Technical Report: From CountMin to Super kJoin Sketches for Flow Spread Estimation in Data Streaming

***Abstract*—Processing data streams in real time is key to many Internet applications ranging from e-commerce and web services, to social networks, and to network traffic monitoring. Compact data structures called sketches are often used to track large, high-rate data streams and estimate their statistics. However, the traditional streaming model is unsuitable for more sophisticated applications that consider numerous sub-streams (called flows) and require the measurement of flow spread, i.e., the number of distinct elements in each flow. Most existing work modifies traditional CountMin sketches for this new task. Hence, they inherit a similar design and a commonly-used *min* operation from the traditional sketches. This paper casts doubt on these design choices, arguing that the inherited sketch design and the min operation were both intended for a different purpose and are not the best choices for flow spread estimation. Replacing the min operation with new position-aware operations, we propose the kJoin sketches that achieve much better average accuracy in spread estimation. We mathematically derive the error bound for flow spread estimates from the new sketches. Trace-driven experiments on software/GPU/FPGA platforms demonstrate that our work significantly improves estimation accuracy, streaming throughput (or recording throughput), and query throughput, when comparing with the state of the art.**

I. Introduction

Data streaming is a continuous production of data from one or multiple sources, and the data must be immediately processed to support real-time queries based on up-to-the-moment information [1]. Its growing practical importance is evident from industrial pushes (such as Amazon Kinesis Streams [2]) that enable customizable streaming applications, including web services, e-commerce and retailing, social networks, in-game player experience, geospatial services, distributed sensing, and network monitoring [1], [3], [4]. The bedrock underlying such diverse domain-specific data analyses is a set of measurement functions that extract useful information from raw data items and feed them to application software. The past two decades have witnessed continuous development of a large number of measurement functions for item frequencies [5], [6], [7], [8], [9], identifying heavy hitters [10], [11], [12], [13], [14], [15], [16], large deviations [17], persistent patterns [18], data distributions [19], etc. Most of them are built on top of a small number of fundamental streaming algorithms such as CountMin [20], Counting Bloom filters [21], Count Sketch [22], and RCS (randomized counter sharing) [23], which are the first-line processing engines for data streams, recording the presence of data items and their counts as the input to higher-order analysis and measurement.

Traditionally, a data stream D is modelled as a sequence of items, i.e., $D = \{d_1, d_2, d_1, d_3, \cdots\}$. Any item (e.g., d_1) may appear in the stream for an arbitrary number of times. The measurement is to find the frequency of each data item in the steam and the number of different data items, called *stream cardinality*. For example, consider a packet stream received by a router of an ISP network. If we use the source address in each packet as the data item, the source-address frequency is the traffic volume (i.e., the number of packets) that each of its customers (i.e., source addresses) sends through its network. The stream cardinality is the number of customers. Numerous streaming algorithms have been invented to measure such information [24], [8], [14]. To deal with large data streams at high rates, they typically employ compact data structures called *sketches* which provide estimated measurement with great efficiency. They use counters to track the item frequencies.

The above traditional model is however inadequate in supporting more sophisticated Internet applications. In the packet stream example, if we want to know more details about the network traffic from each source, such as how many destination addresses that it has contacted, which is useful in scan or worm detection [25], [26], [27], we will need a *generalized streaming model* that treats each data item as a pair $\langle f, e\rangle$, where f is a flow ID, and e is additional information that is of interest. All items carrying the same ID f form a flow (sub-stream). In the above packet stream example, f will be source address, and e will be destination address, both of which can be found in the packet header. With the generalized model, we can still measure the traditional item frequency as how many times each flow ID f (e.g., source address) appears in the stream, which is also called the *size* of flow f. Beyond that, we can now make new measurement about the *spread* of flow f, i.e., the number of *distinct* elements e in flow f. For example, we can measure the number of distinct destinations that each source has contacted. If a source's spread is too large, i.e., it sends packets to too many different destinations, then an alert is raised that the source may be performing network reconnaissance.

We give two more application examples that fit with the generalized model. Consider a data stream of items $\langle f, e\rangle$ at an Internet search engine running on a datacenter, where f is a search key and e is a source IP address that issues the search. Measuring the spreads of all search keys over time reveals the popularity of different search keys and the evolution of user interest. Consider a data stream of items $\langle f, e\rangle$ at a social

networking system, where f is a post and e is a user that accesses the post. Measuring the spreads of all posts reveals the popularity of different posts, providing information on prioritizing how the posts are pushed to user s.

Challenge: Measuring the number of distinct elements in each flow is a challenging problem. No matter how many times an element shows up in a flow, it should be counted once. Duplicate removal requires us to remember all elements in each flow that has appeared so far. In the previous packet stream example, to count the number of packets from a source, we only need a counter of 32 bits; to count the number of distinct destinations that a source has contacted (i.e., flow spread), we need a duplicate-filtering data structure [28], [17], [29], [30], which takes hundreds or thousands of bits [31]. If each flow takes that much and the number of flows is astronomical — which is the case for Internet traffic, the total memory requirement will be huge. The existing work has taken two solution paths. The first path is based on virtual sketches [32], [33], which however has poor query performance [31]. To address query performance, the second path is to modify the existing sketches for item frequencies [21], [20], [34] by replacing their counters with bitmaps (or other duplicate-filtering data structures) for flow spread estimation [35], [31], [36], [37]. They all follow a similar design structure that hashes each flow to multiple bitmaps, which are shared by other flows, thus recording their elements as well and resulting in inter-flow noise. They rely on the *min operation* to remove inter-flow noise due to bitmap sharing. They estimate the spread of a flow as the minimum spread estimated from the flow's multiple bitmaps [31], [36], [37]; this smallest spread carries the least noise from other flows.

Contributions: This paper casts doubt on the fundamental design choices of existing works on flow spread estimation in their *sketch design structure* and their *min operation*, which are inherited from a different measurement of item frequency — we believe that what's suited there is not the best here.

First, the min operation treats each bitmap (or any other duplicate-filtering data structure) as a whole, while ignoring the fact that a bitmap has internal details that a counter does not have. We observe that as flow elements are recorded by individual bits in each bitmap, different bit positions of recording can help us remove inter-flow noise across bitmaps. Based on this observation, we design a new set of *kJoin sketches* that replace the min operation with position-aware operations (which are specific to bitmaps or other duplicate-filtering data structures [28], [17], [29], [30] in use).

Second, we observe that their design structure of sharing at the bitmap level causes inter-flow noise concentration. Based on this observation, we design a new set of *super kJoin sketches* that logically separate the bitmaps used by different flows and disburse inter-flow noise more evenly among the flows, allowing more accurate removal of such noise.

Third, through theoretical analysis, software/GPU/FPGA implementations, and trace-driven experiments, we demonstrate that the new sketches significantly outperform the existing works.

II. Background

A. Generalized Data Streaming Model

A data stream is modelled as a sequence of data items $\langle f, e\rangle$, where f is a flow ID and e is an element of interest. All items that carry the same flow ID f form a flow, referred to as flow f for convenience. Consider a packet stream where we treat all packets from the same source address as a flow. Then f is source address. If we are interested in how many destination addresses that the source has contacted, then e is destination address.

Item $\langle f, e\rangle$ may appear in the stream for an arbitrary number of times. For example, there can be an arbitrary number of packets sent from the same source to the same destination. Hence, we consider each flow f as a *multi-set* that may contain multiple occurrences (duplicates) of any element e. The size of flow f, denoted as $|f|$, is defined as the total number of elements in the flow, including duplicates. It is in fact a measure of item frequency, i.e., the frequency of flow ID f in the stream. The spread of flow f, denoted as $||f||$, is defined as the number of distinct elements in the flow, after duplicates are removed. For example, consider a data stream of $\langle f_1, e_1\rangle$, $\langle f_1, e_2\rangle$, $\langle f_1, e_1\rangle$, $\langle f_2, e_3\rangle$. We have two flows, f_1 with three elements $\{e_1, e_2, e_1\}$, and f_2 with one element $\{e_3\}$, where $|f_1| = 3$, $||f_1|| = 2$, and $|f_2| = ||f|| = 1$. For another example, consider a packet flow of 1,000 packets from the same source to the same destination. Its size is 1,000 and its spread is 1.

The problem in this paper is to design sketches that can measure both flow size and flow spread, that are compact, efficient and accurate, and that can support real-time online queries as well as multi-stream measurement. It is known that if a sketch can measure spread, it can measure size [33], [31]. Our technical description will focus on spread.

B. CountMin and Counting Bloom Filter

We begin with a review of CountMin [20] and counting Bloom filter [34], which are used for item frequencies (or flow sizes) and can be modified for flow spreads.

CountMin uses k arrays of m counters, which are denoted as $C_i[j], 0 \le i < k, 0 \le j < m$ and initialized to zeros. For each arrival data item $\langle f, e\rangle$, we hash f to k counters, one in each array, and increase those counters by one, i.e., $C_i[H_i(f)] = C_i[H_i(f)] + 1$, $0 \le i < k$, where $k > 1$ and $H_i(\cdot)$, $0 \le i < k$, are independent hash functions. For brevity, we omit the modulo operation on $H_i(f)$ that keeps it in the range $[0, m)$. To query for $|f|$, we again hash f to its k counters and use their minimum value, $\min\{C_i[H_i(f)], 0 \le i < k\}$, as an estimate for $|f|$. The reason for the min operation is that each counter $C_i[H_i(f)]$ measures the sum of $|f|$ and the combined size of other flows (noise) that are also hashed to this counter. It is equal to or greater than $|f|$. Hence, taking the minimum gives the best estimate.

For Counting Bloom filter, the only difference is that it combine C_i into one array and randomly hashes f to k counters in the array. It cannot ensure that the k counters for flow f are

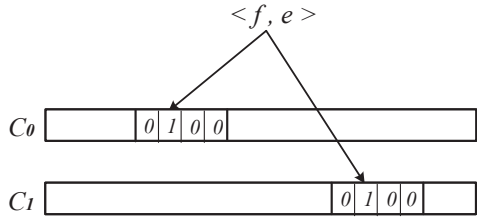


Fig. 1: $C_i, 0 \leq i < k$ are of bitmaps. A flow f is hashed to k bitmaps, one in each array, where $k = 2$ in this example. When receiving $\langle f, e\rangle$, we hash e to a bit position and set that bit in each bitmap of the flow to one, where it is the second bit in this example.

