Single Update Sketch with Variable Counter Structure

Dimitrios Melissourgos*
Grand Valley State University
Allendale, MI, USA
dmelissourgos@gmail.com

Haibo Wang*
University of Kentucky
Lexington, KY, USA
wanghaibo@uky.edu

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

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

Shiping Chen
University of Shanghai for Science and Technology
Shanghai, China
chensp@usst.edu.cn

ABSTRACT

Per-flow size measurement is key to many streaming applications and management systems, particularly in high-speed networks. Performing such measurement on the data plane of a network device at the line rate requires on-chip memory and computing resources that are shared by other key network functions. It leads to the need for very compact and fast data structures, called sketches, which trade off space for accuracy. Such a need also arises in other application context for extremely large data sets. The goal of sketch design is two-fold: to measure flow size as accurately as possible and to do so as efficiently as possible (for low overhead and thus high processing throughput). The existing sketches can be broadly categorized to multi-update sketches and single update sketches. The former are more accurate but carry larger overhead. The latter incur small overhead but their accuracy is poor. This paper proposes a Single update Sketch with a Variable counter Structure (SSVS), a new sketch design which is several times faster than the existing multi-update sketches with comparable accuracy, and is several times more accurate than the existing single update sketches with comparable overhead. The new sketch design embodies several technical contributions that integrate the enabling properties from both multi-update sketches and single update sketches in a novel structure that effectively controls the measurement error with minimum processing overhead.

PVLDB Reference Format:
Dimitrios Melissourgos, Haibo Wang, Shigang Chen, Chaoyi Ma, and Shiping Chen. Single Update Sketch with Variable Counter Structure. PVLDB, 16(13): 4296 - 4309, 2023.
doi:10.14778/3625054.3625065

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/DimitrisMel/SSVS. Visited on 09/17/2023.

*Both authors contributed equally to this research.

Proceedings of the VLDB Endowment, Vol. 16, No. 13 ISSN 2150-8097.
doi:10.14778/3625054.3625065

1 INTRODUCTION

Streaming algorithms process a continuous sequence of data items at high rate by scanning the items once for useful statistics. One typical example of high-rate data streams are the packet streams on the Internet, where each packet carries a flow ID, all packets with the same ID form a flow, and a classic measurement is per-flow size (i.e., number of packets in each flow), as a key function of NetFlow [10], which is in turn the key tool for numerous network management systems. Measuring per-flow size on a high-speed network has applications in billing, traffic engineering, load balancing, anomaly detection, heavy hitter detection, etc.[6, 12, 20, 29, 47, 48, 58]. While we will use packet streams to motivate for our work in this paper, it should be stressed that data streaming and its algorithms have broad applications in web services, e-commerce, stock trading, social networks, in-game player experience, geospatial services, distributed sensing, and network monitoring [7, 17, 19, 24, 25, 34, 41]. Its practical importance is evident from industrial pushes such as Amazon Kinesis Streams [44].

Implementing per-flow size measurement on the data plane of a modern router or switch is a challenging task [4, 9, 11, 26, 27, 58] because it competes for limited on-chip resources on network processors that process and forward packets at extremely high speeds in the order of tens or hundreds of millions of packets per second. Because the on-chip memory (e.g., SRAM) and the processing unit have to be committed to the key network functions such as routing-table lookup, traffic shaping, access control, and deep packet inspection, the resources that can be allocated to a measurement function are often limited, especially when there are multiple co-existing measurement functions, each for a different purpose. Compact and efficient data structures, called sketches, have been the preferred choice to provide per-flow size estimations with limited memory allocation [4, 8, 9, 11, 15, 26, 27, 30, 32, 50, 52, 53, 59]. Beyond networks, in the broader context of applications, even with datacenter resources, resource contention (including memory) can still be a challenge for very large datasets, which makes sketches useful [28]. That is more true if one wants to process large datasets by using ordinary computing resources (such as desktops) for cost or convenience reasons.

Given a certain amount of allocated memory, there are two key performance metrics to consider in sketch design, *flow-size estimation accuracy* and *per-packet processing overhead*. Accuracy is critical to supporting the applications that are built on flow size

information. Overhead is critical to ensuring that the measurement function does not cause a bottleneck that constrains the streaming throughput. The existing work can be classified into two categories, *multi-update sketches* [4, 8, 11, 15] and *single update sketches* [26, 27, 59]. The former focuses on estimation accuracy, whereas the latter focuses on processing overhead.

When the number of counters in an allocated memory is far fewer than the number of flows to be measured, each counter has to record the data items (e.g., packets) from multiple flows, causing inter-flow noise. An interesting approach is to instead record each flow in multiple counters, mixing with different flows for noise control or reduction. These multi-update sketches have to visit multiple counters for each arrival data item and update one or multiple counters, resulting in higher overhead. They include CountMin (CM) [11], Count Sketch (CS) [8], Counter Update (CU) [15], and their numerous variants, including those that replace regular counters with self-adjusting counters (SC) [4], active counters (AC) [39] or Counter Pyramid [52] (which incurs more updates in the worse case).

To minimize per-item processing time, the single update sketches still record each flow in multiple counters, but only update one of these counters for each data item. They include randomized counter sharing (RCS) [26, 27] and its variant using active counters (RCS-AC) [59]. It has another variant that uses counter tree [9], which has higher processing time and incurs multiple counter updates for some data items while making one counter update for other items. The problem of the existing single update sketches is that their accuracy is very poor.

The state of the art is illustrated in Figure 1, where the x-axis is the average error in flow size estimation and the y-axis is the per-packet processing time. The existing sketches are placed in the figures based on the experimental results that will be explained later. For example, CU-SC is Counter Update [15] with self-adjusting counters [4]; it achieves the best accuracy among multi-update sketches but has high processing overhead. The figure shows that multi-update sketches are clustered in the upper-left portion, whereas the single update sketches are clustered in the lower-right portion, making tradeoff between estimation accuracy and processing overhead.

Can we design a novel sketch that fills the empty lower-left portion of the figure, with both high accuracy and low overhead? This has been an unanswered question, without a proof on the hard limitation of accuracy-overhead tradeoff or a new sketch that demonstrates the feasibility of achieving both. This paper proposes a new sketch called SSVS, which fits in that void, with an average error in the rank of the best multi-update sketch and a processing overhead in the rank of the best single update sketch. We have three technical contributions. The first contribution is to integrate self-adjusting counters and active counters in a variable counter structure where the counters expand first in size and then in exponent, creating a very large range and a dynamically adjusting small error, with a maximum counter size of just 16 bits. The second contribution is to integrate positive/negative noise cancellation with single counter update, which is key to achieve both accuracy and efficiency. The third contribution is to introduce the concept of noise interval that blocks out large noise component in flow size estimation and thus improve estimation accuracy. Our experimental

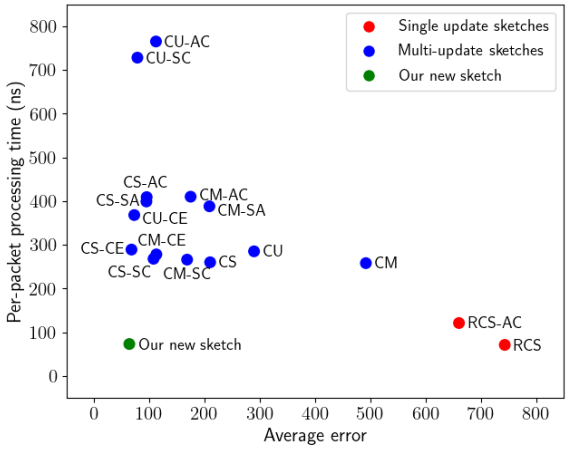

Figure 1: Comparison of sketches in the overhead-accuracy space for the CAIDA data set. The horizontal axis is the average error of all flows. The vertical axis is per-packet processing time. Each sketch is placed based on the experimental results (discussed later in the paper).

results show that (1) the new sketch achieves accuracy comparable to the most-accurate existing sketch, CU-SC [4], with one-tenth of its overhead, and (2) the new sketch achieves overhead similar to the most-efficient existing sketch, RCS [26, 27], with multi-fold better accuracy.

2 PRELIMINARIES

2.1 System Model

Consider a data stream measurement module that processes a large sequence of data items at a high rate. Each data item carries a flow ID, all items with the same ID form a flow, and the module measures the size of each flow, i.e., the number of items in each flow. As an example, it can be a traffic measurement module implemented on the data plane of a high-speed network device, where one of its functions is to measure the number of packets (i.e., data items) in each flow, under limited available on-chip resources in memory and processing. The flow ID is defined based on the application need. It may be source address, destination address, source-destination address pair, TCP five-element tuple, or any combination of header fields carried by the packet. The measurement happens in epochs of a certain length. The module records statistics into a sketch data structure as it processes the data items. At the end of each epoch, the sketch with its recorded statistics is offloaded to a server where queries on flow sizes are answered. The module then resets its data structure to start the next epoch. For offline query, given a flow ID, the server can estimate the size of the flow in any past epoch from the stored sketches. For online query, given a flow ID, the module can estimate the size of the flow in the current epoch.

2.2 Related Work

To enhance accuracy and efficiency, the prior work has pursued two orthogonal directions: improving sketch structure and designing efficient counters. Table 1 provides a summary of existing work.

2.2.1 Sketch Structures. There are broadly two sketch structures for per-flow size measurement: *multi-update sketches* and *single update sketches*. The multi-update sketches include CountMin (CM) [11], Counter Update (CU) [15], Count Sketch (CS) [8] and their variants. CM hashes each flow f to d counters in a two-dimensional counter array, and increases all the d counters by one for each arrival packet of the flow. To estimate the size of the flow, it returns the minimum value of the d counters. CU differs from CM by only increasing the smallest one(s) of the d counters by one for each arrival packet of the flow. However, it still has to perform d memory accesses to retrieve the current values of the counters. CS either increases or decreases all the d counters by one, based on a pseudo-random hash flag of the flow. Each counter provides an unbiased estimate of the flow size. CS uses the median value of the d counters as the final estimate. CM/CU/CS and their variants are widely used in network traffic measurement research [18, 30, 50, 54–58].

The most notable single update sketch is Randomized Counter Sharing (RCS) [26, 27]. RCS also maps each flow f to l counters, but only randomly selects one of the l counters to increase for each arrival item of flow f. Because the counters are shared by all flows, each counter of flow f carries noise from other flows. To estimate the size of flow f, RCS returns the sum of its l counters subtracted by an average noise measured across all counters. RCS adopts a large value for l (such as 50 in [26, 27]) to ensure sufficiently randomized noise distribution, but the total noise present in a flow's counters is proportional to l.

