Randomized Error Removal for Online Spread Estimation in Data Streaming

Haibo Wang
University of Florida
Gainesville, FL
wanghaibo@ufl.edu

Chaoyi Ma
University of Florida
Gainesville, FL
ch.ma@ufl.edu

Olufemi O Odegbile
University of Florida
Gainesville, FL
oodegbile@ufl.edu

Shigang Chen
University of Florida
Gainesville, FL
sgchen@cise.ufl.edu

Jih-Kwon Peir
University of Florida
Gainesville, FL
peir@cise.ufl.edu

ABSTRACT

Measuring flow spread in real time from large, high-rate data streams has numerous practical applications, where a data stream is modeled as a sequence of data items from different flows and the spread of a flow is the number of distinct items in the flow. Past decades have witnessed tremendous performance improvement for single-flow spread estimation. However, when dealing with numerous flows in a data stream, it remains a significant challenge to measure per-flow spread accurately while reducing memory footprint. The goal of this paper is to introduce new multi-flow spread estimation designs that incur much smaller processing overhead and query overhead than the state of the art, yet achieves significant accuracy improvement in spread estimation. We formally analyze the performance of these new designs. We implement them in both hardware and software, and use real-world data traces to evaluate their performance in comparison with the state of the art. The experimental results show that our best sketch significantly improves over the best existing work in terms of estimation accuracy, data item processing throughput, and online query throughput.

PVLDB Reference Format:
Haibo Wang, Chaoyi Ma, Olufemi O Odegbile, Shigang Chen, and Jih-Kwon Peir. Randomized Error Removal for Online Spread Estimation in Data Streaming. PVLDB, 14(6): 1040-1052, 2021.
doi:10.14778/3447689.3447707

1 INTRODUCTION

Measuring statistics in real time from large, high-rate data streams has numerous applications [10, 11, 20, 30, 33, 36, 41, 43, 45]. Traditionally, a data stream is modeled as a sequence of data items $\{d_1, d_2, d_3, d_2, \ldots\}$, and the statistics of interest include the frequencies of the data items appearing in the stream [17, 18, 21, 22, 27, 40, 45] and the number of different items called the *cardinality* of the stream [22, 37, 40].

Proceedings of the VLDB Endowment, Vol. 14, No. 6 ISSN 2150-8097.
doi:10.14778/3447689.3447707

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. This paper studies how to measure the *spread* of each flow, which is defined as the number of distinct elements in each flow. As a special case, if we treat the whole stream as a single flow, its spread is the traditional cardinality.

The general model has many applications such as P2P hot-spot localization [28], web caching prioritization [3, 46], detection of DDoS attacks [2, 24], port scanning measurement [10] and worm propagation detection [6, 29], which do not fit with the traditional model. Below we give several examples in the context of Internet applications. Consider a packet stream that arrives at a high-speed router, with each packet modelled as $\langle f, e \rangle$, where f and e can be defined arbitrarily based on information from packet header or payload for specific application need. For example, if we consider all packets from the same source address as a flow and use destination addresses as the elements under monitoring, a flow-spread measurement module deployed at a gateway router will detect potential external adversaries that are scanning the internal network — these are external sources with large spreads (i.e., their flows contain too many distinct destinations), or in case of stealthy scanning they are sources with modest spreads at any measurement period but persisting at a spread level higher than normal over a long time [44]. As an opposite example, if we use destination addresses as flow labels and source addresses as elements, spread measurement will help identify the victims of possible DDoS attacks — these are internal destination addresses with large spreads (i.e., their flows contain too many distinct source addresses). In yet another example, a large server farm may learn the popularity of its content by tracking the number of distinct users accessing each file [5], where all users accessing the same file form a flow. Finally, spread measurement has also been applied in various data analysis systems at Google [16]. For instance, Sawzall [26], Dremel [23] and PowerDrill [15] estimate the number of distinct users that search the same key, where we can model all search requests for the same key as a flow and user identities (e.g., their IP addresses) as the elements in each flow.

This paper is interested in per-flow spread measurement, allowing users to query the spread of any flow online in real time. We have three performance requirements for the design of a spread measurement module. First, it should incur low processing overhead per data item in order to support high-rate streaming. Second, it should be memory-efficient in order to support software/hardware implementation on the data plane of a streaming device which may operate on cache memory. Third, it should support efficient online spread queries to support real-time applications. We again use packet stream in high-speed networks as example to justify the requirements. Modern routers forward packets at hundreds of gigabits or even terabits per second (at least 8.3M packets or 83M packets per second considering Maximum transmission unit for Ethernet is 1500 bytes). Tracking a large number of flows simultaneously can be a serious challenge. Specifically, if one wants to perform online flow spread measurement in real time, one way is to implement the measurement module on data-plane network processors. Since their on-chip circuitry and cache memory have to be shared among many other routing/performance/security functions, low overhead and memory efficiency become highly desirable properties in order not to create performance bottleneck.

There are two categories of solutions for per-flow spread measurement. One category estimates each flow with a separate data structure, called *spread estimator*. To count distinct elements, it must be able to remember the elements that it has seen. Such single-flow estimators include bitmap [38], FM [13], multi-resolution bitmap [11], KMV [4], LogLog [9] and HyperLogLog (HLL) [12, 16]. They require hundreds or thousands of bits for each flow in order to achieve good accuracy and range. When the number of flows is numerous, monitoring all flows with separate data structures can be too costly. We need more compact data structures called *spread sketches* that monitor all flows simultaneously without linearly increasing the memory overhead, which leads to the second category of solutions [8, 39, 42, 46]. They share a certain number of spread estimators among all flows when recording their elements. But sharing causes error in spread estimation. When a flow shares an estimator with other flows, the estimator produces the combined spread of all those flows instead of the spread of an individual flow. To reduce the error, the current approach [8, 42, 46] follows the idea of CountMin [7] by mapping each flow to multiple estimators, making multiple spread estimations, and taking the minimum answer. However, it is well known that this approach has a positively biased error that can be very large when the multiple estimators are all shared with other large flows. Moreover, since each flow has to be recorded in multiple estimators and each query has to be computed from multiple estimators, both the processing overhead per data item and the query overhead for each flow are increased multi-fold.

The goal of this paper is to design new spread sketches for online per-flow spread estimation (in the scenarios described before) that incur much smaller processing overhead and query overhead than the start of the art, yet result in much less error in spread estimation. More specifically, we want the processing overhead and the query overhead to be multiple times smaller, and the error to be an order of magnitude smaller. To achieve these seemingly conflicting objectives, we cannot follow the prior approaches but need to explore new paths toward compact and efficient recording of data items in a way that enables error removal. We introduce two new sketch designs, called randomized error-reduction sketch (rSkt) and unit-level randomized error-reduction sketch (rSkt2). Their basic idea is to spread the error due to estimator sharing evenly between a primary estimator and a complement estimator, so that the error can be subtracted away. We formally analyze the performance of these new sketches. We implement them in both hardware and software, and use real-world data traces to evaluate their performance in comparison with the state of the art. The experimental results show that our best sketch significantly improves over the best existing work in terms of estimation accuracy (up to 99.5% estimation error reduction), data item processing throughput (up to 126% throughput improvement), and online query throughput (up to 3 times throughput improvement), thanks to its randomized error-reduction design.

2 BACKGROUND AND PRIOR ART

2.1 Problem Statement

The traditional model considers a data stream as a sequence of data items, . . . d . . . The classical measurements include the frequencies of the data items [17, 18, 21, 22, 27, 40, 45] and the *spread* (or cardinality) of the stream [22, 40], which is defined as the number of distinct items in the stream. For example, for a data stream of $\{d_1, d_2, d_1, d_1\}$, the frequency of d_1 is 3, that of d_2 is 1, and the spread of the whole stream is 2 because there are two distinct items, d_1 and d_2.

This paper adopts a more general model where a data stream consists of a continuous sequence of data items $\langle f, e\rangle$, where f is a flow label and e is an element identifier. All data items with the same label form a flow. Essentially, we divide the stream into multiple sub-streams, called flows, each with a separate label and containing a set of elements — in fact, it is a multi-set because an element may appear multiple times as we will explain shortly with an example. There can be different measurement tasks: (1) finding the size of each flow f, i.e., the number of data items in flow f, (2) finding the number of different flows, and (3) find the *spread* of each flow f, i.e., the number of distinct elements in flow f. The first two tasks are identical to those in the traditional model; we simply replace f with d and ignore e. The third task is what this paper will focus on, which leads to our problem statement: *design an efficient sketch (i.e., compact data structure) that records the data items of a given stream, and support online queries for spread estimates on any given flow labels. Online queries are performed live when we process the data stream. They are important for applications that require real-time responses. In contrast, offline queries are performed after the data stream has been processed [19, 39, 41, 46], and thus not subject to any real-time requirement.*

For a continuous data stream (such as a packet stream on the Internet), measurement is typically done in each pre-defined period, and the content of the sketch is offloaded to a server for long-term storage after each period.

We use a simple example to explain the above model. Consider the gateway of an enterprise network. Suppose that it is configured to monitor the inbound packet stream for scan detection. Each packet is abstracted as a date item $\langle f, e\rangle$. We define all packets from the same source address as a flow; hence, the flow label f

is the source address carried in the packet header. We define the destination address/port in the packet header as element e under measurement.

