Real-time Spread Burst Detection in Data Streaming

HAIBO WANG*, University of Florida, USA
DIMITRIOS MELISSOURGOS*, Grand Valley State University, USA
CHAOYI MA, University of Florida, USA
SHIGANG CHEN, University of Florida, USA

Data streaming has many applications in network monitoring, web services, e-commerce, stock trading, social networks, and distributed sensing. This paper introduces a new problem of real-time burst detection in flow spread, which differs from the traditional problem of burst detection in flow size. It is practically significant with potential applications in cybersecurity, network engineering, and trend identification on the Internet. It is a challenging problem because estimating flow spread requires us to remember all past data items and detecting bursts in real time requires us to minimize spread estimation overhead, which was not the priority in most prior work. This paper provides the first efficient, real-time solution for spread burst detection. It is designed based on a new real-time super spreader identifier, which outperforms the state of the art in terms of both accuracy and processing overhead. The super spreader identifier is in turn based on a new sketch design for real-time spread estimation, which outperforms the best existing sketches.

CCS Concepts: • **Networks** → **Network measurement**; **Network monitoring**;

Additional Key Words and Phrases: Spread Burst, Real-time, Data Streaming

ACM Reference Format:
Haibo Wang, Dimitrios Melissourgos, Chaoyi Ma, and Shigang Chen. 2023. Real-time Spread Burst Detection in Data Streaming. *Proc. ACM Meas. Anal. Comput. Syst.* 7, 2, Article 35 (June 2023), 29 pages. https://doi.org/10.1145/3578338.3593566

1 INTRODUCTION

Data streaming is the continuous production of data items which must be immediately processed to support real-time queries based on up-to-the-moment information. It has wide applications in network monitoring, web services, e-commerce, stock trading, social networks, and distributed sensing. Its growing practical importance is evident from industrial pushes (such as Amazon Kinesis Streams [1]) that enable customizable streaming applications.

For example, the stream of packets that are received by a router's network interface at tens of millions of packets per second can be modeled as a data stream, with each data item (i.e., packet) carrying a flow ID f and a data element e of interest, where f and e are defined based on application need. All items (packets) with the same flow ID form a flow. We may also treat the stream of user queries that arrive at an Internet search engine, the stream of purchases at an e-commerce site, the

*co-first authors with equal contribution.

Authors' addresses: Haibo Wang, University of Florida, Gainesville, FL, USA, wanghaibo@ufl.edu; Dimitrios Melissourgos, Grand Valley State University, Allendale, MI, USA, dmelissourgos@gmail.com; Chaoyi Ma, University of Florida, Gainesville, FL, USA, ch.ma@ufl.edu; Shigang Chen, University of Florida, Gainesville, FL, USA, sgchen@cise.ufl.edu.

2476-1249/2023/6-ART35 $15.00
https://doi.org/10.1145/3578338.3593566

stream of stock trades at an electronic exchange, or the stream of posts at a social network as a data stream.

Much research interest in data streaming has been directed toward recording the data items in compact and efficient data structures called *sketches* and extracting useful statistics from the sketches [2, 4, 6, 7, 13, 18, 30, 31, 36, 37, 41–43, 46, 47]. They are very useful in dealing with an extremely high data rate using limited resources, such as (1) monitoring a packet stream on the data plane of a router at the network processor chip using SRAM and computation circuitry that are also needed by key network functions such as packet forwarding, or (2) processing a data stream of Internet searches, e-commerce purchases, stock trades, or social network posts by an ordinary computer for cost and convenience reasons. Two basic statistics of common interest are the number of items in each flow (called *size*) and the number of *distinct* items in each flow (called *spread*). Sketches for flow spread are much more complex and expensive to operate than those for flow size because they have to remember the past items and count only the new ones.

This paper investigates real-time spread burst detection in data streaming. We want to detect *burst increase*, in which a flow's spread suddenly jumps larger, *burst decrease*, in which a flow's spread suddenly drops significantly, and *spread burst*, which starts with a burst increase and follows with a burst decrease. Detecting such patterns in real time has many important applications. We give a few examples below.

- *Cybersecurity:* Consider the task of monitoring a packet stream with flow ID f being the source address and data element e being the destination address/port in each packet. A burst increase may suggest the onset of network scanning activity. Regularly-reoccurring spread bursts may suggest scheduled scanning activities (such as some Internet worms, CodeRed [22, 48] for instance). In another example, if we let f be the destination address and e being the source address, a burst increase may suggest the onset of a denial-of-service attack. Regularly-reoccurring spread bursts may suggest scheduled botnet activities.
- *Network engineering:* TCP's congestion control can be gamed by creating a large number of parallel connections. If we let f be the source address and e be the source port and destination address/port, detection of burst increase and burst decrease provides additional information for a router to intelligently drop packets against parallel connections during congestion. In another example, if we let f be the URL in HTTP packets and e be the source address, detection of bursts informs a web proxy to determine its caching priorities based on the changing popularity of web content.
- *Internet search:* If we let f be the search keyword and e be the host address (or cookie) that issues the search, a burst increase of a keyword indicates rising interest. If it follows with a burst decrease, it suggests the interest is temporary.
- *E-commerce:* If we let f be the product ID and e be the customer that makes the purchase, a burst increase (decrease) indicates the product is gaining (losing) popularity.

If we can detect bursts in real time, we can react to them in real time, by blocking out potentially malicious sources, taking timely actions to improve network performance, or optimizing digital ads based on the trends on Internet search or e-commerce.

Burst detection in data streaming has drawn research interest recently [23, 40, 45], but only for bursts in flow size. This paper investigates burst detection in flow spread, which has never been studied before. We use an example to illustrate the difference. Consider a packet stream with flow ID f being the source address and data element e being the destination address/port. Suppose a burst increase in flow size is defined as the number of packets sent by a source host jumps ten-fold from one time unit to the next, whereas a burst increase in flow spread is defined as the number of *distinct* destinations (that the source host contacts) jumps ten-fold from one time unit to the

next. On the one hand, if a source host sends 1 packet in the first time unit and then 1 million packets in the next unit, all to the same destination, then it has a burst increase in flow size, but not in flow spread. On the other hand, if a source host sends 10 packets in the first time unit to the same destination and then 10 packets in the next unit to different destinations, then it has a burst increase in flow spread, but not in flow size.

The detection of spread bursts in real time is a technically challenging problem because it requires us to estimate flow spreads at a high rate at the same time as we receive data items, whereas most existing sketches for flow spread are optimized for recording (and compressing) data items in their compact data structures [2, 4, 36, 37, 42, 47], but their spread estimation is much more expensive and not suitable for real-time operations. This paper addresses the challenge with three major contributions.

(1) We are the first to introduce the problem of detecting spread bursts and provide an efficient, real-time solution that achieves good accuracy in detecting burst increase, burst decrease and spread burst in our experiments using real network traffic traces.
(2) To support our work on spread burst detection, we design a new solution for real-time super spreader identification, which outperforms the state of the art in terms of both accuracy and processing overhead.
(3) An enabling component to our super spreader work is a new sketch design for per-flow real-time spread estimation. It adopts a novel self-adaptive data structure to improve the accuracy of spread estimation and lower the overhead in the meanwhile. It outperforms the best existing sketches for flow spread.

The rest of the paper is organized as follows: Section 2 defines burst increase, burst decrease and spread burst. Section 3 presents a self-adaptive sketch for spread estimation. Section 4 designs a new sketch for real-time super spreader identification, on top of which Section 5 introduces our solution for identifying spread bursts in real time. Section 6 presents our experimental evaluation results. Finally, Section 7 draws the conclusion.

2 BURST INCREASE, BURST DECREASE AND SPREAD BURST

A data stream is a continuous sequence of data items that often arrive at a high rate, allowing one to look at (or process) each item once before moving on to the next item without storing the previous items. Each item is a pair of $\langle f, e\rangle$, where f is a flow ID and e is a data element. All items carrying the same flow ID form a flow. The size of a flow is the number of items in the flow. The spread of a flow is the number of *distinct* items in the flow, which is the focus of this paper. We can use a counter to keep track of the size of a flow, but that is not adequate for the spread because we need a data structure to remember the elements that have been seen so that we can filter out duplicate items in the stream. Because the number of distinct elements in a large flow can be in thousands or even millions, such a data structure has to be compact and efficient to operate, but lossy due to compression of all the received items. Such a data structure is called *sketch* [8–12, 15, 28, 29, 32, 34, 38, 39], which provides an estimate $\hat{n}_f$ for the true spread n_f of flow f.

Problem Definition: We divide a data stream into epochs based on time (e.g., every 5 minutes being an epoch). Consider an arbitrary flow f. Let $n_{f,i}$ be the spread of flow f in the ith epoch, $i \geq 0$. We define a *burst increase* of flow f as it meets the following condition: Given two consecutive $(i-1)$th and ith epochs, with $i > 0$,

$$\begin{aligned} &n_{f,i} \geq \beta; \\ &n_{f,i-1} < \alpha n_{f,i} \end{aligned} \tag{1}$$

where β is a threshold value and α is a fraction, which are both user-defined. For example, suppose $\beta = 100$ and $\alpha = 0.1$. If we observe that the spread of a flow f is 10 in the 2nd epoch and the spread is 110 in the 3rd epoch, then there is a burst increase of flow f between these two epochs.

Similarly, we define a *burst decrease* of flow f as it meets the following condition: Given two consecutive $(i-1)$th and ith epochs, with $i > 0$,

$$\begin{aligned} &n_{f,i-1} \geq \beta; \\ &\alpha n_{f,i-1} > n_{f,i}. \end{aligned} \tag{2}$$

We define a *spread burst* of flow f as it meets the condition below: Given a sequence of consecutive epochs from the $(j-1)$th to the ith epochs, with $j > 0$ and $1 \leq i - j < K$,

$$\begin{aligned} &\text{burst increase happens to flow } f \text{ from the } (j-1)\text{th epoch to the } j\text{th epoch;} \\ &n_{f,k} \geq \beta, \ \forall k \in [j, i); \\ &\text{burst decrease happens to flow } f \text{ from the } (i-1)\text{th epoch to the } i\text{th epoch,} \end{aligned} \tag{3}$$

where K is a user-defined parameter. With this definition, the burst of high spreads is from the jth epoch to the $(i-1)$th epoch, with a length of $(i-j)$ epochs. Note that the total length of the burst (including the spread increase and spread decrease) is $i - j + 2$, which is in the range $[1, K)$. For example, suppose $\beta = 100$, $\alpha = 0.1$ and $K = 10$. If we observe that the spread of a flow f is 10 in the 2nd epoch, 110 in the 3rd epoch, 115 in the 4th epoch, and 9 in the 5th epoch, there is a spread burst of flow f from the 2nd epoch to the 5th epoch, with a total length of 4. The burst of high spreads is from the 3rd to the 4th epoch with length 2.

Recently, burst detection has gained interest in the research community [23, 40, 45]. However, the prior work only considers bursts in terms of flow size (i.e., number of items in the flow), an easier problem than the detection of burst increase and burst decrease in terms of flow spread (i.e., number of distinct items in the flow), which has not been studied before, to the best of our knowledge.

Real-time Challenge: We are interested in real-time detection, allowing real-time reaction to the underlying issue such as an Internet worm outbreak or a denial-of-service attack. That requires us to have an updated spread estimate $\hat{n}_f$ each time after we process a data item $\langle f, e \rangle$.

Most existing sketches for flow spread are optimized for online recording of the items in their compact data structures [2, 4, 36, 37, 42, 47], while providing spread estimate at the end of an epoch offline. The online recording has to be done in real time at the arrival rate of the items, but spread estimation can be done later. Hence, existing sketches often make the tradeoff in simplifying the recording operation, while making spread estimation much more complex, which cannot be done at a per-item level in real time.