There are layered sketch structures, including Counter Braids [32] and Pyramid Sketch [52], built on top of CM/CU, and Counter Tree [9], built on top of RCS. They use small counters at the bottom layer, and these counters will overflow into higher layers recursively. They are more memory efficient, thanks to small counters, but they have variable processing time due to the need to operate on multiple layers of CM/CU/RCS when overflow happens. Their worst-case processing time is determined by the number of layers. Recent research shows that using more efficient counter designs outperforms multiple layers of small counters [4, 51].

The Bucketized Rank Indexed Counter (Brick) [22] partitions a counter array into groups of k counters each, called *bricks*. Each brick is multi-layered, with k small counters at the bottom, which overflows recursively to higher layers. Different from Counter Braids where each counter overflows into multiple higher-layer counters, each counter in Brick overflows into a single higher-layer counter. However, each layer requires an index array to keep information about where each of its counters overflows into. It thus has higher memory overhead than Counter Braids under the same counting range.

CM, CU, CS and RCS are generic sketch structures that can adopt different counter designs, which will be elaborated next.

2.2.2 Efficient Counters. A regular counter of r bits has a range of $[0, 2^r)$. To expand the range, DIScount COunting (DISCO) [21] sacrifices counting accuracy by mapping the counter values to a sequence of integers with increasing gaps, $\{0, 1, \frac{b^2-1}{b-1}, ..., \frac{b^{2^r-1}-1}{b-1}\}$, where $b > 1$, which spans a much wider range but has a much coarser counting granularity. DISCO increments its counter probabilistically, where the exact probability is determined by the current counter value. Its range is $O(b^{2^r})$, but its counting is highly inaccurate. Counter Estimation Decoupling for Approximate Rates (CEDAR) [42] improves over DISCO with a mapping function that minimizes the maximum relative error in counting. The Independent Counter Estimation Buckets (ICE-Buckets) [13] partitions a counter array into buckets of k counters each. Each bucket begins with a small-ranged mapping function. Whenever a counter overflows, it switches to a larger-ranged mapping function.

Self-adjusting counters (SCs) in [4] begin as 8-bit regular counters. When counters overflow, they will merge with neighboring counters in the array to create larger-sized counters. Self-Adjusting Lean Streaming Analytics (SALSA) implement SCs in various sketch structures such as CM, CU, and CS to measure per-flow size; they are denoted as CM-SC, CU-SC, and CS-SC, respectively.

Another approach to expand counter range is through sampling. Additive error counter [3] begins with a sampling probability of 1. Each time overflow occurs, the sampling probability is halved. CM with additive error counters is called Additive Error Estimator (AEE) [3]. All counters in AEE share the same sampling probability, which is determined by the counter that overflows the most. This approach is efficient in tracking the sizes of large flows, but aggressive sampling across all counters may result in poor size estimation for small and medium flows or even completely miss some small flows [23, 28, 31], which is undesirable for per-flow size measurement, as is considered in this paper.

Active counter (ACs) [39] splits its bits in two parts: a number part v and an exponent part e. Its value is $v \times 2^e$. To increase the counter by one, we must do so probabilistically, with a probability of $\frac{1}{2^e}$. Combining RCS [26, 27] and active counters produce a sketch denoted as RCS-AC [59].

Self-adaptive counters [51] also have a number part and an exponent part. The exponent part has a variable length, its bits must be all ones, and the number of ones is the exponent value. The two parts are separated by a bit zero. Its range is limited, comparing with AC [39]. For example, for a 16-bit counter, if we want at least 10 bits in the number part for resolution (i.e., counting accuracy), the exponent for a self-adaptive counter can only be up to 5, whereas the exponent for AC can be up to 31, with a range 2^{26} times larger.

2.2.3 Hash Table, Flow Spread and Heavy Hitters. Hash tables [14, 16] can be used to keep track of the size of each flow. However, if the number of flows exceeds the number of hash entries (such that sketches become necessary), hash tables can only keep the large flows for heavy hitter detection [38]. Sketches for a different task of measuring per-flow spread [46, 49, 58], i.e., number of *distinct* data items in each flow, may also be used for estimating per-flow size. But their performance is generally much worse [58]. Sketches for detecting heavy hitters [2, 5, 30] do not perform per-flow size measurement.

2.3 Motivation

We want to explore a new sketch design that possesses the benefits from both worlds: the accuracy of the multi-update sketches and

Table 1: Performance comparison of the proposed SSVS sketch and existing solutions. SSVS is the only one that performs single-update (low processing time for recording) and high-accuracy per-flow size measurement. Layered sketches need additional processing overhead to update possibly a chain of counters at the upper layers when counters at the bottom layer overflow. Solutions with bold font are considered as the state of the art.

Group of Solutions	Solutions	Counters Used	Measure Per-flow?	Counter Updates per Item	Accuracy
Generic sketch structures	CM [11]	Regular counters	Yes	Multi-update	Medium
	CU [15]	Regular counters	Yes	Multi-update	Medium
	CS [8]	Regular counters	Yes	Multi-update	Medium
	RCS [26, 27]	Regular counters	Yes	Single-update	Low
Layered sketches	Counter Braids [32]	Small-size regular counters	Yes	Multi-update and recursive update	Low
	Brick [22]	Small-size regular counters	No	Multi-update and recursive update	High
	Pyramid Sketch [52]	Small-size regular counters	Yes	Multi-update and recursive update	Medium
	Counter Tree [9]	Small-size regular counters	Yes	Recursive update	Low
Efficient counter designs	AEE [3]	Small-size regular counters	No	Multi-update	High
	DISCO [21]	Small-size counters	Yes	Multi-update	Low
	CEDAR [42]	Small-size counters	Yes	Multi-update	Medium
	ICE-buckets [13]	Small-size counters	No	Multi-update	High
	Self-adaptive counters [51]	Small-size counters	Yes	Multi-update	High
	SC/SALSA [4]	Small to large counters	Yes	Multi-update	High
	AC [39]	Small-size counters	Yes	Multi-update	High
Hash tables	Cuckoo filter [16]	Regular counters	No	Single-update	High
	Tinytable [14]	Regular counters	No	Single-update	High
This paper	SSVS	Small-size variable counters	Yes	Single-update	High

the efficiency of the single-update sketches. For that, we have to integrate the enabling properties from both multi-update sketches and single-update sketches in a novel structure that resolves their incompatibility.

First, to minimize the processing overhead, we prefer a single update sketch, which means the multi-update sketch structures and their variants (including the generic sketch structures and the layered sketches in Table 1) [4, 8, 9, 11, 15, 22, 32, 52] are out of consideration. The existing single-update sketches, RCS [26, 27] and its variant RCS-AC [59], have very poor accuracy, as shown in Figure 1. They map each flow to l counters and record each data item of the flow by increasing one of the l counters by one. The expectation of the noise (from other flows) in each counter is estimated as the average value of all counters. This approach is valid only if noise is about randomly distributed in all counters, which requires the value of l to be large (e.g., 50 in [27] and 512 in [59]). However, because the overall noise level in a flow's size estimate increases with l, the large value of l causes the poor accuracy of RCS-AC. Now the question is how to reduce l. Our single-update sketch design will use noise cancelation to ensure that each of the l counters has a noise expectation of zero and l can be any small value. Moreover, it eliminates the overhead in RCS/RCS-AC to scan the whole counter array for an estimate of the noise expectation per counter (as it is zero in our design).

Second, by increasing the number of counters, we can further improve the accuracy of a single-update sketch. With a given amount of memory, more counters mean fewer bits per counter. We have a three-way tradeoff to play: number of counters, range, and counting accuracy. The existing work has their limitations in this space of tradeoff. Some counter designs in Table 1 such as DISCO [21] and CEDAR [42] achieve large range by sacrificing counting accuracy, particularly in the low end of its range. They are suitable for large flows, but not for small flows or medium flows (depending on the counter configuration). Other designs are adaptive to ensure more accurate counting for small flows at cost of limited range [51], expansion in counter size [4], or processing overhead [13, 39]. To address these issues, we integrate self-adjusting counters [4] and active counters [39] in an efficient variable counter structure that expands the range to very large values, ensures precise counting up to 2^{16}, bounds the relative counting error beyond 2^{16} to a small value, and limits the counter size to 16 bits in the worst case.

3 SINGLE UPDATE SKETCH WITH VARIABLE COUNTER STRUCTURE (*SSVS*)

In this section, we propose a new single update sketch, denoted as SSVS, with a variable counter structure. The performance gap between our new sketch and the existing single update sketches, RCS/RCS-AC, is significant, as shown in Figure 1. To achieve such a performance boost, its design differs from the existing work in counter structure, sketch design, data recording, and query operation.

3.1 Variable Counter Structure

Our idea of variable counter structure is motivated from the limitations of the counters used in RCS/RCS-AC. For a range of 2^{32}, RCS will need 32-bit regular counters; observing the byte boundary, RCS-AC will need 16-bit active counters (ACs), each with 5-bit exponent. From Figure 1, RCS-AC is more accurate than RCS. The reason is that it has twice the number of counters. However, active counters perform probabilistic counting and thus incur counting errors. Can we create even more counters, nearly twice as many as RCS-AC has at least initially, yet count *precisely* until it becomes infeasible with 16 bits per counter? Our insight is that each counter should be made dynamic, counting precisely up to 16 bits and then switching to probabilistic counting with progressively increasing error. Because we do not know beforehand how many data items each counter will

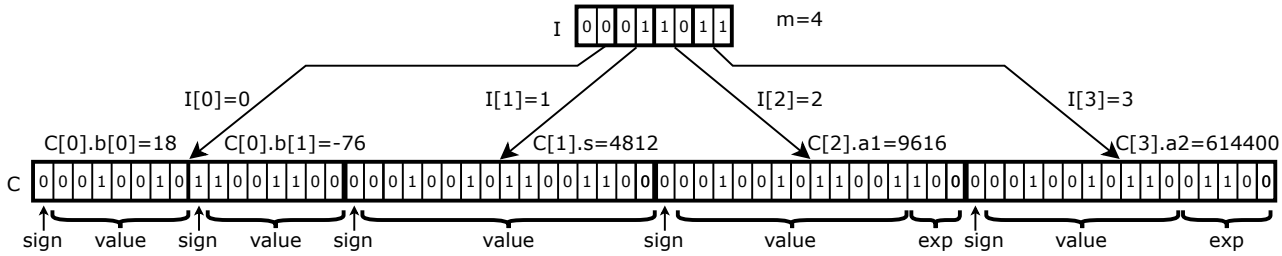

Figure 2: Variable Counter Structure. A 64-bit block of SRAM, divided into 4 intervals of 16 bits each. The interpretation of the counters in the counter array C depends on the values of the indicator array I. In this example, each 16-bit interval has the same bit value of 0001001011001100_2. The first bit of each counter is the sign. Since I[0]=0, the first 16 bits of memory are interpreted as two 8-bit counters, b[0] and b[1]. The value of the first 8-bit counter, with its sign being 0, is $C[0].b[0] = 0010010_2 = 18_{10}$. The value of the second 8-bit counter, with its sign being 1, is $C[0].b[1] = -1001100_2 = -76_{10}$. For the second counter we have I[1]=1, so the second 16-bit interval is a short counter. With the sign being 0, its value is $C[1].s = 001001011001100_2 = 4812_{10}$. For the third counter we have I[2]=2, so we interpret the first bit as the sign, the next 12 bits as the value and the last 3 bits as the exponent. The value part is $C[2].a1.v = 001001011001_2 = 601_{10}$ and the exponent part is $C[2].a1.e = 100_2 = 4_{10}$. Therefore, the counter's value is $C[2].a1.v \cdot 2^{C[2].a1.e} = 601 \cdot 2^4 = 9616$. For the last counter we have I[3] = 3, so its value part is $C[3].a2.v = 0010010110_2 = 150_{10}$ and its exponent part is $C[3].a2.e = 01100_2 = 12_{10}$. Therefore, the counter's value is $C[3].a2.v \cdot 2^{C[3].a2.e} = 150 \cdot 2^{12} = 614400$.

