Accurate and O(1)-Time Query of Per-Flow Cardinality in High-Speed Networks

Qingjun Xiao, *Member, IEEE, ACM*, Yuexiao Cai, Yunpeng Cao, and Shigang Chen, *Fellow, IEEE*

***Abstract*— On a high-speed link, there may be tens of millions of IP packets per second and millions of active flows. Maintaining the state of each flow is a fundamental task underlying many network functions, such as load balancing and network anomaly detection. There are two important kinds of per-flow states: per-flow size (e.g., the number of packets received by an arbitrary destination IP) and per-flow cardinality (e.g., the number of distinct source IP addresses that contacted each destination IP). In this paper, we focus on the latter kind of states, and define a new problem: *online query of per-flow cardinality*, in which we query any given flow's cardinality entirely on the data plane with low time complexity. For this problem, we propose three solutions named On-vHLL, Ton-vHLL and Aton-vHLL, whose time cost are $O(1)$ even for the query operation. Our proposed techniques are three folds. First, we redesign the traditional vHLL with new supplementary data structures called incremental update units (IUUs). When a certain flow's cardinality is queried, these IUUs can avoid scanning the whole data structure and reduce the time complexity to $O(1)$. Second, we apply a HLL register compression technique called TailCut to the On-vHLL sketch, which can save memory cost by 50%. Third, we add a prefilter based on min-heap, alongside the Ton-vHLL sketch. The prefilter is to give each currently sampled top-k superspreader a dedicated HyperLogLog estimator for better accuracy. It can also absorb the superspreaders' packets bypassing the sketch. We evaluate our new sketches by simulation with CAIDA traces. The results show that our On-vHLL, Ton-vHLL and Aton-vHLL sketches need about 5 memory accesses per packet. The time cost of query operation decreases by hundreds of times than the traditional vHLL that can only be queried offline. Meanwhile, the estimation error of flow spread by our Aton-vHLL is comparable to vHLL.**

***Index Terms*— Data stream, cardinality estimation, network traffic measurement.**

Manuscript received 1 August 2022; revised 2 February 2023; accepted 9 April 2023; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor M. Caesar. Date of publication 1 May 2023; date of current version 19 December 2023. This work was supported in part by the National Key Research and Development Plan under Grant 2020YFB1805204, in part by the Science Foundation of Jiangsu Province under Grant BK20201266, and in part by a Grant from the Key Laboratory of Computer Network Technology of Jiangsu Province. *(Corresponding author: Qingjun Xiao.)*

Qingjun Xiao is with the Jiangsu Provincial Key Laboratory of Computer Networking Technology, School of Cyber Science and Engineering, Southeast University, Nanjing 210096, China, and also with Purple Mountain Laboratories, Nanjing 211100, China (e-mail: csqjxiao@seu.edu.cn).

Yuexiao Cai and Yunpeng Cao are with the Jiangsu Provincial Key Laboratory of Computer Networking Technology, School of Cyber Science and Engineering, Southeast University, Nanjing 210096, China (e-mail: yuexiaocai@seu.edu.cn; csypcao@seu.edu.cn).

Shigang Chen is with the Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL 32611 USA (e-mail: sgchen@cise.ufl.edu).

Digital Object Identifier 10.1109/TNET.2023.3268980

I. INTRODUCTION

IN RECENT years, commodity switches deployed on Internet backbone or data center networks have reached unprecedented high line rate of 100Gbps and 400Gbps, when transmitting IP packets on optical fibers. Such high line rate has placed great stress on the packet processing throughput of line card. As a result, they must rely on the size-limited SRAM (Static RAM, typically tens of MBs on-chip or hundreds of MBs off-chip) to provide memory resources for network functions. An important network function is to collect statistical information about the ongoing network flows, to monitor their transmission quality and detect malicious attacks. It has already become an integral part of OpenFlow standard to record the number of received packets/bytes and the durations for each active flow in the flow table [1], [2], [3]. Researchers have also suggested measuring the per-flow cardinality to detect fan-in or fan-out traffic patterns [4], [5], for example, the number of distinct destination (or source) IP addresses that has been contacted by a source (or destination) IP. Quickly detecting the fan-in and fan-out patterns is essential for many value-added network functions, such as load balancing [6], network fault diagnosis [7], DDoS detection [8], and malware spreading detection.

Compared with counting the per-flow size, it is a more difficult problem to track the per-flow cardinality. If solved improperly, the per-flow spread tracking function may occupy precious SRAM space by tens-of-folds larger than the per-flow size estimation. We define *flow size* as the number of elements in each flow [2], [3], where elements can be packets or bytes. We define *flow spread* (or cardinality) as the number of *distinct* elements in each flow [4], [5], [9], where elements may be source/destination IP addresses, source/destination ports, or content elements in packet payload. Since tracking the cardinality of a flow needs to filter duplicated elements, practitioners often use data sketch, such as Bitmap [10] or HyperLogLog (HLL) [9]. Regretfully, allocating an exclusively owned sketch for each flow needs thousands of bytes per flow, which cannot fit into on-chip SRAM with only tens of MBs.

Researchers have designed several algorithms for tracking the per-flow cardinality in a memory-compact way, including virtual bitmap [4] and virtual HyperLogLog (vHLL) [5], whose memory cost can be less than one-bit memory per flow. Their idea is to exploit the highly skewed distribution of per-flow spreads, which are commonly observed in network traffic. Hence, they design a virtual-physical double-layer structure, where the spread-counting sketches of all flows are

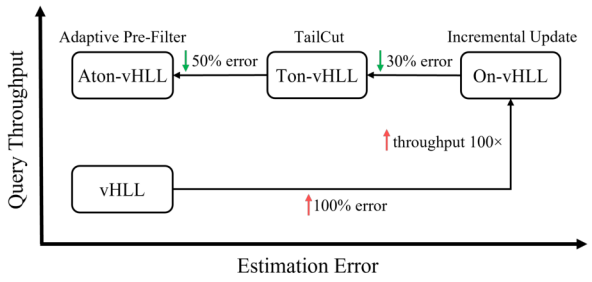


Fig. 1. Design objectives of On-vHLL, Ton-vHLL and Aton-vHLL.

squeezed into one-big-shared physical sketch. By this way, they ensure good estimation accuracy of superspreaders, and unavoidably sacrifice the accuracy of small flows, so that the overall memory cost of all flows is small enough to fit into SRAM.

However, most of previous works such as vHLL overlook the importance of online querying the per-flow spread. They follow the "Online Update, Offline Query" paradigm, in which the sketch is online updated by the data plane as each IP packet arrives, and at regular time intervals, transferred via PCIe bus to the control plane for offline sketch query. As a result, the time delay of the flow query operation can be several minutes in this offline query scenario. The main reason is that the time cost of querying a flow-spread sketch includes hundreds of memory accesses for each query, which is too expensive to run on the data plane and must be delayed to the control plane.

We argue that "Online Query" may become a new paradigm that can give real-time self-decisive capacity to the programmable data plane. We have already seen that, for the problem of measuring per-flow size, CountMin sketch [11] and HashPipe [12] have gained their dominant adoption both due to their memory compactness and their $O(1)$ query complexity. They deliver the promise that the heavy hitters whose flow sizes are abnormally large can be online detected entirely on the data plane. With such knowledge, the data plane can apply instant actions to the heavy flows, for example, rate limiting or route rescheduling. Similarly, for the problem of estimating the per-flow spread, we expect a new sketch to become popular if it can be queried with $O(1)$ time complexity. This allows the online detection of the top-k superspreaders, which have applications in cybersecurity and load balancing. For example, the destination IPs that have been contacted by the largest numbers of unique source IPs are perhaps under DDoS attacks.

For this problem of online tracking the per-flow cardinality, our paper presents three solutions named On-vHLL, Ton-vHLL, and Aton-vHLL, respectively. We illustrate their design rationales in Fig. 1. On-vHLL is to increase the query throughput by about 100 times than the traditional vHLL [5] that can only be queried offline. But the On-vHLL sketch provides worse per-flow spread estimation accuracy than vHLL. Next, we further propose Ton-vHLL to decrease the error by 30% than On-vHLL, and then Aton-vHLL for 50% error reduction than Ton-vHLL. Finally, the accuracy of Aton-vHLL becomes comparable with vHLL, without the loss of the online query feature. We introduce these three sketches as follows.

Firstly, we propose a sketch named On-vHLL (Online virtual HyperLogLog), which needs only $O(1)$ time cost to query a sketch for an arbitrary flow's spread. To tame the high time cost of flow cardinality query, we add auxiliary data structures to the traditional vHLL sketch [5] to cache their intermediate query result. These auxiliary structures called *incremental update units* (IUUs) can be online maintained as each IP packet arrives, and increase the query throughput by about 100 times. Although these IUUs can be used to online estimate the per-flow cardinality, how to make the estimated results unbiased needs tremendous development efforts. We have modified the method of hashing flow IDs to the sketch, so that the intermediate query results can be easily cached in IUUs. We have also incorporated a multi-stage design into On-vHLL to mitigate the impact of hash collision among flows. At the same time, On-vHLL can be deployed on the data plane based on multi-pipeline FPGA, with only some minor changes to avoid floating numbers and other complex operations.

Secondly, we propose a Ton-vHLL sketch (Tailcut online virtual HyperLogLog). Although On-vHLL reduces the query time cost to $O(1)$, its estimation accuracy is still worse than that of traditional vHLL. In this journal extension, our Ton-vHLL sketch applies a HLL register compression technique called *TailCut* [13] to On-vHLL. This technique reduces the size of each HLL register from 5 bits (or 8 bits in practice) to only 4 bits, thus reducing the estimation error by 30%.

Thirdly, we design another enhanced data structure named Aton-vHLL (Adaptive tailcut online virtual HyperLogLog). The Aton-vHLL can significantly slash the estimation error of flow cardinality by 50%. This is because it relocates the current top-k superspreaders from a Ton-vHLL sketch into a prefilter, where each superspreader has a unique 16-bit fingerprint and is given an exclusively-owned HyperLogLog [9] sketch to track its current cardinality. Of course, we can associate an auxiliary data structure named IUU to this dedicated HLL estimator, so that its time cost of cardinality query reduces to $O(1)$. We implement the prefilter by a min-heap so that the smallest flow can be quickly located at the tree root. This can facilitate the flow-level swap in/out between prefilter and sketch. We also accelerate the flow ID lookup operation in the min-heap, by the modern CPU's SIMD (Single Instruction Multiple Data) instruction sets named AVX2 and AVX-512.

We have conducted extensive experiments to compare our proposed three sketches with vHLL [5], rSkt1 [14], rSkt2 [14], AROMA [15], and AROMA+ (online query version of AROMA). Among them, rSkt1 and AROMA+ are the recent solutions that can support online estimation of per-flow spread. Our experiments show that On-vHLL is over 100 times faster than vHLL, rSkt2 and AROMA for the sketch query operation, assuming the number of registers d is 1024. This is because the number of memory accesses by vHLL, rSkt2 and AROMA is over 1026, 4098, and 67738 per packet, respectively, for sketch update and query combined. By contrast, the number of memory accesses of On-vHLL is smaller than 5. However, the flow spread estimation error of On-vHLL is 100% higher than vHLL. To compensate the accuracy loss, we propose the Ton-vHLL with TailCut and the Aton-vHLL with prefilter. In our experiments, Aton-vHLL provides 33% and 65% smaller identification error than rSkt1 and AROMA+, respectively, for the top-k superspreaders. Moreover, Aton-vHLL achieves

comparable accuracy with vHLL that can only be queried offline, and meanwhile it only needs 6 memory accesses per packet, including sketch update and query.

II. Related Work

Streaming data processing is a theoretical domain with over three decades of prior research. A data stream is a sequence of data elements that can be scanned by only one pass and in order [9], [10], [11], [16], [17]. Data stream processing techniques has many real-world applications. An important one is network traffic measurement, which is to process a stream of IP packets and extract useful statistics for each flow of packets that carry a common ID. The ID may be defined as source/destination IP addresses, or further include other fields in the packet header, such as protocol and source/destination ports. A high-speed switch, which is deployed at a vantage point to inspect the IP traffic, may process millions of active flows simultaneously. The challenge is the contradiction between the numerous flows and the limited size of the on-chip SRAM on a switch. There are mainly two categories of works to solve this problem.

The first category is data sketches. When querying an arbitrary flow ID, solutions in this branch are to answer an approximate value about the flow statistics. There are two kinds of per-flow statistics that are fundamental and extremely important: per-flow size and per-flow spread (or cardinality). The former counts the number of elements in a flow, while the latter counts the number of *unique* elements, which must filter the duplicated elements in a flow. There is a plethora of works to approximately count the per-flow size with low memory cost, such as CountSketch [16], CountMin [11], CounterBraid [18] and VirtualActiveCounter [3], etc. Our paper however is to address the second problem.

Perhaps due to its higher memory cost, per-flow spread estimation problem has a relatively smaller number of existing works than per-flow size estimation, such as virtual Bitmap (vBitmap) [4], virtual HyperLogLog (vHLL) [5], WavingSketch [19], ExtendedSketch [20], Self-Morphing Bitmaps [21] and randomized error-reduction sketch (rSkt) [14]. As mentioned before, vBitmap and vHLL have high query time cost, preventing them from online query. WavingSketch [19] can support online query, but it has low memory efficiency. It relies on a bloom filter to filter the duplicated pairs, and then uses a size counting sketch to track each flow's cardinality. Regretfully, the bloom filter is very memory consuming, whose number of bits is proportional to the sum of spreads of all flows. ExtendedSketch designs a reversible sketch that encodes both the flow cardinality and the flow ID [20]. Decoding the IDs of superspreaders without errors in a reversible sketch is often a very difficult task. By contrast, we avoid this problem by explicitly recording the fingerprints of superspreaders based on the online query results of flow spreads. Self-Morphing Bitmaps [21] can support online query, but does not allow the merging of multiple sketches obtained from distributed monitoring sites. The rSkt [14] can support merging, which allocates a primary HyperLogLog estimator and a complementary estimator for each flow. rSkt has two variants named rSkt1 and rSkt2. rSkt1 reduces the query overhead of rSkt to $O(1)$ by maintaining an array of HyperLogLog estimators, and giving each estimator an additional integer to accelerate its query. rSkt2 achieves better accuracy than rSkt through register-level memory sharing, but at the cost of $O(d)$ level query overhead and cannot support online queries. In summary, our goal is to design an accurate online per-flow spread tracking sketch that has $O(1)$ query time cost and can also support mergeability.