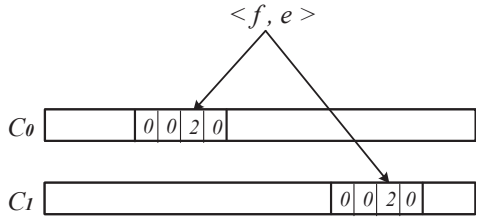


Fig. 2: $C_i, 0 \leq i < k$ are k arrays of HLL-maps. A flow f is hashed to k HLL-maps, one in each array, where $k = 2$ in this example. When receiving $\langle f, e\rangle$, we hash e to a HLL register position and set that register in each HLL-map of the flow to the maximum one between geometric hash value $G(e)$ and the original value of the register, where the second register is set to 2 in this example.

different, whereas the chance for flow f to occupy a counter all by itself is slightly larger — when that happens, the minimum counter value will be precisely $|f|$.

C. cSketch and bSketch

CountMin [20] and Counting Bloom filter [34] present specially designed data structures which use counters as units for tracking flow sizes. However, they are not suitable for estimating flow spreads. That is because a counter cannot remember individual elements for duplicate removal. To address this problem, a natural thought is to simply replace each counter in B with another data structure such as bitmap [38], FM-map [29] or HLL-map [30] that can remove duplicates. This idea produces two families of solutions for flow spread measurement, called bSketch and cSketch [37], [39], [31].

We briefly introduce for cSketch and bSketch by giving two examples, cSkt(bitmap) and cSkt(HLL). cSkt(bitmap) replaces each counter in CountMin with a bitmap as Fig. 1 shows. $C_i, 0 \leq i < k$ are k arrays of bitmaps, each of l bits. For each data item $\langle f, e\rangle$, we hash f to k bitmaps, one in each array, then hash e to a bit in each bitmap, and set that bit to one:

$$C_i[H_i(f)][H(e)] = 1, \quad 0 \leq i < k, \tag{1}$$

All duplicates of element e will be hashed to the same bit, which is already one, and thus have no additional impact on C_i, equivalent to duplicate removal. According to [28], from each bitmap $C_i[H_i(f)]$, $0 \leq i < k$, we can estimate the number $\hat{c}_{f,i}$ of distinct elements that it records as follows:

$$\hat{c}_{f,i} = -l \ln(1 - \frac{u_{f,i}}{l}), \tag{2}$$

where $u_{f,i} = \sum_{0 \leq j < l} C_i[H_i(f)][j]$ and l is the length of each array. Note that $u_{f,i}$ is the number of ones in bitmap $C_i[H_i(f)]$ and that the estimate $\hat{c}_{f,i}$ includes the elements from flow f and those from other noise flows also hashed to this bitmap. The final estimate $\hat{c}_f$ for spread $||f||$ is the minimum estimate from the k bitmaps.

$$\hat{c}_f = \min\{\hat{c}_{f,i}, 0 \leq i < k\}. \tag{3}$$

Similar to cSkt(bitmap), cSkt(HLL) replaces each counter in CountMin with a HLL-map as Fig. 2 shows. $C_i, 0 \leq i < k$ are k arrays of HLL-maps, each of l HLL registers which are counters with b bits. For each data item $\langle f, e\rangle$, we hash f to k HLL-maps, one in each array, then hash e to a HLL register in each HLL-map, and set that register as follows:

$$C_i[H_i(f)][H(e)] = \max\{G(e), C_i[H_i(f)][H(e)]\}, \quad 0 \leq i < k, \tag{4}$$

where $G(\cdot)$ is a geometric hash function which has a probability of $\frac{1}{2}$ to produce 1, a probability of $\frac{1}{2^2}$ to produce 2, ... , a probability of $\frac{1}{2^t}$ to produce t.

The duplicates are removed automatically since the same e will always mapped to the same register and produce the same $G(e)$. From each HLL-map, we can estimate the number $\hat{c}_{f,i}$ of distinct elements that it records as follows [30] :

$$\hat{c}_{f,i} = \alpha_l l^2 / (\sum_{j=0}^{l-1} 2^{-C_i[H_i(f)][j]}). \tag{5}$$

Similar to cSkt(bitmap), it chooses the estimate contains the least elements from other noise flows as the finally estimate:

$$\hat{c}_f = \min\{\hat{c}_{f,i}, 0 \leq i < k\}. \tag{6}$$

Instead of bitmap and HLL-map, we may use other duplicate-removing data structures as well, such as FM-map [29]. Hence, one can turn a CountMin into a family of CountMin sketches, called cSketch [31], where $C_i, 0 \leq i < k$ is k arrays of plug-ins, which may be bitmaps, FM-maps [29] or HLL-maps [30], each plug-in producing an instance sketch of the family, denoted as cSkt(bitmap), cSkt(FM) and cSkt(HLL), respectively [31]. Similarly, one can turn counting Bloom Filter [34] into a family of sketches, denoted as bSkt(bitmap), bSkt(FM) and bSkt(HLL), which replace each counter with a bitmap, FM-map, and HLL-map, respectively. All of them use the min operation in (6) for the final spread estimate $\hat{c}_f$, while their formulas of computing $\hat{c}_{f,i}$ from $C_i[H_i(f)]$ differ.

cSketches and cSketches successfully give solutions for flow spread measurement based on a natural idea of replacing counters with other plug-ins. However, this kind of replacing ignores the differences between counters and those plug-ins for flow spread measurement. This ignorance results in significant inaccuracy by cSketches and bSketches that could be avoided, which we explain in Section III-A.

D. Virtual Sketches, Randomized Sketches, and Other Related Work

Another line of research for spread estimation is virtual sketches such as CSE [32], vHLL [33] and vSketch [31]. While bSketch and cSketch use the min operation to reduce the influence of noise flows which share the same plug-ins, the virtual sketches calculate an average noise value over the whole data structure and remove it from the estimate. In addition, they have to perform numerous hashes per query (128 or more). Although virtual sketches are usually more accurate than bSketch and cSketch, they do not support online spread queries for real-time applications because the large number of hashes and the need of scanning the whole data structure to compute the average noise for each query are too expensive. Instead, they are typically designed to support offline queries where the virtual sketches are offloaded to a server after each measurement period and the server will answer non-real time spread queries [32], [33], [31]. Unlike online queries, the offline queries are made against the past virtual sketches whose average noise does not change and thus can be calculated once; therefore, the overhead of offline queries is much smaller than that of online queries. CountHLL sketch [40] also performs better than bSketch and cSketch in terms of estimation accuracy. However, it requires performing complex empirical analysis over its whole data structure, and is therefore not suitable for online spread queries as well.

The randomized error-reduction sketch (rSkt2) [41] uses a primiary/complement data structure design for noise cancelling. Compared to bSketch and cSketch, its greatest advantage is that it only needs to record each data item once, which improves the recording throughput. While the primary/complement design wins on reading throughput, this paper will demonstrate that the CountMin structure can be redesigned to provide better accuracy for large flows and smaller worse-case error than rSkt2. We will show that rSkt2 and the sketches proposed in this paper provide different performance tradeoffs and have their respectively target application areas, which will be revealed through our experimental study.

UnivMon [24] and Elastic Sketch [14] estimate the number of different flows but do not consider each flow's spread. The recent work of SpreadSketch [35] also extends from CountMin by replacing each counter with a multi-resolution bitmap [42] and other auxiliary data structures for super spreader detection.

III. kJoin Sketches

We introduce a new family of kJoin sketches for measuring flow spread. They also serve as the stepping stone for super kJoin in the next section.

A. Motivation

The existing works have followed the most natural path to extend CountMin (or counting Bloom filer) by simply replacing each counter in $C_i, 0 \leq i < k$ with different plug-ins, while keeping the min operation and the overall structure intact. This paper casts doubt on these two fundamental design choices. We will first question the idea of applying the min operation — which was originally designed based on counters — to other plug-ins. Its pitfall leads to a new family of *kJoin sketches*. In the following section, we will question the idea of keeping the overall structure of CountMin for flow spread estimation. It leads to our second family of *super kJoin sketches*.

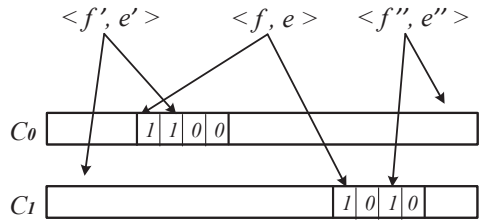


Fig. 3: cSkt(bitmap) does not exploit the position information for noise removal. The two bitmaps each have a noise bit at a different location, which could help remove them.

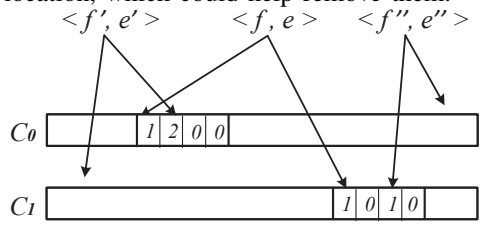


Fig. 4: cSkt(HLL) does not exploit the position information for noise removal. The two HLL-maps both have noise at a different register, which could help remove them.

Consider an example of cSkt(bitmap) [31] with $k = 2$ in Fig. 3. Flow f's spread is one. Its single element is hashed to 2 bitmaps and sets a bit in each of them to one — without losing generality, suppose that's the first bit in the two bitmaps, i.e., $C_0[H_0(f)][0] = 1$ and $C_1[H_1(f)][0] = 1$. Let f' and f'' be two other flows of spread one, each sharing a bitmap with f due to hash collision. Suppose the element of f' is hashed to the second bit and the element of f'' is hashed to the third bit, causing noise in the two bitmaps of f, with $C_0[H_0(f)] = 1100...$ and $C_1[H_1(f)] = 1010....$ Both of them have two bits of ones. Hence, the two estimates, $\hat{c}_{f,0}$ and $\hat{c}_{f,1}$, from them will be the same and carry the same amount of noise, which makes the min operation of (6) ineffective in noise removal as $\hat{c}_f = \hat{c}_{f,0} = \hat{c}_{f,1}$.

