where most of the bits go to the flow label field in each slot. In comparison, SpreadSketch, GMF, and the proposed solution is scalable for the key length.

5.4 Recording Throughput Comparison

This section evaluates the online item-recording speed, which is also the maximum streaming speed that the solution can support. We use the recording throughput as performance metric, defined as the number of items processed per second. We use the CAIDA data set and allocate 2Mb memory for each sketch. We want to stress that the type of data set and memory allocation will not affect the recording throughput. The results in Table 5 show that all sketches can process more than 10 million items per second, which is a very high throughput. Relatively AROAM has the highest recording throughput as it only records each item once while the proposed solution and SpreadSketch records d times, and GMF records twice (once in the filter and once in the vSketch). We argue that all solutions that are specially design for high-speed data streams can meet the requirement of practical applications. For the CAIDA packet stream, a recording throughput of 1 million packets per second is equivalent to 1Gbps of bandwidth, assuming that the average packet size is 1Kbits [27]. A >10 Gbps of throughput can match the line rate of current commodity high-speed networks [4, 10, 52]. For the e-commerce data stream, a recording throughput of >10 millions items per second (which is true for the proposed solution, AROMA, SpreadSketch and GMF) can safely support the main-stream e-commerce platform— Taobao, one of the biggest e-commerce platforms in China, receives 0.1 million click logs per seconds [37].

Table 7: Memory transmitted for remote super spreader restoring, under 2Mb memory and CAIDA data set.

Sketches	Ours	AROMA	SpreadSketch	GMF
Memory	0.11Mb	2Mb	2Mb	2Mb

5.5 Restoring Time Comparison

After item recording, all sketches need to report the super spreaders by restoring the super spreaders in the recorded sketches. We want this restoring process to be as fast as possible and adopt the restoring time as metric. The results are shown in Table 6, where the memory is 2Mb, T=200, and the data set used is CAIDA. Our solution needs 0.001 second to restore all super spreaders while the slowest one, i.e., GMF, needs 0.5 second. Longer restoring time means more computational overhead, which is not desirable when the computing resource is limited.

5.6 Transmission Bandwidth Consumption

In some networking scenarios, due to limited resources on online processors, we may expect to restore or store the super spreaders remotely on a powerful server, so that 1) the server can store measurements across multiple measurement windows or among multiple measurement points and 2) release the computation resources on online processors. In this case, we need to transmit the measurements through networks. One issue is the bandwidth usage. We want to minimize that. In Table 7, we list the size of memory that should be transmitted. The memory allocation is 2Mb. As we can see, AROMA, SpreadSketch and GMF all need to send the whole sketch while the proposed solution only needs to send array K and V, which only take 0.11Mb. In other words, the proposed solution can reduce 94.5% of the bandwidth usage compared to AROMA, SpreadSketch and GMF.

5.7 Impact of Number of Flows

We use the CAIDA data set to evaluate the impact of the number of flows to accuracy, where the number of flows ranges from 50k to 300k, and the memory is 2Mb. The results in Figure 11 show that the proposed solution increases the F1 score by [12.5%, 18.1%], [17.9%, 36.1%] , [9.5%, 23.1%] compared to AROMA, SpreadSketch, and GMF, respectively, under different numbers of flows.

6 CONCLUSION

This paper proposes a new sketch design for super spreader identification. We turn the existing single-flow spread estimators into non-duplicate samplers, enabling accurate counting of flow spread with just counters. By employing the effectively exponential-weakening delay technique, large flows stand out among all flows with their spread being accurately measured. Trace-driven experiments demonstrate that the proposed solution statistically improving the super spreader identification accuracy compared to the existing state-of-the-art. The proposed solution also has the lowest super spreader restoring time and one order of magnitude less bandwidth usage if the offline restoring is performed remotely.

ACKNOWLEDGMENTS

This work is supported by the University of Kentucky start-up fund.

REFERENCES

[1] 2016. Retailrocket Recommender System Dataset, last accessed date: July 10 2024. https://www.kaggle.com/retailrocket/ecommerce-dataset.

[2] A. Akella, A. Bharambe, M. Reiter, and S. Seshan. 2003. Detecting DDoS Attacks on ISP Networks. In *Proceedings of the Twenty-Second ACM SIGMOD/PODS Workshop on Management and Processing of Data Streams*. 1–3.