• 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.

Now suppose an external source (f) sends 1,000,000 packets through the gateway. If the packets go to the same destination/port (e), the spread for the external source is 1. But if the packets all go to different destination address/port pairs, the spread is 1,000,000; it is likely that the external source is scanning the enterprise network. If the gateway monitors the spreads of all sources simultaneously, it can detect the scanners in real time. This application fits well with the general model, but not with the traditional model. Additional application examples can be found in the introduction.

In some applications, we only need to monitor the super spreaders (those with very large spreads) [8, 31]. However, there are other scenarios where the spread information of non-super spreaders is also useful. For example, to avoid detection, stealthy scanners may probe a small number of destination addresses/ports at any time but do so persistently over a long period. If we measure the spreads of all flows and analyze such information over time, we will be able to find these stealthy scanners that are not aggressive at any instant but persist in low-rate scanning. In another example, suppose that an intrusion detection system identifies a set of worm-infected hosts that perform probing to infect others. With per-flow measurement, we will be able to examine these hosts in the measurements taken from the previous periods and find out when each of them begins its probing (which results in spread increase). This helps us to establish infection timeline among the hosts for traceback purpose. Moreover, we can query their current probing rates in real time as per-flow measurement is performing in the current period. In general, measuring the spreads of all flows enables us to perform broad analysis over long-term flow behaviors in order to detect subtle anomalies that deviate from the norm. Additional applications of per-flow measurement on stealthy attack detection, fine-grained traffic analysis, flow loss map, ECMP debugging, and TCP timely attack detection can be found in [18].

It is more difficult to measure the spread of a flow than doing so for the size of a flow (which requires only a counter). The reason is that we have to "remember" the elements that have been seen before in order to remove duplicate elements, and that takes a lot of memory space if the flow has a very large number of distinct elements. The overhead can be greatly reduced if we provide a spread estimate. In this paper, we refer to a data structure that records the elements of a flow and provides a spread estimate as an *estimator*. Below we discuss the related work, beginning with single-flow spread estimators.

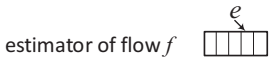


Figure 1: An estimator is an array of units (bits, FM registers or HLL registers). Any element e of flow f will be recorded in one of the units.

Table 1: Memory need for different single-flow spread estimators with m units.

Estimators	Memory	Remark
bitmap	m	$m \ln m > n$. n is the real spread of flow
FM	$32m$	recommended as $m = 128$
HLL	$5m$	recommended as $m = 128$

2.2 Single-flow Spread Estimators

To monitor the spread of a flow, a naive solution is to use a hash table to store the received elements for duplicate removal [14, 34], but this is costly as a flow may have millions or billions of distinct elements.

More efficient single-flow spread estimators include bitmap [8, 11, 38], FM sketch [13], and HLL sketch [12, 16]. We unify their description as follows: As shown in Figure 1, each estimator is an array U of m units, where each unit, $U[i], 0 \le i < m$, is a bit, a 32-bit register, or a 5-bit register for bitmap, FM or HLL, respectively. The memory consumption of single-flow spread estimators are shown in Table 1.

When receiving an element e of the flow, we hash e to one of the units, $U[h(e)]$, for recording, where $h(.)$ is a hash function. The recording operation depends on the unit type. In case of bitmap, we set $U[h(e)]$ to one. In case of FM, we choose one bit in $U[h(e)]$ to set, with the ith bit being chosen at probability $(\frac{1}{2})^{i+1}$, where $0 \le i < 32$. In case of HLL, we update $U[h(e)] = \max\{U[h(e), i\}$, where value i is randomly chosen from $[1, 32)$ with probability $(\frac{1}{2})^i$.

When we estimate the flow's spread, we average across the array. This computation also depends on the unit type. In case of bitmap [38],

$$avg = \frac{\sum_{i=0}^{m-1} U[i]}{m}.$$

In case of FM [13], let $\rho(U[i])$ be the number of consecutive ones starting from the lowest-order bit in $U[i]$.

$$avg = \frac{\sum_{i=0}^{m-1} 2^{\rho(U[i])}}{m}$$

In case of HLL [12, 16], harmonic averaging is used to tame the impact of outliers.

$$avg = \frac{m}{\sum_{i=0}^{m-1} \frac{1}{2^{U[i]}}}$$

From the average, we can estimate the flow spread based on the formulas from the papers cited above. For example, in case of bitmap [11, 38], the spread is estimated as $-m \ln(1 - avg)$.

While UnivMon [22] and ElasticSketch [40] are designed to measure the sizes of flows, they also estimate the number of flows, called *cardinality*, in a packet stream. For this purpose, they treat the whole stream as a single giant flow and the flow labels as elements. They belong to single-flow spread estimators, and their memory overhead is very large when comparing with bitmap/FM/HLL.

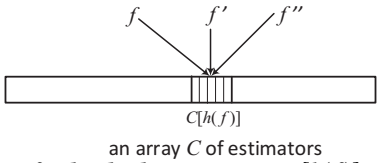


Figure 2: Flow f is hashed to estimator $C[h(f)]$, which carries error from other flows due to hash collision. Recall that each estimator is an array of units.

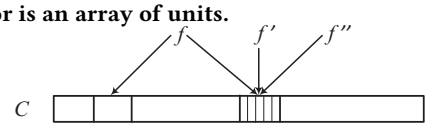


Figure 3: Hashing each flow f to d estimators, where $d > 1$. Note that f' and f'' are also hashed to d estimators each, which are not drawn.

2.3 Multi-flow Overhead Challenge

To monitor multiple flows, we may assign each flow a separate spread estimator. With 5,000 bits, a bitmap estimator has an estimation range up to just $-5000 \times \ln(1/5000) = 42,586$, according to [38]. To achieve good accuracy, an FM (HLL) estimator will need hundreds or thousands of bits [39]. For example, when an HLL estimator takes 640 bits when it uses 128 registers of 5 bits each [46]. Consider one million concurrent flows. The memory requirement for one million bitmaps is 5000 Mb, while that for one million HLL estimators is 640 Mb, which can be a serious problem for online operations, particularly when its implementation uses on-chip cache memory for high speed [46], such as on a network processor for Internet traffic. Today's switches may have 128MB SRAM [25], but this cache memory has to store the routing table and support essential routing/security/performance functions. Moreover, there may be multiple measurement tasks. Therefore, it is highly desirable to minimize the memory consumption of any measurement task.

To save space, if we use fewer estimators than the number of flows, each estimator will have to handle multiple flows. For example, we may hash the flows to an array C of estimators, as is shown in Figure 2, where flows f, f' and f'' are hashed to the same estimator. When we query for the spread of flow f, the estimator will produces an estimate that carries noise (error) from f' and f'' due to hash collision.

We give an example to show the error can be very large in practical scenarios. Suppose that the allocated memory is 10Mb and the number of flows is 10^6, which is validated by the fact that a 10-min CAIDA dataset contains 2.52M flows if we consider each source-destination pair as a flow. Considering the most compact single-flow spread estimator, i.e., HLL, which occupies 640 bits under recommended setting, if each data item is recorded in one estimator, on average, 64 flows share one estimator. Therefore, the error can be very large. Below we explain how the existing literature handles such error.

2.4 Existing Multi-flow Estimators

There are two approaches to reduce error caused by hash collision. One is to hash each flow f to d estimators, as shown in Figure 3, where $d = 2$. The d estimators each produce a spread estimation for flow f. The smallest of the d estimations carries the least error. Essentially, this approach [8, 42] uses the CountMin idea [7] but replaces counters with spread estimators. A generalized design called bSkt can be found in [46].

Table 2: Notations.

$C, \bar{C}$	hash table of estimators
$V(.)$	result of an estimator
f, e	flow label and element identifier
$s_f / \hat{s}_f$	actual/estimated spread of f
d	No. of hashed estimators per data item
w	No. of estimators per hash table
m	No. of units per estimator
$h(.) \in [0, w)$	uniform hash function
$g(.) \in \{0, 1\}$	uniform hash function
$g'(.,.) \in \{0, 1\}$	two-input uniform hash function

For online spread queries, the above approach has two problems. First, even though error is reduced by taking the smallest, our experiments still show significant error. Second, the query computation overhead increases d-fold because, for each flow, we have to perform estimation from d estimators instead of one. This is fine if it is done offline, but will be a problem if it is done online as we process the live stream. Note that in order to ensure decent error reduction, d cannot be too small. Our goal is to design a new sketch that reduces error to a level much lower than bSkt [46] and in the meantime reduces query computation as well.

Another approach in the literature for reducing estimation error is virtual sketches [19, 39, 41, 46], which share a large array of units (e.g., bits, FM/HLL registers) for all flows. More specifically, they construct virtual estimators for individual flows from these shared units. Each flow has its own virtual estimator, which produces a spread estimation that carries error from other flows due to unit sharing. Removing this error will require memory access to the whole unit array and computation across the whole unit array. Such overhead is many times larger than that of bSkt [46], making it unsuitable for online spread queries. Specifically, in our test, the online query throughput of bSkt is at least 50-300 times larger than that of virtual sketches. We will not consider this approach further in the paper for the desire of supporting online queries.