Our challenge is to design a new spread sketch that minimizes the spread estimation overhead for real-time operation, while not increasing the per-item recording overhead, in the meantime increasing the accuracy of spread estimation. Moreover, we need a new design to identify the flows whose spreads are beyond a threshold, also called *super spreader identification*, which is needed by (1), (2) and (3). Again, the existing work either cannot identify such flows in real time [2, 4] or is less accurate in doing so [24]. Our new design needs to identify super spreaders in real time as data items are continuously processed and do so with an accuracy better than the state of the art.

When we use the estimated spread to check the conditions of (1), (2) and (3), it may result in false positive (in which a burst is mistakenly reported) or false negative (in which a true burst is not reported). Reducing false positive and false negative depends on the accuracy of real-time spread estimation by our new designs.

3 SELF-ADAPTIVE SKETCH

In order to support timely burst detection, we need an efficient solution to the problem of real-time super spreader identification, which in turn requires an efficient and accurate solution for real-time flow spread estimation. In this section, we introduce a new self-adaptive sketch design, which significantly outperforms the state of the art in spread estimation for a single flow.

3.1 Existing Sketches for Single-flow Spread Estimation

To measure the spread of a single flow, most prior work was based on bitmaps [9, 10, 28, 29, 34], FM (Flajolet-Martin) sketches [12], LogLog sketch [8] or HLL (HyperLogLog) sketches [11, 15, 25, 32, 38, 39]. Among them, HLL sketches perform the best, with the largest estimation ranges and the best overall estimation accuracy.

The data structure of HLL [11] 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, such that the probability of $G(e) = i$ is $\frac{1}{2^i}$, $i \geq 1$. 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

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

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 [11, 15] for α_m under other values of m. With 5-bit registers, HLL can estimate flow spread up to many billions (specifically $\alpha_m \cdot mcdot 2^{31}$), with a relative standard error of $\frac{1.04}{\sqrt{m}}$.

The state of the art in HLL sketches includes an improved estimation approach for small flows in [15], denoted as HLL++, and a Markov-chain-based design in [25], referred to as Streamed HLL. Other variants of HLL include (1) using geometric hashes whose probability of $G(e) = i$ being a^i with a base a other than $\frac{1}{2}$ [32], which achieves modestly better accuracy in spread estimation than [15] in some range of flow spread, but is less accurate in other ranges, or (2) using register distribution with the maximum likelihood method for spread estimation [38, 39], which achieves modestly better accuracy than [15], but incurs heavy computation overhead, making it not suitable for supporting real-time flow spread queries. Overall, the stream HLL [25] achieves the best computational efficiency (especially in query overhead) and the best accuracy as well. It reduces the average relative estimation error by 22.1% over HLL++ [15] in our experiments (Section 6.2), using real Internet traffic traces.

Below we introduce a new self-adaptive sketch design, which further reduces the average relative estimation error by 36.4% over the Streamed HLL.

3.2 Self-Adaptive Sketch (SAS)

The estimation accuracy of HLL sketches is controlled by the number of registers m. Given a fixed amount of memory, there is a tradeoff between the number of registers (estimation accuracy) and the size of the registers (estimation range). To measure a flow of large spread, we should keep 5 bits per register for a large range. To measure a flow of medium or small spread, we do not need 5 bits for each register (where the higher-order bits would be mostly unused anyway); we may use 2 bits per register such that we have more registers for better accuracy. The problem is that we do not know whether the flow's spread will be large or small beforehand; that is what we want to measure. Our idea of a self-adaptive sketch design, referred to as SAS, is to begin with 2-bit

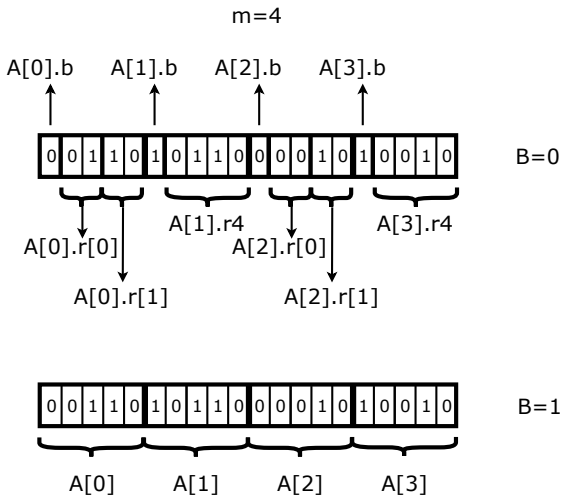

Fig. 1. Illustration of Self-Adaptive Sketch (SAS) with $m = 4$, i.e., four units of five bits each for A. The interpretation of the units in A depends on the value of the sketchwide indicator B and the local indicator of each unit (its first bit). When $B = 0$, each 5-bit unit will be interpreted either as two 2-bit registers or as one 4-bit register. For unit $A[0]$, because its local indicator $A[0].b$ is 0, the remaining four bits are interpreted as two registers, $A[0].r[0] = 01_2$ and $A[0].r[1] = 10_2$. For unit $A[1]$, because $A[1].b = 1$, the remaining four bits are interpreted as a single 4-bit register, $A[1].r4 = 0110_2$. Notice that the last four bits in both $A[0]$ and $A[1]$ are 0110_2, but they are interpreted differently, depending on the local indicator. When $B = 1$, each 5-bit unit will be interpreted as a single 5-bit register. For example, $A[0] = 00110_2$ and $A[1] = 10110_2$.

registers and merge 2 registers into a 4-bit register if one of them overflows. If any 4-bit register overflows (which indicates a large-spread flow), we convert all registers to five bits. We inherit five-bit registers from HLL, which measures flow spreads in the order of $O(2^{31})$. If one would use six-bit registers, it could measure flow spreads in the order of $O(2^{63})$, but that would be unnecessary for most practical applications.

Data Structure: SAS consists of (1) an array A of m five-bit *units*, denoted as $A[i]$, $0 \le i < m$, which are all initialized to zeros, (2) a one-bit *sketchwide indicator* B, initialized to zero, and (3) a probability variable P, initialized to one. Let N be a spread estimate of a single flow, initialized to zero.

When $B = 0$, each unit $A[i]$ contains either two 2-bit registers or one 4-bit register, depending on the value of the register's local indicator (which will be introduced shortly). But after B is set to one, all units $A[i]$ will be interpreted as five-bit registers.

We focus on explaining the case of $B = 0$. Let's first define some notations. Consider an arbitrary unit $A[i]$, $0 \le i < m$. Its first bit is a *local indicator*, denoted as $A[i].b$, which is initially zero. Its second and third bits are denoted as $A[i].r[0]$, which can be used as a 2-bit register. Similarly, its fourth and fifth bits are denoted as $A[i].r[1]$, another 2-bit register. We may also combine $A[i].r[0]$ and $A[i].r[1]$ into a four-bit register, denoted as $A[i].r4$. We interpret unit $A[i]$ as follows: When $A[i].b = 0$, we interpret the other four bits in $A[i]$ as two registers, $A[i].r[0]$ and $A[i].r[1]$; when $A[i].b = 1$, we interpret the other four bits in $A[i]$ as one register $A[i].r4$. Refer to Fig. 1 for an illustration.

Data Item Recording: For any arrival data item $\langle f, e \rangle$, we record the item in SAS by Algorithm 4 in Appendix A. We hash the item to unit $A[h(f, e)]$. If $B = 1$, this is a five-bit register where the

item will be recorded. If $B = 0$ and $A[h(f,e)].b = 1$, the item will be recorded by the four-bit register $A[h(f,e)].r4$ instead. If $B = 0$ and $A[h(f,e)].b = 0$, we need another hash $h'(f,e) \in \{0,1\}$ to further map the item to $A[h(f,e)].r[h'(f,e)]$, which is a two-bit register. In all the above cases, we compute a geometric hash $G(f,e)$, and the mapped register will be updated to $G(f,e)$ only if $G(f,e)$ is larger than the current register value. Hence, this update is probabilistic. We maintain a variable P, which at all time equals the probability for a new arrival item to cause an *update* to A, which will be formally stated in a theorem and proved in Appendix B. P is the sum of the update probabilities over all registers. Let $p(a)$ be the probability that a register $a \in A$ is updated, where the array A can be interpreted as a set of registers. $P = \sum_{a \in A} p(a)$, where $p(a)$ is the product of the probability for a new item $\langle f,e \rangle$ to be hashed to a and the probability of $G(f,e) > a$, which causes update. Initially, because all registers are zeros and $G(f,e) \geq 1$, we must have $P = 1$. With an update of a register a to value $G(f,e)$, we need to update P because $p(a)$ changes. The expected number of new items to cause an update is $\frac{1}{P}$. So when an update event happens, we increase the spread estimate N by $\frac{1}{P}$.

We now explain the details on how to update the register and the value of P. We must handle register overflow. First, consider the cases when $B = 0$.

(1) If $A[h(f,e)].b = 0$, $G(f,e) > A[h(f,e)].r[h'(f,e)]$ and $G(f,e) \leq 3$, then we need to update the 2-bit register $A[h(f,e)].r[h'(f,e)]$ to $G(f,e)$, where $A[h(f,e)].r[h'(f,e)]$ is the register that the data item $\langle f,e \rangle$ is mapped to. It will not cause overflow because $G(f,e) \leq 3$. We also need to update P because $p(A[h(f,e)].r[h'(f,e)])$ changes. The probability of hashing to unit $A[h(f,e)]$ is $\frac{1}{m}$. There are two 2-bit registers in it. So the probability of hashing to $A[h(f,e)].r[h'(f,e)]$ is $\frac{1}{2m}$. The probability for a geometric hash $G(f,e)$ to be greater than $A[h(f,e)].r[h'(f,e)]$ is $2^{-A[h(f,e)].r[h'(f,e)]}$. Hence $p(A[h(f,e)].r[h'(f,e)])$ is equal to $\frac{2^{-A[h(f,e)].r[h'(f,e)]}}{2m}$ before register update. With the update of $A[h(f,e)].r[h'(f,e)]$ to $2^{-G(f,e)}$, this probability becomes $2^{-G(f,e)}$ for future data items.

$$\begin{aligned} &P := P - \frac{2^{-A[h(f,e)].r[h'(f,e)]}}{2m} + \frac{2^{-G(f,e)}}{2m}; \\ &A[h(f,e)].r[h'(f,e)] := G(f,e), \end{aligned} \tag{4}$$

where ":=" is the assignment operator.

