Memory-Efficient and Flexible Detection of Heavy Hitters in High-Speed Networks

HE HUANG, Soochow University, China
JIAKUN YU, Soochow University, China
YANG DU*, Soochow University, China
JIA LIU, Nanjing University, China
HAIPENG DAI, Nanjing University, China
YU-E SUN*, Soochow University, China

Heavy-hitter detection is a fundamental task in network traffic measurement and security. Existing work faces the dilemma of suffering dynamic and imbalanced traffic characteristics or lowering the detection efficiency and flexibility. In this paper, we propose a flexible sketch called SwitchSketch that embraces dynamic and skewed traffic for efficient and accurate heavy-hitter detection. The key idea of SwitchSketch is allowing the sketch to dynamically switch among different modes and take full use of each bit of the memory. We present an encoding-based switching scheme together with a flexible bucket structure to jointly achieve this goal by using a combination of design features, including variable-length cells, shrunk counters, embedded metadata, and switchable modes. We further implement SwitchSketch on the NetFPGA-1G-CML board. Experimental results based on real Internet traces show that SwitchSketch achieves a high F_β-Score of threshold-t detection (consistently higher than 0.938) and over 99% precision rate of top-k detection under a tight memory size (*e.g.*, 100KB). Besides, it outperforms the state-of-the-art by reducing the ARE by 30.77%∼99.96%. All related implementations are open-sourced[1].

CCS Concepts: • **Networks** → **Network measurement**; **Network monitoring**; • **Theory of computation** → **Sketching and sampling**; • **Information systems** → **Data stream mining**.

Additional Key Words and Phrases: Sketch, Heavy hitter detection, Network traffic measurement

ACM Reference Format:
He Huang, Jiakun Yu, Yang Du, Jia Liu, Haipeng Dai, and Yu-E Sun. 2023. Memory-Efficient and Flexible Detection of Heavy Hitters in High-Speed Networks. *Proc. ACM Manag. Data* 1, N3 (SIGMOD), Article 214 (September 2023), 24 pages. https://doi.org/10.1145/3617334

1 INTRODUCTION

Heavy hitters are extremely large continuous flows measured over a network link, which are not numerous but occupy a significant share of the total bandwidth over a period of time. Detecting

*Co-corresponding Authors.
[1]Source Code. https://github.com/duyang92/switch-sketch-paper/

Authors' addresses: He Huang, Soochow University, Suzhou, Jiangsu, China, huangh@suda.edu.cn; Jiakun Yu, Soochow University, Suzhou, Jiangsu, China, yujiakun22@gmail.com; Yang Du, Soochow University, Suzhou, Jiangsu, China, duyang@suda.edu.cn; Jia Liu, Nanjing University, Nanjing, Jiangsu, China, jialiu@nju.edu.cn; Haipeng Dai, Nanjing University, Nanjing, Jiangsu, China, haipengdai@nju.edu.cn; Yu-E Sun, Soochow University, Suzhou, Jiangsu, China, sunye12@suda.edu.cn.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
2836-6573/2023/9-ART214 $15.00
https://doi.org/10.1145/3617334

heavy hitters, *i.e.*, flows with a size exceeding a pre-determined threshold in a measurement time window, becomes a fundamental task that enables network management and a broad range of network applications, ranging from traffic engineering [24] and flow-size-aware routing [22] to anomaly detection [30]. Typically, a network flow is a set of network packets with the same flow label customized by user applications, where the flow label can be one field or a combination of fields in packet headers, such as the source IP address, the destination IP address, or the 5-tuple fields (*i.e.*, a combination of source IP, source port, destination IP, destination port, and protocol header fields) in a TCP flow. The size of a network flow indicates the number of packets belonging to this flow during a measurement period.

A key challenge in detecting heavy hitters is the significant mismatch between the limited capacity of on-chip memory (usually less than 10MB [21]) of a router and the great volume of traffic to be measured in the high-speed network: we cannot simply store each flow and count them afterward; not to mention the unpredictable arrival time [15] and the dynamic and skewed distributions of flows [8]. To address these problems, enterprises usually sample and analyze a small fraction (generally lower than 1/1000) [5] of the packets using the Netflow protocol or the sFlow protocol at the cost of significant information loss, adversely affecting the accuracy of final analysis results [9, 12]. Alternatively, many advanced algorithms have been proposed by using compact data structures. These algorithms process the streaming packets one by one, striving to compress the whole traffic's information into a compact module placed on high-speed on-chip memory. We want to stress that these algorithms based on compact data structures can detect heavy hitters with high accuracy and consume a small number of computational and memory resources, thus earning widespread recognition as a promising solution for heavy-hitter detection. These algorithms generally fall into two categories: *sketch-based algorithms* and *KV-based algorithms*, according to the way of recording elements.

The *sketch-based algorithms* [1, 4, 7, 10, 11, 27] enable approximate flow measurement by counting each flow with a number of counters shared by different flows. Since each counter needs not to store flow labels and the number of counters is much smaller than that of flows, sketch-based algorithms are space compact. However, they suffer from network traffic with highly skewed distribution — a small number of heavy hitters and a large number of small-size flows (mice flows) for the task of heavy-hitter detection: for one thing, they require large counters to record heavy hitters, which is a waste of memory for mice flows; for another, the numerous mice flows will significantly cost memory space and introduce noises to the measurement of heavy hitters.

The *KV-based algorithms* [14, 16, 25, 28, 29] separate potential heavy hitters from a great number of mice flows by setting key-value (KV) pairs, *i.e.*, pairs of flow labels and flow sizes, which frees up memory occupied by mice flows and thereby improves memory efficiency. However, most of them have to fix the number and the size of KV-pairs during deployment, which cannot function well in the practical case of dynamic and skewed traffic distribution. To address this problem, the most recent work DHS [29] uses an extra counter array to record the numbers and usage of KV-pairs as well as support merging small KV-pairs into large ones. However, this is not free: DHS requires more memory accesses per packet and also has the limitation of small counting ranges (no larger than 65536). Therefore, KV-based algorithms face the dilemma of suffering dynamic traffic characteristics or lowering the system efficiency and flexibility.

In this paper, we propose a flexible sketch called SwitchSketch, the first heavy-hitter detection algorithm that offers high efficiency and high accuracy by resolving the dilemma of suffering dynamic traffic characteristics or lowering the system's efficiency and flexibility. The key idea of SwitchSketch is allowing the sketch to dynamically switch among different cell combinations to accommodate unknown network traffic and imbalanced flow distribution. We design a flexible bucket structure together with an encoding-based switching scheme to jointly achieve this goal.

The bucket structure has the competitive advantage of flexibility that benefits from two design principles. First, each bucket consists of variable-length cells that can adaptively count flows with various flow sizes (flexibility to flows). Second, given a cell, its counter field can be shrunk to some degree, which frees up some memory to accommodate more cells in size-constrained memory (flexibility to memory). For encoding-based switching, SwitchSketch uses a growing cell switching scheme, *i.e.*, replacing small cells with larger ones to make the sketch gravitate toward heavy hitters, meanwhile alleviating the noise caused by mice flows. In addition, SwitchSketch embeds a small-size encoding field (*i.e.*, metadata) into each bucket to offer efficient switching by reducing extra memory access and improving time efficiency. With these designs, SwitchSketch can detect heavy hitters in a flexible, efficient, and accurate way, regardless of the uncertainty and the imbalance of network traffic. The contributions of this work are summarized as follows.

- We propose SwitchSketch, a novel solution to the problem of heavy-hitter detection in high-speed networks, which resolves the dilemma of suffering dynamic traffic characteristics or lowering the system's efficiency and flexibility.
- We design an encoding-based switching scheme together with a flexible bucket structure to jointly improve memory efficiency and flexibility by using a combination of special design features, including variable-length cells, shrunk counters, embedded metadata, and switchable modes.
- We implement SwitchSketch on both a CPU-based server and a NetFPGA-1G-CML board to demonstrate its compatibility with both software and hardware platforms.
- We deploy SwitchSketch on different scenarios of heavy-hitter detection. Experimental results based on real traces show that SwitchSketch achieves a high F_β-Score of threshold-t detection (always higher than 0.938) and over 99% precision rate of top-k detection under a tight memory size (*e.g.*, 100KB). Besides, it outperforms the state-of-the-art by reducing the ARE by 30.77%∼99.96%.

2 PROBLEM STATEMENT

In a high-speed network, a packet stream over a period of measurement time usually has a large number of packets, which can be classified into T different flows $F = \{f_1, f_2, ..., f_T\}$ according to the flow label, where f_i indicates the flow label and the set of all packets carrying the same flow label f_i. The flow label is customized by user applications, *e.g.*, the source IP address, the destination IP address, or a subset of packet header fields. Let c_i be the size of flow f_i, which represents the number of packets (or bytes) in flow f_i, *i.e.*, $c_i = |f_i|$. Heavy-hitter detection is to identify the threshold-t heavy hitters or top-k heavy hitters over the network packet stream and figure out their flow labels and sizes, where threshold-t heavy hitters mean the flows with sizes larger than a predefined threshold t and top-k heavy hitters indicate k flows with the largest sizes in all flows F.

3 DESIGN OF SWITCHSKETCH

In this section, we first introduce two design features of SwitchSketch: flexible bucket structure and encoding-based switching scheme. The former consists of some variable-length cells that can achieve the flexibility to both flows and memory, and the encoding-based switching scheme allows a bucket to switch in different modes for better accommodating dynamic network traffic. With these designs, SwitchSketch can detect heavy hitters with higher memory efficiency and flexibility. Finally, we put things together and present the workflow of SwitchSketch.

3.1 Flexible Bucket Structure

As shown in Fig. 1, SwitchSketch consists of m fixed-size buckets that form a bucket array $B = \{B_1, B_2 \cdots, B_m\}$, where B_i $(1 \le i \le m)$ represents the i-th bucket with the length of W bits. Each

bucket is used to count the sizes of flows falling into this bucket via a random hash. Namely, given a packet with the flow label f_j, we calculate the hash value $h(f_j)\%m$ and use the bucket B_i to record the current size of the flow f_j, where $i = h(f_j)\%m$ indicates the index of the bucket picked by the flow f_j and $h(\cdot)$ is a hash function that returns a pseudo-random positive integer. After all packets are counted, we can figure out threshold-t heavy hitters or top-k heavy hitters by observing the counters stored in all buckets.