In another example of cSkt(HLL) [31] with $k = 2$ in Fig. 4, flow f's spread is one and its single element is hashed to two HLL-maps and sets a register t in each of them to one. If we use a single HLL-map to estimate f's spread, the values in it should be $1000....$ However, in the two HLL-maps in cSkt(HLL), the values are $1200...$ and $1010...$, respectively. Due to the recording of elements belonging to f' and f'', both HLL-maps will have noise when producing the estimate. Even if we choose the one with less noise as the final estimate in cSkt(HLL), we still have noise.

B. Replacing Min with Position-Aware Operations

The reason for the min operation to not work well in the above example is that it relies only on the number of ones in each bitmap (which sums up both information in the first bit and noise in other bits), but it disregards the position of noise, which is in the *second* bit of $C_0[H_0(f)]$ and the *third* bit of $C_1[H_1(f)]$ — such position information did not exist with a

counter in the original CountMin design, but does exist with a bitmap. To exploit this new position information, we should no long use the min operation universally for all plug-ins, but tailor the operation for each specific plug-in type to join the information from $C_i[H_i(f)]$, $0 \le i < k$, in a position-aware manner for more effective noise removal. This change in design results in our first family of new sketches called kJoin.

When $C_i, 0 \le i < k$ are k arrays of bitmaps, we will use the bit-wise AND to replace min. We first combine the k bitmaps of f into one, denoted as A_f, where

$$A_f[j] = C_0[H_0[f]][j] \wedge ... \wedge C_{k-1}[H_{k-1}[f]][j], \quad 0 \le j < l.$$

Consider the example we discussed before that two bitmaps of f are set to $C_0[H_0(f)] = 1100...$ and $C_1[H_1(f)] = 1010...$ due to hash collisions with flows f' and f''. After this bit-wise AND operation, the combined bitmap is $A_f = C_0[H_0(f)] \wedge C_1[H_1(f)] = 1000...$, which eliminates the noise caused by f' and f''.

A_f is the combined bitmap. After noise is removed by bit-wise AND, according to [28], we can estimate the number of distinct elements recorded in the bitmap as follows:

$$\hat{c}_f = -l \ln(1 - \frac{u_f}{l}), \tag{7}$$

where

$$u_f = \sum_{0 \le j < l} A_f[j].$$

The above design is called kJoin(bitmap). Below we give two more sketches for flow spread measurement as additional examples in this family.

• *kJoin(HLL),* in which $C_i, 0 \le i < k$ are k arrays of HLL-maps [30], each consisting of l HLL registers, and each register is 5 bits long. Flow f is hashed to k HLL-maps, $C_i[H_i(f)]$, $0 \le i < k$. To record a data item $\langle f, e \rangle$, we assign

$$C_i[H_i(f)][H(e)] = \max\{C_i[H_i(f)][H(e)], G(e)\}, \ 0 \le i < k,$$

where $H(e)$ is a uniformly distributed hash function and $G(e)$ is a geometric hash function whose value is v with probability 2^{-v} for $v \ge 1$. When querying the spread of flow f, we first join $C_i[H_i(f)]$, $0 \le i < k$, into a single HLL-map A_f through *register-wise min* as follow:

$$A_f[j] = \min\{C_i[H_i(f)][j], \quad 0 \le i < k\}, \quad \forall 0 \le j < l. \tag{8}$$

$$\hat{c}_f = \alpha_l l^2 / (\sum_{j=0}^{l-1} 2^{-A_f[j]}), \tag{9}$$

where α_l is a pre-computed constant determined by l. Refer to [30] for details about the above formula and how to compute $G(e)$. We stress that register-wise min is different from the min operation of the existing works, such as cSkt(HLL) and bSkt(HLL) [31], which do not perform min at the register level as in (8) but instead perform min on k spread estimates as in (6). Consider the example we discussed before that two HLL-maps of f are set to $C_0[H_0(f)] = 2100...$ and $C_1[H_1(f)] = 1010...$

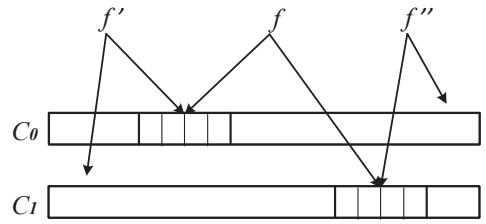


Fig. 5: kJoin(bitmap) with $k = 2$. $C_i, 0 \le i < 2$ are 2 arrays of bitmaps. Flow f is hashed to 2 bitmaps, one in each array. Hash collision with f' and f'' causes noise in spread estimation. If both f' and f'' are large, the final estimate will carry large error.

due to hash collisions with flows f' and f''. After this register-wise min operation, the combined HLL-map is $A_f = 1000...$, which eliminates the noise caused by f' and f''. The proposed kJoin(HLL) performs considerably better than cSkt(HLL) and bSkt(HLL) as our experiments will show.

• *kJoin(FM),* in which $C_i, 0 \le i < k$ are k arrays of FM-maps [29], each consisting of l FM registers, and each register is 32 bits long. Flow f is hashed to k FM-maps, $C_i[H_i(f)]$, $0 \le i < k$. To record a data item $\langle f, e \rangle$, we set the following bits to ones.

$$C_i[H_i(f)][H(e)][G(e)] = 1, \quad 0 \le i < k,$$

where $C_i[H_i(f)]$ is an FM-map, $C_i[H_i(f)][H(e)]$ is one of its register, and $C_i[H_i(f)][H(e)][G(e)]$ is a bit of the register.

For querying the spread of flow f, we first join $C_i[H_i(f)]$, $0 \le i < k$, into a single FM-map A_f through *register-wise AND*. $\forall 0 \le j < l, \ 0 \le q < 32$,

$$A_f[j][q] = C_0[H_0(f)][j][q] \wedge ... \wedge C_{k-1}[H_{k-1}(f)][j][q].$$

Define a function $\rho(\cdot)$ that counts the number of consecutive ones starting from the lowest order bit in the input. We then apply the following formula on A_f for our final spread estimate.

$$\hat{c}_f = l 2^{\sum_{0 \le j < l} \rho(C^f[j])/l} / \phi, \tag{10}$$

where ϕ is a pre-computed constant. Refer to [29] for details about the above formula.

IV. Super kJoin Sketches

We further improve the design structure of kJoin for better performance.

A. Motivation

We observe in our experiments that spread estimation produced by kJoin(bitmap) for small flows sometimes carries very large error (Figure 10 in Section VI-C). We explain the reason via an example in Figure 5, where $k = 2$ and a small flow f is hashed to two bitmaps. Due to hash collision, other flows may be hashed to the same bitmaps. In this example, flows f' and f'' each share one bitmap with f. The two bitmaps measure the total spread of $\{f, f'\}$ and that of $\{f, f''\}$, respectively. Therefore, f' and f'' are noise to the measurement of f. If they are both small, the noise is light in both bitmaps. If one of the noise flows is large but the other is small, when we join the two bitmaps by bit-wise AND, the noise that

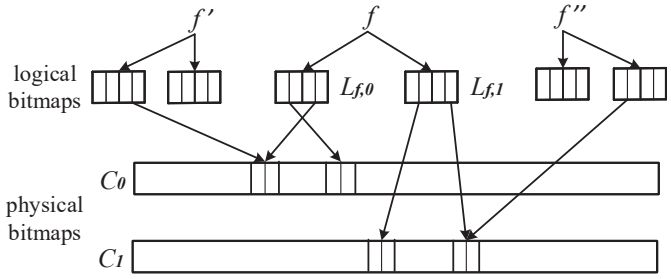


Fig. 6: skJoin(bitmap) with $k = 2, l = y \times z = 2 \times 2 = 4$. $C_i, 0 \le i < 2$ is an array of bit segments, each containing $z = 2$ bits. Flow f is mapped to 2 logical bitmaps, taking 2 bit segments from C_0, C_1, respectively. Hash collision only results in a portion of f' (f'') being noise. For example, f' shares only one bit segment with f, due to the logical bitmap construction. Even when flow f' is large, only a fraction of its elements causes noise to f.

remains in the resulting bitmap A_f will be small, which means the spread estimate from A_f will be good. However, if both noise flows are large and they set most bits in both bitmaps to ones, when we join them together to A_f, most bits in A_f will remain ones, which causes the final spread estimate to be large, carrying a large error. The same thing is true for kJoin(FM) and kJoin(HLL).

B. New Design with Extra Layer of Abstraction

To mitigate the impact of large flows, we borrow the virtualization idea from vSketch [31], but use a different sketch design, called *super kJoin(bitmap)*, abbreviated as skJoin(bitmap), whose structure is shown in Figure 6. The new design avoids the problem of high overhead for online spread queries (see Section II-D) as vSketch does. Their difference will be detailed in the next subsection.

The physical data structure of skJoin(bitmap) is k arrays of m bits each, $C_i, 0 \le i < k$. Each bit array is divided into segments of z bits, where z is a parameter that will control the hash overhead per query, as we will explain shortly. The jth segment in C_i contains bits $C_i[j \times z + t]$, $0 \le t < z$.

Flow f is hashed to k *logical bitmaps*, each of l bits, denoted as $L_{f,i}$, $0 \le i < k$. The number of bit segments in each logical bitmap is $y = \frac{l}{z}$, assuming that l, m and z are chosen such as l and m are divisible by z. The jth segment in $L_{f,i}$ contains bits $L_{f,i}[j \times z + t]$, $0 \le t < z$.

A logical bitmap $L_{f,i}$ does not occupy its own memory. It is logically constructed by taking its bit segments randomly from C_i, i.e.,

$$L_{f,i}[j \times z + t] = C_i[H_{i,j}(f) \times z + t], \quad 0 \le j < y, \ 0 \le t < z, \tag{11}$$

where $L_{f,i}[j \times z + t]$ is the tth bit in the jth segment of the ith logical bitmap of flow f, and $H_{i,j}(\cdot)$ are independent hash functions. Only one hash, $H_{i,j}(f)$, is needed to locate a segment of z bits for a logical bitmap. There are $\frac{lk}{z}$ segments in all k logical bitmaps of flow f, which means $\frac{lk}{z}$ hashes are needed. For example, if we choose $z = \frac{l}{4}$, then $4k$ hashes are needed.