(2) If $A[h(f,e)].b = 0$, $G(f,e) > A[h(f,e)].r[h'(f,e)]$ and $3 < G(f,e) \leq 15$, then there will be overflow if we set the 2-bit register $A[h(f,e)].r[h'(f,e)]$ to $G(f,e)$. We need to combine two 2-bit registers to one 4-bit register $A[h(f,e)].r4$ by setting the local indicator $A[h(f,e)].b$. Recall that P is the sum of the update probabilities over all registers. Because we combine two registers into one, we must subtract the update probabilities of the two registers, $A[h(f,e)].r[0]$ and $A[h(f,e)].r[1]$, from P and then add the update probability of the new register $A[h(f,e)].r4$, whose value is $G(f,e)$. Note that the probability of hashing to $A[h(f,e)].r[0]$ or $A[h(f,e)].r[1]$ is $\frac{1}{2m}$, and the probability of hashing to $A[h(f,e)].r4$ to $\frac{1}{m}$

$$\begin{aligned} &P := P - \frac{2^{-A[h(f,e)].r[0]} + 2^{-A[h(f,e)].r[1]}}{2m} + \frac{2^{-G(f,e)}}{m}; \\ &A[h(f,e)].r4 := G(f,e); \\ &A[h(f,e)].b := 1. \end{aligned} \tag{5}$$

(3) If $A[h(f,e)].b = 1$, $G(f,e) > A[h(f,e)].r4$ and $G(f,e) \leq 15$, then we need to update the 4-bit register $A[h(f,e)].r4$ to $G(f,e)$. Note that with $A[h(f,e)].b = 1$, $A[h(f,e)]$ is interpreted as having a 4-bit register $A[h(f,e)].r4$. The probability of hashing to $A[h(f,e)]$ is $\frac{1}{m}$, and the

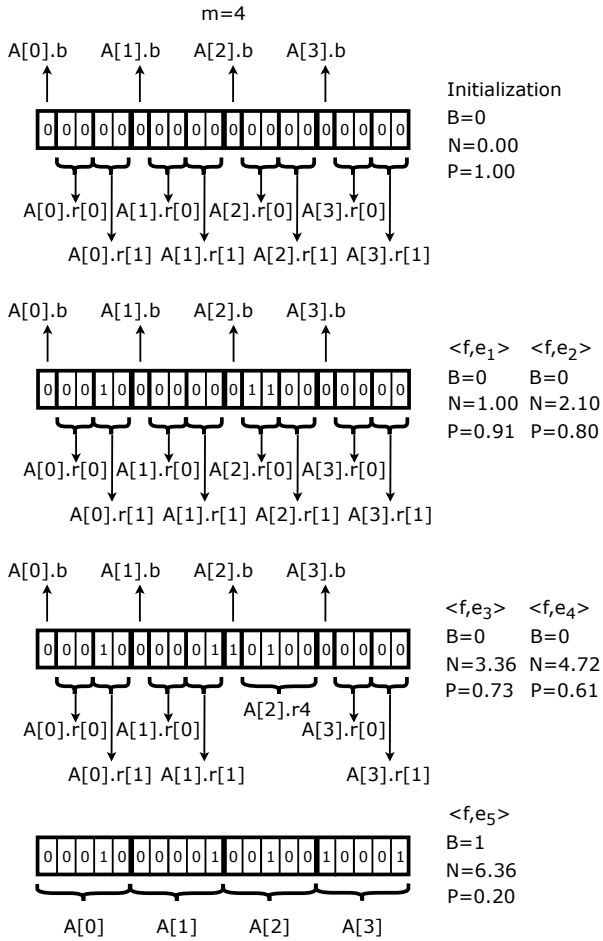

Fig. 2. Illustration on how the registers in SAS evolve as data items are recorded. There are four rows of register arrays in the figure. **Initialization** is shown in the first row, where A has four units of 5 bits each, with $B = 0$, $N = 0$ and $P = 1$. **When** $\langle f, e_1\rangle$ **arrives**, suppose that it is hashed to $A[0].r[1]$ and $G(f, e_1) = 2$, thus $A[0].r[1] = 2 = 10_2$, $N = 1.00$, and $P = 0.91$, according to our algorithm, as shown in the second row. **When** $\langle f, e_2\rangle$ **arrives**, suppose that it is hashed to $A[2].r[0]$ and $G(f, e_2) = 3$, thus $A[2].r[0] = 3 = 11_2$, N is increased by $\frac{1}{P}$ to 2.10, and then P is changed to 0.80, also shown in the second row. **When** $\langle f, e_3\rangle$ **arrives**, suppose that it is hashed to $A[1].r[1]$ and $G(f, e_3) = 1$, thus $A[1].r[1] = 1 = 01_2 = 1$, $N = 3.36$, and $P = 0.73$, as shown in the third row. **When** $\langle f, e_4\rangle$ **arrives**, suppose that it is hashed into $A[2].r[0]$ and $G(f, e_4) = 4$, which is larger than the current value of $A[2].r[0]$. Because $A[2].r[0]$ cannot store $G(f, e_4)$ without overflow, we combine $A[2].r[0]$ and $A[2].r[1]$ into $A[2].r4$ by setting $A[2].b = 1$ and $A[2].r4 = 4 = 0100_2$. Then, $N = 4.72$ and $P = 0.61$. Finally, **when** $\langle f, e_5\rangle$ **arrives**, suppose that it is hashed to $A[3].r[0]$ and $G(f, e_5) = 17$, which will overflow $A[3].r[0]$. We need a 5-bit register to store $G(f, e_5)$. We turn all units to 5-bit registers and let $B = 1$. We set $A[3].r[0] = 17 = 10001_2$, $N = 6.36$ and $P = 0.20$, as shown in the last row.

probability for this register to be updated by a new item changes from $2^{-A[h(f,e)]}$ to $2^{-G(f,e)}$ as the register value is changed.

$$\begin{aligned} &P := P - \frac{2^{-A[h(f,e)].r4}}{m} + \frac{2^{-G(f,e)}}{m}; \\ &A[h(f,e)].r4 := G(f,e). \end{aligned} \tag{6}$$

(4) If $G(f,e) > 15$, there will be overflow because with $B = 0$, all registers are 2-bit or 4-bit long and none can hold $G(f,e)$. We set $B = 1$ and combine the registers in each unit $A[i]$, $0 \le i < m$, to a single 5-bit register by taking the maximum value of the registers in the unit. Since the values of many registers may have changed, we recompute P by summing the new update probabilities over all 5-bit registers:

$$P := \sum_{i=0}^{m} \frac{2^{-A[i]}}{m}. \tag{7}$$

We then update the value of $A[h(f,e)]$ to $G(f,e)$ unless $G(f,e) > 31$, in which case $A[h(f,e)] = 31$. With this update, we need to change the value of P accordingly.

$$P := \begin{cases} P - \frac{2^{-A[h(f,e)]}}{m} + \frac{2^{-G(f,e)}}{m}, & \text{if } G(f,e) < 31; \\ P - \frac{2^{-A[h(f,e)]}}{m}, & \text{if } G(f,e) \ge 31. \end{cases} \tag{8}$$

The correctness of the formulas for updating P is stated in Theorem 1, which is proven in Appendix B.

Next, consider the case when $B = 1$. If $G(f,e) > A[h(f,e)]$ and $A[h(f,e)] < 31$, then we update the value of P the same way as in (8) and let $A[h(f,e)] := \min\{G(f,e), 31\}$.

Theorem 1. *At any time the value of P in SAS is equal to the probability for the next arrival data item to update the value of a register.*

The proof of the above theorem can be found in Appendix B. Figure 2 gives an example of how data items are recorded by SAS.

Estimation Accuracy: The standard error of spread estimate by HLL [11] is $\frac{\beta_m}{\sqrt{m}} + \delta_2(n)$, where n is the real flow spread, m is the number of registers, $|\delta_2(n)| < 5 \cdot 10^{-4}$ for $m \ge 16$ as $n \to \infty$, and β_m is a function of m: $\beta_{16} = 1.106$, $\beta_{32} = 1.070$, $\beta_{64} = 1.054$, $\beta_{128} = 1.046$, $\beta_\infty = \sqrt{3\log(2) - 1} = 1.03896$. The standard error decreases when m increases.

This result applies to SAS, with its standard error being $\frac{\beta_{m'}}{\sqrt{m'}} + \delta_2(n)$, where m' is the total number of variable-sized registers in A. In the worst case, SAS becomes HLL when $B = 1$, i.e., all registers are converted to five bits long and thus $m' = m$. But its performance is better than HLL when $B = 0$ and $m' > m$.

Spread Estimation: The variable N provides an up-to-date estimate of the flow's spread at any time. There is essentially no query overhead.

4 REAL-TIME SUPER SPREADER IDENTIFICATION

We now consider a data stream of numerous flows. We divide the time into epochs. To support timely burst detection (in the next section), we design a real-time super spreader identifier that processes the data stream in each epoch and implements the following two functions:

(1) *real-time super spreader identification.* It identifies in real time the flows whose spreads are greater than a user-specified threshold in an epoch, and returns the IDs and the estimated spreads of those flows.
(2) *per-flow spread estimation.* It can provide an estimate for the spread of any given flow at any time.

4.1 Existing Work on Super Spreader Identification

The prior work on super spreader identification can be broadly categorized as either sampling-based [2, 5, 17, 27, 44] or sketch-based [7, 19, 20, 24, 33].

The sampling-based solutions [5, 17, 27, 44] monitor only a portion of the flows by sampling. They only consider the data items whose hash values are smaller than a pre-specified threshold, which controls the sampling probability. They store the flow IDs of the sampled items and estimate the spread of each flow based on its sampled items. They can then identify the super spreaders from the sampled flows. Note that the super spreaders are more likely to be sampled while most small flows will be filtered out. The best sampling-based solution is a recently published work called AROMA [2, 4]. AROMA does not support real-time identification of super spreaders. That would require spread estimation at a per-item basis, which AROMA does not support, because of high overhead. None of the sampling-based solutions support per-flow spread estimation. Nonetheless, we will compare with AROMA on the accuracy of super-spreader identification.

The sketch-based solutions summarize the information of all items in sketches. Most of them [7, 19, 20, 33] are designed to recover the super spreaders offline. They usually suffer from high overhead of recovering super spreaders. The most recent work, called SpreadSketch [24], stresses the importance of fast detection. It outperforms the prior sketch-based solutions in terms of detection accuracy and detection overhead. Yet its processing overhead is still significant as our experiments will demonstrate. Moreover, its accuracy in super spreader identification is much worse than AROMA [2, 4].

4.2 New Design for Real-time Super spreader Identification

We introduce a new design for Real-time Super spreader Identifier, referred to as RSI, which adopts SAS as a building block for its efficiency and accuracy in spread estimation. It detects super spreaders in real time and provides a spread estimation for any flow at any time. Our experimental results show that its processing overhead is much smaller than SpreadSketch [24], while its overall accuracy is better than AROMA [2, 4], under the same memory usage.

Data Structure: The data structure of RSI consists of (1) a hash table T that stores a subset of selected flows, with each table entry having a flow ID field and an estimated spread field — we will report a flow as a candidate super spreader if its estimated spread reaches a user-specified threshold t; (2) an array C of n SAS sketches, each of which operates independently — we do not use each SAS sketch for a single flow, but use all SAS sketches together for spread estimation of numerous flows in the data stream; and (3) a variant of conservative counter update sketch (CU) [14], denoted as U, which enables per-flow spread estimation. The value of n is determined based on the amount of memory allocated for RSI.

Notations: The ith SAS sketch in C is denoted as $C[i]$, for $0 \leq i < n$. Its array of 5-bit units is denoted as $C[i].A$, its indicator is denoted as $C[i].B$, and its probability variable is denoted as $C[i].P$. We do not need the variable N for spread estimation, which is now the job of U. The jth 5-bit unit in $C[i].A$ is denoted as $C[i].A[j]$, $0 \leq j < m$. If $C[i].B = 0$ and $C[i].A[j].b = 0$, we interpret the remaining four bits of the unit as two registers, $C[i].A[j].r[0]$ and $C[i].A[j].r[1]$, each having two bits. If $C[i].B = 0$ and $C[i].A[j].b = 1$, we interpret the remaining four bits of the unit as a single 4-bit register, $C[i].A[j].r4$. If $C[i].B = 1$, any unit $C[i].A[j]$ is considered as a 5-bit register. The CU sketch, i.e., U, is a two-dimensional counter array with d rows and w columns. Each counter has $\lceil \log(t+1) \rceil$ bits. The jth counter of the ith row in U is denoted as $U[i][j]$, where $0 \leq i < d$, $0 \leq j < w$. The value of d is typically set to three or four. The value of w is determined based on the amount of memory allocated.