3 RANDOMIZED ERROR-REDUCTION SKETCHES

In this section, we introduce two new randomized sketches for flow spread measurement that produce asymptotically unbiased estimation with much lower error and much lower computation time than the prior art.

3.1 Hash Table

Let's first revisit the hash table approach in Section 2.3 and Figure 2. The hash table is an array C of w estimators. The ith estimator in the table is denoted as $C[i]$, $0 \le i < w$. Each estimator is an array of m units which may be bits, FM registers or HLL registers as explained in Section 2.2 and Figure 1. The jth unit in the ith estimator is denoted as $C[i][j]$, $0 \le i < w$, $0 \le j < m$.

• *Recording*: Consider an arbitrary flow f. It is hashed to $C[h(f)]$. After we receive an item $\langle f, e \rangle$, we hash the element e to a unit, $C[h(f)][h(e)]$, where it is recorded based on the unit type according to Section 2.2. Note that a modulo operation is always assumed

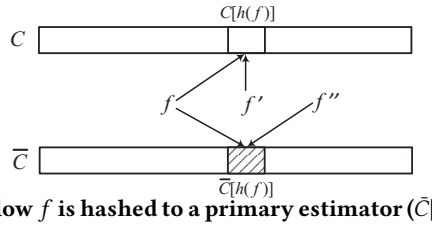


Figure 4: Flow f is hashed to a primary estimator ($\bar{C}[h(f)]$ for this example) and a complement estimator ($C[h(f)]$ for this example). The noise in $F_f - \{f\}$ is split between these two estimators with equal probability. For example, f' is recorded in $C[h(f)]$ and f'' is recorded in $\bar{C}[h(f)]$.

in this paper to keep the hash output in the proper range. For example, $C[h(f)][h(e)]$ should actually be $C[h(f) \bmod w][h(e) \bmod m]$. Some important notations are shown in Table 2 for reference.

• *Querying*: Upon receiving a spread query on flow f, we produce an estimate from $C[h(f)]$ based on its type (bitmap, FM or HLL); see Section 2.2. The result is denoted as $V(C[h(f)])$, where $V(.)$ refers to the type-dependent estimation formula.

Let F_f be the set of flows that are hashed to $C[h(f)]$, i.e., $\forall f' \in F_f, h(f') = h(f) \mod w$. Let s_f be the true spread of flow f. The number of distinct data items $\langle f, e\rangle$ that are actually recorded by $C[h(f)]$ is $\sum_{f' \in F_f} s_{f'}$, which is greater than or equal to s_f since $f \in F_f$. The noise with respect to flow f is $\sum_{f' \in F_f - \{f\}} s_{f'}$, which is what we want to remove.

3.2 Baseline Randomized Error-Reduction Sketch - rSkt

Our first solution, called randomized error-reduction sketch (rSkt), is to use two hash tables, C and $\bar{C}$, each of w spread estimators, as shown in Figure 4, where each flow f is hashed to a pair of candidate estimators, $C[h(f)]$ and $\bar{C}[h(f)]$. Let $g(.)$ be a function that maps f to 0 or 1 pseudo-randomly with equal probability. All elements of flow f will be recorded in $C[h(f)]$ if $g(f) = 0$ or in $\bar{C}[h(f)]$ if $g(f) = 1$. We call the estimator that records elements of f as the flow's *primary estimator* and the other as the flow's *complement estimator*. In practice, we may implement $g(f)$ by taking the least-order bit of hash value $h'(f)$ using another uniform hash function $h'(.)$ or taking the highest-order bit of $h(f)$ before modulo w.

Consider an arbitrary noise flow $f' \in F_f - \{f\}$, where F_f is the set of flows that $\forall f' \in F_f, h(f') = h(f) \mod w$. It is also either recorded in $C[h(f)]$ or $\bar{C}[h(f)]$, with equal probability, depending on the value of $g(f')$. Hence, it is either recorded in flow f's primary estimator or its complement estimator, with equal probability. Therefore, our solution splits error $F_f - \{f\}$ between f's primary estimator and its complement estimator, with equal probability. Roughly speaking, the flow's primary estimator records s_f and about half of the error, and its complement records about half of the error, allowing us to subtract error away. The flow's estimated spread, denoted as $\hat{s}_f$, is computed as follows.

$$\hat{s}_f = (1 - 2g(f))[V(C[h(f)]) - (V(\bar{C}[h(f)])] \tag{1}$$

We stress that the splitting operation of $F_f - \{f\}$ between $C[h(f)]$ and $\bar{C}[h(f)]$ will not be perfect and residual error will remain after subtraction. Due to the pseudo-randomness of hashing, $F_f - \{f\}$ may happen to contain a large flow f' that dominates in $F_f - \{f\}$. Even if the flows in $F_f - \{f\}$ are evenly distributed between $C[h(f)]$ and $\bar{C}[h(f)]$, the error caused by these flows is not evenly distributed between the two. Most error will go where f' goes. In this case, subtraction will not serve its purpose.

One approach to solve the above problem is to use d independent hash functions, $h_i(f)$, $0 \le i < d$, each mapping f to a pair of candidate estimators, $C[h_i(f)]$ and $\bar{C}[h_i(f)]$. We also use d independent pseudo-random functions, $g_i(f)$, $0 \le i < d$, each choosing a primary estimator from two candidates: For $0 \le i < d$, if $g_i(f) = 0$, all elements of f will be recorded in $C[h_i(f)]$; if $g_i(f) = 1$, all elements of f will be recorded in $\bar{C}[h_i(f)]$. Hence, for each received item $\langle f, e\rangle$, it will be recorded for d times, once in each of f's primary estimators. The recording and querying operations of rSkt are given in Algorithms 1 and 2, respectively.

Algorithm 1 Recording a data item in rSkt

```
1: Input: data item ⟨f, e⟩
2: Action: record e of f in hash tables C and C̄
3: for i = 0 to d − 1 do
4:     if g_i(f) = 0 then
5:         record ⟨f, e⟩ to estimator C[h_i(f)]
6:     else
7:         record ⟨f, e⟩ to estimator C̄[h_i(f)]
8:     end if
9: end for
```

Algorithm 2 Querying on a flow in rSkt

```
1: Input: flow label f, maximum integer value MAX_VALUE
2: Output: spread estimate
3: X=MAX_VALUE
4: for i = 0 to d − 1 do
5:     if X > V(C[h_i(f)]) + V(C̄[h_i(f)]) then
6:         X = V(C[h_i(f)]) + V(C̄[h_i(f)])
7:         x=i
8:     end if
9: end for
10: if g_x(f) = 0 then
11:     return V(C[h_x(f)]) − V(C̄[h_x(f)])
12: else
13:     return V(C̄[h_x(f)]) − V(C[h_x(f)])
14: end if
```

To estimate the spread of flow f, we first find the pair of candidate estimators, $C[h_x(f)]$ and $\bar{C}[h_x(f)]$, that has the smallest combined estimation, i.e.,

$$\exists x \in [0, d), V(C[h_x(f)]) + V(\bar{C}[h_x(f)]) = \min\{V(C[h_i(f)]) + V(\bar{C}[h_i(f)]), 0 \le i < d\}. \tag{2}$$

This is the pair that carries the least combined error in $F_f - \{f\}$ and is thus less likely to contain large flows. Finally, we estimate $\hat{s}_f$ as follows:

$$\hat{s}_f = (1 - 2g(f))[V(C[h_x(f)]) - (V(\bar{C}[h_x(f)])] \tag{3}$$

We find through experiments that by choosing $d > 1$, the estimation accuracy may be improved (in case of using HLL estimators) or may be worse (in case of using bitmap/FM estimators). The

reason is that although using the pair with the smallest combined error helps avoid large flows, each element is now recorded for d times in C and $\bar{C}$, which boost overall error among all estimators. Whether estimation accuracy becomes better or worse will depend on the joint impact of the above two factors.

There are two additional consequences of choosing $d > 1$. One is that we will have to compute the estimation formula $V(.)$ for $2d$ times, increasing the query overhead. The other is that we will have to record each data item d times, increasing the recording overhead.

Next we will introduce a new randomized error reduction design that not only improves estimation accuracy for all bitmap/FM/HLL types to a level that rSkt cannot achieve with any d value, but also does so with low query overhead of computing $V(.)$ only twice and with low processing overhead of recording each data item just once.

3.3 Unit-level Randomized Error-Reduction Sketch - rSkt2

We use an example to illustrate the idea behind our second design, referred to as rSkt2. Consider the baseline sketch rSkt with $d = 1$. A flow f is hashed to $C[h(f)]$ and $\bar{C}[h(f)]$. Without loss of generality, suppose that $g(f) = 0$ and f is recorded in $C[h(f)]$. Suppose there are only two flows, f and f', in F_f. Flow f' is a flow of large spread. There are two possible cases.

Case 1: f' is recorded in $C[h(f)]$. Because all elements of f and f' are recorded in $C[h(f)]$, $V(C[h(f)])$ is an estimate of the combined spread of f and f'. Because no element is recorded in $\bar{C}[h(f)]$, $V(\bar{C}[h(f)]) = 0$. Hence, the estimate by (1) becomes $\hat{s}_f = V(C[h(f)]) - V(\bar{C}[h(f)]) = V(C[h(f)])$, which carries large positive error introduced by f'.