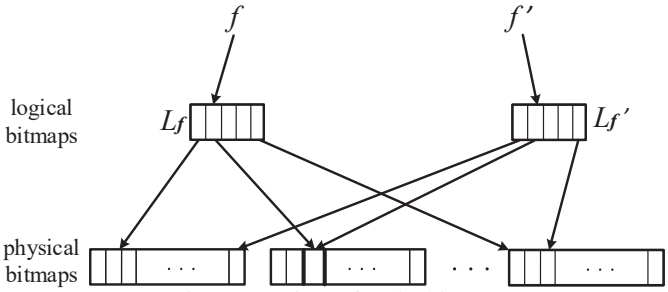


Fig. 7: Design of vSkt(bitmap)

Consider an arbitrary large flow f' other than f in Figure 6. Its logical bitmaps $L_{f',i}$ also take segments randomly from C_i. It may share some of the same segments that f takes due to hash collision. When that happens, as elements of f' are recorded onto the bits of the shared segments, it causes noise to f. However, unlike kJoin(bitmap) where an entire large flow f' becomes noise, in our new design, even when f' causes noise to f, only a fraction of its elements that are recorded in the shared segments does so, and its elements that are recorded in other segments do not cause noise to f.

• *Recording.* To record a data item $\langle f, e \rangle$, skJoin(bitmap) hashes e to a bit in each of its logical bitmaps, and sets that bit to one, i.e., $L_{f,i}[H(e)] = 1$, $0 \le i < k$. But L_f is logically constructed and does not actually exist. By (11), $L_{f,i}[H(e)]$ is actually $C_i[H_{i,\lfloor H(e)/z \rfloor}(f)][H(e) \mod z]$. Hence, what's actually performed is

$$C_i[H_{i,\lfloor H(e)/z \rfloor}(f)][H(e) \mod z] = 1, \tag{12}$$

which takes two hashes and sets one bit.

• *Query.* We combine the k logical bitmaps, $L_{f,i}$, $0 \le i < k$, into one, denoted as L_f, using the bitwise AND operation as follows: for $0 \le j < \frac{l}{z}, 0 \le t < z$,

$$\begin{aligned} L_f[j \times z + t] =& L_{f,0}[j \times z + t] \wedge ... \wedge L_{f,k-1}[j \times z + t] \\ =& C_0[H_{0,j}(f)][t] \wedge ... \wedge C_{k-1}[H_{k-1,j}(f)][t]. \end{aligned} \tag{13}$$

Performing the above computation requires $\frac{lk}{z}$ hashes, in order to locate the bit segments of all k bitmaps of flow f.

L_f is the combined logical bitmap after noise removal by bit-wise AND. According to [28], we can estimate the number of distinct elements still recorded in the bitmap as follows:

$$\hat{c}_f = -l \ln(1 - \frac{u_f}{l}), \tag{14}$$

where

$$u_f = \sum_{0 \le j < l} L_f[j].$$

C. Design Difference between skJoin(bitmap) and vSk(bitmap)

The design of skJoin(bitmap) in Fig. 6 is different from that of vSketch [31] in Fig. 7, where vSkt(bitmap) is a version of vSketch that uses bitmaps. The design difference has significant performance impact, particularly on the overhead of online queries. vSketch has one logical bitmap L_f of l bits per flow f, where l is in hundreds or even thousands. Each bit in L_f is mapped by hashing to a bit in one of the l physical bit arrays. Therefore, for a spread query on flow f, vSketch needs

l hashes functions to locate all l bits of f from the physical arrays, whereas we know that skJoin requires $\frac{lk}{z}$ hashes — for example, if $k = 4$ and $z = \frac{l}{4}$, that will be 16 hashes. Moreover, vSketch has a single logical bitmap for flow f. It cannot use the min operation to reduce noise. Instead, it scans the whole physical arrays to compute an average noise to be subtracted from, which is significant overhead for large arrays.

D. Additional skJoin Sketches

We give two more sketches in the super kJoin family, with logical bitmaps being replaced with other types of *logical maps*. More specifically, we replace each logical bitmap with a logical HLL-map (or FM-map), which is an array of HLL registers (or FM registers). The resulting sketch is referred to as skJoin(HLL) (or skJoin(FM)), whose details are provided below.

• *skJoin(HLL),* in which $C_i, 0 \leq i < k$ are arrays of HLL register segments[30]. A HLL register segment consists of z HLL registers, each of 5 bits long. Flow f is hashed to k logical HLL-maps $L_{f,i}$, $0 \leq i < k$. Each logical HLL-map $L_{f,i}$, $0 \leq i < k$, consists of $l = y \times z$ registers. The mapping from the registers in $L_{f,i}$ to the registers in register segments from C_i is still (11). For the recording operation, we change (12) to

$$\begin{aligned} &C_i[H_{i,\lfloor H(e)/z \rfloor}(f)][H(e) \mod z] \\ =&\max\{C_i[H_{i,\lfloor H(e)/z \rfloor}(f)][H(e) \mod z], G(e)\}, \end{aligned}$$

where $G[e]$ is an exponentially distributed hash function whose value is v with probability $2^{-(v+1)}$ for $v \geq 0$. For the query operation, we change (13) and (14) to, $0 \leq j < y, 0 \leq t < z$,

$$\begin{aligned} L_f[j \times z + t] &= \min\{L_{f,i}[j \times z + t], \quad 0 \leq i < k\} \\ &= \min\{C_i[H_{i,j}(f)][t], \quad 0 \leq i < k\} \end{aligned} \tag{15}$$

$$\hat{c}_f = \alpha_l l^2 / (\sum_{j=0}^{l-1} 2^{-L_f[j]}), \tag{16}$$

where α_l is a pre-computed constant determined by l [30].

• *skJoin(FM),* in which $C_i, 0 \leq i < k$ are arrays of FM register segments [29]. A FM register segment consists of z FM registers, each of 32 bits long. Each logical FM-map $L_{f,i}$, $0 \leq i < k$, consists of $l = y \times z$ registers. For the recording operation, we change (12) to

$$C_i[H_{i,\lfloor H(e)/z \rfloor}(f)][H(e) \mod z][G(e)] = 1.$$

For the query operation, we change (13) to, $\forall 0 \leq j < y, 0 \leq t < z, 0 \leq q < 32$,

$$\begin{aligned} L_f[j \times z + t][q] =& L_{f,0}[j \times z + t][q] \wedge ... \wedge L_{f,k-1}[j \times z + t][q] \\ =& C_0[H_{0,j}(f)][t][q] \wedge ... \wedge C_{k-1}[H_{k-1,j}(f)][t][q]. \end{aligned} \tag{17}$$

We change (14) to

$$\hat{c}_f = l 2^{\sum_{0 \leq j < l} \rho(L_f[j])/l} / \phi, \tag{18}$$

where ϕ is a pre-computed constant [29]. Recall that $\rho(\cdot)$ is a function that counts the number of consecutive ones starting from the lowest order bit in the input.

As we can see, by changing y, which is the number of register segments we used to construct a logical map, a series of different skJoin sketches are produced. When $y = 1$, they are identical to kJoin sketches, which means kJoin sketches are actually special cases of skJoin sketches. When we increase y, the length of a register segment decreases. Since the noise coming from a large flow will be distributed into y different segments, the noise distribution will be more smooth in the array, which helps reduce the worst case error. When $y = l$, we are expected to have a best estimation accuracy. However, a large y will reduce the querying throughout since we need to calculate $k \times y$ hash functions to location $k \times y$ register segments of one flow. Therefore, choose a suitable y is a trade between accuracy and querying throughput.

E. Error Bound

Let c^* be the total number of distinct elements from all flows. Thus, $c^* - c_f$ is the number of distinct elements from flows other than f. They are called *noise elements* with respect to f. Next, we first give two assumptions and then present the following theorem to show the error bound of flow spread estimates produced by skJoin sketches. We make two assumptions in skJoin: (1) For any flow f with a spread of c_f, we have $c^* \gg c_f$ where c^* is sum of spread of all flows; (2) the noise elements are recorded in $C_i, 0 \leq i < k$ uniformly at random. These two assumptions should always be true for large data streams even with Zipf distribution [43] such as network packet streams. For example, Table I shows the flow spread distribution of a 1 minute packet streams from CAIDA [44] where the flow ID is the destination address and element is the source address. The largest spread of a flow is less than 20000, which is much smaller than the sum of spreads of all flows (around 430K). Therefore, our first assumption stands. For the second assumption, it is not always true that noise elements are distributed among all segments uniformly at random. A simple example is that when there are only two flows, the noise will not be distributed among all segments. However, this assumption is approximately true when there are a large number of flows and the spread of any flow is negligible when comparing with the total size/spread of all flows, which is generally true for large network traffic where sketches are needed.

Theorem 1: Consider an arbitrary flow f. Let $\hat{c}_f$ be the final estimate from skJoin(T_{L_f}), where T_{L_f} is the type of logical maps (plug-ins). As long as the logical map's estimate for a single flow f without noise satisfies

$$\begin{aligned} &c_f(1 - \alpha_1) \leq E(\hat{c}_f) \leq c_f(1 + \alpha_1) \\ &c_f^2(\frac{\beta}{\sqrt{l}} - \alpha_2)^2 \leq Var(\hat{c}_f) \leq X_f^2(\frac{\beta}{\sqrt{l}} + \alpha_2)^2, \end{aligned} \tag{19}$$

Flow Spread Range	≥ 1	≥ 128	≥ 256	≥ 512	≥ 1024	≥ 2048	≥ 4096	≥ 8192
Flow number	112712	256	119	49	25	15	10	2

TABLE I: Flow Spread Distribution of 1 Minute Internet Traffic

where $\alpha_1, \alpha_2, \beta$ are constants, the probability for the relative error of $\hat{c}_f$ to be smaller than a given bound ϵ is

$$\text{Prob}(|\frac{\hat{c}_f}{c_f} - 1| \leq \epsilon) \geq 1 - \delta, \tag{20}$$

where

$$\delta = \frac{4(\frac{\beta}{\sqrt{l}} + \alpha_2)^2(1+\frac{\epsilon}{2})^2}{(\epsilon - 2\alpha_1 - \alpha_1\epsilon)^2} + \frac{(\frac{\beta}{\sqrt{l}} + \alpha_2)^2(1+\frac{\epsilon}{2})^2}{(\epsilon - \alpha_1)^2} + (\frac{2yc_*}{\epsilon c_f m})^k (2 - \frac{4(\frac{\beta}{\sqrt{l}} + \alpha_2)^2(1+\frac{\epsilon}{2})^2}{(\epsilon - 2\alpha_1 - \alpha_1\epsilon)^2} - \frac{(\frac{\beta}{\sqrt{l}} + \alpha_2)^2(1+\frac{\epsilon}{2})^2}{(\epsilon - \alpha_1)^2}), \tag{21}$$