record, the counters must individually adapt from exact counting to probabilistic counting on the fly. None of the existing counter designs, including active counters, can do this well.[1] In comparison, our variable counter design is structured specifically with such a goal in mind. Observing the byte boundary, it begins with byte counters in order to maximize the number of counters; note that the accuracy of all sketches improves with a larger number of counters. Each byte counter will overflow into a 16-bit counter, still for exact counting, which will then overflow into a 16-bit active counter with 3-bit exponent, which will expand to 5-bit exponent upon overflow.[2] This design of dynamic adaptation from exact counting to probabilistic counting progressively in a variable counter structure has advantage over the existing designs in maximizing the number of counters and minimizing the counting error at the small end.

SSVS uses an array C of m *words*, each of 16 bits or two bytes,[3] and an array I of m indicators, each of 2 bits. I is the indicator for C, specifying how C should be interpreted, as explained below. Consider any $j \in [0, m)$.

- When $I[j] = 0$, we treat $C[j]$ as two *byte counters*, referred to as $C[j].b[0]$ and $C[j].b[1]$, each of 8 bits. The first bit is the sign and the remaining 7 bits are the value of the counter.
- When $I[j] = 1$, we treat $C[j]$ as a *short counter* of two bytes, denoted as $C[j].s$. The first bit is the sign and the remaining 15 bits are the value of the counter.
- When $I[j] = 2$, we treat $C[j]$ as a *small-ranged active counter* of two bytes, denoted as $C[j].a1$, with its first bit as the sign, the next $(15 - \alpha)$ bits as the value part, together denoted as $C[j].a1.v$, and the remaining α bits as the exponent part, denoted as $C[j].a1.e$, where α is a small integer parameter, such as 3 used in our experiments. We abbreviate *Active Counter* as *AC*. With 3 bits of exponent, the range of a small-ranged AC is $(-2^{19}, 2^{19})$. With 12 bits of value, the rounding error is less than $\frac{1}{2^{11}}$.
- When $I[j] = 3$, we treat $C[j]$ as a *large-ranged active counter* of two bytes, denoted as $C[j].a2$, with its first bit as the sign, the next $(15 - \beta)$ bits as the value part, denoted as $C[j].a2.v$, and the remaining β bits as the exponent part, denoted as $C[j].a2.e$, where β is another integer parameter, such as 5 used in our experiments. Its range is $(-2^{41}, 2^{41})$, with its rounding error less than $\frac{1}{2^9}$.

We will adopt $\alpha = 3$ and $\beta = 5$ in the rest of the paper. These parameter values cover a broad range, while the user can certainly change them to other values based on application need.

The array C has a variable counter structure, defined by the indicator array I, which initially sets all indicators to zero and evolves as the data items of the flows are recorded. C is initialized with $2m$ small byte counters, aligning with our goal of maximizing the number of counters to enhance accuracy. Each byte counter counts precisely until overflow. When that happens, we expand the counting range by merging two adjacent byte counters into a short counter to continue exact counting. When a short counter overflows, it becomes a small-ranged active counter and then a large-ranged active counter, which is controlled by the counter's indicator. We illustrate how arrays C and I work with an example in Figure 2, in which each 16-bit segment has the same bit value, but the content of C is interpreted differently, depending on I.

3.2 Mapping Flows to Counters

Each flow f is mapped to l counters in C using l hash functions $h_i(.)$, $0 \le i < l$, where l is a system parameter that controls the estimation error, which we will analyze later. For a single update

[1] Paper [47] is a variant of active counter, with a less efficient exponent design. For example, it uses $\underline{1010}0\underline{11111}_2$ to represent $1010_2 \times 2^5$, where the five trailing ones represent an exponent of 5. In contrast, our design uses five-bit exponent for a multiplying factor of 2^0 through 2^{31}.

[2] Given 16 bits in total, with 2 more bits in exponent, there are two fewer bits in the value part, which increases probabilistic counting error.

[3] A word is typically 32 or 64 bits long. One may refer to 16 bits as a short word, but we refer to it as word for simplicity.

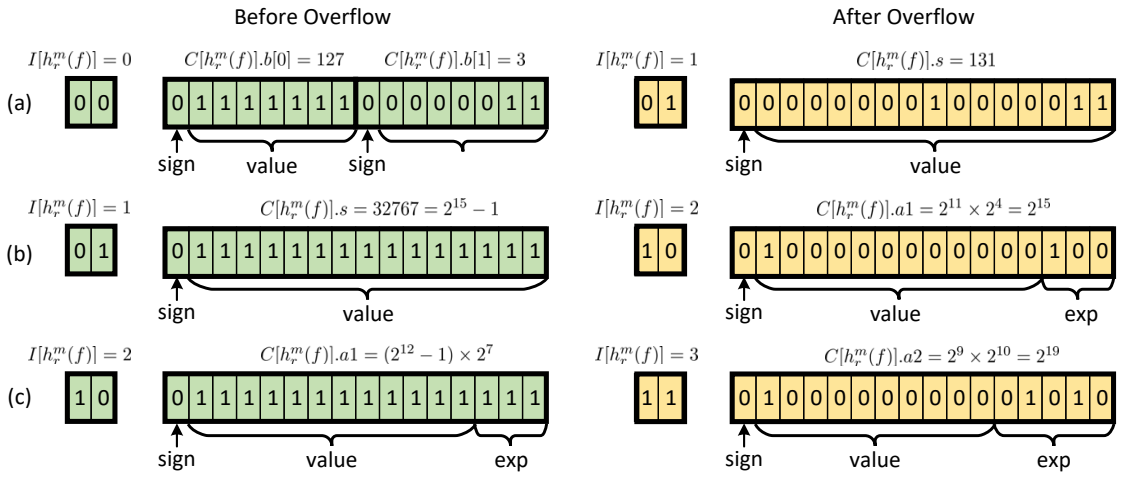

Figure 3: Examples illustrating how we handle overflows in $C[h_r^m(f)]$: Example (a): The byte counter $C[h_r^m(f)].b[0]$ is on the verge of overflowing, with a pending increase. We merge $C[h_r^m(f)].b[0]$ and $C[h_r^m(f)].b[1]$ to create the short counter $C[h_r^m(f)].s$. The resulting value is the sum of the two byte counters plus 1, which equals 127 + 3 + 1 = 131. $I[h_r^m(f)]$ is updated from 0 to 1. Example (b): The short counter $C[h_r^m(f)].s = 2^{15} - 1$ is approaching its overflow point. When faced with a pending increase to 2^{15}, we convert it into a small-ranged active counter by setting the value part (12 bits) to 2^{11} and the exponent part (3 bits) to 4. $I[h_r^m(f)]$ is accordingly incremented to 2. Example (c): The small-ranged active counter $C[h_r^m(f)].a1 = (2^{12} - 1) \times 2^7$ is nearing its overflow threshold. When confronted with a pending increase in the value part to 2^{12}, we transform it into a large-ranged active counter by setting the value part (10 bits) to 2^9 and the exponent part (5 bits) to 10. $I[h_r^m(f)]$ is updated to 3. By following this approach, we ensure that counter transformations are performed without incurring any errors.

sketch, each data item of flow f will be recorded by one of its l counters. According to [58] and [46], practically, one may implement l hash functions from a master hash function H as $h_i(x) = H(x \oplus i)$, $0 \le i < l$, where $\oplus$ is the XOR operator.

The hash value of flow f is an integer, denoted as $h_i(f)$, $0 \le i < l$. Let $h_i^0(f)$ and $h_i^1(f)$ be the the first and second bits of $h_i(f)$, respectively. Let $h_i^{2+}(f)$ be the remaining bits. Suppose that the range of $h_i^{2+}(f)$ is larger that the range of m. We define $h_i^m(f) = h_i^{2+}(f) \mod m$.

We use $h_i^m(f)$, $0 \le i < l$, as an index to map f to a word in C, i.e., $C[h_i^m(f)]$, and to an indicator in I, i.e., $I[h_i^m(f)]$. We use $I[h_i^m(f)]$ to interpret the counter(s) in $C[h_i^m(f)]$. If there are two byte counters in case of $I[h_i^m(f)] = 0$, we use $h_i^1(f)$ to further map f to one of the two byte counters. The details are given below.

- If $I[h_i^m(f)] = 0$, we map f to $C[h_i^m(f)].b[h_i^1(f)]$, which is a byte counter.
- If $I[h_i^m(f)] = 1$, 2, or 3, we map f to $C[h_i^m(f)].s$, $C[h_i^m(f)].a1$, or $C[h_i^m(f)].a2$, respectively, which are the same two bytes but interpreted differently.

We haven't used $h_i^0(f)$ yet, which is left for the data recording operation below.

3.3 Recording Data Items

Each data item of a flow will be recorded by one of the l counters that the flow is mapped to. Which counter to use is randomly selected, and the counter is either increased or decreased by one, pseudo-randomly determined based on the flow ID. As the counter may be shared by other flows (noise), some of those flows will increase the counter and others will decrease the counter, resulting in noise cancelation and lowering the residual noise, which may be positive or negative, with an expectation of zero. Such a technique was used in CS [8], a multi-update sketch. Below we adopt it in a single update sketch with a variable counter structure.

At the beginning of each measurement period, all bits in C and I are set to zeros. When processing the next arrival data item, which carries a flow ID f, we generate a random number r in the range of $[0, l)$. We compute the hash $h_r(f) = H(f \oplus r)$, compute the index $h_r^m(f)$ and record the data item in the counter $C[h_r^m(f)]$. The exact recording operation is based on the value of $h_r^0(f)$. If $h_r^0(f) = 0$, we increase $C[h_r^m(f)]$ by one; if $h_r^0(f) = 1$, we decrease $C[h_r^m(f)]$ by one.