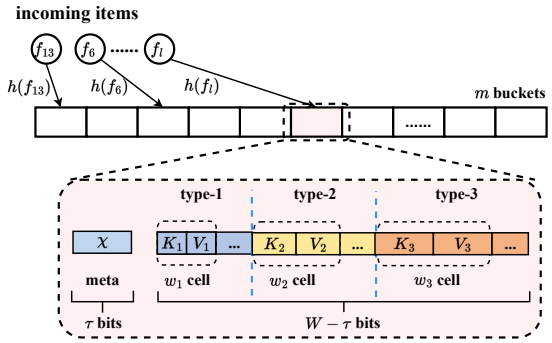


Fig. 1. Data Structure of SwitchSketch (W-bit bucket, τ-bit metadata)

3.1.1 Two Design Features. The bucket structure has the competitive advantage of flexibility to dynamic network traffic due to two design features. First, each bucket contains a couple of variable-length cells that are used to store different flow sizes. More specifically, each cell consists of two fields as shown in Fig. 1: the key field and the counter field. Given a packet with the flow label f_j that is hashed to the i-th bucket B_i, *i.e.*, $i = h(f_j)\%m$, we will choose one of the cells in the bucket B_i to record or update the size of the flow f_j, where the key field of the cell is used to store the fingerprint of this flow f_j, *i.e.*, the flow label's hash value $h(f_j)$, and the counter is storing the flow size of f_j at present. The reason why we use the fingerprint rather than the flow label is that the flow label is likely to be too long (*e.g.*, the 5-tuple header is 104 bits long) to be stored in a cell. The fingerprint is a good choice for identifying a flow, with high data compression at a very small cost of hash collisions. We will explain shortly how to choose and update the cell. Because the length of the cell is variable, we can use large cells to track big flows and smaller ones to record mice flows, which provides the bucket with the ability to accommodate different-size flows.

Second, given a cell, its counter field can be dynamically adjusted to some degree, which frees up some memory to accommodate more cells in the fixed-size bucket. More specifically, once a couple of cells are given, the fixed-size bucket might leave a few unoccupied bits that are not sufficient to generate a new cell, leading to a waste of memory space. In this case, we allow the counter field of a cell to shrink some bits to make room for a new cell and use ε to denote the number of shrunk bits for a cell. For example, when setting $0 \le \varepsilon \le 2$, a 16-bit cell with the 8-bit key field and the 8-bit counter field could be reorganized to a 15-bit cell with an 8-bit key field and a 7-bit counter or a 14-bit cell with an 8-bit key field and a 6-bit counter, which frees up one or two bits unoccupied for a new cell. To guarantee the same counting range as the cell before shrinking, we increase or decrease the value of the new counter with a probability of $\frac{1}{2^\varepsilon}$. By this means, we can maximize the number of cells and take full use of each bit of the bucket.

3.1.2 Cell & Mode. In SwitchSketch, L different-size cells (before shrink) are available to be chosen by a bucket, where C_i $(1 \le i \le L)$ denotes the i-th kind of cells and $w_i = |C_i|$ indicates its length. Without loss of the generality, we assume $w_i < w_j$, where $1 \le i < j \le L$. A large-size cell is tailored to a big flow and a small-size one is for a mouse flow. Given the cell C_i, half of the room ($\lceil \frac{w_i}{2} \rceil$ bits)

is used for the key field and half is left for the counter field. When the counter of a small-size cell tracking a flow f overflows, we replace the current cell with a larger one to accommodate this flow. Note that once a larger cell is used, we need to update the key value of the flow f: we can use the first $\lceil \frac{w_{i'}}{2} \rceil$ bits of the hash value $h(f)$ for the new cell $C_{i'}$.

A combination of cells in a bucket is referred to as a *mode* of the bucket at present. At the beginning of the detection, we assign small-size cells to a bucket for the purpose of tracking as many flows as possible. As the number of packets to be counted increases, we need larger cells to accommodate big flows, so the initial mode with only small-size cells should be transferred to a new mode with larger cells. This mode transfer is referred to as *switching*, which happens when the counter of any cell overflows. For example, given a 128-bit bucket with eight 16-bit cells (8 bits for the counter), if the size of a flow picking this bucket exceeds 255 (the counting range of the 16-bit cell), a mode transfer happens. The next mode could be a possible combination of one 32-bit cell and six 16-bit cells, where the 32-bit cell can track a flow with a size less than 65536. It is worth noting that if the counter of the longest cell C_L overflows, we can borrow the idea of *two-mode active counter* that use smaller memory to track extremely large flows at a low cost of accuracy [23]. This allows us to shorten cells, which improves the space efficiency to track elephant flows. We now give a brief introduction to *two-mode active counter*:

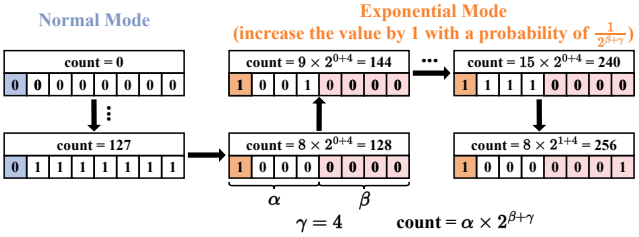


Fig. 2. Two-mode active counter

***Two-mode active counter*:** As shown in Fig. 2, *two-mode active counter* works and switches between two different modes (*i.e.,* normal mode and exponential mode). We define α as the value of the coefficient part, β as the value of the exponent part, and γ as the length of the exponent part. Given an 8-bit *two-mode active counter*, it initializes from normal mode, increments the count by 1 on each update, and switches to the exponential mode when its leftmost bit turns to '1' (*i.e.,* '10000000' whose value is 128). In exponential mode, the counter will be divided into two different parts (*i.e.,* coefficient part and exponent part) and adopt a sampling update strategy. At this time, the counter increases its coefficient part with a sampling rate of $\frac{1}{2^{\beta+\gamma}}$. When the counter's coefficient part overflows, the coefficient part is cleared to 0, but its leftmost bit should still be set to '1'. The *two-mode active counter*'s value can be estimated as:

$$c = \alpha \times 2^{\beta+\gamma}. \tag{1}$$

Mode switching helps the sketch gravitate toward heavy hitters and alleviates the noise caused by mice flows. However, the design of multiple modes requires the bucket to know which mode it is using. In SwitchSketch, we resort to the metadata as shown in Fig. 1 — a small-size field in the bucket that records the current mode, where $\lceil \log_2 n \rceil$ bits are sufficient to encode n modes. Embedding the metadata into the bucket also reduces extra memory access, which further improves the time efficiency of heavy-hitter detection.

3.1.3 Cell Settings. Now we discuss how to optimize the cell settings. Since an 8-bit byte is usually the smallest addressable unit of memory in many computer architectures, we assume that the length of a cell is an integral multiple of 8 bits, *i.e.*, the cell C_i is $8(i+1)$ bits long, where the minimized cell

C_1 is 16 bits long. The principle is to minimize the number of unused bits in a bucket. To achieve this goal, we propose a heuristic method called min-sum.

As mentioned above, a cell C_i can free up ε bits to make room for a new cell. The length w_i' of the final cell C_i is $w_i' = w_i - \varepsilon = 8(i+1) - \varepsilon$, where ε is an integer ranging from 0 to 2 (see details in Section 6.6.2). Besides cells, each bucket has embedding metadata for recording the current mode. The length τ of the metadata is $\lceil \log_2 |\mathcal{N}| \rceil$, where $\mathcal{N}$ is the mode set and $|\mathcal{N}|$ is the number of different modes. The optimization objective is to minimize the total number of unused bits, *i.e.*, $\sum_{i=1}^{L} \lambda_i$, where L is the number of different cells and λ_i represents the left bits unoccupied if only the cell C_i is used in the W-bit bucket. Thus, we have: $\lambda_i = W - \tau - \lfloor \frac{W-\tau}{w_i'} \rfloor w_i'$. We attempt to minimize the sum of all λ_i:

$$\min \sum_{i=1}^{L} \lambda_i, \quad \text{s.t.} \begin{cases} \lambda_i = W - \tau - \lfloor \frac{W-\tau}{w_i'} \rfloor w_i', & 1 \le i \le L \\ w_i' = 8(i+1) - \varepsilon, & 0 \le \varepsilon \le 2 \end{cases} \tag{2}$$

The min-sum optimization can be solved by brute-force searching all possible combinations of C_i and selecting the optimal one that yields the minimum sum of λ_i. For instance, when $W = 128$ and $L = 3$, the optimal cell settings are (C_1, C_2, C_3) with the size $(w_1', w_2', w_3') = (15, 24, 31)$ and $\tau = 4$. How we derive the value of τ will be shown below. For ease of presentation, we use the term w_i rather than w_i' afterward to indicate the length of cell C_i after cell settings.

3.2 Encoding-based Switching

With the flexible bucket structure, we design an encoding-based switching scheme that embraces dynamic and skewed traffic for heavy-hitter detection. As aforementioned, switching is the mode transfer among different modes, where a mode is a combination of cells for a bucket at a particular moment. Now we discuss how to choose the cells in a mode and how to encode different modes with metadata.

Consider a W-bit bucket and the optional cell set $\{C_1, C_2, ..., C_L\}$, where w_i is the length of the cell C_i satisfying $w_1 < w_2 \cdots < w_L$. We first enumerate all possible combinations of cell numbers. Let $\mathcal{N}$ be the set of all modes (cell combinations). A feasible mode is to make full use of every bit in the bucket — the memory occupied by all selected cells should be no larger than $W - \tau$ and the left unoccupied room cannot accommodate the smallest cell C_1, where τ is the length of metadata. Formally, we have:

$$\mathcal{N}(w_1, \cdots, w_L) = \{(n_1, \cdots, n_L) \mid W - w_1 < \sum_{i=1}^{L} n_i w_i + \tau \le W\}, \tag{3}$$

where n_i is the number of cell C_i chosen by a mode, and n_1 of each feasible mode can be computed according to $(n_2, \cdots, n_L)$ as follows:

$$n_1 = \lfloor \frac{W - \tau - \sum_{i=2}^{L} n_i w_i}{w_1} \rfloor \tag{4}$$

At the beginning of the detection, we try to use the modes with small-size cells for the purpose of tracking as many flows as possible. As the number of packets to be counted increases, we do mode switching and transfer to modes with larger cells for tracking big flows. Therefore, we encode modes increasingly according to the cells in the mode — the larger the cell size as well the more the large-size cells, the larger the serial number of the mode. Given the mode set $\mathcal{N}$, we can use τ-bit metadata to reflect the mode, where $\tau = \lceil \log_2 |\mathcal{N}| \rceil$ bits. All modes form a code table, which determines the transition direction between two different modes, from low-level modes to higher-level ones.