Case 2: f' is recorded in $\bar{C}[h(f)]$. Because all elements of f are recorded in $C[h(f)]$, $V(C[h(f)])$ is an estimate of the spread of f. Because all elements of f' are recorded in $\bar{C}[h(f)]$, $V(\bar{C}[h(f)])$ is an estimate of the spread of f'. As $\hat{s}_f = V(C[h(f)]) - V(\bar{C}[h(f)])$, it is the estimated spread of f minus the estimated spread of f', thus carrying large negative error introduced by f'.

To resolve the above dilemma, we have to look deeper at $C[h(f)]$ and $\bar{C}[h(f)]$ into their unit-level structures and break up f' into pieces such that half of the pieces are stored with f and half are stored away from f, allowing them to be subtracted away. In fact, we need to break up every flow in such a way because any flow has a potential to cause error to other flows due to hash collision. Below we describe how $rSkt2$ will record the elements of an arbitrary flow f differently from $rSkt$.

Recall from Section 2.2 that each estimator in the hash table C (or $\bar{C}$) is an array of m units, which may be bits, FM registers or HLL registers. Flow f is hashed to a pair of estimators, $C[h(f)]$ and $\bar{C}[h(f)]$. Different from rSkt, our new idea will not use either of them to record f in its entirety. Instead, we construct a logical primary estimator L_f from the units of $C[h(f)]$ and $\bar{C}[h(f)]$ to record f. L_f is also an array of m units. Its ith unit is taken from the ith unit of either $C[h(f)]$ or $\bar{C}[h(f)]$, with equal probability. Let $g'(f, i)$ be a pseudo-random function taking two input parameters i and f and returning a bit, 0 or 1, with equal probability, where $0 \le i < m$. We define

$$L_f[i] \equiv \begin{cases} C[h(f)][i], & \text{if } g'(f,i) = 0 \\ \bar{C}[h(f)][i], & \text{if } g'(f,i) = 1 \end{cases} \quad (4)$$

When we receive an element e of flow f, it is recorded as usual in $L_f[h(e)]$, which is $C[h(f)][h(e)]$ if $g'(f, h(e)) = 0$ or $\bar{C}[h(f)][h(e)]$ if $g'(f, h(e)) = 1$. The actual recording operation is explained in Section 2.2. The recording and querying operations of rSkt2 are formally presented in Algorithms 3 and 4, respectively.

Algorithm 3 Recoding a data item in rSkt2

```
Input: data item ⟨f, e⟩
Action: record e of f in hash tables C and C̄
if g'(f, h(e)) = 0 then
    record ⟨f, e⟩ to C[h(f)][h(e)]
else
    record ⟨f, e⟩ to C̄[h(f)][h(e)]
end if
```

Algorithm 4 Querying on a flow in rSkt2

```
Input: flow label f
Output: spread estimate
for i= 0 to m do
    if g'(f, i) = 0 then
        L_f[i] = C[h(f)][i]
        L̄_f[i] = C̄[h(f)][i]
    else
        L_f[i] = C̄[h(f)][i]
        L̄_f[i] = C[h(f)][i]
    end if
end for
return V(L_f) − V(L̄_f)
```

The logical complement estimator of flow f, denoted as $\bar{L}_f$, is constructed from the units that L_f does not use.

$$\bar{L}_f[i] \equiv \begin{cases} \bar{C}[h(f)][i], & \text{if } g'(f,i) = 0 \\ C[h(f)][i], & \text{if } g'(f,i) = 1 \end{cases} \quad (5)$$

Consider an arbitrary flow $f' \in F_f - \{f\}$, where $h(f') = h(f)$ by definition. The flow is recorded in its own logical primary estimator $L_{f'}$ that is constructed similarly from the units of $C[h(f)]$ and $\bar{C}[h(f)]$. Each unit from $C[h(f)]$ or $\bar{C}[h(f)]$ has 50% chance to be in L_f and also independently 50% chance to be in L'_f. Hence, when an element e' of f' is recorded in a unit of L'_f, it has 50% chance to be in L_f as well because that unit has 50% chance to be in L_f. By the same token, element e' has 50% chance to be in $\bar{L}_f$. Hence, we are successful in splitting f' to two halves. One half is stored in L_f, and the other half in $\bar{L}_f$, allowing us to subtract them away. We estimate the spread of flow f based on its logical primary estimator and the logical complement estimator as follows:

$$\hat{s}_f = V(L_f) - V(\bar{L}_f), \quad (6)$$

which not only solves the accuracy problem raised at the beginning of this subsection, but does so with a low query overhead of computing $V(.)$ only twice. Moreover, each element is recorded for d times in rSkt, and it is recorded just once in rSkt2. This smaller processing overhead allows rSkt2 to handle an incoming stream of data items at higher throughput.

3.4 Multiple Streams, Super Spreaders, Flow Labels

Spread estimation for multiple streams. The proposed randomized error reduction sketches can be deployed on multiple locations to jointly measure multiple data streams concurrently [8, 46]. For example, it may be deployed on multiple routers to support network-wide flow spread monitoring. Suppose there are k measurement points in a network, each running an instance of rSkt2 with the same parameter setting, e.g., d, w and hash functions. The recorded hash tables, C_j and $\bar{C}_j$, $0 \le j < k$, are sent to a central controller for merging together into two tables, C_* and $\bar{C}_*$. The merge operation is dependent on the estimator type.

- bitmap or FM: Bitmap/FM estimators from k routers, $C_j[i]$, $0 \le i < k$, are merged to $C_*[i]$ by bitwise OR. Similarly, $\bar{C}_j[i]$, $0 \le i < k$, are merged to $\bar{C}_*[i]$ by bitwise OR.
- HLL: HLL estimators from k routers, $C_j[i]$, $0 \le i < k$, are merged to $C_*[i]$ by taking the maximum unit values, i.e., $C_*[i][z] = \max\{C_j[i][z], 0 \le j < k\}$, $0 \le z < m$. Likewise, $\bar{C}_*[i][z] = \max\{\bar{C}_j[i][z], 0 \le j < k\}$, $0 \le z < m$.

After merging, spread estimation is performed on C_* and $\bar{C}_*$ as described earlier in the section.

There exists unified frameworks for implementing diverse measurement tasks by combining basic building blocks [22, 40]. One example is OpenSketch [42]. Our sketch can serve a building block in such a framework to implement flow spread estimation. In case of OpenSketch, without changing the hash stage and classification stage, we can use two consecutive bitmaps in its framework to implement rSkt2 (with bitmap estimators). We can also replace the bitmaps with FM/HLL estimators as plug-ins to fit in the framework.

Super spreader detection. The low query overhead makes rSkt2 ideal for supporting real-time detection of super spreaders, where queries are made as we receive data items. Similar to rSkt, most prior art on super spreader detection requires using $O(d)$ estimators and computing $O(d)$ estimations per query [31, 42, 46], comparing with $O(1)$ of rSkt2. Because their query overhead for spread estimation is higher than the recording overhead, we will not be able to query the spread of f for each received $\langle f, e\rangle$ for super spreader detection, but instead perform sampling on the received items to query at a lower rate. The smaller query overhead of rSkt2 will allow it to perform queries at a higher rate and therefore improve on real-time detection of super spreaders.

Discussion about flow labels. In many applications, the flow labels are pre-known. For example, if we monitor the data flows from all hosts in a data center, the flow labels are host addresses which are known. In other cases, we do not need to keep flow labels. For example, we perform real-time super spreader detection by querying on the flow labels from the received data items. Some prior art [31] extends each estimator to include a flow label field, which tracks the flow that is estimated to have the largest spread among all flows hashed to this estimator. This idea can be incorporated into our design quite easily. We may also use a separate data structure to keep flow labels if needed — for example, using a hash map or a heap to store the flow labels of super spreaders that have been detected.

4 ANALYSIS

Let s_f be the actual spread of flow f, $\hat{s}_f$ be the spread estimate of flow f, S be the total number of distinct items $\langle f, e\rangle$ in the data stream, and F_f be the set of flows f' such that $h(f') = h(f)$ in case of rSkt2 or that $\exists i \in [0, d)$, $h_i(f') = h_i(f)$, in case of rSkt. The data items in flow $f' \in F_f - \{f\}$ are called *error data items* with respect to f.

Theorem 4.1. *For any given flow f, the expectation of $\hat{s}_f$ produced by rSkt/rSkt2 satisfies*

$$|E(\hat{s}_f) - s_f| \le \begin{cases} o(s_f) + o(\frac{S-s_f}{2w}), \text{ if using HLL/FM estimators;} \\ \frac{s_f}{2m} + \frac{1}{2}(e^{\frac{s_f}{m}} - 1) + o(\frac{S-s_f}{2w}), \\ \text{if using bitmap estimators.} \end{cases}$$

The proof can be found in the supplementary materials and [35]. Note that w is a large value: when using HLL estimators with 128 units and allocated 10Mb memory, w is about 8k. The bound for expectation of the spread estimate produced by rSkt/rSkt2 can be small when f's spread is large. For instance, $\frac{S-s_f}{2w}$ represents the average error in each estimator. If it is smaller than s_f, the bound will become $o(s_f) \ll s_f$. Consider a special case where each flow is allocated an estimator and $2w$ approaches the number of flows. If using HLL estimators, the bound is $o(s_f) + o(\frac{S-s_f}{2w})$. $\frac{S-s_f}{2w}$ is equal to the average flow spread among all flows. Under this circumstance, $|E(\hat{s}_f) - s_f| \le o(s_f)$ if f' spread is above the average flow spread, which is much smaller compared to its actual flow spread.