TABLE I
Notations

Symbol	Description
f	identifier of a particular flow
n_f	true cardinality of flow f
n_d	number of elements in the column picked by f
n	sum of the cardinalities of all flows $n = \sum_f n_f$
s	number of estimator stages
M	matrix of $d \times w$ registers in sketch
$\mathcal{K}$	upper bound of register value
d	number of rows allocated to the matrix M
w	number of columns allocated to the matrix M
h	a general hash function shared by all stages
Q	an array of w IUUs to accelerate estimation of n_d in sketch
N	a matrix IUU to accelerate estimation of n
B	an array of w base registers, each for a column
k	maximum size of prefilter to hold the top-k superspreaders
Φ	an array of k IUUs to accelerate estimation of n_d in prefilter
η	snapshot of n in when flow f swap-in
β	an array of k base registers, each for a column

Note: A symbol with the subscript f, e.g., Q_f, means it is hash-mapped by the flow f. A symbol with the superscript (l), e.g., $B^{(l)}$, means it belongs to the l-th stage. A symbol with the upper tilde symbol $\tilde{}$, e.g., $\tilde{M}$, means it is a structure processed by the TailCut technique. A symbol with the hat symbol $\hat{}$, e.g., $\hat{n}$, means it is an estimation of the true value.

The second category of works focus on capturing the IDs of "heavy/large" flows and only measure their flow sizes or spreads. Note that the flows with extra large sizes are called *heavy hitters*. The flows with extra large cardinalities are called *superspreaders*. The works that detect heavy hitters or superspreaders include LossyCounting [22], SpaceSaving [23], and HashPipe [12], TwoLayerSampling [24], SimpleSampling [25], TwoPhaseFiltering [26], Non-DuplicateSampling [27] and AROMA [15], etc. Their common strategy is to ignore the "light/small" flows, either by sampling techniques that automatically eliminate the records of light flows from a size-limited flow cache, or by filtering techniques that ignore the packets of light flows. Their shortcomings are that they inevitably lose track of "light/small" flows, and they must hold the IDs of the sampled large flows to detect hash collisions, which will occupy a large part of the memory space. By contrast, the data sketches indeed provide per-flow measurement, and they do not store any flow IDs or their short fingerprints, in order to avoid the extra memory cost and improve accuracy. Moreover, the accuracy of a data sketch can be significantly enhanced by adding a prefilter to sample and hold the "important" flows, which exist in the long tail of a highly skewed per-flow size/spread distribution [28]. Our paper will investigate the data sketching techniques.

III. Problem Definition

In this section, we formulate the problem of *online estimating per-flow cardinality*, and describe its key performance metrics. In Table I, we list the notations used by this paper.

Stream Model. We define a flow as a sequence of elements that can be scanned by an engine with only one pass. Note that a same element of a flow may appear multiple times. Hence, when counting the cardinality (i.e., the number of *distinct* elements) of a flow, we need to filter the duplicated elements.

Suppose there are m flows transmitted concurrently over an optical fiber. A stream processing engine (or more precisely, packet processing pipeline of a high-speed switch) will receive a stream of IP packets, sorted by their arrival time. From these IP packets, we can extract a sequence of flow-element pairs:

$$\mathcal{S} = \langle f_1, e_1\rangle, \ldots, \langle f_t, e_t\rangle, \ldots, \langle f_\ell, e_\ell\rangle,$$

where t is the arrival time in the range $[1, \ell]$, f_t is the flow ID of the t-th packet, and e_t is the element ID. Let n_f be the spread or cardinality of a flow f. Then, we have the following formula to calculate the exact value of the flow cardinality n_f.

$$n_f = |\{\langle f_t, e_t\rangle \mid t \in [1, \ell] \wedge f_t = f\}| \tag{1}$$

This formula captures all the pairs whose flow IDs f_t are equal to f, uses these pairs to construct a set $\{\langle f_t, e_t\rangle \mid \ldots\}$, and computes the cardinality of the set. Let n be the number of distinct flow-element pairs in the stream $\mathcal{S}$, where the two pairs $\langle f_{t1}, e_{t1}\rangle$ and $\langle f_{t2}, e_{t2}\rangle$ are considered different if $f_{t1} \neq f_{t2}$ or $e_{t1} \neq e_{t2}$. Clearly, this implies $n = \sum_f n_f$, or say, the total cardinality is equal to the sum of the cardinality of each flow.

Our research problem is to scan the packet stream $\mathcal{S}$ by one pass, and determine the cardinality n_f for each flow f. A naive solution is to *exactly* count the flow cardinality n_f, following its definition in (1). This needs to allocate an exclusively owned buffer to each flow f. When processing the packet stream, we may capture all the pairs $\langle f_t, e_t\rangle$ with $f_t = f$, and put them in the buffer of the flow f to exactly count its cardinality n_f. However, in a high-speed network, there can be tens of millions of flows whose IP packets are transmitted during the monitoring time period. Capturing all the pairs of each flow f will consume a variably large amount of memory, which can easily exceed the SRAM capacity (tens of MBs) of the line card on a high-speed switch. It is also unnecessary to know the exact value of flow spread n_f. We can use a memory-compact probabilistic data sketch to record the arrival event of the pair $\langle f_t, e_t\rangle$ and *approximately* count the flow spread n_f.

There are many data structure designs of data sketches to approximately count the cardinality of each flow, when there exist a very large number of flows. A straightforward solution is to roughly count the flow cardinality n_f, by allocating an exclusively owned HyperLogLog (HLL) estimator [9] for each flow f. This can ensure the relative approximation error of each n_f is $\frac{1.04}{\sqrt{d}}$, where d is the number of registers given to an HLL estimator. For example, if we can give each HLL estimator $d = 1024$ registers, which means $5d$ bits or 5KB memory for each flow f, then the flow cardinality n_f will have $\frac{1.04}{\sqrt{1024}} = 3.25\%$ *relative estimation error*. However, giving 5KB memory to each flow is not sufficiently memory-compact, when there are millions of flows. Note that the flow spreads follow a highly skewed distribution, e.g., zipf distribution, in which a small proportion of flows have much larger spreads than other flows. Therefore, the challenge is how to design an extremely memory-compact data structure to exploit this fact.

Performance Metrics. For each flow f, we will give a rough estimation about its spread n_f, which is denoted by $\hat{n_f}$. We must ensure the *absolute estimation error* $\hat{n_f} - n_f$ is bounded by a threshold $\pm\epsilon(n_f + pn)$, with a probability above $1 - \delta$.

$$Pr\{|\hat{n_f} - n_f| \leq \epsilon(n_f + pn)\} \geq 1 - \delta \tag{2}$$

Here, p is a global-noise disturbance ratio, which is decided by the memory fraction, i.e., the memory given to a flow f divided by the memory given to all flows, according to proof [29].

Besides the estimation accuracy of flow spread in (2), there are three other performance metrics. The first metric is the *query throughput*, i.e., the number of query operations that can be performed per time unit. As a packet arrives with the tuple $\langle f_t, e_t\rangle$, we will call hash function to select multiple memory units and perform read-modify-write. The fewer memory units we have to access, the higher throughput we will achieve. The second performance metric is the memory overhead to meet the error bounding constraint in (2). If giving more memory, we can attain better flow spread estimation accuracy. The third metric is the identification accuracy of the top-k superspreader.

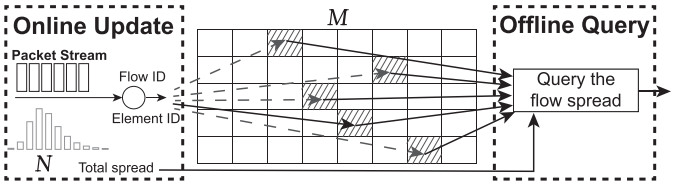


Fig. 2. Online update and offline query of per-flow spread by virtual HLL.

IV. On-vHLL Sketch

In this section, we present our On-vHLL (Online virtual HyperLogLog) sketch, which requires a constant number of memory accesses for both sketch insertion and query.

A. Basic Data Structure

In following, we describe the data structure of our On-vHLL sketch. As shown in Fig. 2, for tracking the per-flow cardinality, we create a matrix of registers M. Each register is a small memory unit with only five bits. Let d be the number of rows, and w be the number of columns. Its register at jth row and ith column is denoted by $M[j, i]$, with $0 \leq j < d$ and $0 \leq i < w$. This matrix is shared by all flows.

To squeeze millions of flows into this compact memory space, we adopt a virtual-physical memory sharing scheme. Each flow f is given an array of d registers to track its cardinality. Let M_f be this array of "virtual" registers, whose ith register is randomly chosen from the ith row of matrix M.

$$M_f[j] = M[j,\ h(f \oplus j) \bmod w], \qquad \text{with } 0 \leq j < d \tag{3}$$

Here, h is a hash function for randomly picking a register, and $\oplus$ is the concatenation operator. Clearly, this virtual estimator M_f is not dedicated to f. Its virtual register $M_f[i]$ may be selected by another flow f' due to hash collision.

A similar virtual-physical sharing scheme is used by vHLL (virtual HyperLogLog) [5]. As shown in Fig. 2, the update operation of vHLL sketch needs to access only one register, which is efficient. But its query operation needs to access all the registers in M_f, which is quite slow. We will reduce the query time cost to $O(1)$, by the techniques to describe later.

B. Accelerate Total Spread Estimation

In this subsection, we describe how to estimate the total cardinality n at $O(1)$ time cost. By contrast, for vHLL sketch, its query of total spread n has to scan the entire matrix M.

Online Update. We use the following procedure to update our sketch for each packet. When a packet with tuple $\langle f, e\rangle$ arrives, we use $f \oplus e$ to decide which register to update in the virtual estimator M_f, where $\oplus$ is the concatenation operator. Applying hash function h, we generate a 32-bit hash value x.

$$x = h(f \oplus e) \qquad j = \langle x_1 x_2 \dots x_b \rangle \qquad q = \langle x_{b+1} x_{b+2} \dots \rangle$$

The hash value x is split into two parts: j is the initial b bits that are used to select a register in the virtual estimator M_f; q takes the remaining bits that are used to update the register $M_f[j]$. So here we assume that the number of rows of matrix M is a power of two: $d = 2^b$, whose configured value ranges from 2^7 to 2^{13} or larger, depending on the predefined accuracy requirement. Applying HyperLogLog's register updating rule,

$$M_f[j] := \max\big(M_f[j], \rho(q)\big), \tag{4}$$

where $:=$ is the assignment operator, $\rho(q)$ is one plus the longest run of leading zeros in the binary representation of the hash value q, and the virtual estimator M_f is defined in (3).

Offline Query. For the vHLL sketch [5], the spread of a flow f is queried by the following formula. It can explain well why vHLL is slow and can only be queried in an offline way.

$$\hat{n_f} = \tfrac{w}{w-1}(\hat{n_d} - \tfrac{1}{w}\hat{n}) \tag{5}$$

Here, $\hat{n_d}$ is the estimated number of flow-element pairs that are mapped to the virtual estimator M_f, using the following equation given by the renowned HyperLogLog paper [9]:

$$\hat{n_d} = \alpha_d \cdot d^2 \cdot \big(\textstyle\sum_{j=0}^{d-1} 2^{-M_f[j]}\big)^{-1}, \tag{6}$$

where α_d is a constant bias corrector that depend on d configuration. Specifically, $\alpha_{16} \approx 0.673$, $\alpha_{32} \approx 0.697$, $\alpha_{64} \approx 0.709$, $\alpha_d \approx 0.7213/(1 + 1.079/d)$ as soon as $d \geq 128$. To make this estimation formula unbiased in the entire operating range, it must be combined with LinearCounting [10] and a maximum likelihood estimator for estimating small cardinalities. Please refer to [13] for detailed formula, and we omit them here for simplicity. The symbol $\hat{n}$ in (5) is the estimated total spread from the register matrix M, given by a similar formula to (6).

$$\hat{n} = \alpha_{wd} \cdot (wd)^2 \cdot \big(\textstyle\sum_{j=0}^{d-1} \sum_{i=0}^{w-1} 2^{-M[j,\,i]}\big)^{-1} \tag{7}$$

Clearly, this formula has to scan the entire register matrix M.

Total Spread IUU. We can reduce the time cost of estimating the total spread to $O(1)$. Since each register has five bits to count cardinality within $2^{2^5} \approx 4 \times 10^9$, (7) can be rewritten as

$$\hat{n} = \alpha_{wd} \cdot (wd)^2 \cdot \big(\textstyle\sum_{v=0}^{31} N[v] \cdot 2^{-v}\big)^{-1}, \tag{8}$$

where the array N records the number of registers among M that takes the value v, which is illustrated as a histogram in Fig. 2. Clearly, the time cost of (8) is $O(1)$, as it reads the array N with 32 integers, which can be prefetched into data cache.

This histogram array N consisting of thirty-two 16-bit integers has low memory cost. It is also easy to maintain per packet: When a packet arrives, if the register $M_f[j]$ is modified by (4), we can correspondingly update the histogram array N. Since N essentially caches the intermediate query result, we call N the *incremental update unit* (IUU) for the total spread.

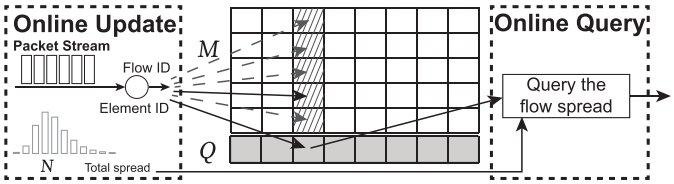


Fig. 3. Online per-flow sketch by mapping flows to columns.