Proof 4.1: Consider an arbitrary flow f. Let $x_{i,f}$ be the number of distinct elements from all flows that are recorded by one of flow f's logical maps, $L_{f,i}$, $0 \leq i < k$. We have

$$x_{i,f} = c_f + n, \tag{22}$$

where n is the number of noise elements recorded in $L_{f,i}$ from flows other than f, and it follows a binomial distribution $Bin(c_* - c_f, \frac{l}{mz})$, which can be closely approximated as $Bin(c_*, \frac{y}{m})$, because $c_* \gg c_f$. We have $\text{E}(n) = \frac{yc_*}{m}$. By Markov's inequality,

$$\text{Prob}(n \geq \frac{\epsilon c_f}{2}) \leq \frac{\text{E}(n)}{\frac{\epsilon}{2}} \tag{23}$$

After substituting (22) in (23), we have

$$\text{Prob}(x_{i,f} \geq c_f(1 + \frac{\epsilon}{2})) \leq \frac{2yc_*}{\epsilon c_f m}. \tag{24}$$

Let X_f be the number of distinct elements recorded in the combined logical map L_f. Based on the operations that join the logical maps in skJoin(bitmap), skJoin(HLL) and skJoin(FM), which keep all elements from f as well as noise elements common in all k logical maps, we have

$$c_f \leq X_f \leq \min\{x_{i,f} | 0 \leq i < k\} \tag{25}$$

Applying (25) to (24), we have

$$\text{Prob}(X_f \leq c_f(1 + \frac{\epsilon}{2})) \geq 1 - \left(\frac{2yc_*}{\epsilon c_f m}\right)^k. \tag{26}$$

Recall that B is an array of plug-ins, which forms a logical map. According to 19, the expectation and variance of $\hat{c}_f$ computed from the combined logical map can be given as follows:

$$\begin{aligned} &X_f(1-\alpha_1) \leq E(\hat{c}_f) \leq X_f(1+\alpha_1) \\ &X_f^2(\frac{\beta}{\sqrt{l}} - \alpha_2)^2 \leq Var(\hat{c}_f) \leq X_f^2(\frac{\beta}{\sqrt{l}} + \alpha_2)^2. \end{aligned} \tag{27}$$

According to Chebyshev's inequality,

$$\begin{aligned} &\text{Prob}(|\hat{c}_f - \text{E}(\hat{c}_f)| \geq (\epsilon - \alpha_1)c_f) \leq \frac{Var(\hat{c}_f)}{((\epsilon - \alpha_1)c_f)^2}, \\ &\text{Prob}(|\hat{c}_f - \text{E}(\hat{c}_f)| \geq \frac{\epsilon - 2\alpha_1 - \alpha_1\epsilon}{2} c_f) \\ &\leq \frac{4Var(\hat{c}_f)}{((\epsilon - 2\alpha_1 - \alpha_1\epsilon)c_f)^2} \end{aligned} \tag{28}$$

Define a function $g(x) = x^2(\frac{\beta}{\sqrt{l}} + \alpha_2)^2$. Applying (27) to (28), we have

$$\begin{aligned} &\text{Prob}(\hat{c}_f \geq (1-\alpha_1)X_f - (\epsilon - \alpha_1)c_f) \geq 1 - \frac{g(X_f)}{((\epsilon - \alpha_1)c_f)^2}, \\ &\text{Prob}(\hat{c}_f \leq (1+\alpha_1)X_f + \frac{\epsilon - 2\alpha_1 - \alpha_1\epsilon}{2} c_f) \\ &\geq 1 - \frac{4g(X_f)}{((\epsilon - 2\alpha_1 - \alpha_1\epsilon)c_f)^2}. \end{aligned} \tag{29}$$

From (25), we know $X_f \geq c_f$, which means

$$\begin{aligned} &\text{Prob}(\hat{c}_f \geq 1 - \epsilon c_f) \\ =&\text{Prob}(\hat{c}_f \geq (1-\alpha_1)c_f - (\epsilon - \alpha_1)c_f) \\ \geq&\text{Prob}(\hat{c}_f \geq (1-\alpha_1)X_f - (\epsilon - \alpha_1)c_f) \end{aligned} \tag{30}$$

Combining (29) and (30), we have

$$\text{Prob}(\hat{c}_f \geq c_f - \epsilon c_f) \geq 1 - \frac{g(X_f)}{((\epsilon - \alpha_1)c_f)^2}. \tag{31}$$

When $X_f \leq c_f(1 + \frac{\epsilon}{2})$, we know

$$\text{Prob}(\hat{c}_f \geq c_f(1-\epsilon) | X_f \leq c_f(1 + \frac{\epsilon}{2})) \geq 1 - \frac{g(c_f(1+\frac{\epsilon}{2}))}{((\epsilon - \alpha_1)c_f)^2}. \tag{32}$$

Applying (26) and (32) to

$$\begin{aligned} \text{Prob}(\hat{c}_f \geq c_f(1-\epsilon)) \geq&\text{Prob}(X_f \leq c_f(1 + \frac{\epsilon}{2})) \\ &\cdot\text{Prob}(\hat{c}_f \geq c_f(1-\epsilon) | X_f \leq c_f(1 + \frac{\epsilon}{2})), \end{aligned} \tag{33}$$

we have

$$\begin{aligned} &\text{Prob}(\hat{c}_f \geq c_f(1-\epsilon)) \\ &\geq \left[1 - \left(\frac{2yc_*}{\epsilon c_f m}\right)^k\right] \cdot \left[1 - \frac{g(c_f(1+\frac{\epsilon}{2}))}{((\epsilon - \alpha_1)c_f)^2}\right], \\ &\text{Prob}(\frac{\hat{c}_f}{c_f} \geq 1 - \epsilon) \\ &\geq \left[1 - \left(\frac{2yc_*}{\epsilon c_f m}\right)^k\right] \cdot \left[1 - \frac{(\frac{\beta}{\sqrt{l}} + \alpha_2)^2(1+\frac{\epsilon}{2})^2}{(\epsilon - \alpha_1)^2}\right]. \end{aligned} \tag{34}$$

Similarly, when $X_f \leq c_f(1 + \frac{\epsilon}{2})$, from (29) we know

$$\begin{aligned} &\text{Prob}(\hat{c}_f \leq c_f + \epsilon | X_f \leq c_f + \frac{\epsilon}{2}) \\ =&\text{Prob}(\hat{c}_f \leq (1+\alpha_1)c_f(1+\frac{\epsilon}{2}) + \frac{\epsilon - 2\alpha_1 - \alpha_1\epsilon}{2} c_f | X_f \leq c_f + \frac{\epsilon}{2}) \\ \geq&\text{Prob}(\hat{c}_f \leq X_f + \frac{\epsilon}{2} | X_f \leq c_f + \frac{\epsilon}{2}) \\ \geq&1 - \frac{4g(X_f)}{((\epsilon - 2\alpha_1 - \alpha_1\epsilon)c_f)^2} \geq 1 - \frac{4g(c_f(1+\frac{\epsilon}{2}))}{((\epsilon - 2\alpha_1 - \alpha_1\epsilon)c_f)^2}. \end{aligned} \tag{35}$$

Then applying (26) and (35) to

$$\text{Prob}(\hat{c}_f \geq c_f(1+\epsilon)) \geq \text{Prob}(X_f \leq c_f(1+\frac{\epsilon}{2})) \cdot \text{Prob}(\hat{c}_f \geq c_f(1+\epsilon) | X_f \leq c_f(1+\frac{\epsilon}{2})), \tag{36}$$

we have

$$\begin{aligned} &\text{Prob}(\hat{c}_f \geq c_f(1+\epsilon)) \\ &\geq \left[1 - \left(\frac{2yc_*}{\epsilon c_f m}\right)^k\right] \cdot \left[1 - \frac{4g(c_f(1+\frac{\epsilon}{2}))}{((\epsilon - 2\alpha_1 - \alpha_1\epsilon)c_f)^2}\right] \\ &\text{Prob}(\frac{\hat{c}_f}{c_f} \geq 1+\epsilon) \\ &\geq \left[1 - \left(\frac{2yc_*}{\epsilon c_f m}\right)^k\right] \cdot \left[1 - \frac{4(\frac{\beta}{\sqrt{l}} + \alpha_2)^2(1+\frac{\epsilon}{2})^2}{(\epsilon - 2\alpha_1 - \alpha_1\epsilon)^2}\right]. \end{aligned} \tag{37}$$

Finally, by applying (34) and (37) to

$$\begin{aligned} \text{Prob}(|\frac{\hat{c}_f}{c_f} - 1| > \epsilon) =& (1 - \text{Prob}(\frac{\hat{c}_f}{c_f} \geq 1-\epsilon)) \\ &+ (1 - \text{Prob}(\frac{\hat{c}_f}{c_f} \leq 1+\epsilon)), \end{aligned} \tag{38}$$

we conclude

$$\begin{aligned} &\text{Prob}(|\frac{\hat{c}_f}{c_f} - 1| > \epsilon) \\ \leq& (\frac{2yc_*}{\epsilon c_f m})^k (2 - \frac{4(\frac{\beta}{\sqrt{l}} + \alpha_2)^2(1+\frac{\epsilon}{2})^2}{(\epsilon - 2\alpha_1 - \alpha_1\epsilon)^2} - \frac{(\frac{\beta}{\sqrt{l}} + \alpha_2)^2(1+\frac{\epsilon}{2})^2}{(\epsilon - \alpha_1)^2}) \\ &+ \frac{4(\frac{\beta}{\sqrt{l}} + \alpha_2)^2(1+\frac{\epsilon}{2})^2}{(\epsilon - 2\alpha_1 - \alpha_1\epsilon)^2} + \frac{(\frac{\beta}{\sqrt{l}} + \alpha_2)^2(1+\frac{\epsilon}{2})^2}{(\epsilon - \alpha_1)^2}. \end{aligned} \tag{39}$$

If we denote

$$\begin{aligned} \delta =& \frac{4(\frac{\beta}{\sqrt{l}} + \alpha_2)^2(1+\frac{\epsilon}{2})^2}{(\epsilon - 2\alpha_1 - \alpha_1\epsilon)^2} + \frac{(\frac{\beta}{\sqrt{l}} + \alpha_2)^2(1+\frac{\epsilon}{2})^2}{(\epsilon - \alpha_1)^2} + \\ &(\frac{2yc_*}{\epsilon c_f m})^k (2 - \frac{4(\frac{\beta}{\sqrt{l}} + \alpha_2)^2(1+\frac{\epsilon}{2})^2}{(\epsilon - 2\alpha_1 - \alpha_1\epsilon)^2} - \frac{(\frac{\beta}{\sqrt{l}} + \alpha_2)^2(1+\frac{\epsilon}{2})^2}{(\epsilon - \alpha_1)^2}), \end{aligned} \tag{40}$$

we have $\text{Prob}(|\frac{\hat{c}_f}{c_f} - 1| \leq \epsilon) \geq 1 - \delta$.