Theorem 4.2. *For any given flow f, the variance of the spread estimate $\hat{s}_f$ from rSkt can be derived as*

$$Var(\hat{s}_f) = \begin{cases} \frac{1.04^2}{m}(s_f^2 + \frac{(S-s_f)s_f}{w} + T_f) + Q_f + o(s_f^2 + Q_f), \\ \text{if using HLL estimators;} \\ \frac{0.78^2}{m}(s_f^2 + \frac{(S-s_f)s_f}{w} + T_f) + Q_f + o(s_f^2 + Q_f), \\ \text{if using FM estimators;} \\ (T_f - (\frac{S-s_f}{2w})^2)(\lambda_1+\lambda_5)^2 + \lambda_4 + \lambda_8 + (\lambda_1+\lambda_5)o(\frac{Q_f}{w}) \\ \text{if using bitmap estimators.} \end{cases} \quad (7)$$

where $\beta = S - s_f$, $Q_f = \sum_{f' \ne f} \sum_{f'' \ne f', f'' \ne f} s_{f'} s_{f''} (1 - (1 - \frac{1}{w})^d)^2 + \sum_{f' \ne f} s_{f'}^2 (1 - (1 - \frac{1}{w})^d)$, $T_f = \sum_{f' \ne f} \sum_{f'' \ne f', f'' \ne f} \frac{s_{f'} s_{f''}}{4} (1 - (1 - \frac{1}{w})^d)^2 + \sum_{f' \ne f} \frac{s_{f'}^2}{2} (1 - (1 - \frac{1}{w})^d)$, $\lambda_1 = (\frac{e^{\frac{s_f + \frac{\beta}{2w}}{m}} - 1}{2m} + 1)$, $\lambda_4 = m(e^{\frac{s_f + \frac{\beta}{2w}}{m}} - \frac{s_f + \frac{\beta}{2w}}{m} - 1)$, $\lambda_5 = (\frac{e^{\frac{\beta}{2mw}} - 1}{2m} + 1)$, *and* $\lambda_8 = m(e^{\frac{\beta}{2mw}} - \frac{\beta}{2mw} - 1)$.

Theorem 4.3. *For any given flow f, the variance of the spread estimate $\hat{s}_f$ from rSkt2 can be derived as*

$$Var(\hat{s}_f) = \begin{cases} \frac{1.04^2}{m}(s_f^2 + \frac{\beta(s_f + \frac{1}{2})}{w} + \frac{R_f}{2}) + \frac{\beta}{w} + o(s_f^2 + R_f), \\ \text{if using HLL estimators;} \\ \frac{0.78^2}{m}(s_f^2 + \frac{\beta(s_f + \frac{1}{2})}{w} + \frac{R_f}{2}) + \frac{\beta}{w} + o(s_f^2 + R_f), \\ \text{if using FM estimators;} \\ \frac{S-s_f}{4w}(\lambda_1 + \lambda_5)^2 + \lambda_4 + \lambda_8 + (\lambda_1 + \lambda_5)o(\frac{S}{w}), \\ \text{if using bitmap estimators.} \end{cases} \quad (8)$$

where $R_f = \sum_{f' \neq f} \sum_{f'' \neq f', f'' \neq f} \frac{s_{f'} s_{f''}}{w^2} + \sum_{f' \neq f} \frac{s_{f'}^2}{w}$, *and* $\beta, \lambda_1, \lambda_4, \lambda_5, \lambda_8$ *are the same as those in Theorem 4.2.*

The proof of Theorems 4.2 and 4.3 is deferred to the supplementary materials and [35].

Interpretaion of Theorems 4.2 and 4.3 After the rigorous derivation of the variances of the spread estimate produced by rSkt/rSkt2. We interprete Theorems 4.2 and 4.3 with some approximation. We first do approximation on R_f.

$$\begin{aligned} R_f &= \sum_{f' \neq f} \sum_{f'' \neq f', f'' \neq f} \frac{s_{f'} s_{f''}}{w^2} + \sum_{f' \neq f} \frac{s_{f'}^2}{w} \\ &= \sum_{f' \neq f} \sum_{f'' \neq f', f'' \neq f} \frac{s_{f'} s_{f''}}{w^2} + \sum_{f' \neq f} \frac{s_{f'}^2}{w^2} - \sum_{f' \neq f} \frac{s_{f'}^2}{w^2} + \sum_{f' \neq f} \frac{s_{f'}^2}{w} \\ &\leq \frac{(\sum_{f' \neq f} s_{f'})^2}{w^2} + n(\frac{S - s_f}{n})^2 \frac{1}{w}) \leq 2(\frac{S - s_f}{w})^2 \end{aligned} \tag{9}$$

where n is the number of flows. The last inequality holds as $n \geq w$ usually holds in practical settings. Similarly, we can do approximation on Q_f and T_f.

$$Q_f \approx ((S - s_f)(1 - (1 - \frac{1}{w})^d))^2 \approx 2(\frac{d(S - s_f)}{w})^2 \tag{10}$$

$$T_f \approx (\frac{d(S - s_f)}{w})^2 \tag{11}$$

Consider approximation on λ_1. $s_f + \frac{\beta}{2w}$ represents the expected spread stored in f' primary estimator, which is $O(m)$ as the estimation upper bound of bitmap (also called linear counting) is linear to the bitmap length m. Therefore, we have

$$\lambda_1 = (\frac{e^{\frac{s_f + \frac{\beta}{2w}}{m}} - 1}{2m} + 1) \approx (\frac{1 + \frac{s_f + \frac{\beta}{2w}}{m} - 1}{2m} + 1) \approx 1$$

Doing the similar approximation on λ_5, λ_4, and λ_8, we have

$$\begin{aligned} \lambda_5 &\approx 1 \\ \lambda_4 &= m(e^{\frac{s_f + \frac{\beta}{2w}}{m}} - \frac{s_f + \frac{\beta}{2w}}{m} - 1) \approx \frac{(s_f + \frac{\beta}{2w})^2}{2m} \\ \lambda_8 &= m(e^{\frac{\beta}{2mw}} - \frac{\beta}{2mw} - 1) \approx \frac{1}{2} m(\frac{\beta}{2mw})^2 \approx 0 \end{aligned}$$

From the above approximations, we give a concise version of the variance of estimate produced by rSkt/rSkt2.

• *For any given flow* f*, the rigorous variance of the spread estimate* $\hat{s}_f$ *from rSkt in* (7) *can be approximately bounded as*

$$Var(\hat{s}_f) \leq \begin{cases} \frac{1.04^2}{m}(s_f + \frac{d(S-s_f)}{w})^2 + 2(\frac{d(S-s_f)}{w})^2, \\ \text{if using HLL estimators;} \\ \frac{0.78^2}{m}(s_f + \frac{d(S-s_f)}{w})^2 + 2(\frac{d(S-s_f)}{w})^2, \\ \text{if using FM estimators;} \\ \frac{1}{2m}(s_f + \frac{S-s_f}{2w})^2 + 4(4d^2 - 1)(\frac{(S-s_f)}{2w})^2, \\ \text{if using bitmap estimators.} \end{cases} \tag{12}$$

• *For rSkt2, variance in* (8) *can be approximately bounded as*

$$Var(\hat{s}_f) \leq \frac{c^2}{m}(s_f + \frac{(S - s_f)}{w})^2 + \frac{(S - s_f)}{w} \tag{13}$$

where c is 1.04, 0.78, and $1/\sqrt{2}$ *if using HLL, FM, and bitmap estimators, respectively.*

Comparing (13) with (12), we can find the variance of spread estimate produced by rSkt2 is smaller than that of rSkt. Consider (13) for rSkt2. $\frac{S - s_f}{w}$ can be interpreted as the average error in the each pair of candidate estimators. When the spread of flow f is far larger than the average error, the variance is bounded by $\frac{c^2}{m}(s_f)^2 + \frac{(S-s_f)}{w}$. When the spread of flow f is far smaller than the average error, the variance is bounded by $\frac{(S-s_f)}{w}$.

5 PERFORMANCE EVALUATION

We evaluate the performance of the proposed rSkt and rSkt2 on both hardware and software platforms through experiments based on real-world data traces. We also compare them with the state of the art under various performance metrics. In addition, we perform an application case study on super spread detection in comparison with the prior art.

5.1 Implementation

We have implemented the following sketches: (1) the proposed sketches, rSkt and rSkt2, (2) the state-of-the-art prior work that performs per-flow spread estimation, bSkt [46] and cSkt-CM [8, 46], and (3) the state-of-the-art prior work that performs super spreader detection, SS [31]. SS uses multi-resolution bitmaps [11]. The other sketches can work with bitmaps, FM estimators, and HLL estimators, which are explained in Section 2.2. With different estimators, they are denoted respectively as rSkt(bitmap), rSkt(FM), and rSkt(HLL); rSkt2(bitmap), rSkt2(FM), and rSkt2(HLL); bSkt(bit-map), bSkt(FM), and bSkt(HLL); cSkt-CM(bitmap), cSkt-CM(FM), and cSkt-CM(HLL). Our implementation is done on three platforms.