Table 1. Mode Table ($W = 128$)

(n_1, n_2, n_3)	Encoding	(n_1, n_2, n_3)	Encoding
(8, 0, 0)	0000	(3, 2, 1)	1000
(6, 1, 0)	0001	(1, 3, 1)	1001
(5, 2, 0)	0010	(4, 0, 2)	1010
(3, 3, 0)	0011	(2, 1, 2)	1011
(1, 4, 0)	0100	(0, 2, 2)	1100
(0, 5, 0)	0101	(1, 0, 3)	1101
(6, 0, 1)	0110	(0, 1, 3)	1110
(4, 1, 1)	0111	(0, 0, 4)	1111

For example, when W is 128 and there are 3 types of cells. The optimal parameters w_1, w_2, w_3 are 15, 24, and 31, which are computed according to the min-sum optimization. The mode set and code table are shown in Table 1. In this case, there are 16 legal combinations such that we can use only a 4-bit metadata field to achieve the switching. If the i-th mode cannot meet the counting requirements (overflow happens), we conditionally replace the mode with the j-th mode that has larger cells for resolving the overflow, where $j > i$. Notice that once a bucket's mode is changed, the metadata needs to be adjusted accordingly.

3.3 Workflow for Heavy Hitter Detection

SwitchSketch performs heavy-hitter detection with four operations: Initialization, Insertion, Switch, and Query. Initialization prepares the sketch for heavy hitter detection by setting all buckets to the first mode. During detection, the sketch uses the Insertion operation to deal with each incoming packet such that the flow information is updated accordingly. The Switch operation is executed during the insertion operation when an incoming flow's size exceeds the counting range of the cell picked by this flow. At this time, a mode switching happens from the previous one to the latter based on the metadata. Query can be called either online (*i.e.*, during the measurement) or offline (*i.e.*, when a measurement period ends) to obtain a list of heavy hitters. The four operations are detailed below. Without loss of the generality, we assume each cell is 128 bits ($W = 128$) and there are 3 types of cells ($L = 3$) when we introduce the workflow of SwitchSketch.

3.3.1 **Initialization.** To prepare the sketch for heavy hitter detection, we traverse each bucket, set the bucket mode to the first mode (*i.e.*, the mode with the code 0), and initialize all keys and counters to zeros.

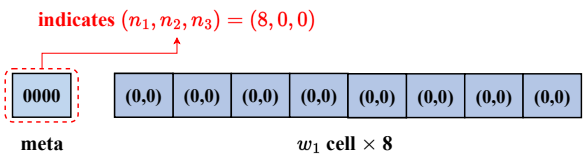


Fig. 3. Initial State

3.3.2 **Insertion.** As shown in Algorithm 1, an insertion operation is executed once a packet with flow label f_i arrives, where the flow information is updated accordingly. More specifically, SwitchSketch first extracts flow label f_i from the packet and then performs insertion. According to the hash function, the flow f_i is mapped to the j-th bucket B_j, where $j = h(f_i)\%m$ and m is the number of buckets. To perform insertion, we load the W-bit bucket data and decode all cells according to the metadata χ, which is embedded in the bucket without extra memory access. We check each cell from type L to 1 to see whether f_i has been recorded. If f_i is recorded by the key field of a cell, we move to one of the following two cases. If the counter is large enough to track this flow, we execute the insertion without switching (see *Case 1*); otherwise, we do insertion with

switching (see *Case 2*). If f_i is a new flow that has not been recorded yet, SwitchSketch will choose the smallest empty cell to record this flow (see *Case 3*). If no empty cell is available, we try to free up the limited memory space and replace the recorded small flow with a potential elephant flow (*i.e.*, do an *exponential decay* operation on the smallest occupied cell, see *Case 4*). The details of the four cases are listed below.

Algorithm 1: Insertion ()

```
Input: A packet belonging to flow f_i, exponential base number
       b = 1.08, threshold t, cell type k
Query and check whether f_i is recorded;
if f_i has been recorded in the cell C_j then
    if C_j's counter field can be increased without overflow then
        Do insertion without switching;          /* Case 1 */
    else
        Do insertion with switching;             /* Case 2 */
else
    for k ← 1 to L do
        if there are C_k then
            if an empty cell C_k exists then
                Insert (h(f_i), 1) into the C_k;  /* Case 3 */
            else
                /* Case 4 */
                find the cell C_k with the smallest counter;
                if the counter works in the normal mode then
                    Do exponential decay () on the cell;
            break;
```

- *Case 1:* The flow f_i's counter field can be increased without overflow. SwitchSketch would increase f_i's size by 1 directly. It is worth noting that if the cell is shrunk by ε bits, we increase its counter field by 1 with a probability of $\frac{1}{2^{\varepsilon}}$ in order to have the same counting range as the normal counter. For the largest cell C_L, we use the *two-mode active counter* [23] to save memory space.
- *Case 2:* The flow f_i's counter field would overflow if increased by 1. Suppose the current cell is C_j, where $j < L$. In this case, we call a switch operation (the detailed process will be explained shortly) to create a larger cell C_{j+1} for recording the current flow. If the switch operation successfully creates C_{j+1} without affecting existing large flows, we can increase f_i's size by 1 and record it with the new cell. However, because the bucket size is limited, we might fail to do the switch. At this point, we conditionally remove the smallest flow (called target flow) at cell C_k, where $k > j$, to make room for the flow f_i. Specifically, we call an *exponential decay* function [25, 28, 29] that decreases the target flow's counter by 1 with a probability of $Pr = b^{-\log_2 c_k}$, where b is a predefined constant number, *e.g.*, $b = 1.08$, and makes elephant flows replace small flows with high probability, where c_k is the counter value of the target cell C_k. Notice that as a type-L cell's counter can be a *two-mode active counter*, SwitchSketch will not do *exponential decay* when the counter works in the exponential mode. Our explanations for this design can be boiled down to two aspects. First, the *two-mode active counter*'s probability count cannot be routinely decayed. Second, when a counter works in the exponential mode, its recorded size is large enough. The recorded flow is probably an elephant flow, and hence we do not need to decay it.

- *Case 3:* The flow f_i has not been recorded yet and there exist empty cells. SwitchSketch chooses the smallest empty cell, *e.g.*, C_j, and inserts the first $\lceil \frac{w_j}{2} \rceil$ bits as the key and 1 as the current flow size into the cell C_j,
- *Case 4:* The flow f_i has not been recorded yet and no empty cell exists as well. SwitchSketch finds out the smallest occupied cell and does *exponential decay* on it. If the smallest cell's counter field is decayed to 0, we insert f_i as the same way in Case 3. It is worth noting that when the counter is the *two-mode active counter* and works in an exponential manner, no decay is required. Besides, an ε-bit shrunk cell is decayed with a probability of $\frac{1}{2^\varepsilon}$.

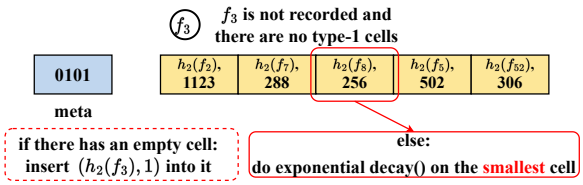


Fig. 4. An example of decaying the smallest flow

Example: In Fig. 4, we present an example of Case 4, which uses an *exponential decay* to remove the smallest flow. Case 2 also applies the *exponential decay* strategy when the switch operation cannot produce a larger cell. In this example, the incoming packet carrying flow label f_3 is not recorded and there is no empty cell in the bucket. As there exist no empty cells, *exponential decay* is applied to the smallest cell. If the cell's counter field is equal to 0 after decaying, the recorded flow f_8 will be replaced with the new incoming f_3.

3.3.3 **Switch.** The Switch operation is triggered when a flow f_i located at the cell C_j $(i < L)$ requests for a larger cell. SwitchSketch first decodes the metadata χ to obtain the current mode $(n_1, n_2, \cdots, n_L)$. The task is to switch the bucket to a new mode which keeps all flows larger than f_i unchanged and adds an extra cell C_{j+1} to record this flow, *i.e.*, $n_{j+1} = n_{j+1} + 1$ and $n_k = n_k$, where $k > j + 1$. Clearly, this attempt is not always successful because of the fixed and limited bucket size, which in this case needs to remove empty cells or the smallest flows to free up memory room.

The Switch process is detailed in Algorithm 2. Given a flow f_i stored in the cell C_j of a bucket, we reassign this flow to a new cell C_{j+1} as follows: We repeatedly remove the smallest cells from C_1 cells to C_j cells increasingly. After each removal, we check whether the existing cells larger than the removed one can be allocated successfully, *i.e.*, the memory space occupied by all cells is no larger than the available bits $(W - \tau)$. If we can find such an allocation (corresponding to a mode), we will switch the bucket to the new mode by updating the metadata, updating the flow information, and ending the switch operation. If we cannot find a feasible allocation until all C_j cells are deleted, it means the bucket cannot assign a new cell C_{j+1}. At this point, we report a failure event of the switch operation and remain the current mode unchanged.

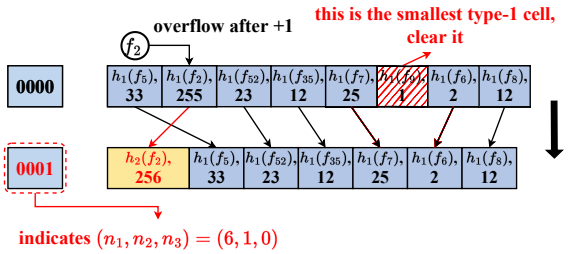


Fig. 5. An example of switching mode

Algorithm 2: Switch

Input: flow f_i, current cell type s, switch $flag$

$flag \leftarrow false$;
get the current mode χ;
traverse the data field to get the non-empty cell numbers, *i.e.*, $(n'_1, n'_2, \cdots, n'_L)$;
$n'_{s+1} \leftarrow n'_{s+1} + 1$;
$n'_s \leftarrow n'_s - 1$;
for *type* k *from* 1 *to* s **do**
  while $n'_k \geq 0$ *and* $flag = false$ **do**
    $remain\_bits = W - \tau - \sum_{k'=k}^{L} n'_{k'} w_{k'}$;
    if $remain\_bits \geq 0$ **then**
      $\chi' = (n'_k + \lfloor \frac{remain\_bits}{w_k} \rfloor, n'_{k+1}, \cdots, n'_L)$;
      Switch the bucket mode to χ';
      Insert $(f_i, c+1)$ into the new type-(s+1) cell;
      $flag \leftarrow true$;
      break;
    else
      $n'_k \leftarrow n'_k - 1$;
      Delete the smallest flow's KV-pair in type-k cells;
if $flag = true$ **then**
  Return χ';
else
  Return χ and report a switch-failure event;

Example: As shown in Fig. 5, the new incoming f_2 will make the cell C_1 overflow after increasing by 1. That means we need to reconstruct a new cell C_2, which triggers a Switch operation. We first decode the metadata $\chi = 0000$ to get the current mode $(n_1, n_2, n_3) = (8, 0, 0)$. Since we need a new cell C_2, the new mode is χ'$= (6, 1, 0)$ after Switching. That means we should remove the smallest cell C_1 and put $(h(f_2), 2^8)$ into the newly constructed cell C_2. Finally, the metadata χ should also be updated to '0001'.