The increase (or decrease) of an AC is different from a byte/short counter. The AC increase (or decrease) is done probabilistically: Depending on the value of $h_r^0(f)$, for a small-ranged AC, $C[h_r^m(f)].a1.v$ is increased (or decreased) by one with probability $\frac{1}{2^{C[h_r^m(f)].a1.e}}$; for a large-ranged AC, $C[h_r^m(f)].a2.v$ is increased (or decreased) by one with probability $\frac{1}{2^{C[h_r^m(f)].a2.e}}$.

If $C[h_r^m(f)]$ is a byte counter and it overflows, we need to expand its size to a short counter. If $C[h_r^m(f)]$ is a short counter and it overflows, we need to turn it into a small-ranged AC. When that counter overflows, we turn it into a large-ranged AC. We do not expect a large-ranged AC to overflow, since its range is $(-2^{41}, 2^{41})$. But if it does, it is easy to redefine the size of its exponent part from 5 bits, to 6 bits or more. Below we explain how exactly to handle the problem that a pending increase (or decrease) would cause $C[h_r^m(f)]$ to overflow.

Algorithm 1 Data-Item Recording

Input: flow ID f, a master hash function H, counter array C and indicator array I
Output: single counter update to C and I

r = random.nextInt(0, l-1);
$h_r(f) = H(f \oplus r)$;
$h_r^m(f) = h_r^{+2}(f) \mod m$;
if $I[h_r^m(f)] = 0$ **then**
    run Algorithm 2 Update byte counter;
else if $I[h_r^m(f)] = 1$ **then**
    run Algorithm 3 Update short counter;
else if $I[h_r^m(f)] = 2$ **then**
    if random.nextInt(0, $2^{C[h_r^m(f)].a1.e} - 1) = 0$ **then**
        run Algorithm 4 Update small-ranged AC;
else
    if random.nextInt(0, $2^{C[h_r^m(f)].a2.e} - 1) = 0$ **then**
        run Algorithm 5 Update large-ranged AC;
return updated C and I;

Algorithm 2 Update byte counter

Input: bit $h_r^0(f)$, bit $h_r^1(f)$, counter $C[h_r^m(f)]$, and indicator $I[h_r^m(f)]$
Output: updated counter $C[h_r^m(f)]$ and indicator $I[h_r^m(f)]$

if $h_r^0(f) = 0$ **then**
    if $C[h_r^m(f)].b[h_r^1(f)] = 127$ **then**
        $I[h_r^m(f)] = I[h_r^m(f)] + 1$;
        $C[h_r^m(f)].s = C[h_r^m(f)].b[0] + C[h_r^m(f)].b[1]$;
        $C[h_r^m(f)].s = C[h_r^m(f)].s + 1$;
    else
        $C[h_r^m(f)].b[h_r^1(f)] = C[h_r^m(f)].b[h_r^1(f)] + 1$;
else
    if $C[h_r^m(f)].b[h_r^1(f)] = -127$ **then**
        $I[h_r^m(f)] = I[h_r^m(f)] + 1$;
        $C[h_r^m(f)].s = C[h_r^m(f)].b[0] + C[h_r^m(f)].b[1]$;
        $C[h_r^m(f)].s = C[h_r^m(f)].s - 1$;
    else
        $C[h_r^m(f)].b[h_r^1(f)] = C[h_r^m(f)].b[h_r^1(f)] - 1$;
return updated $C[h_r^m(f)]$ and $I[h_r^m(f)]$;

- *Case 0:* $I[h_r^m(f)] = 0$.
  $C[h_r^m(f)]$ is either $C[h_r^m(f)].b[0]$ or $C[h_r^m(f)].b[1]$. We need to combine the two byte counters into a short counter. We set $I[h_r^m(f)] = 1$, add the values of $C[h_r^m(f)].b[0]$ and $C[h_r^m(f)].b[1]$ to $C[h_r^m(f)].s$, i.e. $C[h_r^m(f)].s = C[h_r^m(f)].b[0] + C[h_r^m(f)].b[1]$. Then, we increase (or decrease) $C[h_r^m(f)].s$ by one, based on the value of $h_r^0(f)$. We give an example for this case in Figure 3 (a).
- *Case 1:* $I[h_r^m(f)] = 1$.
  We need to turn a short counter into a small-ranged AC. We set $I[h_r^m(f)] = 2$ and turn $C[h_r^m(f)].s$ into $C[h_r^m(f)].a1$ by right-shifting the counter by 4 bits and then setting $C[h_r^m(f)].a1.e$ (i.e., the right-most 3 bits) to 4. Then, we increase (or decrease) $C[h_r^m(f)].a1$ by one, based on the value of $h_r^0(f)$. We give an example for this case in Figure 3 (b).
- *Case 2:* $I[h_r^m(f)] = 2$.
  We need to turn a small-ranged AC to a large-ranged AC. We set $I[h_r^m(f)] = 3$ and turn $C[h_r^m(f)].a1$ into $C[h_r^m(f)].a2$ by right-shifting the counter by 3 bits and then setting $C[h_r^m(f)].a2.e$ (i.e., the right-most 5 bits) to 10. Then, we increase (or decrease) $C[h_r^m(f)].a2$ by one, based on the value of $h_r^0(f)$. We give an example for this case in Figure 3 (c).
- *Case 3:* $I[h_r^m(f)] = 3$.
  If a large-ranged AC overflows, it means that the exponent part requires more than 5 bits. We have to increase the size of the exponent part for all large-ranged ACs, which may be done by right-shifting the value part for one bit and thus allowing the expansion of the exponent part by one bit.

The detailed recording operations are given in Algorithm 1. The update operations of byte counters are given in Algorithm 2, and of other counters are provided in Github [36]. For each arrival data item, at most one counter in C will be updated. Because an active counter is updated probabilistically, there is a chance that no counter update is actually needed. Occasionally, we may also need to update an indicator, but that is rare.

3.4 Size Query and SSVS-1

To answer a query for the size of flow f, we retrieve the flow's l indicators, $I[h_i^m(f)]$, $0 \le i < l$, and l counters, $C[h_i^m(f)]$. A simple method is to estimate the flow size $\hat{n}_f$ as

$$\hat{n}_f = \sum_{i=0}^{l-1} \delta C[h_i^m(f)], \tag{1}$$

where $\delta = 1$ when $h_i^0(f) = 0$ and $\delta = -1$ when $h_i^0(f) = 1$.

Each of the l counters carries noise from other flows. But those noises come randomly as positive or negative, and they statistically cancel out each other. Let n_f be the true size of flow f. We have the following theorem. Its proof can be found in the supplementary material and in GitHub [36].

Theorem 1. *For any flow f, the expectation and variance of $\hat{n}_f$ produced by SSVS-1 follow:*

$$E(\hat{n}_f) \begin{cases} = n_f, & if \forall 0 \le i < l, I[h_i^m(f)] \in \{0, 1\}; \\ \in [(1 - 0.01)n_f, (1 + 0.01)n_f], & otherwise; \end{cases} \tag{2}$$

$$Var(\hat{n}_f) \le 2.0402 l^2 (n/m - n/(2m^2)). \tag{3}$$

We refer to our sketch design, the recording operations and the query method (1) together as SSVS-1.

3.5 Modified Size Estimation Method and SSVS-2

From Theorem 3, the standard error in $\hat{n}_f$ is minimized when $l = 1$, which is confirmed by our experiments discussed later. As explained in Section 2.3, a small value of l will help SSVS-1 be more accurate than RCS/RCS-AC, which is also confirmed by our experiments. It is well known that network traffic traces follow power-law distributions [1, 35, 40], with most flows being small or medium, and very few flows being very large. For such data sets, as each flow is

split among few counters (small l), the sizes of large flows are concentrated in a small number of counters, causing big noise (called noise outlier) to other flows that share these counters. To further improve accuracy, we need a way to block out the noise outliers.

We attempt to exclude the noise outliers from the estimation formula by establishing a so-called *noise interval* and only the counters within the noise interval are used for flow size computation.

Before any query, we generate a large set F of fake flow IDs (corresponding to flows of size zero). We use (1) to estimate their flow sizes, which are in fact the residual noises after cancellation. Let w be the average residual noise, i.e., $w = \frac{\sum_{f' \in F} |\hat{n}_f|}{|F|}$, which is a measure of overall residual noise level.

Given a query on flow f, we sort its l counters, $C[h_i^m(f)]$, $0 \le i < l$, and find the closest two counters, denoted as c and c' with $c \le c'$, which tend to locate at the center of the distribution. We define a *noise interval* for flow f as $[c - \frac{w}{k}, c' + \frac{w}{k}]$, where k is a parameter that controls the width of the interval. We will study this parameter experimentally. We abbreviate the noise interval as $\pm\frac{w}{k}$. The purpose of noise interval is to keep out the noise outliers. We estimate the size of flow f based on the subset N_f of counters that fall within the interval.

$$N_f = \{\delta C[h_i^m(f)] \mid c - \frac{w}{k} \le \delta C[h_i^m(f)] \le c' + \frac{w}{k}, 0 \le i < l\}$$
$$\hat{n}_f^* = \frac{l}{|N_f|} \sum_{x \in N_f} x, \tag{4}$$

where $\delta = 1$ when $h_i^0(f) = 0$ and $\delta = -1$ when $h_i^0(f) = 1$.

We refer to the version of our sketch using (4) as SSVS-2. The only difference between SSVS-1 and SSVS-2 is their estimation formulas. We know that SSVS-1 is optimized at $l = 1$. That is not the case for SSVS-2. In fact, because the noise interval contains at least two counters, SSVS-1 and SSVS-2 will be identical if they use the same number of counters per flow at $l = 1$ or 2. Our experiments will show that SSVS-2 with $l = 4$ consistently outperforms SSVS-1 with $l = 1$.