Even when the total spread n can be estimated by constant time cost in (8), the query operation for a flow f's spread in (5) has $O(d)$ time cost. It needs to read every register of M_f as in (6), which incurs d memory accesses. Making things worse, to guarantee high estimation accuracy, d must be configured to a few thousands. This is because the expected relative error of HyperLogLog [9] is $\frac{1.04}{\sqrt{d}}$. Moreover, the virtual estimator M_f has its thousands of registers distributing randomly in matrix M as in Fig. 2. This random memory access pattern is unpredictable and difficult to optimize by cache prefetching. As a result, reading all the registers of virtual estimator M_f is impossible in the data plane of a high-speed switch.

C. Design of a Single Stage

In this subsection, we present our first attempt to reduce the time cost of sketch query to $O(1)$. For the vHLL sketch, there are w^d different combinations for the register set of a virtual estimator M_f, assuming the matrix M in Fig. 2 has w columns and d rows. Then, for vHLL, the probability of two flows sharing a same virtual estimator with d registers is as small as $\frac{1}{w^d}$. This can help a flow to prevent its d registers to completely hash-collide with those of a superspreader, thus effectively improving its spread estimation accuracy. However, the explosion of combination number w^d also prevents us from caching the intermediate query result of a virtual estimator.

Matrix Column IUU. For query speedup, we let each virtual estimator M_f have its d registers in a same column as in Fig. 3, or say, we map each flow ID f to a column of registers.

$$M_f[j] = M[j,\ h(f) \bmod w], \qquad \text{with } 0 \leq j < d \tag{9}$$

Note that this column of d registers are stored contiguously in memory, so that (6) can be computed more time efficiently. Afterwards in this paper, we assume all the matrices are stored by the *column-major order*, where the cells of a column are arranged contiguous in memory. Since each flow now choose a random column of registers for estimating its cardinality, we can cache the intermediate query result for that column, which is called an *incremental update unit (IUU)*. In Fig. 3, we illustrate the IUUs of all columns as a row of gray blocks, each of which can accelerate the estimation of n_d, i.e., the sum of cardinalities of the flows that are mapped to that column.

This data structure design allows us to reduce the time cost of sketch query to $O(1)$. Let Q be the array of IUUs, for each column of the matrix M. Let Q_f be the IUU of flow f. Then,

$$Q_f = Q[h(f) \bmod w],$$

where $Q_f[v]$ records the number of registers among M_f that carry a value v. Note that the IUU Q_f is a small array with 32 integers, called a histogram, since a register with 5 bits only have 32 possible integer values. To maintain the histogram Q_f per packet arrival, we check whether the register $M_f[j]$ will be updated by (4), and if it does, before updating $M_f[j]$, we reduce $Q_f[M_f[j]]$ by one and increment $Q_f[\rho(q)]$ by one.

We leverage the IUU Q_f to quickly estimate the number of unique flow-element pairs mapped to that column. Clearly, the time cost of equation (10) is $O(1)$.[1]

$$\hat{n_d} = \alpha_d \cdot d^2 \cdot \left(\sum_{v=0}^{31} Q_f[v] \cdot 2^{-v}\right)^{-1} \tag{10}$$

This formula (10) allows each register to give an independent cardinality estimation $2^{M_f[i]}$, and computes the harmonic average of all registers in the virtual estimator M_f. Next, we estimate the total cardinality n of all the flows as follows.

$$\hat{n} = \sum_{i=0}^{w-1} \alpha_d \cdot d^2 \cdot \left(\sum_{v=0}^{31} Q[i][v] \cdot 2^{-v}\right)^{-1} \tag{11}$$

Different from (8), this equation estimates the total cardinality n by computing the sum of cardinality estimations for each ith column of HLL registers, $0 \le i < w$. Thus, (11) is unbiased since each column gives an unbiased cardinality estimation. By contrast, (8) contains a minor bias, but it runs much faster. Finally, with $\hat{n_d}$ and $\hat{n}$, we estimate the flow f's cardinality n_f.

$$\hat{n_f} = \frac{w}{w-1}(\hat{n_d} - \frac{1}{w}\hat{n}) \tag{12}$$

This sketch design has $O(1)$ time cost for query operation. However, the price is the severe degradation of spread estimation accuracy. Suppose there are top-k superspreaders whose spreads are extra larger than other flows. The probability for a flow f to collide with any of the superspreaders in its mapped column is $1-(1-\frac{1}{w})^k \approx 1-e^{-k/w}$. If a flow f is by chance mapped to a column occupied by a superspreader, its cardinality estimation will be severely inflated.

D. Multi-Stage on-vHLL Sketch

In this subsection, we present our On-vHLL (Online virtual HyperLogLog) sketch, which needs $O(1)$ time cost for sketch query. Of course, a single estimator in Fig. 3 will have high variance due to the chance of hash collision in a same column with superspreaders. To reduce the variance, a common practice in data sketching algorithms is to run independent copies of the estimator in parallel and combine their outputs. Many well-known sketches such as CountSketch [16] and CountMin [11] use this technique to tame the high variance of a single estimator. Our single estimator in (12) can produce an unbiased result by removing the noise $\frac{1}{w}\hat{n}$. From the perspective of combining the results of multiple stages, we regard our On-vHLL is more similar to CountSketch [16], which applies the median or average operator for result aggregation.

We illustrate our multiple-stage sketch design in Fig. 4. Each stage has a matrix of HLL registers which can give an independent estimation for an arbitrary flow's spread.

[1]The basic units of data transfer in CPU cache are not bytes, but cache lines. So we can speedup Eq. (10) by CPU cache alignment. Each IUU occupies 32 (or 64) bytes DRAM, which can fit into a single CPU cache line. Then, Eq. (10) needs only one bus transaction to fetch histogram Q_f into cache.

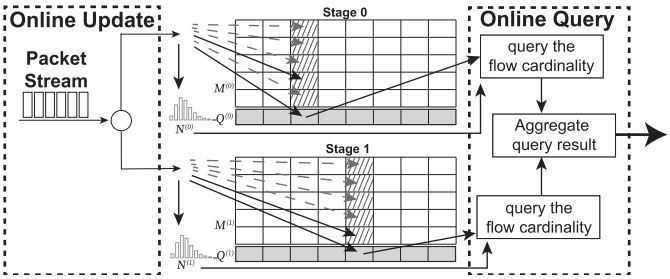


Fig. 4. On-vHLL sketch consists of multiple stages.

To mitigate the impact of outlier stages which have hash-collided with superspreaders, we apply the average operator to aggregate the flow spread estimation results of all the stages.

Symbol Definitions. Let s be the number of stages, which is typically configured to four. In the lth stage with $0 \le l < s$, let $M^{(l)}$ be the register matrix, let $Q^{(l)}$ be the array of IUUs for each column of $M^{(l)}$, and let $N^{(l)}$ be the IUU of the entire matrix $M^{(l)}$. Suppose each stage configures the register matrix with the same dimensions, i.e., the same number of rows d and the same number of columns w. Each stage is associated with a different hash function $h^{(l)}$ for its column selection. Let $M_f^{(l)}$ be the virtual estimator of flow f in the lth stage, and let $Q_f^{(l)}$ be the IUU of flow f in the lth stage. Then,

$$M_f^{(l)}[j] = M^{(l)}\left[j,\ h^{(l)}\big(h(f)\big) \bmod w\right],\ \text{with } 0 \le j < d, \tag{13}$$

$$Q_f^{(l)} = Q^{(l)}\left[h^{(l)}\big(h(f)\big) \bmod w\right]. \tag{14}$$

We give each stage $l \in [0, s)$ a unique hash function $h^{(l)}$. We apply $h^{(l)}$ to the 32-bit fingerprint $h(f)$ of a flow ID f, so that f can be randomly mapped to different columns $h^{(l)}\big(h(f)\big) \bmod w$ in different stages. We have shown this phenomenon in Fig. 4. Here, we apply a stage's unique hash function $h^{(l)}$ to the fingerprint $h(f)$, instead of flow ID f, to implement the swap-out mechanism of prefilter in Section VI.

The procedure of On-vHLL can be divided into two parts: sketch update and sketch query. We describe them separately.

Online Sketch Update. As a packet arrives carrying flow ID f and element ID e, in order to track the cardinality of flow f, we need to update $M_f^{(l)}$, $Q_f^{(l)}$ and $N^{(l)}$ in each stage $l \in [0, s)$.

We present the Algorithm 1 to processes an arrival element e with flow ID f. For each lth stage, we run the code in lines 2-6. At line 2, we apply the hash function $h^{(l)}$ to the arrival flow element $f \oplus e$, where $\oplus$ is the concatenation operator. Then, we extract the initial b bits as j and the remaining bits as q. Line 3 calculates $\rho(q)$, which is 1 plus the longest run of leading zeros in the binary format of q, and compares it with the register $M_f^{(l)}[j]$. If $\rho(q)$ is larger, we update the IUU $N^{(l)}$ of total cardinality at line 4, update the IUU $Q_f^{(l)}$ at line 5, and update the register $M_f^{(l)}[j]$ at line 6. We still maintain $N^{(l)}$, since (8) is a faster way to estimate total cardinality than (11).

Online Sketch Query. After inserting the arrival packet $\langle f, e\rangle$ into the sketch, we estimate the spread of the flow f. The query procedure is shown in the right-hand side of Fig. 4.

Algorithm 1 Update On-vHLL When a Packet Arrives

Data: Flow ID f, Element ID e
1 **foreach** $l \in [0, s)$ **do**
2 $x = h^{(l)}(f \oplus e)$, $j = \langle x_1 x_2 \dots x_b \rangle$, $q = \langle x_{b+1} \dots \rangle$
3 **if** $\rho(q) > M_f^{(l)}[j]$ **then**
4 $N^{(l)}[M_f^{(l)}[j]] -= 1$, $N^{(l)}[\rho(q)] += 1$
5 $Q_f^{(l)}[M_f^{(l)}[j]] -= 1$, $Q_f^{(l)}[\rho(q)] += 1$
6 $M_f^{(l)}[j] = \rho(q)$

Firstly, we query the s stages in parallel, each of which gives an independent estimation of the flow spread denoted by $\hat{n}_f^{(l)}$. For On-vHLL, each stage has w columns of registers, and each column is associated with a query acceleration unit, called IUU. Let $Q_f^{(l)}$ be the IUU of the column picked by flow f in stage l, which is defined in (14). Then, we have

$$\hat{n}_d^{(l)} = \alpha_d \cdot d^2 \cdot (\textstyle\sum_{v=0}^{31} Q_f^{(l)}[v] \cdot 2^{-v})^{-1} \tag{15}$$

$$\hat{n}_f^{(l)} = \tfrac{w}{w-1}\big(\hat{n}_d^{(l)} - \tfrac{1}{w}\hat{n}^{(l)}\big), \tag{16}$$

where α is bias corrector with $\alpha_{16} \approx 0.673$, $\alpha_{32} \approx 0.697$, $\alpha_{64} \approx 0.709$, $\alpha_d \approx 0.7213/(1+1.079/d)$ when $d \geq 128$, and

$$\hat{n}^{(l)} = \textstyle\sum_{i=0}^{w-1} \alpha_d \cdot d^2 \cdot \big(\sum_{v=0}^{31} Q^{(l)}[i][v] \cdot 2^{-v}\big)^{-1}, \tag{17}$$

$$\text{or } \hat{n}^{(l)} = \alpha_{wd} \cdot (wd)^2 \cdot (\textstyle\sum_{v=0}^{31} N^{(l)}[v] \cdot 2^{-v})^{-1}, \tag{18}$$

where the matrix IUU $N^{(l)}$ and the column IUU $Q^{(l)}$ have the relation $N^{(l)}[v] = \sum_{i=0}^{w-1} Q^{(l)}[i][v]$. Eq. (17) is completely unbiased, while Eq. (18) has a small degree of bias but runs faster. We can prove $\hat{n}_f^{(l)}$ is an unbiased estimation of n_f [29].

Secondly, we aggregate the estimated results of all stages to obtain a more accurate estimate $\hat{n_f}$ of the flow f's cardinality.

$$\hat{n_f} = \tfrac{1}{s} \textstyle\sum_{0 \leq l < s} \hat{n}_f^{(l)} \tag{19}$$

This query result $\hat{n_f}$ may be inserted into a min-heap to help record the flow IDs of top-k superspreaders, or be compared with a predefined threshold for security alarming, depending on the detailed applications built upon the On-vHLL sketch.

Accuracy Evaluation. For the flow spread estimator in (19), we have analyzed its bias and variance in the appendix online [29]. We have proved that it is unbiased with $E(\hat{n_f}) \approx n$, and

$$Var(\hat{n}_f) = \tfrac{1}{s}\big(\tfrac{w}{w-1}\big)^2\big(\tfrac{\gamma_d^2}{d}A^2 + (\tfrac{\gamma_d^2}{d}+1)B + \tfrac{\gamma_d^2}{wd}(\tfrac{n}{w})^2\big), \tag{20}$$

where γ_d is 1.04 for On-vHLL, when the number of registers in a virtual estimator $d \geq 128$. The definitions of A and B are

$$A = n_f + (n - n_f)\tfrac{1}{w}, \quad B = (n - n_f)\tfrac{1}{w}(1 - \tfrac{1}{w}). \tag{21}$$

E. Implementation Issues on Hardware Data Plane

Our multi-stage sketch design may be deployed on either software data plane (e.g., Open vSwitch and VPP) or hardware data plane (e.g., Intel Tofino). The latter platform can provide much higher packet processing throughput, but has much more programming restrictions than the former. Although the query operation of our On-vHLL has been redesigned to have $O(1)$ time cost, it still needs a few modifications before the deployment on hardware data plane with many implementation restrictions. To prove that our On-vHLL sketch is indeed suitable for hardware data plane, we have developed a system prototype based on the P4-programmable Intel Tofino switch.