[3] J. Anton, L. Jacobs, X. Liu, J. Parker, Z. Zeng, and . Zhong. 2002. Web Caching for Database Applications with Oracle Web Cache. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. 594–599.

[4] Shunji Aoyagi, Jong-Deok Kim, Kien Nguyen, and Hiroo Sekiya. 2023. An Extensive Evaluation of TCP Congestion Control in 10 Gbps Network. In *2023 24st Asia-Pacific Network Operations and Management Symposium (APNOMS)*. IEEE, 306–309.

[5] Ran Ben Basat, Xiaoqi Chen, Gil Einziger, Shir Landau Feibish, Danny Raz, and Minlan Yu. 2020. Routing Oblivious Measurement Analytics. In *2020 IFIP Networking Conference (Networking)*. IEEE, 449–457.

[6] Ran Ben-Basat, Gil Einziger, Roy Friedman, and Yaron Kassner. 2016. Heavy Hitters in Streams and Sliding Windows. In *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*. IEEE, 1–9.

[7] Ran Ben Basat, Gil Einziger, Roy Friedman, Marcelo C Luizelli, and Erez Waisbard. 2017. Constant Time Updates in Hierarchical Heavy Hitters. *Proc. of ACM SIGCOMM* (2017).

[8] A. Bronselaer, S. Debergh, D. Van Hyfte, and G. D. Tré. 2010. Estimation of Topic Cardinality in Document Collections. In *SIAM Conference on Data Mining (SDM 10)*. SIAM, 31–39.

[9] Jing Cao, Yu Jin, Aiyou Chen, Tian Bu, and Z-L Zhang. 2009. Identifying high cardinality internet hosts. In *IEEE INFOCOM 2009*. IEEE, 810–818.

[10] Jiale Chen, Xingshu Chen, Liangguo Chen, Xiao Lan, and Yonggang Luo. 2023. ANTI: An Adaptive Network Traffic Indexing Algorithm for High-Speed Networks. In *GLOBECOM 2023-2023 IEEE Global Communications Conference*. IEEE, 1699–1704.

[11] S. Chen and Y. Tang. 2004. Slowing Down Internet Worms. In *24th International Conference on Distributed Computing Systems, 2004. Proceedings.* IEEE, 312–319.

[12] Peter Cogan, Matthew Andrews, Milan Bradonjic, W Sean Kennedy, Alessandra Sala, and Gabriel Tucci. 2012. Reconstruction and analysis of twitter conversation graphs. In *Proceedings of the First ACM International Workshop on Hot Topics on Interdisciplinary Social Networks Research*. 25–31.

[13] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2009. *Introduction to Algorithms*. MIT press.

[14] Graham Cormode and S. Muthukrishnan. 2004. An Improved Data Stream Summary: the Count-Min Sketch and Its Applications. *Proc. of LATIN* (2004).

[15] 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.

[16] E. Demaine, A. Lopez-Ortiz, and J. Munro. 2002. Frequency Estimation of Internet Packet Streams with Limited Space. *Proc. of 10th ESA Annual European Symposium on Algorithms* (September 2002).

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

[18] M. Durand and P. Flajolet. 2003. Loglog Counting of Large Cardinalities. In *European Symposium on Algorithms*. Springer, 605–617.

[19] Z. Durumeric, M. Bailey, and J A. Halderman. 2014. An Internet-wide View of Internet-wide Scanning. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*. 65–78.

[20] Gil Einziger and Roy Friedman. 2019. Counting with Tinytable: Every Bit Counts! *IEEE Access* (2019).

[21] Cristian Estan, George Varghese, and Mike Fisk. 2003. Bitmap algorithms for counting active flows on high speed links. In *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*. 153–166.

[22] C. Estan, G. Varghese, and M. Fisk. 2003. Bitmap Algorithms for Counting Active Flows on High Speed Links. In *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*. 153–166.

[23] C. Estan, G. Varghese, and M. Fisk. 2006. Bitmap Algorithms for Counting Active Flows on High-speed Links. *IEEE/ACM Transactions on Networking* 14, 5 (2006), 925–937.

[24] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier. 2007. Hyperloglog: the Analysis of a Near-optimal Cardinality Estimation Algorithm. In *Discrete Mathematics and Theoretical Computer Science*. Discrete Mathematics and Theoretical Computer Science, 137–156.