3.3.4 **Query.** The query process of SwitchSketch follows a *highest type first* principle, which means it starts from the high-level cells to the low-level cells, *i.e.*, from C_L to C_1. As a small number of heavy hitters constitute the most of the network traffic [20], most of the queries can be finished at a high level, which helps improve the throughput. Specifically, for each incoming packet belonging to flow f_i, the Query operation traverses all the cells in $B[h(f_i)\%m]$. If f_i's fingerprint is recorded in one of the cells, the Query operation reports the value of the counter field. If no cell matches, the Query returns 1.

We consider two cases of heavy-hitter detection: online detection and offline detection. Online detection represents that the detection operation is executed during measurement. Offline detection indicates that we periodically download the sketch to off-chip and report heavy hitters afterward.

- **Online detection:** For online heavy-hitter detection, SwitchSketch uses an auxiliary list to store the flow labels whose sizes first exceed a predefined threshold t during a measurement period. By setting appropriate t, it can track top-k heavy hitters in the auxiliary list as well. The performance of online detection heavily relies on the algorithm's accuracy and effectiveness, as plenty of misidentifications recorded in the auxiliary list may consume scarce on-chip memory.

- **Offline detection:** For offline heavy-hitter detection, SwitchSketch traverses all the known flows to estimate their sizes after a measurement period. If a flow's estimated size exceeds a predefined threshold t (or belongs to the estimated top-k largest ones), it will be considered as a threshold-t (or top-k) heavy hitter. The key to offline detection is not just counting each flow's size accurately but establishing a precise correspondence between each flow and its query result.

For the offline scenarios, we notice that the fingerprint design suffers from the false positives of mice flows. Therefore, we propose to design a new mechanism called *double-check* to reduce the chances of false positives.

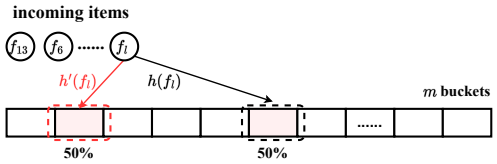


Fig. 6. Double-check Mechanism

As shown in Fig. 6, consider an incoming packet belonging to flow f_i. Besides $h(.)$, we use another hash function $h'(.)$ to do bucket selection, where f_i is mapped into two different buckets through $h(.)$ and $h'(.)$ with an equal probability of 0.5. It is worth noting that each packet is still hashed once for each insertion. In the process of offline traversal query, we double-check the flow f_i's two corresponding buckets and compare the two counters corresponding to this flow (two estimates with great differences are very likely to be false positives). Assume the recorded size of f_i is c_i and c_i', f_i's estimated size is legal only if

$$1 - \frac{min(c_i, c_i')}{max(c_i, c_i')} < \theta, \tag{5}$$

where θ can be a small positive number that is smaller than 1 (*e.g.*, 0.6). A legal flow size will be estimated as $c_i + c_i'$, while an illegal flow size will be estimated as 1. The double-check mechanism can significantly reduce the ARE of offline heavy-hitter size estimation and improve the accuracy of offline heavy-hitter detection. Moreover, it does not add an extra overhead of online insertion, as it still hashes once for each packet's mapping.

3.4 Applications

Outside the network traffic measurement, heavy-hitter detection, also as a key task in broader data stream processing, has gained significant attention and interest within the data management community. The analysis of heavy hitters' statistics is essential for database optimization [25, 31] and data mining [14, 29]. For instance, in database optimization, it is important to find predominate attribute values in the large database table for query planning and approximate query answering. Also, Google may be interested in the hot web pages that users set as the default homepage of Chrome.

4 MATHEMATICAL ANALYSIS

In this section, we analyze the error bound of our SwitchSketch. Assume there are m buckets in SwitchSketch and there has been a network packet stream with N packets and T flows recorded in SwitchSketch. Let f_i be the i^{th} largest flow, whose size is c_i. We use l_s to represent the bit length of type-s cell's fingerprint field. The ratio of packets falling in type-s segments across all buckets is ω_s. We define γ as the length of *two-mode active counter*'s exponent part. The estimated size of f_i is

$$\hat{c}_i = c_i - X_i + Y_i + O_i + P_i, \tag{6}$$

where X_i is the decrement caused by *exponential decay* and Y_i is the increment caused by fingerprint collisions. O_i represents the increment that might be caused by the shrunk cell's probabilistic update strategy, and P_i represents the increment that might be caused by *two-mode active counter*.

4.1 Error Bound Analysis

Lemma 4.1. *Given a ϕ-bit two-mode active counter whose exponent part's length is γ, its counting range is* $[0, 2^{2^\gamma+\phi-1} - 2^{2^\gamma-1+\gamma}]$.

Proof. In the extreme case, counter's γ-bit exponent part and (ϕ-γ)-bit coefficient part are all 1. The value of exponent part is $2^\gamma - 1$ and the value of coefficient part is $2^{\phi-\gamma} - 1$, so its maximum counting value is

$$max = (2^{\phi-\gamma} - 1)2^{2^\gamma-1+\gamma} = 2^{2^\gamma+\phi-1} - 2^{2^\gamma-1+\gamma}. \tag{7}$$

□

For any two flows f_i and f_j which are hashed into the same bucket and belong to the same type-s cell. Let $I_{i,j,s}$ be a binary random variable, defined as

$$I_{i,j,s} = \begin{cases} 0 & h_s(f_i) \neq h_s(f_j) \\ 1 & h_s(f_i) = h_s(f_j) \end{cases}. \tag{8}$$

$I_{i,j,s}$ indicates whether f_i and f_j share the same fingerprint in type-s cell, and then we have $E(I_{i,j,s}) = 2^{-l_s}$. Therefore, the expectation of Y_i

$$E(Y_i) \leq \sum_{s=1}^{L-1} min(\omega_s \frac{N}{m} 2^{-l_s}, 2^{l_s} - 1) + min(\omega_L \frac{N}{m} 2^{-l_L}, 2^{2^\gamma+\phi-1} - 2^{2^\gamma-1+\gamma}), \tag{9}$$

where $\omega_s \frac{N}{m} 2^{-l_s}$ is the expected collided packets in type-s cell, and the second term is the maximum counting range in each type of cell which has been proved in lemma 4.1.

Then we focus on O_i, which represents the increment that might be caused by the shrunk cell's probabilistic update strategy. We define ε as the shrunk bits and $\varepsilon = 1, 2$. Assume that the shrunk cell is type-k whose fingerprint field is l_k bits, so its counter field is $l_k - \varepsilon$ bits. In the worst case, the f_i's recorded counter will increase for each incoming packet until it overflows (*i.e.*, the counter field reaches $2^{l_k-\varepsilon}$). The original flow size is $2^{l_k-\varepsilon}$, but it can be estimated as 2^{l_k} in the worst case. Therefore, $E(O_i)$ is subject to

$$E(O_i) \leq 2^{l_k} - 2^{l_k-\varepsilon}. \tag{10}$$

We also consider the worst case of P_i in which the f_i's recorded counter will increase for each incoming packet. We assume that the counter finally works in the exponential mode. We define its value of coefficient part as α_i and its value of exponent part as β_i. The actual flow size can be calculated as

$$n_{worst} = 2^{\phi-1} + 2^{\phi-1-\gamma}\beta_i + \alpha_i - 2^{\phi-1-\gamma}, \tag{11}$$

where the first term denotes the number of packets that make the counter work into the exponential mode, the second term denotes the number of packets that make the exponent part from initial 0 to β_i, and the final term denotes the extra number of packets that make the coefficient part from $2^{\phi-1-\gamma}$ to α_i. Its size is finally estimated as

$$\hat{n}_{worst} = \alpha_i 2^{\beta_i+\gamma}. \tag{12}$$

Therefore, the expectation of P_i is no larger than $\hat{n}_{worst} - n_{worst}$, which is:

$$E(P_i) \leq \alpha_i 2^{\beta_i+\gamma} - 2^{\phi-1} - 2^{\phi-1-\gamma}\beta_i - \alpha_i + 2^{\phi-1-\gamma}. \tag{13}$$

Finally, given a small positive number ϵ and based on Markov inequality we have

$$\begin{aligned} Pr\{\hat{n}_i - n_i \geq \epsilon N\} &\leq Pr\{Y_i + O_i + P_i \geq \epsilon N\} \\ &\leq \frac{E(Y_i) + E(O_i) + E(P_i)}{\epsilon N} \\ &\leq \sum_{s=1}^{L-1} min(\frac{\omega_s}{\epsilon m} 2^{-l_s}, \frac{2^{l_s} - 1}{\epsilon N}) + min(\frac{\omega_L}{\epsilon m} 2^{-l_L}, \frac{2^{2^\gamma + \phi - 1} - 2^{2^\gamma - 1 + \gamma}}{\epsilon N}) \\ &+ \frac{2^{l_k} - 2^{l_k - \varepsilon}}{\epsilon N} + \frac{\alpha_i 2^{\beta_i + \gamma} - 2^{\phi - 1} - \alpha_i + 2^{\phi - 1 - \gamma}(1 - \beta_i)}{\epsilon N}. \end{aligned} \tag{14}$$

5 IMPLEMENTATIONS

In this section, we briefly describe the implementations of hardware and software versions of the SwitchSketch on FPGA and CPU platforms, respectively. The source code of all platforms is available at Github [6].

5.1 Hardware Implementation

We implement the hardware version of SwitchSketch on a NetFPGA-embedded prototype, where a NetFPGA-1G-CML development board connects with a workstation (with Ryzen7 1700 @3.0GHz CPU and 64GB RAM) through PCIe Gen2 X4 lanes. As shown in Fig. 7, the SwitchSketch is placed in the dual-ported BRAM and we use one port for memory read and the other port for memory write. For each incoming packet, we need to retrieve the corresponding bucket, update it and write it back to the BRAM. Specifically, in Stage 1 (memory read stage), the target bucket is retrieved from the BRAM using the hash value computed from the flow ID f. In Stage 2 (sketch update stage), we update the bucket following the mechanism of SwitchSketch. In Stage 3 (memory write stage), the modified bucket is written back into the memory.