Flow-SAS Mapping: We consider the case that the number of flows in a large data stream is far greater than the number n of SAS sketches in C, so we cannot allocate one SAS sketch per flow. The flows have to share the SAS sketches. To prevent a very large flow from turning all SAS sketches

Algorithm 1: Record a data stream in RSI

Input: $\langle f, e\rangle$, C, U

```
r = h''(f, e)
p = C[H_r(f)].P
Record ⟨f, e⟩ in C[H_r(f)] with return value b
if b = true then
    v := min{U[i][H'_i(f)] | 0 ≤ i < d}
    for i ∈ [0, d) do
        if U[i][H'_i(f)] < v + 1/p then
            U[i][H'_i(f)] := v + 1/p
```

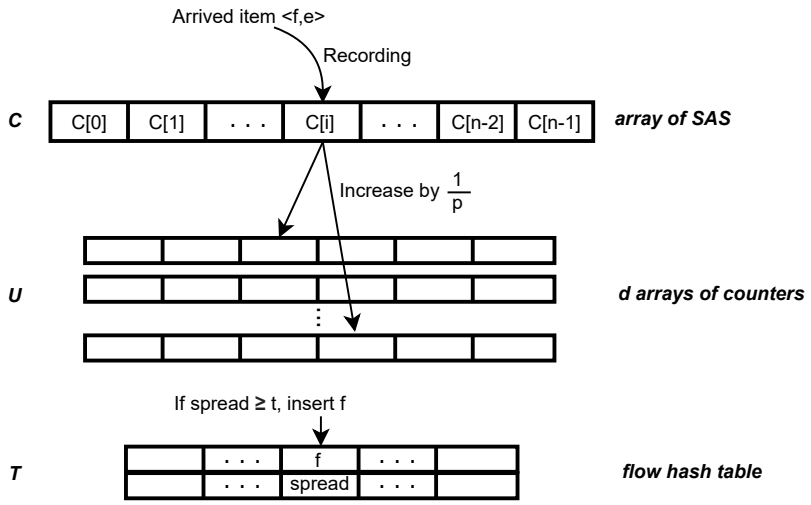

Fig. 3. Illustration on the operations of RSI for super spreader identification. Each arrival item $\langle f, e\rangle$ is recorded by one SAS selected from C. If it causes a register update, we increase the estimated spread of flow f in U. If the estimated spread reaches the threshold, we will insert flow f and its estimated spread in T. For all subsequent items of flow f, after they are recorded and if they cause register updates in C, we need to update the flow's entry in T for its increased spread estimate.

into 5-bit registers, we pseudo-randomly map each flow to k SAS sketches through hashing, where $k \ll n$, and record its data items only in these k sketches, so that any large flow will not severely impact any small flow, unless their k sketches completely overlap.

More specifically, each flow f is mapped to $C[H_i(f)]$, $0 \le i < k$, where $H_i(.)$ is a hash function whose range is $[0, n)$.

Flow-CU Mapping: Based on the standard operation of CU [14], each flow f is mapped to d counters, $U[i][H'_i(f)]$, $0 \le i < d$, where $H'_i(.)$ is a hash function whose range is $[0, w)$.

Data Item Recording: For any arrival data item $\langle f, e\rangle$, we record the item in a selected SAS sketch in C by Algorithm 1. First, we use a hash value, $r = h''(f, e) \in [0, k-1]$, to select an SAS, $C[H_r(f)]$. Then we record the item in $C[H_r(f)]$ by Algorithm 4, which can be found in Appendix A. By Theorem 1, the probability for the item to cause a register update is $p = C[H_r(f)].P$. Hence, we should increase flow f's spread estimate by $\frac{1}{p}$. This is done by increasing some of the d counters in

U that f is mapped to as follows: Let v be the smallest value of the d counters. For the counters that are smaller than $v + \frac{1}{p}$, we increase them to $v + \frac{1}{p}$. For the counters that are equal to or greater than $v + \frac{1}{p}$, we keep them unchanged.

For each arrival data item, besides the overhead of recording it in a SAS sketch (which has been discussed in Section 3.2) the additional overhead includes one hash $h''(f, e)$ to select the SAS sketch and d hashes for updating U. Because d is typically three or four, we may take bits from one hash computation $H^*(f)$, $\log_2 w$ bits at a time to replace $H'_i(f)$, $0 \le i < d$, if the number of output bits in $H^*(f)$ is at least $d \log_2 w$. That reduces d hashes to one hash for updating U.

Spread Estimation: Because flows share counters in U, as they increase their counters in recording, they introduce inter-flow noise to other flows that share the same counters. To query for the spread of any flow f, we return the minimum value of the d counters, $U[i][H'_i(f)]$, $0 \le i < d$, which carries the smallest noise.

Real-time Super Spreader Identification: For each arrival data item $\langle f, e \rangle$, after recording it in C and increasing the d counters, we have the real-time estimate of the flow's spread by taking the minimum of the d counters. If it reaches the threshold t, we insert f into the hash table T as a super spreader if it is not already there, and we set its estimated spread. If f is already in T, we increase its spread estimate in the table by $\frac{1}{p}$. Figure 3 illustrates the operations of RSI for super spreader identification.

5 REAL-TIME BURST DETECTION (RBD)

By the definitions in Section 2, we may detect a burst increase in real time; as the arrival of a data item in flow f pushes the spread estimate higher to meet the condition of burst increase in (1), we are able to detect it right away if we have the updated spread estimate. However, we can only detect a burst decrease at the end of an epoch because the condition of burst decrease in (2) holds at the beginning of each epoch but may be violated at any time as the flow's spread increases. Only if (2) holds at the end of the epoch, we can be sure that we have a burst decrease. Because a spread burst consists of a burst increase and then a burst decrease, it is also true that we can only detect a spread burst at the end of the epoch.

Consider a device processing a continuous high-rate data streaming, such as the network processor chip in a router processing an incoming packet stream at tens of millions of packets per second. Suppose that real-time spread burst detection, denoted as RBD, is one of the tasks by a measurement module implemented in cache memory (such as SRAM) for high speed. Suppose that RBD starts from the 0th epoch and processes the data stream, epoch by epoch, to detect all burst increases, burst decreases, and spread bursts.

Data Structure: At the ith epoch, $i > 0$, the data structure of RBD consists of (1) RSI's data structure for recording the data items in the current epoch, denoted as C_i, U_i, and T_i, with the threshold $t = \beta$, (2) RSI's data structure from the previous $(i-1)th$ epoch, denoted as U_{i-1}, and T_{i-1}, and (3) a hash table F storing the flows that had a burst increase less than K epochs ago and kept their spreads above the threshold in each epoch since. These flows are candidates for spread burst detection.

At the end of the ith epoch, we will send the content of C_i, U_i, T_i and F to an offline server, which keeps all measurement results for long-term storage. We delete C_{i-1}, U_{i-1}, and T_{i-1} locally, keep C_i, U_i and T_i for one more epoch, always keep F, and create C_{i+1}, U_{i+1} and T_{i+1} to start the $(i+1)$th epoch.

Per-item Operation and Real-time Detection of Burst Increase: For any arrival data item $\langle f, e \rangle$, we record it by Algorithm 1 in C_i, U_i and T_i. After recording, if the flow f is in T_i and its

Algorithm 2: Real-time detection of burst increase by RBD

Input: $\langle f, e\rangle$, C_i, U_i, T_i, U_{i-1}, T_{i-1}, F

Record $\langle f, e\rangle$ in C_i, U_i and T_i by Algorithm 1
if $f \in T_i$ *and* $\hat{n}_{f,i}$ *is increased* **then**
    Look up in T_{i-1} and U_{i-1} for $\hat{n}_{f,i-1}$
    if $\hat{n}_{f,i-1} < \alpha\hat{n}_{f,i}$ **then**
        Report a burst increase of flow f
        Insert f and i into F

Algorithm 3: Detection of burst decrease and spread burst by RBD

Input: $\langle f, e\rangle$, C_i, U_i, T_i, U_{i-1}, T_{i-1}, F

for *each flow* $f \in T_{i-1}$ **do**
    Look up in T_i and U_i for $\hat{n}_{f,i}$
    if $\alpha\hat{n}_{f,i-1} > \hat{n}_{f,i}$ **then**
        Report a burst decrease of flow f
        if $f \in F$ **then**
            Report a spread burst of flow f
for *each flow* $f \in F$ **do**
    Look up in T_i and U_i for $\hat{n}_{f,i}$
    Flow f was inserted to F during the jth epoch
    if $\hat{n}_{f,i} < \beta$ *or* $i - j = K$ **then**
        Remove f from F

estimated spread in the current epoch, denoted as $\hat{n}_{f,i}$, is just updated, since $\hat{n}_{f,i}$ must be greater than the threshold β, we need to check the condition for burst increase. To do so, we look up in T_{i-1} for the estimated spread of flow f in the previous epoch, denoted as $\hat{n}_{f,i-1}$. If f is not in T_{i-1}, we compute $\hat{n}_{f,i-1}$ from U_{i-1}; see Section 4.2. If $\hat{n}_{f,i-1} < \alpha\hat{n}_{f,i}$, we report a burst increase for flow f, and if f is not already in F, we insert flow f, together with i, into F. The pseudo code can be found in Algorithm 2.

Detection of Burst Decrease and Spread Burst: At the end of the ith epoch, for each flow f in T_{i-1}, we know that its estimated spread $\hat{n}_{f,i-1}$ in the $(i-1)$th epoch must be greater than the threshold β. We want to check the condition for burst decrease. To do so, we look up in T_i for the flow's estimated spread in the ith epoch, $\hat{n}_{f,i}$. If f is not in T_i, we compute $\hat{n}_{f,i}$ from U_i. If $\alpha\hat{n}_{f,i-1} > \hat{n}_{f,i}$, we report a burst decrease for flow f. If f is in F, then we report a spread burst.

For each flow f in F, if $\hat{n}_{f,i} < \beta$, we remove it from F. Suppose flow f was inserted into F in the jth epoch, if $i - j = K$, we also remove f from F. Refer to Algorithm 3 for the end-of-epoch operations.

6 EXPERIMENTAL EVALUATION

6.1 Experimental Setup

We use four datasets for our experiments: (1) a synthetic dataset used for the evaluation of SAS, (2) 12 hours of packet stream, extracted from one of the backbone infrastructure routers on our

campus, which contains 23,595,264 packets, (3) five CAIDA traces [26], which cumulatively contain 8,194,919,166 packets, and (4) an E-commerce dataset [16], where each item is a product review record and there are 109,950,747 items in total. For all network datasets, each packet is modeled as a data item $\langle f, e\rangle$. In the second dataset above, we identify 237304 flows in the packet stream. From each packet's headers, we extract the source IP address as the flow ID f, and the combination of source port, destination IP and destination port as the element ID e. That helps a router to catch parallel TCP connections during congestion by measuring flow spread, identifying super spreaders, detecting burst increase and spread burst, as is explained in the introduction. We divide the 12 hours of packet stream into 144 epochs of 5 minutes each. As an example, the first epoch contains 173851 packets, among which 107708 are distinct. In the third dataset, we use the source address as the flow ID and the destination address as the element ID e. The five traces, gathered by five different routers in different years, each contain one hour of Internet traffic. Each of the five one-hour CAIDA traces is divided into 60 one-minute epochs. For the E-commerce dataset, we extract the product name as the flow ID, and the user name in each review as the element ID e. Detecting burst increase, burst decrease, and spread burst can help track the popularity change of products over time. The dataset contains a total of 206859 flows. It spans 61 days in the whole October and November of 2019. We divide the dataset into 61 epochs, each lasting for 24 hours.