Therefore, the theorem holds.

We argue that the prerequisite for Theorem 1 in (19) for logical maps is usually satisfied when using primarily maps, i.e., bitmaps, FM-maps and HLL-maps, as plug-ins. In particular, we have the follow lemma for validation and the values of $\alpha_1, \alpha_2, \beta$ for different types of plug-ins.

Lemma 1: The estimate $\hat{c}_f$ from bitmap, FM-map, HLL-map for a single flow with spread c_f satisfies:

$$\begin{aligned} &c_f(1-\alpha_1) \leq E(\hat{c}_f) \leq c_f(1+\alpha_1) \\ &c_f^2(\frac{\beta}{\sqrt{l}} - \alpha_2)^2 \leq Var(\hat{c}_f) \leq X_f^2(\frac{\beta}{\sqrt{l}} + \alpha_2)^2, \end{aligned} \tag{41}$$

where $\alpha_1, \alpha_2, \beta$ are constants. For bitmap $\beta = \frac{1}{\sqrt{2}}$, $\alpha_1 \leq \frac{X_f}{4l^2}, \alpha_2 \leq \frac{\sqrt{X_f}}{\sqrt{6}l}$; for FM, $\beta = 0.78$, $\alpha_1, \alpha_2 \leq 1 \times 10^{-5}$; for HLL, $\beta = 1.04$, $\alpha_1 \leq 5 \times 10^{-5}, \alpha_2 \leq 5 \times 10^{-4}$, when $l \geq 16$.

The proof for bitmap, FM-map and HLL-map can be found in [38], [29], [30], respectively.

V. Multi-stream Measurement

In multi-stream measurement, we consider a number d of geographically dispersed data streams, D_t, $0 \leq t < q$. The elements of a flow f may appear in multiple streams. The problem is to measure flow spread $||f||$ across all data streams.

Consider an example with $d = 2$, where D_0 contains $\langle f_1, e_1\rangle$, $\langle f_1, e_2\rangle$, and $\langle f_2, e_3\rangle$, while D_1 contains $\langle f_1, e_1\rangle$, $\langle f_2, e_4\rangle$, and $\langle f_1, e_1\rangle$. We have two flows, f_1 with elements $\{e_1, e_2, e_1, e_1\}$, and f_2 with elements $\{e_3, e_4\}$, where $|f_1| = 4$, $||f_1|| = 2$, and $|f_2| = ||f_2|| = 2$.

In a possible application, a search engine has multiple datacenters at different places. Each datacenter receives a stream of search requests. We model all requests on the same keyword (flow ID) as a flow. User IPs are elements. Multiple datacenters may receive search requests on the same keyword, which means the elements of a flow may appear in different streams. Multi-stream measurement allows us to know the aggregate popularity on keywords, with flow spread being the number of different users that have searched a keyword.

Below we show that the skJoin sketches can support multi-stream measurement. After each measurement period, the device that measures D_t, $0 \leq t < d$, will have recorded all data items of the period onto its array of plug-ins, now denoted as $C_{t,i}, 0 \leq i < k$. It sends $C_{t,i}, 0 \leq i < k$ to a central server and resets them for the next measurement period. After the server receives $C_{t,i}, 0 \leq i < k$, $0 \leq t < q$, for all data streams, it will combine them into a single array C^*, which is dependent on the type of plug-ins:

- bitmap: $C_i^*[j][p] = C_{0,i}[j][p] \vee ... \vee C_{d-1,i}[j][p]$, $0 \leq i < k, 0 \leq j < m, 0 \leq p < z$.
- FM: $C_i^*[i][j][p][w] = C_{0,i}[j][p][w] \vee ... \vee C_{q-1,i}[j][p][w]$, $0 \leq i < k, 0 \leq j < m, 0 \leq p < z$, $0 \leq w < 32$.
- HLL: $C_i^*[j][p] = \max\{C_{t,i}[j][p], 0 \leq t < q\}$, $0 \leq i < k, 0 \leq j < m, 0 \leq p < z$.

The query on the combined logical map B^* is performed in the same way as described in Section IV.

VI. Evaluation

We evaluate the performance of kJoin and skJoin in comparison with the existing works.

A. Experimental Setting

We implement kJoin and skJoin with three types of plug-ins, bitmaps [38], FM-maps [29] and HLL-maps [30], [45], which together cover six new sketches, kJoin(bitmap), kJoin(FM), kJoin(HLL), skJoin(bitmap), skJoin(FM), and skJoin(HLL). We implement the most related work, bSketch and cSketch [37], [39], [31], including six sketches of cSkt(bitmap), cSkt(FM), cSkt(HLL), bSkt(bitmap), bSkt(FM) and bSkt(HLL), as was

explained in Section II-C. In addition, we implement the virtual sketches of CSE [32], vHLL [33], vSketch [31], which has three versions, vSkt(bitmap), vSkt(FM) and vSkt(HLL) for different plug-ins, and rSkt2 [41], which also has three versions, rSkt2(bitmap), rSkt2(FM) and rSkt2(HLL).

We know from Section IV-B that the number of hashes per query under skJoin is proportional to $\frac{l}{z}$, which is the number of segments in each virtual bitmap (FM-map or HLL-map). The value of $\frac{l}{z}$ is tunable by choosing different values of z. In our experiments, we set $z = \frac{l}{2}$, $\frac{l}{4}$ and $\frac{l}{8}$, for $\frac{l}{z} = 2$, 4 and 8, respectively, to study its impact on processing overhead, with the corresponding sketches denoted as skJoin-2, skJoin-4 and skJoin-8. By default, without explicitly mentioning the value of $\frac{l}{z}$, skJoin means skJoin-8 when we present the results.

The data streams that we use in the experiments are real-world network traffic traces that we have downloaded from CAIDA [44]. Each trace contains a packet stream recorded by an Internet router for 1 minute long. We use 10 such traces for each experiment and take the average results. We define a flow as all packets going to the same destination address (which serves as flow ID f). The elements e in our experiments are the source addresses in the packet headers. Flow spread is the number of distinct elements in a flow, i.e., the number of different sources communicating with a destination. Monitoring such information in each traffic trace helps detect botnet-based DDoS attacks where overwhelming packets are sent from a large number of bots (flow spread) to the same destination.

We set $k = 2$ for all sketches. In order to have sufficient memory to measure the maximum spread in our traces, we set $l = 5000$ for the size of a bitmap or a logical bitmap, and set $l = 128$ for the size of an FM-map, an HLL-map, a logical FM-map or a logical HLL-map as their estimation ranges are far larger than a bitmap under the same value of l.

Our evaluation is based on three metrics: online query overhead, estimation accuracy and throughput. For query overhead, we measure the average time it takes to process each query on flow spread. For estimation accuracy, we compute the average absolute error (defined as $|\hat{c}_f - ||f|||$) of all flows over the 10 traces, the average relative error (defined as $\frac{|\hat{c}_f - ||f|||}{||f||}$) of all flows over the 10 traces, and the worse-case error among all flows in each trace. For throughput, we measure the average number of data items recorded per second.

We implement the sketches in Java and perform trace-driven experiments on a computer with an Intel(R) Xeon CPU @ 3.7GHz and 32GB of RAM. We also evaluate using hardware acceleration for higher throughput shortly.

B. Online Query Overhead

We use experiments to validate our design choice of using the CountMin structure, instead of following the path of virtual sketches, which incur far higher online query overhead as claimed in Section II-D. Under the described experimental setting, we perform queries on the spreads of 1000 randomly selected flows from each of the 10 traces during the recording of the traces by each sketch under comparison. The average processing time per query under each sketch is presented in Table II, where the first column shows the sketch type and the first row shows the three plug-ins. The results confirm that the virtual sketches of CSE [32], vHLL [33], and vSketch [31] cannot support online querying as their per-query processing overhead is between one to three orders of magnitude higher than the other sketches. This is significant because query overhead competes for processing cycles that would otherwise be used to record the data stream. High query overhead means that queries can only be performed sparsely, which is undesired for real-time applications that rely on frequent queries to catch events (such as worm attacks) at they happen.

Sketches \ Plug-ins	bitmap	FM-map	HLL-map
vSkt [31]	1.644	0.376	9.260
CSE [32]	1.745	-	-
vHLL [33]	-	-	7.299
cSkt,bSkt [37], [39], [31]	0.042	0.001	0.015
kJoin	0.051	0.001	0.007
skJoin-2	0.052	0.002	0.007
skJoin-4	0.054	0.002	0.007
skJoin-8	0.055	0.002	0.008
rSkt2	0.057	0.002	0.008

TABLE II: Query processing time (ms) of vSkt, CSE, vHLL, cSkt, bSkt, kJoin, and skJoin using different plug-ins, bitmaps, FM-maps, and HLL-maps. In comparison, the virtual sketches of vSkt, CSE and vHLL are not suitable for online spread queries due to high overhead.