We optimize the hardware version with full pipelining in order to reduce the initiation interval for a function or loop by allowing the concurrent execution of operations, achieving high throughput of packet processing and forwarding. In terms of system throughput, according to the estimated Fmax given by Vivado HLS, the highest processing throughput of 64-bit, 128-bit, and 256-bit are 114.29, 114.29, and 109.96 Mpps (million packets per second), separately. Benefiting from the pipeline optimization, there is no difference in throughput between the first two versions. Due to the complexity of the SwitchSketch with 256-bit buckets, the throughput drops slightly.

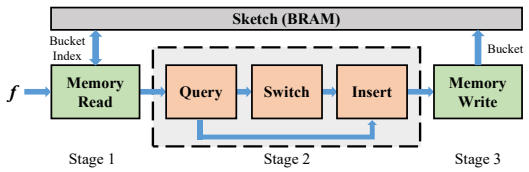


Fig. 7. Pipelined SwitchSketch maintenance in FPGA

Table 2 shows the hardware resource usage of different versions of SwitchSketch under different bucket sizes. The main resources consumed by FPGA implementation include Block RAM Tile (BRAM), DSP48E, Flip Flop (FF), and Look-Up-Table (LUT). Block RAM Tile is on-chip block memory, which is the main storage resource. DSP48E logic slices are digital signal-processing logic units that can be used to implement different arithmetic operations. It can be seen from the table that when the bucket size is fixed, the resource overheads of different versions are very close because the main structure of the mechanism is the same. Under the same version, the resource overhead

increases with the expansion of the bucket size because a larger size means more cells, upgrades, and complex conversion mechanisms.

Table 2. Resource Usage of Hardware Implementations

Version	Bucket Size	BRAM	DSP48E	FF	LUT
Online	64	210	136	34505	55426
	128	210	136	35519	69259
	256	210	146	39020	97905
Offline	64	210	136	34558	55450
	128	210	136	35572	69283
	256	210	146	39105	97942

5.2 Software Implementation

The software version of SwitchSketch is also implemented for the sake of comparison with the state-of-the-arts. All methods are written in C++ and evaluated on a server equipped with two Intel Xeon E5-2643 v4 @3.40GHz CPU and 256GB RAM. Besides, the hash functions are implemented by Murmur Hash.

6 EVALUATION

6.1 Experimental Setup

Platform: We conducted our evaluation on a server equipped with two six-core Intel Xeon E5-2643 v4 3.40GHz CPU and 256GB RAM.

Dataset: The datasets we use are two 1-minute traces downloaded from CAIDA-2016 [2] and CAIDA-2019 [3]. In the traces, each flow's label is identified by the source IP address. The 1-minute CAIDA-2016 trace contains over 31M packets belonging to 0.58M flows, and the 1-minute CAIDA-2019 trace contains over 36M packets belonging to 0.37M flows.

Baselines and Implementations: We compare SwitchSketch with the state-of-the-art solutions including HeavyGuardian [25], WavingSketch [14] and DHS [29]. All methods are implemented in C++. We did not implement some classical methods, *e.g.*, SpaceSaving [16] and Count-min [7] because the recent methods we implemented above have been proven to achieve better performance in most of the evaluation metrics [29]. Besides, we did not compare with ChainSketch [13] because its main contribution, *i.e.*, hash chain operations, can also optimize SwitchSketch and the baselines to improve memory efficiency at the cost of lowering processing throughput.

Parameter Setting: The largest flow size in our 1-minute trace is 569743, which at least needs a 20-bit counter field. So we adjust the bits of the counter field in HeavyGuardian and WavingSketch to 20 bits. For the DHS, we implement one adjusted version for comparison, *i.e.*, DHS with a new $levelup()$ mechanism (DHS for short). Specifically, we increase the bits of the level-3 counter field to 20 bits such that three-level cells of DHS are 16, 24, 36 bits. Then we design a new $levelup()$ mechanism, *i.e.*, three level-1 cells for a level-3 one, three level-1 cells for two level-2 ones, and three level-2 cells for two level-3 ones. As DHS's original mechanism is strictly restricted, when the largest counter field is turned into 20 bits, it cannot find a perfect $levelup()$ mechanism without any waste of memory. For the other parameters in these algorithms, we tune them as the original paper recommended for the best performance. For SwitchSketch, we set each bucket to 128 bits, the same as DHS does. The exponential base number b is set to 1.08, the *double-check* base number θ is set to 0.6, and the total memory size determines the number of buckets m. The *two-mode active counter*'s exponent part length is $\gamma = 4$ (see details of parameter setting in 6.6.1).

Online Query and Offline Query: In our experimental setup, the online queries of the three fingerprint-based methods (*i.e.*, HeavyGuardian, DHS and SwitchSketch) use an auxiliary list to

store the flows whose sizes exceed a predefined threshold t during a measurement period. It's worth noting that WavingSketch saves the flow labels directly in its structure. For fairness, the memory usage of fingerprint-based method's auxiliary list should be added in the implementation of WavingSketch. Offline query simply traverses all the known flows to query their sizes after a measurement period, and we do not need the auxiliary list in the offline query.

6.2 Metrics

False Positive Ratio (FPR) and False Negative Ratio (FNR) in Online Threshold-t Heavy-Hitter Detection: We define threshold t, and the gap between $bottom$ and top, where $t = (top + bottom)/2$ and $bottom = top/2$. Then, we define the FPR (False Positive Ratio) as the fraction of all of the flows with a size smaller than $bottom$ that is mistakenly identified. The FNR (False Negative Ratio) is defined as the fraction of all of the flows with a size greater than top that fails to be identified.

F_β-Score in Online Threshold-t Heavy-Hitter Detection: $\frac{(1+\beta^2)\times PR\times RR}{(\beta^2\times PR)+RR}$, where PR (Precision Rate) is the fraction of true heavy hitter instances reported among all the instances reported and RR (Recall Rate) is the fraction of reported true heavy hitter instances among all the heavy hitters. The adjusted F_β-Score allows us to weigh PR or RR more highly if it is more important. In the case of heavy-hitter detection, we assume that RR carries a higher reward than PR [17] which means we are more concerned about finding out all the heavy hitters, so we set $\beta = 2$.

Precision Rate (PR) in Offline Top-k Heavy-Hitter Detection: $\frac{l}{k}$, where l is the number of reported flows that belong to the real top-k flows.

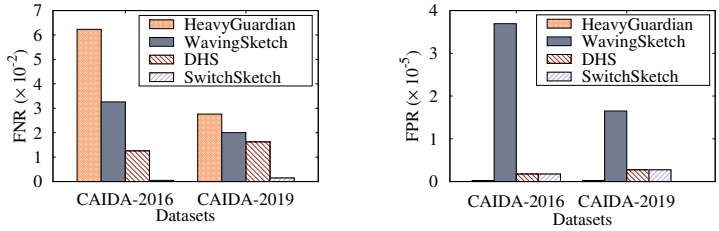
Average Relative Error (ARE) in Offline Threshold-t Heavy-Hitter Size Estimation: $\frac{1}{|\psi|}\times \sum_{f_i\in\psi}\frac{|c_i-\hat{c}_i|}{c_i}$, where ψ is the final query set of heavy hitters and c_i is the real size of flow f_i in ψ, $\hat{c}_i$ is its estimated size.

Million Operations per Second (Mops) in Throughput: $\frac{N}{t_N}$, where N is the number of packets and t_N is the time used.

6.3 Experiments on Online Queries

We compare four approaches: HeavyGuardian, WavingSketch, DHS, and SwitchSketch in the experiments of online threshold-t heavy-hitter detection. Except for WavingSketch, the other three algorithms are fingerprint-based. The fingerprint is irreversible that we cannot simply derive the original flow label from the fingerprint. Hence, the fingerprint-based methods need the auxiliary list to record the heavy hitters' flow labels, which incurs extra memory overhead (the list's memory is 25KB in our experiments). We define M as the baseline of memory size, and we aim to keep the module that is actually used for detection sharing the same memory. For HeavyGuardian and SwitchSketch, we set the sketch's memory size to M KB. As DHS has an extra data structure called metadata to identify each bucket's structure for cell adjustment, which is 24-bit per 128-bit bucket, its sketch's memory size is set to $\frac{128}{128+24}\times M$ KB. As WavingSketch records the entire flow label in its structure and does not need the auxiliary list, its memory size is set to $M + 25$ KB. Specifically, we use FNR, FPR, and F_β-Score as the metrics to evaluate detection accuracy.

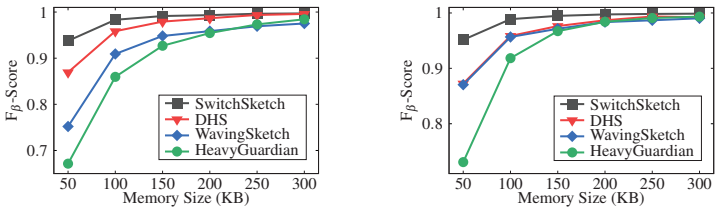
FNR/FPR vs. Two Datasets: In this experiment, we set M to 150 KB and set threshold t to 750 ($bottom = 500, top = 1000$). As shown in Fig. 8(a), SwitchSketch gets the lowest FNR compared to the other algorithms. Take the result on CAIDA-2016 as an example; there are 4141 flows whose sizes are larger than $top = 1000$. SwitchSketch fails to identify only 2 flows, while HeavyGuardian, WavingSketch, and DHS fail to identify 258, 135, and 52 flows, respectively. As shown in Fig. 8(b), all the algorithms can get high performance in the metric of FPR, and the FPR of HeavyGuardian is always 0 because it has been proven to be no over-estimation error. Specifically, there are 568467



(a) FNR vs. Two Datasets (b) FPR vs. Two Datasets

Fig. 8. FNR and FPR on Online Threshold-t Heavy-Hitter Detection

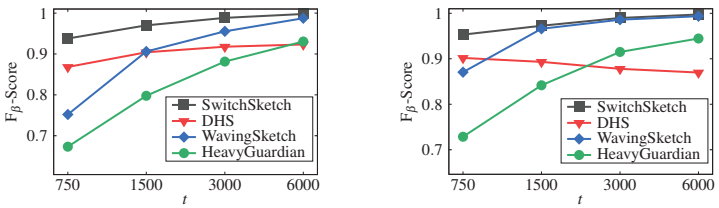
flows whose sizes are smaller than $bottom = 500$ on the dataset of CAIDA-2016, the mistakenly identified flows of HeavyGuardian, WavingSketch, DHS, and SwitchSketch are 0, 21, 1, and 1, respectively, of which the gap is negligible.