Firstly, our original design of IUU is a small array consisting of 32 integers (called a histogram), which is difficult to be scanned by the hardware data plane when we query the sketch, since the data plane allows only a limited number of on-chip memory accesses per packet. Therefore, we simplify the definitions of the matrix IUU $N^{(l)}$ and the column IUU $Q_f^{(l)}$ to

$$N^{(l)} = \textstyle\sum_{j=0}^{d-1} \sum_{i=0}^{w-1} 2^{31 - M^{(l)}[j,i]}, \tag{22}$$

$$Q_f^{(l)} = \textstyle\sum_{j=0}^{d-1} 2^{31 - M_f^{(l)}[j]}. \tag{23}$$

Clearly, the IUUs are not encoded as histograms anymore, but as these two integer numbers. Since the value v of a 5-bit HLL register ranges from 0 to 31, 2^{31-v} is always an integer after the amplification by 2^{31}. So for the register matrix $M^{(l)}$ of the lth stage, $N^{(l)}$ records the amplified denominator of the harmonic mean of all its registers, and $Q_f^{(l)}$ records the amplified denominator of the harmonic mean of the registers on its column $h^{(l)}(f) \bmod w$. The IUUs in (22) and (23) can be incrementally updated per packet. So we can change the IUU update commands at the lines 4 and 5 of Algorithm 1 to

$$N^{(l)} -= 2^{31 - M_f^{(l)}[j]} - 2^{31-\rho(q)}, \tag{24}$$

$$Q_f^{(l)} -= 2^{31 - M_f^{(l)}[j]} - 2^{31-\rho(q)}. \tag{25}$$

Secondly, the flow spread estimation formulas in (15), (17) and (18) relies on harmonic average computation, which has to manipulate a series of floating numbers 2^{-v}, $1 \leq v \leq 31$. However, in order to keep up with high line speed at hundreds of Gbps, many hardware implementations of data plane, for example, by P4 language [30], do not support the floating number calculations and other complex operations, e.g., logarithmic and exponential functions. So using the above new definitions of IUUs, we simplify the flow spread estimation formulas as

$$\hat{n}_d^{(l)} = \tfrac{\alpha_d \cdot d^2 \cdot 2^{31}}{Q_f^{(l)}}, \tag{26}$$

$$\hat{n}^{(l)} = \textstyle\sum_{0 \leq i < w} \tfrac{\alpha_d \cdot d^2 \cdot 2^{31}}{Q^{(l)}[i]} \text{ or } \hat{n}^{(l)} = \tfrac{\alpha_{wd} \cdot (wd)^2 \cdot 2^{31}}{N^{(l)}}, \tag{27}$$

$$\hat{n}_f^{(l)} = \tfrac{w}{w-1}\big(\hat{n}_d^{(l)} - \tfrac{1}{w}\hat{n}^{(l)}\big). \tag{28}$$

Note that the division operator is also not supported by the P4 language, but can be implemented by the advanced computing units named MathUnit, available on the Intel Tofino switch.

Another subtle issue originates from the memory access restriction of P4-programmable Tofino switch: During hardware synthesis, it allows a packet processing "pipeline" to apply at most one read and one write operations to an on-chip register. In order for our multi-stage design not to violate this rule, we implement each stage of the On-vHLL sketch by a pipeline of the Tofino switch. As a result, the memory allocation and per-packet memory accesses of these pipelines (or stages) are completely isolated from each other. Then, for the lth pipeline, we allocate the three kinds of registers $M_f^{(l)}$, $Q_f^{(l)}$ and $N^{(l)}$. When each packet passes through the lth pipeline, it needs to apply only one read and one write to each kind of the registers.

V. More Accurate Sketch With Smaller Registers

Although our On-vHLL sketch can reduce the time cost of querying a flow f's spread to $O(1)$, its spread estimation error nearly doubles as compared with vHLL [5], as shown later by Fig. 11 and Fig. 12 in the experiments. So in this section, to compensate the accuracy loss, we propose a new sketch named Ton-vHLL (Tail-cut Online virtual HyperLogLog). It can reduce the memory cost by 50% as compared with On-vHLL, or equivalently, it can reduce the spread estimation error by 30% when they are given the same amount of memory.

The key technique of Ton-vHLL is to compress each HLL register used by On-vHLL from 5 bits to 4 bits, without degrading its flow spread estimation accuracy. Moreover, for many industrial projects, each 5-bit HLL register is implemented by a byte, so that each register can be quickly located in a byte array, which however will waste 3 bits in each byte. If we can compress each register from 5 bits to 4 bits, it means we allow each byte to hold two registers without wasting any bit. Thus, a 4-bits HLL register can in fact save 50% memory.

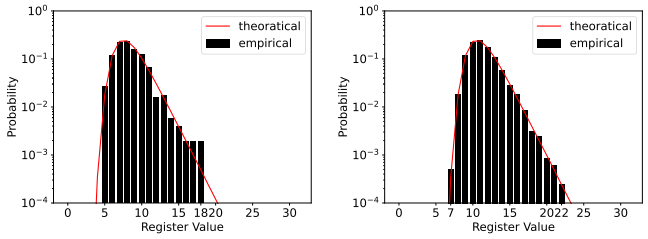
Basic Idea. To motivate our new register design, we re-examine the old design of a HLL register. HyperLogLog is an excellent cardinality estimation algorithm, providing $\frac{1.04}{\sqrt{d}}$ relative estimation error at the expense of $5d$ bits memory, where d is the number of HLL registers. Each register is given 5 bits, so that the counting range is as large as $2^{2^5} = 2^{32} \approx 4 \cdot 10^9$. Let n_d be the number of unique elements that are mapped to these d registers. Consider a column of registers M_f picked by the flow f in the matrix M. Here, for simplicity, we omit the index of the stage l. According to [9], the probability for a HLL register $M_f[j]$ defined in (13) to carry a value v is

$$Pr\{M_f[j] = v\} = \begin{cases} (1 - \frac{1}{d})^{n_d} & \textbf{if } v = 0 \\ (1 - \frac{1}{d\,2^v})^{n_d} - (1 - \frac{1}{d\,2^{v-1}})^{n_d} & \textbf{if } v \geq 1, \end{cases}$$

where n_d is the number of unique elements that are mapped to the column M_f, and may or may not belong to the flow f.

In Fig. 5, we show the probability distribution for a HLL register to carry different values. The red curve is theoretical probability given by the above formula. The black histogram shows the empirical distribution. In plot (a), the number of registers $d = 512$ and the number of elements $n_d = 128d$. In plot (b), $d = 8192$ and $n_d = 1024d$. Clearly, the shape of the distribution is not significantly affected by d and n_d. As the load factor $\frac{n_d}{d}$ decreases or increases, the distribution only shifts leftwards or rightwards. As the number of registers d grows from 512 to 8192 in plot (a) and (b), the empirical histogram becomes more consistent with the theoretical curve.

Moreover, according to our observation in experiments, the histogram of register values always shows a strongly right-skewed distribution, whose left tail follows a steep slope, and whose right tail is long and thin. The distance between the minimum register and the maximum register is no larger than 16 for most circumstances, both when the number of registers d is configured to a small value 512 or a large value 8192. This inspires us to track the current minimum value for the column of HLL registers in M_f by additionally maintaining



(a) $d = 512, n_d = 128d$ (b) $d = 8192, n_d = 1024d$

Fig. 5. Probability distributions of register values, for a varying number of registers d and a varying number of elements n_d.

a base register B_f. With this known minimum register value, the registers in M_f can store their own offsets relative to B_f, which can be encoded by only four bits without accuracy loss.

The previous work [13] mentioned a similar technique that compresses a 5-bit HLL register to a 4-bit offset register. However, this technique needs to maintain a base register that records the minimum value for an array of HLL registers. This cannot be realized for the vHLL [5], since the registers of a virtual estimator are scattered in the vHLL's register matrix in Fig. 2. But it becomes possible for our new On-vHLL, since the registers of a virtual estimator are hashed to one same column in Fig. 3. For that column of registers, it is possible to maintain a base register. Additionally, the time cost of updating the base register in [13] is $O(d)$ by scanning the column of d registers, but we can leverage the column's IUU histogram to update the base register more efficiently with $O(1)$ time cost.

Base Register and Offset Registers. Thanks to our mapping of a flow ID f to a column as shown in Fig. 3, it now becomes possible to maintain a base register $B[i]$ for each ith column of registers in the matrix M, which can help compress that column of 5-bits registers to an array of 4-bits offset registers.

Suppose we have a multi-stage design as in Fig. 4. Let $M^{(l)}$ be the $d \times w$ matrix of HLL registers in the l-th stage. Let $B^{(l)}$ be the array of base registers in the l-th stage, whose length is w. The ith base register $B^{(l)}[i]$, $0 \leq i < w$, is to record the smallest value among the ith column of HLL registers in $M^{(l)}$.

$$B^{(l)}[i] = \min_{0 \leq j < d} M^{(l)}[j, i] \tag{29}$$

This base register can be updated with $O(1)$ time cost as each packet arrives, thanks to the column-wise IUU $\tilde{Q}_f^{(l)}$ defined latter in (36). Let $\tilde{M}^{(l)}$ be the $d \times w$ matrix of offset registers in the l-th stage, relative to the base registers $B^{(l)}$. Then, we have

$$\tilde{M}^{(l)}[j, i] = \max\left(\tilde{M}^{(l)}[j, i],\ \rho(q) - B^{(l)}[i]\right), \tag{30}$$

where j is the row index, i the column index, and for the arrival element e with flow ID f, we calculate the following four hash values $x = h^{(l)}(e)$, $j = \langle x_1 x_2 \ldots x_b \rangle$, $q = \langle x_{b+1} x_{b+2} \ldots \rangle$, and $i = h^{(l)}(f) \bmod w$. Recall that $\rho(q)$ is 1 plus the longest run of leading zeros in the binary format of the hash value q.

However, since each offset register $\tilde{M}^{(l)}[j, i]$ is given only four bits memory, it has an upper bound of recording the offset value $\rho(q) - B^{(l)}[i]$, which is denoted as $\mathcal{K} = 2^4 = 16$. Considering the upper bound $\mathcal{K}$, (30) needs to be modified as

$$\tilde{M}^{(l)}[j, i] = \max\left(\tilde{M}^{(l)}[j, i],\ \min(\rho(q) - B^{(l)}[i],\ \mathcal{K} - 1)\right), \tag{31}$$

where min operator is to round down the offset to its largest possible value $\mathcal{K} - 1$, when it surpasses the bound $\mathcal{K}$. We call

this rounding technique *TailCut*, because it essentially cuts off the long tail of the probability distribution of HLL register values shown in Fig. 5 beyond a floating bound $B[i]+\mathcal{K}$.

With the base register array $B^{(l)}$ and the offset register matrix $\tilde{M}^{(l)}$, we can estimate the cardinality of a flow $\hat{n}_f^{(l)}$ by

$$\hat{n}_d^{(l)} = \alpha_d \cdot d^2 \cdot \left(\sum_{j=0}^{d-1} 2^{-B_f^{(l)} - \tilde{M}_f^{(l)}[j]}\right)^{-1}, \tag{32}$$

$$\hat{n}^{(l)} = \sum_{i=0}^{w-1} \alpha_d \cdot d^2 \cdot \left(\sum_{j=0}^{d-1} 2^{-B^{(l)}[i] - \tilde{M}^{(l)}[j,i]}\right)^{-1}, \tag{33}$$

and then we estimate $\hat{n}_f^{(l)}$ by (16). Here, $B^{(l)}$ is the array of base registers in (29), and we define the jth offset register $\tilde{M}_f^{(l)}[j]$ and the base register $B_f^{(l)}$ selected by the flow f as

$$\tilde{M}_f^{(l)}[j] = \tilde{M}^{(l)}\left[j, h^{(l)}\left(h(f)\right) \bmod w\right], \text{ with } 0 \le j < d, \tag{34}$$

$$B_f^{(l)} = B^{(l)}\left[h^{(l)}\left(h(f)\right) \bmod w\right] = \min_{0\le j<d} M_f^{(l)}[j], \tag{35}$$

where $h^{(l)}$ is a unique hash function given to the lth stage, which is applied to the 32-bit fingerprint $h(f)$ of the flow f to pseudo-randomly select a column, and $\tilde{M}^{(l)}$ is the matrix of offset registers for lth stage, which has been explained in (31).

Online Sketch Query. The cardinality query formula in (32) has high time cost at the scale of $O(d)$. It has not yet leveraged the column-wise IUU (incremental update units) in Fig. 3 to reduce the query time cost. We will present our Ton-vHLL sketch, whose matrix of registers $\tilde{M}_f^{(l)}$ are compressed to four bits each, and which can be queried online by $O(1)$ time cost.

Let $\tilde{Q}^{(l)}$ be an array of w IUUs, each of which is used to accelerate the cardinality query of a column of offset registers in $\tilde{M}^{(l)}$. The ith IUU $\tilde{Q}^{(l)}[i]$ consists of $\mathcal{K}$ counters, which are to record for any integer value $v \in [0, \mathcal{K})$ how many offset registers carry the v value among the ith column of $\tilde{M}^{(l)}$. More specifically, the vth counter $\tilde{Q}^{(l)}[i, v]$ records the number of offset registers that carry the value v, among the offset registers $\tilde{M}^{(l)}[j, i]$, $0 \le j < d$. Similar to (35) and (34), we can define the IUU hashed by a flow f, which can simplify our formula:

$$\tilde{Q}_f^{(l)}[v] = \tilde{Q}^{(l)}\left[h^{(l)}\left(h(f)\right) \bmod w\right][v] = \sum_{0\le j<d} \mathbf{1}_{\tilde{M}_f^{(l)}[j]=v}, \tag{36}$$

where $\mathbf{1}_{\tilde{M}_f^{(l)}[j]=v}$ is an indicator function which equals 1 if the condition $\tilde{M}_f^{(l)}[j] = v$ satisfies, and equals 0 otherwise. With $B_f^{(l)}$ in (35) and $\tilde{Q}_f^{(l)}[v]$ in (36), we rewrite (32) and (33) as

$$\hat{n}_d^{(l)} = \alpha_d \cdot d^2 \cdot \left(\sum_{v=0}^{\mathcal{K}-1} \tilde{Q}_f^{(l)}[v] \cdot 2^{-B_f^{(l)} - v}\right)^{-1}, \tag{37}$$

$$\hat{n}^{(l)} = \sum_{i=0}^{w-1} \alpha_d \cdot d^2 \cdot \left(\sum_{v=0}^{\mathcal{K}-1} \tilde{Q}^{(l)}[i][v] \cdot 2^{-B^{(l)}[i] - v}\right)^{-1}. \tag{38}$$

The time overhead of (37) is $O(1)$, since $\mathcal{K}$ is a constant. The time cost of (38) is also $O(1)$, because it can be incrementally updated as each packet arrives. Their results can be applied to (16) to obtain $\hat{n}_f^{(l)}$, the estimated flow cardinality by the lth stage, which can be further applied to (19) to obtain $\hat{n}_f$, the estimated cardinality of flow f by all the stages.