• CPU Implementation: This is software implementation. The experiments are performed on a computer with Intel Core Xeon W-2135 3.7GHz and 32 GB memory.

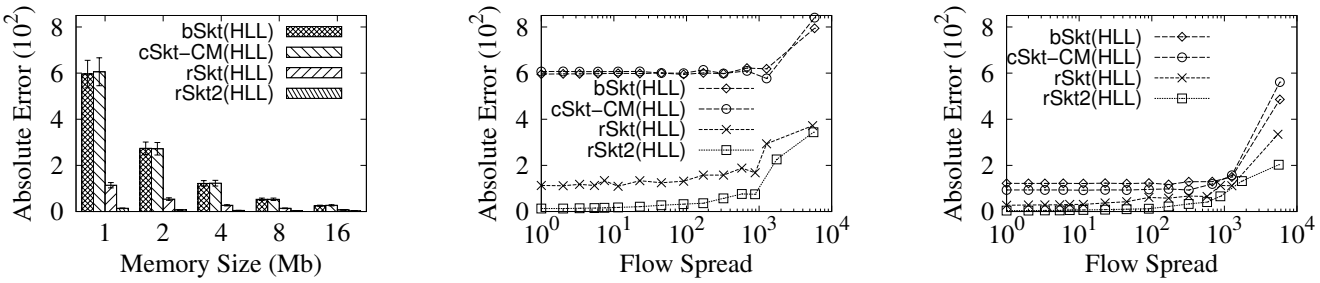
• Implementation: GPU has become cheaper and widely available. We find that it serves well as a low-cost accelerator for software implementation. With CUDA toolkit, all sketches are programmed to support parallel execution on a computer equipped with GeForce GTX 1070, 8GB GDDR5 memory and 1920 CUDA cores, each at a rate of 1506-1683 MHz clock rate.

• FPGA Implementation: This is hardware implementation. All sketches are implemented on XILINX NEXYS 4DDR/A7 -100T FPGA platform, with 128MB DDR2 DRAM, 4860Kb Block RAM, and 100MHz clock rate.

5.2 Datasets

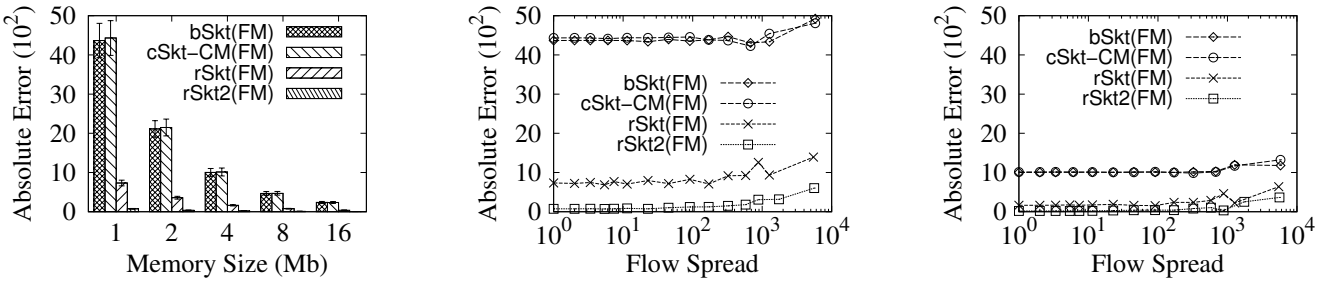
We conduct the evaluation using two real-world datasets.

CAIDA dataset:The data streams used in our evaluation are real Internet traffic traces downloaded from CAIDA [32]. We use 10 traces, each of tens of millions of packets. Each experiment is performed over these 10 data streams independently, and we present



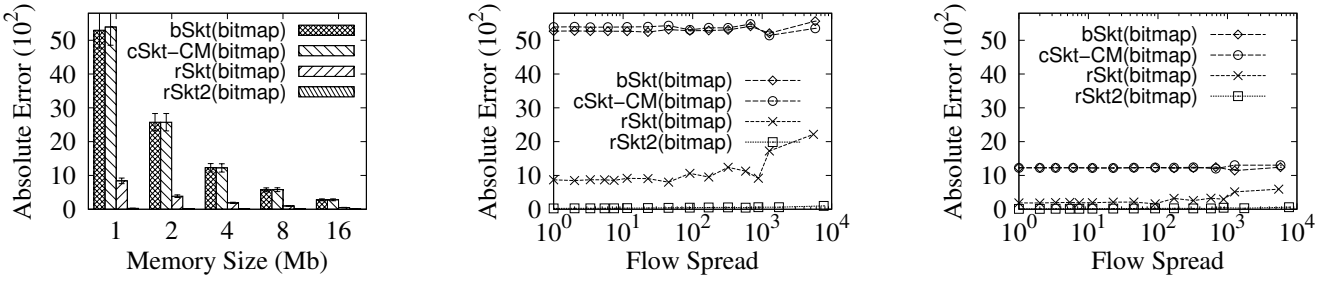
(a) Various memory allocations (b) 1Mb Memory (c) 4Mb Memory

Figure 5: Accuracy comparison of rSkt(HLL), rSkt2(HLL), bSkt(HLL) and cSkt-CM(HLL) under CAIDA dataset. (a) Average absolute error of all flows w.r.t memory size, (b)-(c) error distribution under a given memory size. Compared to the prior art bSkt(HLL) and cSkt-CM(HLL), the proposed rSkt(HLL) and rSkt2(HLL) reduce absolute error by 73.6%-81.1% and 93.9%-97.8%, respectively, in plot (a).



(a) Various memory allocations (b) 1Mb Memory (c) 4Mb Memory

Figure 6: Accuracy comparison of rSkt(FM), rSkt2(FM), bSkt(FM) and cSkt-CM(FM) under CAIDA dataset. (a) Average absolute error of all flows w.r.t memory size, (b)-(c) error distribution under a given memory size. Compared to the prior art bSkt(FM) and cSkt-CM(FM), the proposed rSkt(FM) and rSkt2(FM) reduce absolute error by 83.5%-84.5% and 97.9%-98.4%, respectively, in plot (a).

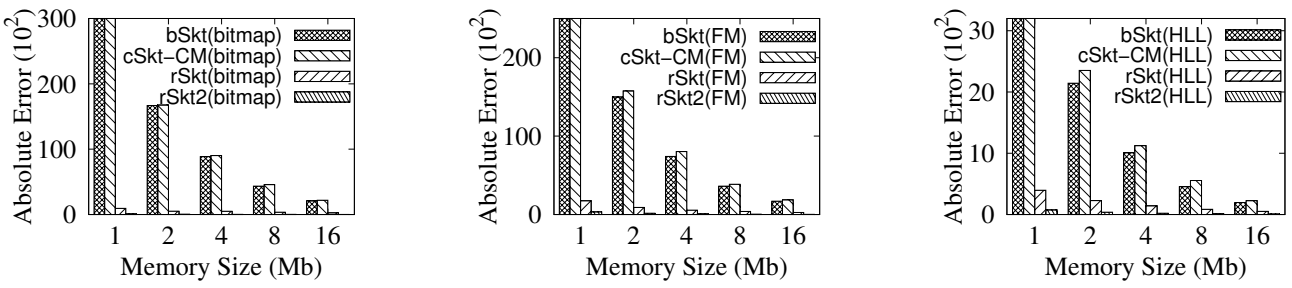


(a) Various memory allocations (b) 1Mb Memory (c) 4Mb Memory

Figure 7: Accuracy comparison of rSkt(bitmap), rSkt2(bitmap), bSkt(bitmap) and cSkt-CM(bitmap) under CAIDA dataset. (a) Average absolute error of all flows w.r.t memory size, (b)-(c) error distribution under a given memory size. Compared to the prior art bSkt(bitmap) and cSkt-CM(bitmap), the proposed rSkt(bitmap) and rSkt2(bitmap) reduce absolute error by 83.0%-84.5% and 98.7%-99.5%, respectively, in plot (a).

the average results. Flow label f is defined as destination address carried in each packet's header. Each trace contains around 110k flows and around 400k distinct data items. Element e is source address also from packet header. All packets toward the same destination form a flow. Flow spread is the number of distinct sources that communicate with a destination. Anomaly in flow spread may signal flash crowd in service requests or denial-of-service attack against a destination service (which could be judged in conjunction with flow size); both cases will require immediate attention from service admin.

E-commerce dataset: The dataset is collected from a real-world e-commerce website [1], which contains three files and we use the visitor behavior data. Each row in the file is a product view record with particular properties: visitor ID, timestamp, item ID and so on. There are totally about 1.4M visitors, and 235k items. Flow label f is defined as item ID and element e is visitor ID. The number of distinct items $\langle f, e\rangle$ is about 1.2M. All view records of the same item form a flow. Flow spread is the number of distinct visitors viewing the item, which reflects the popularity of the product.



(a) bitmap (b) FM (c) HLL

Figure 8: Average absolute error of rSkt, rSkt2, bSkt and cSkt-CM w.r.t. memory size using bitmap, FM, and HLL estimators, respectively, under E-commerce dataset. Compared to the prior art bSkt and cSkt-CM, the proposed rSkt and rSkt2 reduce absolute error by 74.4%-97.3% and 96.5%-99.7%, respectively.