The performance metric for evaluating SAS (Self-Adaptive Sketch) is the standard error [35], which is defined as $\sqrt{\sum_{f\in\Omega}(\frac{n_f-\hat{n}_f}{n_f})^2/(|\Omega|-1)}$, where n_f is the true spread of flow f, $\hat{n}_f$ is the estimate, and Ω is the set of all flows in a stream.

The performance metrics for evaluating RSI (Real-time Super spreader Identifier) include (1) number of true positives (TP), which are the reported flows that are truly super spreaders, (2) number of false positives (FP), which are the reported flows that are not super spreaders, (3) number of false negatives (FN), which are true super spreaders not reported, (4) F1-score given by the formula $F1 = \frac{TP}{TP+\frac{1}{2}(FP+FN)}$, which combines the impact of FP and FN with respect to TP, and (5) average time for processing a data item (such as packet). Note that a higher F1-score indicates a better performance, with F1 = 1 for the case of no FP and no FN.

The performance metrics for evaluating RBD (Real-time Burst Detection) include (1) number of true positives (TP), (2) number of false positives (FP), (3) number of false negatives (FN), and (4) F1-score.

In the next three subsections, we will evaluate SAS, RSI and RBD, respectively. For SAS, we evaluate its estimation accuracy with respect to the state-of-the-art sketches for spread measurement and we keep the memory a constant of 640 bits for all sketches and the register size (unit size in our case) to 5 bits for all sketches. For RSI, we evaluate its performance on super spreader identification with respect to the state-of-the-art solutions. It inherits the aforementioned parameters of SAS. Let M be the total memory and $M(U)$ be the memory allocated for its array U. In the experiments for RSI, we set M to 2Mb, and vary the memory distribution $M(U)/M$ and the number of hash functions d. We will use the parameter configuration that performs best in the subsequent evaluation, including the experiments for RBD.

6.2 Evaluation of SAS for Spread Estimation

We compare SAS with the state of the art, Streamed HLL [25], as well as the original HLL [11] and its improvement HLL++ [15] on their estimation accuracy. As recommended by [15], the memory is set to 640 bits for each sketch. That is, HLL++ has 128 5-bit HLL registers; Streamed HLL has 115 5-bit registers, two floats for storing the estimate and the probability respectively; SAS has 115 5-bit units, a 1-bit indicator, two floats for storing the estimate and P respectively.

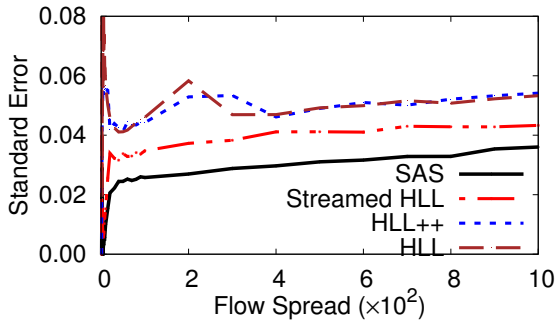

Fig. 4. Standard error comparison among SAS, Streamed HLL, HLL++ and HLL.

We first use artificially generated flows with spreads ranging from 1 to 1000 (which covers most flows in our packet stream to be used next). For each spread value, we generate 5000 flows of that spread and run the four sketches in turn to provide a spread estimate before computing the standard error. Fig. 4 presents the standard error comparison. Due to the self-adjusting design, SAS outperforms HLL++ and Streamed HLL significantly. For example, SAS reduces the standard error by 41.6% and 26.1%, compared to HLL++ and Streamed HLL, respectively, when the flow spread is 100. This accuracy improvement is important for the performance of RSI and RBD that are built on top of it.

Next, we use our campus packet stream to evaluate. For each flow in each epoch, we run the three sketches to each produce a spread estimate and calculate the standard error. The standard error of all flows in all epochs (or flows with spread greater than or equal to 20) is presented in Table 1. SAS reduces the standard error by 22.4% compared to Streamed HLL, which in turn reduces the standard error by 16.2% over HLL++. From Fig. 4, we see that larger flows (with spreads no smaller than 20) have larger standard errors. If we only consider these flows, the error reduction by SAS over Streamed HLL is 20.4%, and the reduction by Streamed HLL over HLL++ is 16.2%.

Table 1. Standard errors for SAS, Streamed HLL and HLL++, using the campus packet stream. SAS reduces standard error of all flows by 22.4% over Streamed HLL.

	All flows	Flows with spread ≥ 20
SAS	0.00194	0.0301
Streamed HLL	0.00250	0.0378
HLL++	0.00298	0.0451

6.3 Evaluation of RSI for Super Spreader Identification

The experimental configuration is described as follows: We use the campus packet stream. In each epoch, the flows whose spreads are 100 or greater are super spreaders. Each counter in U is 7 bits long, and we set t to 90. The value of t controls the tradeoff between FP and FN; a smaller threshold t will increase FP but reduce FN (which is often more important). Let M be the total memory and $M(U)$ be the memory allocated for U. Hence, the memory allocated for C is about $M - M(U)$ as the hash table T for super spreaders is typically small. We use the ratio $M(U)/M$ to characterize the memory distribution in RSI. When we increase an integer in U by a real number $1/p$, our actual implementation is to increase the counter by $\lceil 1/p \rceil$ with a probability of $\frac{1/p}{\lceil 1/p \rceil}$.

We first evaluate the impact of d (number of counter arrays in U) and memory distribution on the performance of RSI. Table 2 presents the average number of FPs, the average number of FN, and the F1-scores with respect to d, under $M(U)/M = 0.5$ and $M = 2$Mb. The total number of true super spreaders across all epochs is 23962, as shown in the second column under Ground Truth. Super

Table 2. Accuracy of RSI in FP, FN and F1-score with respect to d.

d	Ground Truth	Reported	FP	FN	F1-score
1	23962	29184	5403	181	0.895
2	23962	26513	2689	138	0.944
3	23962	26428	2638	172	0.944
4	23962	26441	2633	154	0.945
5	23962	26469	2661	154	0.944
6	23962	26412	2627	177	0.944
7	23962	26481	2689	170	0.943

Table 3. Accuracy of RSI in FP, FN and F1-score with respect to $M(U)/M$.

$M(U)/M$	Ground Truth	Reported	FP	FN	F1-score
0.4	23962	26426	2645	181	0.944
0.5	23962	26441	2633	154	0.945
0.6	23962	26517	2696	141	0.944
0.7	23962	26411	2634	185	0.944

spreaders are counted independently in each epoch. If a flow's spread is 100 or greater in multiple epochs, we try to detect it in all those epochs. The third through sixth columns present the number of reported super spreaders, the number of false positives, the number of false negatives, and the F1-score, respectively.

As we increase d from the minimum value 1, there are two opposing factors that affect the accuracy of spread estimation by U, which in turn affects the accuracy of super spreader detection. On the one hand, with a large value of d, the min operation inherited from CU [14] helps reduce the error in spread estimation; on the other hand, as data items are recorded for up to d times, it increases the inter-flow noise in U and thus increases the error in spread estimation. With these two opposing factors, as we can see in Table 2, the F1-score increases first, peaks at $d = 4$, and decreases after that. Hence, we will set $d = 4$ in the remaining experiments, which agrees with the choice of d in existing papers that adopt CU [3, 21, 46].

Table 3 presents FP, FN and F1-score by varying $M(U)/M$ from 0.4 to 0.7, with $d = 4$ and $M =$ 2Mb. The best F1-score is achieved when $M(U)/M = 0.5$, which means that the memory is about evenly distributed between C and U.

Next, we compare RSI with the state of the art on super spreader identification, i.e., AROMA [2] and SpreadSketch [24], in terms of accuracy in super spreader identification. For RSI, we set $d = 4$ and $M(U)/M = 0.5$. The memory M allocated to each sketch is 2Mb. The parameter settings of AROMA and SpreadSketch follow those in the original papers. We use all the epochs of our packet stream. Table 4 shows experimental results in FP, FN, and F1-score. RSI has the smallest number of false positives, much fewer than AROMA, which is in turn much better than SpreadSketch. The table also shows that all three sketches have very few false negatives, comparing with the number of true super spreaders in the second column. For example, RSI's FN only accounts for 0.6% of super spreaders. The reason that SpreadSketch has no false negatives but many false positives is because it usually overestimates flow spread. In terms of the performance in F1-score results, RSI is the best, maintaining a F1-score of 0.945. In comparison, AROMA's F1-score is 0.889 and SpreadSketch's F1-score is only 0.670.

Finally, we compare the sketches in terms of per-packet processing overhead. AROMA is not evaluated because it does not support real-time super spreader identification due to its high overhead on spread estimation. RSI needs 212 ns to process a packet on average while SpreadSketch needs 4889 ns, which is 22 times larger.

Table 4. Performance of RSI, AROMA and SpreadSketch in super spreader identification.

Epoch	Ground Truth	Reported	FP	FN	F1-score
AROMA	23962	28742	5300	520	0.889
RSI	23962	26441	2633	154	0.945
SpreadSketch	23962	47620	23658	0	0.670

6.4 Evaluation of RBD for Burst Detection

Since this paper is the first that studies the spread burst detection, there is no prior work that we can compare with. We focus more on evaluating the performance of RBD under different parameter values.

We first study the performance of RBD under different memory allocations, using the campus dataset. We use the entire 144 epochs of the packet stream. If not specified otherwise, the default parameter settings are $d = 4$, $M(U)/M = 0.5$, $\beta = 100$, $\alpha = 0.1$, M =2Mb, and $K = 10$. The left half of Table 5 presents the F1-scores for burst increase, burst decrease, and spread burst detection, under the campus dataset, where the first column varies the memory allocation from 100Kb to 10000Kb (i.e., 10Mb). The experimental results show that the performance generally improves as the memory increases; small deviation is the result of statistical variance in execution. When the memory is increased from 100Kb to 500Kb, the performance improvement is significant, with the F1-score for burst increase increasing from 0.753 to 0.944, but the gain becomes negligible when the memory is increased further. This is because when C and U are small, there will be many hash collisions as we map flows to them, causing inter-flow noise. As we increase memory, collisions (thus noise) are reduced. Once collisions are already kept at a low level, further increasing memory does not offer much help.

The detailed experimental results on TP, FP, and FN for burst increase, burst decrease and spread burst detection are provided in Tables 13-15 in Appendix C. They cannot be included in the main text due to space limitation.

Table 5. F1-score of RBD for burst increase, burst decrease, and spread burst detection, w.r.t. memory allocation (Kb), under $\beta = 100$, $\alpha = 0.1$, and $K = 10$, using the campus dataset and the CAIDA-1 dataset, respectively. CAIDA-1 begins from a higher memory of 500Kb because it contains much more items than the campus dataset.

	Campus dataset			CAIDA-1 dataset		
Mem. (Kb)	Burst increase	Burst decrease	Spread burst	Burst increase	Burst decrease	Spread burst
100	0.753	0.698	0.663	-	-	-
200	0.905	0.884	0.839	-	-	-
500	0.944	0.934	0.926	0.785	0.808	0.788
1000	0.956	0.924	0.926	0.951	0.930	0.953
2000	0.953	0.932	0.928	0.944	0.953	0.957
5000	0.959	0.936	0.932	0.957	0.966	0.958
10000	0.953	0.934	0.933	0.953	0.978	0.957

In the remaining experiments, we study how the performance of RBD is affected by different parameter settings. Each time we vary one parameter while fixing the others to their default values: $d = 4$, $M(U)/M = 0.5$, $\beta = 100$, $\alpha = 0.1$, $K = 10$, and $M = 2$Mb. Using the campus dataset, Table 6 presents the performance of RBD with respect to β, which varies from 20 to 1000. When the threshold β is very small, the numbers of false positives and false negatives are large because the error in spread estimation can easily overcome the threshold. As β increases, there are fewer spread

bursts and the numbers of false positives and false negatives drop even faster, improving F1-score above 0.9 when β is 100 or more.