[25] P. Flajolet and G N. Martin. 1985. Probabilistic Counting Algorithms for Data Base Applications. *Journal of computer and system sciences* 31, 2 (1985), 182–209.

[26] S. Ganguly, M. Garofalakis, R. Rastogi, and K. Sabnani. 2007. Streaming Algorithms for Robust, Real-Time Detection of DDoS Attacks. In *27th International Conference on Distributed Computing Systems (ICDCS '07)*. 4–4.

[27] Eimantas Garsva, Nerijus Paulauskas, and Gediminas Grazulevicius. 2015. Packet size distribution tendencies in computer network flows. In *2015 Open Conference of Electrical, Electronic and Information Sciences (eStream)*. IEEE, 1–6.

[28] J. Gong, T. Yang, H. Zhang, H. Li, S. Uhlig, S. Chen, L. Uden, and X. Li. 2018. HeavyKeeper: An Accurate Algorithm for Finding Top-k Elephant Flows. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 909–921. https://www.usenix.org/conference/atc18/presentation/gong

[29] A. Hall, O. Bachmann, R. Büssow, S. Gănceanu, and M. Nunkesser. 2012. Processing a Trillion Cells per Mouse Click. *Proceedings of the VLDB Endowment* 5, 11 (2012), 1436–1446.

[30] S. Heule, M. Nunkesser, and A. Hall. 2013. HyperLogLog in Practice: Algorithmic Engineering of a State of the Art Cardinality Estimation Algorithm. In *Proceedings of the 16th International Conference on Extending Database Technology*. ACM, 683–692.

[31] Q. Huang, X. Jin, P. P. Lee, R. Li, L. Tang, Y. Chen, and G. Zhang. 2017. Sketchvisor: Robust Network Measurement for Software Packet Processing. In *Proceedings of the 2017 ACM SIGCOMM Conference*. 113–126.

[32] Q. Huang, P. P. C. Lee, and Y. Bao. 2018. SketchLearn: Relieving User Burdens in Approximate Measurement with Automated Statistical Inference. *Proc. of ACM SIGCOMM* (August 2018), 576–590.

[33] Nicole Immorlica, Kamal Jain, Mohammad Mahdian, and Kunal Talwar. 2005. Click fraud resistant methods for learning click-through rates. In *Internet and Network Economics: First International Workshop, WINE 2005, Hong Kong, China, December 15-17, 2005. Proceedings 1*. Springer, 34–45.

[34] 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.

[35] Noriaki Kamiyama, Tatsuya Mori, and Ryoichi Kawahara. 2007. Simple and adaptive identification of superspreaders by flow sampling. In *IEEE INFOCOM 2007-26th IEEE International Conference on Computer Communications*. IEEE, 2481–2485.

[36] Raymond YK Lau, SY Liao, Ron Chi-Wai Kwok, Kaiquan Xu, Yunqing Xia, and Yuefeng Li. 2012. Text mining and probabilistic language modeling for online review spam detection. *ACM Transactions on Management Information Systems (TMIS)* 2, 4 (2012), 1–30.

[37] Xiang Li, Chao Wang, Jiwei Tan, Xiaoyi Zeng, Dan Ou, Dan Ou, and Bo Zheng. 2020. Adversarial multimodal representation learning for click-through rate prediction. In *Proceedings of The Web Conference 2020*. 827–836.

[38] Y. Li, R. Miao, C. Kim, and M. Yu. 2016. Flowradar: A Better Netflow for Data Centers. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*. 311–324.

[39] 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.

[40] Y. Liu, W. Chen, and Y. Guan. 2016. Identifying High-Cardinality Hosts from Network-Wide Traffic Measurements. *IEEE Transactions on Dependable and Secure Computing* 13, 5 (Sep. 2016), 547–558. https://doi.org/10.1109/TDSC.2015.2423675

[41] Z. Liu, R. Ben-Basat, G. Einziger, Y. Kassner, V. Braverman, R. Friedman, and V. Sekar. 2019. Nitrosketch: Robust and General Sketch-based Monitoring in Software Switches. In *Proceedings of the 2019 ACM SIGCOMM Conference*. 334–350.

[42] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman. 2016. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with Univmon. In *Proceedings of the 2016 ACM SIGCOMM Conference*. 101–114.