5.3 Experimental Setting

For rSkt(bitmap), rSkt2(bitmap), bSkt(bitmap) and cSkt-CM(bitmap), we set the bitmap size to be 5000 bits, which produces a spread estimation range that covers all flows in the traces. The bitmap size of SS is chosen according to the original paper [31]. For rSkt(FM), rSkt2(FM), bSkt(FM) and cSkt-CM(FM), each register is 32 bits. For rSkt(HLL), rSkt2(HLL), bSkt(HLL) and cSkt-CM(HLL), each register is 5 bits. The number of registers in each estimator is 128. For rSkt, bSkt and cSkt-CM, $d = 4$. Namely, each data item is recorded in four estimators and each query requires estimation from four estimators; see Section 3.2. The above parameter setting is in line with those in [46].

The sketches are evaluated and compared under the following three performance metrics.

• *Estimation Accuracy.* We use absolute error to measure estimation accuracy. Let $\hat{s}_f$ be the estimated spread of flow f, and s_f be the actual spread of flow f. The absolute error is calculated as $|\hat{s}_f - s_f|$, and the average absolute error is defined as $\sum_f |\hat{s}_f - s_f|/N$, where N is the number of flows in the data stream.

• *Recording Throughput.* We measure the rate at which the data items $\langle f, e\rangle$ are recorded by each sketch on any given software/hardware platform. The unit is million data per second, abbreviated as Mdps.

• *Online Query Throughput.* We measure the rate at which the queries can be performed on f after each item $\langle f, e\rangle$ from a stream is recorded. For each query, we produce an estimate of f's spread up to the time when the query is performed.

5.4 Estimation Accuracy

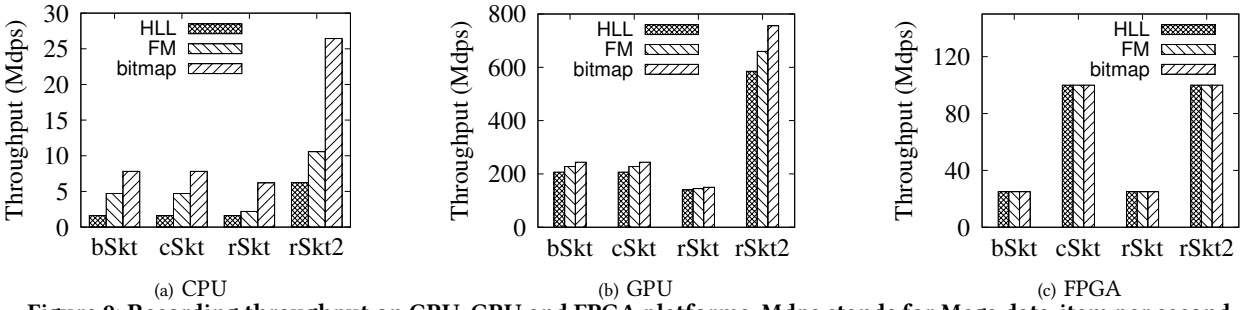
Our first set of experiments compare the proposed sketches with the state of the art in terms of estimation accuracy. Note that accuracy is the same across different implementation platforms, which only affect throughput. We begin by comparing rSkt(HLL), rSkt2(H-LL), bSkt(HLL) and cSkt-CM(HLL) using CAIDA dataset. Figure 5 (a) shows the average absolute error among all flows under 1Mb-16Mb memory allocations to each sketch. In contrast, if we ideally assign each flow a single-flow spread estimator, it needs 70Mb/450Mb/550Mb memory using HLL, FM, and bitmap estimators, respectively. bSkt(HLL) and cSkt-CM(HLL) performs similarly. Compared to the better one of them, rSkt(HLL) reduces average error by more than 73.6%, and rSkt2(HLL) reduces average error by more than 93.9%. Figures 5(b)-5(c) show the detailed error distribution at a given memory allocation, 1Mb and 4Mb, respectively. The flows are placed in bins based on their true spreads (which can be found directly from the traffic traces). The spread bins are $[2^i, 2^{i+1}]$ for $i \geq 0$. We average the absolute error of flows in each bin and plot a point in the figure.

In Figure 5(a), when memory allocation increases, the average absolute error of rSkt(HLL), rSkt2(HLL), bSkt(HLL) or cSkt-CM(H-LL) decreases, which is expected because the probability of hash collision decreases. The figure shows that rSkt and rSkt2 are much more accurate than bSkt(HLL) and cSkt-CM(HLL), especially under tight memory. For example, when 1Mb memory is used, rSkt(H-LL) and rSkt2(HLL) reduce the average absolute error by 81.1% and 97.8%, respectively, compared to bSkt(HLL). Figure 5(a) also shows the error bars of average absolute error of rSkt(HLL), rSkt2(HLL), bSkt(HLL) and cSkt-CM(HLL) under 10 traces. As we can see, the advantages of rSkt(HLL), rSkt2(HLL) over bSkt(HLL) and cSkt-CM(HLL) in terms of estimation accuracy hold under different traces.

Figures 5(b)-5(c) show that absolute error is larger for flows of larger spreads. The proposed rSkt(HLL) and rSkt2(HLL) have much smaller error distributions than bSkt(HLL) and cSkt-CM(HLL), thanks to their randomized error reduction design. rSkt2(HLL) is more accurate than rSkt(HLL) due to its logical estimator design that splits noise flows into pieces. Its improvement over rSkt will be more pronounced when we use FM estimators and bitmaps below and when we consider throughput shortly.

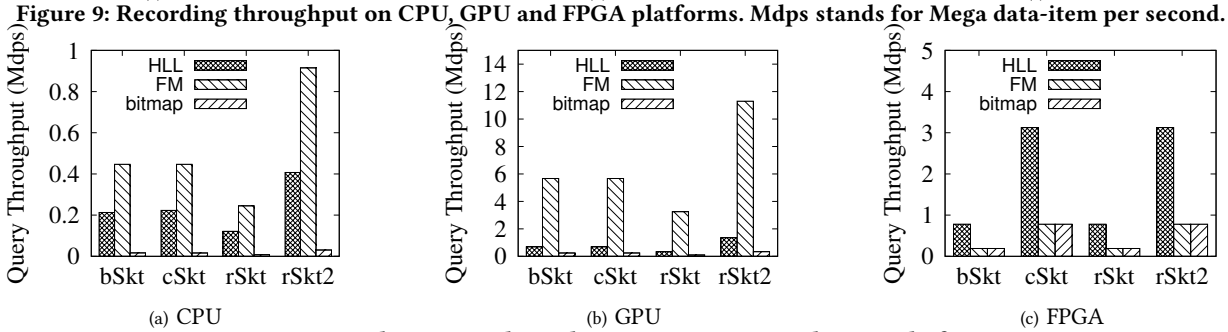
The experimental results that compare rSkt(FM), rSkt2(FM), bSkt-(FM) and cSkt-CM(FM) are shown in Figure 6. The results that compare rSkt(bitmap), rSkt2(bitmap), bSkt(bitmap) and cSkt-CM(bit-map) are shown in Figure 7. Similar conclusion can be drawn as those from Figure 5. For example, from Figure 6(a), using bSkt(FM) as a baseline, rSkt(FM) reduces average error by more than 83.5%, and rSkt2(FM) reduces average error by more than 97.9%. From Figure 7 (a), using bSkt(bitmap) as a baseline, rSkt(bitmap) reduces average error by more than 83.0%, and rSkt2(bitmap) reduces average error by more than 98.7%. From error distributions in Figure 6 (b)-(c) and Figure 7(b)-(c), rSkt2 performs consistently better than rSkt, which is in turn much better than bSkt and cSkt-CM.

We also conduct experiments using the E-commerce dataset. The average absolute errors of rSkt, rSkt2, bSkt, and cSkt-CM using bitmap, FM, and HLL estimators respectively are shown in Figures



(a) CPU (b) GPU (c) FPGA

Figure 9: Recording throughput on CPU, GPU and FPGA platforms. Mdps stands for Mega data-item per second.



(a) CPU (b) GPU (c) FPGA

Figure 10: Online query throughput on CPU, GPU and FPGA platforms.

8(a)-(c). Similar to the results under the CAIDA dataset, the results in Figure 8 also show that the proposed randomized sketches, especially rSkt2, can significantly reduce the average absolute error compared to bSkt and cSkt-CM. The figures that show the detailed error distribution with respect to actual flow spread are similar to those under the CAIDA dataset and we do not repeatedly display them due to space limit.

5.5 Recording Throughput

Our second set of experiments compare the proposed sketches with the state of the art in terms of recording throughput (at which rate the incoming data items can be processed on different platforms). The experimental results of recording throughput on the CPU platform are shown in Figure 9(a). The recording throughput of rSkt2 is highest for any type of estimators (i.e., bitmap, FM and HLL) because it records each data item just once, whereas the other three sketches records each item d times. The throughputs of bSkt and rSkt are similar, while that of rSkt is slightly lower due to computing an additional function g. As example, the throughput of rSkt2(HLL) is 2.26 times that of bSkt(HLL) or cSkt-CM(HLL), and it is 3.88 times that of rSkt(HLL). For all sketches, the highest throughput is achieved when bitmaps are used. That is because FM and HLL require an additional geometric hash operation; see Section 2.2. The throughput is lowest when HLL is used because it incurs more memory accesses.