Online Sketch Update. We present the Algorithm 2 to update the column of offset registers $\tilde{M}_f^{(l)}$, the base register $B_f^{(l)}$, the column IUU $\tilde{Q}_f^{(l)}$, and the matrix IUU $N^{(l)}$, upon the arrival of a flow element. At the beginning of a measurement period, all of them $\tilde{M}_f^{(l)}$, $B_f^{(l)}$, $\tilde{Q}_f^{(l)}$ and $N^{(l)}$ are reset to zeros.

Algorithm 2 Update Ton-vHLL When a Packet arrives

```
Data: Flow ID f, Element ID e
1  foreach l ∈ [0, s) do
2      x = h^(l)(f ⊕ e), j = ⟨x_1 x_2 ... x_b⟩, q = ⟨x_{b+1} ...⟩
3      if ρ(q) − B_f^(l) ≥ K then
           // ΔB = min_{0≤j<d} M̃_f^(l)[j] has O(d) cost
4          ΔB = min({v | Q̃_f^(l)[v] ≠ 0 ∧ 0 ≤ v < K})
5          if ΔB > 0 then
6              B_f^(l) += ΔB
7              foreach j ∈ [0, d) do M̃_f^(l)[j] −= ΔB
8              foreach v ∈ [ΔB, K) do Q̃_f^(l)[v − ΔB] = Q̃_f^(l)[v]
9      y = min(ρ(q) − B_f^(l), K − 1)
10     if y > M̃_f^(l)[j] then
11         N^(l)[B_f^(l) + M̃_f^(l)[j]] −−, N^(l)[B_f^(l) + y] ++
12         Q̃_f^(l)[M̃_f^(l)[j]] −−,        Q̃_f^(l)[y] ++
13         M̃_f^(l)[j] = y
```

Whenever a packet arrives with a flow ID f and an element e, we use Algorithm 2 to update the multi-stage Ton-vHLL sketch. For each lth stage, we run the code in lines 2-13, which are explained below. At line 2, we apply the lth stage's hash function $h^{(l)}$ to the arrival flow element $f \oplus e$, where $\oplus$ is the concatenation operator. Then, we extract the initial b bits as j, and treat the remaining bits as q. Line 3 computes the offset of $\rho(q)$ relative to the base register $B_f^{(l)}$. If it is larger than or equal to the upper bound $\mathcal{K}$, the offset $\rho(q) - B_f^{(l)}$ exceeds the capacity of the register $\tilde{M}_f^{(l)}$, which is called "overflow".

We handle this overflow event by lines 4-8. Line 4 computes the increment ΔB to the base register $B_f^{(l)}$. There are two calculation methods. The first is proposed by [13], i.e., $\Delta B = min_{0\le j<d}\tilde{M}_f^{(l)}[j]$, which scans the column of offset registers $\tilde{M}_f^{(l)}[j]$ defined in (34) to find the minimum. The time cost of this method is $O(d)$. By contrast, the second method computes ΔB, leveraging the column-wise IUU $\tilde{Q}_f^{(l)}$ defined in (36). Clearly, this method has $O(1)$ time cost. So we use it at Line 4. If $\Delta B > 0$ at line 5, we add it to the base register $B_f^{(l)}$ at line 6, update the offset registers $\tilde{M}_f^{(l)}[j]$ at line 7, and update the column IUU $\tilde{Q}_f^{(l)}$ at line 8. With a high probability, the overflow event will disappear after increasing the base with ΔB. If not, we must round down the offset $\rho(q) - B_f^{(l)}$ to $\mathcal{K}-1$ at line 9, to obtain an offset value y. If $y > \tilde{M}_f^{(l)}[j]$ at line 10, we increase the offset register $\tilde{M}_f^{(l)}[j]$ to y at line 13. Before that, we update the matrix IUU $N^{(l)}$ for total spread at line 11, and update the flow f's column IUU $\tilde{Q}_f^{(l)}$ at line 12.

Sketch Merging. For a data sketch, an important feature is the mergeability, i.e., any two On-vHLL or Ton-vHLL sketches that are collected from different locations can be merged to

capture the global information about per-flow spreads. Assume that the two sketches are configured with the same parameters, including the number of stages s, the number of rows d, the number of columns w, and the hash seeds of $h^{(l)}$ and h.

The merging of two sketches must be performed column by column for each stage, since each column of offset registers has its own base register and IUU. To ease the presentation, we show only the column merging process in Algorithm 3. Suppose we want to merge the second sketch into the first sketch. In the lth stage of the first sketch, let $\tilde{M}_f^{(l)}$ be the column of offset registers chosen by a flow f, which has its own base register $B_f^{(l)}$ and IUU $\tilde{Q}_f^{(l)}$. In the lth stage of the second sketch, we use $\tilde{W}_f^{(l)}$ to denote its column of offset registers chosen by the same flow f, which is associated with the base register $\beta_f^{(l)}$ and the column IUU $\tilde{\Phi}_f^{(l)}$. Therefore, Algorithm 3 is to merge the second column into the first.

Algorithm 3 Merge Second Column of Registers Into first

Data: offset registers $\tilde{M}_f^{(l)}$, base register $B_f^{(l)}$, IUU $\tilde{Q}_f^{(l)}$
offset registers $\tilde{W}_f^{(l)}$, base register $\beta_f^{(l)}$, IUU $\tilde{\Phi}_f^{(l)}$

1 $\Delta B = \min\left(\{v \mid \tilde{Q}_f^{(l)}[v] \neq 0 \ \wedge \ 0 \leq v < \mathcal{K}\}\right)$
2 $\Delta\beta = \min\left(\{v \mid \tilde{\Phi}_f^{(l)}[v] \neq 0 \ \wedge \ 0 \leq v < \mathcal{K}\}\right)$
3 $newB = \max(B_f^{(l)} + \Delta B,\ \beta_f^{(l)} + \Delta\beta)$, $new\tilde{Q} = \mathbf{0}$
4 **foreach** $j \in [0, d)$ **do**
5 $\quad N^{(l)}\left[B_f^{(l)} + \tilde{M}_f^{(l)}[j]\right]--$
6 $\quad \tilde{M}_f^{(l)}[j] = \min\big(\mathcal{K}-1,$
$\quad \max(B_f^{(l)} + \tilde{M}_f^{(l)}[j],\ \beta_f^{(l)} + \tilde{W}_f^{(l)}[j]) - newB\big)$
7 $\quad N^{(l)}\left[newB + \tilde{M}_f^{(l)}[j]\right]++, \quad new\tilde{Q}\left[\tilde{M}_f^{(l)}[j]\right]++$
8 $B_f^{(l)} = newB, \qquad \tilde{Q}_f^{(l)} = new\tilde{Q}$

At lines 1 and 2, we use the IUUs $\tilde{\Phi}_f^{(l)}$ and $\tilde{Q}_f^{(l)}$ to quickly compute the increments to the base registers $\Delta\beta$ and ΔB, respectively. Then, at line 3, we calculate the new base $newB$ after the merging, and initialize the new IUU $new\tilde{Q}$ to zeros. At line 5, before updating the offset register $\tilde{M}_f^{(l)}[j]$, we decrement its corresponding bar in the matrix IUU $N^{(l)}$. At line 6, we use the maximum operator to merge the jth register $\beta_f^{(l)}+\tilde{W}_f^{(l)}[j]$ and the jth register $B_f^{(l)}+\tilde{M}_f^{(l)}[j]$, which is a common practice to merge two HyperLogLog registers. If the merging result minusing the new base $newB$ exceeds the bound $\mathcal{K} = 2^4$, we round it down to $\mathcal{K}-1$, so that a 4-bit offset register $\tilde{M}_f^{(l)}[j]$ can encode the result. After merging the register $\tilde{W}_f^{(l)}[j]$ into $\tilde{M}_f^{(l)}[j]$, at line 7, we correspondingly increment the matrix IUU $N^{(l)}$, and the column IUU $new\tilde{Q}$. After finishing the register merging, at line 8, we update the column base $B_f^{(l)}$, and the column IUU $\tilde{Q}_f^{(l)}$, to finish merging a pair of columns. Note that to finish merging a pair of Ton-vHLL sketches, we must merge all their $s \cdot w$ pairs of columns.

VI. More Accurate Sketch Augmented by Prefilter

There is a common design for vHLL [5], On-vHLL, and Ton-vHLL. The matrix of registers $M^{(l)}$ or $\tilde{M}^{(l)}$ are shared by all flows. As a result, when some flows are hashed to the same registers with a flow f, their elements will become *external noises* to the cardinality estimation of the flow f. Although the cardinality of noises has been removed in (16) by subtracting their expected value $\frac{1}{w}\hat{n}^{(l)}$, the noises inevitably fluctuate when a different set of flows are hashed to share the same registers as f. This noise fluctuation problem becomes more severe for our On-vHLL and Ton-vHLL than vHLL [5], because the memory sharing design shifts from register-level to a much coarser column-level, as shown in Fig. 2 and Fig. 3.

In this section, we separate the top-k *superspreaders*, whose cardinalities are the top-k largest among all flows from other smaller flows, and give each of them an exclusively owned column of HyperLogLog registers, which are free from external noises, therefore appreciably improving their cardinality estimation accuracy. This is a mission impossible in this past, because the traditional sketches for tracking the per-flow cardinality, such as vBitmap [4] and vHLL [5], are too time expensive to query online. By contrast, our On-vHLL and Ton-vHLL sketches can be queried with $O(1)$ time complexity.

As a result, when a packet arrives with a flow ID f, we can check whether the cardinality of f is above a threshold or ranked top-k. If the answer is yes, we can move the flow f from the sketch to the prefilter. This has two benefits: It dramatically improves the estimation accuracy of the top-k superspreaders, since in the prefilter they will have exclusive owned memory for their spread estimation and no interference from other flows. It can also moderately improve the accuracy of small and medium flows. This is because the sketch has much smaller noises, after the top-k superspreaders are swapped into the prefilter, whose future arrival packets will be absorbed by the prefilter, bypassing the sketch. Note that this optimization is orthogonal with the improvement in Section V.

Data Structure. We propose an algorithm named Aton-vHLL (Adaptive tail-cut online virtual HyperLogLog). As shown in Fig. 6, its data structure has two components: a prefilter and a multi-stage Ton-vHLL sketch. Each stage of the sketch is implemented by the base register array $B^{(l)}$, the offset register matrix $\tilde{M}_f^{(l)}$, the column IUU $\tilde{Q}^{(l)}$, and the matrix IUU $N^{(l)}$, similar to the symbols defined in Section V. We will leverage the sketch's online query result to sample and hold the top-k superspreaders into the prefilter. In Fig. 6, we illustrate the basic idea how to maintain the prefilter: When a flow f grows to be a top-k superspreader as its packets arrive, we will swap it from the sketch into the filter. When f is no longer ranked top-k, we will swap it out from the filter to the sketch.

We implement the prefilter by a min-heap structure, so that the flow with the smallest cardinality in the prefilter is always placed at the root. This can help quickly evict the smallest flow that is no longer ranked top-k. More specifically, as shown in Fig. 6, the prefilter consists of a key array K, an index array X, and a register matrix $\tilde{W}$. We elaborate them in following.

- The key array K is to record the 16-bit *fingerprints* of the top-k superspreaders. For an arbitrary flow f, we apply the general hash function h to obtain a 16-bit fingerprint $h(f) \bmod 2^{16}$, which is treated as the shortened ID of the flow. To check whether a flow f exists in the prefilter, we need to scan the key array K to search for a fingerprint that exactly matches $h(f) \bmod 2^{16}$. Let $K[f]$ be this flow searching operation in the key array K. If the flow f does

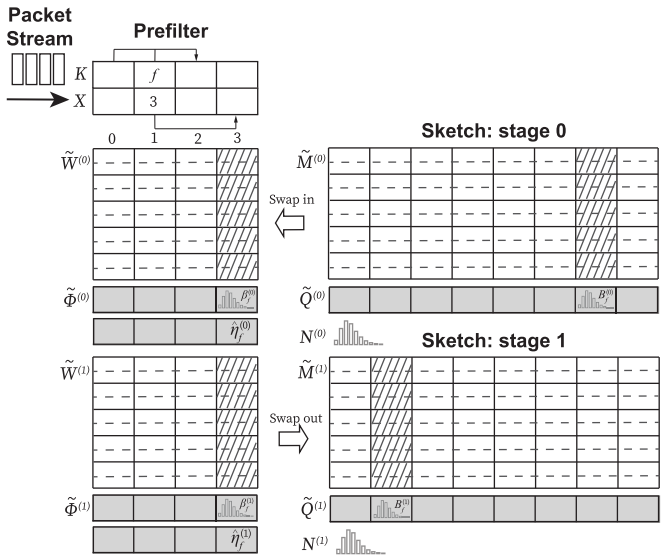


Fig. 6. Aton-vHLL consists of a prefilter and a multi-stage sketch.

not exist, it returns **NIL**; Otherwise, it returns the array index where f is found. Note that we can accelerate the flow searching speed by $\frac{256\text{bits}}{16\text{bits}} = 16$ times or $\frac{512\text{bits}}{16\text{bits}} = 32$ times, if we use the SIMD (Single Instruction Multiple Data) instructions of modern CPUs, i.e., AVX2 or AVX-512 [28].