Table 6. Performance of RBD in spread burst detection w.r.t. β, under $\alpha = 0.1$, $K = 10$, and M = 2Mb, using the campus dataset

β	Ground truth	Reported	TP	FP	FN	F1-score
20	2905	3396	2600	796	305	0.825
50	1122	1278	1030	248	92	0.858
100	580	594	545	49	35	0.928
200	298	300	280	20	18	0.936
500	103	104	97	7	6	0.937
1000	51	48	47	1	4	0.949

Table 7 presents the performance of RBD with respect to α, which varies from 0.5 to 0.01. RBD performs well across the whole range. As we decrease α, there are fewer bursts, which is expected as it becomes more difficult to meet the condition (3), and F1-score decreases slightly.

Table 7. Performance of RBD in spread burst detection w.r.t. α, under $\beta = 100$, $K = 10$, and M = 2Mb, using the campus dataset

α	Ground truth	Reported	TP	FP	FN	F1-score
0.5	1383	1415	1331	84	52	0.951
0.2	895	913	835	78	60	0.923
0.1	580	594	545	49	35	0.928
0.05	402	406	368	38	34	0.910
0.02	279	287	260	27	19	0.918
0.01	237	232	214	18	23	0.912

Using the campus dataset, the left half of Table 8 presents the performance of RBD by varying both α and β. When $\beta \geq 100$, RBD performs very well across the whole range of α. But when β is small such as 20, the flow's spread at the low end of a burst increase (or decrease) is even smaller, proportional to α. The error in spread estimation by sketch U can overcome such a small spread, resulting in lower F1-score.

Table 8. F1-score of RBD in spread burst detection, w.r.t. to β and α, where $K = 10$, using the campus dataset (M = 2Mb) and the CAIDA-1 dataset (M = 5Mb), respectively

	Campus dataset						CAIDA-1 dataset					
β \ α	0.5	0.2	0.1	0.05	0.02	0.01	0.5	0.2	0.1	0.05	0.02	0.01
20	0.90	0.87	0.85	0.84	0.83	0.81	0.36	0.80	0.86	0.85	0.84	0.85
50	0.93	0.90	0.89	0.88	0.86	0.88	0.86	0.94	0.94	0.95	0.85	0.86
100	0.96	0.94	0.94	0.93	0.92	0.92	0.95	0.95	0.96	0.96	0.95	0.87
200	0.96	0.95	0.94	0.92	0.93	0.94	0.97	0.98	0.99	0.97	0.99	0.99
500	0.97	0.97	0.97	0.97	0.96	0.97	1.00	0.99	1.00	0.99	0.99	1.00
1000	0.95	0.98	0.97	0.99	0.98	1.00	1.00	1.00	1.00	1.00	1.00	1.00

Using the campus dataset, the left half of Table 9 presents the performance of RBD with respect to K, which varies from 2 to 100. When K increases, there are more bursts, which can be easily seen from the definition (3) as a large K gives more room for bursts to form. But K does not have significant impact on the detection performance of RBD.

Table 9. Performance of RBD in spread burst detection with respect to K, using the campus dataset, under the parameter settings of $\beta = 100$, $\alpha = 0.1$, and $M = 2$Mb, and using the CAIDA-1 dataset, under the parameter settings of $\beta = 100$, $\alpha = 0.1$, and $M = 5$Mb, respectively.

	Campus dataset			CAIDA-1 dataset		
K	Ground truth	Reported	F1-score	Ground truth	Reported	F1-score
2	507	532	0.916	103	107	0.971
5	555	566	0.910	112	117	0.969
10	580	594	0.928	118	120	0.958
20	584	594	0.927	123	124	0.955
50	589	605	0.918	125	126	0.956
100	589	602	0.911	125	126	0.956

Additional Experiments using CAIDA Datasets: We expand our evaluation of RBD with five additional datasets, which are packet traces from different Internet routers downloaded from CAIDA [26]. These five datasets are denoted as CAIDA-1, CAIDA-2, CAIDA-3, CAIDA-4, and CAIDA-5, with 1,389,150,056 packets, 1,080,151,501 packets, 1,837,095,662 packets, 2,284,636,747 packets, and 1,603,885,200 packets, respectively. Each dataset is 1 hour long. Because their traffic intensity is much larger than our campus dataset, we set each epoch to 1 minute. We repeat the previous experiments of RBD for the campus dataset on each of the CAIDA datasets. If not specified otherwise, the default parameters are α =0.1, β = 100, and K = 10. The default memory allocation is M = 5Mb for CAIDA-1, CAIDA-2, and CAIDA-5, and M = 10Mb for CAIDA-3 and CAIDA-4, depending on the size of the dataset. For CAIDA-1, the experimental results on F1-scores are presented in the right half of Tables 5, 8 and 9.

From these experimental results, we can draw similar conclusions as we did from the results of the campus dataset: The right half of Table 5 presents the performance of RBD in F1-score using CAIDA-1 dataset under different memory allocations. The performance of RBD is improved with more memory, e.g., from 500Kb to 1Mb, but as the memory further increases, the rate of performance improvement becomes small. The right half of Table 8 presents the performance of RBD with respect to β and α. RBD works very well when β is large (e.g., 100 or greater for steep bursts), but works less well when β is small (e.g., 20), particularly when α is also very small or is very large (e.g., 0.5 for very shallow bursts). We stress that in network applications such as anomaly detection, steep bursts are of more interest. The right half of Table 9 presents the performance of RBD with respect to K. We can see that the performance of RBD is not very sensitive to K.

Similar conclusions can be drawn from the results for CAIDA-2 through CAIDA-5, which are included in Appendix C for verification.

Additional Experiments using E-commerce Dataset: Finally we present our evaluation results on the E-commerce dataset, which is smaller in comparison to the network packet traces, containing 206859 products (flows) and 109,950,747 reviews (items) recorded in 61 days, with each epoch being a day. Detecting spread burst can help track the popularity of the products. We repeat the same experiments on RBD over this dataset. If not specified otherwise, the default parameters are α =0.1, β = 400, K = 10, and M = 2Mb.

Table 10 presents the performance of RBD with respect to memory allocation, in terms of burst increase, burst decrease and spread burst detection. The performance of RBD is improved with more memory, but after the memory reaches a certain level (such as 2000Kb or 2Mb), the rate of performance improvement is generally moderate with additional memory. Table 11 presents the performance of RBD with respect to β and α. RBD works well for steep bursts with large β values, but works less well when β is small, particularly when α is also very small or is very large (e.g., 0.5 for very shallow bursts). We believe that steep bursts (jumping popularity of products) are again of

more interest. Table 12 presents the performance of RBD with respect to K. We can see that the performance of RBD has only modest sensitivity to K.

Table 10. F1-score of RBD in burst increase, burst decrease, and spread burst detection, w.r.t. memory allocation, with $\beta = 400$, $\alpha = 0.1$, and $K = 10$, using the E-commerce dataset.

Memory (Kb)	Increase	Decrease	Spread burst
500	0.267	0.314	0.366
1000	0.603	0.750	0.642
2000	0.837	0.811	0.913
5000	0.930	0.952	0.932
10000	1.000	0.952	0.952

Table 11. F1-score of RBD in spread burst detection, w.r.t. β and α, under K=10 and M = 2Mb, using the E-commerce dataset.

β \ α	0.5	0.2	0.1	0.05	0.02	0.01
200	0.71	0.73	0.80	0.79	0.57	0.52
400	0.78	0.77	0.91	0.80	0.83	0.67
600	0.77	0.85	0.92	0.89	1.00	0.86
800	0.77	0.71	0.80	0.80	0.80	1.00
1000	0.77	0.91	1.00	1.00	1.00	1.00

Table 12. Performance of RBD in spread burst detection, w.r.t. K, under $\beta = 400$, $\alpha = 0.1$, and M = 2Mb, using the E-commerce dataset.

K	Ground truth	Reported	F1-score
2	14	15	0.965
5	19	23	0.904
10	21	25	0.913
20	21	26	0.893
50	22	27	0.897

7 CONCLUSION

This paper introduces a new problem of detecting burst increases, burst decreases and spread bursts in real time. It proposes a new self-adaptive sketch (SAS) for recording data items in an evolving data structure and providing flow estimation at any time with low overhead. It uses the self-adaptive sketch as the building block to design a new super spreader identifier (RSI), which detects super spreaders in real time with low overhead. It then uses the super spreader identifier as the building block to design an efficient, real-time solution (RBD) for spread burst detection. We evaluate SAS, RSI and RBD experimentally based on six real network traffic traces and an E-commerce dataset. The results demonstrate that SAS and RSI significantly outperform the state of the art, and RBD detects spread bursts with good accuracy and efficiency. As a future work, we will experimentally study the proposed solution in other application contexts to evaluate its generality and derive context-specific optimizations.

ACKNOWLEDGMENT

This work is supported in part by the National Science Foundation under grants SCC-2124858 and CNS-1909077, and by the National Institutes of Health under grant R01 LM014027.

REFERENCES

[1] [n. d.]. Amazon Kinesis Data Streams. https://aws.amazon.com/kinesis/data-streams/.

[2] 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.

[3] Ran Ben Basat, Gil Einziger, Michael Mitzenmacher, and Shay Vargaftik. 2021. SALSA: Self-adjusting Lean Streaming Analytics. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 864–875.

[4] Ran Ben-Basat, Gil Einziger, Shir Landau Feibish, Jalil Moraney, Bilal Tayh, and Danny Raz. 2021. Routing-Oblivious Network-Wide Measurements. *IEEE/ACM Transactions on Networking* 29, 6 (2021), 2386–2398.

[5] 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.

[6] G. Cormode. 2011. Sketch Techniques for Approximate Query Processing. *Foundations and Trends in Sample, NOW publishers* (2011).

[7] Graham Cormode and S Muthukrishnan. 2005. Space Efficient Mining of Multigraph Streams. In *Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 271–282.

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

[9] 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.

[10] 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.

[11] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric 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.

[12] 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.

[13] 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

[14] Amit Goyal, Hal Daumé III, and Graham Cormode. 2012. Sketch Algorithms for Estimating Point Queries in NLP. In *Proceedings of the 2012 joint conference on empirical methods in natural language processing and computational natural language learning*. 1093–1103.

[15] S. Heule, M. Nunkesser, and A. Hall. 2013. HyperLogLog in Practice: Algorithmic Engineering of a State-of-The-Art Cardinality Estimation Algorithm. *Proc. of EDBT* (2013).

[16] Kaggle. 2020. eCommerce behavior data from multi category store (Dec. 2019 - April 2020). https://www.kaggle.com/datasets/mkechinov/ecommerce-behavior-data-from-multi-category-store?resource=download.

[17] 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.

[18] T. Li, S. Chen, and Y. Ling. 2011. Fast and Compact Per-Flow Traffic Measurement through Randomized Counter Sharing. *IEEE INFOCOM* (2011).

[19] 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.

[20] Yang Liu, Wenji Chen, and Yong Guan. 2015. Identifying High-cardinality Hosts From Network-wide Traffic Measurements. *IEEE Transactions on Dependable and Secure Computing* 13, 5 (2015), 547–558.

[21] Chaoyi Ma, Haibo Wang, Olufemi Odegbile, and Shigang Chen. 2021. Noise Measurement and Removal for Data Streaming Algorithms with Network Applications. In *2021 IFIP Networking Conference (IFIP Networking)*. IEEE, 1–9.

[22] David Moore, Colleen Shannon, and K Claffy. 2002. Code-Red: A Case Study on the Spread and Victims of an Internet Worm. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*. 273–284.