(a) CAIDA-2016 (b) CAIDA-2019

Fig. 9. F_β-Score on Online Threshold-t Heavy-Hitter Detection (Varying Memory Size)

F_β-Score vs. Memory Size: In this experiment, we evaluate each F_β-Score of these four approaches under the same threshold $t = 750$ by varying M from 50 KB to 300 KB. As shown in Fig. 9(a)-9(b), we find that SwitchSketch's F_β-Score is always higher than the others, irrespective of the memory size or the datasets. SwitchSketch's performance increase is more pronounced when the memory size is limited. Take the results on CAIDA-2019 as an example; when M is set to 50 KB, the F_β-Score of SwitchSketch is more than 0.951, while the HeavyGuardian, WavingSketch, DHS's F_β-Score is only 0.731, 0.871, and 0.872, respectively.



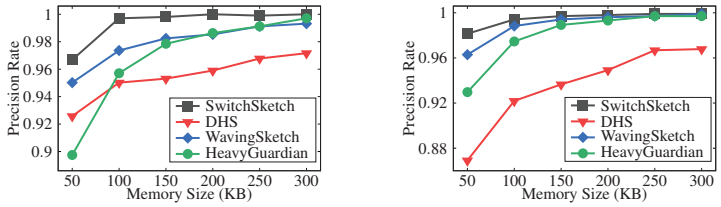
(a) CAIDA-2016 (b) CAIDA-2019

Fig. 10. F_β-Score on Online Threshold-t Heavy-Hitter Detection (Varying t)

F_β-Score vs. t: In this experiment, we evaluate each F_β-Score of these four approaches under the same baseline of memory size $M = 50$ KB by varying t from 750 to 6000 (*i.e.*, varying top from 1000 to 8000). As shown in Fig. 10(a)-10(b), SwitchSketch consistently achieves the highest F_β-Score. For all considered threshold t, the F_β-Score of SwitchSketch does not go below 0.938, while the lowest F_β-Score of HeavyGuardian, WavingSketch, and DHS is only 0.673, 0.752, and 0.868, respectively.
Analysis: According to the results, SwitchSketch can achieve higher detection accuracy than the other algorithms because of its more flexible adjustment scheme. It can dynamically allocate memory bits for accommodating many flows with varied sizes in each bucket, which in other

words, can make the best of memory to increase the detection accuracy. Moreover, the F_β-Score of WavingSketch is close to SwitchSketch when the memory size is enough, or the t is relatively large. However, each flow label in our datasets is a 32-bit IP address. When the key turns larger (*e.g.*, the 5-tuple header, which is 104-bit), WavingSketch needs to allocate more memory bits for recording each flow label. That reduces the number of buckets and hence harms the performance of detection.

6.4 Experiments on Offline Queries

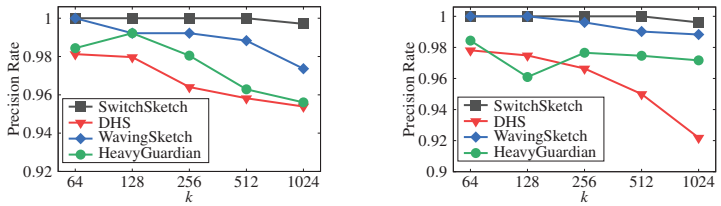


(a) CAIDA-2016 (b) CAIDA-2019

Fig. 11. Precision Rate on Offline Top-k Heavy-Hitter Detection (Varying Memory Size)

In this section, we apply the *double-check* mechanism to the basic SwitchSketch for the scenario of offline queries. We still use M as the baseline of memory size. As the offline queries do not need the auxiliary list, the total memory usage of HeavyGuardian, WavingSketch, and SwitchSketch are all M KB, and the total memory usage of DHS is still $\frac{128}{128+24} \times M$ KB. Moreover, we use PR as the metric to evaluate the accuracy of top-k heavy-hitter detection. To evaluate the precision of heavy-hitter size estimation and the effectiveness of the *double-check* mechanism, we use threshold-t heavy hitters' ARE as the metric. The reason we do not use top-k heavy hitters' ARE is that the number of top-k heavy hitters is relatively small, the fingerprints' collisions may greatly influence the final ARE (*e.g.*, a mouse flow whose size is 1 shares the same fingerprint with an elephant flow whose size is 10001, this misidentification significantly increases the final ARE). Hence, we choose a relatively small threshold $t = 750$ to estimate more heavy hitters and make the final result more stable and rational.

PR vs. Memory Size: In this experiment, we set $k = 1024$ and vary the whole memory size from 50 KB to 300 KB. As shown in Fig. 11(a)-11(b), we find that SwitchSketch always gets the highest PR on both of the datasets. Take the results on the CAIDA-2016 as an example; when the memory size is set to 50 KB, SwitchSketch can get 96.68% PR of top-1024 flows, while that of HeavyGuardian, WavingSketch, DHS is 89.74%, 95.02%, 92.58% respectively.

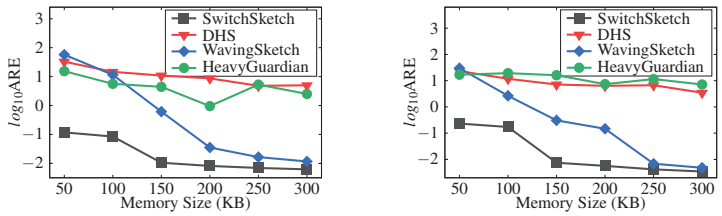


(a) CAIDA-2016 (b) CAIDA-2019

Fig. 12. Precision Rate on Offline Top-k Heavy-Hitter Detection (Varying k)

PR vs. k: In this experiment, we set $M = 100$ KB and vary the k from 64 to 1024. As shown in Fig. 12(a)-12(b), SwitchSketch always achieves higher PR of top-k flows than the others under different

k. Specifically, when k is set to 512, SwitchSketch can find all top-512 flows on both of the datasets, while HeavyGuardian, WavingSketch, DHS can find 493/499, 506/507, 491/487 of top-512 flows on CAIDA-2016/CAIDA-2019 datasets, respectively.

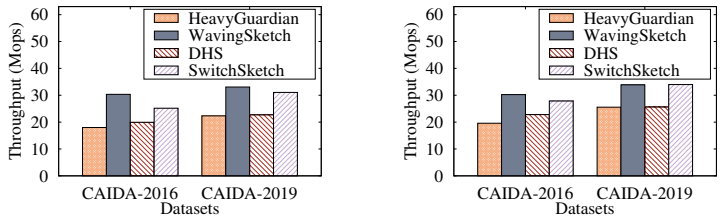


(a) CAIDA-2016 (b) CAIDA-2019

Fig. 13. ARE on Offline Threshold-t Heavy-Hitter Size Estimation (Varying Memory Size)

ARE vs. Memory Size: In this experiment, we set $t = 750$ and vary the whole memory size from 50 KB to 300 KB. As shown in Fig. 13(a)-13(b), on both of the datasets, SwitchSketch's AREs are almost 2-3 orders of magnitude lower than the others. Only when the memory size gets large, WavingSketch's ARE is close to SwitchSketch's. However, SwitchSketch still outperforms the WavingSketch by reducing the ARE by 30.77%-57.82% when the memory size is 300 KB.
Analysis: According to the results, SwitchSketch can track top-k heavy hitters with high accuracy under a tight memory size even if k is relatively large. The results imply that the *double-check* mechanism can significantly reduce the ARE of size estimation, as simple fingerprint-based methods (*e.g.*, HeavyGuardian, DHS) suffer from mouse flows' misidentifications. As we analyzed in 6.3, the performance of WavingSketch is greatly influenced by the flow label's length, which can be a bottleneck for it. Hence SwitchSketch outperforms the other algorithms in the process of offline scenarios.

6.5 Experiments on Throughput



(a) M = 100 KB (b) M = 200 KB

Fig. 14. Mops on Throughput Estimation

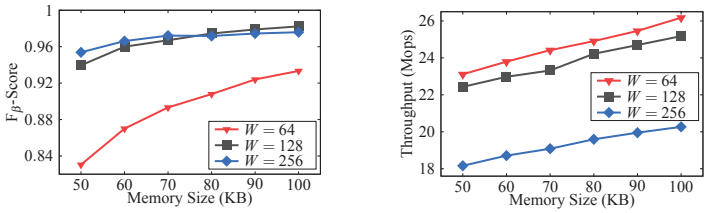
Analysis: In this experiment, we choose the basic version of SwitchSketch (*i.e.*, without double-check mechanism), and set M to 100 KB and 200 KB. As shown in Fig. 14(a)-14(b), WavingSketch achieves the highest throughput because it does not need to calculate the fingerprint for each packet. However, the bottleneck of recording the complete flow label is apparent, as we analyzed in 6.3. SwitchSketch always gets the best performance on throughput compared to the other two fingerprint-based algorithms. When the dataset is CAIDA-2019 and M = 200 KB, SwitchSketch's throughput even gets higher than the WavingSketch's. For HeavyGuardian and WavingSketch, they both need to traverse all cells of a bucket for each insertion. However, SwitchSketch contains several types of cells in each bucket and its *highest type first* principle can make the traversal terminate early in many cases, as we explained in **Query**. Moreover, we embed the metadata in each bucket

to eliminate the extra overhead of memory access, which can be regarded as an optimization of the other algorithms.

6.6 Sensitivity Analysis

6.6.1 *Impact of Parameter Setting in SwitchSketch.*

As shown in 6.1, SwitchSketch has several parameters: the bucket size W, the *double-check* base number θ, the length of *two-mode active counter*'s exponent part γ and the exponential base number b. In this section, we evaluate some sensitivity analysis of SwitchSketch's parameter setting, and we also analyze the impact of different skewness of datasets.



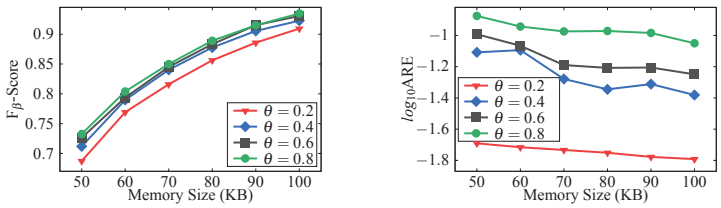
(a) Threshold-t (CAIDA-2016) (b) Throughput (CAIDA-2016)

Fig. 15. Impact of Parameter Setting on W

Impact of W: Here, we set $t = 750$. As shown in Fig. 15(a)-15(b), tuning bucket size is a trade-off between the accuracy of heavy-hitter detection and throughput. We choose the intermediate $W = 128$ bits to achieve both high accuracy (when $W = 128$ and memory size is 100KB, the algorithm's F_β-Score is even higher than that of $W = 256$) and throughput. We also show the occurrences of overflows and switch failures under different bucket sizes in Table 3. We observe that the overflow only happens at the arrival of a few specific packets, and the actual switch functionality to successfully reorganize cell allocations is of lower frequency (triggered once for approximately every 2000 packets). Moreover, the execution frequency of actual switch functionality is only slightly affected by the dataset skewness and bucket sizes.