- The index array X is to translate an index of the key array K to a column index of the register matrix $\tilde{W}$. When the key searching result $K[f] \neq \textbf{NIL}$, we can use $X\big[K[f]\big]$ to find the mapped column index of the flow f in the register matrix $\tilde{W}$. For example, in Fig. 6, the flow f is given a dedicated column of registers with the index 3 in the register matrix $\tilde{W}$. Thanks to the index array, when we adjust the min-heap to keep the smallest flows at the root, we only have to adjust the key array K and the index array X, with no need to relocate the heavy-weighted columns in the matrix $\tilde{W}$.
- The register matrix $\tilde{W}$ is to allocate an exclusively owned column of HLL registers, for each cached flow f in the key array K. As shown in Fig. 6, we partition the value matrix horizontally into equal-size chunks, and each chuck has d HLL registers. Let $\tilde{W}^{(l)}$ be the lth partition of $\tilde{W}$, which is associated with the hash function $h^{(l)}$, the same as the lth stage $\tilde{M}^{(l)}$ of the sketch, $0 \leq l < s$. Let $\tilde{W}_f^{(l)}$ be the HLL estimator allocated to the flow f in the partition $\tilde{W}^{(l)}$. If the flow f exists in the key array with $K[f] \neq \textbf{NIL}$, we can define $\tilde{W}_f^{(l)} = \tilde{W}^{(l)}\big[X\big[K[f]\big]\big]$. Next, we define the jth register of this HLL estimator $\tilde{W}_f^{(l)}[j]$ as follows.

$$\tilde{W}_f^{(l)}[j] = \tilde{W}^{(l)}\Big[X\big[K[f]\big]\Big]^{(l)}[j], \quad \textbf{if } K[f] \neq \textbf{NIL} \tag{39}$$

Of course, it is much better to associate an IUU to each estimator $\tilde{W}_f^{(l)}$, so that we need only $O(1)$ time cost to estimate the cardinality of a flow. These IUUs are shown as gray blocks and denoted as $\tilde{\Phi}_f^{(l)}$ in Fig. 6. Thanks to the accelerated query speed, we can compare the estimated cardinalities of any two flows at low time cost. This helps us to quickly adjust the min-heap, keeping the smallest flow always at the root. Besides IUUs, we will apply the TailCut optimization to the HyperLogLog estimator $\tilde{W}_f^{(l)}$, so that its d registers can be compressed from 5 bits each to 4 bits each for saving memory. In Fig. 6, we show this optimization as horizontal dashed lines cutting through the cells of the register matrices $\tilde{W}^{(l)}$, which splits the bytes into 4-bit offset registers. During the swap-in of a flow f from the sketch to the prefilter, we take snapshots of the total cardinality $\hat{n}^{(l)}$ in the sketch, and copy them to the prefilter, which are denoted by $\hat{\eta}_f^{(l)}$.

Algorithm 4 Query Aton-vHLL as a Packet arrives

Data: Flow ID f
1 **if** $K[f] \neq \textbf{NIL}$ **then** // whether the filter holds f
2 **foreach** $l \in [0, s)$ **do** // query the prefilter
3 $\hat{n}_d^{(l)} = \alpha_d \cdot d^2 \cdot \big(\sum_{v=0}^{\mathcal{K}-1} \tilde{\Phi}_f^{(l)}[v] \cdot 2^{-\beta_f^{(l)}-v}\big)^{-1}$
4 copy the total cardinality $\hat{n}^{(l)}$ from the snapshot $\hat{\eta}_f^{(l)}$
5 estimate the flow cardinality $\hat{n}_f^{(l)}$ by (16)
6 **else**
7 **foreach** $l \in [0, s)$ **do** // query l-th stage of sketch
8 estimate the column cardinality $\hat{n}_d^{(l)}$ by (37)
9 estimate the total cardinality $\hat{n}^{(l)}$ by (38)
10 estimate the flow cardinality $\hat{n}_f^{(l)}$ by (16)
11 **return** $\hat{nn_f}$ estimated by aggregating $\hat{n}_f^{(l)}$, $0 \leq l < s$, by (19)

Online Query. We present the Algorithm 4 to generate a cardinality estimation $\hat{nn_f}$ for an arbitrary flow f. At line 1, we check whether f exists in the key array K of the prefilter. If yes, we query each l-th partition of the prefilter at lines 3-5. Otherwise, we query each lth stage of the Ton-vHLL sketch at lines 8-10. At line 11, we aggregate the flow cardinality estimate $\hat{n}_f^{(l)}$ given by the lth partition or stage, $0 \leq l < s$, to obtain an online estimated result $\hat{nn_f}$.

Online Insertion. We present the Algorithm 5 to insert the arrival packet $\langle f, e\rangle$ into the prefilter and the sketch. At line 1, we check whether the flow f exists in the key array K of the prefilter. If yes, at lines 2-12, for each lth partition of the prefilter, we update the column allocated to the flow f. For example, in Fig. 6, the column 3 is given to the flow f, as indicated by the arrays K and X. We update the base register $\beta_f^{(l)}$ at line 7, the column of registers $\tilde{W}_f^{(l)}$ at lines 8&12, and the column IUU $\tilde{\Phi}_f^{(l)}$ at lines 9&12. Since this part is similar to Algorithm 2, we do not explain it in details. Note that we do not need to update the total cardinality $\hat{\eta}_f^{(l)}$, which is already determined when the flow f is swapped out of the sketch into the filter. Since the spread of flow f is increased, at line 13, we sift-up the flow f to restore the min-heap property. At line 14, we end the function execution to bypass the sketch updating.

When the flow f does not exist in the key array K, at line 15, we insert the packet $\langle f, e\rangle$ into the Ton-vHLL sketch by Algorithm 2. After updating the sketch, at line 16, we use Algorithm 4 to online query the sketch for the cardinality of the flow f at $O(1)$ time cost. If the estimated cardinality $\hat{nn_f}$ is smaller than a predefined threshold, then we stop the execution at line 17 to avoid unnecessary swap in and out. At line 18, if the prefilter is full already, then we retrieve the flow f' at the

Algorithm 5 Update Aton-vHLL as a Packet arrives

Data: Flow ID f, Element ID e
1 **if** $K[f] \neq$ **NIL then**
2 **foreach** $l \in [0, s)$ **do** // update the flow in filter
3 $x = h^{(l)}(h(f \oplus e))$, $j = \langle x_1 \dots x_b \rangle$, $q = \langle x_{b+1} \dots \rangle$
4 **if** $\rho(q) - \beta_f^{(l)} \geq \mathcal{K}$ **then**
5 $\Delta\beta = \min\left(\{v \mid \tilde{\Phi}_f^{(l)}[v] \neq 0 \ \wedge \ 0 \leq v < \mathcal{K}\}\right)$
6 **if** $\Delta\beta > 0$ **then**
7 $\beta_f^{(l)} \mathrel{+}= \Delta\beta$
8 **foreach** $j \in [0, d)$ **do** $\tilde{W}_f^{(l)}[j] \mathrel{-}= \Delta\beta$
9 **foreach** $v \in [\Delta\beta, \mathcal{K})$ **do** $\tilde{\Phi}_f^{(l)}[v - \Delta\beta] = \tilde{\Phi}_f^{(l)}[v]$
10 $y = \min(\rho(q) - \beta_f^{(l)}, \ \mathcal{K} - 1)$
11 **if** $y > \tilde{W}_f^{(l)}[j]$ **then** // modify the filter
12 $\tilde{\Phi}_f^{(l)}\left[\tilde{W}_f^{(l)}[j]\right] --$, $\tilde{\Phi}_f^{(l)}[y] ++$, $\tilde{W}_f^{(l)}[j] = y$
13 **if** *filter has been modified* **then** sift down f in min-heap
14 **return** // bypass the sketch and return directly
15 insert $\langle f, e \rangle$ into Ton-vHLL sketch by Algorithm 2
16 estimate the flow f's cardinality $\hat{n_f}$ by Algorithm 4
17 **if** $\hat{n_f} \leq c \cdot \hat{n} \,/\, w$ **then return** // set a threshold
18 **if** *prefilter is full* **then**
19 $\hat{n_{f'}} =$ minimum spread in the prefilter with flow ID f'
20 **if** $\hat{n_f} \leq \hat{n_{f'}}$ **then return** // not a superspreader
21 **foreach** $l \in [0, s)$ **do** // swap out the flow f'
22 for f', use Algorithm 3 to merge its column of prefilter into its column of Ton-vHLL sketch, i.e., $\tilde{W}_{f'}^{(l)}$, $\beta_{f'}^{(l)}$, $\tilde{\Phi}_{f'}^{(l)} \Rightarrow \tilde{M}_{f'}^{(l)}$, $B_{f'}^{(l)}$, $\tilde{Q}_{f'}^{(l)}$
23 replace flow ID f' by the new superspreader f in K
24 **else**
25 append flow ID f to the end of key array K
26 find an empty column in $\tilde{W}$, and allocate it to f
27 **foreach** $l \in [0, s)$ **do** // swap in new superspreader f
28 $\hat{\eta}_f^{(l)} = \hat{n}^{(l)}$ // take a snapshot of total spread
29 $\tilde{W}_f^{(l)} = \tilde{M}_f^{(l)}$, $\tilde{\Phi}_f^{(l)} = \tilde{Q}_f^{(l)}$, $\beta_f^{(l)} = B_f^{(l)}$ // copy
30 adjust the prefilter to restore the min-heap property

root of the min-heap by line 19, whose estimated cardinality $\hat{n_{f'}}$ is the smallest in the prefilter. If the cardinality $\hat{n_f}$ of the arrival flow f is no larger than $\hat{n_{f'}}$, then at line 20, we can stop the because the flow f is not ranked top-k. Otherwise, the flow f surpasses the flow f', and becomes the new superspreader.

To make room for the new superspreader f, we use line 21 to swap out f' back to the sketch. Note that we do not know the flow ID of the swap-out flow f', and we only know its 32-bit fingerprint $h(f')$ stored in the prefilter's key array K. But still we can use its fingerprint to locate its column in the sketch, since in (13) we compute the mapped column by applying the stage l's unique hash function $h^{(l)}$ to the fingerprint, i.e., $h^{(l)}\big(h(f')\big) \bmod w$. Then, we can reuse Algorithm 3 to merge the column of prefilter of f' back into its column of Ton-vHLL sketch. At line 23, we overwrite the flow f' by the new superspreader f in the key array K, such that f can directly occupy the register column in $\tilde{W}$ previously used by f'.

Lines 25-26 are to handle the case that the prefilter is not full. Lines 27-29 swap in the new superspreader f from the sketch to the prefilter, through directly copying. Finally, the line 27 adjusts the prefilter to restore the min-heap property.

VII. Experimental Evaluation

In this section, we empirically evaluate the performance of On-vHLL, Ton-vHLL and Aton-vHLL sketches, including packet processing throughput, the number of memory accesses and hashes per packet, flow spread estimation error, and superspreader identification error. We also evaluate the impact of parameters settings on proposed sketches.

A. Experiment Settings

We evaluate the proposed algorithms by trace-driven simulations. Our network traces are from CAIDA, each of which contains 1 billion packets [31]. Our paper mainly considers "online query" scenario that a sketch is both updated and queried for the spread of flow f, rather than "offline query" scenario that a sketch is only updated as a packet $\langle f, e \rangle$ arrives.

vHLL [5] is an offline-query solution which uses sublinear memory to estimate the per-flow spreads with excellent accuracy, we choose it for accuracy comparison with our sketches. A more recent sketch is rSkt [14], which has two variants rSkt1 and rSkt2. The former can support online query and has similar accuracy with rSkt. The latter provides better accuracy than rSkt, but has high query time cost proportional to the number of counting units. We also compare our sketches with another recent work named AROMA [15], which allocates an array of slots to sample the $\langle$flowID, elementID$\rangle$ pairs uniformly by the MinHash technique. AROMA can be modified to an online-query variant named AROMA+, which must allocate another $\langle$flowID, number of sampled pairs$\rangle$ hash table to record all the flows whose pairs have been sampled in the MinHash table.

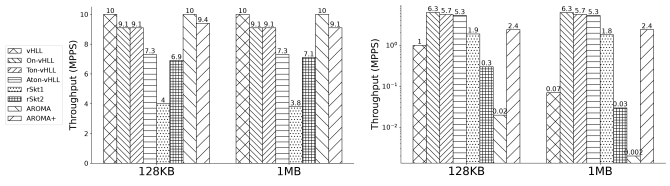
When comparing the proposed sketches with vHLL, rSkt1, rSkt2, AROMA and AROMA+, we evaluate the following metrics. The first is update and query throughput, i.e., the number of update or query operations per second, evaluated on Intel Xeon Sliver 4214. The second is the average number of memory accesses and hashes, needed by sketch update and query. The third is the estimation error of per-flow cardinality. We quantify the estimation error by relative bias, and root-mean-square relative error (RMSRE), which are defined as

$$\text{relative bias}(\hat{X}) = \tfrac{1}{r} \textstyle\sum_{1 \leq i \leq r} \frac{\hat{X}_i - X_i}{X_i}, \tag{40}$$

$$\text{RMSRE}(\hat{X}) = \sqrt{\tfrac{1}{r} \textstyle\sum_{1 \leq i \leq r} \frac{(\hat{X}_i - X_i)^2}{X_i^2}}, \tag{41}$$

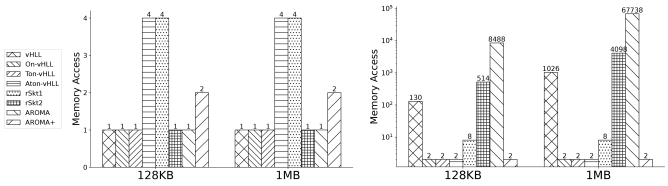
where X_i is the actual value of a flow's cardinality, $\hat{X}_i$ is its estimated cardinality, and r is the number of trials we run a same experiment. The fourth metric is the identification error of top-k superspreaders, quantified by false negative rate (FNR) $\frac{|D \setminus \hat{D}|}{|D|}$, where D is the true set of superspreaders, and $\hat{D}$ is the set of flow IDs reported as superspreaders.