The experimental results of recording throughput on the GPU platform are shown in Figure 9(b). All sketches achieve much higher throughput on GPU than on CPU due to massive parallelism. Still, rSkt2 achieves much higher throughput, around 600 Mdps, about three that of bSkt or cSkt-CM and about four times that of rSkt.

The recording throughput of the sketches on FPGA is shown in Figure 9(c). The proposed rSkt2 achieves a throughput of 100 Mdps, while bSkt and rSkt only support a throughput of 25 Mdps. This is because bSkt and rSkt record each data item four times in the same memory block, which consumes four clock cycles, whereas rSkt2 records each data item once in one clock cycle (with hardware pipelining). Interestingly, cSkt-CM also achieves 100 Mdps because it uses d arrays, which can be placed on different memory blocks, allowing parallel access. Due to pipelining, each sketch achieves the same throughput under different estimator types (bitmap, FM and HLL). One may observe that hardware implementation on FPGA achieves smaller throughput than software implementation on GPU. There are two reasons. First, GPU allows massive parallelism which compensates the software disadvantage. Second, our FPGA is a cheap one. Throughput will be higher if a high-end FPGA is used. We conclude that GPU is a viable alternative to hardware implementation for high throughput.

5.6 Query Throughput

Our third set of experiments compare the proposed sketches with the state of the art in terms of query throughput. We want to stress that the computation of spread estimation is nothing similar to that of size estimation [17, 21, 22, 45]. The latter incurs similar overhead as recording, and therefore its query throughput is similar to recording throughput. But spread estimation is much more computation intensive than recording, and spread query throughput is much smaller than recording throughput. Queries cannot be performed on per data item basis, which makes it practically important to design novel sketches that improve on query throughput.

The experimental results of query throughput on the CPU platform are shown in Figure 10(a). As is expected, the query throughput of rSkt2 is highest for any type of estimators (i.e., bitmap, FM and HLL) because it computes two estimators per query, whereas bSkt and cSkt-CM each compute from d estimators per query, while rSkt computes $2d$ estimators. Because $d = 4$ in our experiments, the throughput of rSkt2 is expected to be about twice that

Table 3: Number of true super spreaders under different super spreader threshold.

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

Table 4: Number of false positives.

Threshold	200	300	400	500	600	700	800
rSkt2(bitmap)	43	15	9	3	4	2	0
SS	1122	113	67	27	19	15	6

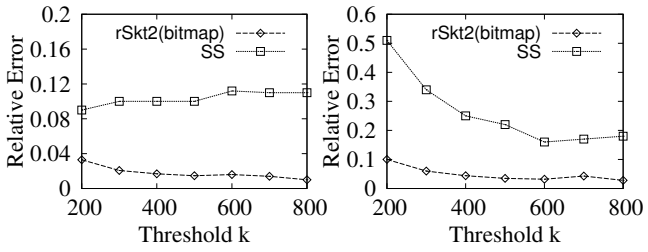
Table 5: Number of false negatives.

Threshold	200	300	400	500	600	700	800
rSkt2(bitmap)	6	2	1	0	0	0	0
SS	12	6	2	1	0	0	0

of bSkt (or cSkt-CM) and about four times that of rSkt, matching well with the experimental results.

The experimental results of recording throughput on the GPU platform are shown in Figure 10(b). Again, GPU is a great accelerator thanks to its numerous cores that process in parallel. The query throughput on GPU is more than an order of magnitude higher than that on CPU across different sketches and different types of estimators. Yet, relative performance between sketches remain similar. The query throughput of rSkt2 is about twice that of bSkt (or cSkt-CM) and about four times that of rSkt.

The complexity of query computation is far greater than that of recording, particularly for FM estimators and HLL estimators, which prevent us from implementing query solely on the FPGA board that we have. Instead, we implement a module that, upon query, will output the estimators of the flow, from which one can compute spread estimation off-board by software (which may be GPU-accelerated) or by ASIC hardware. The throughput of this FPGA module is shown in Figure 10(c). Both cSkt-CM and rSkt2 achieve higher throughput, thanks to pipelining, as the design of cSkt-CM allows parallel access to its d estimators per flow on FPGA, while rSkt2 also allows parallel access to its 2 estimators per flow. Both bSkt and rSkt have lower throughput because their designs do not allow fully parallelized access to multiple estimators per flow on FPGA.



(a) 16Mb memory (b) 2Mb memory

Figure 11: rSkt2(bitmap) incurs much smaller relative error than SS, particularly when memory allocation is small

5.7 Case Study: Super Spreader Detection

We use a case study to investigate how the proposed sketches perform in detecting super spreaders, which are defined as the flows whose spreads exceed a threshold k that the user chooses based on application need. We have shown that the proposed rSkt2 outperforms the state of the art on per-flow spread estimation. This case study is different. It is to identify super spreaders only and estimate their spreads. In this experiment, we compare rSkt2(bitmap) with the state-of-the-art sketch for this purpose, SpreadSketch (SS) [31], which use multi-resolution bitmaps. It is a modified version of cSkt-CM, with each estimator expanded for storing a flow label. With online queries, if the estimated spread of a flow after element record exceeds k, we keep the flow label in a hash map. The parameter setting can be found in Section 5.3. We evaluate the performance with three metrics.

- *Average relative error*, which is defined as $\sum_{f \in \Gamma_s} \frac{|\hat{s}_f - s_f|}{s_f \cdot |\Gamma_s|}$, where Γ_s is the set of super spreaders detected.
- *Number of false positives*, i.e., the number of detected "super spreaders" whose true spreads are below k.
- *Number of false negatives*, i.e., the number of real super spreaders that are not detected.

We perform two experiments with different memory allocations, 16Mb and 2Mb, respectively. Figure 11(a) shows the results under 16Mb. The relative error of SS is between 9.0% and 11.2% when the threshold ranges from 200 to 800, whereas rSkt2(bitmap) performs better with relative error between 1.0% and 3.3%. Figure 11(b) shows the results under 2Mb. The relative error of SS is between 17.3% and 51.0%, whereas rSkt2(bitmap) performs better with relative error between 2.8% and 10.0%. We find that rSkt2(bitmap) works well under tight memory when the performance of SS deteriorates. This is also true in terms of false positives and false negatives. Table 3 shows the true number of super spreaders in the packet traces that we use in this experiment. Under 2Mb, Table 4 shows that SS reports much more false positives than rSkt2(bitmap). In practice, more false positives may lead to additional false alarms and take extra time from system admin to investigate. Table 5 shows that SS also produces more false negatives than rSkt2(bitmap). In practice, more false negatives may allow some true offenders to escape timely detection.

6 CONCLUSION

In this paper, we have proposed two randomized error-reduction sketches for online measurement of flow spread. They provide an implementation framework with bitmaps, FM estimators or HLL estimators as plug-ins to meet different performance-overhead requirements. The new sketch designs split error (introduced by other flows due to estimator sharing) into two halves, one stored with the flow of interest in a primary estimator and the other half stored separately in a complement estimator. By subtracting the complement from the primary estimator, we are able to statistically remove the error and achieve an accuracy one order of magnitude better than the prior art. Through theoretical analysis and experimental studies, we show that our randomized sketches work well on both software platform and hardware platform, producing accurate spread estimates in tight memory at low processing overhead for online queries.

ACKNOWLEDGMENTS

This work is funded by NSF grants CNS-1909077 and CNS-1719222. We thank the Xilinx University Program for hardware/software donation.

REFERENCES

[1] 2016. Retailrocket Recommender System Dataset. 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] K. Beyer, P. Haas, B. Reinwald, Y. Sismanis, and R. Gemulla. 2007. On Synopses for Distinct-value Estimation under Multiset Operations. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. 199–210.

[5] 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.

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

[7] G. Cormode and S. Muthukrishnan. 2003. Estimating Dominance Norms of Multiple Data Streams. In *European Symposium on Algorithms*. Springer, 148–160.

[8] G. 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.

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] P. Lieven and B. Scheuermann. 2010. High-speed Per-flow Traffic Measurement with Probabilistic Multiplicity Counting. In *2010 Proceedings IEEE INFOCOM*. IEEE, 1–9.

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

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. 2015. Scream: Sketch Resource Allocation for Software-defined Measurement. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*. 1–13.

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

[27] 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.

[28] 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.

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

[30] 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.

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

[32] UCSD. 2015. CAIDA UCSD Anonymized 2015 Internet Traces on Jan. 17. https://www.caida.org/data/passive/passive_2015_dataset.xml.

[33] 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.

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

[35] H. Wang, C. Ma, O. Odegbile, S. Chen, and J.K. Peir. 2015. Full Version: Randomized Error Removal for Online Spread Estimation in Data Streaming. https://www.dropbox.com/s/juce51g7as7vt0p/fullversion.pdf?dl=0.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

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

[42] 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.

[43] 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.

[44] Y. Zhou, Zhou Y., Chen M., and Chen S. 2017. Persistent Spread Measurement for Big Network Data Based on Register Intersection. *Proceedings of ACM Sigmetrics* (2017).

[45] 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.

[46] 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.