4 EXPERIMENTAL EVALUATION

4.1 Experimental Setting

We have implemented the following sketches for per-flow size measurement: (1) the proposed SSVS-1 and SSVS-2, which have the same recording operations but different query methods; (2) randomized counter sharing (RCS) [27], which is the best single update sketch in terms of low processing overhead (note that its Counter Tree variant [9] sometimes has to update multiple counters in a hierarchical structure and thus has higher overhead); (3) Randomized Counter Sharing with active counters (RCS-AC) [59], which is also a single update sketch; (4) a group of widely used multi-update sketches that employ regular counters as building blocks, including CountMin (CM) [11], Count Sketch (CS) [8], and Counter Update (CU) [15]; (5) their variants that use self-adjusting counters [4], including CountMin with self-adjusting counters (CM-SC), Count Sketch with self-adjusting counter (CS-SC), and Counter Update with self-adjusting counter (CU-SC); (6) their variants that use active counters [39], including CountMin with active counter (CM-AC), Count Sketch with active counter (CS-AC), and Counter Update with active counter (CU-AC); (7) their variants that use CEDAR counters [42], including CountMin with CEDAR (CM-CE), Count Sketch with CEDAR (CS-CE), and Counter Update with CEDAR (CU-CE); and (8) their variants that use self-adaptive counters [51], including CountMin with self-adaptive counter (CM-SA), Count Sketch with self-adaptive counter (CS-SA), and Counter Update with self-adaptive counter (CU-SA). Layered sketches have higher processing overhead due to counter updates across layers, and it is shown in [4] that CM-SC also outperforms Pyramid sketch in accuracy and shown in [51] that CM-SA, CS-SA and SU-SA outperform Counter Tree in accuracy.

The self-adjusting counters used in CM-SC, CS-SC and CU-SC are up to 32 bits, with a maximum range of 2^{32}. The active counters used in CM-AC, CS-AC and CU-AC are 16 bits with the same structure as our large-ranged ACs, with a maximum range of 2^{41}. To have a range of up to 2^{32}, we allocate each CEDAR counter 12 bits, which is recommended in its original experiments. Self-adaptive counters have two versions: static and dynamic. We use the dynamic one, because it is more accurate, as demonstrated in the original paper. Each self-adaptive counter is 16 bits, which is the same parameter setting as the original paper. We set $l = 50$ for RCS [26, 27], $l = 512$ for RCS-AC [59], as in the original papers, and $d = 4$ for CM, CS, CU, CM-SC, CS-SC, CU-SC, CM-AC, CS-AC and CU-AC. See Section 2.2 for their definitions. For SSVS-1 and SSVS-2, we will experimentally study how they react to different l values.

Table 2: Statistics of the traffic trace from CAIDA used in our experiments

Flow size range	Avg flow Size	No. of flows
[1,10]	3.1	355580
[11,100]	25.7	68057
[101,1000]	308.7	12034
[1001,10000]	2805.2	2218
≥ 10001	19370.7	274

Table 3: Statistics of the web data set in our experiments

Flow size range	Avg flow Size	No. of flows
[1,10]	1.8	913742
[11,100]	30.1	65053
[101,1000]	315.9	14393
[1001,10000]	3072.7	3860
≥ 10001	22209.3	751

Our evaluation uses two sets of performance metrics, one set for estimation accuracy and the other set for recording overhead. Estimation accuracy is evaluated by the average absolute error and the average relative error. Consider a set F of flows. $\forall f \in F$, let $\hat{n}_f$ and n_f be the flow size estimate and the true flow size, respectively. The average absolute error is defined as $\sum_{f \in F}(|\hat{n}_f - n_f|)/|F|$. The average relative error is defined as $\sum_{f \in F} \frac{|\hat{n}_f - n_f|}{n_f}/|F|$. The absolute error is more useful for small flows, whereas the relative error is more useful for large flows. For example, we consider $\hat{n}_f = 5$ to be a good estimation for $n_f = 1$ because it is off only by 4 although the relative error is 400%. We consider $\hat{n}_f = 100200$ to be a good estimation for $n_f = 100000$ although the absolute error 200 is much worse, but the relative error is only 0.2%.

Table 4: Average absolute error of SSVS-1 with respect to l, under 1Mbit memory

Flow size range	$l = 1$	$l = 2$	$l = 4$	$l = 8$	$l = 16$	$l = 32$
[1,10]	**132.6**	237.2	380.2	534.4	624.8	667.0
[11,100]	**134.8**	242.9	389.8	536.4	637.7	670.8
[101,1000]	**157.2**	311.4	470.4	627.4	753.4	794.9
[1001,10000]	**224.0**	393.9	675.8	963.9	1126.5	1202.7
≥ 10001	**217.3**	639.5	1021.7	1055.8	1186.6	1418.9

Table 5: Average absolute error of SSVS-2 with respect to l, under 1Mbit memory

Flow size range	$l = 1$	$l = 2$	$l = 4$	$l = 8$	$l = 16$	$l = 32$
[1,10]	132.6	237.2	**60.8**	80.2	142.7	195.9
[11,100]	134.8	242.9	**68.5**	90.3	153.6	211.9
[101,1000]	157.2	311.4	**108.8**	130.0	229.9	359.5
[1001,10000]	224.0	393.9	**188.2**	319.7	510.9	854.1
≥ 10001	217.3	639.5	**241.8**	420.1	833.7	1865.5

Recording overhead is evaluated by the average processing time of data item recording, the recording throughput in millions of data items per second, the number of memory accesses per data item, the number of hashes per data item, and the number of counter updates per data item during recording. The latter three metrics can be obtained from the algorithm designs. The average processing time will be obtained through experiments. The recording throughput is the inverse of the average processing time. An average processing time of 100 ns corresponds to a recording throughput of 10 thousand packets per second.

We use three data sets: (1) A real Internet traffic trace downloaded from CAIDA [43]. It consists of 18,215,144 packets (data items). We designate the source-destination IP address pair as the flow ID and there are 438,163 different flows. To record the size of each flow, we could assign a 32-bit regular counter per flow, which would require 42Mb memory without considering the indexing overhead. In contrast, the sketches used in our evaluation only require 1Mb memory. (2) A collection of web html documents downloaded from [33]. We set the flow ID to be the web document's unique number in the database. Each data item is a URL reference from other documents to the flow ID (i.e. a given document). There are 997,800 flows and 36,680,934 data items. (3) We generate seven synthetic data sets, each of them following the power-law (Zipf) distribution [37] with different degrees of skewness. Each synthetic data set contains 32 million items and a varying number of flows depending on the skewness. We gradually increase the skewness from 0.0 to 1.5. As the skewness increases, there will be a fewer number of flows that are larger. Beyond 1.5, the number of flows becomes too small for sketches to be useful. With too few flows, we can simply use a hash table and assign each flow a counter, instead of using a sketch.

To show the distribution of the data sets, we segregate the flows into five size ranges: [1-10], [11-100], [101-1000], [1001-10000] and larger than 10001. Tables 2 and 3 show the number of flows and the average flow size for each size range for the CAIDA data set and the web data set, respectively. Our experimental results in estimation accuracy will also be given for each range separately.

The experiments are performed on a desktop computer equipped with an AMD 5950X CPU with 16 cores at 3.4 GHz and 64 GB of RAM. We have uploaded our implementation on github [45].

4.2 Comparison between SSVS-1 and SSVS-2

We compare SSVS-1 and SSVS-2 on the CAIDA data set in terms of accuracy by varying the value of l from 1, 2, 4, 8, 16 to 32. The memory allocated is 1Mbit. The noise level is set to $\pm w/4$. The experiment first records the traffic trace, then queries the size of each flow, and finally measures the errors in the flow size estimations.

Table 4 presents the average absolute errors in size estimations by SSVS-1 for flows in different ranges (rows) under different l values. It shows that the errors are minimized at $l = 1$, which is consistent with Theorem 1. Table 5 presents the average absolute errors by SSVS-2. It shows a different behavior. The errors in SSVS-2 decrease at first as l increases, bottom at $l = 4$, and then increase as l further increases. The value of l has direct impact on two factors that contribute to the errors. First, as l increases, every flow f is split into more pieces (each piece recorded in a counter that f is mapped to). It is therefore less likely to create noise outliers, which helps reduce the estimation error. Second, each counter carries a certain amount of noise from other flows. The more counters that f uses for its size estimation, the more aggregate noise it will have in its estimation. For SSVS-1, the second factor dominates, but for SSVS-2, it's a balancing game, with the first factor winning for small l values and the second factor dominating for larger l values.

Comparing the best results in Table 4, i.e., the column of $l = 1$, with the best results in Table 5, i.e., the column of $l = 4$, SSVS-2 clearly outperforms SSVS-1 in estimation accuracy. While the average absolute error increases with flow size, as we show in Table 9, the average relative error actually decreases rapidly with flow size, suggesting good accuracy for large flows as well.

Next we evaluate the impact of the noise interval on estimation accuracy of SSVS-2. We vary the noise interval from $\pm 2w$, $\pm w$, $\pm \frac{w}{2}$, $\pm \frac{w}{4}$, to $\pm \frac{w}{8}$. Table 10 shows that the errors first decrease as the noise interval decreases, bottom at $\pm \frac{w}{4}$, and then increase as the noise interval further increases. This is the aggregate result of two factors. With a smaller noise interval, noise outliers are less likely to be included in the interval for size estimation, which helps improve accuracy. But in the meantime there are fewer counters in the interval and thus fewer data from the flow under query are included in the estimation, which reduces accuracy.

4.3 Accuracy Comparison between SSVS-2 and Prior Work

We now compare our best sketch SSVS-2 with the prior work in terms of estimation accuracy. For SSVS-2, $l = 4$ and the noise interval is set to $\pm w/4$. The parameter settings for the prior work are discussed in Section 4.1. Table 6 presents the absolute errors of the size estimations by various sketches in different flow size ranges. We do not include the results for CM, CS and CU because they perform worst than their variants in the table that use efficient counter designs. The most relevant work is the single update sketches, RCS and RCS-AC, which were designed to minimize per-packet processing overhead, but have much lower accuracy, compared to multi-update sketches. SSVS-2 achieves far better accuracy than

Table 6: Comparison of various sketches on average absolute error, on the CAIDA data set, under 1Mbit memory and $l = 4$.

Size range	CM-SC	CS-SC	**CU-SC**	CM-AC	CS-AC	CU-AC	CM-CE	**CS-CE**	CU-CE	CM-SA	CS-SA	CU-SA	**RCS**	**RCS-AC**	**SSVS-2**
[1,10]	165.8	105.4	**81.6**	166.3	91.9	108.0	112.3	**65.2**	61.5	208.8	92.3	107.7	**731.4**	**626.2**	**60.8**
[11,100]	172.9	113.4	**70.2**	173.1	101.5	93.1	111.9	**75.0**	47.0	209.0	103.7	93.5	**751.3**	**638.3**	**68.5**
[101,1000]	209.2	161.5	**43.5**	209.7	143.8	27.1	105.6	**107.4**	105.2	212.3	147.4	27.46	**907.7**	**754.4**	**108.8**
[1001,10000]	206.5	206.1	**14.2**	205.8	190.4	782.8	186.8	**203.7**	1371.4	210.8	172.2	12.1	**1405.0**	**1013.8**	**188.2**
≥ 10001	265.2	376.2	**14.3**	231.4	322.7	9285.0	1693.9	**873.0**	10215.0	192.4	185.1	2204.3	**1560.9**	**1036.1**	**241.8**

Table 7: Comparison of various sketches on average relative error, on the CAIDA data set, under 1Mbit memory and $l = 4$.