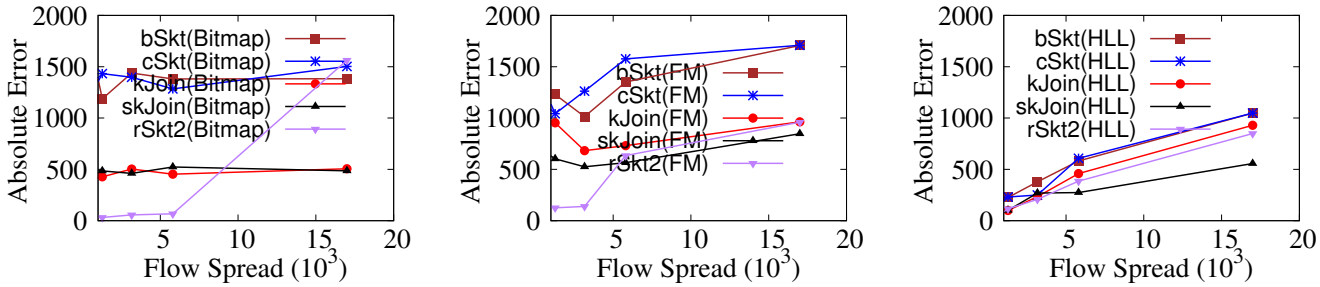
The query processing overhead is comparable among kJoin, skJoin and some other related work, cSkt, bSkt [37], [39], [31] and rSkt2 [41], with bitmaps and FM-maps as plug-ins. The overhead of kJoin/skJoin is half or less than that of cSkt and bSkt with HLL-maps as plug-ins. In the rest of the evaluation, we will focus on comparing the sketches that are suitable for online queries, i.e., kJoin, skJoin, cSkt and bSkt.

C. Estimation Accuracy

Our second set of experiments compares the proposed kJoin/skJoin with cSkt, bSkt and rSkt2 on spread estimation accuracy, when using different plug-ins of bitmaps, FM-maps and HLL-maps, with each sketch being allocated 2 Mb memory. Each sketch produces identical results with the same accuracy under different software/GPU/FPGA platforms; their impact is on throughput, which will be discussed shortly.

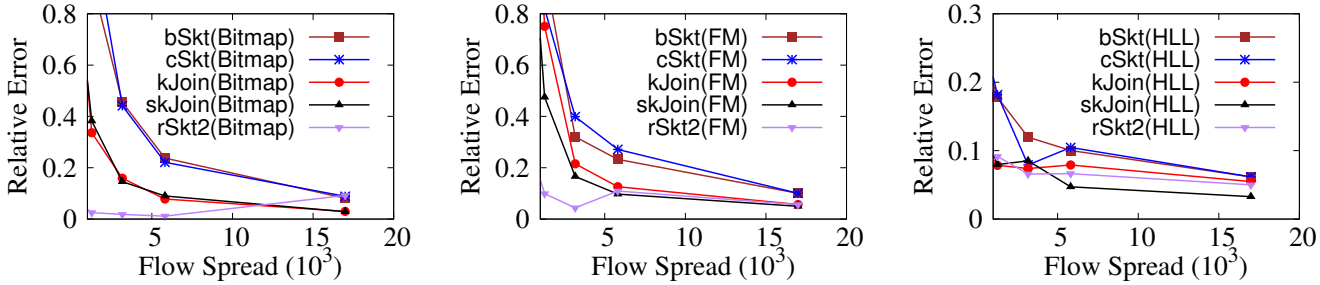
The average absolute errors of the sketches are presented in Fig. 8, where we put all flows into a number of bins based on their actual flow spreads, average the absolute errors of the flows' spread estimates, and plot a point for each bin, with the y-coordinate being the average error and the x-coordinate being the average spread of all flows in the bin.

The figure shows that skJoin(bitmap), skJoin(FM) and skJoin(HLL) are lightly to modestly better than its kJoin counterparts, thanks to the extra logical layer that reduces the impact of noise introduced by large flows. They both outperform cSkt and bSkt, in particular under bitmap and FM plug-ins, where the error reduction is up to 67%. As for the comparison with rSkt2, skJoin has much smaller error for large flows while rSkt2 has smaller error for small flows. Both algorithms have different scopes of applications. For the



(a) under bitmaps (b) under FM-maps (c) under HLL-maps

Fig. 8: skJoin(bitmap), skJoin(FM) and skJoin(HLL) are consistently more accurate in spread estimation with much smaller absolute error (up to 67% smaller) than their cSkt and bSkt counterparts, respectively. As expected, kJoin is slightly to modestly worse than skJoin, but much better than cSkt and bSkt.



(a) under bitmaps (b) under FM-maps (c) under HLL-maps

Fig. 9: skJoin(bitmap), skJoin(FM) and skJoin(HLL) have much smaller relative error (up to 67% smaller) than their cSkt and bSkt counterparts. As expected, kJoin is slightly to modestly worse than skJoin, but much better than cSkt and bSkt.

applications that care about large flows, e.g., super spreader detection, skJoin is a better choice.

Fig. 9 compares the proposed skJoin/kJoin sketches with cSkt, bSkt and rSkt2 in terms of average relative error. Similar conclusions can be drawn. skJoin outperforms kJoin, which in turn outperforms cSkt and bSkt, particularly under bitmap and FM plug-ins. skJoin performs better than rSkt2 for large flows, particularly under HLL plug-in. Another observation is that the relative errors are smaller for large-spread flows. That is because relative error is computed by dividing absolute error with $||f||$, which varies in different spread bins.

Fig. 10 evaluates the worst-case errors of skJoin (for different y), kJoin, cSkt, bSkt and rSkt2. We conduct experiments on all 10 traces, and plot the average worst-case errors and the error bars in the figure. The error bars show the range of the worst-case error in 10 traces. We can see that while kJoin has smaller worse-case error than the existing works, skJoin has far smaller worse-case error than kJoin. Recall that reducing large error was the motivation for us to introduce skJoin on top of kJoin. Consider the performance of skJoin under different values of y. The worst-case error decreases when y increases from 1 (where skJoin becomes kJoin) to l (where each bit, FM register or HLL register is a segment). The experimental results validate our new design of introducing an extra logical layer to control the impact of noise from large flows. When compred to rSkt2, skJoin-4 has similar worst case error as rSkt2 and skJoin-8 has smaller worst case error than rSkt2.

D. Recording Throughput

Our third set of experiments evaluate online recording throughputs of skJoin, kJoin, cSkt, bSkt and rSkt2 on CPU implementations, under different plug-ins. Fig. 11 compares the throughputs of skJoin with those of kJoin, cSkt, bSkt and rSkt2. The extra hash operation to implement the logical layer makes the throughput of skJoin smaller than kJoin, cSkt and bSkt by 8% under bitmap, by 7% under FM and just 4% under HLL. rSkt2 has modestly higher recording throughput than kJoin and skJoin in the CPU implementation. In the hardware (i.e., FPGA) implementation, they will have the have the same recording throughput, which we will show shortly.

E. GPU and FPGA Implementations

In addition to the CPU implementation described above, we also implement the sketches on GPU and FPGA.

GPU Implementation: We use the CUDA 10 toolkit [46] to program an NVIDIA GTX 1070 GPU, with 8GB GDDR5 memory @ 1506MHz. The GPU has 1920 cuda cores. Parallel processing is utilized for speed-up.

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.

We stress that hardware accelerations will not affect the estimation accuracy of the sketches as long as the same hash functions, parameter settings and datasets are used. But they affect the recording throughput, which will be presented below.

Fig. 12(a) compares the throughputs of skJoin, kJoin, cSkt, bSkt and rSkt2 on GPU implementation which supports a massive scale of parallel processing by numerous cores, allowing over a thousand of elements to be processed simultaneously. rSkt2 has a higher recording throughput since it only needs to update one register for an item.

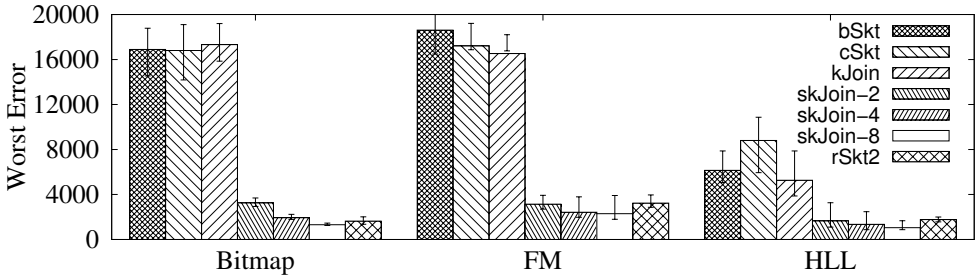


Fig. 10: The worst-case error of skJoin under different values of y (denoted as skJoin-y) in comparison with kJoin, cSkt and bSkt when using bitmaps, FM-maps and HLL-maps, respectively. The histogram shows the average results from 10 traces, and the error bar shows the range of worse-case error. skJoin has smaller worst-case error (up to 96%, 95% and 95% smaller) than their kJoin, cSkt and bSkt counterparts. With y increasing, the worst error decreases.

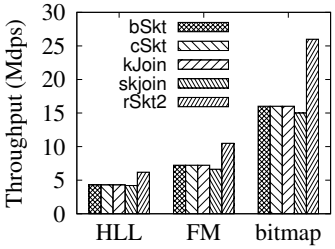
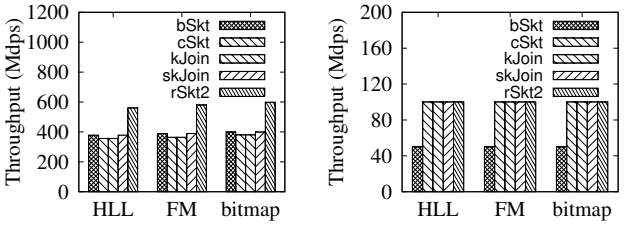


Fig. 11: Comparison of online element-recording throughput between skJoin, kJoin, cSkt and bSkt on the CPU implementations. skJoin achieves modestly worse throughput than others due to its additional computation overhead for an extra logical layer.



(a) GPU Implementation (b) FPGA Implementation

Fig. 12: Comparison of online element-recording throughput between skJoin, kJoin, cSkt and bSkt on GPU and FPGA implementations. skJoin achieves the highest throughput for GPU because its logical layer actually allows more parallel processing among numerous cores. Benefiting from pipelining, on FPGA implementation, skJoin, kJoin and cSkt records each item in one clock cycle, four times faster than bSkt.

Fig. 12(b) compares the throughputs of skJoin, kJoin, cSkt and bSkt on FPGA implementation. Note that, on FPGA implementation for skJoin and kJoin, we split the memory block into k independent ones, each experiencing the recording of any data item once. The throughputs of skJoin, kJoin, cSkt and rSkt2 are identical for all the plug-ins, k ($k = 2$) times that of bSkt. This is because for the recording of any item, we pipeline k memory accesses for skJoin, kJoin and cSkt, consuming one clock cycle, while bSkt records each item k times in the same memory block, consuming k clock cycles. We stress that the throughputs of all benchmarks will be higher if a high-end FPGA possessing a high clock rate is used.