Table 3. Occurrences of Overflows and Switch Failures (M=100KB)

Datasets	Bucket Size	Overflows	Switch Failures	Packets	Flows
CAIDA-2016	64	12997	0	31612721	576582
	128	21984	7729		
	256	15068	168		
CAIDA-2019	64	11573	0	36144349	370907
	128	19742	7420		
	256	12666	0		

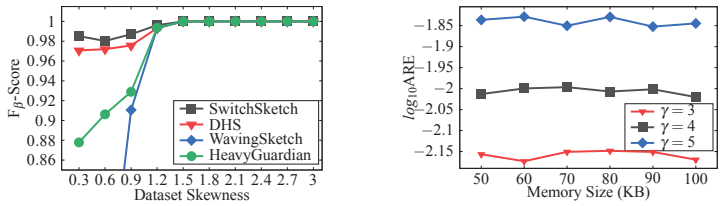


(a) Threshold-t (CAIDA-2016) (b) Threshold-t (CAIDA-2016)

Fig. 16. Impact of Parameter Setting on θ

Impact of θ: Here, we set $t = 750$. As shown in Fig. 16(a)-16(b), when the *double-check* base number θ gets larger, the accuracy of offline heavy-hitter detection increases. However, larger θ leads to a

higher ARE of size estimation. An intermediate value of θ (*e.g.,* 0.6) can contribute to high accuracy and a low ARE.

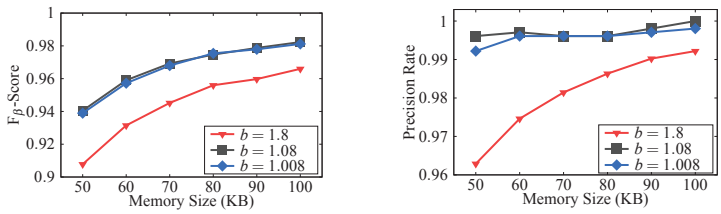


(a) Threshold-t (Synthetic Datasets) (b) Varying γ (CAIDA-2016)

Fig. 17. Impact of Parameter Setting on Dataset Skewness and γ

Impact of Dataset Skewness: We generate ten synthetic datasets that follow the Zipf distribution (skewness from 0.3 to 3.0) by using Web Polygraph [19] and each dataset contains 30M packets. As shown in Fig. 17(a), when the skewness is small, WavingSketch's F_β-Score is low (when the skewness is 0.3, its F_β-Score is only 0.15) and SwitchSketch significantly outperforms the other three algorithms. When the skewness gets large, all four algorithms can get high performance of heavy-hitter detection.

Impact of γ: Here, we set $t = 2^{14}$ because the length of *two-mode active counter*'s exponent part γ only affects the exponential mode's estimation (*i.e.,* 15-bit counter C_L's leftmost bit is '1' whose value is larger than $t = 2^{14}$). We use ARE of offline threshold-t heavy-hitter size estimation as metric and eliminate the estimation error caused by the fingerprint collisions to accurately evaluate γ's effects on size estimation. As shown in Fig. 17(b), smaller γ leads to lower ARE. However, when γ is small, the counter's counting range is insufficient (*e.g.,* When the counter is 15-bit and $\gamma = 3$, according to Lemma. 4.1, the counter's counting range is $[0, 2^{22} - 2^{10}]$. However, the largest flow size in CAIDA-2019 5-minute trace is already close to 2^{22}). To achieve both estimation accuracy and a sufficient counting range, we choose an intermediate $\gamma = 4$.



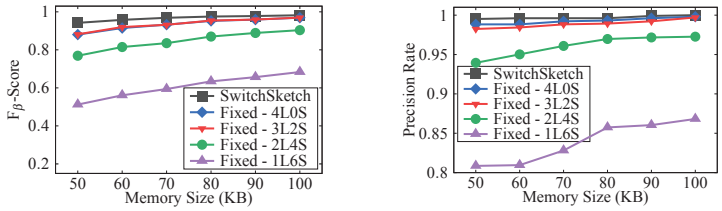
(a) Threshold-t (CAIDA-2016) (b) Top-k (CAIDA-2016)

Fig. 18. Impact of Parameter Setting on b

Impact of b: Here, we set $t = 750$ and $k = 1024$. As shown in Fig. 18(a)-18(b), the intermediate $b = 1.08$ gets the best performance of heavy-hitter detection. The reason is that a larger b (*e.g.,* 1.8) will overly swap out the elephant flows and a smaller b (*e.g.,* 1.008) will make the algorithm insensitive to elephant flows.

6.6.2 *Ablation Study.*

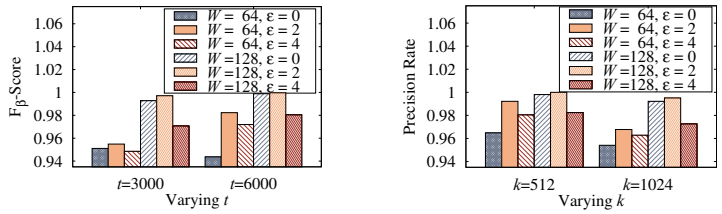
Impact of Dynamic Allocation: Here, we set $t = 750$ and $k = 1024$. We implement the version of SwitchSketch with fixed cell allocation for comparison. Given a 128-bit bucket, we allocate two different sizes of cells, *i.e.,* the 32-bit large cell (L for short) and the 16-bit small cell (S for short). There are four legal allocations, *i.e.,* 4L0S (four large cells), 3L2S (three large cells and two small cells), 2L4S, and 1L6S. As shown in Fig. 19(a)-19(b), SwitchSketch's dynamic allocation outperforms



(a) Threshold-t (CAIDA-2016) (b) Top-k (CAIDA-2016)

Fig. 19. Impact of Dynamic Allocation

the other fixed schemes as it can adaptively accommodate unknown network traffic and imbalanced flow distribution. Moreover, due to the dynamic and unpredictable network traffic, it is impractical for fixed allocation to be well implemented.



(a) Threshold-t (CAIDA-2016) (b) Top-k (CAIDA-2016)

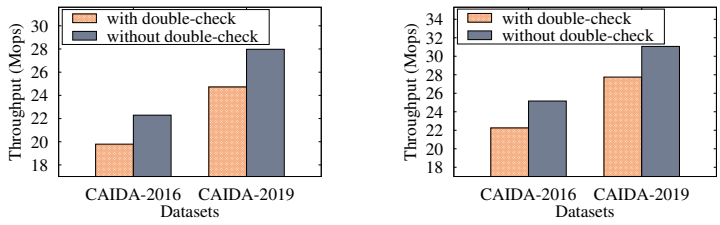
Fig. 20. Impact of Shrinking Cells

Impact of Shrinking Cells: Here, we set M =50KB. We implement the version of SwitchSketch without shrinking cells ($\varepsilon = 0$) and the version of shrinking more bits ($\varepsilon = 4$) for comparison. Given a 128-bit bucket and the types of cells $L = 3$, we fixedly allocate C_1=16 bits, C_2=24 bits, and C_3=32 bits for the former and allocate C_1=12 bits, C_2=20 bits, and C_3=30 bits (calculated by the min-sum optimization in 3.1.3) for the latter. As shown in Fig. 20(a)-20(b), SwitchSketch's scheme of shrinking cells can increase the accuracy of heavy-hitter detection as it can allocate different types of cells in each bucket more granularly. However, when the shrunk bits get large (*e.g.*, $\varepsilon = 4$), the algorithm's accuracy decreases as the updating probability is low. Therefore, we choose to shrink cells by at most $\varepsilon = 2$ bits.

Table 4. ARE on Offline Threshold-t Heavy-Hitter Size Estimation (with/without Double-check)

	Scheme / Dataset	Double-check	without Double-check
M=50KB	CAIDA-2016	**0.10**	17.8
	CAIDA-2019	**0.23**	17.1
M=100KB	CAIDA-2016	**0.06**	8.11
	CAIDA-2019	**0.17**	9.82

Impact of Double-check: Here, we set $t = 750$ and implement SwitchSketch with/without *double-check* for offline threshold-t heavy-hitter detection. As shown in Table 4, *double-check* can significantly reduce the AREs of heavy-hitter size estimation with little impact on the throughput (see Fig. 21(a)-21(b)). The reason is that *double-check* can widely eliminate the estimation error caused by the fingerprint collisions of mice flows. Moreover, *double-check* nearly doubles the flow hashed in each bucket, leading to extra overhead compared to the scheme without *double-check* and making little impact on decreasing the throughput.



(a) M=50KB (b) M=100KB

Fig. 21. Impact of Double-check

7 RELATED WORK

In this section, we introduce two categories of network traffic measurement algorithms used for detecting heavy hitters, *i.e.*, *sketch-based algorithms* and *KV-based algorithms*.

Sketch-based algorithms: The *sketch-based algorithms* refer to some kinds of synopsis data structures to record the sizes of all flows (*e.g.*, Count [4], Count-Min [7], and CM-CU [11]). These traditional *sketch-based algorithms* share a similar data structure with $d \times m$ counters. Count-Min has been widely adopted, and it offers an efficient method to record each flow's size. In the processing of network traffic, Count-Min hashes each incoming packet to d counters and increases all of them by one. In the case of querying, it returns the smallest value among the d hashed counters. The CM-CU is derived from Count-Min and can achieve higher accuracy. The only difference between CM-CU and Count-Min is that CM-CU only increases the smallest counter(s) of the d hashed counters when updating. These algorithms' counters need to be large enough to satisfy a small number of heavy hitters. Due to the highly skewed distributions of network traffic, most counters will only represent a small value, and the higher bits in these counters are all '0', leading to a waste of memory. Some recent works [1, 10, 27] aim to optimize this problem by proposing a dynamic adjusted counter structure. However, when it comes to heavy-hitter detection, a large number of mice flows may introduce significant noises that are hard to be removed, and recording these mice flows seems to be a waste of on-chip memory as well.