[23] Debjyoti Paul, Yanqing Peng, and Feifei Li. 2019. Bursty Event Detection Throughout Histories. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1370–1381.

[24] Lu Tang, Qun Huang, and Patrick PC Lee. 2020. SpreadSketch: Toward Invertible and Network-wide Detection of Superspreaders. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE, 1608–1617.

[25] Daniel 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.

[26] UCSD. 2019. The CAIDA Anonymized Internet Traces Dataset (April 2008 - January 2019). https://www.caida.org/catalog/datasets/passive_dataset/.

[27] Shobha Venkataraman, Dawn Song, Phillip B Gibbons, and Avrim Blum. 2004. *New Streaming Algorithms for Fast Detection of Superspreaders.* Technical Report. Carnegie-Mellon Univ Pittsburgh Pa School Of Computer Science.
[28] Haibo Wang, Chaoyi Ma, Shigang Chen, and Yuanda Wang. 2022. Fast and Accurate Cardinality Estimation by Self-Morphing Bitmaps. *IEEE/ACM Transactions on Networking* (2022).
[29] 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.
[30] 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.
[31] 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).
[32] 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.
[33] 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.
[34] 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.
[35] Wikipedia. 2023. Corrected sample standard error. https://en.wikipedia.org/wiki/Standard_deviation.
[36] Q. Xiao, S. Chen, M. Chen, and Y. Ling. 2015. Hyper-compact Virtual Estimators for Big Network Data Based on Register Sharing. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 43. ACM, 417–428.
[37] 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.
[38] Q. Xiao, S. Chen, Y. Zhou, and J. Luo. 2020. Estimating Cardinality for Arbitrarily Large Data Stream with Improved Memory Efficiency. *IEEE/ACM Transactions on Networking* 28, 2 (2020), 433–446.
[39] Q. Xiao, Y. Zhou, and S. Chen. 2017. Better with Fewer Bits: Improving the Performance of Cardinality Estimation of Large Data Streams. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*. IEEE, 1–9.
[40] Wei Xie, Feida Zhu, Jing Jiang, Ee-Peng Lim, and Ke Wang. 2016. Topicsketch: Real-time Bursty Topic Detection from Twitter. *IEEE Transactions on Knowledge and Data Engineering* 28, 8 (2016), 2216–2229.
[41] T. Yang, H. Zhou, Y.and Jin, S. Chen, and X. Li. 2017. Pyramid Sketch: A Sketch Framework for Frequency Estimation of Data Streams. *Proceedings of the VLDB Endowment* 10, 11 (2017), 1442–1453.
[42] M. Yoon, T. Li, S. Chen, and J. Peir. 2009. Fit a Spread Estimator in Small Memory. In *IEEE INFOCOM 2009*. IEEE, 504–512.
[43] M. Yu, L. Jose, and R. Miao. 2013. Software Defined Traffic Measurement with OpenSketch. *Proc. of USENIX Symposium on Networked Systems Design and Implementation* (2013).
[44] Qi Zhao, Abhishek Kumar, and Jun (Jim) Xu. 2005. Joint Data Streaming and Sampling Techniques for Detection of Super Sources and Destinations.. In *Internet Measurement Conference*. 77–90.
[45] Zheng Zhong, Shen Yan, Zikun Li, Decheng Tan, Tong Yang, and Bin Cui. 2021. BurstSketch: Finding Bursts in Data Streams. In *Proceedings of the 2021 International Conference on Management of Data*. 2375–2383.
[46] 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.
[47] 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.
[48] Cliff Changchun Zou, Weibo Gong, and Don Towsley. 2002. Code Red Worm Propagation Modeling and Analysis. In *Proceedings of the 9th ACM conference on Computer and communications security*. 138–147.

APPENDIX A. SAS RECORDING ALGORITHM

Algorithm 4: Record a data item $\langle f, e\rangle$ in SAS

Input: $\langle f, e\rangle$, A
Output: True if an update event occurs or false otherwise

compute $h(f,e)$, $G(f,e)$
if $B = 1$ **then**
  if $A[h(f,e)] < 31$ *and* $A[h(f,e)] < G(f,e)$ **then**
    if $G(f,e) < 31$ **then**
      $P := P - \frac{2^{-A[h(f,e)]}}{m} + \frac{2^{-G(f,e)}}{m}$, $A[h(f,e)] := G(f,e)$
    else
      $P := P - \frac{2^{-A[h(f,e)]}}{m}$, $A[h(f,e)] := 31$
    return true
else if $G(f,e) \geq 16$ **then**
  $B := 1,\ P := 0$
  for $i \in [0, m)$ **do**
    if $A[i].b = 0$ **then**
      $A[i] := \max\{A[i].r[0], A[i].r[1]\}$
    else
      $A[i] := A[i].r4$
    $P := P + \frac{2^{-A[i]}}{m}$
  if $G(f,e) < 31$ **then**
    $P := P - \frac{2^{-A[h(f,e)]}}{m} + \frac{2^{-G(f,e)}}{m}$, $A[h(f,e)] := G(f,e)$
  else
    $P := P - \frac{2^{-A[h(f,e)]}}{m}$, $A[h(f,e)] := 31$
  return true
else if $A[h(f,e)].b = 0$ **then**
  compute $h'(f,e)$
  if $A[h(f,e)].r[h'(f,e)] < G(f,e)$ **then**
    if $G(f,e) \leq 3$ **then**
      $P := P - \frac{2^{-A[h(f,e)].r[h'(f,e)]}}{2m} + \frac{2^{-G(f,e)}}{2m}$,$A[h(f,e)].r[h'(f,e)] := G(f,e)$
    else
      $P := P - \frac{2^{-A[h(f,e)].r[0]}+2^{-A[h(f,e)].r[1]}}{2m} + \frac{2^{-G(f,e)}}{m}$, $A[h(f,e)].r4 := G(f,e)$,
       $A[h(f,e)].b := 1$
    return true
else
  if $A[h(f,e)].r4 < G(f,e)$ **then**
    $P := P - \frac{2^{-A[h(f,e)].r4}}{m} + \frac{2^{-G(f,e)}}{m}$, $A[h(f,e)].r4 := G(f,e)$
    return true
return false

APPENDIX B. PROOF OF THEOREM 1

Due to the pseudo randomness of the hash function $h(\cdot,\cdot) \in [0, m-1]$, any data item $\langle f, e\rangle$ will be randomly hash to a five-bit unit $A[h(f,e)]$ with the probability of $\frac{1}{m}$. Without loss of generality, we consider an arbitrary unit $A[i]$ with $0 \le i < m$. Define ΔP_i as the accumulative variation in P caused by the update in $A[i]$. Summing up accumulative variation in P across all units, we have

$$P = 1 + \sum_{i=0}^{m-1} \Delta P_i \tag{9}$$

We have the following lemma for ΔP_i.

Lemma 1. *At any time the probability for the next arrival data item to update $A[i]$ with $0 \le i < m$ is $\frac{1}{m} + \Delta P_i$.*

Proof. Depending on the interpretation of $A[i]$, ΔP_i is represented differently. There are three cases.

• Case 1: $A[i].b = 0$. In this case, $A[i]$ contains two two-bit registers, $A[i].r[0]$ and $A[i].r[1]$. Since each item will go to either one with even probability, we consider $A[i].r[0]$ without loss of generality. Let the number of data items that update $A[i].r[0]$ be z and the arrival sequence of these items are $\langle f_1, e_1\rangle$, $\langle f_2, e_2\rangle$, ..., $\langle f_z, e_z\rangle$. Since $A[i].b = 0$, we know these data items must follow the first case of Section 3.2 as otherwise other cases will change the status of $A[i].b$. Each time an item $\langle f_j, e_j\rangle$ with $1 \le j \le z$ arrives, the value of P will be changed by

$$\begin{cases} -\frac{2^{-G(f_{j-1},e_{j-1})}}{2m} + \frac{2^{-G(f_j,e_j)}}{2m}, & \text{if } j \ge 2; \\ -\frac{1}{2m} + \frac{2^{-G(f_1,e_1)}}{2m}, & \text{if } j = 1. \end{cases} \tag{10}$$

Moreover, the value of $A[i].r[0]$ will be changed from $G(f_{j-1}, e_{j-1})$ to $G(f_j, e_j)$ if $j \ge 2$ or from 0 to $G(f_1, e_1)$ if $j = 1$. Apparently, the current value of $A[i].r[0]$ is $G(f_z, e_z)$. After $\langle f_1, e_1\rangle$, $\langle f_2, e_2\rangle$, ..., $\langle f_z, e_z\rangle$ are recorded in $A[i].r[0]$, the value of P will be changed by

$$\begin{aligned} &-\frac{1}{2m} + \frac{2^{-G(f_1,e_1)}}{2m} + \sum_{j=2}^{z}[-\frac{2^{-G(f_{j-1},e_{j-1})}}{2m} + \frac{2^{-G(f_j,e_j)}}{2m}] \\ &= -\frac{1}{2m} + \frac{2^{-G(f_z,e_z)}}{2m} \\ &= -\frac{1}{2m} + \frac{2^{-A[i].r[0]}}{2m} \end{aligned} \tag{11}$$

The same holds for $A[i].r[1]$. Combining $A[i].r[0]$ and $A[i].r[1]$, we have

$$\Delta P_i = -\frac{1}{m} + \frac{2^{-A[i].r[0]}}{2m} + \frac{2^{-A[i].r[1]}}{2m} \tag{12}$$

Consider an arbitrary data item $\langle f, e\rangle$. It will change the data structure of $A[i]$ if (1) $h(f,e) = i$, $h'(f,e) = 0$ and $G(f,e) > A[i].r[0]$; or (2) $h(f,e) = i$, $h'(f,e) = 1$ and $G(f,e) > A[i].r[1]$. The probability for either condition to happen is $\frac{2^{-A[i].r[0]}}{2m} + \frac{2^{-A[i].r[0]}}{2m} = \frac{1}{m} + \Delta P_i$. Therefore, the lemma holds for the case of $A[i].b = 0$.

• Case 2: $A[i].b = 1$. In this case, $A[i]$ contains a four-bit register, $A[i].r4$. Let the number of data items that update $A[i].r4$ be z and the arrival sequence of these items are $\langle f_1, e_1\rangle$, $\langle f_2, e_2\rangle$, ..., $\langle f_z, e_z\rangle$. Now, we consider three processes.

The first process is the recording of items before (excluding) the arrival of $\langle f_1, e_1\rangle$. Since $A[i].b = 0$ holds all the time, according to the analysis of Case 1, the variation in the value of P is $-\frac{1}{m} + \frac{2^{-A[i].r[0]}}{2m} + \frac{2^{-A[i].r[1]}}{2m}$.

The second process is the recording of item $\langle f_1, e_1\rangle$. At that time $A[i].b = 0$ and $3 < G(f_1, e_1) \leq 15$, corresponding to the second case of Section 3.2, from which we know the variation in the value of P is $-\frac{2^{-A[i].r[0]}}{2m} - \frac{2^{-A[i].r[1]}}{2m} + \frac{2^{-G(f_1,e_1)}}{m}$.

The third process is the recording of item $\langle f_2, e_2\rangle$,..., $\langle f_z, e_z\rangle$. Since $A[i].b = 1$ and $3 < G(f_j, e_j) \leq 15$ $\forall 2 \leq j \leq z$ always hold, this process follows the third case of Section 3.2. For each item $\langle f_j, e_j\rangle$, its recording will change the value of P by $-\frac{2^{-G(f_{j-1},e_{j-1})}}{m} + \frac{2^{-G(f_j,e_j)}}{m}$ with $2 \leq j \leq z$. Overall the variation in P in this process is $-\frac{2^{-G(f_1,e_1)}}{m} + \frac{2^{-G(f_z,e_z)}}{m}$. Moreover, we know the update of the last item $\langle f_z, e_z\rangle$ will set $A[i].r4 = G(f_z, e_z)$.