Size range	CM-SC	CS-SC	**CU-SC**	CM-AC	CS-AC	CU-AC	CM-CE	**CS-CE**	CU-CE	CM-SA	CS-SA	CU-SA	**RCS**	**RCS-AC**	**SSVS-2**
[1,10]	95.7	61.1	**47.3**	101.4	52.6	62.9	64.9	**37.3**	36.0	122.6	52.7	62.8	**421.3**	**364.5**	**35.0**
[11,100]	9.3	6.0	4.0	9.6	5.4	5.3	6.12	**4.0**	2.81	11.4	5.5	5.4	**40.4**	**34.4**	**3.6**
[101,1000]	0.97	0.72	**0.24**	0.82	0.64	0.16	0.50	**0.47**	0.28	1.00	0.65	0.16	**4.0**	**3.4**	**0.42**
[1001,10000]	0.10	0.10	**0.0090**	0.10	0.09	0.20	0.07	**0.087**	0.47	0.11	0.084	0.0069	**0.71**	**0.51**	**0.096**
≥ 10001	0.013	0.026	**0.00047**	0.017	0.017	0.46	0.083	**0.043**	0.51	0.012	0.010	0.064	**0.085**	**0.062**	**0.016**

Table 8: Comparison of various sketches on average absolute error, on the CAIDA data set, under 256Kbits memory and $l = 4$.

Size range	CM-SC	CS-SC	**CU-SC**	CM-AC	CS-AC	CU-AC	CM-CE	**CS-CE**	CU-CE	CM-SA	CS-SA	CU-SA	**RCS**	**RCS-AC**	**SSVS-2**
[1,10]	1565.3	478.5	**937.0**	1330.3	438.3	776.2	878.0	**324.6**	467.9	1327.5	444.5	791.1	**1827.7**	**1335.0**	**465.1**
[11,100]	1569.0	488.7	**931.0**	1332.7	439.4	772.7	877.4	**331.3**	450.2	1324.2	454.4	773.0	**1843.6**	**1352.1**	**478.2**
[101,1000]	1572.1	494.1	**919.4**	1331.0	450.0	757.8	876.9	**439.1**	260.8	1319.8	565.4	564.7	**2009.5**	**1341.9**	**560.4**
[1001,10000]	1581.4	586.3	**704.5**	1320.7	550.4	551.6	799.8	**616.7**	1197.6	1336.9	761.4	124.6	**2749.4**	**1490.4**	**867.9**
≥10001	1584.9	1009.7	**163.4**	1327.5	793.8	715.2	1236.1	**1272.7**	10098.9	1320.0	849.2	2214.8	**3971.7**	**2139.4**	**915.2**

Table 9: Average relative error of SSVS-2 with respect to l, under 1Mbit of memory

Flow size range	$l = 1$	$l = 2$	$\boldsymbol{l = 4}$	$l = 8$	$l = 16$	$l = 32$
[1,10]	77.0	136.2	**35.0**	45.8	81.6	111.7
[11,100]	7.4	13.1	**3.6**	4.7	8.1	11.2
[101,1000]	0.72	1.5	**0.43**	0.57	1.0	1.5
[1001,10000]	0.11	0.20	**0.097**	0.15	0.22	0.37
≥ 10001	0.013	0.037	**0.015**	0.024	0.048	0.11

Table 10: Average absolute error of SSVS-2 with respect to noise interval, under 1Mbit memory

Flow size range	$\pm 2w$	$\pm w$	$\pm w/2$	$\boldsymbol{\pm w/4}$	$\pm w/8$	$\pm w/16$
[1,10]	77.9	65.2	61.6	**60.8**	61.7	62.2
[11,100]	86.7	74.2	68.8	**68.5**	69.6	70.1
[101,1000]	132.3	115.4	112.4	**108.8**	109.8	108.5
[1001,10000]	185.9	187.6	177.7	**188.2**	196.6	190.5
≥ 10001	237.9	239.3	257.2	**241.8**	267.4	232.2

RCS and RCS-AC. Its absolute errors are even smaller than most multi-update sketches except for CU-SC and CS-CE. Comparing with CU-SC, SSVS-2 has lower errors for small flows and comparing with CS-CE, SSVS-2 has lower errors for large flows. Although its errors are higher than CU-SC for large flows, if we consider the average relative errors in Table 7, which are more relevant for large flows, they remain small (in the last two rows). Figure 1 shows that our new sketch (SSVS-2) has a smaller average (absolute) error than CU-SC over all flows; that is because there are many more small flows than large ones.

From Table 6, the average absolute errors of RCS-AC are 626.2 on flows of size [1,10] and 1036.1 on flows of size ≥ 10001. When it comes to SSVS-2, the average absolute errors are 60.8 on flows of size [1,10] and 241.8 on flows of size ≥ 10001. The errors of SSVS-2 are less than one tenth and one fourth of RCS-AC's, respectively. The advantage of SSVS-2 is more pronounced for small flows. The reason is due to the variable counter structure in Section 3.1. Recall that each counter in SSVS counts precisely until its 16 bits overflow. After that, it counts probabilistically. Because the counters of a small flow are likely to have small values, SSVS records the flow's packets more precisely than RCS-AC. As the counters of a large flow are likely to overflow into probabilistic counting, SSVS records packets less precisely than its small-flow case, but still more precisely than RCS-AC because it counts precisely up to $\pm 2^{15}$ and then counts probabilistically, whereas RCS-AC always counts probabilistically.

We continue comparing SSVS-2 to the prior work by varying the amount of memory allocated to the sketches from 256Kb, 512Kb, to 2Mb. The average absolute errors are presented in Tables 8, 11, and 12. When the memory is very tight, such as 256Kb in Table 8, CS-CE performs the best for small flows, CU-SC performs the best for large flows, while SSVS-2 is in between, whose errors are larger than CS-CE but smaller than CU-SC for small flows, while being smaller than CS-CE but larger than CU-SC for large flows. As we increase the memory, the performance of CU-SC and SSVS-2 is improved faster and outperforms CS-CE. Note that CU-SC and CS-CE are multi-update sketches that are optimized for accuracy, whereas SSVS-2 is designed to perform well both in accuracy and in overhead. Its overhead is much smaller than those of CU-SC and CS-CE, as we will show next.

4.4 Overhead Comparison between SSVS-2 and Prior Work

We compare SSVS-2 with the prior work on the CAIDA data set in terms of per-packet processing overhead with the same experiments

Table 11: Comparison of various sketches on average absolute error, on the CAIDA data set, under 512Kbits memory and $l = 4$.

Size range	CM-SC	CS-SC	**CU-SC**	CM-AC	CS-AC	CU-AC	CM-CE	**CS-CE**	CU-CE	CM-SA	CS-SA	CU-SA	**RCS**	**RCS-AC**	**SSVS-2**
[1,10]	584.0	227.2	**351.2**	491.5	205.6	297.6	318.6	**147.7**	173.0	491.0	204.4	298.6	**1296.5**	**936.2**	**183.9**
[11,100]	583.8	233.8	**347.7**	493.3	208.4	293.9	318.7	**157.2**	156.6	490.9	217.2	282.4	**1300.1**	**946.5**	**194.2**
[101,1000]	584.7	237.0	**337.3**	492.7	220.9	280.9	313.2	**218.0**	99.6	491.0	300.0	139.3	**1423.8**	**957.4**	**261.4**
[1001,10000]	580.6	336.0	**193.1**	491.7	298.8	139.5	304.4	**342.4**	1326.1	496.3	403.5	37.5	**2161.0**	**1056.3**	**404.7**
≥10001	588.8	514.2	**47.9**	478.8	421.4	756.3	1372.0	**1030.3**	10206.4	491.8	330.5	2215.7	**2479.6**	**1639.5**	**473.4**

Table 12: Comparison of various sketches on average absolute error, on the CAIDA data set, under 2Mbits memory and $l = 4$.

Size range	CM-SC	CS-SC	**CU-SC**	CM-AC	CS-AC	CU-AC	CM-CE	**CS-CE**	CU-CE	CM-SA	CS-SA	CU-SA	**RCS**	**RCS-AC**	**SSVS-2**
[1,10]	37.1	42.0	**21.8**	60.8	39.2	37.6	38.1	**27.4**	20.8	60.7	39.4	37.2	**648.4**	**472.5**	**19.9**
[11,100]	37.0	44.3	**18.8**	60.7	41.0	34.4	37.7	**33.8**	12.7	60.7	47.8	26.3	**667.3**	**476.9**	**25.6**
[101,1000]	37.9	49.0	**13.8**	60.6	47.0	26.2	36.9	**53.8**	122.0	61.0	66.2	6.0	**799.2**	**480.1**	**46.2**
[1001,10000]	71.7	74.4	**10.2**	60.7	65.2	6.1	198.1	**150.6**	1390.5	61.5	79.6	3.8	**1208.1**	**603.2**	**105.8**
≥10001	71.9	88.5	**4.3**	68.5	90.1	795.2	1633.4	**877.3**	10250.3	84.5	113.8	2214.0	**1300.0**	**913.6**	**189.4**

Table 13: Comparison of various sketches on per-packet processing overhead, on the CAIDA data set, under 1Mbit memory and $l = 4$

Per-packet overhead	CM-SC	CS-SC	**CU-SC**	CM-AC	CS-AC	CU-AC	CM-CE	**CS-CE**	CU-CE	CM-SA	CS-SA	CU-SA	**RCS**	**RCS-AC**	**SSVS-2**
processing time (ns)	266	268	**728**	410	409	765	278	**289**	368	388	399	1937	**71**	**121**	**73**
memory accesses	$O(l)$	$O(l)$	$\boldsymbol{O(l)}$	$O(l)$	$O(l)$	$O(l)$	$O(l)$	$\boldsymbol{O(l)}$	$O(l)$	$O(l)$	$O(l)$	$O(l)$	$\boldsymbol{O(1)}$	$\boldsymbol{O(1)}$	$\boldsymbol{O(1)}$
hashes	l	l	$\boldsymbol{l}$	l	l	l	l	$\boldsymbol{l}$	l	l	l	l	**1**	**1**	**1**
counter updates	l	l	$\boldsymbol{O(l)}$	$O(l)$	$O(l)$	$O(l)$	$O(l)$	$\boldsymbol{O(l)}$	$O(l)$	$O(l)$	$O(l)$	$O(l)$	**1**	$\boldsymbol{\leq 1}$	$\boldsymbol{\leq 1}$

Table 14: Comparison of various sketches on throughput in million packets per second (Mpps), on the CAIDA data set, under 1Mbit memory and $l = 4$

Sketch	CM-SC	CS-SC	**CU-SC**	CM-AC	CS-AC	CU-AC	CM-CE	**CS-CE**	CU-CE	CM-SA	CS-SA	CU-SA	**RCS**	**RCS-AC**	**SSVS-2**
Throughput (Mpps)	3.76	3.73	**1.37**	2.44	2.44	1.31	3.59	**3.46**	2.71	2.57	2.51	0.52	**14.08**	**8.26**	**13.70**

as in the previous subsection. Table 13 presents the average per-packet processing times of various sketches; see the second row. The single update sketches, RCS and SSVS-2, have similar processing times, which are less than the processing time of RCS-AC, which are in turn far less than the times of the multi-update sketches. In particular, when we compare SSVS-2 with CS-CE and CU-SC, their processing times are 73ns, 289ns and 728ns, respectively, almost ten-fold difference between SSVS-2 and CU-SC. The reason for CU-SC, the best in overall accuracy among the prior work, to have much larger processing time is because it has to decode l self-adjusting counters before finding the smallest counter(s) for update. Generally speaking, the overhead comparison between a multi-update sketch and a single update sketch is $O(l)$ v.s. $O(1)$ in terms of number of memory accesses, number of hash computations, and number of counter updates. One interesting observation is that SSVS-2 incurs less than one counter update per packet on average. That is because its active counters are updated *probabilistically*; see Section 3.3 for details. Table 14 presents the throughput that each sketch can handle in millions of packets per second under our experimental setting. The throughput of SSVS-2 is about 10 times that of CU-SC.