These sketches will be given the same amount of memory in experiments. The memory cost of vHLL is $w \cdot 8d$ bits, On-vHLL with multiple stages is $w \cdot s \cdot 8d$ bits, Ton-vHLL without the prefilter is $w \cdot s \cdot 4d$ bits, Aton-vHLL is $(k + w) \cdot s \cdot 4d$ bits,



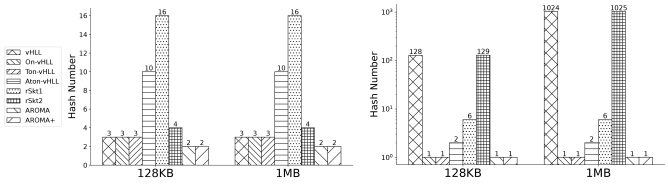
(a) Update (normal scale) (b) Query (log scale)

Fig. 7. Compare vHLL, On-vHLL, Ton-vHLL, Aton-vHLL, rSkt1, rSkt2, AROMA and AROMA+ on (a) Update and (b) Query throughput, with 128KB ($d = 128$, AROMA $slots = 10922$, AROMA+ $slots = 6553$) or 1MB ($d = 1024$, AROMA $slots = 87381$, AROMA+ $slots = 52428$) memory.



(a) Update (normal scale) (b) Query (log scale)

Fig. 8. Compare number of memory accesses of vHLL, On-vHLL, Ton-vHLL, Aton-vHLL, rSkt1, rSkt2, AROMA and AROMA+ with the same 128KB and 1MB memory, when (a) updating and (b) querying.



(a) Update (normal scale) (b) Query (log scale)

Fig. 9. Compare number of hashes of vHLL, On-vHLL, Ton-vHLL, Aton-vHLL, rSkt1, rSkt2, AROMA and AROMA+ with the same 128KB and 1MB memory, when (a) updating and (b) querying.

if we ignore the relatively small memory cost of IUUs. The memory cost of rSkt1 and rSkt2 is $w \cdot s \cdot 8d$ bits, where $s = 2$, since both of them consist of a primary HLL estimator and a complement HLL estimator. The memory cost of AROMA is $96 \cdot 8 \cdot slots$ bits. The memory cost of AROMA+ is $160 \cdot 8 \cdot slots$ bits, since it enhances AROMA with another HashMap for online query, which occupies extra $64 \cdot 8 \cdot slots$ bits.

B. Throughput, Number of Memory Accesses and Hashes

In this subsection, we give the same amount of memory to vHLL, rSkt1, rSkt2, AROMA, AROMA+, and our proposed sketches. Then, we compare their performance on the packet processing throughput, the number of memory accesses per packet, and the number of hash function calls per packet.

In Fig. 7, we evaluate the update and query throughput in the unit of mega packets per second (MPPS). Fig. 7a shows that the update throughput of these sketches does not decline as the estimator size grows. Our sketches can be updated faster than rSkt1 and rSkt2, but slightly slower than vHLL, AROMA and AROMA+. Fig. 7a also shows that Aton-vHLL has 20% lower query throughput than On-vHLL and Ton-vHLL, due to the additional CPU cycles according to Algorithm 5. In the online query scenario, query throughput is more important than update throughput, since query is much slower than update and is the bottleneck of packet processing. Fig. 7b shows that On-vHLL, Ton-vHLL, Aton-vHLL, rSkt1 and AROMA+ (or vHLL, rSkt2 and AROMA) have their query throughput to stay the same (or reduce linearly), when the memory size increases with virtual estimator size d. Therefore, vHLL, rSkt2 and AROMA will be regarded as "offline query" sketches. Fig. 7b also shows that our On-vHLL, Ton-vHLL, Aton-vHLL have 2 to 3 times higher query throughput than rSkt1 and AROMA+. This is because our sketches are based on the multi-stage design shown in Fig. 7b, in which the stages can execute parallelly as multiple pipelines in data plane.

To figure out the causes for the disparity in the throughput of these sketches, we break down the packet processing time cost into the number of memory accesses and hash function calls per packet, which are evaluated separately in Fig. 8 and Fig. 9.

In Fig. 8, we evaluate the number of memory accesses per packet when updating or querying. Fig. 8a shows that all the sketches under comparison need less than 5 memory accesses when updating, regardless of the estimator size. Aton-vHLL needs more memory accesses when updating, since it occasionally swaps in/out flows, and adjusts the prefilter to restore the min-heap property. Whereas, situation becomes different when querying. Fig. 8b shows that vHLL, rSkt2 and AROMA need to access $O(d)$ memory units per query, and thus their number of per-packet memory accesses can be hundreds or even thousands, depending on the configuration of virtual estimator size d. By contrast, this number decreases to 2, when On-vHLL, Ton-vHLL or Aton-vHLL are applied. Such dramatic reduction comes from the caching of the intermediate results in IUUs. Fig. 8b also shows that AROMA+ and rSkt1 can also be online queried. Their numbers of memory accesses are 2 and 8, respectively, for the query operation.

In Fig. 9, we evaluate the number of hash function calls per packet when updating or querying. Fig. 9a shows that Aton-vHLL and rSkt1 require more than 10 times of hashes, which makes their updating throughput slightly slower than other sketches. Fig. 9b shows that the sketches strongly differ, with respect to their number of hashes for the query operation. Our sketches need no more than 2 times of hashes, whereas vHLL and rSkt2 need over 1000 times when querying, which will dramatically slow down their packet processing throughput.

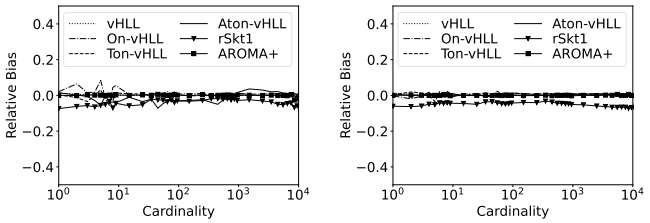
Since this paper focuses on online per-flow spread estimation, in the rest of the evaluation, we will focus on comparing with rSkt1 and AROMA+, which support online query. As our proposed sketches can be regarded variants of vHLL to support online query, we will also compare with vHLL, in order to evaluate the degree of accuracy loss.

C. Flow Spread Estimation Error

We evaluate the relative bias and the RMSRE of per-flow spread estimation, when all the solutions are given the same amount of memory, which is either 128KB (i.e., 0.1 bit per flow) or 1MB (i.e., 1 bit per flow).

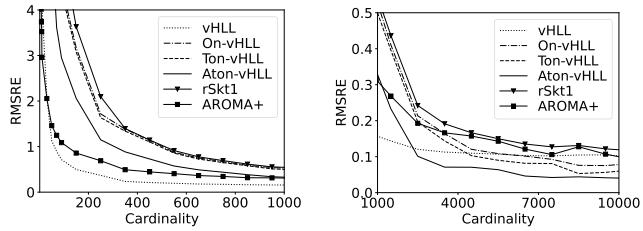
In Fig. 10, we show that our sketches are unbiased at any flow spread value, no matter whether the memory is 128KB or 1MB. This is consistent with the theoretical analysis in Eq. (20). We find that vHLL and AROMA+ are also unbiased, but rSkt1 has -5% bias due to hash collision.

In Fig. 11, we illustrate the RMSRE of all spread values, when all the sketches are given the same 128KB memory. It shows that vHLL has excellent estimation accuracy. On-vHLL trades its online query capability for more than 100% higher error for small flows whose spreads range from 1 to



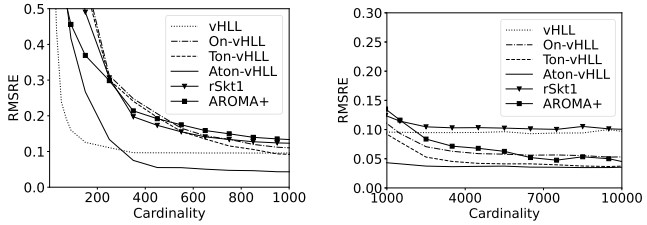
(a) 128KB (b) 1MB

Fig. 10. Compare the relative bias of per-flow spread estimation under the same (a) 128KB memory, or (b) 1MB memory.



(a) Flows with spread<1000 (b) Flows with spread≥1000

Fig. 11. Compare per-flow spread estimation error under 128KB memory.



(a) Flows with spread<1000 (b) Flows with spread≥1000

Fig. 12. Compare per-flow spread estimation error under 1MB memory.

5000 due to more intense hash collisions. However, the estimation error of On-vHLL is always lower than rSkt1 and is lower than AROMA+ for larger flows, thanks to our multi-stage design in Fig. 4. In order to further improve estimation accuracy on small flows, we need a larger number of registers swd, according to (20). Therefore, TailCut technique is proposed to compress a register from one byte into 4 bits. Smaller register size means more registers can be allocated for the sketch, while the total memory remains the same. Therefore, we can double the parameter d of Ton-vHLL and Aton-vHLL, such that their virtual estimators are twice the size of other sketches. As shown in Fig. 11, Ton-vHLL reduces estimation error by 30% compared to On-vHLL. With the addition of the pre-filter, Aton-vHLL dramatically improves the estimation accuracy of flow spreads. It always outperforms On-vHLL, Ton-vHLL and rSkt1 in Fig. 11b. Its estimation error is over 50% lower than Ton-vHLL and AROMA+ for flows whose spreads are above 1000. Aton-vHLL also attains better accuracy than vHLL when flow spreads are larger than 2500.

In Fig. 12, our proposed sketches still perform well under 1MB memory. For small spreads smaller than 1000 in (a), the accuracy of On-vHLL and Ton-vHLL is similar to rSkt1, but for large flows in (b), their accuracy are better than rSkt1 by 50%. Aton-vHLL has smaller errors than AROMA+ and vHLL when the spread is larger than 100 and 300, respectively. After being modified into the online-query version, AROMA+ suffers a relatively high cost in terms of memory consumption. We need to pre-allocate at least $\alpha \cdot M$ slots and each slot in the HashMap requires 64 bits, which results in a 66.7% increase in memory usage and worse accuracy under the same memory. As we know, the average flow spread in CAIDA is about 1.8, which means that a significant proportion of slots of AROMA+ will be filled with small flows. Therefore, AROMA+ does not perform well when estimating medium and large flows. By contrast, Aton-vHLL not only ensures the accuracy of the estimation of superspreaders in the prefilter, but also improves the accuracy of the estimation of small and medium flows compared to On-vHLL and Ton-vHLL, since the large flows will collide with the smaller flows less frequently.

D. Superspreaders Identification Error

In this subsection, we compare different solutions on the identification error of the top-k superspreaders, which is measured by false negative rate (FNR) in Fig. 13 and Fig. 14.

Fig. 13 shows that our three proposed sketches outperform both rSkt1 and AROMA+ in detecting superspreaders under 128KB of memory. Aton-vHLL have comparable ability with vHLL to detect the top-k superspreaders, slightly better than On-vHLL and Ton-vHLL. Aton-vHLL's FNR for top-500 superspreaders is 65% lower than AROMA+ and is 33% lower than rSkt1. Since the size of Aton-vHLL's prefilter is only 64, it cannot fit all the top-1000 superspreaders, so this can prove that Aton-vHLL's prefilter not only improves the accuracy of large flows, but also reduces the noise in sketch, thus improving the accuracy of small and medium flows.

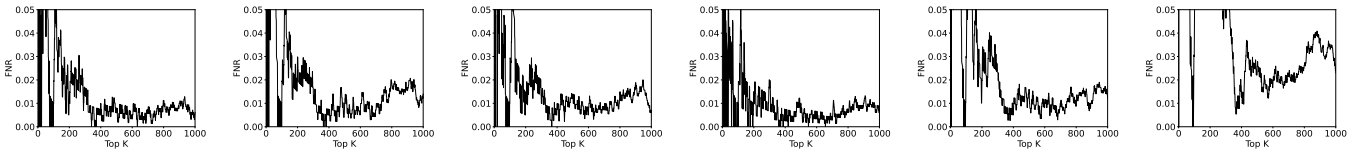
When given more memory, like 1MB in Fig. 14, FNRs of all the sketches become lower than 2% when $k > 200$. In Fig. 14, our On-vHLL, Ton-vHLL and Aton-vHLL outperform rSkt1 and AROMA+. Aton-vHLL in (d) has the lowest FNRs of superspreader detection, comparing with vHLL in (a), On-vHLL in (b), Ton-vHLL in (c), rSkt1 in (e) and AROMA+ in (f).

E. Impact of Parameters on Spread Estimation Accuracy

In this subsection, we evaluate the impact of parameters on the accuracy of our sketches, including the number of stages s, the size of prefilter k, the number of columns in sketch w, and the number of rows in both sketch and prefilter d.

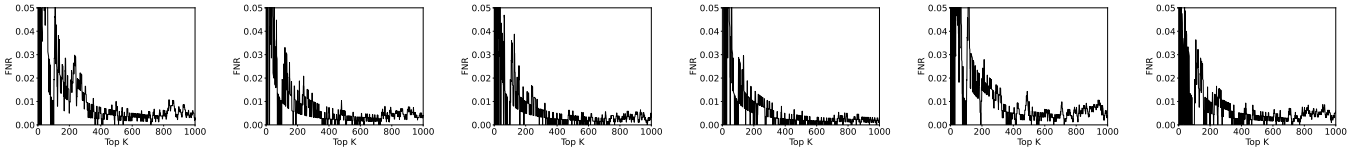
1) Impact of k and w: The prefilter improvew accuracy by allocating dedicated memory to the top-k superspreaders. If k is configured larger, more flows can be held in prefilter and fewer flows will be squeezed in sketch, resulting in smaller noise and better accuracy. Meanwhile, if w is configured larger, flows will be less disturbed by hash collision. However, the sum of k and w is a fixed value due to memory constraint. So we try to figure out the optimal setting of k, that is to say, how much memory partitioned off from sketch is the most worthy.