Combing the above three processes, we know the total variation in the value of P is

$$\begin{aligned}\Delta P_i = &-\frac{1}{m} + \frac{2^{-A[i].r[0]}}{2m} + \frac{2^{-A[i].r[1]}}{2m} \\ &- \frac{2^{-A[i].r[0]}}{2m} - \frac{2^{-A[i].r[1]}}{2m} + \frac{2^{-G(f_1,e_1)}}{m} \\ &- \frac{2^{-G(f_1,e_1)}}{m} + \frac{2^{-G(f_z,e_z)}}{m} \\ &= -\frac{1}{m} + \frac{2^{-G(f_z,e_z)}}{m} \\ &= -\frac{1}{m} + \frac{2^{-A[i].r4}}{m}\end{aligned} \tag{13}$$

Consider an arbitrary data item $\langle f, e\rangle$. It will change the data structure of $A[i]$ if $h(f, e) = i$ and $G(f, e) > A[i].r4$, which happen with the probability of $\frac{2^{-A[i].r4}}{m} = \frac{1}{m} + \Delta P_i$. Therefore, the lemma holds for the case of $A[i].b = 1$.

• Case 3: $B = 1$. In this case, $A[i]$ is a five-bit register. Let the number of data items that update $A[i]$ be z and the arrival sequence of these items are $\langle f_1, e_1\rangle$, $\langle f_2, e_2\rangle$, ..., $\langle f_z, e_z\rangle$. We know $G(f_j, e_j) > 15$ with $\forall 1 \leq j \leq z$, corresponding to the fourth case of Section 3.2, from which we know initially P was recalculated as $\sum_{i=0}^{m-1} \frac{2^{-A[i]}}{m}$. So the variation caused by $A[i]$ in the value of P is $\frac{2^{-A[i]}}{m}$. According to (8), we will change the value of P for when recording an item $\langle f_j, e_j\rangle$ by

$$\begin{cases} -\frac{2^{-A[i]}}{m} + \frac{2^{-G(f_j,e_j)}}{m}, & \text{if } G(f_j, e_j) < 31 \\ -\frac{2^{-A[i]}}{m}, & \text{if } G(f_j, e_j) \geq 31; \end{cases} \tag{14}$$

and update $A[i] = \min\{G(f_j, e_j), 31\}$. After recording all items, $A[i] = G(f_z, e_z)$ and the accumulative variation in the value of P is

$$\Delta P_i = \begin{cases} -\frac{1}{m} + \frac{2^{-A[i]}}{m}, & \text{if } A[i] < 31 \\ -\frac{1}{m}, & \text{if } A[i] = 31. \end{cases} \tag{15}$$

Consider an arbitrary data item $\langle f, e\rangle$. It will change the data structure of $A[i]$ if $h(f, e) = i$ and $G(f, e) > A[i]$, which happens with the probability of $\frac{2^{-A[i]}}{m} = \frac{1}{m} + \Delta P_i$ if $A[i] < 31$, and the probability of $0 = \frac{1}{m} + \Delta P_i$, if $A[i] = 31$ as $A[i]$ can not be updated any more. Therefore, the lemma holds for the case of $B = 1$. □

Lemma 1 is applicable to any unit $A[i]$, $\forall 0 \leq i < m$. Combining all units in A, we know at any time the probability for the next arrival data item to update A is $\sum_{i=0}^{m-1}[\frac{1}{m} + \Delta P_i] = P$. The equation is derived because of (9). Therefore, the theorem holds.

APPENDIX C. ADDITIONAL EXPERIMENTAL RESULTS FOR CAMPUS DATASETS AND CAIDA DATASETS

In Tables 13-15, the first column varies the memory allocation from 100Kb to 10000Kb (i.e., 10Mb), the second column shows the actual number of burst increases in the data (ground truth), the third column shows the number of reported burst increases by RBD, the fourth column shows the number of TPs, the fifth column shows the number of FPs, the sixth column shows the number of FNs, and the last column shows the F1-score.

Table 13. Performance of RBD in spread burst increase detection with respect to memory allocation, using the campus dataset.

Memory (Kb)	Ground truth	Reported	TP	FP	FN	F1-score
100	1091	1347	919	428	172	0.753
200	1091	1144	1012	132	79	0.905
500	1091	1108	1038	70	53	0.944
1000	1091	1103	1049	54	42	0.956
2000	1091	1091	1040	51	51	0.953
5000	1091	1110	1056	54	35	0.959
10000	1091	1097	1043	54	48	0.953

Table 14. Performance of RBD in burst decrease detection with respect to memory allocation, using the campus dataset.

Memory (Kb)	Ground truth	Reported	TP	FP	FN	F1-score
100	807	1170	690	480	117	0.698
200	807	891	751	140	56	0.884
500	807	832	766	66	41	0.934
1000	807	817	751	66	56	0.924
2000	807	818	758	60	49	0.932
5000	807	824	764	60	43	0.936
10000	807	833	766	67	41	0.934

Table 15. Performance of RBD in spread burst detection with respect to memory allocation, using the campus dataset.

Memory (Kb)	Ground truth	Reported	TP	FP	FN	F1-score
100	580	729	434	295	146	0.663
200	580	628	507	121	73	0.839
500	580	608	550	58	30	0.926
1000	580	608	550	58	30	0.926
2000	580	594	545	49	35	0.928
5000	580	600	550	50	30	0.932
10000	580	598	550	48	30	0.933

Table 16. F1-score of RBD in burst increase, burst decrease, and spread burst detection, w.r.t. memory allocation, with $\beta = 100$, $\alpha = 0.1$, and $K = 10$, using the CAIDA-2 dataset, containing 1,080,151,501 packets.

Memory (Kb)	Increase	Decrease	Spread burst
500	0.831	0.814	0.819
1000	0.918	0.862	0.918
2000	0.947	0.929	0.920
5000	0.938	0.958	0.920
10000	0.969	0.958	0.912

Table 17. F1-score of RBD in spread burst detection, w.r.t. β and α, where K=10 and M = 5Mb, using the CAIDA-2 dataset, containing 1,080,151,501 packets.

β \ α	0.5	0.2	0.1	0.05	0.02	0.01
20	0.54	0.67	0.78	0.82	0.69	0.91
50	0.82	0.86	0.85	0.84	0.70	0.92
100	0.89	0.94	0.92	0.94	0.96	0.92
200	0.94	0.93	0.95	0.96	1.00	1.00
500	1.00	1.00	1.00	1.00	1.00	1.00
1000	1.00	1.00	1.00	1.00	1.00	1.00

Table 18. Performance of RBD in spread burst detection, w.r.t. K, with $\beta = 100$, $\alpha = 0.1$, and M = 5Mb, using the CAIDA-2 dataset, containing 1,080,151,501 packets.

K	Ground truth	Reported	F1-score
2	43	48	0.923
5	46	51	0.928
10	47	53	0.920
20	52	59	0.919
50	52	59	0.919

Table 19. F1-score of RBD in burst increase, burst decrease, and spread burst detection, w.r.t. memory allocation, with $\beta = 100$, $\alpha = 0.1$, and $K = 10$, using the CAIDA-3 dataset, containing 1,837,095,662 packets.

Memory (Kb)	Increase	Decrease	Spread burst
1000	0.676	0.718	0.705
2000	0.913	0.906	0.914
5000	0.954	0.971	0.988
10000	0.937	0.965	0.909
20000	0.948	0.954	0.954

Table 23. F1-score of RBD in spread burst detection, w.r.t. β and α, where K=10 and M = 10Mb, using the CAIDA-4 dataset, containing 2,284,636,747 packets.

β \ α	0.5	0.2	0.1	0.05	0.02	0.01
20	0.63	0.81	0.83	0.83	0.90	0.87
50	0.91	0.91	0.91	0.92	0.92	0.88
100	0.94	0.93	0.91	0.95	0.96	0.91
200	0.92	0.94	0.92	0.95	0.94	0.95
500	0.98	0.98	0.98	0.98	0.99	0.99
1000	0.98	0.97	0.97	0.98	0.99	0.99

Table 20. F1-score of RBD in spread burst detection, w.r.t. β and α, where K=10 and M = 10Mb, using the CAIDA-3 dataset, containing 1,837,095,662 packets.

β \ α	0.5	0.2	0.1	0.05	0.02	0.01
20	0.56	0.76	0.76	0.49	0.78	0.76
50	0.85	0.92	0.91	0.90	0.79	0.76
100	0.93	0.92	0.91	0.91	0.93	0.80
200	0.96	0.97	0.94	0.95	0.99	0.99
500	1.00	1.00	0.98	0.98	1.00	1.00
1000	0.97	0.97	1.00	1.00	1.00	1.00

Table 21. Performance of RBD in spread burst detection, w.r.t. K, with $\beta = 100$, $\alpha = 0.1$, and M = 10Mb, using the CAIDA-3 dataset, containing 1,837,095,662 packets.

K	Ground truth	Reported	F1-score
2	78	78	0.897
5	88	88	0.909
10	88	88	0.909
20	89	89	0.910
50	90	90	0.911

Table 22. F1-score of RBD in burst increase, burst decrease, and spread burst detection, w.r.t. memory allocation, with $\beta = 100$, $\alpha = 0.1$, and $K = 10$, using the CAIDA-4 dataset, containing 2,284,636,747 packets.

Memory (Kb)	Increase	Decrease	Spread burst
500	0.344	0.335	0.328
1000	0.692	0.703	0.699
2000	0.842	0.836	0.852
5000	0.902	0.901	0.897
10000	0.918	0.915	0.914
20000	0.928	0.926	0.904
50000	0.913	0.915	0.906

Table 24. Performance of RBD in spread burst detection, w.r.t. K, with $\beta = 100$, $\alpha = 0.1$, and M = 10Mb, using the CAIDA-4 dataset, containing 2,284,636,747 packets.

K	Ground truth	Reported	F1-score
2	506	516	0.927
5	627	646	0.910
10	668	690	0.914
20	675	696	0.914
50	677	699	0.914

Table 25. F1-score of RBD in burst increase, burst decrease, and spread burst detection, w.r.t. memory allocation, with $\beta = 100$, $\alpha = 0.1$, and $K = 10$, using the CAIDA-5 dataset, containing 1,603,885,200 packets.

Memory (Kb)	Increase	Decrease	Spread burst
500	0.707	0.713	0.716
1000	0.897	0.898	0.907
2000	0.962	0.966	0.955
5000	0.962	0.966	0.969
10000	0.963	0.969	0.965

Table 26. F1-score of RBD in spread burst detection, w.r.t. β and α, where K=10 and M = 5Mb, using the CAIDA-5 dataset, containing 1,603,885,200 packets.

β \ α	0.5	0.2	0.1	0.05	0.02	0.01
20	0.47	0.66	0.77	0.85	0.89	0.85
50	0.82	0.95	0.97	0.97	0.89	0.85
100	0.97	0.97	0.97	0.97	0.96	0.86
200	0.97	0.96	0.97	0.97	0.97	0.95
500	0.97	0.97	0.97	0.97	0.97	0.96
1000	1.00	1.00	0.99	0.99	1.00	0.99

Table 27. Performance of RBD in spread burst detection, w.r.t. K, with $\beta = 100$, $\alpha = 0.1$, and M = 5Mb, using the CAIDA-5 dataset, containing 1,603,885,200 packets.

K	Ground truth	Reported	F1-score
2	324	319	0.970
5	344	338	0.971
10	346	341	0.969
20	351	345	0.971
50	355	349	0.972

Received October 2022; revised December 2022; accepted April 2023