Combining the experimental results on the accuracy of all sketches presented in Section 4.3, we provide a summary of the performance of SSVS-2 and the prior work in Figure 1. In this analysis, we use the average absolute error of all flows as the overall accuracy metric, represented by the x-axis. Additionally, we use the per-packet processing time as the metric for recording overhead, represented by the y-axis. For consistency, we set the memory allocation to 1Mb. Figure 1 demonstrates that SSVS-2 achieves slightly better overall accuracy compared to the most accurate existing method, while significantly reducing recording overhead. Furthermore, in comparison to the most lightweight existing sketch, SSVS-2 incurs similar recording overhead, while improving measurement accuracy multi-fold.

4.5 Comparison between SSVS-2 and Prior Work on Web Data Set

We present our evaluation results on the web data set. Table 15 compares SSVS-2 with the prior work in terms of the average absolute error under memory 1Mb. Table 16 gives the average relative errors. We can draw the same conclusion as those from the CAIDA data set: CU-SC performs the best for large flows, CS-CE performs the best for small flows, and SSVS-2 performs in between. SSVS-2 is much more accurate than the existing single update sketches, that is, RCS and RCS-AC. Table 17 compares SSVS-2 with the prior work in terms of processing time per data item and throughput in Mpps. Again, the overhead of SSVS-2 is similar to RCS and much better than multi-update sketches, an order of magnitude better than CU-SC.

Table 15: Comparison of various sketches on average absolute error, on the web data set, under 1Mbit memory and $l = 4$.

Size range	CM-SC	CS-SC	**CU-SC**	CM-AC	CS-AC	CU-AC	CM-CE	**CS-CE**	CU-CE	CM-SA	CS-SA	CU-SA	**RCS**	**RCS-AC**	**SSVS-2**
[1,10]	290.7	234.0	**146.4**	262.1	189.9	163.7	166.3	**128.3**	94.9	262.7	190.4	163.6	**1475.6**	**1064.4**	**135.7**
[11,100]	299.3	244.8	**132.1**	261.5	200.2	143.1	166.4	**142.1**	74.8	263.8	200.0	144.5	**1494.4**	**1075.4**	**149.7**
[101,1000]	319.8	317.2	**81.2**	263.4	265.2	44.3	160.9	**186.2**	99.0	261.5	265.0	46.6	**1715.1**	**1202.6**	**205.0**
[1001,10000]	319.7	403.4	**6.7**	250.0	365.6	882.2	227.8	**328.1**	1472.5	263.4	348.2	1.27	**2563.9**	**1735.8**	**300.6**
≥ 10001	352.3	1112.4	**0.25**	401.7	527.8	10450.8	1709.0	**1088.6**	11451.3	257.2	364.7	3340.4	**2975.4**	**1970.2**	**455.3**

Table 16: Comparison of various sketches on average relative error, on the web data set, under 1Mbit memory and $l = 4$.

Size range	CM-SC	CS-SC	**CU-SC**	CM-AC	CS-AC	CU-AC	CM-CE	**CS-CE**	CU-CE	CM-SA	CS-SA	CU-SA	**RCS**	**RCS-AC**	**SSVS-2**
[1,10]	232.8	187.3	**117.6**	210.1	152.1	131.5	133.1	**102.6**	76.4	210.7	151.8	131.4	**1181.7**	**852.6**	**108.6**
[11,100]	14.1	11.4	**6.5**	12.5	9.4	7.1	7.96	**6.62**	3.8	12.6	9.6	7.2	**71.2**	**50.9**	**7.1**
[101,1000]	1.5	1.4	**0.47**	1.2	1.1	0.28	0.78	**0.83**	0.26	1.2	1.2	0.30	**7.7**	**5.4**	**0.89**
[1001,10000]	0.15	0.19	**0.0045**	0.12	0.17	0.20	0.08	**0.14**	0.46	0.13	0.17	0.00094	**1.1**	**0.80**	**0.14**
≥ 10001	0.019	0.044	**0.000020**	0.019	0.026	0.45	0.07	**0.051**	0.51	0.014	0.023	0.081	**0.16**	**0.11**	**0.025**

Table 17: Comparison of various sketches on throughput in million packets per second (Mpps), on the web data set, under 1Mbit memory and $l = 4$

Per-packet overhead	CM-SC	CS-SC	**CU-SC**	CM-AC	CS-AC	CU-AC	CM-CE	**CS-CE**	CU-CE	CM-SA	CS-SA	CU-SA	**RCS**	**RCS-AC**	**SSVS-2**
processing time (ns)	280	281	**756**	428	431	796	289	**299**	386	402	414	2109	**75**	**128**	**77**
Throughput (Mpps)	3.57	3.56	**1.32**	2.33	2.32	1.25	3.46	**3.34**	2.59	2.48	2.41	0.47	**13.33**	**7.81**	**12.90**

Table 18: Comparison of various sketches on average absolute error of all flows, on the Zipf data set with varying skewness, under 1Mbit memory and $l = 4$.

Skewness	CM-SC	CS-SC	**CU-SC**	CM-AC	CS-AC	CU-AC	CM-CE	**CS-CE**	CU-CE	CM-SA	CS-SA	CU-SA	**RCS**	**RCS-AC**	**SSVS-2**
0	1813.4	108.1	**904.6**	1590.0	105.5	799.7	1127.4	**92.9**	566.2	1591.6	104.8	800.0	**164.1**	**210.3**	**116.1**
0.25	1792.7	112.2	**900.6**	1571.4	108.5	797.1	1112.2	**96.2**	562.7	1574.0	108.5	797.0	**168.0**	**211.7**	**120.7**
0.50	1661.2	134.7	**861.8**	1451.1	127.0	763.0	1021.3	**110.1**	530.6	1450.9	127.4	764.2	**234.9**	**232.2**	**142.5**
0.75	1251.5	159.8	**669.9**	1086.7	147.3	599.3	754.0	**121.2**	409.4	1086.1	147.0	596.5	**1096.0**	**788.2**	**156.1**
1	549.3	131.0	**292.6**	471.9	113.5	278.0	321.2	**88.0**	193.2	480.2	121.9	275.5	**5924.2**	**4657.3**	**93.2**
1.25	59.6	53.5	**26.5**	91.0	44.1	87.4	62.4	**34.2**	72.3	130.4	84.6	97.5	**15403.1**	**13116.2**	**24.2**
1.5	3.6	15.4	**1.6**	16.9	15.8	150.4	33.7	**19.3**	155.9	217.2	219.5	226.7	**23516.7**	**21199.8**	**7.4**

4.6 Comparison between SSVS-2 and Prior Work on the Zipf Data Set

Finally, we present our evaluation results on the synthetic Zipf dataset under memory 1 Mb. We use the average absolute error of all flows as the accuracy metric. The results are shown in Table 18. Both SSVS-2 and CS-CE consistently achieve higher accuracy than others. CU-SC only achieves superior accuracy when the skewness of the dataset is very large (greater than 1.25). This discrepancy arises due to CU's positively biased estimation, and it works better for large flows. When the skewness increases, there are fewer larger flows. RCS and RCS-AC exhibit diminishing accuracy as the skewness level increases. As explained in Section 2.3, this behavior can be attributed to their utilization of large l values. In particular, the impact of larger flows corrupting a greater number of counters is amplified when confronted with higher levels of skewness. As we have explained earlier, when the skewness is beyond 1.5, there are so few flows that sketches no longer make sense, because we can use a small hash table.

5 CONCLUSION

This paper designs an accurate and fast sketch called SSVS for per-flow size measurement. The design of SSVS contains several novel components: (1) a new variable counter, (2) a recording operation that requires only one hash and at most one counter update for recording each packet, which is key in both noise cancellation (i.e. accuracy) and efficiency, and (3) a query method, based on fine-tuned noise intervals, which blocks out counters that are heavily impacted by noise. Compared to the most accurate sketches, i.e., multi-update sketches, SSVS reduces the recording overhead significantly, while maintaining overall comparable measurement accuracy. Compared to the most lightweight sketches, i.e., single-update sketches, SSVS is much more accurate and incurs similar recording overhead. The experimental results demonstrate that the proposed sketch achieves both high measurement accuracy and low recording overhead simultaneously.

ACKNOWLEDGMENTS

This work is supported in part by National Science Foundation under grants CNS 1909077 and CNS-2312676.

REFERENCES

[1] Lada A Adamic and Bernardo A Huberman. 2000. Power-law Distribution of the World Wide Web. *science* 287, 5461 (2000), 2115–2115.

[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. 2020. Faster and More Accurate Measurement Through Additive-error Counters. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE, 1251–1260.

[4] 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.

[5] 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.

[6] Ran Ben Basat, Gil Einziger, Roy Friedman, Marcelo C Luizelli, and Erez Waisbard. 2017. Constant Time Updates in Hierarchical Heavy Hitters. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. 127–140.

[7] Martin Breunig, Patrick Erik Bradley, Markus Jahn, Paul Kuper, Nima Mazroob, Norbert Rosch, Mulhim Al-Doori, Emmanuel Stefanakis, and Mojgan Jadidi. 2020. Geospatial data management research: Progress and Future Directions. *ISPRS International Journal of Geo-Information* 9, 2 (2020), 95.

[8] Moses Charikar, Kevin Chen, and Martin Farach-Colton. 2002. Finding Frequent Items in Data Streams. In *International Colloquium on Automata, Languages, and Programming*. Springer, 693–703.