[43] Zaoxing Liu, Hun Namkung, Georgios Nikolaidis, Jeongkeun Lee, Changhoon Kim, Xin Jin, Vladimir Braverman, Minlan Yu, and Vyas Sekar. 2021. Jaqen: A {High-Performance} {Switch-Native} approach for detecting and mitigating volumetric {DDoS} attacks with programmable switches. In *30th USENIX Security Symposium (USENIX Security 21)*. 3829–3846.

[44] 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.

[45] Chaoyi Ma, Olufemi O Odegbile, Dimitrios Melissourgos, Haibo Wang, and Shiping Chen. 2023. From CountMin to Super kJoin Sketches for Flow Spread Estimation. *IEEE Transactions on Network Science and Engineering* (2023).

[46] Ankush Mandal, He Jiang, Anshumali Shrivastava, and Vivek Sarkar. 2018. Topkapi: parallel and fast sketches for finding top-k frequent elements. *Advances in Neural Information Processing Systems* 31 (2018).

[47] G. Manku and R. Motwani. 2002. Approximate Frequency Counts over Data Streams. *Proc. of VLDB* (August 2002).

[48] Dimitrios Melissourgos, Haibo Wang, Shigang Chen, Chaoyi Ma, and Shiping Chen. 2023. Single Update Sketch with Variable Counter Structure. *Proceedings of the VLDB Endowment* 16, 13 (2023), 4296–4309.

[49] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. 2010. Dremel: Interactive Analysis of Web-scale Datasets. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 330–339.

[50] A. Metwally, D. Agrawal, and A. E. Abbadi. 2005. Efficient Computation of Frequent and Top-k Elements in Data Streams. *Proc. of 10th International Conference*

on Database Theory (ICDT) (January 2005).

[51] J. Mirkovic and P. Reiher. 2004. A Taxonomy of DDoS Attack and DDoS Defense Mechanisms. *ACM SIGCOMM Computer Communication Review* 34, 2 (2004), 39–53.

[52] Duc-Minh Ngo, Cuong Pham-Quoc, and Tran Ngoc Thinh. 2018. An efficient high-throughput and low-latency syn flood defender for high-speed networks. *Security and Communication Networks* 2018 (2018), 1–14.

[53] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. 2005. Interpreting the Data: Parallel Analysis with Sawzall. *Scientific Programming* 13, 4 (2005), 277–298.

[54] David Plonka. 2000. FlowScan: A Network Traffic Flow Reporting and Visualization Tool.. In *LISA*. 305–317.

[55] Martin Roesch et al. 1999. Snort: Lightweight intrusion detection for networks.. In *Lisa*, Vol. 99. 229–238.

[56] P. Roy, A. Khan, and G. Alonso. 2016. Augmented Sketch: Faster and More Accurate Stream Processing. In *Proceedings of the 2016 International Conference on Management of Data*. 1449–1463.

[57] S. Sen and J. Wang. 2002. Analyzing Peer-to-peer Traffic Across Large Networks. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*. ACM, 137–150.

[58] S. Singh, C. Estan, G. Varghese, and S. Savage. 2004. Automated Worm Fingerprinting. In *OSDI*, Vol. 4. 4–4.

[59] A. Sperotto, G. Schaffrath, R. Sadre, C. Morariu, A. Pras, and B. Stiller. 2010. An Overview of IP Flow-based Intrusion Detection. *IEEE communications surveys & tutorials* 12, 3 (2010), 343–356.

[60] L. Tang, Q. Huang, and P. Lee. 2020. SpreadSketch: Toward Invertible and Network-Wide Detection of Superspreaders. (2020).

[61] D. Ting. 2014. 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*. 442–451.

[62] UCSD. 2015. CAIDA UCSD Anonymized 2015 Internet Traces on Jan. 17, last accessed date: July 10 2024,. https://www.caida.org/data/passive/passive_2015_dataset.xml.

[63] M. Vartak, V. Raghavan, and E. Rundensteiner. 2010. Qrelx: Generating Meaningful Queries That Provide Cardinality Assurance. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 1215–1218.

[64] Ss Venkataraman, Ds Song, Ps B Gibbons, and As Blum. 2005. *New Streaming Algorithms for Fast Detection of Superspreaders*. Technical Report.