In the first experiment, we configure $s = 4$, $k + w = 2048$, $d = 256$ under 1MB memory and evaluate the impact of k and w on accuracy. Fig. 15a shows that Aton-vHLL is unbiased for all spread values, whatever k and w is. Fig. 16a and Fig. 16b reveals that prefilter helps to reduce RMSRE dramatically, not only the superspreaders, but also the flows that are not accommodated in prefilter. For a flow whose spread is 200 outside the prefilter, RMSRE decreases 24.5% due to the 64-column prefilter. When k is configured larger, 512 for example, Aton-vHLL can achieve 47.4% lower



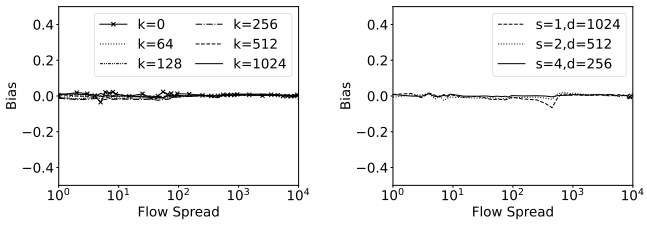
(a) vHLL (b) On-vHLL (c) Ton-vHLL (d) Aton-vHLL (e) rSkt1 (f) AROMA+

Fig. 13. Compare the false negative rates of different algorithms for identifying the top-k superspreaders, when they are given the same 128KB memory.



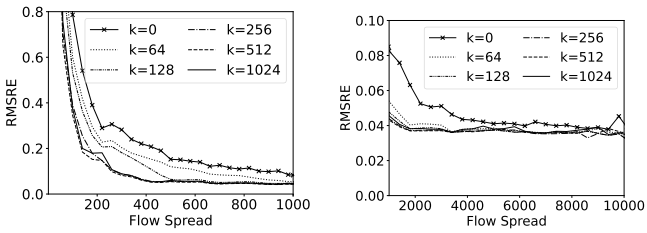
(a) vHLL (b) On-vHLL (c) Ton-vHLL (d) Aton-vHLL (e) rSkt1 (f) AROMA+

Fig. 14. Compare the false negative rates of different algorithms for identifying the top-k superspreaders, when they are given the same 1MB memory.



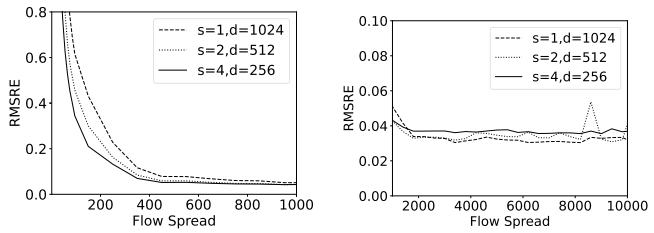
(a) different k and w settings (b) different s and d settings

Fig. 15. Evaluate the relative bias of per-flow spread estimation under 1MB memory, in (a) different k and w settings, or (b) different s and d settings.



(a) Flows with spread$<$1000 (b) Flows with spread$\geq$1000

Fig. 16. Evaluate the RMSRE of per-flow spread estimation under the same 1MB memory, but assuming the different k and w settings.



(a) Flows with spread$<$1000 (b) Flows with spread$\geq$1000

Fig. 17. Evaluate the RMSRE of per-flow spread estimation under the same 1MB memory, but assuming the different s and d settings.

RMSRE comparing with none-prefilter. However, the marginal benefit of increasing k value decreases, and diminishes when the size of prefilter reaches a certain bound. Also, oversized prefilter may crowd sketch's memory space, resulting in performance degradation when estimating small and medium flows. As a result, the accuracy when $k = 512$ outperforms that when $k = 1024$. Afterwards, we choose $k = 512$ and $w = 1792$ by default for Aton-vHLL under 1MB memory. We conducted similar experiments with 128KB of memory and found the best overall performance when $k = 64$ and set it as the default parameter.

2) Impact of s and d: According to (20), larger s helps mitigate the impact of hash collision among flows, and larger d improves the accuracy of a virtual estimator. However, keeping both parameters large will be infeasible due to memory limit.

In the second experiment, we configure $k = 512$, $w = 1792$, $s \cdot d = 1024$, and evaluate the impact of s and d on estimation error. Fig. 15b shows that Aton-vHLL is unbiased under every pair of s and d setting. In Fig. 17a and Fig. 17b, when s is configured larger, Aton-vHLL attains better accuracy for small and medium flows under 1000. When d is configured larger, Aton-vHLL attains better accuracy for large flows, since large flows are less affected by noise but sensitive to HLL estimator size. Besides, increasing s will not affect the throughput, because the multi-stage Aton-vHLL can be implemented in multi-core or multi-pipeline systems, as explained in Section IV-E. We conducted experiments with 128KB of memory, and get similar result. To strike a balance between accuracy of small flows and large flows, we assign the number of stages $s = 4$ by default.

VIII. Conclusion

For network traffic measurement, it is an important problem to estimate the per-flow cardinality, i.e., approximately count the number of *unique* elements in each flow, which demands the ability to filter *duplicated* elements. If each flow cardinality can be estimated on a per-packet basis with low time complexity, then the super-spreading flows with a significant number of unique elements can be tracked by an online manner. For this new problem, we propose three algorithms named On-vHLL, Ton-vHLL and Aton-vHLL, whose time cost of online query are strictly $O(1)$. We adopt three optimization techniques: incremental update units, HLL register compression by TailCut, and sample-and-hold the top-k superspreaders in a prefilter, where they are given the exclusively owned HyperLogLog estimators for better accuracy. We evaluate the throughput and accuracy improvements that can be brought about by these techniques, using CAIDA traffic traces. Furthermore, we show that, upon the arrival of each packet, our On-vHLL, Ton-vHLL and Aton-vHLL sketches need about 5 memory accesses for the sketch updating and querying combined. Our Aton-vHLL can not only attain this high query speed, but also provide good estimation accuracy of per-flow spread comparable to vHLL.

REFERENCES

[1] P. Lieven and B. Scheuermann, "High-speed per-flow traffic measurement with probabilistic multiplicity counting," in *Proc. IEEE INFOCOM*, Mar. 2010, pp. 1–9.

[2] T. Li, S. Chen, and Y. Ling, "Fast and compact per-flow traffic measurement through randomized counter sharing," in *Proc. IEEE INFOCOM*, Apr. 2011, pp. 1799–1807.

[3] Y. Zhou, Y. Zhou, S. Chen, and Y. Zhang, "Highly compact virtual active counters for per-flow traffic measurement," in *Proc. IEEE Conf. Comput. Commun.*, Apr. 2018, pp. 1–9.

[4] M. Yoon, T. Li, S. Chen, and J.-K. Peir, "Fit a spread estimator in small memory," in *Proc. IEEE INFOCOM*, Apr. 2009, pp. 504–512.

[5] Q. Xiao, S. Chen, M. Chen, and Y. Ling, "Hyper-compact virtual estimators for big network data based on register sharing," in *Proc. ACM SIGMETRICS Int. Conf. Meas. Modeling Comput. Syst.*, Jun. 2015, pp. 417–428.

[6] M. Alizadeh et al., "CONGA: Distributed congestion-aware load balancing for datacenters," in *Proc. ACM Conf. SIGCOMM*, Aug. 2014, pp. 503–514.

[7] Y. Zhu et al., "Packet-level telemetry in large datacenter networks," in *Proc. ACM Conf. Special Interest Group Data Commun.*, Aug. 2015, pp. 479–491.

[8] A. Lakhina, M. Crovella, and C. Diot, "Characterization of network-wide anomalies in traffic flows," in *Proc. 4th ACM SIGCOMM Conf. Internet Meas.*, Oct. 2004, pp. 201–206.

[9] P. Flajolet, E. Fusy, and O. Gandouet, "HyperLogLog: The analysis of a near-optimal cardinality estimation algorithm," in *Proc. AOFA*, 2007, pp. 137–156.

[10] K.-Y. Whang, B. T. Vander-Zanden, and H. M. Taylor, "A linear-time probabilistic counting algorithm for database applications," *ACM Trans. Database Syst.*, vol. 15, no. 2, pp. 208–229, Jun. 1990.

[11] G. Cormode and S. Muthukrishnan, "An improved data stream summary: The count-min sketch and its applications," *J. Algorithms*, vol. 55, no. 1, pp. 58–75, Apr. 2005.

[12] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, "Heavy-hitter detection entirely in the data plane," in *Proc. Symp. SDN Res.*, Apr. 2017, pp. 164–176.

[13] Q. Xiao, S. Chen, Y. Zhou, and J. Luo, "Estimating cardinality for arbitrarily large data stream with improved memory efficiency," *IEEE/ACM Trans. Netw.*, vol. 28, no. 2, pp. 433–446, Apr. 2020.

[14] H. Wang, C. Ma, O. O. Odegbile, S. Chen, and J.-K. Peir, "Randomized error removal for online spread estimation in high-speed networks," *IEEE/ACM Trans. Netw.*, vol. 31, no. 2, pp. 558–573, Apr. 2023.

[15] R. B. Basat, X. Chen, G. Einziger, L. Shir, D. Raz, and M. Yu, "Routing oblivious measurement analytics," in *Proc. IFIP Netw.*, 2020, pp. 449–457.

[16] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," in *Proc. EATCS ICALP*, 2002, pp. 693–703.

[17] M. Durand and P. Flajolet, "Loglog counting of large cardinalities," in *Proc. Eur. Symposia Algorithms*, 2003, pp. 605–617.

[18] Y. Lu and B. Prabhakar, "Robust counting via counter braids: An error-resilient network measurement architecture," in *Proc. IEEE INFOCOM*, Apr. 2009, pp. 522–530.

[19] J. Li et al., "WavingSketch: An unbiased and generic sketch for finding top-k items in data streams," in *Proc. 26th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Aug. 2020, pp. 1574–1584.

[20] X. Jing, Z. Yan, H. Han, and W. Pedrycz, "ExtendedSketch: Fusing network traffic for super host identification with a memory efficient sketch," *IEEE Trans. Dependable Secure Comput.*, vol. 19, no. 6, pp. 3913–3924, Nov. 2022.

[21] H. Wang, C. Ma, S. Chen, and Y. Wang, "Fast and accurate cardinality estimation by self-morphing bitmaps," *IEEE/ACM Trans. Netw.*, vol. 30, no. 4, pp. 1674–1688, Aug. 2022.

[22] G. S. Manku and R. Motwani, "Approximate frequency counts over data streams," in *Proc. 28th Int. Conf. Very Large Databases*, 2002, pp. 346–357.

[23] A. Metwally, D. Agrawal, and A. E. Abbadi, "Efficient computation of frequent and top-k elements in data streams," in *Proc. ICDT*, 2005, pp. 398–412.

[24] S. Venkataraman, D. Song, P. B. Gibbons, and A. Blum, "New streaming algorithms for fast detection of superspreaders," in *Proc. NDSS*, 2005, pp. 1–16.

[25] N. Kamiyama, T. Mori, and R. Kawahara, "Simple and adaptive identification of superspreaders by flow sampling," in *Proc. IEEE INFOCOM*, May 2007, pp. 2481–2485.

[26] J. Cao, Y. Jin, A. Chen, T. Bu, and Z.-L. Zhang, "Identifying high cardinality internet hosts," in *Proc. IEEE INFOCOM*, Apr. 2009, pp. 810–818.

[27] Y.-E. Sun, H. Huang, C. Ma, S. Chen, Y. Du, and Q. Xiao, "Online spread estimation with non-duplicate sampling," in *Proc. IEEE Conf. Comput. Commun.*, Jul. 2020, pp. 2440–2448.

[28] P. Roy, A. Khan, and G. Alonso, "Augmented sketch: Faster and more accurate stream processing," in *Proc. Int. Conf. Manage. Data*, Jun. 2016, pp. 1449–1463.

[29] *Paper Extended Version With Appendix*. Accessed: Apr. 18, 2023. [Online]. Available: https://www.dropbox.com/s/q865fahvnxeupt9/paper_extended.pdf?dl=0

[30] P. Bosshart et al., "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, pp. 87–95, Jul. 2014.

[31] (Jan. 17). *CAIDA UCSD Anonymized 2017 Internet Traces*. Accessed: Dec. 31, 2020. [Online]. Available: http://www.caida.org/data/passive/passive_2017_dataset.xml

Qingjun Xiao (Member, IEEE) received the B.Sc. degree from the Department of Computer Science, Nanjing University of Posts and Telecommunications, China, in 2003, the M.Sc. degree from the Department of Computer Science, Shanghai Jiao Tong University, China, in 2007, and the Ph.D. degree from the Department of Computer Science, The Hong Kong Polytechnic University, in 2011. After graduation, he joined Georgia State University and the University of Florida, and worked for three years combined as a Post-Doctoral Researcher. He is currently working as an Associate Professor with Southeast University, China. His research interests include protocol and algorithm design in network traffic measurement, network security, and wireless sensor networks. He is a member of ACM.

Yuexiao Cai received the B.Sc. degree from the Department of Computer Science, Nanjing University of Aeronautics and Astronautics, China, in 2021. He is currently pursuing the M.Sc. degree in cyber security with Southeast University, China. His advisor is Prof. Q. Xiao. His research interests include network traffic measurement, programmable networks, and network security.

Yunpeng Cao received the B.Sc. degree from the Department of Communication Engineering, Yunnan University, China, in 2020. He is currently pursuing the M.Sc. degree in cyber security with Southeast University, China. His advisor is Prof. Q. Xiao. His research interests include stream algorithms, traffic measurement, and network security.

Shigang Chen (Fellow, IEEE) received the B.S. degree in computer science from the University of Science and Technology of China, China, in 1993, and the M.S. and Ph.D. degrees in computer science from the University of Illinois at Urbana–Champaign in 1996 and 1999, respectively.

After graduation, he had worked with Cisco Systems for three years before joining the University of Florida in 2002. He is currently a Professor with the Department of Computer and Information Science and Engineering, University of Florida. He has published more than 200 peer-reviewed journal/conference papers. He holds 13 U.S. patents, and many of them were used in software products. His research interests include computer networks, the Internet of Things, big data, cybersecurity, data privacy, edge-cloud computing, intelligent cyber-transportation systems, and wireless systems.