[9] Min Chen, Shigang Chen, and Zhiping Cai. 2016. Counter Tree: A Scalable Counter Architecture for Per-flow Traffic Measurement. *IEEE/ACM Transactions on Networking* 25, 2 (2016), 1249–1262.

[10] Cisco. 2023. Cisco IOS NetFlow. http://www.cisco.com/c/en/us/products/ios-nx-os-software/ios-netflow/index.html.

[11] Graham Cormode and Shan Muthukrishnan. 2005. An Improved Data Stream Summary: the Count-min Sketch and Its Applications. *Journal of Algorithms* 55, 1 (2005), 58–75.

[12] Xenofontas Dimitropoulos, Paul Hurley, and Andreas Kind. 2008. Probabilistic Lossy Counting: An Efficient Algorithm for Finding Heavy Hitters. *ACM SIGCOMM Computer Communication Review* 38, 1 (2008), 5–5.

[13] Gil Einziger, Benny Fellman, and Yaron Kassner. 2015. Independent Counter Estimation Buckets. In *2015 IEEE Conference on Computer Communications (INFOCOM)*. IEEE, 2560–2568.

[14] Gil Einziger and Roy Friedman. 2016. Counting with Tinytable: Every Bit Counts!. In *Proceedings of the 17th International Conference on Distributed Computing and Networking*. 1–10.

[15] Cristian Estan and George Varghese. 2002. New Directions in Traffic Measurement and Accounting. In *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*. 323–336.

[16] Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. 2014. Cuckoo Filter: Practically Better Than Bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. 75–88.

[17] Julian Frommel, Cody Phillips, and Regan L Mandryk. 2021. Gathering Self-Report Data in Games through NPC Dialogues: Effects on Data Quality, Data Quantity, Player Experience, and Information Intimacy. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–12.

[18] Xiangyang Gou, Long He, Yinda Zhang, Ke Wang, Xilai Liu, Tong Yang, Yi Wang, and Bin Cui. 2020. Sliding Sketches: A Framework using Time Zones for Data Stream Processing in Sliding Windows. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 1015–1025.

[19] Xiaojie Guo, Shugen Wang, Hanqing Zhao, Shiliang Diao, Jiajia Chen, Zhuoye Ding, Zhen He, Jianchao Lu, Yun Xiao, Bo Long, et al. 2022. Intelligent Online Selling Point Extraction for E-Commerce Recommendation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 36. 12360–12368.

[20] Fang Hao, Murali Kodialam, and TV Lakshman. 2004. ACCEL-RATE: a Faster Mechanism for Memory Efficient Per-flow Traffic Estimation. In *Proceedings of the joint international conference on Measurement and modeling of computer systems*. 155–166.

[21] Chengchen Hu, Bin Liu, Hongbo Zhao, Kai Chen, Yan Chen, Chunming Wu, and Yu Cheng. 2010. Disco: Memory Efficient and Accurate Flow Statistics for Network Measurement. In *2010 IEEE 30th International Conference on Distributed Computing Systems*. IEEE, 665–674.

[22] Nan Hua, Bill Lin, Jun Xu, and Haiquan Zhao. 2008. Brick: A Novel Exact Active Statistics Counter Architecture. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. 89–98.

[23] Qun Huang, Xin Jin, Patrick PC Lee, Runhui Li, Lu Tang, Yi-Chao Chen, and Gong Zhang. 2017. Sketchvisor: Robust Network Measurement for Software Packet Processing. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. 113–126.

[24] Fangyu Li, Rui Xie, Zengyan Wang, Lulu Guo, Jin Ye, Ping Ma, and WenZhan Song. 2019. Online Distributed IoT Security Monitoring with Multidimensional Streaming Big Data. *IEEE Internet of Things Journal* 7, 5 (2019), 4387–4394.

[25] Junyi Li, Xintong Wang, Yaoyang Lin, Arunesh Sinha, and Michael Wellman. 2020. Generating Realistic Stock Market Order Streams. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 727–734.

[26] Tao Li, Shigang Chen, and Yibei Ling. 2011. Fast and Compact Per-flow Traffic Measurement through Randomized Counter Sharing. In *2011 Proceedings IEEE INFOCOM*. IEEE, 1799–1807.

[27] Tao Li, Shigang Chen, and Yibei Ling. 2012. Per-flow Traffic Measurement Through Randomized Counter Sharing. *IEEE/ACM Transactions on Networking* 20, 5 (2012), 1622–1634.

[28] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. 2016. Flowradar: A Better Netflow for Data Centers. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*. 311–324.

[29] Chuan Lin, Guangjie Han, Jiaxin Du, Tiantian Xu, and Yan Peng. 2020. Adaptive Traffic Engineering based on Active Network Measurement Towards Software Defined Internet of Vehicles. *IEEE Transactions on Intelligent Transportation Systems* 22, 6 (2020), 3697–3706.

[30] Zaoxing Liu, Ran Ben-Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. 2019. Nitrosketch: Robust and General Sketch-based Monitoring in Software Switches. In *Proceedings of the ACM Special Interest Group on Data Communication*. 334–350.

[31] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. 2016. One Sketch to Rule Them All: Rethinking Network flow Monitoring with Nnivmon. In *Proceedings of the 2016 ACM SIGCOMM Conference*. 101–114.

[32] Yi Lu, Andrea Montanari, Balaji Prabhakar, Sarang Dharmapurikar, and Abdul Kabbani. 2008. Counter Braids: A Novel Counter Architecture for Per-flow Measurement. *ACM SIGMETRICS Performance Evaluation Review* 36, 1 (2008), 121–132.

[33] Claudio Lucchese, Salvatore Orlando, Raffaele Perego, and Fabrizio Silvestri. 2023. Real-life transactional dataset. http://fimi.uantwerpen.be/data/.

[34] Yao Ma, Ziyi Guo, Zhaocun Ren, Jiliang Tang, and Dawei Yin. 2020. Streaming Graph Neural Networks. In *Proceedings of the 43rd international ACM SIGIR conference on research and development in information retrieval*. 719–728.

[35] Aniket Mahanti, Niklas Carlsson, Anirban Mahanti, Martin Arlitt, and Carey Williamson. 2013. A Tale of the Tails: Power-laws in Internet Measurements. *IEEE Network* 27, 1 (2013), 59–64.

[36] Dimitrios Melissourgos, Haibo Wang, Shigang Chen, Chaoyi Ma, and Shiping Chen. 2023. Full Version of Single Update Sketch with Variable Counter Structure. https://github.com/haiporwang/ssvs/blob/main/main.pdf.

[37] David MW Powers. 1998. Applications and Explanations of Zipf's Law. In *New methods in language processing and computational natural language learning*.

[38] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, Shan Muthukrishnan, and Jennifer Rexford. 2017. Heavy-hitter Detection Entirely in the Data Plane. In *Proceedings of the Symposium on SDN Research*. 164–176.

[39] Rade Stanojevic. 2007. Small Active Counters. In *IEEE INFOCOM 2007-26th IEEE International Conference on Computer Communications*. IEEE, 2153–2161.

[40] Andrew T Stephen and Olivier Toubia. 2009. Explaining the Power-law Degree Distribution in a Social Commerce Network. *Social Networks* 31, 4 (2009), 262–270.

[41] Toste Tanhua, Sylvie Pouliquen, Jessica Hausman, Kevin Obrien, Pip Bricher, Taco De Bruin, Justin JH Buck, Eugene F Burger, Thierry Carval, Kenneth S Casey, et al. 2019. Ocean FAIR Data Services. *Frontiers in Marine Science* 6 (2019), 440.

[42] Erez Tsidon, Iddo Hanniel, and Isaac Keslassy. 2012. Estimators Also Need Shared Values to Grow Together. In *2012 Proceedings IEEE INFOCOM*. IEEE, 1889–1897.

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

[44] Jinesh Varia, Sajee Mathew, et al. 2014. Overview of Amazon Web Services. *Amazon Web Services* 105 (2014).

[45] Haibo Wang, Melissourgos Dimitrios, and Chaoyi Ma. 2022. Source code. https://github.com/DimitrisMel/SSVS.

[46] 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).

[47] Haibo Wang, Hongli Xu, Liusheng Huang, Jianxin Wang, and Xuwei Yang. 2018. Load-balancing Routing in Software Defined Networks with Multiple Controllers. *Computer Networks* 141 (2018), 82–91.

[48] Haining Wang, Danlu Zhang, and Kang G Shin. 2002. Syn-dog: Sniffing Syn Flooding Sources. In *Proceedings 22nd International Conference on Distributed Computing Systems*. IEEE, 421–428.

[49] 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 Ssharing. *IEEE/ACM Transactions on Networking* 25, 6 (2017), 3738–3752.

[50] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. 2018. Elastic Sketch: Adaptive and Fast Network-wide Measurements. In *Proceedings of the 2018 Conference of the ACM Special*

Interest Group on Data Communication. 561–575.

[51] Tong Yang, Jiaqi Xu, Xilai Liu, Peng Liu, Lun Wang, Jun Bi, and Xiaoming Li. 2019. A Generic Technique for Sketches to Adapt to Different Counting Ranges. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 2017–2025.

[52] Tong Yang, Yang Zhou, Hao Jin, Shigang Chen, and Xiaoming Li. 2017. Pyramid Sketch: A Sketch Framework for Frequency Estimation of Data Streams. *Proceedings of the VLDB Endowment* 10, 11 (2017), 1442–1453.

[53] Yinda Zhang, Zaoxing Liu, Ruixin Wang, Tong Yang, Jizhou Li, Ruijie Miao, Peng Liu, Ruwen Zhang, and Junchen Jiang. 2021. CocoSketch: High-performance Sketch-based Measurement over Arbitrary Partial Key Query. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 207–222.

[54] Yikai Zhao, Kaicheng Yang, Zirui Liu, Tong Yang, et al. 2021. LightGuardian: A Full-Visibility, Lightweight, In-band Telemetry System Using Sketchlets.

[55] Hao Zheng, Chen Tian, Tong Yang, Huiping Lin, Chang Liu, Zhaochen Zhang, Wanchun Dou, and Guihai Chen. 2022. FlyMon: Enabling On-the-fly Task Reconfiguration for Network Measurement. In *Proceedings of the ACM SIGCOMM 2022 Conference*. 486–502.

[56] 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.

[57] Yang Zhou, Tong Yang, Jie Jiang, Bin Cui, Minlan Yu, Xiaoming Li, and Steve 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.

[58] You Zhou, Youlin Zhang, Chaoyi Ma, Shigang Chen, and Olufemi 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.

[59] You Zhou, Yian Zhou, Shigang Chen, and Youlin Zhang. 2018. Highly Compact Virtual Active Counters for Per-flow Traffic Measurement. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 1–9.