[65] Haibo Wang, Chaoyi Ma, Shigang Chen, and Yuanda Wang. 2022. Fast and accurate cardinality estimation by self-morphing bitmaps. *IEEE/ACM Transactions on Networking* 30, 4 (2022), 1674–1688.

[66] Haibo Wang, Chaoyi Ma, Shigang Chen, and Yuanda Wang. 2022. Online Cardinality Estimation by Self-morphing Bitmaps. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 1–13.

[67] 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.

[68] Haibo Wang, Chaoyi Ma, Olufemi O Odegbile, Shigang Chen, and Jih-Kwon Peir. 2022. Randomized error removal for online spread estimation in high-speed networks. *IEEE/ACM Transactions on Networking* (2022).

[69] Haibo Wang, Dimitrios Melissourgos, Chaoyi Ma, and Shigang Chen. 2023. Real-time Spread Burst Detection in Data Streaming. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 7, 2 (2023), 1–31.

[70] H. Wang, H. Xu, L. Huang, and Y. Zhai. 2020. Fast and Accurate Traffic Measurement with Hierarchical Filtering. *IEEE Transactions on Parallel and Distributed Systems* 31, 10 (2020), 2360–2374.

[71] L. Wang, T. Yang, H. Wang, J. Jiang, Z. Cai, B. Cui, and X. Li. 2019. Fine-grained Probability Counting for Cardinality Estimation of Data Streams. *World Wide Web* 22, 5 (2019), 2065–2081.

[72] 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.

[73] Pinghui Wang, Peng Jia, Xiangliang Zhang, Jing Tao, Xiaohong Guan, and Don Towsley. 2019. Utilizing dynamic properties of sharing bits and registers to estimate user cardinalities over time. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1094–1105.

[74] Jianshu Weng, Ee-Peng Lim, Jing Jiang, and Qi He. 2010. Twitterrank: finding topic-sensitive influential twitterers. In *Proceedings of the third ACM international conference on Web search and data mining*. 261–270.

[75] K. Whang, B. T Vander-Zanden, and H. M Taylor. 1990. A Linear-time Probabilistic Counting Algorithm for Database Applications. *ACM Transactions on Database Systems (TODS)* 15, 2 (1990), 208–229.

[76] 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 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. 417–428.

[77] Qingjun Xiao, Shigang Chen, You Zhou, Min Chen, Junzhou Luo, Tengli Li, and Yibei Ling. 2017. Cardinality estimation for elephant flows: A compact solution based on virtual register sharing. *IEEE/ACM Transactions on Networking* 25, 6 (2017), 3738–3752.

[78] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig. 2018. Elastic Sketch: Adaptive and Fast Network-wide Measurements. In *Proceedings of the 2018 ACM SIGCOMM Conference*. 561–575.

[79] M. Yoon, T. Li, S. Chen, and J. Peir. 2009. Fit a Spread Estimator in Small Memory. In *IEEE INFOCOM 2009*. IEEE, 504–512.

[80] M Yoon, Tao Li, Shigang Chen, and J-K Peir. 2009. Fit a spread estimator in small memory. In *IEEE INFOCOM 2009*. IEEE, 504–512.

[81] MyungKeun Yoon, Tao Li, Shigang Chen, and Jih-Kwon Peir. 2010. Fit a compact spread estimator in small high-speed memory. *IEEE/ACM Transactions on Networking* 19, 5 (2010), 1253–1264.

[82] M. Yu, L. Jose, and R. Miao. 2013. Software Defined Traffic Measurement with OpenSketch. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. 29–42.

[83] X. Yu, H. Xu, D. Yao, H. Wang, and L. Huang. 2018. Countmax: A Lightweight and Cooperative Sketch Measurement for Software-defined Networks. *IEEE/ACM Transactions on Networking* 26, 6 (2018), 2774–2786.

[84] Y. Zhou, T. Yang, J. Jiang, B. Cui, M. Yu, X. Li, and S. Uhlig. 2018. Cold Filter: A Meta-framework for Faster and More Accurate Stream Processing. In *Proceedings of the 2018 International Conference on Management of Data*. 741–756.

[85] Y. Zhou, Y. Zhang, C. Ma, S. Chen, and O. O Odegbile. 2019. Generalized Sketch Families for Network Traffic Measurement. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 3, 3 (2019), 1–34.