KV-based algorithms: The *KV-based algorithms* are typically designed for detecting heavy hitters, *e.g.*, Space-Saving [16], HeavyKeeper [28], WavingSketch [14], HeavyGuardian [25], DHS [29], and ChainSketch [13]. They aim to maintain a data structure to adaptively evict mice flows and preserve heavy hitters' KV-pairs. Space-Saving and WavingSketch keep the entire flow label (*e.g.*, IP address) recorded in their corresponding data structure (*i.e.*, Stream-Summary and Heavy Part). For each flow that is not recorded and arrives when the data structure is full, Space-Saving estimates its corresponding flow's size as a little bit larger than the minimum one in Stream-Summary (*i.e.*, $n_{min} + 1$, where n_{min} is the minimum one's recorded size), and then replaces the minimum one. Recent work WavingSketch proposes an unbiased way to estimate each flow's size. When the flow's estimated size is not smaller than the smallest one in the bucket (each bucket stores several KV-pairs), WavingSketch replaces the flow's key-size pair with the smallest one. As a flow's label may be large (*e.g.*, the 5-tuple header, which is 104 bits), it is a significant memory waste to record all candidate flow labels (flows that are more likely to be heavy hitters) in the limited on-chip memory. Some recent studies [25, 28, 29] use fingerprint for compression, which is actually a hashed flow label with a fixed size (*e.g.*, 16 bits). DHS is one of the most efficient algorithms. It can achieve high accuracy and throughput in heavy-hitter detection because of its adaptive $levelup()$ operations for allocating different sizes of flows. Different from the above methods that directly discard evicted streams, ChainSketch proposed the hash chain operation, which can provide more chances to store the replaced heavy flow in the sketch structure and help to protect heavy flows, making the detection results much more accurate. However, most of the existing *KV-based algorithms* suffer

from low memory efficiency because of the fixed allocation for KV pairs and their complicated bucket structures [14, 25, 29] lead to extra memory accesses. Although DHS proposes an adaptive memory allocation scheme, its cell adjustment is coarse-grained, restricting its efficiency and flexibility.

To address their limitations, this work proposes SwitchSketch, a KV-based sketch framework with a flexible bucket structure and a dynamic switching scheme to accommodate unpredictable and imbalanced network traffic. SwitchSketch's encoding-based switching scheme is fine-grained and can be flexibly adjusted according to different sizes of bucket settings. Moreover, SwitchSketch uses the embedded *metadata* to eliminate the extra memory accesses and proposes the *double-check* mechanism to optimize offline detection.

8 CONCLUSION AND FUTURE WORK

This paper proposes SwitchSketch, a new solution to the problem of heavy-hitter detection with high memory efficiency, flexibility, and accuracy. SwitchSketch can dynamically allocate memory bits to accommodate varied-sized flows by jointly using the flexible bucket structure and the encoding-based switching scheme. Because of SwitchSketch's high memory efficiency and flexibility, it outperforms baselines in different scenarios of heavy-hitter detection.

In the future, we plan to explore the SwitchSketch's feasibility in a broader setting of data stream processing. Specifically, many other stream datasets, *e.g.*, web page [31], database [18], etc., can be used to evaluate SwitchSketch's efficiency under different data stream scenarios. We additionally plan to investigate if SwitchSketch can be deployed on more different platforms [26], enabling more generic heavy-hitter detection in data stream processing.

ACKNOWLEDGMENTS

This work of He Huang and Yu-E Sun is supported by the National Natural Science Foundation of China (NSFC) under Grant 62332013, Grant 62072322, and Grant U20A20182. The work of Yang Du is supported in part by NSFC under Grant 62202322 and in part by the Natural Science Foundation of Jiangsu Province under Grant BK20210706. The work of Jia Liu is supported by NSFC under Grant 62072231. The work of Haipeng Dai is supported by NSFC under Grant 62272223.

REFERENCES

[1] Ran Ben Basat, Gil Einziger, Michael Mitzenmacher, and Shay Vargaftik. 2021. SALSA: Self-adjusting lean streaming analytics. In *Proc. of the International Conference on Data Engineering (ICDE 2021)*. IEEE, 864–875.

[2] CAIDA. 2016. The CAIDA UCSD Anonymized Internet Traces 2016. http://www.caida.org/data/passive/passive_2016_dataset.xml. Accessed: 2019-7-28.

[3] CAIDA. 2019. The CAIDA UCSD Anonymized Internet Traces 2019. https://catalog.caida.org/details/dataset/passive_2019_pcap. Accessed: 2021-12-27.

[4] Moses Charikar, Kevin Chen, and Martin Farach-Colton. 2002. Finding frequent items in data streams. In *Proc. of the International Colloquium on Automata, Languages, and Programming (ICALP 2002)*. Springer, 693–703.

[5] Cloudflare. 2023. Cloudflare General Recommendations for Sampling Rates. https://developers.cloudflare.com/magic-network-monitoring/routers/recommended-sampling-rate/.

[6] Source Code. 2023. The source codes of SwitchSketch and other related algorithms. https://github.com/duyang92/switch-sketch-paper/.

[7] Graham Cormode and Shan Muthukrishnan. 2005. An improved data stream summary: The count-min sketch and its applications. *Journal of Algorithms* 55, 1 (2005), 58–75.

[8] Zhenwei Dai, Aditya Desai, Reinhard Heckel, and Anshumali Shrivastava. 2021. Active sampling count sketch (ascs) for online sparse estimation of a trillion scale covariance matrix. In *Proc. of the ACM International Conference on Management of Data (SIGMOD 2021)*. 352–364.

[9] Y. Du, H. Huang, Y. Sun, S. Chen, and G. Gao. 2021. Self-Adaptive Sampling for Network Traffic Measurement. In *Proc. of the IEEE Conference on Computer Communications (INFOCOM 2021)*. 1–10.

[10] Junzhi Gong, Tong Yang, Yang Zhou, Dongsheng Yang, Shigang Chen, Bin Cui, and Xiaoming Li. 2017. Abc: A practicable sketch framework for non-uniform multisets. In *Proc. of the International Conference on Big Data (Big Data 2017)*. IEEE, 2380–2389.
[11] Amit Goyal, Hal Daumé III, and Graham Cormode. 2012. Sketch algorithms for estimating point queries in NLP. In *Proc. of the Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP 2012)*. 1093–1103.
[12] He Huang, Yu-E Sun, Chaoyi Ma, Shigang Chen, Yang Du, Haibo Wang, and Qingjun Xiao. 2021. Spread Estimation With Non-Duplicate Sampling in High-Speed Networks. *IEEE/ACM Transactions on Networking (TON)* 29, 5 (2021), 2073–2086.
[13] Jiawei Huang, Wenlu Zhang, Yijun Li, Lin Li, Zhaoyi Li, Jin Ye, and Jianxin Wang. 2022. ChainSketch: An efficient and accurate sketch for heavy flow detection. *IEEE/ACM Transactions on Networking (TON)* (2022), 1–16 (Early Access).
[14] Jizhou Li, Zikun Li, Yifei Xu, Shiqi Jiang, Tong Yang, Bin Cui, Yafei Dai, and Gong Zhang. 2020. WavingSketch: An unbiased and generic sketch for finding top-k items in data streams. In *Proc. of the ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD 2020)*. 1574–1584.
[15] Hongyan Liu, Yuan Lin, and Jiawei Han. 2011. Methods for mining frequent items in data streams: an overview. *Knowledge and Information Systems* 26, 1 (2011), 1–30.
[16] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. 2005. Efficient computation of frequent and top-k elements in data streams. In *Proc. of the International Conference on Database Theory (ICDT 2005)*. 398–412.
[17] Shanmugavelayutham Muthukrishnan. 2005. *Data streams: Algorithms and applications*.
[18] Laurel Orr, Magdalena Balazinska, and Dan Suciu. 2020. Sample debiasing in the themis open world database system. In *Proc. of the ACM International Conference on Management of Data (SIGMOD 2020)*. 257–268.
[19] Alex Rousskov and Duane Wessels. 2004. High-performance benchmarking with Web Polygraph. *Software: Practice and Experience* 34, 2 (2004), 187–211.
[20] Nadi Sarrar, Steve Uhlig, Anja Feldmann, Rob Sherwood, and Xin Huang. 2012. Leveraging Zipf's law for traffic offloading. *ACM SIGCOMM Computer Communication Review* 42, 1 (2012), 16–22.
[21] Y. Sun, H. Huang, C. Ma, S. Chen, Y. Du, and Q. Xiao. 2020. Online Spread Estimation with Non-duplicate Sampling. In *Proc. of the IEEE Conference on Computer Communications (INFOCOM 2020)*. 2440–2448.
[22] You-Chiun Wang and Siang-Yu You. 2018. An efficient route management framework for load balance and overhead reduction in SDN-based data center networks. *IEEE Transactions on Network and Service Management (TNSM)* 15, 4 (2018), 1422–1434.
[23] Mengkun Wu, He Huang, Yu-E Sun, Yang Du, Shigang Chen, and Guoju Gao. 2021. ActiveKeeper: An accurate and efficient algorithm for finding top-k elephant flows. *IEEE Communications Letters* 25, 8 (2021), 2545 – 2549.
[24] Qingjun Xiao, Zhiying Tang, and Shigang Chen. 2020. Universal online sketch for tracking heavy hitters and estimating moments of data streams. In *Proc. of the IEEE Conference on Computer Communications (INFOCOM 2020)*. 974–983.
[25] Tong Yang, Junzhi Gong, Haowei Zhang, Lei Zou, Lei Shi, and Xiaoming Li. 2018. HeavyGuardian: Separate and guard hot items in data streams. In *Proc. of the ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD 2018)*. 2584–2593.
[26] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. 2018. Elastic sketch: Adaptive and fast network-wide measurements. In *Proc. of the Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM 2018)*. 561–575.
[27] Tong Yang, Jiaqi Xu, Xilai Liu, Peng Liu, Lun Wang, Jun Bi, and Xiaoming Li. 2019. A generic technique for sketches to adapt to different counting ranges. In *Proc. of the IEEE Conference on Computer Communications (INFOCOM 2019)*. 2017–2025.
[28] Tong Yang, Haowei Zhang, Jinyang Li, Junzhi Gong, Steve Uhlig, Shigang Chen, and Xiaoming Li. 2019. HeavyKeeper: An accurate algorithm for finding top-k elephant flows. *IEEE/ACM Transactions on Networking (TON)* 27, 5 (2019), 1845–1858.
[29] Bohan Zhao, Xiang Li, Boyu Tian, Zhiyu Mei, and Wenfei Wu. 2021. DHS: Adaptive memory layout organization of sketch slots for fast and accurate data stream processing. In *Proc. of the ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD 2021)*. 2285–2293.
[30] Hao Zheng, Chen Tian, Tong Yang, Huiping Lin, Chang Liu, Zhaochen Zhang, Wanchun Dou, and Guihai Chen. 2022. FlyMon: Enabling on-the-fly task reconfiguration for network measurement. In *Proc. of the Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM 2022)*. 486–502.
[31] Yang Zhou, Tong Yang, Jie Jiang, Bin Cui, Minlan Yu, Xiaoming Li, and Steve Uhlig. 2018. Cold filter: A meta-framework for faster and more accurate stream processing. In *Proc. of the ACM International Conference on Management of Data (SIGMOD 2018)*. 741–756.

Received January 2023; revised April 2023; accepted May 2023