When compared to rSkt2 [41], skJoin has better accuracy for large flows and smaller worse-case error while rSkt2 has better accuracy for small flows and modestly higher recording throughput on software platform. kJoin/skJoin has a comparable querying throughput than rSkt2 and they have the same recording throughput on hardware platform. The results shows that rSkt2 and the sketches proposed in this paper provide different performance tradeoffs and have their respectively target application areas with specific performance requirements. In conclusion, if an application focuses on measuring large flows (which is common) and wants to control the worst-case error, it should use skJoin. If the focus is on recording throughput, it should use rSkt2.

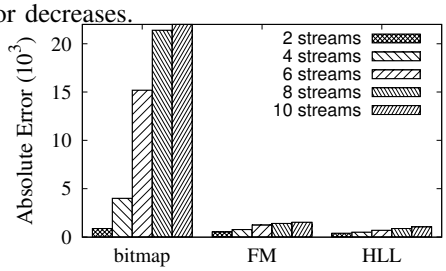


Fig. 13: Multi-stream estimation accuracy of skJoin. Using HLL-maps reduces the average absolute error by up to 95.2% for flow spread measurement, comparing with bitmaps.

F. Multi-Stream Measurement

Our last set of experiments evaluates the proposed multi-stream measurement for skJoin, with 2, 4, 6, 8 and 10 data streams, respectively. The memory allocated to each measurement location is 2 Mb. Fig. 13 shows flow-spread estimation accuracy of skJoin(bitmap), skJoin(FM) and skJoin(HLL). With more data streams, the combined error will be bigger as expected. In all cases, skJoin(HLL) provides most accurate multi-stream measurement results.

VII. Conclusion

This paper investigates the problem of flow spread estimation under a generalized streaming model. The challenge is to provide spread estimates for numerous flows with good accuracy, low overhead, and high online recording throughput. We question two fundamental design choices in the existing sketches. By replacing them with new ones for better performance, we propose two families of sketches, called kJoin and super kJoin. We mathematically derive the error bounds for their spread estimates. We implement the new sketches on both software, GPU and FPGA platforms and evaluate them using real-world data traces, in comparison with the existing works. The experimental results demonstrate the superior performance of the proposed sketches.

REFERENCES

[1] Amazon, "What is Streaming Data?," Online.

[2] Amazon, "Amazon Kinesis Streams," Online.

[3] L. Querzoni and N. Rivetti, "Models and Issues in Data Stream Systems," *in Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, June 2017.

[4] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Data Streaming and its Application to Stream Processing: Tutorial," *in Proceedings of the 21st ACM ACM SIGACT-SIGMOD-SIGART*, June 2002.

[5] Y. Lu, A. Montanari, B. Prabhakar, S. Dharmapurikar, and A. Kabbani, "Counter Braids: A Novel Counter Architecture for Per-Flow Measurement," *Proc. of ACM SIGMETRICS*, June 2008.

[6] Y. Li, R. Miao, C. Kim, and M. Yu, "FlowRadar: A Better NetFlow for Data Centers," *in Proc. of USENIX NSDI*, 2016.

[7] M. Chen, S. Chen, and Z. Cai, "Counter Tree: A Scalable Counter Architecture for Per-flow Traffic Measurement," *IEEE/ACM Transactions on Networking*, vol. 25, no. 2, pp. 1249–1262, 2016.

[8] T. Yang, H. Zhou, Y.and Jin, S. Chen, and X. Li, "Pyramid Sketch: A Sketch Framework for Frequency Estimation of Data Streams," *Proceedings of the VLDB Endowment*, vol. 10, no. 11, pp. 1442–1453, 2017.

[9] Y. Zhou, Y. Zhou, S. Chen, and Y. Zhang, "Highly Compact Virtual Active Counters for Per-flow Traffic Measurement," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pp. 1–9, IEEE, 2018.

[10] Y. Zhang, S. Singh, S. Sen, N. Duffield, and C. Lund, "Online Identification of Hierarchical Heavy Hitters: Algorithms, Evaluation, and Application," *Proc. of ACM SIGCOMM IMC*, October 2004.

[11] X. Dimitropoulos, P. Hurley, and A. Kind, "Probabilistic Lossy Counting: An Efficient Algorithm for Finding Heavy Hitters," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 1, pp. 7–16, 2008.

[12] R. B. Basat, G. Einziger, R. Friedman, M. C. Luizelli, and E. Waisbard, "Constant Time Updates in Hierarchical Heavy Hitters," *Proc. of ACM SIGCOMM*, 2017.

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

[14] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, "Elastic Sketch: Adaptive and Fast Network-wide Measurements," *Proc. of ACM SIGCOMM*, August 2018.

[15] J. Gong, T. Yang, H. Zhang, H. Li, S. Uhlig, S. Chen, L. Uden, and X. Li, "HeavyKeeper: An Accurate Algorithm for Finding Top-k Elephant Flows," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, (Boston, MA), pp. 909–921, USENIX Association, July 2018.

[16] Z. Liu, R. Ben-Basat, G. Einziger, Y. Kassner, V. Braverman, R. Friedman, and V. Sekar, "Nitrosketch: Robust and general sketch-based monitoring in software switches," in *Proceedings of the ACM Special Interest Group on Data Communication*, pp. 334–350, 2019.

[17] R. Schweller, A. Gupta, E. Parsons, and Y. Chen, "Reversible Sketches for Efficient and Accurate Change Detection over Network Data Streams," *Proc. of IMC*, 2004.

[18] H. Dai, M. Shahzad, A. X. Liu, and Y. Zhong, "Finding Persistent Items in Data Streams," *Proc. VLDB Endow.*, vol. 10, no. 4, 2016.

[19] A. Kumar, M. Sung, J. Xu, and J. Wang, "Data Streaming Algorithms for Efficient and Accurate Estimation of Flow Size Distribution," *Proc. of ACM SIGMETRICS*, June 2004.

[20] G. Cormode and S. Muthukrishnan, "An Improved Data Stream Summary: the Count-Min Sketch and Its Applications," *Proc. of LATIN*, 2004.

[21] S. Cohen and Y. Matias, "Spectral Bloom Filters," *Proc. of ACM SIGMOD*, June 2003.

[22] M. Charikar, K. Chen, and M. Farach-Colton, "Finding Frequent Items in Data Streams," *Proc. of International Colloquium on Automata, Languages, and Programming (ICALP)*, July 2002.

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

[24] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman†, "One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon," *Proc. of ACM Sigcomm*, 2016.

[25] C. C. Zou, L. Gao, W. Gong, and D. Towsley, "Monitoring and Early Warning for Internet Worms," in *Proceedings of the 10th ACM conference on Computer and communications security*, pp. 190–199, 2003.

[26] C. C. Zou, W. Gong, D. Towsley, and L. Gao, "The Monitoring and Early Detection of Internet Worms," *IEEE/ACM Transactions on networking*, vol. 13, no. 5, pp. 961–974, 2005.

[27] S. Chen and Y. Tang, "Slowing Down Internet Worms," *Proc. of IEEE ICDCS'04*, March 2004.

[28] K. Whang, B. T. Vander-Zanden, and H. M. Taylor, "A Linear-time Probabilistic Counting Algorithm for Database Applications," *ACM Transactions on Database Systems*, vol. 15, pp. 208–229, June 1990.

[29] P. Flajolet and G. N. Martin, "Probabilistic counting algorithms for database applications," *Journal of Computer and System Sciences*, vol. 31, pp. 182–209, September 1985.

[30] P. Flajolet, E. Fusy, O. Gandouet, and F. Meunier, "Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm," *Proc. of AOFA*, pp. 127–146, 2007.

[31] Y. Zhou, Y. Zhang, C. Ma, S. Chen, and O. O. Odegbile, "Generalized Sketch Families for Network Traffic Measurement," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 3, Dec. 2019.

[32] M. Yoon, T. Li, S. Chen, and J. Peir, "Fit a Spread Estimator in Small Memory," *Proc. of IEEE INFOCOM*, April 2009.

[33] Q. Xiao, S. Chen, M. Chen, and Y. Ying, "Hyper-Compact Virtual Estimators for Big Network Data Based on Register Sharing," *in Proc. of ACM SIGMETRICS*, 2015.

[34] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, "Beyond Bloom Filters: From Approximate Membership Checks to Approximate State Machines," *Proc. of ACM SIGCOMM*, 2006.

[35] L. Tang, Q. Huang, and P. P. Lee, "SpreadSketch: Toward Invertible and Network-Wide Detection of Superspreaders," in *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, pp. 1608–1617, IEEE, 2020.

[36] M. Yu, L. Jose, and R. Miao, "Software Defined Traffic Measurement with OpenSketch," *Proc. of NSDI*, pp. 29–42, 2013.

[37] G. Cormode and S. Muthukrishnan, "Space Efficient Mining of Multigraph Streams," *Proc. of ACM PODS*, June 2005.

[38] K. Whang, B. T. Vander-Zanden, and H. M. Taylor, "A Linear-time Probabilistic Counting Algorithm for Database Applications," *ACM Transactions on Database Systems*, vol. 15, no. 2, pp. 208–229, 1990.

[39] M. Yu, L. Jose, and R. Miao, "Software Defined Traffic Measurement with OpenSketch," *Proc. of USENIX Symposium on Networked Systems Design and Implementation*, 2013.

[40] D. Ting, "Approximate Distinct Counts for Billions of Datasets," in *Proceedings of the 2019 International Conference on Management of Data*, pp. 69–86, 2019.

[41] H. Wang, C. Ma, O. O. Odegbile, S. Chen, and J.-K. Peir, "Randomized Error Removal for Online Spread Estimation in Data Streaming," *Proceedings of the VLDB Endowment*, vol. 14, no. 6, pp. 1040–1052, 2021.

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

[43] G. K. Zipf, "Human behavior and the principle of least effort.," 1949.

[44] UCSD, "CAIDA UCSD Anonymized 2015 Internet Traces on Jan. 17." http://www.caida.org/data/passive/passive_2015_dataset.xml, 2015.

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

[46] Nvidia, "Nvidia cuda c programming guide, version 10.0," Online.